MoFQA: A TDD Process and Toolset for Automatic Test Case Generation from MDD Models

Linda Riquelme
Universidad Católica "Nuestra Señora de la Asunción"
Departamento de Electrónica e Informática
Asunción, Paraguay
linda.riquelme@uc.edu.py

and

Magalí González
Universidad Católica "Nuestra Señora de la Asunción"
Departamento de Electrónica e Informática
Asunción, Paraguay
mgonzalez@uc.edu.py

and

Nathalie Aquino
Universidad Católica "Nuestra Señora de la Asunción"
Departamento de Electrónica e Informática
Asunción, Paraguay
nathalie.aquino@uc.edu.py

and

Luca Cernuzzi
Universidad Católica "Nuestra Señora de la Asunción"
Departamento de Electrónica e Informática
Asunción, Paraguay
lcernuzz@uc.edu.py

Abstract

Techniques for quality assurance have to deal with the complexity of software systems and the high probabilities of errors appearing in any stage of the software life cycle. Software testing is a widely used approach but, due to the costs involved in this process, development teams often debate its applicability in their projects. In the endeavor to reduce the complexity of this process, this study presents an approach for software development based on Test-Driven Development (TDD) and supported by Model-Based Testing (MBT) tools that allow the automatic generation of test-cases. The approach, called MoFQA (Model-First Quality Assurance), consists of two main aspects: i) a process based on testing techniques, which drives software development defining steps and recommended practices; and ii) a toolset for testers, end-users and stakeholders, which allow them to model system requirements that represent unit and abstract tests for the system and, ultimately, generate executable tests. The tools that MoFQA provides are applicable to web applications. In order to evaluate the usability of MoFQA tools, two preliminary validation experiences were driven and the results are presented.

Keywords: Software testing, MBT, TDD, Acceptance tests, Web testing, Selenium, TestNG.

1 Introduction

A commonly used technique for Quality Assurance (QA) is software testing. However, the difficulties and limitations that appear when applying it have great impact on costs. Therefore, testers need to find better test sets (i.e., less tests that find more errors).

We aim to promote software testing by inserting it into the software development process in a more natural way, attempting to reduce the time and effort needed to apply it. In this endeavor, we propose to develop software by applying Model-Based Testing (MBT) (1) together with Test-Driven Development (TDD) (2).

MBT is an approach for software testing in which abstract models represent the software system to be tested. These abstract models are used to automatically (or semi-automatically) generate testing code. MBT has been widely adopted and many tools support its use (3).

MBT could reduce the time required to generate, execute and validate test cases and, at the same time, could help to improve the testing process providing tests based on software requirements and constraints as well as increasing the number of tests. Schulze et al. (4) show the results of a comparative study about the effectivity (i.e., the more errors detected, the more effective) and efficiency (in terms of the effort required) of testing using manual tests and MBT. Results show that using MBT it was possible to detect more critical errors with less effort.

On the other hand, TDD is a software development process in which software products are derived from tests: for each software unit to be developed, a set of tests is defined first. This is a common practice in agile software development and allows the construction of software systems with more associated tests.

However, both MBT and TDD have their own limitations. MBT tools, for example, are still far from getting mature enough to allow all the steps of software testing (i.e., design, generation, and execution of tests). Besides, although the tests are represented by abstract models, these abstractions are still hard and tedious to be created by testers (3).

On the other hand, the test design phase in TDD mainly consists of a manual process: test scripts are written based on test frameworks and tools. The tests that result from this process end up being so long that they easily have the same (or even more) lines of code than the tested code itself[1]. If test generation is supported by techniques and tools that shorten and automate the design and execution steps, TDD could be integrated more easily into the software development process.

In this work, we propose the following conjecture: "A software development process based on TDD and supported by MBT for the generation of test cases, could improve team's productivity in terms of the number of resulting tests and the time required to create them. This, in turn, could result on improvements in the quality of software". Our conjecture is based on evidence found in the literature. When using TDD in a rigorous way, each new feature to be developed is guided by a set of previously-defined tests. This allows early error detection and reduces costs related to error corrections **(author?)** (5). Besides, MBT models supported by efficient generation tools could reduce complexity in code generation for different platforms(6). Finally, TDD supported by MBT to generate testing code could give more clarity about the system's requirements and the expected results (7).

Models offer abstraction in the representation of the system's behavior, hiding details related to implementation and target platforms. Model-to-text generation tools could allow automatic generation of tests from those models. The addition of TDD to the approach could facilitate the use of testing techniques, which would allow less error-prone code. Furthermore, we think that the end-user is a key participant who needs more attention in a software development process. His/her participation in the definition and verification phases is crucial in order to guide the development according to his/her needs. Therefore, our aim is to define a software development process based on the generation of tests from abstract models. These abstract models are defined by end-users and developers following TDD steps and practices. The approach, named MoFQA, could be applied in projects in which agile methodologies are used.

As we mention later, one big limitation that still constrains the use of MBT is the lack of easy-to-use and fully-integrated MBT tools. Therefore, we have built testing tools that support our proposed software development process. They can be used to generate tests and automatically execute them in an integrated development environment. The provided tools allow the design and generation of unit and acceptance tests for Web applications, as these applications are still the most popular (8). We also built a special tool, aimed at end-users so they can fully collaborate in the design and testing phases of software development. This last tool is expected to prioritize and facilitate the definition of tests based on the expected results for the end-users, expressed as mockups of the UI (User Interface) elements, as proposed by the approach in (9).

It is worth noting that this work expands a previous one presented in CLEI 2018 (10). The current version extends the previous in different aspects: i) the related work section was updated with new approaches that

[1] https://ianhammondcooper.wordpress.com/2007/03/20/ratio-of-test-code-to-production-code/

combine MBT and TDD, analyzing contributions of MoFQA with regard to those approaches; ii) a general overview of the MoFQA UML profile was introduced, to better understand the relationships among all its internal components; iii) a description of the testing framework and how the different specification elements are translated to the test code to be generated was also introduced, to better understand what kind of code and tests MoFQA generates; and, iv) the results of the first validation experience are analyzed from a different perspective: the comparisons are made by using the results obtained from all students and not only from the more-complete works. This also allowed us to see the difficulties that non-technical people present when defining and executing manual tests.

The next sections of this paper are organized as follows. Section 2 discusses related work. Section 3 presents the MoFQA approach, its process and tools. In Section 4, two validation experiences are presented. Finally, section 5 presents conclusions and suggestions for future work.

2 Related Work

MBTDD (11) is a framework based on the combination of MBT and TDD with the aim of achieving higher levels of abstraction and improving the efficiency of software development. MBTDD defines meta-models that allow the definition of higher levels of software, such as architecture. To evaluate their framework, the authors conducted a case study consisting in the development of an enterprise web application based on a 3-tier architecture (presentation, business logic and persistence levels). Finally, they identified some benefits applying the proposed framework in an industrial environment: (i) more participation of non-technical persons in the TDD process; (ii) traceability of requirements and tests; (iii) automation of test generation; (iv) modeling allows platform independency for tests. Also, they achieved improved code quality and, when maintenance time is included in the efficiency calculation, the increment of efficiency is also predictable. The lack of proper tools, however, is still a limitation for applying the framework. For the modeling phase of their case study, they used the tool FitNesse[2].

MoFQA aims to exploit the observed advantages of the use of TDD together with MBT to drive software development, by supporting limitations found in MBT tools. While looking for existing tools, we found the work of Shappee (12). This work proposes the use of MBT tools following the TDD steps. As well as MBTDD, the tools presented in this work are text-based. This means that constraints and requirements are expressed using a formal language or notation. In this sense, we thought that the active inclusion of end-users in the software development cycle would need the existence of user-friendly tools to define tests in an intuitive and easy-to-understand notation.

The need to reduce the complexity in the use of MBT is also presented in (13). This work mentions that testers need to be capable of knowing modeling languages, defining test coverage criteria, managing the generated output, among other skills. Besides, Peleska (14) mentions that test modeling has several challenges: (i) some details in complex systems need to be abstracted, in order to prevent unmanageability of models; (ii) the skills needed to model tests may be greater than those needed to write testing code. In order to minimize the existing complexity for the application of current MBT techniques, it would be useful to decrease the amount of knowledge necessary for its implementation. These complexities of applying MBT are still mentioned in more recent studies, such as the one from Villalobos-Arias et al. (15).

As can be seen in the work of Rivero et al. (9), the use of artifacts like mockups can improve the requirements gathering in comparison with textual methods and also can facilitate communication among end-users and developers. In their approach, they define an agile methodology for the use of mockups for the definition and generation of a web application, with the active participation of end-users. MoFQA is similar to this approach as the MoFQA Modeler enables end-users to define mockups of the web systems that can later be transformed into executable test cases.

On the other hand, in order to produce a complete software product, tested using an MBT approach, several tools are needed to carry out each of the following processes: test modeling, test generation from models, software development, test execution and test validation. In most observed settings, different tools need to be used to complete all phases, but tool integration is not a simple task. Therefore, Dias Neto et al. (13) highlight the lack of integration between MBT and the software development processes. This is caused by the nonexistence of fully integrated tools that support all the steps of software development and, at the same time, MBT steps to allow testing. The study presented in (15) stands out this limitation pointing out that most MBT tools provide support for the model specification phase, while the other phases of MBT are less covered by tools.

As seen in (12), existing MBT implementations do not take full advantage of TDD. In effect, most of the tools do not reflect the test-first nature of TDD. Instead, tests are generated for an already existing software (or part of it). Besides, in some works like (12), the code for tests and the code of the system under test

[2]http://fitnesse.org/

itself are both derived from the same models and at the same time. We are interested, in contrast, in making both processes (generation of the final system and generation of tests) independent, in order to grant more flexibility to the software development process and to avoid error replications. This way, possible errors in the system under test are not spread to tests (16). For this reason, the reuse of development models for testing activities is not recommended.

Finally, while other authors mention that in some cases, the reuse of existing models can benefit the verification of correctness of the code generation tools (17), we consider that approaches that apply reverse engineering (by generating testing models from the system's code) have the same issues. If tests are defined from the already-built code, some errors are unable to be found (e.g., errors due to miss-understandings in functional requirements or incomplete functionalities).

3 MoFQA: An Approach for Software Development

Our approach, called **MoFQA** (Model-First Quality Assurance), consists of two main aspects:

1. A process based on testing techniques to drive software development, which defines steps and recommended practices.
2. A toolset that allows the proposed approach to be applied. The toolset consists of: (i) a tool for end-users and stakeholders to model system requirements; (ii) UML profiles to support the creation of models that represent unit and abstract tests; (iii) transformation rules to generate executable tests from abstract tests.

While both components of our approach are presented as a whole, they can be used independently: the use of MoFQA tools is not required in order to follow the proposed development process, and vice-versa. The MoFQA process for software development could even be applied to the development of software for platforms different than the web (our tools are targeted to the development of web applications), as long as agile methodologies are applicable.

The following sections describe the components of MoFQA in detail.

3.1 MoFQA: a Process for Software Development

The creation of a new feature in an agile development environment starts with the definition of a user story. From that point, MoFQA (as seen in Figure 1) recommends a test-first approach based on TDD to be followed. Tests are generated from models. These models are defined by end-users of the system under development and developers, collaboratively.

As depicted in Figure 1, both end-users and developers work collaboratively in order to define the new feature (including domain elements of this part of the system), the tests based on the end-users' acceptance criteria and the developers' technical constraints and requirements. For the development of the new feature, a TDD approach is followed, so the red, green and blue arrows in the figure highlight the development path of MoFQA that was inspired by TDD. The red phase represents the first stage of TDD when the tests do not pass. Later comes the green phase, where all tests pass. Finally, a refactor phase can take place in TDD, which is marked with blue arrows in the figure.

End-users, with some help from developers, define the domain elements of the software under development, adding each element on demand. We use the term "domain" to refer to the main elements (entities and relationships) that describe the data and business logic of the software to-be. MBT tools can help to define these domain elements, with the advantage that they could be reused by subsequent models.

End-users also define acceptance criteria that every new feature must accomplish. These criteria will determine the approval or rejection of the product when being reviewed. In MoFQA, those criteria should be based on the previously-defined domain elements. The acceptance criteria are also modeled with the support of MBT tools, and the resulting models consist of the abstract acceptance tests for the feature. Then, with the help of transformation tools, the developer can automatically generate executable acceptance tests, starting the software development process based on TDD.

If the newly-created acceptance tests pass at first, it could be either that the functionality already exists or that the tests are incorrectly designed and must be verified. On the other hand, if the tests fail, the developer enriches the models in order to define tests that will validate the units of code to be written for the current user story. Models that add integration tests could be also considered in this step. The next step is to automatically generate the whole set of tests modeled so far. These tests will guide the TDD process to be followed by the developer in order to write and test the implementation code.

MoFQA aims to guide the development of software features incrementally and, by following a TDD approach, seeks to support each user story with at least one test that verifies their acceptance criteria. The

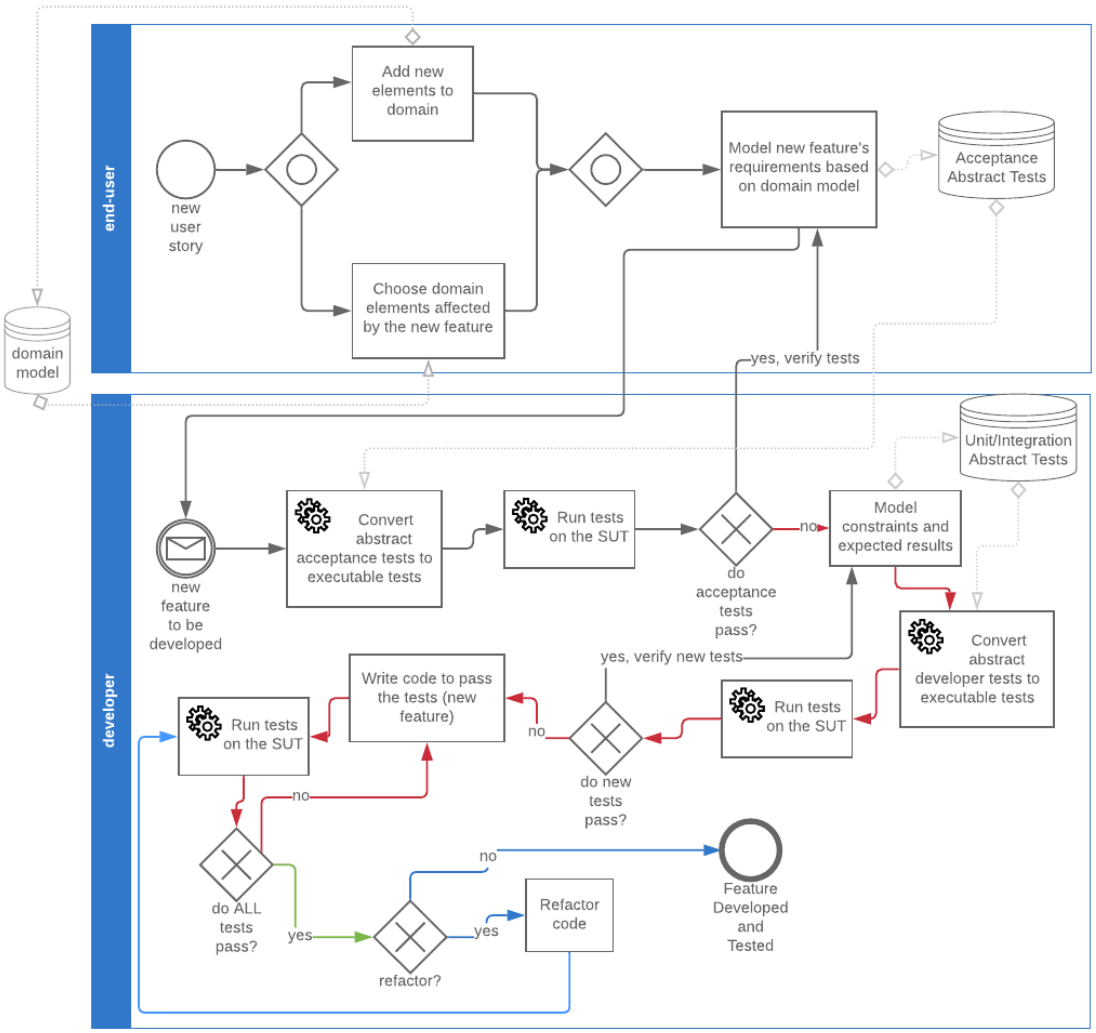

Figure 1: Software development process following the MoFQA approach (in the figure, SUT stands for System Under Test).

use of MBT techniques and tools is proposed in order to ease the design and maintenance of tests while allowing the automatic generation of executable tests from abstract models that represent each requirement of the system. These models also offer documentation about the behavior of each module that comprises the system. Changes in the system's requirements will only require models to be updated. This update would allow the automatic generation of executable tests according to the changes in requirements.

Our proposal welcomes the active participation of stakeholders in the domain definition of the system, the modeling of acceptance criteria for each user story and the validation of implemented requirements by executing the generated acceptance tests.

Only one set of models is used to generate all test types and they are written based on domain elements. Besides, MoFQA forces the system's implementation to be also based on these domain elements (otherwise, some tests will fail). This criterion was defined in order to boost the use of DDD (Domain-Driven Design) [3] principles, seeking to improve communication among end-users and developers, as well as the transmission of knowledge about the domain and business rules for the system-to-be.

3.2 MoFQA Tools

In order to give tool support to the presented approach for software development, MoFQA offers an MBT toolset for the definition, generation and execution of unit and acceptance tests for web systems. The

[3]DDD (Domain-Driven Design) (18) consists of a series of patterns for the construction of software products, with special concern in its domain and business logic, based on the definition of a common model that is built with continuos collaboration among developers and domain experts.

provided tools can be used for the definition of domain elements, to model and generate acceptance and unit tests, as well as, to allow the execution of the generated tests.

One of the issues that we mention in the related work section is the lack of integration among MBT tools to complete both the development and testing of software. This need drove us to build fully-integrated tools to help the development and testing of web systems. These tools only cover the definition and generation of unit and acceptance tests. The definition and generation of tests with MoFQA tools consist of the steps shown in Figure 2.

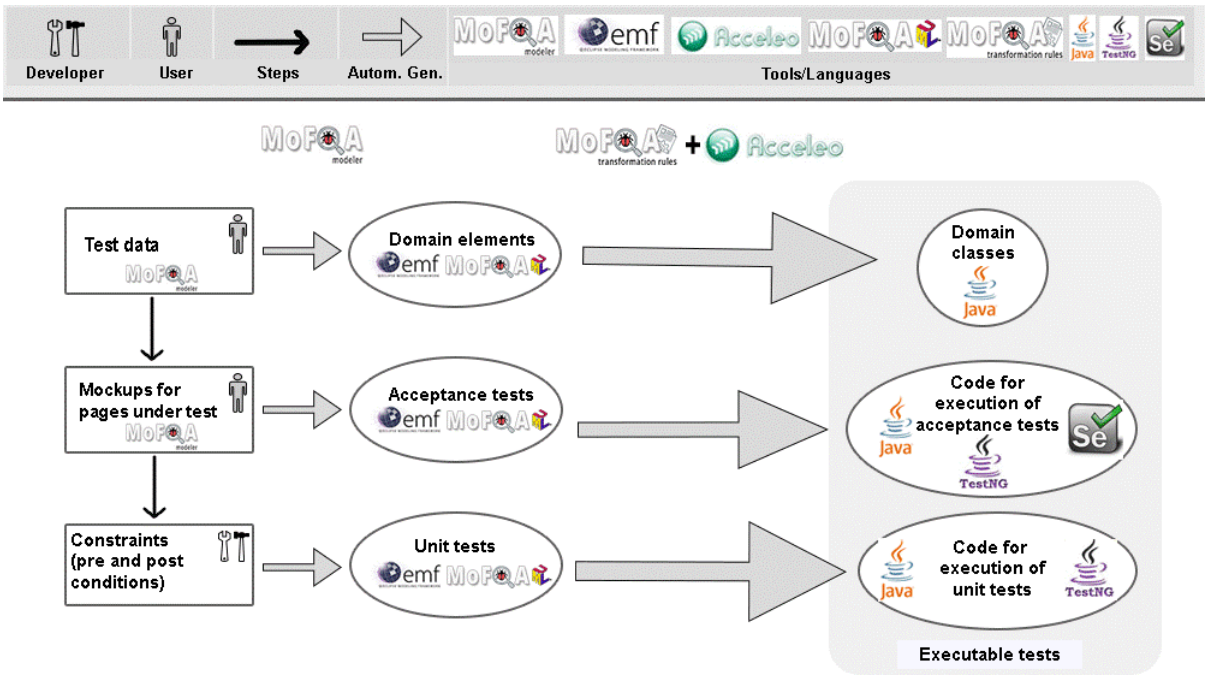

Figure 2: Flow of steps for defining tests with MoFQA tools

The main actors involved in these processes are developers and end-users. End-users define the system's requirements in order to pass the acceptance criteria to be tested. This definition can be done with a tool called **MoFQA Modeler**. In this tool, requirements are drawn by: (i) mockups of the pages that constitute the web system under development and test; (ii) example data to be used for tests. With these elements, the MoFQA Modeler generates models that represent the domain elements of the system under test and the acceptance abstract tests to verify the compliance of acceptance criteria. The generated models are built over a UML profile that MoFQA provides and are expressed in EMF (Eclipse Modeling Framework) format[4]. The developer can later enrich the models generated by the end-user in order to add certain unit tests based on pre-conditions, post-conditions and expected results of the execution of code units. The UML profile provided by MoFQA can be used to model these types of tests.

The generated models can then be imported to an integrated environment (for test generation, execution and system development) in Eclipse[5] where, using Acceleo[6] and the transformation rules provided by MoFQA, executable acceptance and unit tests will be generated. The tests that result from this process are written in Java language[7] and the testing frameworks TestNG[8] and Selenium[9]. The transformation rules defined in MoFQA also generate the domain elements (entities and relationships among them) for the system-to-be, written also in Java.

3.2.1 MoFQA Tools: UML Profiles for the definition of Abstract Tests

MoFQA provides a UML profile to allow abstract tests to be modeled by end-users and developers. The use of this profile is restricted to model web applications.

[4]http://www.eclipse.org/modeling/emf/
[5]https://www.eclipse.org/
[6]https://www.eclipse.org/acceleo/
[7]https://www.java.com/es/
[8]http://testng.org/doc/
[9]http://www.seleniumhq.org/

This profile, called "Acceptance Criteria"[10] is depicted in Figure 3 and consists of four main elements: (i) "Data Provider", which allows the definition of datasets to be used in the execution of tests; (ii) "Domain Specification", which defines the structure of the system's domain elements; (iii) "Content Specification", which allows the inclusion of visual components in tests that consist of elements to be displayed in each web page to be tested; and (iv) "Constraint Specification", where each unit of code to be built by the developer can be linked to a series of pre-conditions and post-conditions required for its execution.

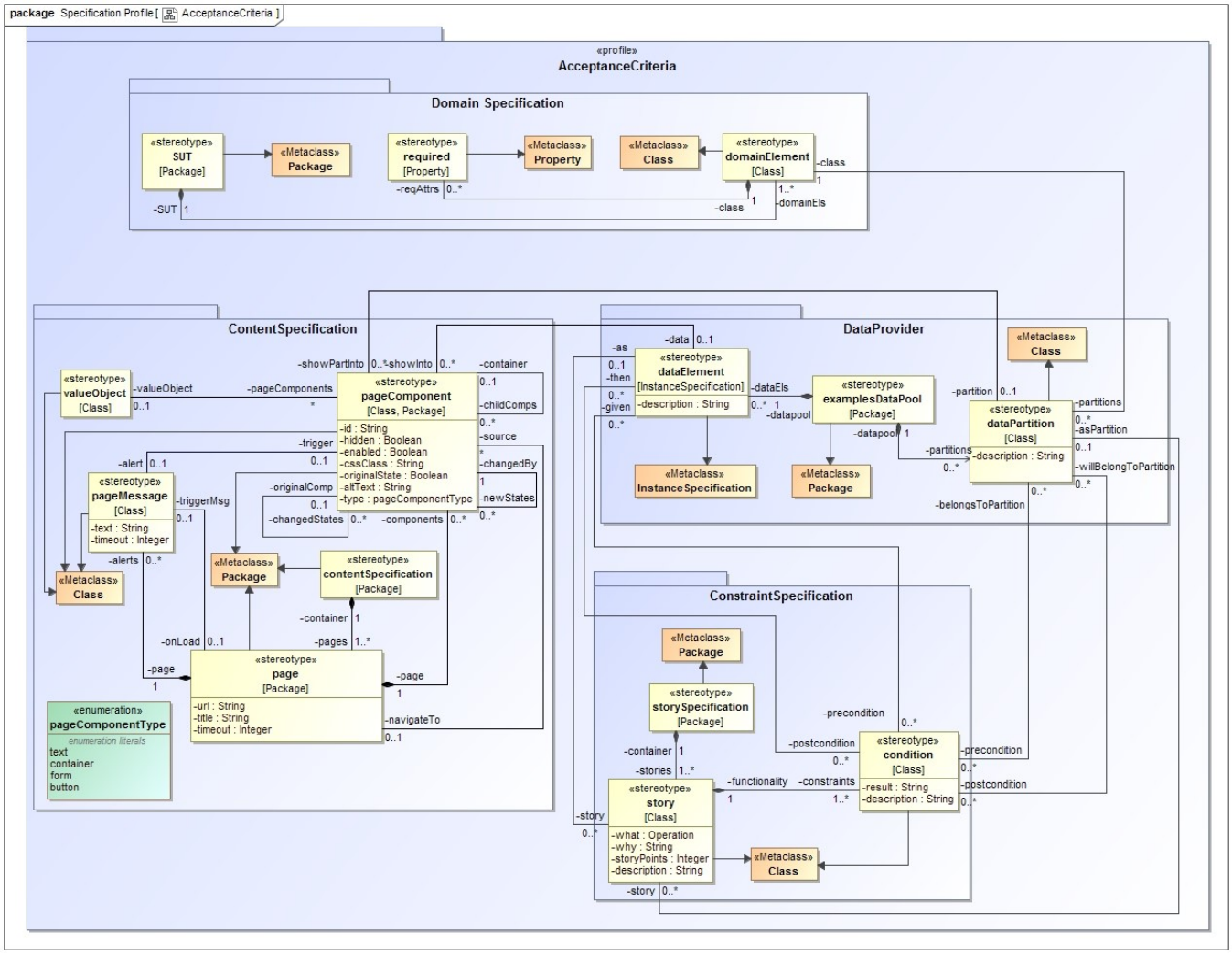

Figure 3: Acceptance Criteria Profile: The global UML Profile defined by MoFQA. It comprises the subsets "Domain Specification", "Content Specification", "Constraint Specification", and "Data Provider"

Figure 3 provides a global glance of the four subsets that conform the "Acceptance Criteria" profile and shows how they are connected. Domain elements, defined by the items in "Domain Specification", can be used to later define data examples for tests[11]. This is represented by the relation between the elements "Domain Specification" and "Data Provider". The relationships among the elements in "Content Specification" and "Data Provider" allow UI (User Interface) elements of the system under test to be tested with existing test data. Finally, the connections among the elements in "Constraint Specification" and "Data Provider" allow the use of test data to express constraints to be met before and after the execution of units of tests.

Figure 4 displays the elements that comprise the **Data Provider** package. The pool of test data is grouped by an «examplesDataPool» package with «dataPartition» classes and «dataElement» instances. Elements stereotyped as «dataElement» are instances of domain classes that have actual test data. On the other hand, elements stereotyped as «dataPartition» are classes that allow the definition of types of entities,

[10] Almost all tests that can be modeled with this profile are acceptance tests. A small subset (called "Constraint Specification") can be used to model abstract unit tests (by defining pre-conditions and post-conditions for the execution of units of code).

[11] The restriction of having example data being derived from domain elements seeks all the entities involved in the business logic to be present in the domain model and these entities to be well known by all actors (development team, end-users and stakeholders).

based on domain entities, by setting the value that each type has for some attributes of the entities. This allows the classification of data and to limit the representation to only the attributes of interest.

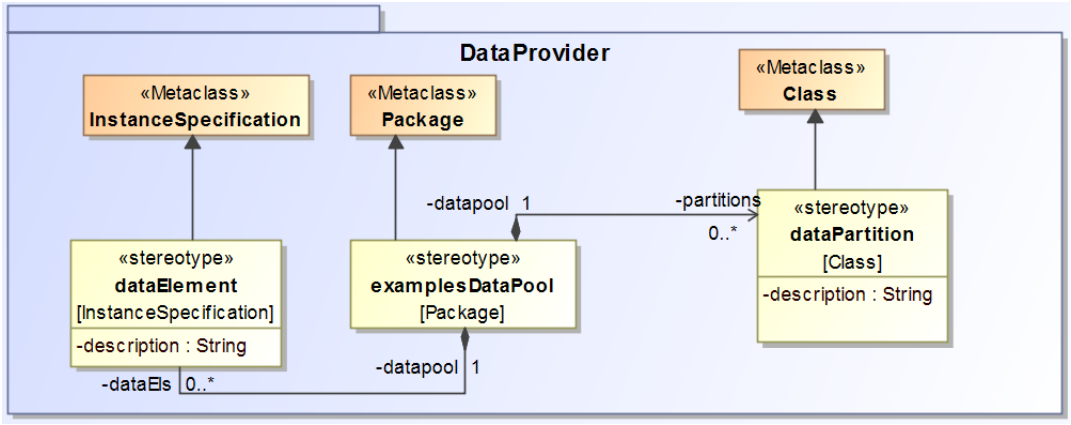

Figure 4: Acceptance Criteria Profile: Elements in group "Data Provider"

The **Domain Specification**, as shown in Figure 5, defines «domainElement» classes that are grouped by «SUT»[12] packages. These classes represent the domain entities of the system under development which can be interconnected by relationships and have required and optional attributes.

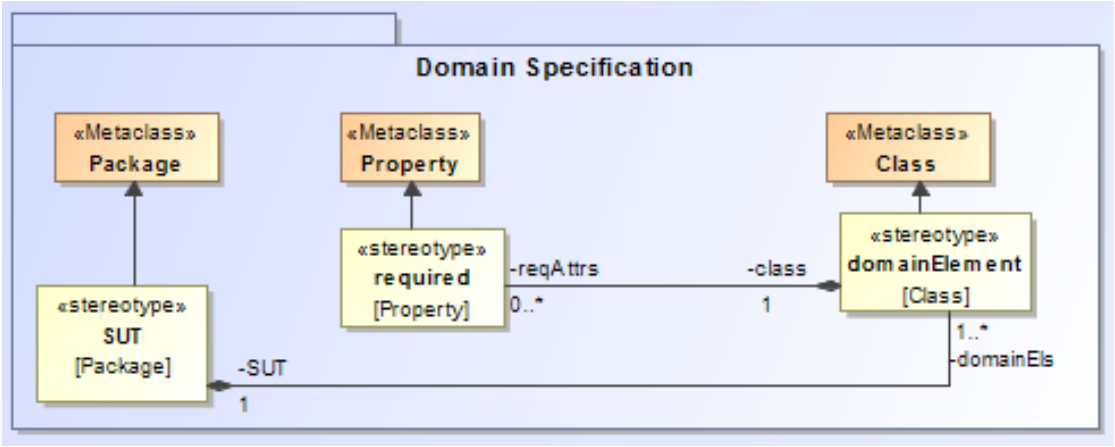

Figure 5: Acceptance Criteria Profile: Elements in group "Domain Specification"

The **Constraint Specification** group (see Figure 6) allows user stories to be recorded in the form of «story» classes with optional constraints (classified as pre-conditions and post-conditions) for their execution. These constraints are classes with the «condition» stereotype. The stories defined are grouped by «storySpecification» packages.

Finally, the elements in **Content Specification** appear in Figure 7 and allow the definition of the UI elements that the web application under test will have. As mentioned before, these elements can be linked to test data in order to simulate the end-users' interaction with the UI and the expected results of these actions. The elements that can be modeled are web pages, stereotyped as «page» and included into «contentSpecification» packages, which can contain «pageComponent» items (i.e. text boxes, HTML elements or containers, forms and buttons) and display alerts («pageMessage» elements) when required.

3.2.2 MoFQA Modeler for the definition of Abstract Acceptance Tests

To support the modeling of the requirements and acceptance criteria of the end-users of the system under development, MoFQA provides an additional tool called **MoFQA Modeler**. It consists of a web application[13] that allows the end-user to model the following elements of the system-to-be:

- Web pages and their components (text and labels, form fields, buttons and alert messages).

[12] SUT: System Under Test.
[13] More details about the tool and user manuals are available at (19) .

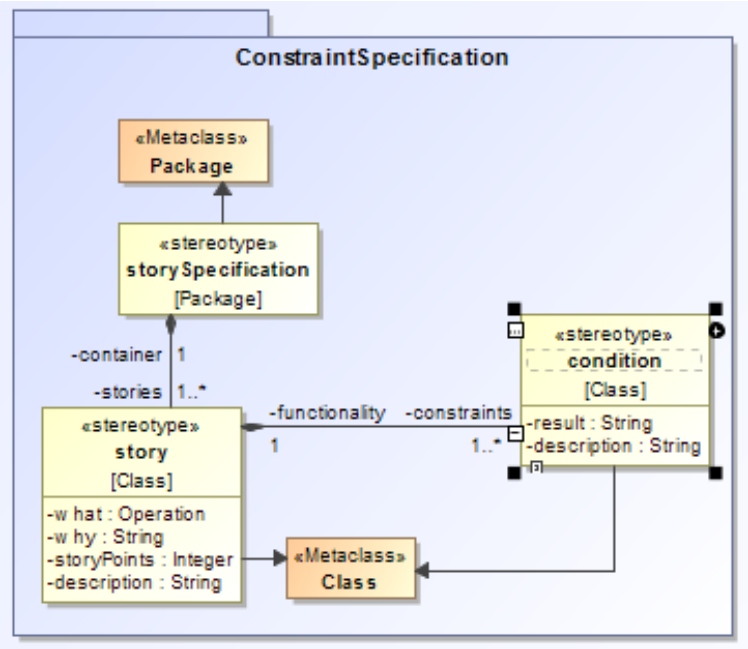

Figure 6: Acceptance Criteria Profile: Elements in group "Constraint Specification"

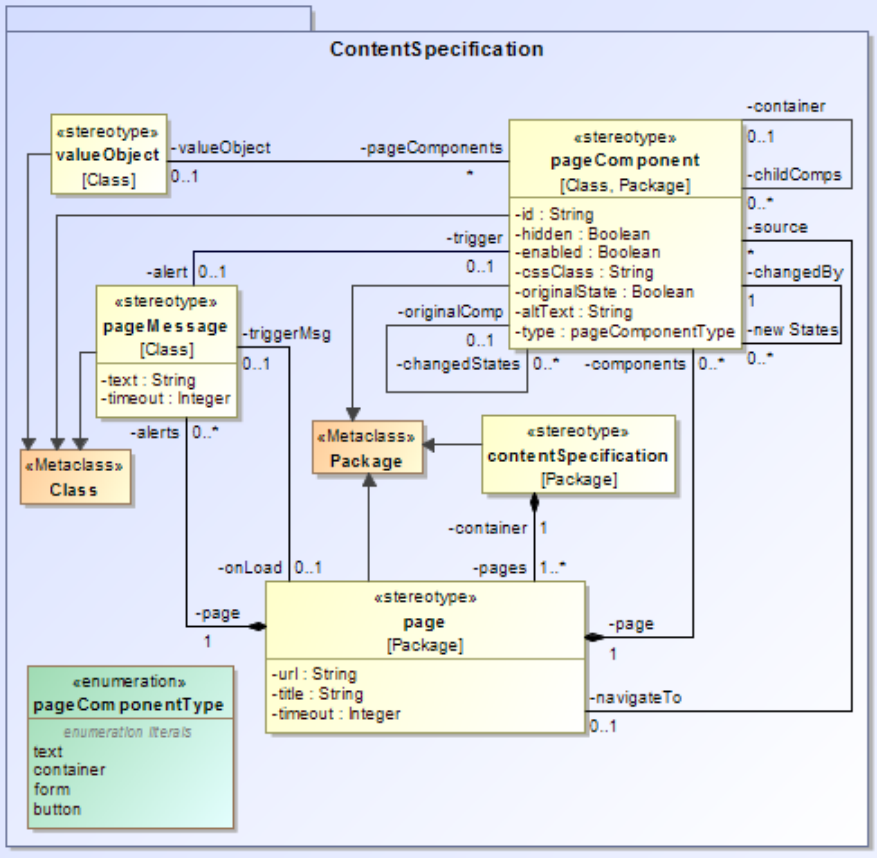

Figure 7: Acceptance Criteria Profile: Elements in group "Content Specification"

- Example data that is expected to be used and displayed when executing the acceptance tests.
- Results of the interactions of end-users with the different types of page components.

The MoFQA Modeler generates EMF models [14] from the elements defined by end-users as mockups of

[14] 5th version of EMF, enriched with the "Acceptance Criteria" UML Profile

the web pages to be tested.

This tool aims to facilitate the modeling tasks by providing more abstraction to the end-user, who would not need to have technical skills to apply MBT techniques. As the MoFQA Modeler automatically generates several model elements, the time needed to manually model the requirements is reduced.

3.2.3 MoFQA Testing Framework and Transformation Rules

The models that both end-user and developer build with the tools provided by MoFQA, represent abstract acceptance and unit tests for the system under development. These tests are transformed into executable tests after running the provided **MoFQA transformation rules** with the tool Acceleo. These rules define how the models will be translated to Java code (with the frameworks TestNG and Selenium to support testing).

The generated acceptance and unit tests can be executed over the system using a completely-integrated development environment in Eclipse. It is worth mentioning that, once this environment is set, all development and testing steps can be performed using only one (centralized) tool as follows:

1. Test design and modeling: by importing UML profiles (in EMF format), models that represent acceptance criteria (for acceptance tests) and conditions in the execution of methods (for unit tests) can be designed. The models generated by the tool MoFQA Modeler may also be imported to this environment, as they are written in EMF format.

2. Automatic test generation: once Acceleo is installed and the transformation tools are set, executable tests can be generated from models designed with the UML Profile defined in MoFQA.

3. Test execution: the generated tests can be executed in this environment once TestNG and Selenium are installed. The execution of an XML file (generated by the MoFQA Transformation Rules) automatically runs all the generated tests.

4. Software development and testing integration: finally, if the system's code is written in Java, testing and development code can be unified into the same Eclipse project. This would allow the reuse of domain classes generated by the transformation rules and that the methods are available to perform unit tests.

The tests provided by MoFQA consist of Java and XML files that are used by the tool TestNG to execute tests. A Java class, called TestManager, is created in each generation of code with MoFQA. This class has the main methods that execute all the intermediate steps of each test. With each transformation also appear the classes AlertMsg, Page, PageComponent and PageStateSetter as well as the interface ValueObject. These classes are instanced and used by the TestManager to orchestrate the test steps. All these generated classes could be reused by the developer in order to write additional tests manually. The following paragraphs describe the model-to-text mapping rules that MoFQA follows in order to generate the executable test code.

As described in Table 1, the elements of the group "Domain Specification" generate Java classes and enumerations, which represent the system's domain elements and the relationships among them. The required attributes are taken into consideration, so methods that verify them and their values (according to the specified types) are included.

Table 1: Mapping among elements in "Domain Specification" and test code

Modeled element	**Code to generate**
«SUT».**«domainElement»**	Java classes with attributes (relationships among entities are considered), a constructor (verifying required attributes), setter and getter methods, *equals* method and additional methods' headers. In an integrated development-testing environment, these classes can be reused to build the development code.
«SUT».**«Enumeration»**	An enumeration into a special Java class reserved to store and group all enumerations.

The elements of the group "Content Specification" generate methods and classes that will derive into acceptance tests. The resulting code defines and initializes instances of the elements to be tested and, helped by the TestManager, a recursive search and verification takes place for the different components from an initial page to the last page, where these components are located. This mapping is described in Table 2.

Table 2: Mapping among elements in "Content Specification" and test code

Modeled element	Code to generate
«valueObject»	Classes that implement the interface ValueObject with the attributes and test methods defined.
«contentSpecification»	A test class that groups all acceptance tests.
«contentSpecification».**«page»**	Acceptance test method.
«contentSpecification».**«pageComponent»**	An instance of PageComponent with attributes (among others): id, visible/not visible, enabled/disabled, element type, CSS (Cascading Style Sheets) class, value object and result after clicking on the element.

The «contentSpecification» elements are transformed to test classes with one test method per each modeled «page» element.

Each page with the attribute URL specified is tested to verify its loading times and success. When loaded, the page's title is compared to the title specified in the model. If an alert message is expected, the test waits until it appears (and timeouts in case it does not appear). Besides, each pageComponent is verified according to the following rules:

- All the modeled components must exist in their corresponding page (the search is made according to the component's ID).
- All components are checked to see if they are visible or not, if they are enabled or disabled, and that their specified CSS classes exist.
- For components of type text or button, their labels are compared to the text indicated in the model.
- An element of type form defines the demo data for the tests to be loaded into an HTML <form>. Each field is directly identified by its name. The current version of MoFQA is limited to test text fields.
- If actions were defined in the model (in the form of changing the states of components or navigating through different pages), each one of them is tested. The tests recursively navigate through all specified components (even if they appear in different pages) and pages.

The elements in "Constraint Specification" generate test methods and classes that will shape the developers' unit tests. As Table 3 shows, the «storySpecification» packages generate tests that create instances of the domain elements and run test methods specified as «story» elements. The elements specified as «condition» in the model generate verifications needed before and after the execution of each method. The resulting tests verify that the pre-conditions are satisfied before running each method. If they pass, the method is executed and the expected results are verified. Finally, the post-conditions are checked to validate the final states of the created instances.

Table 3: Mapping among elements in "Constraint Specification" and test code

Modeled element	Code to generate
«storySpecification»	Test class that groups unit test methods.
«storySpecification».**«story»**	A unit test method.
«storySpecification».**«condition»**	Each story is linked to one or more «condition» elements. Their tagged values generate the preconditions, the method's expected return values and the postconditions.

The elements defined in "Data Provider" are used to generate test data for both acceptance and unit tests, supported by the specified «valueObject» elements.

4 A first validation of the MoFQA tools

In order to carry out a first analysis of the usefulness of the MoFQA tools, we conducted two validation experiences in which we considered:

- Advantages of testing using MoFQA tools over fully manual testing. In this case, both techniques are compared using the following parameters: (i) the amount of time required for the definition and execution of tests; (ii) the number of test steps carried out.
- Advantages of modeling with MoFQA Modeler over modeling using the UML notation enriched with the UML profile defined in this proposal. In this case, we specifically compared the time needed for each modeling technique.
- Perception of usability of the MoFQA Modeler tool, aimed at end-users of the system to be verified.

The following sub-sections describe the two experiences that were carried out and present their results.

4.1 First Experience: Practical experience with students

As a first validation experience, we have carried out a laboratory workshop with Computer Science and Electronic Engineering students. These students were in the initial semesters of college. The objective of this experience was to get a rough idea about the advantages of using the MoFQA Modeler tool for the generation of acceptance tests from user specifications. In this experience, comparisons were made between the techniques of completely manual testing and testing aided by the tools offered by MoFQA.

With this experience we sought to obtain a measure of: (i) the time involved in defining and executing tests; and, (ii) the number of verification's performed on each test. In addition, it was possible to obtain an appreciation of the usability of the tool.

We considered that the profile of the students was adequate, as the tool is aimed at end-users. The involved students comply with the following characteristics: they have no modeling knowledge and have little programming experience, but have affinity and interest in defining the functionalities of software systems. In total, 24 students participated in the entire workshop. In a 90-minute session, the MoFQA Modeler tool was presented and the students were asked to model a requirement for a virtual classroom platform that is used and known by them, called "Claroline"[15]. At the beginning of the session, there was a brief introduction to the concept of acceptance tests, their importance and some techniques to drive them (including manual and automatic tools for the execution of tests). Subsequently, the students were asked to perform an activity consisting in the following steps:

1. Select at least one requirement to be verified in "Claroline', from a predefined list.
2. Decide about the steps to test the selected requirements manually. These steps are later denominated as the phase of "definition of manual tests". The defined manual tests have to be executed by following the steps. The times of definition and execution of these manual tests need to be recorded.
3. Model the selected requirements using MoFQA Modeler and capture the following data: modeling time with the tool, and doubts/complications with the use of the tool.
4. At the end of the experience, we asked the students to complete a SUS[16] questionnaire, in order to obtain an appreciation of the usability of MoFQA Modeler for this group of end-users.

The usability scale presented in (20), establishes that a SUS score of 68 represents approximately a 50% level of usability. We obtained an average SUS value equal to 55.43, which in the same scale represents only a level of 20% of usability. We also ranked the individual SUS values according to the scale of adjectives given by Bangor et al. (21). According to this last scale, 47.83% of respondents scored MoFQA Modeler below "Acceptable", while 52.17% rated the tool as "Acceptable" or better.

Regarding the tests modeled with MoFQA, we compared the manual tests[17] defined by the students and the generated tests from the resulting models. To do this, we identified the test steps followed for each requirement in both the manual tests and the generated tests. We saw that for each defined requirement, the transformation rules allowed the generation of several test steps, with intermediate verifications. In this way, each requirement is accompanied by at least one test with several checks. We were also interested in the amount of time spent defining and executing manual tests with respect to the amount of time spent modeling, generating and executing generated tests.

We saw that each of the 24 students tested at least one of the requirements manually, while only a few of them were able to model the requirements with the tool. Therefore, we obtained 27 manual tests but only 14 modeled tests. Table 4 compares manual tests (MT) and generated tests (GT) per each requirement. It

[15] https://www.claroline.net/

[16] https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html

[17] Before using the tool, the students executed the testing of the selected requirement, in a completely manual way: the definition of the test steps, as well as their execution, was performed manually.

includes the following columns: the number of manual tests defined per each requirement type (MT), the number of modeled tests per requirement type (GT), the average time (in seconds) for the definition and execution of manual tests per requirement type (MT time), the average time (in seconds) for the modeling of tests in MoFQA Modeler per requirement type (GT time), the average number of test verifications made by manual tests (MT steps), the average number of test verifications run by the generated tests (GT steps), the average of lines of code generated for the test scripts per requirement type (GT LoC). The requirements that could not be modeled by the students were marked as N/A (not applicable) in some cells as the comparison between the manual and automated testing techniques cannot be performed.

Table 4: Comparison between manual and modeled tests, with respect to the average time and test steps, for each type of requirement included in the validation experience (MT = manual tests; GT = generated tests; LoC = lines of code)

Req.	**MT**	**GT**	**MT time (s)**	**GT time (s)**	**MT steps**	**GT steps**	**GT LoC**
R1	3	3	140	220	9	22	109
R2	11	8	104.27	258.75	2	6	73
R3	7	3	268.14	600	9	15	341
R4	5	0	N/A	N/A	N/A	N/A	N/A
R5	1	0	N/A	N/A	N/A	N/A	N/A

We observe that the code generated using MoFQA performs a greater number of steps with intermediate verification's that, in addition, are automated for later executions. On average we have eight additional steps generated by the transformation rules, compared to the steps defined by the manual tests. In addition, there is not much difference between the amount of time spent modeling requirements with MoFQA Modeler, compared to the manual definition and execution of tests. Although the difference is not significant, modeling with MoFQA automatically generates test code. These can be executed at no additional cost as many times as necessary, allowing automatic regression tests. All three requirement types could be modeled in an average of 6 minutes, compared to the average time of 2.85 minutes needed for the definition and execution of manual tests for those requirements.

4.2 Second Experience: Modeling time comparison

As a second experience, we defined five requirements to test on the website Amazon.es and generated the corresponding tests. In this experience we compared the times that took modeling these requirements with both approaches: (i) modeling with MoFQA Modeler; versus (ii) modeling with a UML editor (MagicDraw[18]) and the defined UML profile. In addition, we verified that it is possible to carry out the entire testing process using the MoFQA tools in an integrated way.

The experience allowed us to see the time savings involved in modeling acceptance tests (according to the definitions of MoFQA profiles) using MoFQA Modeler, compared to doing it directly using UML profiles and a UML editing tool. We believe that MoFQA Modeler has an advantage because it abstracts several details of the model, which are transparent to the end-user. Its use not only allows end-users without modeling experience to define acceptance tests but also helps more technical users to model those tests in less time. Table 5 compares the elapsed times for modeling tests using MagicDraw and the profiles (case 1) and MoFQA Modeler (case 2). Table 6 compares the total number of lines of generated code[19] (for all requirements) from each model. It can be seen that, for the same requirements to test, MoFQA Modeler generates a greater number of lines of code for most cases.

While it is evident that the modeling time is much shorter in all cases (an average of 12 minutes advantage for MoFQA Modeler in all requirements), it is important to mention that the tool has limitations and it is not possible to generate all the elements available in the MoFQA UML profile. However, for the requirements modeled in this experience, the functionalities offered by MoFQA Modeler were sufficient to generate the required tests.

The difference in the number of lines of generated code in each case is due to the fact that, through the modeling carried out directly with the UML profiles, a greater optimization was achieved, reusing existing model elements. MoFQA Modeler, however, creates a larger number of model elements since the relationships that can be defined between them are only of type 1 to 1 (limitations of the tool in (19)).

[18] https://www.nomagic.com/products/magicdraw

[19] Only the lines of code directly linked to the tests are considered. The MoFQA testing framework is not included in the total count

Table 5: Time (in minutes) to model requirements using: (i) UML Profiles and MagicDraw; (ii) MoFQA Modeler

Req.	UML Profiles + MagicDraw	MoFQA Modeler
1	18 min.	5 min.
2	19 min.	6 min.
3	20 min.	8 min.
4	14 min.	3 min.
5	14 min.	3 min.

Table 6: Number of generated lines of code for each requirement using the models by: (i) UML Profiles and MagicDraw; (ii) MoFQA Modeler

Generated code	UML Profile + MagicDraw	MoFQA Modeler
XML Configuration	29 LoC	27 LoC
Domain Classes	2 (180 LoC)	2 (166 LoC)
Value Objects	3 (196 LoC)	7 (470 LoC)
Test methods	3 (349 LoC)	5 (430 LoC)
Total	754 LoC	1.093 LoC

4.3 Threats to validity

Since this validation was generated in the context of an academic environment,there are threats to validity that need to be considered.

The experience that was driven by the students, had some shortcomings that could have affected the results: (i) the available time for the experience was short; (ii) the students had no previous knowledge about tests and their importance; (iii) during the experience, the Internet connection was lost and some students could not save their work. However, we could collect quantitative and qualitative data that helped us see some advantages in the use of MoFQA Modeler, aimed at end-users.

Even though the examples were relatively simple, we drove the laboratory experiences with real websites. In the first experience, which was driven by students, the virtual classroom website of the university was tested (a system based on the platform https://claroline.net/). The second experience generated and drove tests for the actual website https://www.amazon.es/.

4.4 Discussion

The results show an advantage towards the generation of testing code compared to the definition and execution of manual tests. The time required for the definition of tests does not make a significant difference but the generated tests have the advantage that they can be saved and reused, whenever necessary. In addition, the number of test steps generated (and thus the number of verifications performed) is greater than the number of steps performed manually. The difference is significant even for small tests. Looking into the qualitative data (comments and manual-test descriptions made by the students), we could see that their manual tests are too general and ambiguously-defined (the test steps are not described so it is not possible to reproduce their manual test). This is not a minor issue, as beginner testers (and moreover, end-users) often have difficulties in developing their testing abilities and the testing-learning curve needs to be considered, as seen at the "Experiences in Software Testing Education" in (22). Thus, MBT tools are good options for beginners and end-users as the implicit test steps are abstracted by the higher views that models offer.

Modeling with MoFQA Modeler also proved to be advantageous in terms of the amount of time saved compared to the time it would take to model using profiles and some UML editor. This is due to the fact that MoFQA Modeler only asks that the end-user specifies the aspects that are visible to him/her, abstracting the details regarding to test implementation.

About the usability of MoFQA Modeler, the students participating in the first experience answered a SUS questionnaire. The average score obtained in this questionnaire is still below the desired average, probably due to a series of issues that were presented in that experience. However, for a subset of students who were able to finish the experience, there was a significant increase in the SUS value obtained, reaching almost the average value of the scale. The comments made by the students and the known limitations of the tool need

to be considered in order to keep improving this tool.

The inclusion of the end-user to the process of definition and verification of Web application requirements can be obtained thanks to the advantages of the MoFQA Modeler tool: savings in modeling time, the abstraction of modeling elements and the level of usability of the tool. In this way, the end-user will be able to define the acceptance criteria for each requirement to be verified and generate acceptance tests to verify them. As the tests are generated automatically, they will be a faithful copy of the end-user's requirements, without the intervention of the developers or testers in the definition of those tests. Another interesting point in this sense is the application of DDD principles in the definition of requirements through the use of MoFQA tools: the end-user defines the elements of the domain (business logic) as well as the names of the components of the various pages, thus forcing the developer to use the same concepts during their programming tasks (otherwise, the generated tests would not pass).

Finally, the coverage of requirements by the generated tests could be verified. It was verified that for the requirements modeled by the MoFQA tools, it was possible to obtain 100% coverage through the transformation rules, generating also, for each test, several intermediate steps that increase the number of verifications carried out.

5 Conclusion and Future Work

This article has presented the MoFQA approach, which is oriented to software development based on the TDD process. The software development process defined by MoFQA is integrated with MBT techniques to reinforce (and automate) the generation of tests. In a complementary way, we developed MBT tools as proposals for the definition and generation of unit and acceptance tests for Web platform systems. These tools are general, therefore they could be used or not in the development process proposed by MoFQA.

In order to offer an initial analysis of the usefulness of the tools developed, we carried out two validation experiences: (i) a practical study with students, to obtain an initial perception of the usability of the system; (ii) a comparison, where we compared the times needed to define tests using the UML profiles and the proposed modeling tool. The analysis of the proposal has focused on two specific aspects: (1) test coverage for the defined requirements and the number of test steps generated, with respect to manual testing the requirements; (2) simplicity in the use of the tool aimed at end-users for the generation of acceptance tests; (3) time savings to define tests by using our MBT tools. Both experiences focused on the tools generated as part of this proposal, as one important limitation we observed in the literature is the lack of integrated environments to pursue the entire software testing and development cycle.

With these preliminary experiences we realized that, with little preparation, end-users (non-technical) can automate the execution of acceptance tests aided by MBT tools. From simple models defined by the end-users, thanks to the transformation rules, several intermediate test steps were generated automatically for each requirement, also facilitating the integration of the modeling and definition of tests with the generation of executable test code. This allowed us to verify a decrease in the time needed to model requirements and also, facilitated the active and fluid participation of the end-user in the definition and verification of requirements based on their acceptance criteria.

As future work, we propose the definition of other empirical validation experiences as experiments, to obtain greater evidence both about the usefulness of the software development process, based on TDD and MBT practices, and about the usability of the tools. Also, the generalization/extension of the modeling tool and the transformation rules are being considered so that the tests are expanded to other deployment platforms as well. Finally, MoFQA Modeler showed interesting advantages aimed at end-users and beginner-level testers, thus the improvement critiques (received by the users that this tool had so far) need to be considered and new usability tests need to be run over the tool, in order to allow a more active collaboration of end-users in the entire development cycle.

Acknowledgment

The results presented in this paper were produced as part of the project "Mejorando el proceso de desarrollo de software: Una propuesta basada en MDD" (14-INV-056), financed by the National Council of Science and Technology (CONACYT).

References

[1] M. Utting, "Position paper: Model-based testing," *Verified Software: Theories, Tools, Experiments. ETH Zürich, IFIP WG*, vol. 2, 2005.

[2] K. Beck, *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[3] R. V. Binder, A. Kramer, and B. Legeard, "2014 Model-based Testing User Survey: Results," Tech. Rep., 2014. [Online]. Available: http://www.cftl.fr/wp-content/uploads/2018/10/2014-MBT-User-Survey-Results.pdf

[4] C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, "Assessing model-based testing: an empirical study conducted in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014, pp. 135–144. [Online]. Available: http://dx.doi.org/10.1145/2591062.2591180

[5] C. Jones, P. O'Hearn, and J. Woodcock, "Verified software: A grand challenge," *Computer*, vol. 39, no. 4, pp. 93–95, 2006. [Online]. Available: https://doi.org/10.1007/11813040_45

[6] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice, Second Edition*, ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2017. [Online]. Available: https://doi.org/10.2200/S00751ED2V01Y201701SWE004

[7] J. Hofstader, "Model-driven development applied to microsoft visual studio, 2006." *Available by Internet: https://msdn.microsoft.com/en-us/library/aa964145.aspx.*

[8] J. G. Guerra, "Web 2.0 presente y futuro de las aplicaciones," *Perspect. s*, vol. 4, no. 4, pp. 60–64, 2017.

[9] J. M. Rivero, J. Grigera, G. Rossi, E. R. Luna, F. Montero, and M. Gaedke, "Mockup-driven development: providing agile support for model-driven web engineering," *Information and Software Technology*, vol. 56, no. 6, pp. 670–687, 2014. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2014.01.011

[10] L. Riquelme, M. González, N. Aquino, and L. Cernuzzi, "Mofqa: An approach for automatic TDD test case generation from MDD models," in *XLIV Latin American Computer Conference, CLEI 2018, São Paulo, Brazil, October 1-5, 2018.* IEEE, 2018, pp. 11–20. [Online]. Available: https://doi.org/10.1109/CLEI.2018.00012

[11] A. Sadeghi and S.-H. Mirian-Hosseinabadi, "Mbtdd: Model based test driven development," *International Journal of Software Engineering and Knowledge Engineering*, vol. 22, no. 08, pp. 1085–1102, 2012. [Online]. Available: http://dx.doi.org/10.1142/S0218194012500295

[12] B. A. Shappee, "Test first model-driven development," Ph.D. dissertation, Worcester Polytechnic Institute, 2012.

[13] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007.* ACM, 2007, pp. 31–36. [Online]. Available: http://dx.doi.org/10.1145/1353673.1353681

[14] J. Peleska, "Industrial-strength model-based testing-state of the art and current challenges," *arXiv preprint arXiv:1303.1006*, 2013. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.111.1

[15] L. Villalobos-Arias, C. Quesada-López, A. Martinez, and M. Jenkins, "Model-based testing areas, tools and challenges: A tertiary study," *CLEI Electronic Journal*, vol. 22, no. 1, 2019. [Online]. Available: http://dx.doi.org/10.19153/cleiej.22.1.3

[16] M. Utting and B. Legeard, *Practical model-based testing: a tools approach.* Morgan Kaufmann, 2010.

[17] E. Escott, P. Strooper, J. Steel, and P. King, "Integrating model-based testing in model-driven web engineering," in *Software Engineering Conference (APSEC), 2011 18th Asia Pacific.* IEEE, 2011, pp. 187–194. [Online]. Available: http://dx.doi.org/10.1109/APSEC.2011.61

[18] F. Marinescu and A. Avram, *Domain-driven design Quickly.* Lulu. com, 2007.

[19] L. Riquelme, "Mofqa: Una propuesta para la generación automática de tests a partir de modelos siguiendo el proceso tdd," Universidad Católica "Nuestra Señora de la Asunción", Tech. Rep., 2017. [Online]. Available: http://www.dei.uc.edu.py/proyectos/mddplus/wp-content/uploads/2018/01/mofqa_libro.pdf

[20] J. Sauro, "Measuring usability with the system usability scale (sus)," 2011.

[21] A. Bangor, P. Kortum, and J. Miller, "Determining what individual sus scores mean: Adding an adjective rating scale," *Journal of usability studies*, vol. 4, no. 3, pp. 114–123, 2009.

[22] J. Timoney, S. Brown, and D. Ye, "Experiences in software testing education: some observations from an international cooperation," in *2008 The 9th International Conference for Young Computer Scientists*. IEEE, 2008, pp. 2686–2691. [Online]. Available: http://dx.doi.org/10.1109/ICYCS.2008.209