Received: 16 September 2016 | Revised: 9 February 2017 | Accepted: 10 April 2017
DOI: 10.1002/spe.2503

RESEARCH ARTICLE

WILEY

Assessing the impact of Volatile Functionality removal in web applications: Model-Driven vs Code-Based approaches

M. Urbieta[1,2] | D. Frajberg[1,3] | G. Rossi[1,2]

[1]LIFIA, Facultad de Informática, UNLP, La Plata, Argentina
[2]CONICET, La Plata, Argentina
[3]Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milan, Italy

Correspondence
M. Urbieta, LIFIA, Facultad de Informática, UNLP, Argentina.
Email: murbieta@lifia.info.unlp.edu.ar

Summary
Web applications must quickly adapt to new business demands to keep clients onboard. When unexpected and unforeseen requirements appear, the changes pose challenges to software engineers as they were not considered in the application design and such new functionality can be only valid only for a period in certain situations.
This work presents a comparison of maintenance consequences in the software quality when using a Model-Driven approach against a Code-Based one where changes are managed in an ad hoc way. We used the removal of volatile functionality as case of study due to the fact that its characteristics stress the development process. We considered both external and internal quality of deliverables.
For assessing the quality of deliverables, we evaluated the production of more than 35 students using both approaches considering their perception gathered through questionnaires and their deliverables.
We present the preliminary evidence that there is no significant difference between approaches. Model-Driven performed slightly better than Code-Based, but both responded negatively in terms of deliverables' quality. Results show that maintenance tasks are detrimental to software quality where Model-Driven approach does not excel over Code-Based approach. The Model-Driven approach only highlighted on productivity.

KEYWORDS
maintenance, Model-Driven web application, quality, requirement, Volatile Functionality

1 | INTRODUCTION

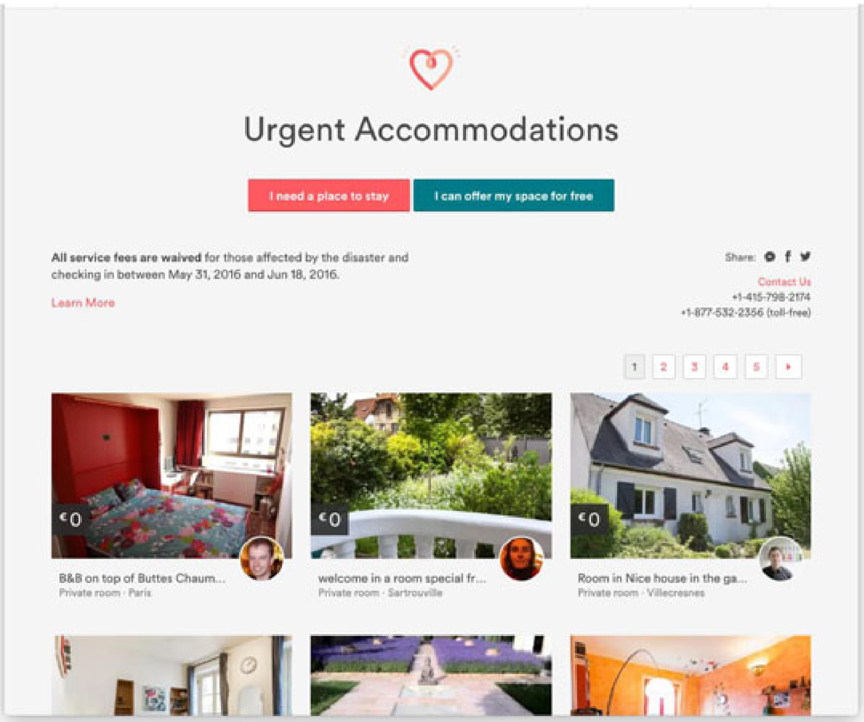
Nowadays business must adapt to global trends in order to keep users engaged; unplanned marketing campaigns, season promotions (final season sales), crisis management, among other business requirements are examples of unexpected requisites that stress the whole applications' infrastructure.

Volatile Functionalities[1] are unexpected and unforeseen features developed once that they are enabled within the application by business or time event and deactivated in the same way. They are common in most popular web applications; sometimes these are new features, which are experimented for a period of time and then discarded because the user does not find them useful (aka, Beta). Some other times, they are triggered punctually in response to a specific event or set of conditions. More often, they are periodically activated and deactivated in coincidence with specific periods of the year. The need for these functionalities arises after the application has been implemented and deployed for the first time and, as a consequence, they are not taken into consideration during the application design phase. Including new web application features for a period of time requires not only coding efforts but managing and designing. Business context makes matters worse proposing an unattainable release date that has been reported as a source of a Technical Debt leak.[2] Technical Debt[2] metaphor is used to explain the impact of these changes in the application source code, ie, (according to the metaphor), which additional costs we must pay for not being able to anticipate these changes during requirement analysis and development.

In an e-commerce website, such as Amazon.com, typical examples of such functionalities are (i) the special offers available at certain periods of the year (eg, Christmas, St Valentine's, etc.) on specific products; (ii) the customization of contents for new releases (such as videos from Related Media performances); and (iii) the functionality for fundraising after a catastrophe, and many others. Similar examples can be found in news sites, such as CNN.com, to organize discussions on unexpected events, to include new advertisement types (eg, during presidential elections), etc.

As an example, we show in Figure 1.A an AirBnB campaign to link housing providers with Parisian families that were left homeless by the last flooding in June 2016. Said campaign allowed affected citizens from Paris to access accommodation for free during a certain period of time as shown in Figure 1.B. Clearly, the flooding is an unpredictable catastrophe, which no one is able to anticipate and therefore it has as consequence software requirements with the same characteristics. Moreover, while the AirBnb campaign was enabled for Paris, the same query in Rome area resulted in non-free accommodation showing that the intent was only restricted to some business objects. The set of changes comprised a registration form for displaced people, free accommodation index and business rules to apply discounts to homeless people.

We will study the internal quality impact of maintaining Model-Driven web applications and on how a Model-Driven approach performs against a Code-Based one while software is maintained. As case study, we used Volatile Functionalities because their singular characteristics stress maintenance tasks. Their unexpected arrival may constrain engineers because their design must be based on a running application. On the other hand, the removal of Volatile Functionalities is error prone. Code artifacts are overlooked or left behind, being source of issues with a diverse severity in the technical debt analysis. Changes performed by developers at presentation layer do not consider behind layers' artifacts such as controllers or business objects, leaving orphan methods, classes and even databases. The emergence of this kind of issues after several maintenance cycles may make the internal application's quality unsustainable. Additionally, this situation demands much effort on testing the application after introducing the new volatile functionalities and assessing that the application works as it did originally after removing such functionalities.


Urgent Accommodations

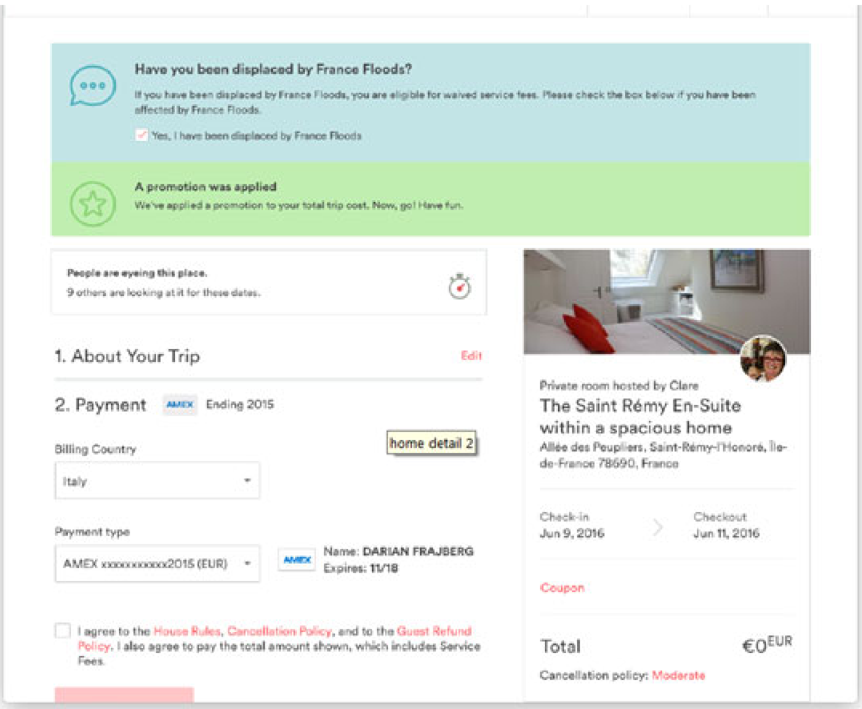
All service fees are waived for those affected by the disaster and checking in between May 31, 2016 and Jun 18, 2016.

Learn More

Contact Us
+1-415-798-2174
+1-877-532-2356 (toll-free)

I need a place to stay
I can offer my space for free
Share:
1 2 3 4 5
€0
B&B on top of Buttes Chaum...
Private room · Paris
€0
welcome in a room special fr...
Private room · Sartrouville
€0
Room in Nice house in the ga...
Private room · Villecresnes

FIGURE 1.A: Index of urgent accommodation at AirBnB during Paris flooding [Colour figure can be viewed at wileyonlinelibrary.com]


Have you been displaced by France Floods?
If you have been displaced by France Floods, you are eligible for waived service fees. Please check the box below if you have been affected by France Floods.
[✓] Yes, I have been displaced by France Floods

A promotion was applied
We've applied a promotion to your total trip cost. Now, go! Have fun.

People are eyeing this place.
9 others are looking at it for these dates.

1. About Your Trip — Edit

2. Payment AMEX Ending 2015

home detail 2

Billing Country: Italy

Payment type: AMEX xxxxxxxxxxx2015 (EUR)

Name: DARIAN FRAJBERG
Expires: 11/18

[ ] I agree to the House Rules, Cancellation Policy, and to the Guest Refund Policy. I also agree to pay the total amount shown, which includes Service Fees.

Private room hosted by Clare
The Saint Rémy En-Suite within a spacious home
Allée des Peupliers, Saint-Rémy-l'Honoré, Île-de-France 78690, France

Check-in: Jun 9, 2016
Checkout: Jun 11, 2016

Coupon

Total €0 EUR

Cancellation policy: Moderate

FIGURE 1.B Discount rule for displaced people that allows booking accommodation for free

We aim at analyzing the set of changes introduced by this kind of functionalities in Model-Driven web applications during the maintenance phase and also at measuring how software quality is affected when they are removed in ad-hoc way by applying web software quality metrics. We claim volatile functionalities' removal maintenance tasks are error prone and therefore a source of issues that increase the Technical Debt.

In this paper, we will present an empirical study of how both Model-Driven and Code-Based web applications' quality is affected by changes and how their internal qualities are compromised while being maintained. We will use an application designed with IFML and implemented with WebRatio[3] and an open source Groovy-Based Web application; both of which were modified with a set of volatile functionalities to evaluate different source-code metrics. As many web application implementations implement the Model-View-Controller (MVC) pattern, we will use different sets of metrics. For example, in the View layer, we will evaluate the presence of broken links, whereas in the Model layer, we will evaluate the lack of cohesion. We use these metric as Groovy application is based on an Object-Oriented Paradigm. Different empirical research works[4-6] compare Code-Based and Model-Driven approaches for either developing applications from scratch or maintaining applications. However, they focus on developers' perception of the experience. This work proposes a novel contribution the studying internal quality evolution when performing a specific kind of maintenance tasks.

Students from the National University of La Plata, Argentina, and from Johannes Kepler Universität of Linz, Austria, performed Volatile Functionality maintenance of a Code-Based and Model-Driven web Application. This experiment was executed in order to analyze the impact on the applications' quality by using automatic and manual assessment methods, as well as their perception of such tasks by using questionnaires.

The rest of the paper is structured as follows, using guidelines for empirical assessments[7]: in Section 2, we discuss some related work; in Section 3, we present some background related to Volatile Functionality; in Section 4, we introduce the research method used for evaluating software quality as well as the metrics used; and in Section 5, we summarize the results obtained from the empirical analysis; in Section 5.7, we discuss some experiment's threats to validity; finally in Section 6, we conclude by presenting the gathered results, and we provide some further work we are pursuing.

2 | RELATED WORK

Systems evolution has been largely studied in order to avoid common mistakes, predict costs of maintenance and reduce changes impact. In a previous study,[8] a study of variability is presented as the ability to change software artifacts for specific contexts. Small and Medium Enterprises were surveyed, with a resulting outcome of an enumeration of challenges such as documentation, change traceability, change extent, among others, when facing changes in a system. Unfortunately, change impact analysis was not deepened in products' source code and quality.

The traceability of changes in the lifecycle of evolution/maintenance requirements has been combined with Impact Analysis (IA) in a previous study.[9] It provides an integrated approach that gives traceability support from the change request, going through model's artifacts, source code changes, and execution behavior. This approach can be used for tracing any Volatile Requirement, but its removal is still compromised and should be studied.

Model-Driven Engineering aims at providing methods and tools for designing web Applications, where developers abstract from source code aspects and focus on functional requirements instead. In a previous study,[4] authors studied how maintenance tasks, in both OOHRIA approach and code centric approach, were perceived by practitioners. The experiment was performed by 26 students, and their perception was captured through questionnaires. The results showed that OOH4RIA improved efficiency and effectiveness of maintenance tasks over the same tasks when using .NET technology. The same authors also performed this study by using an application development from scratch as case study.[5] The empirical study result highlighted that developers find Model-Driven Development (MDD) more useful than Code-Based approaches even through its learning curve, and the need to replicate the evaluation in other commercial and academic context. Another study regarding MDD performance against Code-Based was performed by Panach et al,[6] and it focused on verifying some of the most cited benefits of MDD. By means of empirical evaluations with students, they analyzed quality, effort, productivity, and satisfaction aspects on MDD and Code-Based projects. Authors concluded that MDD does not always yield better results than a traditional method but they also highlighted that MDD provides better accuracy when developing functional requirements.

However, the abovementioned evaluations were focused on subjective metrics gathered from students' perception, and it did not cover source code's metrics in order to assess internal quality such as object-oriented metrics[10] for code centric solution and model metrics. Additionally, most of the empirical research works focused on the development from a scratch study case, while we study maintenance challenges.

Regarding the use of Unified Modeling Language (UML) diagrams, in a previous study,[11] authors conducted a controlled experiment where they studied the relationship between the maintenance of system's UML documents and the functional correctness of the changes. UML usage had a significantly positive impact.

Technical debt has been studied largely in industry projects[12] and how to manage it,[13] as it highly relates with application's budget. In this field, we can find plenty of tools that allow measuring technical debt from source code,[14-17] but also taking into account other points of view such as architecture-sourced debt[18] on components coupling[19] or database schemas technical debt.[20] Technical debt must be analyzed using different approaches in order to gain as much information as possible. In a previous study,[21] Hadoop an open source No-sql database was processed using code smells, automatic static analysis issues, grime buildup, and modularity violations, taking into account several application versions. The outcome remarks that techniques are loosely coupled, and therefore a Technical debt analysis should combine several techniques.

In a previous study,[12] a large set of 745 business applications from 160 companies in 10 different industry segments were analyzed using Appmarq tool.[14] They proposed a formula to calculate applications' technical debt. Additionally, they provided a discussion about specific characteristics of technical debt depending on the application's field. The paper was focused on code analyst meanwhile in a preevious study,[12] different architects were surveyed in order to understand sources and decision making associated to technical debt.

Feature Oriented Programming has been an approach for facing those developments that compose functionality mainly focused on Software Families. In a previous study,[22] authors introduce the relevant factors in variability realization of Software Product Families. As part of the work, they proposed taxonomy of variability introducing different realization techniques for software changes. The presented techniques were mapped to surveyed and/or studied software companies, which adapt their software to changes.

Maintenance effort was studied in a previous study,[23] where 10 PhD and Master students were asked to maintain a desktop application. The work studied deeply the processes and tools used by students for solving the problem. The outcome was a characterization of problems, relevant code and APIs, and testing processes.

In a previous study,[24] the authors have presented the idea of defining a domain-specific language for web applications, with the goal of detecting inconsistencies (not caught by web frameworks) during compilation-time. They state that modern web application development frameworks provide web application developers with high-level abstractions to improve their productivity. However, their support for static verification of applications is limited. Inconsistencies in an application are often not detected statically, but appear as errors at run-time. Therefore, these researchers have proposed this alternative in order to ease the developers' work, providing them with accurate and clear error messages.

Because their proposed Domain-Specific Language (DSL) considers some of the syntactic inconsistencies as errors, just as we do, it could be useful for diminishing the rate of syntactic errors while maintaining web applications. Nevertheless, it differs from our research, as its main objective is not focused on the maintenance phase and even less on the removal of volatile functionalities. Consequently, this approach does not cover the entire set of syntactic inconsistencies that we have detected, and what is more, it does not consider any kind of semantic errors.

3 | VOLATILE FUNCTIONALITY BACKGROUND

The addition of volatile functionalities in web applications must not be considered as a particular case of web software evolution because of its singular unique characteristics. A Volatile Functionality can be characterized at least along 3 dimensions that we defined as Intent, Extent, and Volatility Pattern.

The *Intent* of a Volatile Functionality can be identified by answering the following questions: which aspects of the web application does the Volatile Functionality impact? For each aspect, which application object types (or classes) have to be added or modified and how?

By taking as reference the design dimensions identified for a web application by most web engineering methods,[25] aspects that the *Intent* of a Volatile Functionality may span include: content, navigation, user interface (or presentation), and behavior. Correspondingly, application objects types include, among others: content types, navigation nodes and paths, interaction widgets, business process workflows, business process activities, and business rules. Names adopted for the mentioned aspects (aka, concerns or layers) and object types (aka, modeling concepts) may vary depending on the considered web engineering method.

The *Extent* of a Volatile Functionality identifies the set of application objects (instances of the types identified by the Intent) that it impacts. The Extent of a Volatile Functionality is indeed determined by answering the following questions: which application objects have to be added or modified for the types determined by the Intent? Is the Volatile Functionality complete? (ie, does it affect all instances of the types?) or does it apply to some specific instances? For each of the application objects types in the Intent, a way to determine the involved instances has to be specified.

The *Volatility Pattern* deals with describing the life-cycle of the Volatile Functionality, ie, the rules governing its activation/deactivation during the time. In fact, this dimension distinguishes a Volatile Functionality from an evolution requirement. Although both an evolution requirement and a Volatile Functionality can be unforeseen at the beginning of the application development, a Volatile Functionality has a well-defined Volatility Pattern which describes its life-cycle. This dimension of the characterization can be defined by answering the following questions: when does the Volatile Functionality have to be activated? (eg, after a specific event has occurred; on a fixed date; during a specific day of the week; at a specific time of the day; when certain business rules are satisfied, etc.). How long does it have to remain active? (eg, for a certain number of days, hours, minutes) or when does it have to be deactivated? (eg, when a certain time has passed from the activation; when a certain business rule is satisfied; when a certain day or time comes, etc.).

4 | EXPERIMENT PLANNING

In this research, we aim at evaluating how Model-Driven Development approaches for designing web applications and Code-Based application perform when maintaining Volatile Functionalities. In this case, we will use an IFML-Based web application, but the same experiment could be applied over other approaches[25] (UWE, OOHDM, OOHRIA, etc.). On the other hand, we will conduct the same evaluation in a Code-Based application that does not rely on any Model-Driven approach. In both cases, maintenance tasks will be managed in an ad-hoc way without the support of any development methodology such as agile approaches (ie, Scrum), unified approaches (ie, RUP), or advance Model-Driven Web Engineering (MDWE) approaches in order to keep the experiment defined by one independent variable with the intention of the complexity is under control. For this last, we left as further work the study of whether or not the development methodology is a performance constraint. Consequently, several existing MDWE are excluded from this work that can be used to tackle Volatile Functionalities[1] or support evolutive requirements such as WebComposition Process Model,[26] Distributed Concern Delivery[27] or, more general principles, such as refactoring,[28] and patterns.[29]

In order to study how different components of both a IFML web application and a Code-Based one are affected by the introduction and later removal of volatile functionalities, we conducted an experiment with a set of developers. We have focused on the removal of functionalities and not on the introduction due to the fact that, as it was mentioned in the related work, consequences of adding functionalities during maintenance have already been studied.[4-6] In these works, authors remark an improved efficiency and effectiveness of evolutive maintenance tasks as well as better learning curve than programming languages. Therefore, by considering this background, we will skip the study of the new Volatile Functionality introduction, and instead, we will consider maintenance tasks involving Volatile Functionality removal, which was an unattended topic. Both this and the following sections will describe the experiment and its results by using guidelines suggested in some previous studies.[7,30,31] During the experiment, we will evaluate a set of representative quality metrics that covers business model and controller layer, usually coded using programming languages such as Java, Python, C#, etc., and interface layer implemented with HTML, JavaScript, and CSS. As Code-Based and Model-Driven applications belong to different paradigms, we performed 2 independent evaluations based on each kind, and we defined rules that aim at measuring internal quality fairly.

By the introduction and later removal of such Volatile Functionality as introduced in Section 3, web applications pass through different development stages. These 2 applications were modified twice and, a snapshot was captured for each version, which is used next for evaluating its quality. We will call Original Version (OV) the starting stable version that resolves core business requirements. Next, we will have a version that introduces a set of volatile functionalities extending the application, which will be outlined later, called Volatile Version (VV); and, finally, we have a version that will be the result of removing such Volatile Functionalities, named Maintenance Version (MV), that presents the set of requirements served by OV. At first sight, MV should be alike OV, and we will study any difference found focusing on those that are detrimental to the overall quality. The transition of web Applications' versions is shown in Figure 2.

We have selected 2 applications as running examples that have been modified to introduce volatile functionalities: a Code-Based application that allows managing a Petshop (from now on Petshop) and a Model-Driven application for selling furniture (from now on ACME). For these applications, we have conceived volatile functionalities like those already discussed in a previous study.[1] The applications and their volatile functionalities will have a thorough description later on.

The evaluation is performed by a 2-fold process in order to determine both internal and external web application quality. Firstly, developers and Quality Assurance analysts are surveyed in order to assess external software quality and development process quality. Development experience can be sloppy, error prone and excessively time consuming; thus, we aim at capturing Volatile Functionality maintenance challenges from performers' point of view.

Secondly, we perform a layer-focused semi-automatic analysis of source code in order to assess internal quality. Each variance in between OV and MV will be analyzed and discussed.

4.1 | Evaluation and research questions

Application development involves facets that cannot be measured automatically by tools that analyze source code; for these aspects, we used questionnaires as information gathering tools (see appendixes A and B for more information).

We have designed 4 artifacts to gather information from the experiment:

1. Final Developers' Questionnaire, which has the goal of collecting features, development experience, and development process experience;
2. Assignment, it describes application context and required maintenance task abstractly.

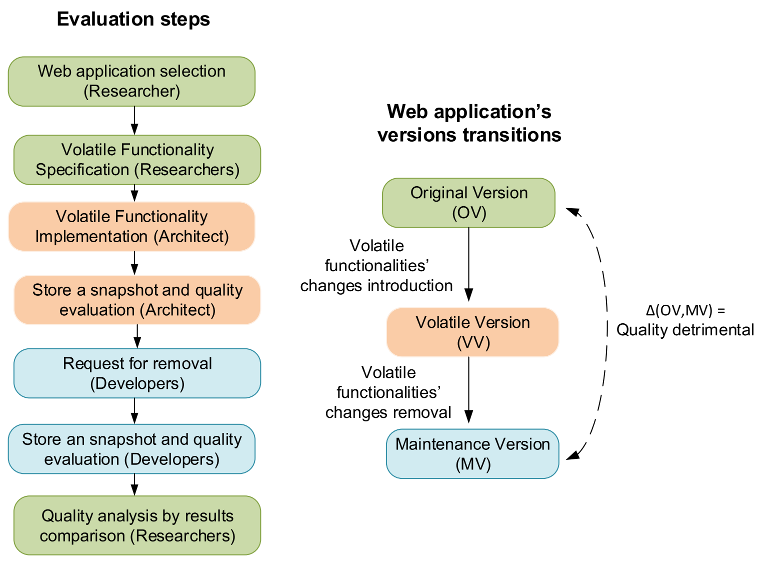

Evaluation steps
Web application selection (Researcher)
Volatile Functionality Specification (Researchers)
Volatile Functionality Implementation (Architect)
Store a snapshot and quality evaluation (Architect)
Request for removal (Developers)
Store an snapshot and quality evaluation (Developers)
Quality analysis by results comparison (Researchers)
Web application's versions transitions
Original Version (OV)
Volatile functionalities' changes introduction
Volatile Version (VV)
Volatile functionalities' changes removal
Maintenance Version (MV)
Δ(OV,MV) = Quality detrimental

FIGURE 2 Application's evolution scheme [Colour figure can be viewed at wileyonlinelibrary.com]

3. Technical documents, supporting documents were provided to describe how to import, set up and run the application.
4. Reviewer's Quality Report, which captures web application internal and external quality issues raised by comparing OV against MV.

Documents (i), (ii), and (iii) were provided to developers during the experiment while (iv) was filled out by the architect in charge of reviewing the application's quality.

In order to evaluate Model-Driven development performance against traditional Code-Based approach, we have defined research questions to be answered during the following sections. For each research question, null and alternative hypothesis were defined, as well as the metrics that will be used to test them. Because of the impedance mismatch, we have defined diverse rules that aim at measuring the presence of an issue in applications artifacts. Depending on whether a given metric evaluates a discrete or Nominal value, T-Student and Fisher-Irwin hypothesis testing[32] methods are used respectively. In Table 1, we present a summary of Research Questions, considered metrics, Null Hypothesis, and statistical method used for its analysis. In following sections, we will discuss how each research question was answered by using a metric and statistical method.

Values obtained during the assessment process can be found in Appendix B.

4.2 | Case studies

We selected 2 small web applications implemented with different approaches, a Model-Driven approach and a full-stack agile framework. In order to stress the software development process, we selected and implemented a few Volatile Functionalities that introduce changes into different application layers satisfying Volatile Functionality's characteristics introduced in Section 3.

Below, we introduce each application and its Volatile Functionalities.

4.2.1 | ACME application

Acme is an e-commerce web application that shows the product catalog of the Acme brand, which is dedicated to the sale of furniture. Registered users are allowed to search and to add products to their carts. There are also administrator users, who have

TABLE 1 Research questions and considered metrics for hypothesis testing

	Research question	Metrics	Source	H_0
1	Is Model-Driven Development more productive than Code-Based?	Spent time	FDQ	$TS_{MDD} = TS_{CB}$ (T-student)
2	Is Code-Based approach more prone to have application external quality issues?	Pending User Interface layer changes	RQR	$PUILC_{MDD} = PUILC_{CB}$ (Fisher-Irwin)
3	Is application internal quality more negatively affected by Code-Based approach than Model-Driven Development?	Unreferenced domain model variables and classes	RQR	$UDMVC_{MDD} = UDMVC_{CB}$ (Fisher-Irwin)
		Non-executable behavior	RQR	$NEB_{MDD} = NEB_{CB}$ (Fisher-Irwin)
		Improper behavior	RQR	$IB_{MDD} = IB_{CB}$ (Fisher-Irwin)
		Unreferenced User Interface resources	RQR	$UUIR_{MDD} = UUIR_{CB}$ (Fisher-Irwin)
		Unused User Interface artifacts	RQR	$UUIA_{MDD} = UUIA_{CB}$ (Fisher-Irwin)
4	Is developer confidence affected by Model-Driven Development?	Application stability perception	FDQ	$ASP_{MDD} = ASP_{CB}$ (Fisher-Irwin)
		Testing requirement	FDQ	$TR_{MDD} = TR_{CB}$ (Fisher-Irwin)
5	Is application data quality compromised by Model-Driven Development?	Complete database information loss	RQR	$CDIL_{MDD} = CDIL_{CB}$ (Fisher-Irwin)
		Volatile Functionality's data loss	RQR	$VFDL_{MDD} = VFDL_{CB}$ (Fisher-Irwin)
		Useless data generation	RQR	$UDG_{MDD} = UDG_{CB}$ (Fisher-Irwin)

FDQ, Final Developers' Questionnaire; RQR, Reviewer's Quality Report.

access to the backend, from where they can manage the entire product catalogue and other relevant information. This application was developed using the WebRatio platform and uses a MySQL database. It is a sample provided by the WebRatio Community.

For this application, we have chosen the following 2 volatile functionalities: Mother's day promotion and Product comments.

In Figure 3A, we show the changes introduced into the conceptual layer by both Volatile functionalities. These modifications were designed using a UML class diagram and include the new attributes and entities related to them. Modifications concerning the navigational layer can be seen in Figures 3 B and C with representations extracted from the Webratio's platform project. The first one presents the navigational model for the Product comments feature, whereas the second one presents the navigational model for Mother's day banner, product index and Term of Condition features.

Next, we present a brief description and source code's impact of each Volatile Functionality:

- Mother's day promotion

  Description: The ACME site has decided to put a special promotion on furniture online in order to increase the sales on the days close to Mother's day. Therefore, some of the offered products will be advertised and sold including important discounts before such special event.

  Intent and Extent: An attribute (discount) is introduced into the Product class with the purpose of defining different and independent discounts for certain products. Additionally, the functionality requires considering the product's discount when computing the price of the purchase; this logic is implemented as a derivated attribute called *finalprice*. This requirement impacts on the application with a wide set of changes at the navigation and presentation layers. First, it introduces a promotion banner (with an associated new style) for highlighting the promotion existence. This banner contains a link to a new page, which lists all the products “in promotion”. It adds another new page with the terms of condition of the promotion as a landmark, too. This requirement also adds 2 new data elements to display the discount and final price into every place where the products' information is presented. Moreover, the *price* attribute references were replaced by the *finalprice* so as to consider the value of the products´ prices with their corresponding discounts. Finally, an initialize

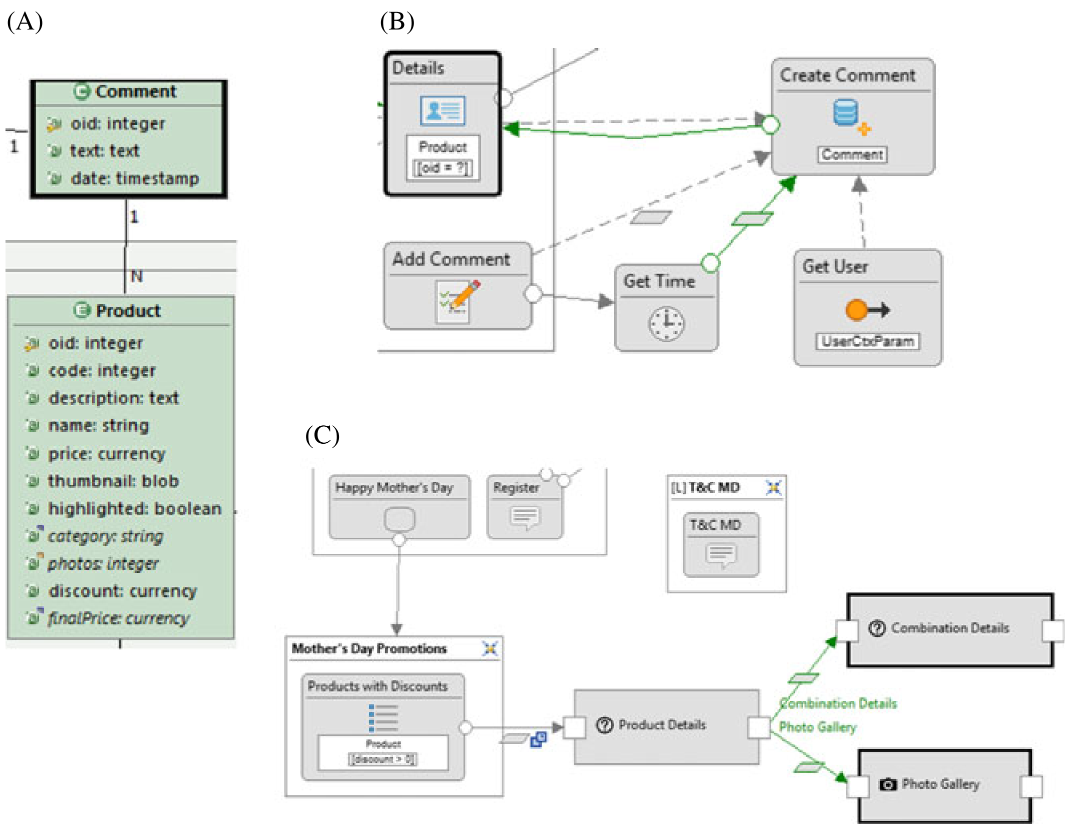


FIGURE 3 A, Attributes and entities related to the Volatile functionalities. B, The navigation model for Comment feature. C, The navigation model for Mother's day banner, product index and Term of Condition features [Colour figure can be viewed at wileyonlinelibrary.com]

Job is needed in order to set the not promoted products' discounts from null to 0.
Volatile Pattern: The set of changes must be available from October 1–20.

- Product comments

  Description: The ACME site has decided to take advantage of the site's high-access rate period to include beta functionality in order to assess its adoption. The beta functionality to be tested allows customer leaving a comment on published products. Only logged-in customers can leave such comments.
  Intent and Extent: A class (*Comment*) is added to the model with the aim of registering comments associated to the offered products. Moreover, a small beta-tested feature is introduced for such task. When accessing the details of a specific product, customers are allowed to read and to leave comments about it. The logic needed to extract the logged user's account name and timestamp was introduced as well. For the sake of simplicity, we have only depicted the most important changes.
  Volatile Pattern: The set of changes must be available from October 1–20.

4.2.2 | Petshop application

Petshop is a web application that allows registering pets and their owners, recording their visits to the Petshop and access stored information. This tool was developed by using Grails framework, which combines a full-stack of dependency injection framework (Spring Framework), an object-relational mapping framework (Hibernate framework) and a development process that, by means of convention over configuration, allows an agile web application development. The application was downloaded from the Grails site, which is provided by Grails community.

For this application, we have thought about the following 2 volatile functionalities: Castration Promotion and Holiday Pet Care.

In Figure 4, simplified and reduced application models (conceptual, navigational, and user) are shown, where application changes are pointed out through each model, as well as their volatile pattern. Models at conceptual and navigational layer were designed using UML class diagram, and Abstract Data Views (ADV) was used for modeling user interfaces. The orange

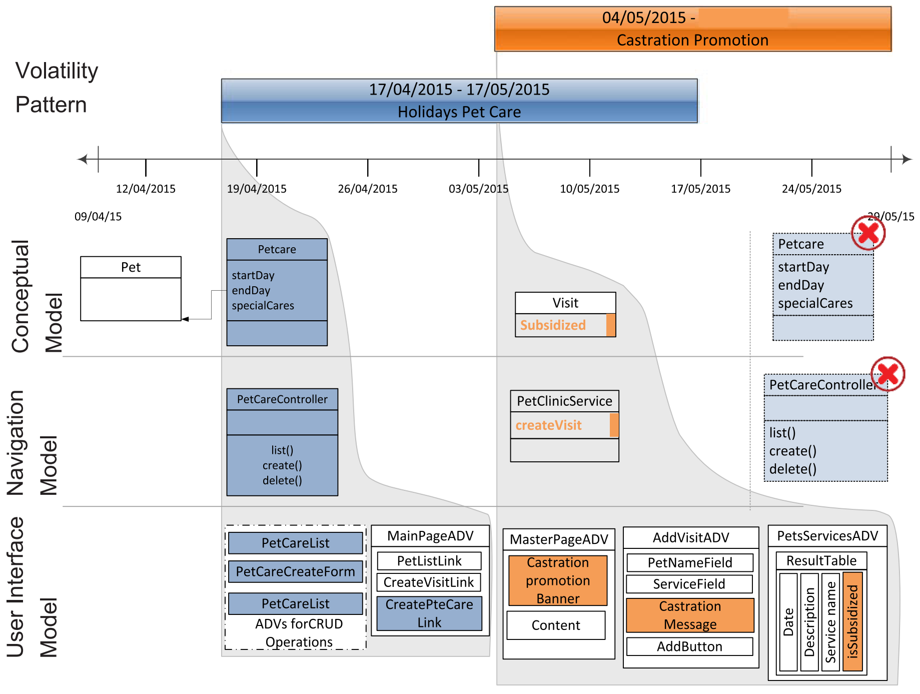


FIGURE 4 Castration promotion and Holiday Pet Care changes' schema for intent, extent, and volatile pattern [Colour figure can be viewed at wileyonlinelibrary.com]

elements (methods, classes, or ADVs) Abstract Data Views in the diagram represent changes concerning the Castration Promotion requirement, whereas the blue changes belong to the Holiday Pet Care requirements.

Below, we present a brief description and source code impact for each Volatile Functionality:

- Castration promotion
  Description: The government aims to promote pets' castration for an undetermined period of time in order to reduce stray dogs and cats present in the cities' streets. Castration services will not pay taxes, and the government will refund 50% of the charge.
  Intent and Extent: The Volatile Functionality comprises the application's 3 layers, as shown in Figure 4. At the conceptual model layer, a flag (subsidized) is introduced into Visit class in order to store whether the pet visit applied for the promotion. At the navigational layer, the logic when persisting a new visit was modified by incorporating a new logic to set it as subsidized when it corresponds. At the user interface layer, there are some adaptations that let users be aware of the active requirement. First, it adds a banner (promotion banner) that notifies users about the promotion, as it can be seen in the Master Page ADV. Then, the user interfaces for registering a pet visit have also been modified. During its workflow, the user selects the required service from the available set of services displayed in a drop-down list. According to this new functionality, the Castration service was added to this drop-down list, as well as a related message container. The message container is used to inform dynamically whether the visit being requested includes the promotion or not, accompanied by a brief description. Additionally, when listing previous Visits, a list of Visits is shown where, for each one, there is a row with the visit's date, its description and 2 new columns have been added. One of them shows the requested service and the other one shows if the visit included the promotion.
  Volatile Pattern: The set of changes must be available since May 4, and it does not have a defined date to be deactivated yet, as it is a Beta feature.
- Holiday Pet Care
  Description: There is an increasing number of holidays in a natural year disposed of several long weekends, which can be used by citizens for traveling. Taking this situation as a business opportunity, the Petshop starts promoting a service for pet care on holidays. For example, customers can leave their pets at the shop and enjoy a long weekend.
  Intent and Extent: This Volatile Functionality also comprises the 3 layers of the web application. At the Conceptual Model layer, it adds the PetCare class, which stores a reference to the involved pet and a check-in and check-out date. At the navigational and user interface layers, it incorporates a complete set of controllers and pages for the Holiday Pet Care Service management with operations for creating, editing, listing, and showing pet care bookings. Finally, it introduces a side link to the Master Page ADV to access the pet care list page, and also a button for requesting a new pet service care from the owner's pets page.
  Volatile Pattern: The set of changes must be available for a month, from April 17–May 17.

4.2.3 | Changes taxonomy discussion

Petshop and ACME applications support different business requirements, implemented using different paradigms; nonetheless, the changes' types are equal in both applications. In spite of the domain difference, changes of these 2 applications share the application impact in business, navigation and interface models. In order to compare maintenance consequences in both applications, we decided to consider only those changes that can be found in both applications. Some of them, such as application initialization scripts and specific new pages, were excluded because they do not emerge in both applications.

To show the equivalence of changes between both applications built using different paradigms and business domains, we have performed a change characterization[33] comparing source code/model level changes. For each equivalent pair of changes belonging to both applications, we have specified its mechanism and the factors that influence such mechanisms.

All the changes share few specific *Temporal Properties* characteristics:

- Time of change. The changes were *static* because they were introduced at compilation time. The lack of volatile support makes it impossible to provide the *dynamic* nature required to enable, become perceivable by the user and deactivate at run-time;
- Change history. Requirements are statically *versioned* prohibiting different application versions from coexisting (only one version is available at a time), carried out in *parallel* with other changes and performing *synchronous changes* on the unique database.

- Change frequency. Changes are applied arbitrarily depending on business requirements.
- Anticipation. Changes are *anticipated* because their volatile nature based on unexpected and unforeseen requirements is only valid on a dynamic period of time.

Changes differ on *Object of Change's* dimension because they impact on different software artifacts. The compromised software *artifact* in the Code-Based application is the source code, where the changes do have partial *propagation* support across application's layers at compilation time because programming languages support static checking. In the case of IFML, the model is the affected artifact with partial *propagation* support between Business model and site view model.

Firstly, we classified changes by application's concern: Database, Business layer, Navigational and Interface Layer. Then, for each kind of change, we assigned a type to classify it as removal, when it comprised the removal of component/table/class or modification, when it comprised the modification of component/table/class. In Table 2, we can see how each pair shares the same purpose (Change type column), but with a different application (Concept column). Finally, we classified the existent equivalence of the changes for both applications defining whether they are equal, different, or not considered in the analysis.

The system must be *always available* despite the changes are *reactive* to business requirements.

Finally, regarding the applications' size, both applications are built by composing different primitives that are not comparable syntactically. The PetShop application comprises 12 pages, 9 groovy classes, 45 methods, and 539 lines of code. Meanwhile, ACME application combined 249 IFML model elements: 11 Entities, 56 Attributes, 54 Content Units, 15 Pages, and 78 Operation Units (ie, Create and Delete units) among others.

4.3 | Participants and material

Because of the complexity and extension of the experiment, we decided to split it into 2 subexperiments, where each kind of application was maintained by different groups of developers that are presented in Table 3.

Said groups' disposition was as follows:

- Model-Driven experiment. A group of 19 developers performed the Model-Driven experiment. It was a mixed group of Master and post-grade students. Nineteen developers were students of a Master degree at Johannes Kepler Universität of Linz, Austria, and 4 students were PhD students at the National University of La Plata.
  Material: English
  CASE Tool: WebRatio
- Code-Based experiment. A group of 17 developers performed the experiment. They were all students attending the last year at the National University of La Plata's bachelor degree (comparable with European Master Degree). More than 95% of developers reported knowledge on Java language enabling them to maintain Groovy-Based code because of the compatibility of both languages. Additionally, more than 65% of developers reported knowledge on modern MVC frameworks (ie, Symphony, Grails, Struts, or Ruby on Rails). In this case, we provided an application's introduction covering architecture, components and core technology for 1 hour before starting the experiment.
  Material: Spanish
  CASE Tool: Groovy/Grails Tool Suite™ (GGTS)

The subjects were on average 25 years old in both experiments. By using the Wilcox hypothesis testing technique for nonparametric samples, we detected that there is no significant evidence to support a difference in subjects' ages; the *P* value was 0.473. In Graph 1, we present 2 Boxplot graphs comparing the distribution of students' ages in both experiments.

The participants were asked to complete the provided expertise questionnaire, which aimed at measuring and confirming whether they had the minimum required skills so as to carry out the presented experiments. Regarding the professional experience in the Software industry, more than 47% of students in the case of Code-Based experiment and more than 63% of students in the case of Model-Driven experiment claimed to have been working as Developer or Tester in a software company for 25 and 27 months on average, respectively. The Wilcox test showed that there is not enough evidence to support the existence of difference in the experience with a *P* value = 0.172. In Graph 2, we present 2 Boxplot graphs comparing the distribution of students' experience in both experiments.

Developers were mostly master students with programming experience where they had been receiving training and development tasks for at least 4 years. In the experiment, they received the target applications in VV and were later asked to remove the volatile functionalities on their own, without any extra guide other than the business requirement introduction: to remove deprecated functionality. We wanted to reproduce a real-life scenario in IT companies, where programmers (employees) are

TABLE 2 Application's changes mapping

Cat.	Change type	Type	Code-Based Concept	Model-Driven Concept	Equivalence
Database quality	Remove a table	⊖	PetCare	Comment	√
	Remove a column	✎	Subsidized in Visit table	Discount in Product table	√
	Delete logically data	N/A	Pets		Excl.
		N/A	Users	Users	√
		N/A	Visits (with subsidized values)	Products (with discount values)	√
		N/A	PetCares	Comments	√
Model	Remove a class	⊖	PetCare	Comment	√
	Remove attribute	✎	Subsidized in Visit class	Discount in Visit class	√
View	Remove entity management page / component	⊖	Castration promotion page	Mother's day promotion page	√
		⊖	Register PetCare request page	New comment component	√
		⊖	Requested PetCare list page	Comment list component	√
		⊖	PetCare update and delete page		≠
		⊖		TOC page	Excl.
	Remove static interface elements	✎	Castration promotion banner	Mother's day promotion banner	√
	Remove dynamic interface elements	✎	Castration promotion notification		≠
	Remove link	✎		TOC	Excl.
		✎	PetCare list		Excl.
		✎	Castration promotion	Mother's day promotion	√
	Remove column	✎	Subsidized in Visits table	Discount and final price in Products table	√
	Remove user interface resources	⊖		Component template	≠
		⊖	Image/Icon	Image/Icon	√
	Libraries references	✎	CSS, Javascript	CSS	√
Controller	Remove entity management	✎	PetCare	Comment	√
	Backend logic modification	✎	Subsidized visits' discounts application	Products' final price calculation	√

References:
√ Equal
≠ Different
Excl. Not considered in the analysis
⊖ Resource or source code removal
✎ Source code modification
TOC, terms of condition.

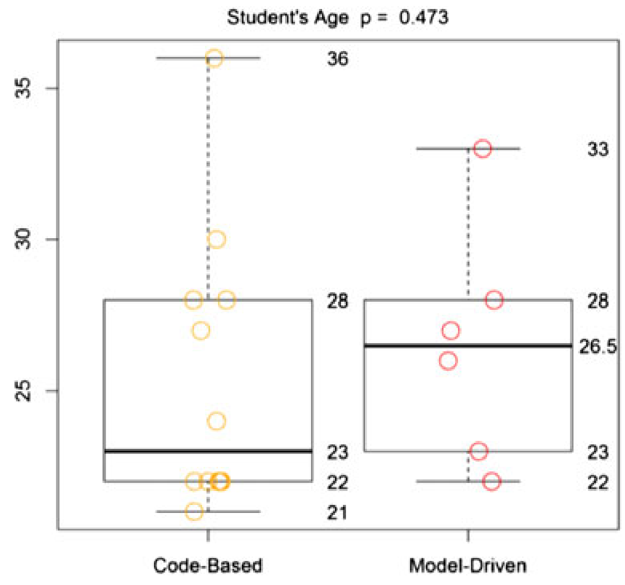
assigned projects dynamically based on human resources' availability, and where experience in the application domain problem is desired but not mandatory. Although they fit in a junior/semi-senior level,[7] developers took several university courses that allowed them to develop their programming skills, making them suitable for these tasks. The use of students as subjects can

TABLE 3 Experiment design

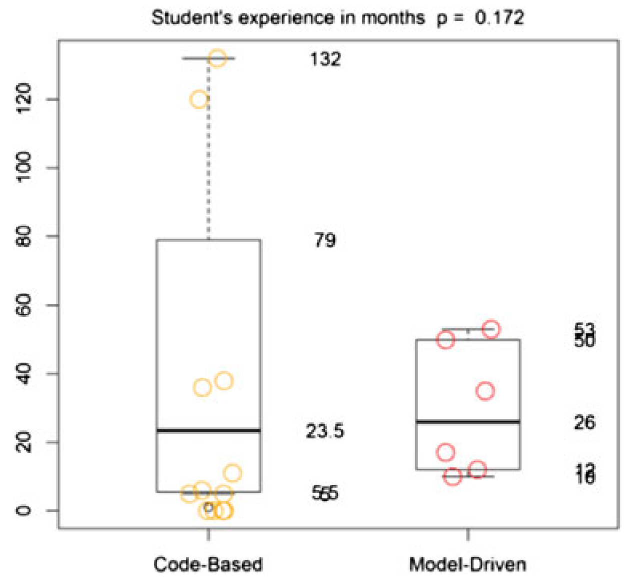
Experiment	Total	Distribution	Country	Language
Code-Based	17	17 students of UNLP	Argentina	Spanish
Model-Driven	19	15 students of JKU 4 students UNLP	Austria Argentina	English Spanish

JKU, Johannes Kepler Universität; UNLP, National University of La Plata.

be considered adequate, as it was suggested in some previous studies,[34-36] and has some benefits[36]: they are more up to date technically than most of the "professionals" practitioners, they are well trained to do the job, the experiment helps to establish a trend, and they have the same awareness of factors that impacts the lead-time than professionals.[35] Since the arise of MDWE, 25 years ago, the adoption of MDWE approaches by the industry has not been the expected. The problem of adopting Model-Driven development has been widely discussed in the general software domain; specifically, many studies show that the problem is more than technical and involves also cultural, managerial, and other kind of issues.[37] The low availability of professionals with experience on MDWE in comparison with the one of coders makes the use of student as subjects suitable for a



GRAPH 1 Comparison of participants' ages [Colour figure can be viewed at wileyonlinelibrary.com]



GRAPH 2 Comparison of participants' experience in months [Colour figure can be viewed at wileyonlinelibrary.com]

fair performance comparison. Furthermore, a preliminary report[38] shows that the performance of students and professionals is not significantly different when applying a new approach without the influence of the experience.

After the developers removed the Volatile Functionalities and got the MV, we performed a quality evaluation of each version (OV, VV, and MV) in order to detect whether changes were detrimental to the overall quality.

Students were motivated and committed to the experiment, as it was part of their final examination, thus earning education credit.

During the experiment, the developer received 2 questionnaires and a document that described the maintenance task. These materials were available in Spanish and English and were provided to developers depending on their language convenience. Additionally, each developer used the official CASE tool for the underlying technology: the Groovy/Grails Tool Suite™[39] for PetCare application and the WebRatio[3] for ACME Application.

4.4 | Experiment procedure

The experiment protocol was executed in the same way with all the subjects of both groups and comprised a well-defined workflow. First of all, the participants of the experiments were asked to complete an expertise questionnaire. Next, they received a brief introduction to the application in which they would have to perform the removal task, along with the required material.

The participants of the Code-Based experiment were given the Petshop application by providing them with a repository with the corresponding application's source code. Each participant counted on an independent SVN repository and SQL database (already loaded with data). Once they had finished the requested task, the participants were asked to commit the resulting source code to their assigned SVN repositories.

On the other hand, the participants of the Model-Driven experiment received the Acme application's source models via email. Besides, each of them was also provided with an independent SQL database (already loaded with data). After completing the removal task, the participants were asked to send us the resulting application's source models via email.

Finally, the participants of both experiments had to complete the Final developers' questionnaire, and also to write a report about the executed analysis.

4.5 | Results processing

In order to achieve the task of processing the collected results, both automatic as well as manual processing were used.

The analysis was performed mostly in an automated way. We wrote different tools for analyzing models and source code for samples obtained from subjects in each experiment. The result of such analysis was a report considering rules, which will be discussed in detail in following sections. Next, we briefly introduce developed tools:

- Source Code Analyzer: a simple Bash script that orchestrates automatic code review for tasks such as check out, code comparison and run Sonar[17] analysis over all of the samples in the case of Code-Based application. Then, these results were processed and displayed in a spreadsheet by using a developed Java project that consumes Sonar's API. Sonar has a myriad of interesting rules, but we have only considered a subset of them, which are listed in appendix C.
- Models Analyzer: a Python script for processing models definitions. The script automatically compared samples against the OV detecting pending changes. Additionally, WebRatio report tools were used to gather the number of model's elements.

Afterward, we performed a manual post-processing task in both cases, which assessed those metrics not well supported by Sonar. Finally, we performed a last manual double-checking review of the script's result in order to avoid incorrect values.

4.6 | Experiment reproducibility

The experiment can be reproduced in other applications based on different modern web technology stacks and Model-Driven approaches. The Code-Based experiment could be reproduced with minor changes because of the fact that modern web frameworks[*,†,‡,§] share an analogous architecture composed by components which give support to MVC pattern, and Object-Relational Mapping (ORM). In the case of the Model-Driven application, the issues analysis is based on comparing models;

[*]https://www.djangoproject.com/

[†]http://rubyonrails.org/

[‡]https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

[§]https://symphony.com/

therefore, the experiment can be reproduced with the most mature MDWE approaches[25,40] because the these share the same primitives for modeling web applications.

The material developed during the experiments is publicly available in a repository.[¶]

5 | ANALYSIS

In this section, we will discuss the assessed consequences of managing volatile in ad-hoc way.

5.1 | Developers' skills

The expertise questionnaire was used in order to measure and confirm the skills of the selected developers. We decided to treat them all equally, focusing on whether they had the required knowledge or not. Because this required knowledge depended on the experiment in which each participant was involved, 2 different questionnaires had been created.

The first one was addressed to the Code-Based experiment participants and can be found in Appendix A as Expertise Code-Based developers' questionnaire. After analyzing the results, we can highlight that all developers had already used a web application, were trained for designing applications using Object Oriented Programming, and had experience programming with Java and PHP languages. Indeed, 95% of the developers have experience of programming using Java language, and 56% had experience with a MVC framework.

On the other hand, the second questionnaire was addressed to the Model-Driven experiment participants and can be found in Appendix A as Expertise Model-Driven developers' questionnaire. It is important to mention that before the Model-Driven experiment presentation, the involved participants received training on WebRatio. Finally, the results show that 89% of the developers claimed to be confident enough to design and run basic web Applications based on WebML. To sum up, the questionnaires answers highlight a homogeneous knowledge of both groups regarding the technologies used in Code-Based and Model-Driven experiments.

5.2 | Developers' perception

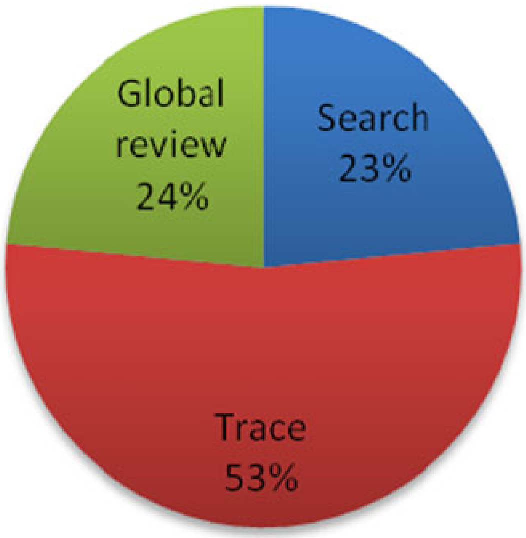
Handling volatile requirements demands modifying and testing for both introduction and later removal of VF to the development team. We analyzed the samples of the 2 subexperiments and focused on the developers' perception regarding the VF removal, as it is where we detected that many of the maintenance problems occur.

5.2.1 | Strategies for identifying changes

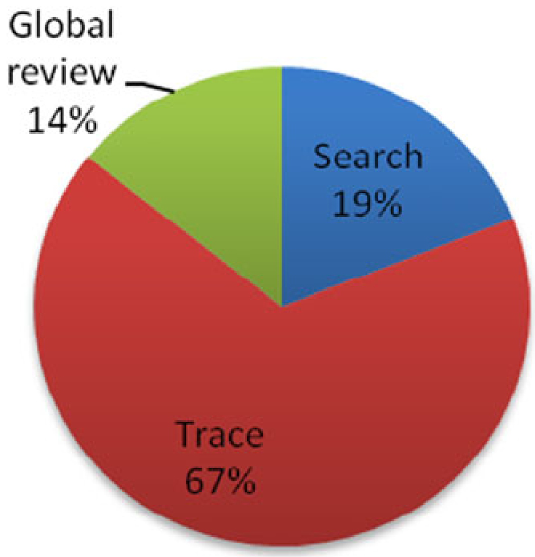
The first step for removing VF is the identification of its software component within any application that should be modified or removed. As part of the *Final developers'* questionnaire, the developers were asked to describe how they identified where to perform changes. The result showed that the developers adopted different approaches for performing this task, which can be summarized as follows:

- Tracing Model-View-Controller flow: The developers started the analysis from the user interface layer, going through controller component up to domain model fields. Their main intent was to get used to the application source code and its MVC framework. After they got the required understanding, they performed some changes: Removing HTML and MVC tags, modifying controllers and business object methods and fields. The developers described their tasks using explicit MVC concepts in the Code-Based case or Model elements in the case of WebML, and tracing the action.
- Search text: based on specific functionality keywords present along different application components, such as “petcare”, “exceeded”, “promotion”, “notified” and so on, the developers performed a lexical search in the source code, which allowed them to identify components compromised by VF. The developers claim that by using this technique they were able to identify artifacts they had not taken into account at the beginning. This strategy has as main drawback; the fact that most web applications are internationalized and, therefore, if the Volatile Functionality's keywords used in the search are not in the same language as the application source code was written in, it will be necessary to map each keyword to the source code language and then perform the search. The developers described their maintenance analysis by referencing explicitly to the

[¶]Material, source code and questionnaires are available at: https://goo.gl/gDmgyD



GRAPH 3 Strategies for identifying changes used by developers on Code-Based approach [Colour figure can be viewed at wileyonlinelibrary.com]



GRAPH 4 Strategies for identifying changes used by developers on Model-Driven approach [Colour figure can be viewed at wileyonlinelibrary.com]

search tool provided by the IDE. In a previous study,[23] authors highlight that this strategy may require to review code irrelevant for maintenance task to developers.

- Global review: A few developers decided to review each project resource in order to understand how the application worked without using the aforementioned strategies. While describing their tasks, they described in a vague way how they performed resource reviewing tasks.

Developers opted for one of these strategies when performing the maintenance tasks. In our questionnaire, we asked them to describe how they were able to identify application sections where changes should be made. After analyzing the result, in the case of Code-Based application, the MVC pattern was taken as reference by 53%, leaving almost half of the developers search by string or ad-hoc techniques for the analysis. The latter results are presentedin Graph 3. On the other hand, werealized that underlying models in the used approach (WebML in MDD) drove the analysis done by developers in the 67%of the cases, as can be seen in Graph 4.

5.2.2 | Time spent on maintenance

As depicted in Figure 2 at Section 4, maintenance was performed in 2 steps: Volatile Functionality introduction and Volatile Functionality removal. The first task was performed by a single Software Architect that evolved both applications considering volatile requirements. He has more than 10 years developing enterprise web applications and more than 5 years developing WebML/IFML-based applications. He only relied on good practices and Object-Oriented paradigm, but he did not use any approach for supporting Volatile Functionality. The task comprised adding and modifying source code and web resources (HTML, Images, etc.), and took 15 net hours. This effort was not considered during the analysis provided that we just focus on the Volatile Functionality removal challenges.

Development tasks demand effort for checking the deliverables' correctness recognized by the software development community. During the maintenance tasks, developers check whether any implemented feature satisfies the software requirements for performing a new release version. These testing tasks consist of testing the application functionality and, if any errors are found, performing a new set of changes to fix such errors, requiring to start the testing cycle all over again. For this process, we have defined 2 metrics:

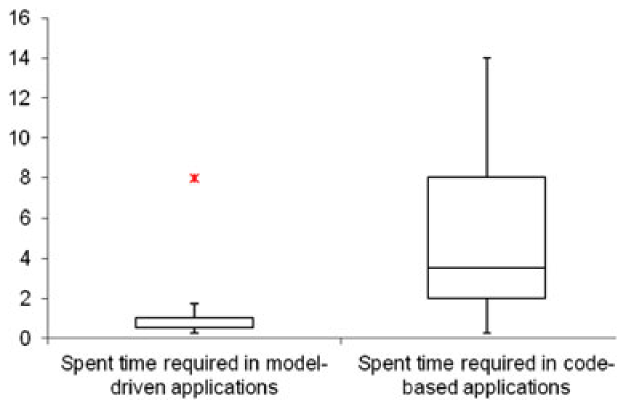
- Amount of testing cycle. As the number of micro loops needed by the developer to introduce any new line of code, test it and, if it fails, start all over again. The more testing cycles are needed, the more time and effort is required.
- Performance confidence. As the confidence over developer's deliverables after introducing any change in an application. To make matters worse, confidence is also affected when the developer is not familiar with the requirements, the application's architecture and the source code. We asked developers if they would suggest that the application was stable and whether they would recommend performing more testing even though the application was considered stable.

Both metrics will help to understand developers' perception about the application's correctness.

After maintenance task, developers expressed that maintaining the ACME application, developed with WebML, required on average 1 hour with an upper outlier of 8 hours. Meanwhile, in the Petshop case, developers spent on average 3 hours and a half, and 14 hours in the worst case. Timekeeping was done by developers during the experiment. In Graph 5, 2 boxpot graphics describe the distribution of the results.

Model-Driven approach showed a better performance for maintenance, requiring about a third of the time for the same activity in a Code-Based approach. As the Code-Based application was implemented using an MVC framework, changes required maintaining artifacts built with several technologies (HTML, JavaScript, CSS, Groovy). Therefore, developers had to consider the relationships among those artifacts across layers; while that was not necessary in ACME application. After applying the T-student method for evaluating the mean relationship, we can conclude that Code-Based approach has a bigger mean than Model-Driven (P value = 0.004 and significance level $\alpha = 0.01$. This confirms a better productivity with more than 360%. This result does not confirm claimed WebRatio productivity of 900% presented in a previous study[41] which even though not being an academic paper (there are no academic performance evaluations yet on IFML) is the result of the evaluated systematically and based on a large number of projects.

Besides, we have also analyzed the resulting time spent by taking into account each of the strategies presented in the previous section individually. In the case of Code-Based experiment, the average time spent when using *Tracing Model-View-Controller flow* was 5 hours and 20 minutes; when using *Search text* was 4 hours and 15 minutes; and when using *Global review*



GRAPH 5 Spent time required in Model-Driven and Code-Based applications [Colour figure can be viewed at wileyonlinelibrary.com]

was 4 hours and 7 minutes. The order of the average time required for each strategy in the case of Model-Driven experiment was exactly the same as *Tracing Model-View-Controller flow*; it took 1 hour and 25 minutes; *Global review* 1 hour and 11 minutes; and *Global review* 50 minutes. Finally, we are not able to suggest that the use of one of the strategies is faster than the others, provided that the fragmentation of the samples into small subsets makes them unsuitable for the application of statistical methods.

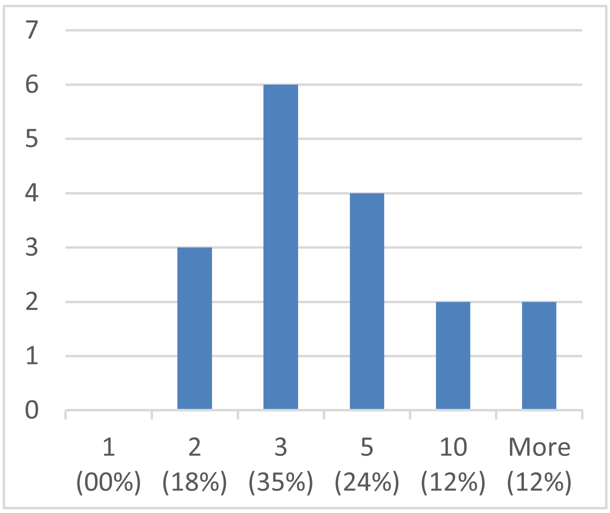
5.2.3 | Change quality and confidence

We aimed at assessing the Developers' confidence in their job and application's quality as long as they performed changes.

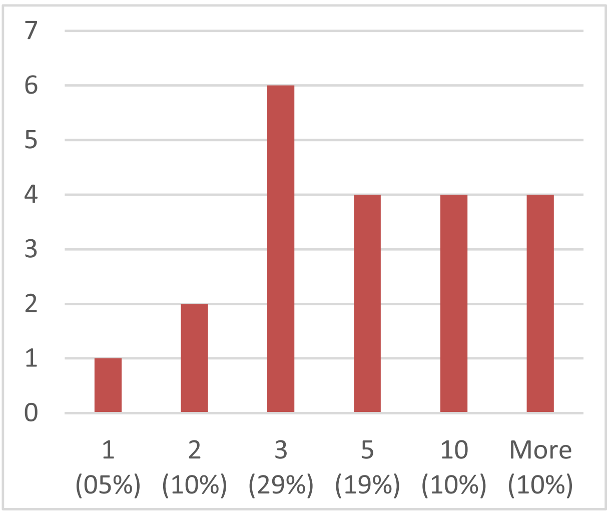
It is noteworthy that during the development process, all developers reported to have introduced errors (in the Final Developers' Questionnaire), no matter which approach they used. These were detected by means of testing and, after that, they were fixed. This finding points that development tasks of Volatile Functionality maintenance are error prone, and therefore, expensive in terms of time and effort. That is, after a few application modifications, the developers performed smoke tests for checking whether the application behaved as expected. When an error was detected the developers needed to restart the development cycle in order to fix the error and start all over again. This metric is the result of combining diverse factors related with the developer: experience, skills, confidence, and motivation. We aimed at capturing effort required on validation quantitatively but not qualitatively; thus, we did not consider the reason why developers performed a given number of testing cycles.

In Graphs 6 and 7, the testing cycles for Code-Based application and Model-Driven one are respectively shown. Regardless of the approach, Volatile Functionality's maintenance proved to require several correction iterations in order to obtain a stable version. Only one participant stated that he used a single testing cycle and lest that 18% of developers argued that they were able to obtain an application in just 2 attempts. The testing cycles metric is one of the key metrics that remarks the lack of separation of concerns technique because it is an activity that can be avoided by using, for example, Aspect-Oriented Programming (AOP) techniques. By relying on AOP, aspect target is not modified manually because the weaving process of aspect's advices is performed by an engine. In the same way, the removal of an aspect is a simple task that only requires to recompile the application excluding the aspect, or disable the aspect depending on whether the aspect technology is runtime or interpreted. For User Interface, there are several tools that allow implementing AOP concepts[42] or advances approaches that support separation of concerns in web applications such as in a previous study[1]

After performing changes, most of the developers assured that their version of the application was stable in both approaches (94% in Code-Based and 95% in Model-Driven). However, Fisher-Irwin test computed a 1 P value, and therefore there is not enough evidence to reject the H_0. On the other hand, 41% of developer in Code-Based experiment and 62% of developer in Model-Driven said the application should have a final testing phase. Fisher-Irwin test reported 0.33 P value. Developers of Model-Driven web applications showed less comfort regarding reliability compared with the Code-Based one. This is not surprising as a specific testing phase is widely adopted in the software development process.



GRAPH 6 Required testing cycles for Code-Based development [Colour figure can be viewed at wileyonlinelibrary.com]



GRAPH 7 Testing cycle for Model-Driven in development phase [Colour figure can be viewed at wileyonlinelibrary.com]

5.3 | Taxonomy of detected defects

Once the developers had finished the required tasks, they delivered the application Maintenance Version. With these projects, we performed a deep review task, covering each application layers: database, business layer, navigational layer, and user interface layer.

5.3.1 | Database quality

Each Volatile Functionality handles, along with different components, databases for storing object states. For instance, depending on the required functionality, new tables or columns may be added in the application database scheme for storing any data associated to the functionality. In the ACME example, 2 new columns were required to point out whether a given furniture included a discount flag and its corresponding discount value. On the other hand, in the Petshop Volatile Functionality examples, a new table was required for storing Pet Care service. Although persisted information belongs to a Volatile Functionality, its associated records and columns are not recommended to be deleted when it is deactivated or removed because of its business intelligence and legal value. On the other hand, a wrong decision about database schema may lead to unnecessary and inaccurate data.[43,44] Therefore, database archiving[45] strategies should be designed.

In the revision process performed over different maintenance versions of running examples, the developer produced frequent issues described next:

- Complete database information loss. This metric evaluates whether or not a developer erased his assigned database by mistake. In order to identify this situation, it is necessary to validate if the database was dropped and optionally populated with new records. After maintenance work, a developer provides an application version where the database is empty. Dropping data is unacceptable in real life projects[45]; therefore, a database scheme migration task should be analyzed, designed, and implemented with its associated time and effort.
  This error arises because both application frameworks use an ORM framework, which simplified the persistence task, avoiding to deal with SQL sentences. Class variables are automatically mapped to a table column. When programmers attempted to remove some no longer needed instance variables, they later faced errors while updating the corresponding tables. This was due to the fact that the columns corresponding to such variables were set as not accepting null values (aka, nullable columns). As a result, the developers decided to drop the whole database and let the ORM framework define the new scheme.
- Volatile Functionality's data loss. Although a Volatile Functionality is alive for a period of time, the information gathered while it is working is worth for different purposes, such as user experience analysis or business intelligence. Volatile Functionality data deletion without its proper backup does not allow its later usage.
- Useless data generation. In this case, data associated to an outdated requirement is generated when new information is stored in the database. For example, in the case of the Petshop application, some developers did not remove or modify

the logic that sets subsidized flag hindering data processing. In the case that an analyst is required to detect the proportion of subsidized service, he may study the information considering all available rows in database instead of only those rows recorded during the period when Volatile application was enabled. When Volatile Functionality is removed, developers overlook the logic responsible of persisting (often SQL sentences) allowing data being persisted. Additionally, as time goes by, there is no way to identify the requirement owner of such data.

It is important to highlight that *Volatile Functionality's data loss* and *Useless data generation* issues are not contradictory In the case of relational database, *Volatile Functionality's data loss* can be avoided by keeping the column and *Useless data generation* can be avoided by allowing null values in the columns. In this way, SQL inserts sentences for new records just by skipping Volatile Functionality columns.

In Table 4, we show Data quality issues identified in the assessment. Results show evidence about a lack of commitment with data management. The loss of database information (*Complete database information loss*) may be read as a sign of developers' immaturity and excess of confidence about the development environment. This situation shows that developers were not aware of data value for a business. From a practical point of view, most of the developers preferred just to modify the application without manipulating the database (useless data generation). For example, deleting columns.

Results show that users using both approaches ignored database schema management. Volatile Functionality data loss rated 21% in the case of Model-Driven approach and Code-Based rated with 35%. Useless data generation was an issue present in most of the applications valued in 84% and 65% for Model-Driven and Code-Based approaches, respectively. Finally, 18% of Petshop applications' developers erased the whole database. During the experiment, we realized that developers faced a problem with the database schema definition and the application ORM framework. After removing a variable, automatically generated *insert* SQL sentences stopped sending as parameter the removed variable's value and then the database raised an error because there was a column expecting a value (the removed variable's column was marked as not-null). In order to continue, developers decided to erase the database and let the ORM framework regenerate the database scheme, completely losing its data. In the case of WebML, the language does not support required attribute flags at business model because it is managed at navigation layer as form fields' constrains. Therefore, ACME's developers did not have to face this issue.

When analyzing Fisher test's P value of CDBIL, we are able to reject an H_0: *“Code-Based and MDD are equally prone to suffer CDBIL issue”* with a significance level $\alpha = 0.1$. Meanwhile in the case of VFDL and UDL, we are not able to reject H_0: “A developer is equally prone to delete volatile information by using Code-Based or MDD approaches” and H_0: “MDD generates as much useless volatile information as Code-Based approach does”; respectively.

5.3.2 | Business and Navigation layer defects

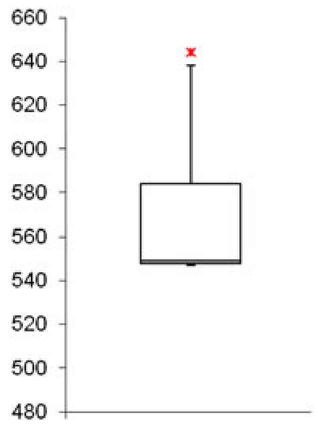
In this section, we describe code defects that pose a problem to the application's maintenance. Problems depicted here were not able to be detected automatically and demanded a manual analysis for each sample.

- Size. This metric counts the number of classes, accessors, functions (methods in Groovy), lines, and statements among others. For the sake of simplicity, we will focus on the number of classes and lines of code in the Code-Based application and number of model's elements in the case of WebML models. Such metric was obtained by using Sonar's ncloc metric for the Code-Based project (box plot shown at Graph 8) and WebRatio tool to count the number of elements present in de diagram for the Model-Driven one (box plot shown at Graph 9).

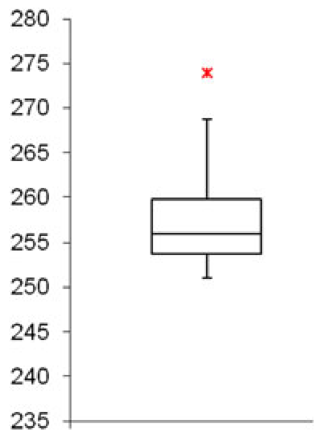
  Conclusion: This metric points out that most developers left pieces of application unattended after the maintenance task. In the case of the Groovy application, an increase of 5.7% of Volatile Functionality's sentences was found; they can be

TABLE 4 Database quality assessment results

	Complete database information loss	Volatile Functionality data loss	Useless data generation
Model-Driven	0%	21%	84%
Code-Based	18%	35%	65%
Fisher test's P value	0.095	0.463	0.255



GRAPH 8 Box plot of Petshop application size in terms of Lines of Code [Colour figure can be viewed at wileyonlinelibrary.com]



GRAPH 9 Box plot of Acme application in amount of diagram elements [Colour figure can be viewed at wileyonlinelibrary.com]

eventually or never executed. In the case of the WebML, an increment of 2.8% model's elements[#] was found with the same consequences. The size has increased in both cases at a low rate, without strong evidence of having an outstanding performance of one approach over the other. Because of both applications belonging to different approaches, it is not possible to compare their size directly.

- Unreferenced Domain Model Variables. One observed problem was the existence of unreferenced variables; that is, variables that were defined but not used in the application. Their only use found was in their never referenced accessors in the case of Code-Based application. In WebML, we only considered regular entity's attributes, because we consider derivate attributes as business logic in next metrics. In the case of the Code-Based application, we computed this metric by combining Sonar's “unused variable” issues and Source Code Analyzer (presented in Section 4.5), while for WebML, we performed a manual processing based on the results obtained by the Models Analyzer (presented in Section 4.5).
  Conclusion: 42% of developers left domain model attributes in WebML's Model diagram. In the case of Code-Based application, 59% of developers left unattended classes' instance variables corresponding to domain model. Some developers argued that they intentionally left these pieces of code, as they believed that the functionality was useful and could be used in the future.
- Unreferenced Domain Model Classes. Related to the previous issue, developers also left classes or entities definition in the business model. These entities make model understanding hard, as their validity is confusing. For the Code-Based application, we identified such defect with Sonar's “classes” metric, and for the WebML application, we performed a manual processing based on the results obtained by the Models Analyzer (presented in Section 4.5).

[#]The size computation considered the amount of WebML abstractions: Entity, Attribute, Relationship, Details, Form, Get, Module, Multiple Form, Message, Output Port, List, Script, Selector, Time, Create, Delete, Init Job, Input Port, Is Not Null, KO Port, Login, Logout, Mail, Update,Module, No Op, OK Port, Parameter, Collector, Reconnect, Selector, Switch and Time.

TABLE 5 Business and Navigation layer assessment results

	Size	Unreferenced domain model variables	Unreferenced domain model classes	Non-executable behavior	Improper behavior
Model-Driven	2.8	42%	16%	16%	47%
Code-Based	5.7	59%	35%	24%	47%
P value	--	0.749	0.255	0.684	1.0

Conclusion: Model-Driven applications were less error prone to this issue, which was identified in 16% of the samples, while in Code-Based it comprised 35% of them.

- Non-executable behavior. Developers introduced this defect by leaving pieces of code unreachable. By leaving the behavior correctly syntactically, but incorrectly semantically or even commenting it out, the code is never considered to be executed. In the case of WebML models, we only consider model's elements that were defined in such a way that are not executable because WebML does not allow commenting model's elements. This metric was obtained manually in both applications by reviewed differences between application's states
  Conclusion: Model-Driven applications presented 16% of non-executable behavior elements, but Code-Based ones showed a higher rate of 24% of commented code.
- Improper behavior. When a developer disables and removes volatile functionalities, he leaves ungoverned pieces of *active code* scattered in the application. Ie, application's behavior that is executed and its result is not shown, and business rules that are executed without a valid business value. When not documented, this sort of behavior, seen from a developer's point of view, is valid because it does not report any compiling errors and the application works without showing any unexpected behavior; thus, it is never called into question. In Model-Driven applications, we considered derivate attributes, which are not considered in any component's setup, and components that are hidden. Both elements are always processed, but never executed. We got this metric for both applications by analyzing manually the comparisons obtained by the tools presented in Section 4.5.
  Conclusion: In this case, both approaches measured that 47% left code that is executable, but its end result is not perceivable to the end-user.

In Table 5, we show Business and Navigation layers' defects identified in the assessment. We can realize that Model-Driven Development approach performed better than Code-Based approach reporting lower rates in the Unreferenced Domain Model Variables, Unreferenced Domain Model Classes, and Non-executable behavior metrics. Regarding the size metric, in both cases, it increased in a small percentage that seems to be unrelated to the other metrics. We can conclude that the size is not an indicator of internal quality detriment. We did not evaluate the statistical significance of size metric because the ordinal values were not directly comparable.

High *P* values for *metrics Unreferenced Domain Model Variables*, *UnReferenced Domain Model Classes*, *Non-executable behavior*, and *Improper behavior* show that we cannot reject H0:$p_{MDD} = p_{CB}$.

5.3.3 | User Interface layer defects

The User Interface is the application's layer containing the set of components that produce the major impact on user perception about the application because a defect in the application's User eXperience may imply a failure in the system adoption. To make matter worst, User Interface layer faces its own development and maintenance challenges.[42,46] Next, we discuss some abstract metrics that allow measuring the maintenance impact in both realms Code-Based and Model-Driven:

- Unreferenced User Interface Resources. In some cases, developers removed business model entities and UI controllers, but they forgot to remove related User Interface documents such as HTML or JavaScript files. For WebML, we considered component template as resources.
  Content introduced by unforeseen requirements often includes media resources, such as videos and images, in order to provide rich user interfaces and gain the visitors' attention. If these resources are not removed when disabling the HTML code, they tend to remain there. This metric was obtained manually.
  Conclusion: Developers of Model-Driven paid more attention when removing resources than Code-Based ones because this metric rated 69% and 88%, respectively.
- Unused User Interface Artifacts. There are resources that can be considered as artifacts containers. JavaScript and cascading style sheets (CSS) files are examples of them. Although these resources can be well referenced because they contain artifacts that

correspond to core functionalities, they are typically not taken into account when removing volatile functionalities. The HTML code introduced by volatile functionalities may be correctly removed, but if there were artifacts that also corresponded to them, they should be considered for the removal too; otherwise, they would be left unused. This metric was obtained manually.
There are 2 main reasons why this occurs. The first one is that developers forget to remove these artifacts, and after some time, they do not have any idea where they are used. And the second one is that developers believe that these artifacts could be used somewhere else and in consequence they leave them untouched.
Conclusion: Artifacts were left less frequently on Model-Driven applications than Code-Based. However, Model-Driven applications presented a high rate of 69%, whereas in Code-Based almost every sample (94%) presented this issue.

In Table 6, we show User Interface layer defects identified in the assessment. Taking into account only these 2 metrics, we have no evidence to suggest that one approach performs significantly better than the other when maintaining user interface artifacts.

When analyzing P values show that we can reject with a significance level $\alpha = 0.1$ that *Unused User Interface Artifacts* issues are prone to appear with the same probability. Although *Unreferenced User Interface Resources*, we cannot reject that developers left equally resources unattended, the P value gives advantage to Model-Driven approach.

5.4 | Overlooked functionality

Maintenance tasks consequences on a web application can be negative either for its internal quality or for its budget. During this section, we posed evidence of issues sources that should be considered in order to preserve the application's quality. When performing the assessment, we came across the interesting fact that developers focused on removing perceivable elements on the User Interface layer associated to the Volatile Functionality in such a way that the application looked and behaved as expected. Nonetheless, the internal application quality has decreased.

In Table 7, we can notice that applications using Model-Driven were less prone to have either perceivable or not perceivable pending changes in 2 of the 3 layers. Additionally, we can remark that the proportion of developers using Model-Driven left less perceivable pending changes than the proportion of Code-Based approach. Based on analyzed data, as WebML derivate most of its interface code, developers did not have to spend time maintaining User Interface's artifacts. Thus, it reflected a better performance at "Pending User Interface layer changes" that is confirmed with a P value 0.0033 below the α level (0.05).

5.5 | Lessons learned

In this section, we can remark 2 important lessons learned. First, during the analysis, we realized that automated tools for analyzing source code quality may fail at detecting source code issues implying false positives. In order to overcome this difficulty, we had to review those doubtful SonarQube metrics manually. Secondly, we realized that developers spent around 20% of the experiment's time on setting up development environments (ie, download IDE, check out source code from repository, download required libraries, and run it) on each Personal Computer. In a second round of experiment, we prepared a virtualization that provided a virtual machine with the IDE already set up and the required libraries already downloaded, which reduced

TABLE 6 User Interface layer assessment results

	Unreferenced user interface resources	Unused user interface artifacts
Model-Driven	69%	69%
Code-Based	88%	94%
P value	0.225	0.085

TABLE 7 Expected modifications overlooked per layer and a count of perceivable elements left behind.

	Business layer changes	Navigational layer changes	User interface layer changes	Perceivable changes
Model-Driven	53%	53%	58%	42%
Code-Based	65%	29%	100%	47%
P value	0.516	0.192	0.0033	1.0

the time substantially and improved developer experience. As it was assigned a personal source code repository and database to developers, virtualization usage only required the project download and database setup for each of them.

5.6 | Research questions analysis

In this section, we will discuss each research question considering the result of the metrics evaluation. In Table 8, we summarize the results for each research question and its metrics. The preliminary result of this work rejects maintainability[4] benefit of Model-Driven Development.

5.6.1 | RQ_1 Is Model-Driven Development more productive than Code-Based?

In favor of Model-Driven Development, it was shown in Section 5.2.2 that it requires considerably less effort to implement web applications with a similar internal quality than using a Code-Based approach. We analysed the time spent by developer on performing maintenance tasks. The maintenance task effort for Code-Based application was more than the needed by the Model-Driven one. There is sufficient evidence to support the claim that Model-Driven Development performed better than Code-Based with an α level 0.01 of significance.

On the other hand, this result rejects some works claiming that there is no performance difference reported in a previous study[6] and confirms empirically IFML better productivity,[41] but with a lower rate in this scenario (about 3.6 times instead of about 9 times).

TABLE 8 Research questions and metrics measurement summary

	Research question	Metrics	H_0	Conclusion
1	Is Model-Driven Development more productive than Code-Based?	Spent time	$TS_{MDD} = TS_{CB}$ (T-student)	√
2	Is Code-Based approach more prone to have application external quality issues?	Pending User Interface layer changes	$PUILC_{MDD} = PUILC_{CB}$ (Fisher-Irwin)	√
3	Is application internal quality more negatively affected by Code-Based approach than Model-Driven Development?	Unreferenced domain model variables	$UDMVC_{MDD} = UDMVC_{CB}$ (Fisher-Irwin)	×
		Unreferenced domain model classes	$NEB_{MDD} = NEB_{CB}$ (Fisher-Irwin)	×
		Non-executable behavior	$IB_{MDD} = IB_{CB}$ (Fisher-Irwin)	×
		Improper behavior	$UUIR_{MDD} = UUIR_{CB}$ (Fisher-Irwin)	×
		Unreferenced User Interface resources	$UUIA_{MDD} = UUIA_{CB}$ (Fisher-Irwin)	×
		Unused User Interface artifacts	$ASP_{MDD} = ASP_{CB}$ (Fisher-Irwin)	√
4	Is developer confidence affected by Model-Driven Development?	Application stability perception	$TR_{MDD} = TR_{CB}$ (Fisher-Irwin)	×
		Testing requirement	$CDIL_{MDD} = CDIL_{CB}$ (Fisher-Irwin)	×
5	Is application data quality compromised by Model-Driven Development?	Complete database information loss	$CDIL_{MDD} = CDIL_{CB}$ (Fisher-Irwin)	×
		Volatile Functionality's data loss	$VFDL_{MDD} = VFDL_{CB}$ (Fisher-Irwin)	×
		Useless data generation	$UDG_{MDD} = UDG_{CB}$ (Fisher-Irwin)	×

5.6.2 | RQ_2 Is Code-Based approach more prone to have application external quality issues?

In Section 5.4, we analysed how required source code changes where overlooked depending on the layer the change belonged. The results showed that all Code-Based application reported pending UI changes; that is, UI elements were overlook and kept in the UI source code, references and UI components were declared but never used, UI code was commented and left unattended, etc. Therefore, there is sufficient evidence to support the claim that Code-Based approach was more prone to have external quality issues with an α level of 0.05.

5.6.3 | RQ_3 Is application internal quality more negatively affected by Code-Based approach than Model-Driven Development?

In Sections 5.3.2 and 5.3.3, we have discussed in detail diverse issues that compromised different application layers. For example, classes were left unattended in the domain model; instance variables were declared but never used; pieces of code were commented, or modified in such a way that it produced a wrong or non behavior; and similar problems were found at Interface layer.

For *Unreferenced Domain Model Variables, Unreferenced Domain Model Classes, Non-Executable Behavior, Improper Behavior*, *Unreferenced User Interface Resources*, and *Unused User Interface Artifacts* metrics, there was not enough evidence to support the claim that Code-Based development had a better performance than Model-Driven one and the other way around.

5.6.4 | RQ_4 Is developer confidence affected by Model-Driven Development?

The developers' confidence during performing the maintenance task was low. In both Code-Based and Model-Driven development, they needed much effort on testing, and the final result was not reliable from their point of view. Base on the *Testing Requirement* and *Application Stability Perception* metrics, there is not enough evidence to reject H_0 where there is no difference in the confidence performance of both approaches.

The maintenance Volatile Functionality removal based on ad-hoc approach requires costly testing during development to be reliable. Although most developers stated that their MV applications were stable, the testing effort during its development required several micro testing cycles for half of the developers. However, many of them also recognized that they would still need a testing phase, and therefore did not entirely rely on them.

5.6.5 | RQ_5 Is application data quality compromised by Model-Driven Development?

In Section 5.3.1, we analysed databases and its data reporting that there is an important issue about how developers take care of databases. Developers deleted valid information useful for business intelligence activities, and also did not consider that the system generate invalid data that could be wrongly considered for decision making. During the analysis, we evaluated *Complete database information loss*, *Volatile Functionality's data loss*, and *Useless data generation metrics* in order to evaluate whether or not there is a performance difference between approaches. The outcome suggests that there is no evidence to reject H_0 and no approach performs better than the other. The data quality analysis was not considered in other performance comparison between Code-Based approaches vs Model-Driven in the context of Web Engineering.[5,6,47]

5.7 | CONCLUSION

As summary, Model-Driven only performs better in 2 aspects *Complete Database Information Loss* and *Unreferenced User Interface Resources*; both were discussed in Sections 5.3.1 and 5.3.3, respectively.

After analyzing registered data, we can conclude that there is not enough evidence to suggest that a Model-Driven approach performs better than a Code-Based approach when maintaining software in an ad-hoc way. The outcome highlights that managing certain maintenance tasks, such as Volatile Functionality removal, is detrimental to the internal and external quality. Using Model-Driven Approaches does not ensure high models quality (in this case conceptual, navigation, and interface models); indeed, it introduces quality issues as Code-Based approaches do in MVC pattern's layers. One possible answer is that a complementary approach that allows separating, encapsulating, and tracing new concerns should be considered during the development process. We have been exploring these kinds of approaches,[1] but they are not widely available; in any case, there is no clear evidence that this kind of solution will perform better in all quality aspects analyzed in this paper.

The results can be generalized to other MDWE approaches leaving aside each approach's specificities. The approaches often have similar primitives[25,40] for modeling web applications as well as a similar responsibility distribution among their models (conceptual, navigation, and user interface models). Firstly, in MDWE approaches, the developer defines a conceptual model where business entities and their attributes are defined (E/R diagrams or Class diagrams are used commonly). Secondly, he must design how the application should behave from a navigation point of view by using custom diagrams or extended UML-based diagrams. Finally, depending on the approach, User interfaces may be designed abstractly. Therefore, the evaluation done with IFML case study can be generalized to other MDWE approaches because the problems abovementioned can be easily found in other approaches like UWE and OOHDM. In the case of Code-Based case study, it was based on a modern web framework that is built upon components which give support for the Model-View-Controller pattern and ORM, and considered in the development approach other coding principles such as "convention over configuration" and "don't-repeat-yourself" which helps in improving the productivity. In this kind of web frameworks, firstly, the developer defines a class model (describing business entities, their attributes, and their behavior encapsulated in methods) that automatically is enhanced with Object-Relational Mapping features for object retrieval and persistence. Secondly, she specifies the controller objects responsible for managing User Interfaces events in order to mediate with business object for presenting information to the user or updating the database with changes. Finally, she develops dynamic User Interfaces based on JSP-like technology rendering HTML and CSS content, which is rendered by the browser. Despite the underlying language of the modern frameworks like Django or Ruby on Rails, we can find the set of discussed issues in these frameworks as they share a similar architecture based on a clear separation of concern provided by MVC pattern which helps managing business entities, controller objects and user interface aspects independently, and ORM for abstracting databases operations.

6 | THREATS TO THE VALIDITY

There are several threats to validity that will be described next.

6.1 | Threats to conclusion

In order to avoid the introduction of bias in the conclusions, we gather results from 2 different groups of 17 subjects each that is a large enough sample for applying statistical test techniques on most of the metrics. The sample's size allows applying T-student technique without the need of satisfying normal distribution.

The use of questionaries' with closed questions and source code as experiment objects made it difficult to influence the results because they are not subjected to researcher point of view. Indeed, the use of tool for analyzing (described in Section 4.5) prevent the introduction of subjective error than manual result processing. Subjects performing the experiment were master students reducing the risk any bias from subjects' heterogeneity.

Applications used as case study satisfy different functional requirements in different business domains. However, this was designed in that way in order to confirm that maintenance challenges are not related only to a given business domain, and to make them comparable, we defined a different change set for each one but with the same change sets' taxonomy. Therefore, the underlying changes from a technical point of view are pretty similar. For example, in both applications, it is required to introduce a new entity (PetCare service in Petshop application and Comment in Acme Application). In Section 4.3, we summarized the change types and how they were instantiated in each application pointing out whether they are similar or not.

6.2 | Threats to construct

The Code-Based web application chosen uses well-known full-stack framework providing an implementation of MVC pattern and Object-Relational Mapping framework. The application's source code was provided by the open source community, as it was described in Section 4.2. Diverse Internet crawlers that are focused on technology trends, such as *W3Techs* and *BuildWith,* highlight PHP and ASP.net as leaders[‖,**] in the server side programming languages, with more than 98% of the Internet Accessed applications. On the other hand, Java has been one of the most popular languages[††] since the 2000s, and Java is preferred[‡‡] in high traffic sites over PHP and .Net. In order to complete this research, an evaluation over PHP and .Net technologies would confirm results presented in this paper.

The Model-Driven application was based on IFML,[48] which has recently been promoted to an OMG standard[49] for describing User Interface interactions allowing to model a complete web application. The maintenance tasks were supported using the official IFML platform: WebRatio.[3]

6.3 | Threats to internal validity

The participation of students in the experiment was in 2 main groups: Code-Based experiment and Model-Driven one. Each of these groups was defined in such a way their members have a similar professional development of subjects preventing Maturation and Historical factors to affect negatively to the research. The subjects' experience is pretty much similar because they have taken most of the lectures together and they are homogenously trained at their university. If we had invited IT industry developers, it would had been quite difficult to ensure similar skills between them.

6.4 | Threats to external validity

Developers were taking bachelor degree's last courses at university with strong background in procedural, object-oriented programming, and even functional programming in a few cases. Although they are skilled developers, it would be convenient to reproduce the experiment with Semi-Senior or Senior developers. Eg., developers who have several years of experience in developing productive software and experience on maintenance problem-solving. The lack of experience in productive projects may be the reason for some issues detected in our previous analysis, such as *Useless data generation,* as described in Section 5.3.1. The occurrences of these issues may be decreased as developers gain experience in projects.

The samples' homogeneity and size helped to present a preliminary analysis, and the selected participants for the execution of the experiment can be considered adequate as was suggested in some previous studies.[34,35] Nonetheless, it would be beneficial to invite a broader range of developers (IT industry developers from several countries) and others target of participant such as analysts or tester in order to present more evidence about the challenges for managing VF.

Regarding to the subject's spirit, experiment observers did not realize any perceivable problem that may influence the experiment result.

7 | CONCLUSIONS AND FURTHER WORK

In this paper, we presented a thorough comparison of managing web application's Volatile Functionality maintenance with ad-hoc approach and a Model-Driven one. During the analysis, we were able to identify how the application's internal quality was negatively affected by the removal of functionality. As the selected applications belong to different realms, we defined abstract rules that allowed us to measure both applications, ensuring fairness in the assessment process. This work presents a preliminary result supporting that there is no significant difference in the performance of a Model-Driven approach against a Code-Based approach when managing unexpected and unforeseen requirements. The analysis showed that equivalent issues appear in application's layer no matter whether it is based on a MDWE approach or coded using a novel Web Framework.

One of the most important findings was the useless data generation issue and the deletion of records on database, as it was discussed in Section 5.3.1. Information plays a first-class role at any business, because it allows to define sale strategies and budget, among others. Developers often do not deal with production environments and therefore they are not aware of data's value.

[‖]http://trends.builtwith.com/framework/programming-language—last accessed February 23 of 2015

[**]http://w3techs.com/technologies/overview/programming_language/all—last accessed February 23 of 2015

[††]http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html—last accessed February 23 of 2015

[‡‡]http://w3techs.com/technologies/market/programming_language/20—last accessed February 23 of 2015

Based on the posed evidence, we can highlight following insights:

- Models help to lookup element references across layers. Models from WebML and Groovy's MVC pattern were used as reference for identifying where to perform changes.
- No approach highlights on developer's confidence about correctness of their deliverables. Both approaches required several testing cycles for achieving a stable application version.
- Data quality should be treated as first-class citizen. In Section 5.3.1, we presented 2 found issues related to data quality, which compromised all of the samples.
- Multi-layers architecture poses challenges to measure application quality. Because of the variety of layers' technology, we showed that specific metrics must be defined and audited in order to identify issues. Additionally, tools for analyzing layers' binding must be considered.
- Volatile Functionality must be supported with an approach. Both developer's effort reduction and internal quality preservation is error prone and costly with an ad-hoc approach. This fact support diverse approaches based on separation of concern such as in previous studies.[1,27]

Last but not least, to our knowledge extent, this study provides the first empirical study about internal quality strengths and weaknesses of the different paradigms with respect to their adoption intent.

During the assessment, identifying quality issues on Model-Driven was difficult as there is not an available catalogue of design, security, and usability flaws. On the other hand, for Code-Based applications, there are plenty of tools for such task. We plan to analyze and catalogue flaws in Model-Driven approaches.

We plan to assess the improvement of software quality when using a framework compliant with guidelines provided in a previous study[1] for handling Volatile Functionality. Additionally, we expect to reproduce the experiment in diverse companies in order to gather more evidence about consequences of maintaining unexpected and unforeseen requirements that are valid for just a period of time.

Regarding ad-hoc web application maintenance, we plan to perform the same assessment with different approaches, such as pair programming, test-driven development, and code review, in order to assess if there is an improvement in the application's quality.

Finally, we will study how issues are distributed across different layers. In the assessment, we will evaluate whether deeper layers are more prone to have issues than perceivable layers. Ie, Business layer is more error prone than Navigational layer.

REFERENCES

1. Urbieta M et al. Modeling, deploying, and controlling volatile functionalities in web applications. *Int J Softw Eng Knowl Eng*. 2012;22:129-155.
2. Klinger, T. et al. An enterprise perspective on technical debt. In: Proceeding of the 2nd working on Managing technical debt - MTD '11. 2011:35.
3. WebRatio Platform, http://www.webratio.com/site/content/en/web-application-development.
4. Martínez Y et al. Empirical study on the maintainability of web applications: Model-Driven engineering vs code-centric. *Empir Softw Eng*. 2013;19(6):1887-1920.
5. Martínez Y et al. MDD vs. traditional software development: a practitioner's subjective perspective. *Inf Softw Technol*. 2013;55(2):189-200.
6. Panach JI et al. In search of evidence for Model-Driven development claims: an experiment on quality,effort, productivity and satisfaction. *Inf Softw Technol*. 2015;62:164-186.
7. Jedlitschka, A. et al. Reporting experiments in software engineering. In: Guide to Advanced Empirical Software Engineering. 2008:201–228.
8. Ihme T, Pikkarainen M, Teppola S, Kääriäinen J, Biot O. Challenges and industry practices for managing software variability in small and medium sized enterprises. *Empir Softw Eng*. 2014;1144-1168.
9. Gethers, M. et al. An adaptive approach to impact analysis from change requests to source code. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). 2011:540–543.
10. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *Softw Eng IEEE Trans*. 1994;20(6):476-493.
11. Dzidek WJ et al. A realistic empirical evaluation of the costs and benefits of UML in software maintenance. *IEEE Trans Softw Eng*. 2008;34(3):407-432.
12. Curtis, B. et al. Estimating the size, cost, and types of technical debt. In: 2012 3rd International Workshop on Managing Technical Debt, MTD 2012 – Proceedings. 2012: 49–53.
13. Seaman, C., Guo, Y. A portfolio approach to technical debt management. In: Proceedings - International Conference on Software Engineering. 2011: 31–34.

14. Application quality benchmarking repository - Appmarq - CAST, http://www.castsoftware.com/products/appmarq.

15. FindBugs™ - Find Bugs in Java Programs, http://findbugs.sourceforge.net/.

16. HP Static Analysis, Static Application Security Testing, SAST, http://goo.gl/qtRf0X.

17. SonarQube™, http://www.sonarqube.org/.

18. Nord, R.L. et al. In search of a metric for managing architectural technical debt. In: Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012. 2012: 91–100.

19. Morgenthaler, J.D. et al. Searching for build debt: experiences managing technical debt at Google. In: 2012 3rd International Workshop on Managing Technical Debt, MTD 2012 – Proceedings. 2012: 1–6.

20. Weber, J.H. et al. Managing technical debt in database schemas of critical software. In: 2014 Sixth International Workshop on Managing Technical Debt. 2014: 43–46 IEEE.

21. Zazworka N et al. Comparing four approaches for technical debt identification. *Softw Qual J*. 2014;22(3):403-426.

22. Svahnberg M et al. A taxonomy of variability realization techniques. *Softw - Pract Exp*. 2005;35:705-754.

23. Ko A et al. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Trans Softw Eng*. 2006;32(12):971-987.

24. Hemel Z et al. Static consistency checking of web applications with WebDSL. *J Symb Comput*. 2011;46(2):150-182.

25. Rossi G et al. *Web Engineering: Modelling and Implementing Web Applications*. London, UK: Springer-Verlag; 2008.

26. Gaedke, M., Gräf, G.: Development and evolution of web-applications using the WebComposition Process Model. In: Web Engineering. Newton, MA, USA 2001: 58–76.

27. Cerny T et al. On distributed concern delivery in user interface design. *Comput Sci Inf Syst*. Newton, MA, USA 2015;12(2):655-681.

28. Fowler M et al. *Refactoring: Improving the Design of Existing Code*. Boston, USA: Addison-Wesley Professional; 1999.

29. Gamma, E. et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, USA (1995).

30. Sjøberg DIK et al. A survey of controlled experiments in software engineering. *IEEE Trans Softw Eng*. 2005;31(9):733-753.

31. Wohlin C et al. *Experimentation in Software Engineering: An Introduction*. Netherlands: Kluwer Academic Publishers Norwell; 2000.

32. Ross SM. *Introduction to Probability and Statistics for Engineers and Scientists*. Cambridge, MA, USA: Academic Press; 2004.

33. Buckley J et al. Towards a taxonomy of software change: research articles. *J Softw Maint Evol*. 2005;17(5):309-332.

34. Basili VR et al. Building knowledge through families of experiments. *IEEE Trans Softw Eng*. 1999;25(4):456-473.

35. Höst M et al. Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empir Softw Eng*. 2000;5(3):201-214.

36. Tichy WF. Hints for reviewing empirical work in software engineering. *Empir Softw Eng*. 2000;5(4):309-312.

37. Whittle J et al. The state of practice in Model-Driven engineering. *IEEE Softw*. 2014;31(3):79-85.

38. Salman, I. et al. Are students representatives of professionals in software engineering experiments? In: Proceedings - International Conference on Software Engineering. 2015: 666–676.

39. Groovy/Grails Tool Suite™ (GGTS), https://spring.io/tools/ggts.

40. Aragón, G. et al. An analysis of Model-Driven web engineering methodologies. 2013.

41. Calculation of the functional size and productivity with the IFPUG method (CPM 4. 3. 1). The DDway experience with WebRatio. 1-16.

42. Cerny T, Donahoo MJ. On separation of platform-independent particles in user interfaces. *Cluster Comput*. 2015;18(3):1215-1228.

43. Olson JE. *Data Quality: The Accuracy Dimension*. Burlington, MA, USA: Morgan Kaufmann; 2003.

44. Redman TC. *Data Quality: The Field Guide*. Newton, MA, USA: Digital Press; 2001.

45. Olson JE. *Database Archiving: How to Keep Lots of Data for a Very Long Time*. Burlington, MA, USA: Morgan Kaufmann; 2010.

46. Kennard R, Leaney J. Towards a general purpose architecture for UI generation. *J Syst Softw*. 2010;83(10):1896-1906.

47. Martinez, Y. et al. Evaluating the impact of a Model-Driven web engineering approach on the productivity and the satisfaction of software development teams. In: Web Engineering - 12th International Conference, {ICWE} 2012, Berlin, Germany, July 23-27, 2012. Proceedings. 2012: 223–237.

48. Brambilla M, Fraternali P. *Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML.* Morgan Kaufmann; 2014.

49. Interaction Flow Modeling Language, http://www.omg.org/spec/IFML/.

How to cite this article: Urbieta M, Frajberg D, Rossi G. Assessing the impact of Volatile Functionality removal in web applications: Model-Driven vs Code-Based approaches. *Softw Pract Exper*. 2017;1–33. https://doi.org/10.1002/spe.2503

APPENDIX A

MATERIALS

Next, we present the used questionnaires.

Expertise Code-Based developers' questionnaire

The first questionnaire is targeted to developers of the Code-Based experiment for gathering information such as their experience, training, and formation. Next, we present the questionnaire's questions, which do not need much description:

- How old are you?
- How many months have you been working in the Software Industry?
- In such case, what was your role in the Software project?
- Have you ever used a Web application? **Yes/No**
- Do you actually know what difference exists between a Desktop and a Web application? **Yes/No**
- Do you know the Model-View-Controller pattern? **Yes/No**
- Have you ever programmed in one of these languages? **PHP, C#.NET, ASP.NET, JAVA, PYTHON, RUBY, Other**.
- Have you ever developed a Web application using one of these frameworks? **Symphony (PHP), Struts 1/2, Tapestry, Django, Ruby on rails, Grails, other.**
- If the previous answer was "Yes", how long have you been doing it?
- Do you know any of Object-Oriented programming patterns (aka, GOF patterns)? **Yes/No**
- Do you know any of Enterprise Patterns? **Yes/No**
- How long did it take you to remove changes?

Expertise Model-Driven developers' questionnaire

This questionnaire aims at measuring the confidence of the Model-Driven experiment participants with Webratio and comprises the next question:

- How old are you?
- How many months have you been working in the Software Industry?
- In such case, what was your role in the Software project?
- Describe your expertise with WebRatio:

  I do not know what WebRatio is.
  I just start learning WebRatio.
  I am able to design basic Web applications but I am not able to run them.
  I am able to design and run basic Web applications.
  I am able to design and run complex Web application with WebRatio.
  I am able to design complex Web applications and also extend WebRatio with new components.

Final developers' questionnaire

In this questionnaire, we measure quantitatively the time spent on tasks that are performed to remove Volatile Functionalities. The time and effort required cannot be inferred from source code analysis and implies a detriment in the software development process, thus compromising the Software budget. The questionnaire's questions are

- How did you detect where changes should be done?
- How long did you spend to determine where to perform Volatile Functionality[§§] removal changes (Components, classes, resources, etc)?
- As long as you were modifying source code, did you introduce errors that needed to be fixed?
- How many testing cycles did you perform until a first Maintenance Version was available?
- How long did removing Volatile Functionality actually take?
- Do you feel the application is reliable?
- Do you think it needs more time on testing?

This initial questionnaire was not fixed and we also asked about any outstanding situation that deserved its study.

Model-Driven Assignment

In order to increase sales on days closer to Mother's day, the ACME site has decided to put a special promotion on furniture online. Therefore, the developer team has conducted several temporal changes that aim at adapting the website to such special event.

The Mother's day promotion involves the following functional changes:

- Banners were included in several web pages for highlighting the promotion's existence.
- A discount was added to specific products. When they are presented to the customer, their regular prices, discounts, and final prices are shown.
- An index of products included in the Mother's day promotion. This page allows only showing products “in promotion”.

Once the Mother's day has passed, these changes, which allowed improving sales, are no longer required and are deprecated.

Additionally, the developer team took advantage of the site's high access-rate period to include beta functionality in order to assess its adoption. The beta functionality to be tested allowed customer leaving a comment on published products. Only logged-in customers can leave such comments.

You have been hired by the ACME Company to be part of the maintenance team in charge of ACME's web site. Your first task is to remove both the Mother's day promotion, as well as the beta-tested product comments feature.

[§§]Volatile Functionality concept was introduced to students when describing the set of requirement that should be removed.

APPENDIX B

Strategies for identifying changes		Required time for maintenance		Requires more testing		Complete database information loss		Volatile functionality data loss		Useless data generation		Unreferenced domain model variables		Unreferenced domain model classes		Non-executable behavior		Improper behavior		Unreferenced user interface resources		Unused user interface artifacts		Pending business layer changes		Pending navigational layer changes		Pending user interface layer changes		Pending perceivable changes	
CB	MDD	CB	MDD	CB	MDD	CB	MDD	CB	MDD	CB	MDD	CB	MDD	CB	MMD	CB	MDD	CB	MDD	CB	MDD	CB	MDD	CB	MDD	CB	MDD	CB	MDD	CB	MDD
T	G	1	1	Y	Y	N	N	Y	N	N	Y	N	Y	N	N	N	N	N	Y	N	N	Y	N	N	Y	N	Y	Y	N	N	Y
T	T	4	1,5	N	Y	Y	N	Y	N	N	Y	N	N	N	N	N	N	N	N	N	Y	Y	Y	N	N	N	N	Y	N	N	N
G	S	0,5	1	N	N	N	N	Y	N	N	Y	N	Y	N	Y	N	N	N	Y	N	Y	Y	Y	N	Y	N	Y	Y	N	N	Y
T	S	1	0,25	N	Y	N	N	Y	N	N	Y	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	Y	N	Y	N
S	T	1	0,25	Y	N	N	N	N	N	Y	Y	Y	Y	N	N	N	N	N	Y	N	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y
G	T	0,5	1,5	N	Y	N	N	N	N	Y	Y	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	Y	Y	N
T	T	0,25	0,25	N	Y	Y	N	Y	N	Y	Y	N	Y	N	N	N	N	N	N	N	N	Y	N	N	Y	N	N	Y	N	N	N
T	G	2	0,5	Y	N	N	N	N	N	Y	Y	Y	N	Y	N	Y	N	Y	N	Y	Y	Y	Y	Y	N	Y	N	Y	Y	Y	N
G	T	7	1	Y	N	Y	N	Y	N	N	Y	N	N	Y	N	Y	N	Y	N	N	Y	Y	Y	Y	N	N	N	Y	N	N	N
S	T	7,5	1	N	Y	N	N	N	N	Y	Y	Y	Y	N	Y	N	N	N	Y	N	Y	N	Y	Y	Y	N	Y	Y	N	N	Y
T	T	8	1	N	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N	N	Y	N	Y	Y	Y	Y	Y	N	Y	Y	Y	N	Y
G	S	6	1	N	Y	N	N	N	N	Y	Y	Y	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y	Y	Y	N	N	Y	Y	N	N
T	G	3	1	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	Y	Y	Y	Y	Y	N	Y	N	Y	Y	N	Y	Y	Y	Y	Y
T	S	6	2,5	Y	Y	N	N	N	N	Y	Y	N	Y	N	N	N	N	N	Y	N	Y	Y	Y	N	Y	N	Y	Y	Y	N	Y
T	T	4	0,5	N	Y	N	N	N	Y	Y	N	Y	N	Y	N	N	Y	Y	N	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	N
S	T	3,5	0,5	N	Y	N	N	N	N	Y	Y	Y	N	N	N	N	Y	Y	N	N	Y	Y	Y	Y	N	Y	Y	Y	N	Y	N
S	T	3	1	Y	N	N	N	N	N	N	Y	Y	N	Y	N	N	N	Y	N	Y	0	Y	0	Y	N	Y	N	Y	Y	Y	N
	T		1		N		N		Y		N		N		N		N		N		0		0		N		N		Y		N
	T		1,5		N		N		Y		N		Y		N		N		Y		0		0		Y		Y		Y		Y
	T		8		N																										
1		0,004		0,33		0,095		0,463		0,255		0,749		0,255		0,684		1		0,225		0,085		0,516		0,192		0,0033		1	
P-Value																															

APPENDIX C

SONAR RULES

Next, we outline by category Sonar's rules (see a previous study[¶¶] for a detailed description of metrics) used in the Code-Based experiment analysis.

Documentation

comment_lines:	Number of lines containing either comment or commented-out code.
comment_lines_density:	Density of comment lines.

Issues

violations:	Number of issues.
blocker_violations:	Number of issues that might make the whole application unstable in production.
critical_violations:	Number of issues that might lead to an unexpected behavior in production without impacting the integrity of the whole application.
major_violations:	Number of issues that might have a substantial impact on *productivity*.
minor_violations:	Number of issues that might have a potential and minor impact on *productivity*.
info_violations:	Number of issues that have unknown or not yet well defined security risk or impact on productivity.

Sqale

sqale_debt_ratio:	Ratio between the cost to develop the software and the cost to fix it.
sqale_index:	Minutes of effort to fix all maintainability issues.
sqale_rating:	Rating given to your project related to the value of your Technical Debt Ratio.

Complexity

complexity:	Complexity calculated based on the number of paths through the code.
class_complexity:	Average complexity by class.
file_complexity:	Average complexity by file.
function_complexity:	Average complexity by function.

Duplications

duplicated_blocks;	Number of duplicated blocks of lines.
duplicated_files:	Number of files involved in duplications.
duplicated_lines:	Number of lines involved in duplications.
duplicated_lines_density:	Density of duplication.

Size

classes:	Number of classes.
directories:	Number of directories.
files:	Number of files.
functions:	Number of functions.
lines:	Number of physical lines (number of carriage returns).
ncloc:	Number of physical lines that contain at least one character which is neither a whitespace or a tabulation or part of a comment.

[¶¶]http://docs.sonarqube.org/display/SONAR/Metric+Definitions