Universidad Católica "Nuestra Señora de la Asunción"

Departamento de Electrónica e Informática

MoFQA: Una Propuesta para la Generación Automática de Tests a partir de Modelos siguiendo el Proceso TDD[1]

Proyecto Final de Carrera

INGENIERÍA EN INFORMÁTICA

Linda Riquelme

Tutora	**Co-Tutor**	**Colaboradora**
Ing. Magalí González	Dr. Luca Cernuzzi	Msc. Nathalie Aquino

Asunción, Paraguay

Mayo, 2017

[1] Este proyecto es financiado por el CONACYT a través del programa PROCIENCIA con recursos del Fondo para la Excelencia de la Educación e Investigación (FEEI) del FONACIDE.

Resumen

Debido a la complejidad de los sistemas de software y a la alta posibilidad de inserción de errores en el producto, durante cada etapa del ciclo de vida del software, la utilización de técnicas de verificación de calidad resulta primordial. El testing de software es un proceso orientado a la ejecución de sistemas de software con la finalidad de encontrar errores en ellos. Pese a su utilidad, el testing es un proceso costoso por lo que su aplicabilidad es puesta bajo la lupa por cada equipo de desarrollo. En este trabajo, proponemos un método de desarrollo de software que utiliza como base el proceso Test-Driven Development (TDD) con el soporte de herramientas Model-Based Testing (MBT) que permitan la generación automática de códigos de test. Además se propone un conjunto de herramientas para la generación de tests de aceptación y unitarios, orientados a sistemas de plataformas web.

Palabras Clave: Software Testing, MBT, TDD, Web Testing, Selenium, TestNG.

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

Lista de Fórmulas

Agradecimientos

A Dios, por las oportunidades que me brindó y por las personas que puso en mi camino. Por ser mi fortaleza en momentos buenos y malos, por permitirme llegar al final de la meta.

A mi familia, por ser mis pilares en la vida, por acompañarme y ayudarme a lo largo de este trayecto.

A mis tutores Magalí, Luca y Nathalie por compartir su vasta experiencia para el desarrollo de este trabajo y por ser guías indispensables hasta la finalización del proyecto. Un especial agradecimiento a la profe Magalí por su ayuda y seguimiento incansables hasta el día de hoy. Su entrega me ayudó a seguir en el camino pese a todas las dificultades.

A mis compañeros de la "pecera" quienes con su amistad y compañerismo hicieron que los días difíciles sean más fáciles de sobrellevar.

Finalmente, a los amigos que siempre están, por el apoyo, acompañamiento y buenos deseos. Este trabajo está especialmente dedicado a mi querido y siempre recordado amigo Derlis Ariel Leiva Esteffen.

Introducción

El proceso del desarrollo de *software* es complejo. Comprende, entre otras tareas: la definición de requisitos y funcionalidades a incluir, la implementación de módulos que interactúan entre sí, la adición de nuevas funcionalidades a medida que van siendo necesarias, o modificaciones en las funcionalidades ya implementadas en cuanto se dan variaciones en los requerimientos. En este proceso, aparecen diversas variables y componentes que pueden provocar situaciones no deseadas.

Ante la necesidad de otorgar garantías sobre la calidad del producto, aparecen técnicas de ***software testing***. De acuerdo a los conceptos introducidos por Myers en [1], definimos *software testing* como "el proceso de ejecutar un programa con la intención de encontrar errores". Con ello, decimos que el *testing* no asegura que un producto funciona correctamente bajo todas las condiciones posibles, solo puede sacar conclusiones con respecto a condiciones específicas [1].

Las limitaciones y dificultades que se presentan en el *software testing* tendrán implicancias en puntos como [1]: los costos y la valorización del proceso de *testing*, la búsqueda de métodos de diseño que permitan obtener *tests* más óptimos (es decir, con la menor cantidad posible de *tests*, encontramos un mayor número de errores) así como algunas abstracciones que se deberán realizar sobre el producto a verificar. En este trabajo buscamos promover el proceso de *testing*, aplicándolo de forma más natural al proceso de desarrollo de *software* (tratando de reducir los tiempos y esfuerzos necesarios para su implementación). Con este objetivo, nos centraremos en dos enfoques principales: *Model-Based Testing* y *Test-Driven Development*.

***Model-Based Testing* (MBT)** [2] aparece como una propuesta MDD (*Model-Driven Development*)[2] para facilitar el *testing* automatizado, permitiendo la generación automática de *tests* a partir de modelos que representan una abstracción del sistema a verificar. Al día de hoy, esta técnica de *testing* ha sido bastante adoptada y posee soporte de varias herramientas en diversas áreas de aplicación, utilizándose tanto con fines académicos así como en la industria[3]. ***Test-Driven Development* (TDD)** [3], por su parte, es un proceso de desarrollo de *software* basado en *tests*: para cada funcionalidad a implementar, se define inicialmente un conjunto de *tests* que dirigirá el proceso del desarrollo; el objetivo principal de este enfoque, es la implementación de soluciones que pasen los *tests* de forma exitosa. Consiste en una práctica definida en algunas metodologías de desarrollo ágil y ofrece, entre otras ventajas, la posibilidad de obtener un *software* (generado de forma incremental en cada iteración) con mayor cantidad de *tests* asociados.

Sin embargo, la definición de *tests* en TDD es un paso que se realiza, generalmente, de forma manual: se elaboran *scripts* utilizando *frameworks* y herramientas específicas de *testing* para definir los *tests*. Los *tests* definidos resultan tan largos que finalmente es posible tener tantas líneas de código de *testing* como líneas de código de implementación del *software* a verificar. Si la generación de *tests* es apoyada por técnicas y herramientas que acorten (y automaticen) los pasos de definición y ejecución de *tests*, podríamos facilitar la integración de técnicas TDD al proceso

[2] MDD (*Model-Driven Development*) consiste en un paradigma para el desarrollo de *software* en el que se utilizan modelos para la representación abstracta del sistema, a partir de los cuales se generará posteriormente el sistema final.

[3] La encuesta realizada por Robert V. Binder, Anne Krammer, Bruno Legeard en el año 2014, refleja un importante grado de aceptación de MBT (para la población encuestada) a nivel de la industria. La encuesta "2014 Model-based Testing User Survey", se encuentra disponible en: http://model-based-testing.info/wordpress/wp-content/uploads/2014_MBT_User_Survey_Results.pdf

de desarrollo de *software*. En particular, en este trabajo buscaremos integrar la idea de generación de *tests* a partir de modelos, utilizando técnicas MBT, al proceso de desarrollo de *software* siguiendo los pasos definidos en el paradigma TDD. Ambos enfoques presentan ventajas al ser aplicados al proceso de desarrollo de *software*, puesto que facilitan la generación de *tests* desde diversos puntos de vista. Además, hemos encontrado una cantidad muy baja de trabajos (un ejemplo es [4]) que integren ambos enfoques (MBT y TDD), mientras que ambos ofrecen ventajas muy interesantes que podrían ser aprovechadas en conjunto.

En nuestra investigación, hemos verificado principalmente dos limitaciones[4] aún existentes en herramientas MBT en general. Primeramente, aún existe una brecha importante en la integración de herramientas que permitan realizar todos los pasos para la automatización de *tests* (definición, generación y ejecución de *tests*). Por otro lado, si bien los modelos representan abstracciones del sistema a implementar, muchas notaciones utilizadas aún resultan complicadas y tediosas, inclusive para los mismos *testers*. Es de nuestro interés, involucrar al usuario final en la definición de *tests* basados en sus propios criterios de aceptación, buscando de esta forma que el sistema se adecue a sus requerimientos.

Así, el objetivo general de nuestra propuesta es la definición de un método de desarrollo de *software* basado en la definición de *tests* a partir de modelos (que representan los requerimientos de cada funcionalidad a implementar) definidos por el usuario y el desarrollador, siguiendo los pasos y prácticas descritos en TDD. Se construirá una herramienta que dará soporte al método propuesto para generar y ejecutar *tests* automáticamente, de forma integrada con el ciclo de desarrollo de *software*. A fin de limitar el dominio, nos pareció interesante enfocarnos en el desarrollo de *software* para plataformas Web 2.0. Realizamos esta elección debido al gran auge de este tipo de aplicaciones, también destacado en [5]: "El *testing* de aplicaciones *web* es un proceso muy importante ya que es el área de mayor crecimiento en ingeniería de *software*".

Hemos definido, además, los siguientes objetivos específicos:

- Definición de un método de trabajo que integre los pasos de TDD, siguiendo las prácticas definidas por MBT.
- Definición de perfiles UML[5] para la representación de *tests* aplicables a sistemas de plataforma Web.
- Definición de reglas de transformación para la generación de *tests* ejecutables a partir de modelos UML, enriquecidos con los elementos del perfil.
- Desarrollo de una herramienta para el modelado de *tests* de aceptación, a ser utilizada por usuarios finales de los sistemas a verificar.
- Validación de resultados.

Con esto se busca la automatización y simplificación del trabajo de *testing*, aprovechando las ventajas de abstracción y generación automática ofrecidas por los modelos y las herramientas de transformación. Siguiendo el proceso TDD, se requiere que cada funcionalidad tenga una serie de *tests* relacionados en la medida que se avanza con la implementación.

---

[4] Ambas limitaciones pueden verse reflejadas en los resultados obtenidos en la encuesta "2014 Model-based Testing User Survey", disponible en: http://model-based-testing.info/wordpress/wp-content/uploads/2014_MBT_User_Survey_Results.pdf

[5] UML (Unified Modeling Language) es un lenguaje de modelado definido por la OMG. Referencia: http://www.uml.org/.

La utilización de perfiles UML para definir los modelos tiene la finalidad de permitir la representación de *tests* a partir de modelos conocidos ampliamente (y simplificados en relación a otras notaciones matemáticas [6] más formales). Las reglas de transformación definidas para modelos que utilicen los perfiles UML mencionados, permitirían la generación automática de *tests*, facilitando además la integración del proceso de modelado y definición de *tests* con la generación de código de *tests* ejecutables. Además, los *tests* generados están escritos en lenguajes y *frameworks* soportados por el entorno de desarrollo integrado Eclipse. Así, todo el proceso de desarrollo y *testing* de un sistema específico podría efectuarse utilizando Eclipse: desde el modelado de *tests* a partir de los perfiles UML hasta la generación y ejecución de código de *tests*.

Finalmente, una herramienta orientada principalmente a los usuarios finales del sistema a verificar permitiría involucrarlos en el proceso de definición de *tests* de aceptación, posibilitando consecuentemente la generación de *tests* asociados a los requerimientos del sistema, tal como los usuarios los definieron. Se busca con esta herramienta una mayor participación de los usuarios finales en la definición de *tests* sin necesidad de que éste tenga experiencia en modelado ni programación.

La validación final de la propuesta se enfoca solo en dos aspectos específicos: (1) cobertura de los *tests* generados, respecto a los requerimientos definidos para el sistema; (2) simplicidad en el uso de la herramienta dirigida a los usuarios finales para la generación de *tests* de aceptación.

El objetivo de este documento es la descripción de nuestra propuesta hacia el cumplimiento de los objetivos mencionados anteriormente. Se desarrolla en 5 capítulos principales:

- Capítulo 1. "*Testing* de *Software*": Se introducen las bases teóricas necesarias para la definición de nuestra propuesta, antecedentes y comparaciones entre los enfoques presentados.
- Capítulo 2. "Revisión del Estado del Arte": Se inicia el capítulo con la definición de una hipótesis que constituyó en la base de nuestra propuesta. Se presenta los resultados obtenidos tras realizar una investigación sobre trabajos previos en las áreas de TDD y MBT, concluyendo sobre el estado del arte en base a algunos criterios de interés. Se llega finalmente a la definición de la problemática y se establecen las bases para la formación de nuestra propuesta.
- Capítulo 3. "MoFQA: Una Propuesta para el Desarrollo de *Software* basada en TDD": Una vez planteada la problemática, se presenta el modelo propuesto para el desarrollo de *software*, además, se describen las herramientas desarrolladas como parte de este trabajo y se presentan dos casos de ejemplo utilizados para ejemplificar el uso de las diferentes herramientas.
- Capítulo 4. "Validación y Análisis de Resultados": Al finalizar el trabajo, se definieron dos pequeñas experiencias para obtener resultados preliminares sobre la utilidad de las herramientas propuestas (en cuanto a la cobertura de los *tests* generados y a la usabilidad de la herramienta de modelado dirigida a usuarios). En este capítulo se describen las experiencias, incluimos los datos recopilados y realizamos una discusión para el análisis de resultados.
- Conclusión: Se establecen las conclusiones de este trabajo y proponemos ideas para Trabajos Futuros.

Capítulo 1. Testing de Software

El proceso de desarrollo de *software* es complejo [1]. Comprende, entre otras cosas: la definición de requerimientos[6] y funcionalidades a implementar, la visualización de interacciones entre los diversos módulos y la adición de nuevas funcionalidades a medida que van siendo necesarias (o cambios de funcionalidades existentes dadas las variaciones en los requerimientos). Es así como los componentes del *software* interactúan e interfieren entre sí y pueden surgir situaciones no deseadas.

Además, las documentaciones que están cargadas con mucho texto, dibujos y diagramas pueden resultar ambiguas e irrelevantes [1]. Detalles importantes del sistema se encuentran así escondidos entre la masa de información, dificultando la detección de errores (éstos suelen detectarse de forma tardía, lo que provoca mayor costo y dificultad para su corrección).

Es necesario que la informática justifique el grado de confianza que la sociedad deposita continuamente sobre ella. Para esto es primordial que el grado de error en la programación (en las etapas de diseño, desarrollo, *testing*, instalación, mantenimiento y evolución del *software*) se reduzca de forma considerable.

Con el objetivo de asegurar la calidad del *software*, bajo ciertos criterios definidos previamente, el ***testing de software*** es un proceso que permite encontrar errores en el producto final. Este proceso consiste en el enfoque principal de este trabajo por lo que, a lo largo de este capítulo, introduciremos conceptos relevantes en esta área. Estos conceptos ayudaron a plantear una hipótesis, presentada en el siguiente capítulo de este documento.

¿Qué es el *Testing* de *Software*?

El *testing* de *software* es un proceso que permite detectar errores en el *software*, apuntando siempre a mejorar la calidad del producto. El *testing* no puede, sin embargo, asegurar que un producto funciona correctamente bajo todas las condiciones posibles [2], solo puede sacar conclusiones bajo condiciones específicas.

La esencia del *testing* de *software* está en determinar un conjunto de *test cases* para el *software* bajo verificación. Un *test case* consiste en la definición de: las precondiciones para la ejecución del *software*, el conjunto de entradas a utilizar en las pruebas, el conjunto de salidas esperadas (para esas entradas y precondiciones), y una descripción de las post-condiciones esperadas. Además, un *test case* completo tendrá los elementos: identificador de *test case*, descripción de los objetivos del *test case* y un historial de ejecución [3].

Realizar pruebas con todas las combinaciones de entrada y precondiciones no resulta factible (ni siquiera si hablamos de un producto pequeño y simple). La cantidad de defectos puede ser muy grande y, muchos de ellos, pueden ocurrir de forma tan infrecuente que son difíciles de detectar. Además, la dimensión no funcional[7] del *software* puede llegar a ser bastante subjetiva, dejando atrás la posibilidad de llevar a cabo su verificación. Éstos son problemas fundamentales que tendrán implicancias en puntos como [2]: análisis del valor y

[6] Requerimientos o especificaciones del *software* a implementar: consisten en un conjunto de funcionalidades y restricciones que el usuario esperaría del sistema. Éstos son definidos en discusión con el cliente y son analizados por el equipo de desarrollo para su posterior implementación. [5]

[7] Mientras que los requerimientos funcionales describen las funciones que el software debe ejecutar, los requerimientos no funcionales son características y restricciones que definen la calidad del producto [26].

costos del proceso de *testing*, la búsqueda de métodos de diseño que permitan obtener *test cases* más óptimos (es decir, con la menor cantidad posible de *tests*, encontramos un mayor número de errores) así como asunciones o abstracciones que los *testers*[8] deberán realizar sobre el producto a verificar.

Estrategias de *Testing*

Las técnicas de *testing* se clasifican en los tipos definidos a continuación:

- ***Black-box testing*:** También conocido con los nombres de *data-driven testing* o *input/output-driven testing*, este tipo de *testing* no tiene en cuenta ni el comportamiento interno ni la estructura del programa a verificar. Se concentra, sin embargo, en encontrar circunstancias en las cuales el programa no se comporta de acuerdo a las especificaciones.
- ***White-box testing*:** Esta estrategia recibe también el nombre de *logic-driven testing*. Al contrario de la técnica anterior, ésta permite examinar la estructura interna del programa, obteniendo información a partir de la lógica misma.

Sin importar la estrategia de *testing* utilizada, no es posible realizar una verificación exhaustiva que nos permita asegurar la correctitud del programa, bajo todas las condiciones posibles [2]. Utilizando técnicas *black-box* sería necesario ejecutar el programa a verificar utilizando todas las combinaciones de entrada posibles. El problema es aún mayor cuando el programa maneja memoria: sus resultados no dependen solamente de las entradas en un momento determinado sino, además, de su historia o datos previamente almacenados. En cuanto a las técnicas *white-box*, un *testing* exhaustivo requeriría la ejecución de todos los caminos del flujo de control del programa. Aunque esto sea posible, se presentan los siguientes problemas [2]: (1) no se garantiza que el programa cumpla con las especificaciones; (2) solo se siguen los caminos existentes y no es posible detectar la ausencia de caminos necesarios; (3) no es posible encontrar errores relacionados a los datos de entrada.

Es por esto que la fase del diseño se vuelve muy importante: el *tester* debe enfocar sus esfuerzos en obtener *test cases* más completos, dentro de lo posible. Esto permitirá encontrar la mayor cantidad de errores en el programa bajo prueba. Para ello, existen técnicas que ayudan a generar mejores *tests* de acuerdo a ciertos criterios (tal como lo presenta [2] en el capítulo 4 *"Test-Case Design"*).

Según [2], es posible desarrollar *tests* razonablemente rigurosos usando algunas técnicas de diseño orientadas al *black-box testing* y suplementarlos con métodos *white-box*, que permitan examinar la lógica del programa. Además, Jorgensen en [3], resalta que ambas técnicas son complementarias entre sí: es posible reconocer y resolver problemas (de redundancias y brechas existentes) en técnicas de *black-box testing*, combinándolas con técnicas *white-box*.

[8] Denominaremos ***tester*** a la persona encargada de diseñar y ejecutar los *tests* durante el proceso de desarrollo de *software*. De acuerdo a la metodología de desarrollo empleada, el *tester* puede ser una persona con el rol exclusivo de mantener y ejecutar los *tests* o el mismo desarrollador del programa bajo verificación.

Niveles de *Testing*

El proceso de desarrollo de *software* conlleva una serie de pasos y en cada uno de ellos existe la posibilidad de inserción de errores al producto final. Además, una gran cantidad de errores puede deberse a ambigüedades y problemas en el traspaso de la información [1]. Es por esto que el *testing* se debe llevar a cabo en varios niveles, cada uno de ellos teniendo en cuenta un aspecto diferente a verificar.

La literatura distingue distintos procesos de desarrollo de *software* [4], entre ellos: el modelo en cascada, el modelo espiral, el proceso unificado, el modelo V y el modelo W. En todos estos modelos, el producto es construido en fases y a cada una de ellas, es posible asociarle un proceso de *testing* que verifique aspectos específicos de dicho nivel. El modelo V, por ejemplo, define niveles de construcción y *testing* para el desarrollo de *software*:

- Se generan los requerimientos (metas para el producto final) a partir de las necesidades del usuario.
- Se definen objetivos a partir de dichos requerimientos luego de realizar un análisis: se resuelven los requerimientos conflictivos, se verifican los costos y las posibilidades/recursos, se establecen prioridades. Se traducen los objetivos en especificaciones más precisas para el producto. El *software* es visto como una caja negra con interfaces e interacciones con el usuario final.
- Se procede al diseño del sistema: el sistema es dividido en programas individuales, componentes o subsistemas siguiendo una estructura jerárquica y se definen interfaces de comunicación entre ellos.
- Se especifica la función de cada uno de los módulos.
- Finalmente, las especificaciones son traducidas al código fuente de cada módulo.

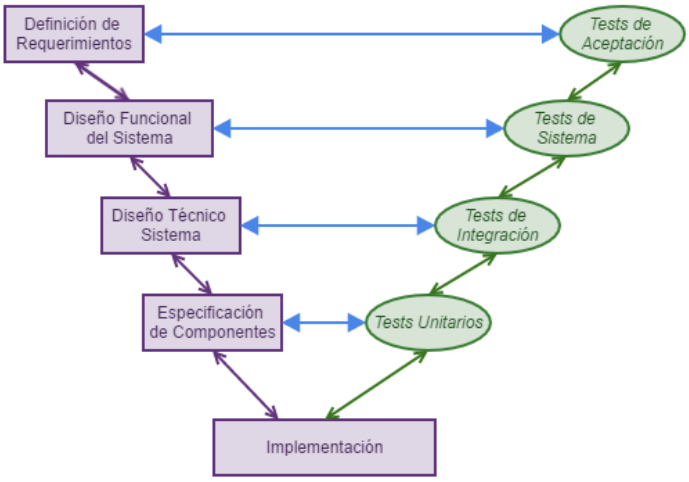
A partir de la implementación de los componentes se puede proceder a los siguientes tipos de *testing* [4]:

- ***Unit Testing* (*Tests* Unitarios):** El *testing* a nivel de módulos (*module testing* o *unit testing*) pone a prueba subprogramas, subrutinas o procedimientos individuales del programa, en lugar de que los *tests* se realicen sobre todo el programa como conjunto. Tiene dos variantes: (1) *testing* no incremental, donde cada módulo es verificado de forma independiente (finalmente se juntan los módulos para formar el programa final); (2) *testing* incremental (también conocido como ***integration testing***), en el cual cada módulo es combinado al conjunto de módulos previamente verificados[9], antes de ser sometido a los *tests.*
- ***Integration Testing* (*Tests* de Integración)*:*** como hemos mencionado, interesa probar a este nivel, varios módulos integrados. Se verifica así como actúan en conjunto y las interfaces definidas para su comunicación.
- ***System Testing* (*Tests* de Sistema):** Los *test cases* se formulan en base al análisis de los objetivos definidos. Los objetivos solo determinan lo que un programa debe hacer y qué tan bien debe hacerlo, no definen las funciones que deberían implementarse para lograr dichos objetivos. A este nivel, es posible llevar a cabo los tipos de *testing* [2]: *facility testing*, *stress testing*, *usability testing*, *security testing*, *performance testing*, *storage testing*, *configuration testing*, *compatibility*

[9] La integración de módulos puede realizarse usando técnicas *top-down* o *bottom-up* [2].

testing, *installability testing*, *reliability testing*, *recovery testing*, *serviceability testing*, *documentation testing*, *procedure testing*. Como se dispone del sistema completo, es posible verificar su funcionamiento global.

- ***Acceptance Testing* (*Tests* de Aceptación):** Tiene el objetivo de comparar el programa con sus requerimientos iniciales y las necesidades de los usuarios finales. Este proceso es llevado a cabo por el usuario final.



En la

Figura 1, se ilustra la asociación de los niveles de *testing* mencionados a cada uno de los pasos del desarrollo de *software*.

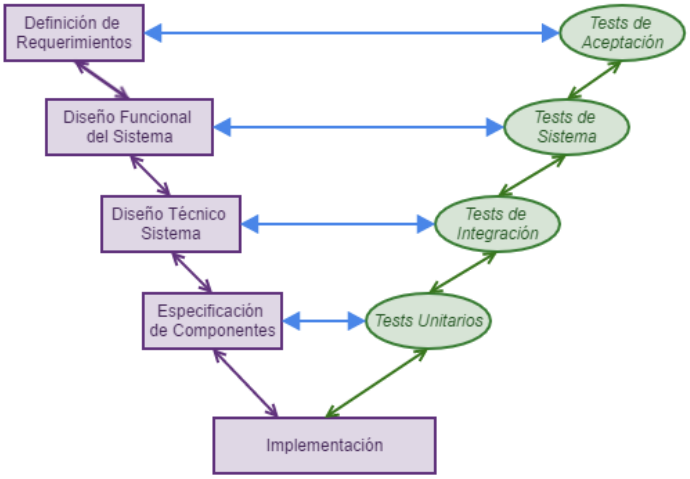


Figura 1: Modelo V para el desarrollo de *software* [4]

Las metodologías de desarrollo utilizadas están basadas con mayor frecuencia en los modelos cascada, espiral y V; todos estos modelos tienen el punto común: la actividad de *testing* se lleva a cabo recién al finalizar la etapa de codificación [5].

Las formas de ejecución del *testing* y los momentos en que es llevado a cabo, varían de acuerdo a la metodología de desarrollo de *software* utilizada. A continuación, presentamos las metodologías ágiles de desarrollo de *software*, las cuales, definen prácticas que utilizamos como puntos de partida para la definición de nuestra propuesta.

Metodologías ágiles en el desarrollo de *software*

En febrero de 2001, tras una reunión celebrada en Utah-EEUU, nace el término "ágil" aplicado al desarrollo de *software*. El objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar *software* rápidamente y respondiendo a los cambios que puedan surgir a lo largo del proyecto. Se pretendía ofrecer una alternativa a los procesos de desarrollo de *software* tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades. [6]

Tras esta reunión se creó "*The Agile Alliance"*, una organización, sin ánimo de lucro, dedicada a promover los conceptos relacionados con el desarrollo ágil de *software* y ayudar a las organizaciones para que adopten dichos conceptos. El punto de partida fue el **Manifiesto Ágil**. Según el Manifiesto se valora [6]:

- Al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas. La gente es el principal factor de éxito de un proyecto *software*. Es más importante construir un buen equipo que construir el entorno.
- Desarrollar *software* que funciona más que conseguir una buena documentación. La regla a seguir es no producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante. Estos documentos deben ser cortos y centrarse en lo fundamental.
- La colaboración con el cliente más que la negociación de un contrato. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.
- Responder a los cambios más que seguir estrictamente un plan. La habilidad de responder a los cambios que puedan surgir a lo largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

Los métodos ágiles, por ejemplo *Extreme Programming* (XP) y *Scrum*, son procesos que cumplen los criterios definidos en la filosofía de desarrollo ágil. En particular, XP es un método ágil para el desarrollo de *software* que promueve el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. Se basa en la retroalimentación continua entre el cliente y el equipo de desarrollo, la comunicación fluida entre todos los participantes, la simplicidad en las soluciones implementadas y el coraje para enfrentar los cambios. [6].

El desarrollo de *software* en XP es un proceso iterativo [7]: cada iteración es un ciclo de desarrollo completo que incluye las actividades de diseño, codificación, verificación y publicación. Generalmente dura entre 1 a 3 semanas. Cada iteración inicia con la selección de historias de usuario[10] a ser implementadas en ese periodo de tiempo, y finaliza con *software* producido por el equipo que puede ser instalado y utilizado por los usuarios finales.

En cuanto al *testing* de *software*, XP define una serie de prácticas, descritas a continuación [7]:

[10] **Historias de usuario:** Son la técnica usada para especificar los requisitos del software en las metodologías ágiles. Se trata de tarjetas en las que el cliente describe brevemente las características que debe poseer el sistema (requisitos funcionales o no funcionales). [6]

Cada miembro del equipo (clientes, desarrolladores y *testers*) contribuye a la calidad del *software*. La técnica propuesta se denomina *Test-Driven Development* (TDD), que será explicada con mayor detalle en la siguiente sección. TDD permite producir *tests* unitarios, de integración e inclusive *tests* de usuario. Estos *tests* permiten verificar si el *software* cumple con los objetivos del equipo de desarrollo. Además, los *tests* de usuario permitirían comprobar si lo desarrollado por los programadores reúne las expectativas de los clientes. Los resultados de la ejecución pueden ser utilizados por el cliente para decidir sobre el avance del proyecto.

Test-Driven Development (TDD)

Test-Driven Development (TDD) es un proceso de desarrollo de *software* fundado en los conceptos de programación de *test-first* usados en *Extreme Programming*. La filosofía usada en TDD es *Red-Green-Refactor*, consistente en un desarrollo iterativo tanto de los *tests* como del producto final.

Como se destaca en [5], TDD no es una técnica de *testing* sino de desarrollo y diseño de *software*, en la cual los *tests* son escritos antes del código de producción. Los *tests* son definidos de forma gradual durante la implementación y, a medida que los *tests* pasan, el código pasa por un proceso de limpieza (*refactor)* para mejorar la calidad del código. El ciclo se repite hasta que todas las funcionalidades deseadas sean implementadas.

En TDD, el desarrollador inicia el proceso escribiendo un *test case* ejecutable que será utilizado para verificar una funcionalidad a ser implementada. La funcionalidad puede estar especificada a través de una historia de usuario y se supone que el producto no cuenta aún con dicha funcionalidad. En lugar de enfocarse en la forma de implementación de las funciones y métodos del *software*, los *tests* definidos en TDD están orientados a probar el comportamiento real. Si los *tests* son escritos al finalizar la implementación, existe un riesgo de éstos estén orientados a satisfacer la implementación y no los requerimientos para dichas funcionalidades [5].

Se espera que el *test* falle (*Red*) luego de ejecutarlo por primera vez puesto que se supone que la funcionalidad no está incluida. Si el *test* pasa puede ocurrir: (i) la funcionalidad en cuestión ya estaba implementada; (ii) el *test* presenta algún problema en su definición y no cumple con la función para lo cual fue creado.

Cuando un *test* falla, el programador escribe el código necesario para pasarlo. En [8], Beck propone tres técnicas posibles para la escritura de código de implementación (destinados a pasar los *tests*): fingir (*fake it*) la implementación del código (retornando los resultados esperados por el *test*), triangulación[11], implementación del código real. Las dos primeras técnicas sirven como pasos auxiliares que pueden permitir a los desarrolladores a visualizar más fácilmente (con un enfoque *top-down*) las formas de implementación de la funcionalidad esperada.

Luego de asegurarse de que se ha pasado toda la serie de *tests* (*Green*), el código puede ser limpiado (*Refactor*) y el proceso vuelve a empezar. El *Refactor*, por su parte, es el proceso de llevar a cabo cambios en el código para introducir mejoras en el diseño de la solución, sin alterar el comportamiento de la porción de código modificada. Permite que el código final resulte más claro y simple, manteniendo la seguridad de que todos los *tests* anteriores vuelvan a pasarse exitosamente.

[11] Triangulación (*triangulation*): Busca un grado de abstracción para la técnica *fake it* [8] cuando la implementación real de la funcionalidad a probar no resulta muy obvia.

La **Figura 2** grafica el proceso descripto.

A continuación, destacamos algunas prácticas definidas para llevar a cabo el proceso TDD (mencionadas en [7]):

- Todo código de producción a escribir debe estar únicamente orientado a pasar los pequeños *tests* definidos. La regla: "No agregar código de producción, a menos que tengamos algún *test* fallido".
- El *test* a definir en cada iteración debe ser corto y simple, debe definirse pensando solo en la funcionalidad esperada (y no en la forma en que sería implementada).
- El *Refactoring* permite revisar el código y buscar posibles mejoras. Como los *tests* definidos volverán a ser ejecutados, no nos preocupamos por los posibles problemas que pueda causar la modificación de código.
- El tiempo que lleva la ejecución de *tests* es importante por lo que se orienta el diseño de *tests* de forma a que sean de ejecución rápida. Por esta razón, la mayoría de los *tests* deberían ser *tests* unitarios[12], unos cuantos *tests* de integración y una pequeña minoría de *tests* de usuarios.

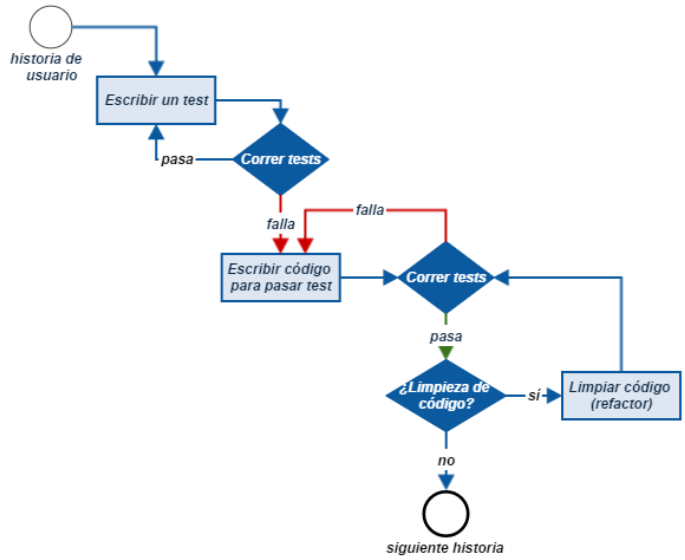


Figura 2: Proceso TDD

Ventajas de TDD

Beck, uno de los mayores contribuyentes en el área de metodologías ágiles y TDD, menciona en [8] las siguientes ventajas de la utilización de TDD:

- El desarrollador en todo momento estará únicamente enfocado en satisfacer los requerimientos de una pequeña funcionalidad a implementar. Esto permitiría que los diseños de las soluciones resulten más simples, limpios y entendibles, lo que a su vez puede resultar en códigos más modularizados y flexibles.

[12] Los *tests* unitarios deberían poder ejecutados a una razón de al menos 100 unidades por segundo puesto que no realiza operaciones como: consultas a bases de datos, comunicación en red, contacto con sistemas de archivos. Los *tests* que, en cambio, realizan ese tipo de operaciones pueden ser considerados *tests* de integración [7].

- Es seguro que se diseñará al menos un *test* por cada funcionalidad a incluir en el *software* final.
- A la hora de escribir los *tests*, el desarrollador debe tener bien en claro las especificaciones y/o requerimientos del producto final por lo que el proceso de diseño de *tests* lo lleva a afianzar los conceptos que necesita.
- La re-ejecución de las pruebas otorga al desarrollador una mayor confianza en la calidad del producto resultante. Permite además asegurar que las limpiezas realizadas al código no afectarán las funcionalidades implementadas.
- Con respecto a la interfaz de usuario: el desarrollador que se enfoca primeramente en las pruebas debe imaginarse sobre la forma en que los usuarios interactuarán con el sistema, por lo que se concentra en la interfaz antes que en la implementación.

Limitaciones de TDD

En cuanto a las limitaciones, [8] menciona los puntos:

- Los *tests* creados en el marco de este proceso generalmente son elaborados por el desarrollador, quien escribirá posteriormente el código a verificar[13]. Esto conlleva un problema, sobre todo, si el desarrollador no tiene bien claras las especificaciones del sistema a implementar.
- El *testing* basado solo en *tests* unitarios no resulta suficiente para situaciones donde es requerido un *testing* funcional completo.
- La gran cantidad de *tests* realizados puede consumir mucho tiempo y, además, se requiere su mantenimiento a medida de que los requerimientos van evolucionando. Además, una cantidad importante de *tests* ejecutados y superados puede dar una falsa sensación de seguridad, resultando finalmente en la disminución de actividades de *testing*.

Estas limitaciones pueden resultar en obstáculos para la utilización de TDD: una barrera principal es la idea de que la tarea de *testing* es tediosa y de que no representa una ventaja real para la evolución de un proyecto. A lo anterior se suma el costo adicional del *refactoring* (realizado en la etapa final de cada iteración de TDD) razón por la cual los desarrolladores pueden tender a enfocarse inicialmente en la implementación y dejar el *testing* como actividad secundaria.

Niveles de TDD

Podemos implementar TDD en dos niveles distintos: *tests* de usuario, *tests* de desarrollador. En particular, se denomina **ATDD** (***Acceptance TDD***) al proceso TDD orientado a la definición de *tests* por parte del usuario final, quien puede definir *tests* de aceptación para cada funcionalidad a implementar. Denominaremos simplemente **TDD** a la serie de pasos seguida por el desarrollador para definir *tests* más específicos y orientados a la forma en que se implementa cada funcionalidad.

[13] Uno de los principios del *testing* incluidos por Myers en [2], hace énfasis en que un desarrollador no debe implementar los *tests* de su propio programa. Como justificación, se mencionan aspectos psicológicos que intervienen sobre una persona (o grupo de personas) poniendo a prueba sus propios productos.

En lo que resta de la sección mencionaremos las características de ATDD y las diferencias entre ambos niveles de TDD.

Acceptance TDD (ATDD)

Acceptance TDD (ATDD), consiste en una adaptación del proceso TDD, orientado a los usuarios finales del *software* a verificar. Permite la derivación de *tests* de aceptación o especificaciones ejecutables para el producto final.

De esta forma, el cliente cuenta con un mecanismo automático para saber si el producto cumple con los requerimientos. El equipo de desarrollo, por su parte, tiene un objetivo específico en el cual está enfocado: satisfacer los *tests* de aceptación especificados por los clientes en las historias de usuario.

Este método es aplicable a proyectos con un dominio lo suficientemente complejo como para causar problemas en la comunicación y el entendimiento de los requerimientos. Están diseñados para soportar el desarrollo de *software* en equipo desde una perspectiva de negocio, de forma a asegurar la calidad del producto. Sus *tests*, sin embargo, no reemplazan a aquellos que tienen una perspectiva más técnica (como los *tests* unitarios o los *tests* de integración) o que evalúan el producto final (como los *security tests*).

La aplicación de este tipo de TDD requiere la validación frecuente de la funcionalidad del *software* contra un conjunto grande de ejemplos. Para ello, las herramientas de automatización están divididas en dos capas: especificación (contiene la interfaz de especificación, frecuentemente en texto plano o HTML, con ejemplos y descripciones auxiliares) y automatización (conecta los ejemplos con el sistema a bajo *test*).

Comparaciones entre los niveles de TDD

TDD brinda a los desarrolladores una técnica que los ayuda a crear unidades de código bien escritas. ATDD, por su parte, facilita la comunicación entre el cliente, el desarrollador y el *tester* para asegurar que los requerimientos del *software* a implementar estén bien definidos.

En la **Figura 3** se diagrama la forma en que ATDD y TDD pueden ser utilizados en conjunto para el desarrollo de *software* [9].

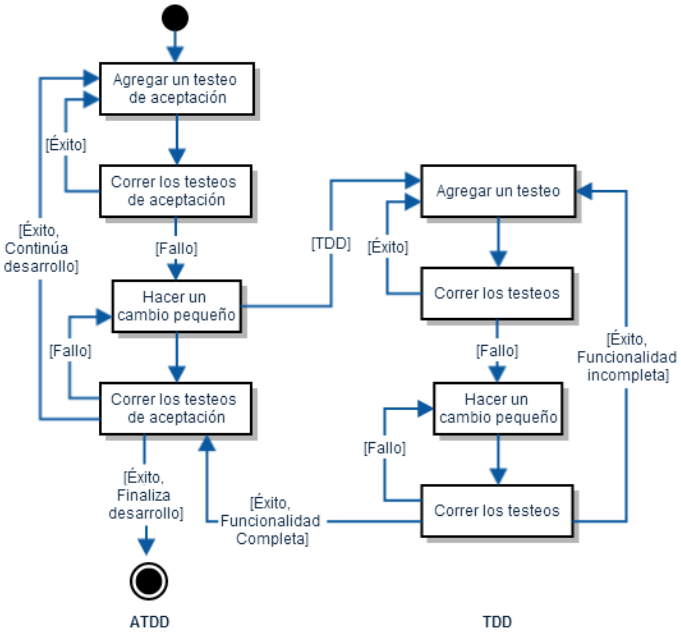


Figura 3: Implementación de TDD y ATDD en conjunto [9]

Automatización del Proceso de *Testing*

En esta sección presentamos los tipos de técnicas de *testing* utilizadas en la industria (según la clasificación presentada en [10]): *testing* manual, *capture/replay testing*, *script-based testing* y *keyword-driven testing*.

Testing Manual

Se diseñan los *test cases* de forma manual a partir de los documentos de especificaciones del sistema. Estos *tests* son interpretados por un *tester*, que los ejecuta también de forma manual y compara los resultados arrojados por el sistema, con los resultados esperados. Finalmente, el *tester* genera los resultados de los *tests* a partir de los datos registrados. Todos los pasos son realizados de forma manual lo que deriva en costos muy altos para realizar el proceso de *testing*.

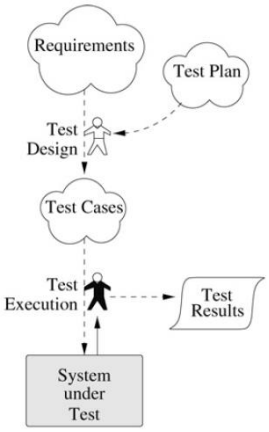


Figura 4: *Testing* Manual [10]

El plan de *test* otorga una visión general de los objetivos de *testing*, los aspectos a ser verificados, las estrategias de *testing* a ser utilizadas y con qué frecuencia deben ejecutarse. La ejecución manual de *tests* es repetida cada vez que es necesaria una verificación de una nueva versión del sistema. Como no existe automatización de las ejecuciones de *test*, el costo es constante y alto por lo que muchas veces es necesario reducir la cantidad de *tests* a ejecutar. Esto finalmente puede resultar en riesgos para la madurez, estabilidad y robustez del sistema.

Capture/Replay Testing

Al igual que el *testing* manual, la definición de *test cases* y su ejecución se realiza de forma manual. Sin embargo, se incorpora una herramienta que graba todos los *test cases* ejecutados, además de sus entradas y salidas. Esto posibilita que los *tests* puedan volver a ser ejecutados más delante de forma automática y generar sus respectivos reportes de resultados.

Esta técnica presenta muchas limitaciones debido a la falta de abstracción en los *tests* almacenados: algún cambio en el sistema puede hacer que muchos *tests* guardados fallen. Los *tests* que fallen por esas razones deberán ser ejecutados manualmente y ser grabados nuevamente para su posterior re-ejecución. Así, el problema de automatización en la ejecución de *tests* es solo parcialmente resuelto por esta técnica. Además, la técnica *capture/replay* es generalmente solo utilizada para automatizar *tests* relacionadas a la interfaz gráfica (GUI) del sistema bajo verificación.

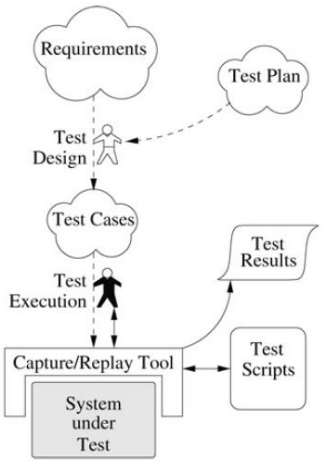


Figura 5: ***Capture/Replay Testing*** **[10]**

Script-based Testing

Busca automatizar la ejecución de *tests*, superando las limitaciones encontradas en el método *capture/replay*. En lugar de que los *tests* generados se ejecuten de forma manual, un programador escribe *scripts* que se encargarán de llevar a cabo *tests* y observar los resultados luego de las ejecuciones. Esto puede abarcar: inicialización del sistema sobre el cual se correrán los *tests*, generación de datos de entrada, registro de respuestas, comparación con resultados esperados y asignación del veredicto (éxito o fracaso, por ejemplo) final de cada *test*.

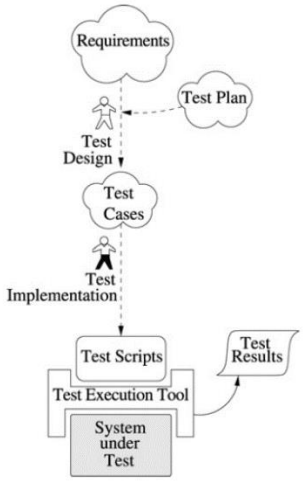


Figura 6: ***Script-based Testing*** **[10]**

Esta técnica resuelve el problema de automatización en la ejecución de *tests*. El problema de este método es que introduce una sobrecarga en el mantenimiento de *scripts* ya que los mismos deben ser modificados ante cualquier cambio de requerimiento y/o implementación. El tamaño total de *scripts* de *test* puede ser tan grande como el sistema a verificar y una misma funcionalidad podría tener más de un *test* asociado por lo que el mantenimiento se vuelve muy costoso. Para superar dicho inconveniente se requiere altos niveles de abstracción en la definición de *scripts*.

Keyword-driven Testing

La idea es aumentar el grado de abstracción de *test cases*; se utiliza un conjunto de *scripts* parametrizados con diferentes valores de datos para cada *test case*. Esto permite que los *scripts* sean más genéricos, reduciendo el problema de mantenimiento. Además, es posible que se utilicen palabras claves que representen diferentes acciones en los *test cases*, por lo que es posible parametrizar datos y acciones. Debido al alto grado de abstracción que presentan los *tests*, su diseño no requiere conocimientos de programación (siempre que un programador se encargue de la implementación de palabras claves).

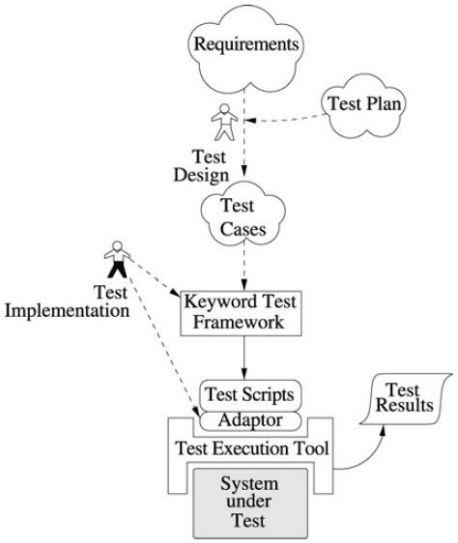


Figura 7: Keyword-driven Testing [10]

Herramientas de Automatización

Para *testing* automatizado orientado a la definición de *tests* de aceptación por parte de los usuarios se usan herramientas como *Fitnesse*[14] o *RSpec*[15], mientras que los desarrolladores que utilizan metodologías ágiles comúnmente usan la familia de herramientas *Open Source xUnit* (como *JUnit*[16] o *VBUnit*[17]). Además, hay herramientas comerciales disponibles. Éstas permiten definir, usando una notación con cierto nivel de formalidad, los *test cases* de manera a que puedan ser ejecutados sobre el *software* bajo prueba.

Pese a las mejoras introducidas por el método *Keyword-driven testing,* la especificación de *tests* y la verificación de la relación entre *tests* y requerimientos del sistema, siguen siendo pasos manuales. El enfoque *Model-based Testing* (MBT) aparece para ofrecer una solución a los problemas pendientes, trataremos estos temas en las siguientes secciones.

Model-Based Testing (MBT)

Model-Driven Development **(MDD)** consiste en un paradigma que utiliza modelos o representaciones abstractas como artefactos primarios para el proceso de desarrollo: la implementación del *software* es generada de forma semi-automática a partir de estos modelos [11].

En [12] se identifican los procedimientos más frecuentes en el ciclo de vida del *software,* y los destacamos a continuación:

- Diseño de la solución mediante la utilización de modelos abstractos, sin la generación del código propiamente: el tiempo que lleva el análisis de la solución podría resultar en pérdidas e inclusive la cancelación del esfuerzo a causa de

[14] **Referencia Fitnesse:** http://fitnesse.org/
[15] **Referencia *RSpec*:** https://www.relishapp.com/rspec
[16] **Referencia *JUnit*:** http://junit.org/
[17] **Referencia *VBUnit***: http://www.vbunit.com/

posibles cambios en los requerimientos y en la arquitectura definida para la solución.

- Codificación directa de los requerimientos funcionales, sin la definición formal de la estructura de la solución: dedicar poco tiempo al análisis general de la estructura de la solución también tiene sus riesgos (en particular, cómo el producto deberá ser extendido y mantenido en el futuro). Una solución implementada sin un análisis y diseño inicial podría requerir que sea necesario realizar re-implementaciones en posteriores lanzamientos del producto. Además, las soluciones complejas pueden resultar en códigos difíciles de mantener a medida que el proyecto evoluciona.

MDD, por su parte, utiliza lenguajes de modelado para proveer un *framework* de alto nivel para la documentación de sistemas de *software*. Permite transformar los modelos (expresados en lenguaje UML, por ejemplo), diseñados de forma independiente de la plataforma destino, usando un conjunto de mapeos entre los modelos y la implementación del sistema o código fuente. Ofrece la posibilidad de definir y documentar la solución y la consecuente generación de artefactos que forman parte de la solución misma. La utilización herramientas que soporten la generación automática de código a partir de modelos facilitaría la sincronización entre los diagramas (diseño de la solución) y el código (implementación de la solución), resultando en que los diagramas terminan siendo representaciones reales de la solución implementada.

***Model-Based Testing* (MBT)** es la automatización de la técnica de *black-box testing*, basada en MDD. Por lo tanto, el enfoque MBT permite la generación automática de *tests* a partir de modelos, y generalmente tiene cuatro etapas [13]:

1. Construcción de un modelo abstracto del sistema bajo verificación: es similar al proceso de definir formalmente el sistema pero con algunas diferencias necesarias para, por ejemplo, clarificar los requerimientos y probar la correctitud.
2. Validación del modelo: este proceso sirve para detectar errores en el modelo. Si algunos errores persisten, podrán ser detectados muy probablemente cuando los *tests* sean ejecutados.
3. Generación de *tests* abstractos[18] a partir del modelo: este paso es generalmente automático pero los *testers* pueden controlar varios parámetros (ej.: qué partes del sistema se verificarán, cuántos *tests* serán generados y criterios de cobertura de *tests*).
4. Refinamiento de los *tests* abstractos, convirtiéndose en *tests* ejecutables: se añaden detalles concretos a partir del modelo abstracto. Esto generalmente es automático, a partir de una función de refinamiento y una plantilla de código definida para cada operación abstracta.

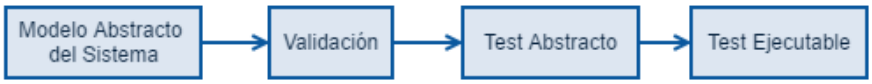


Figura 8: Fases de *Model-Based Testing*

Posteriormente, los *tests* generados pueden ser ejecutados sobre el sistema a fin de detectar fallos (son considerados fallos, aquellos casos en los que las salidas del sistema son diferentes a las esperadas por los *tests*).

[18] Los *tests* abstractos son secuencias de operaciones a realizar generadas a partir del modelo abstracto del sistema. Éstos no poseen los detalles implementativos del sistema por lo que no pueden ser directamente ejecutados.

A diferencia de los métodos vistos en las secciones anteriores, MBT ofrece herramientas para la definición de modelos que permitan la generación automática de *test cases* (y la consecuente disminución del costo de *testing*), posibilitando además la cobertura del modelo de una forma sistemática. Las características mencionadas pueden ayudar a aumentar la calidad y cantidad de *tests* [10].

A continuación, se listan diferentes variables que definen el proceso de generación de *tests* a partir de la técnica MBT [13]:

¿Qué modelar?

El modelo usado para generar los *tests* puede ser un modelo funcional de:

- El sistema bajo *test*: permite predecir las salidas del sistema.
- El entorno del sistema (teniendo en cuenta las formas en las que el sistema será utilizado): se enfoca la generación de *tests* basada en el uso que se le dará al sistema.
- Tanto el sistema como su entorno (el caso más común).

Notación a utilizar

Son muy utilizadas las notaciones para pre y post condiciones, tales como *Z*, *B*, *JML* y *Spec#*. Asimismo, se utilizan las notaciones basadas en transiciones, como diagramas de estado y máquinas de estado.

Control de Generación de *Tests*

Para generar los *tests*, es necesario definir criterios que determinen el alcance de la generación. Dos técnicas son:

- Especificación, además del modelo, de algunos patrones que permiten generar solamente los *tests* que satisfagan dichos patrones.
- Especificación de los criterios de cobertura del modelo, los cuales, determinan qué *tests* interesan.

Generación en línea (*on-line*) o fuera de línea (*off-line*)

En la generación *on-line* se lleva a cabo el proceso de generación en paralelo con su ejecución. Esto facilita el manejo del no determinismo puesto que se puede ver las salidas del sistema, luego de las posibles ejecuciones no deterministas, y realizar los cambios en la generación de *tests* de forma adecuada.

Por su parte, la generación *off-line*, genera los *tests* independientemente de su ejecución. Entre las ventajas de esta técnica se incluye la posibilidad de ejecutar los *tests* repetidamente (facilitando el *regression testing*) y con diferentes entornos o configuraciones.

Seguimiento de requerimientos

Este punto es importante porque es altamente deseable que las implementaciones de MBT incluyan alguna matriz que relacione a cada requerimiento informal con sus *tests* respectivos. Además de validar los requerimientos informales, la matriz puede ser utilizada en la definición de criterios de cobertura de *tests* (en relación a los requerimientos).

Para ello se utilizan técnicas de anotaciones en los modelos, identificando los requerimientos. Dichas anotaciones crean las relaciones entre los requerimientos y los *test cases*, durante el proceso de generación.

Capítulo 2. Revisión del Estado del Arte

Como hemos mencionado en el capítulo anterior, TDD consiste en un conjunto de prácticas para llevar a cabo el proceso del desarrollo de *software* usando *tests* como base. En un entorno de desarrollo ágil presenta ventajas en cuanto a: la simplicidad y limpieza del código resultante, la cantidad de *tests* definidos (que posteriormente pueden ser reutilizados para *tests* de regresión) y la posibilidad de hacer partícipe al cliente en el proceso de definición de *tests* de aceptación.

En [14], por ejemplo, se lleva a cabo un caso de estudio que permite concluir sobre la eficacia de la utilización del proceso TDD en proyectos de tamaño pequeño: se obtuvieron mejores resultados (en términos de calidad del código resultante y de productividad del equipo de desarrollo) utilizando TDD, en comparación al desarrollo utilizando metodologías tradicionales y con enfoques *test-last*. Las conclusiones se realizaron teniendo en cuenta que la cantidad de errores encontrados (luego de ejecutar *tests* unitarios, de integración y de aceptación sobre ellos) en el código desarrollado con el proceso TDD fue mucho menor que la cantidad de errores encontrados en el *software* desarrollado con técnicas tradicionales. Además, la cantidad de horas hombre que llevó el desarrollo usando TDD fue mucho menor en comparación al método tradicional.

El estudio realizado en [5], basado en encuestas dirigidas a profesionales (desarrolladores de *software*, *testers* y otros) de la industria con experiencia trabajando con TDD, concluye mencionando las ventajas de la utilización de TDD. Destacamos las siguientes: (i) potencial para mejorar la calidad del código resultante y la habilidad de afrontar las necesidades de cambio; (ii) permite aumentar la productividad con el paso del tiempo; (iii) el *refactor* permite la reducción en la complejidad del código; (iv) TDD puede reducir de forma significativa la cantidad de errores en el *software*.

En TDD, sin embargo, la generación de los *tests* se realiza comúnmente de forma manual, a partir de la codificación de *scripts* que se encargan de ejecutar los programas y validarlos respecto a las aserciones definidas. De acuerdo a los niveles de *testing* en que estemos trabajando, se puede utilizar herramientas como: *Fitnesse*, *RSpec* o *frameworks* de la familia *x-Unit*. Si bien estas herramientas permiten mantener un conjunto de *tests* que pueden ser reutilizados las veces que sea necesario, el diseño y codificación de *tests* requiere un trabajo considerable: la cantidad de líneas de código de *test* puede fácilmente equiparar (e incluso superar) a la cantidad de líneas de código del programa en sí[19].

Agregamos a esta limitación lo mencionado en [5]: concluyen que la mayor barrera para la utilización de TDD en la industria se debe a la falta de experiencia en TDD, resistencia al cambio de procesos de parte de las organizaciones, aumento en el tiempo de desarrollo, diseño insuficiente de la solución y limitaciones en las herramientas propias del dominio.

Mencionábamos además, las diversas técnicas que surgieron buscando la automatización del proceso de *testing*. En particular, el método *Keyword-driven testing* introduce varias mejoras en cuanto al nivel de abstracción y la posibilidad de reutilización de los *tests*. Sin embargo, la elaboración de *tests* y la verificación de la relación entre *tests* y requerimientos del sistema, siguen siendo pasos manuales. El enfoque *Model-based Testing* (MBT) aparece para ofrecer una solución a los problemas pendientes, permitiendo [10]:

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

- La automatización de la generación de *test cases* (incluyendo los resultados esperados) para reducir costos de diseño.
- Reducción de costos de mantenimiento de *tests.*
- La generación automática de una matriz de trazabilidad que permita establecer una relación entre los requerimientos del *software* y *test cases* asociados.

Debido a las ventajas mencionadas, creemos que la utilización de MBT puede reducir el tiempo requerido para la generación, ejecución y validación de *test cases*, lo cual permitiría a la vez, el incremento de tareas de *testing* durante el desarrollo de *software*.

Esto constituye la base de nuestra propuesta: el desarrollo de *software* basado en *tests* definidos para cada funcionalidad a implementar (siguiendo las prácticas de TDD), apoyado por técnicas y herramientas MBT para facilitar la generación de *tests* y la definición de relaciones entre *tests* y requerimientos del *software*.

En este capítulo, nos planteamos una hipótesis inicial en base a la cual definiremos una propuesta para el desarrollo de *software*, al cual hemos denominado **MoFQA**. Presentaremos además un análisis de trabajos realizados en las áreas de interés, según evidencias halladas en la literatura. Dicho análisis nos llevó a encontrar algunas debilidades y/o limitaciones de MBT en la actualidad.

Hipótesis Planteada

En base a las ventajas de TDD y MBT anteriormente mencionadas, hemos definido la siguiente hipótesis: “El proceso de desarrollo de *software* basado en TDD, apoyado por MBT para la generación de *test cases*, puede mejorar la productividad del equipo de desarrollo en cuanto a tiempo invertido para la elaboración de *tests* y cantidad de *tests* definidos para cada funcionalidad a implementar; con la posibilidad de mejorar, consecuentemente, la calidad del producto final.”.

Además de las evidencias halladas en la literatura sobre las ventajas de los enfoques que deseamos integrar, se tuvieron en cuenta las siguientes bases teóricas para la formulación de la hipótesis mencionada:

- Cuanto antes detectemos fallas en el sistema, menos costoso será solucionar los problemas, ante la menor complejidad existente en el código [1]. Si seguimos de forma rigurosa el proceso TDD, cada funcionalidad a implementar estará guiada por el *test* que deberá pasar. Es más, antes de iniciar la implementación de una nueva funcionalidad, debemos asegurarnos de que todas las funcionalidades previamente incluidas pasen los *tests* definidos.
- Los modelos MDD, apoyados por herramientas eficientes de generación de código, prometen reducir la complejidad del proceso del desarrollo de *software* gracias al nivel de abstracción que ofrecen los modelos. Otra de sus ventajas es que los modelos independientes de la plataforma (PIM) permiten la generación del producto en plataformas diferentes de destino, a partir de un único modelado de origen [11]. En nuestro caso, podemos reutilizar los modelos definidos para implementar *tests* en diferentes plataformas de destino.
- La combinación de TDD con MDD (para definir las unidades de *testing*) puede otorgar al equipo de desarrollo los criterios de aceptación que requiere para el

desarrollo de la solución, consecuentemente reduciendo las dudas sobre los resultados esperados [12].

De esta forma, podemos aprovechar el nivel de abstracción que otorgan los modelos (en este caso, para representar comportamientos del sistema y generar *test cases*) reduciendo además la cantidad de detalles específicos de implementación. Asimismo, el enfoque *test-first* de TDD permitirá reducir los costos del proceso de *testing*.

El objetivo de este trabajo, sin embargo, no será la verificación de la hipótesis antes formulada. Por su amplitud, nos limitaremos a definir un método para el desarrollo de *software* que unifique los enfoques MBT y TDD, aprovechando los conceptos teóricos vistos hasta el momento.

Se espera, además, que el método proponga posibles soluciones a problemáticas aún vistas en el área de *testing* de *software*. En la siguiente sección, presentamos un análisis de los criterios de interés para este trabajo, según evidencias encontradas en la literatura.

Análisis del Estado del Arte

Hemos realizado, como primer paso, una investigación que nos permitió: verificar trabajos publicados, realizar comparaciones entre los resultados recopilados de cada fuente e identificar posibles limitaciones o debilidades que persisten las áreas de estudio de interés.

Para la búsqueda de trabajos, generamos una cadena de búsqueda adaptando las técnicas propuestas en [15]:

1. Teniendo en cuenta los objetivos de este estudio, se identifican los siguientes aspectos:
    - Población: trabajos realizados a la fecha de MBT y TDD orientados o no al entorno Web[20].
    - Contexto: nos interesan tanto los trabajos académicos como aquellos aplicados a la industria. A efectos de los objetivos de este trabajo, prestamos mayor atención a aquellos dirigidos a plataformas Web.
    - Salida (*Outputs*): se consideran los criterios definidos más adelante para el análisis de trabajos, así como los aspectos que conforman las columnas de la **Tabla 1**: Trabajos recopilados para el análisis del Estado del Arte.
    - .
2. Se enumeran las palabras claves, abreviaciones y sinónimos que identifican a:
    - los enfoques que deseamos unificar (MBT y TDD);
    - las metodologías de desarrollo de *software* relacionadas;
    - los tipos o el dominio de *software* en los cuales se enfocaría el estudio.
3. Las cadenas obtenidas se concatenan utilizando los conectores lógicos AND y OR. La cadena de búsqueda obtenida resulta a partir del enlace de las siguientes listas, utilizando el conector lógico **AND**:

[20] A fin de limitar el dominio, nos pareció interesante enfocarnos en el desarrollo de software para plataformas Web 2.0. Realizamos esta elección debido al gran auge de este tipo de aplicaciones, también destacado en [23]: "El *testing* de aplicaciones web es un proceso muy importante ya que es el área de mayor crecimiento en ingeniería de *software*".

- Web **OR** WWW **OR** Internet **OR** World-Wide Web **OR** Software **OR** Software Verification **OR** Software Development **OR** Web Development **OR** Verificación de Software **OR** Pruebas en el Software **OR** Desarrollo de Software **OR** Desarrollo Web
- ((Model-Based Testing **OR** Model-Driven Testing **OR** Model-Driven Development automated testing **OR** MBT **OR** MDT **OR** MDD automated testing) **AND** (Test-Driven Development **OR** Test first **OR** Extreme testing **OR** Xtreme testing **OR** TDD **OR** XP testing))

  OR

  (Model-Based Testing **OR** Model-Driven Testing **OR** Model-Driven Development automated testing **OR** MBT **OR** MDT **OR** MDD automated testing)

La búsqueda realizada nos llevó a una diversidad de trabajos[21] realizados en el contexto de MBT, a partir de los cuales recopilamos algunas limitaciones. La **Tabla 1**: Trabajos recopilados para el análisis del Estado del Arte.

muestra los trabajos seleccionados para el análisis que se presenta en esta sección.

A continuación, se mencionan los criterios definidos para el análisis del estado del arte, mencionando la importancia que se vio en cada uno de dichos criterios:

a. Adaptación del desarrollo de software a los pasos definidos por TDD: Como el objetivo principal de nuestra propuesta consiste en la integración de los enfoques TDD y MBT, buscamos evidencia en la literatura sobre herramientas y/o propuestas metodológicas que integren ambos aspectos.

b. Relación de *tests* con requerimientos del *software*: Otro punto importante, también considerado en la definición de nuestra propuesta, es la capacidad que ofrecen las técnicas y herramientas existentes para establecer una relación entre *tests* y requerimientos del *software*. La existencia de dichas relaciones permitirá verificar más fácilmente qué requerimientos tienen asociados uno o más *tests* (y cuáles no, por lo tanto, no se pueden verificar).

c. Soporte de herramientas: La existencia de herramientas adecuadas que permitan llevar a cabo todos los pasos envueltos en el proceso de *testing* (definición y generación de *tests*, ejecución y validación, etc.) es un punto determinante para la aplicación de las prácticas definidas tanto en TDD como en MBT.

d. Complejidad de definición de *tests*: Se encontró evidencia de que en la práctica aún resulta complejo definir *tests* en MBT(a pesar de la abstracción otorgada por los modelos), razón por la cual resulta muchas veces inaplicable. Como es un punto limitante para la aplicación de MBT en la industria, nos pareció importante incluirlo entre los criterios de análisis.

e. Niveles de automatización: Nos interesa conocer qué pasos del proceso de *testing* son llevados a cabo de forma manual y cuáles han logrado ser automatizados (total o parcialmente).

f. Independencia entre modelos de *test* e implementación: La descripción de aspectos del *software* (funcionalidad, interfaz, interacciones entre módulos, etc.) a

[21] En la recopilación de trabajo, hemos otorgado mayor prioridad a estudios secundarios, así como a los más recientes, ya que éstos nos permiten tener un panorama general del estado del arte.

partir de modelos, posibilita la generación tanto de *tests* como del código de implementación del producto final a partir de un solo modelado común. En este sentido, se verificará: (1) cómo son utilizados los modelos por las herramientas analizadas en los trabajos incluidos; (2) qué tan independientes resultan los modelos de *test* de los artefactos que representan la implementación final.

g. **Adecuadas para *web testing*:** A modo de limitar el dominio de aplicación de este trabajo, nos enfocaremos en el desarrollo de aplicaciones *web*. Al respecto tendremos en cuenta algunas características relevantes para el *testing* de este tipo de aplicaciones, por lo que hemos incluido algunos puntos encontrados en los trabajos analizados.

Título	Tipo de Informe	Año	Tipo de Estudio	Objeto de Estudio
Test First Model-Driven Development [16]	Tesis de Msc. en Informática	2012	Presentación de una herramienta y validación mediante casos de estudio.	Se presenta la herramienta TFMDD que permite generar *tests* y *software* a partir de un modelado único, basado en restricciones a partir de la definición de *pre* y *post* condiciones.
Alignment of requirements specification and testing: A systematic mapping study [17]	Artículo de Conferencia	2011	Mapeo sistemático, incluye el análisis de 35 trabajos seleccionados.	Analiza la relación que guarda el proceso de *testing* con respecto a las especificaciones de requerimientos funcionales y no funcionales del *software*.
A survey on model-based testing approaches: a systematic review [18]	Artículo de Workshop	2007	Revisión sistemática de técnicas MBT, se analizan 78 trabajos seleccionados.	Se realiza comparaciones con respecto a aspectos como: modelos utilizados, soporte de herramientas, criterios de cobertura que soportan, niveles de automatización de pasos de *testing*, complejidad. Sus resultados permiten conocer qué tipos de *tests* están mayormente cubiertos y cuáles son algunas limitaciones pendientes en el área.
Industrial-strength model-based testing-state of the art and current challenges [19]	Artículo de Workshop	2013	Estudio primario. Utilizando una herramienta existente (RT-Tester) como referencia, se describen aspectos de MBT aplicados en la industria.	Utilizando la herramienta RT-Tester, se ilustran aspectos de MBT en la práctica y métodos que dieron buenos resultados en *testing* aplicado a problemas del mundo real. El campo de aplicación considerado es: sistemas embebidos de tiempo real aplicados a la aviación, industria automotriz y vías férreas. Las técnicas y métodos presentados pueden ser utilizados como *benchmarks* para este campo de aplicación.
Towards Testing Future Web Applications [20]	Artículo de Conferencia	2011	Se presenta una metodología que implementa nuevos paradigmas para el *testing* de *webs* futuras.	En el marco del proyecto *FITTEST* se realiza un análisis de las limitaciones de las técnicas de *testing* actuales para su aplicación sobre sistemas de plataforma *web*. Se presenta nuevos métodos y paradigmas que pueden ayudar a hacer frente a los problemas mencionados.
Integrating Model-Based Testing in Model-Driven Web Engineering [21]	Artículo de Conferencia	2011	Estudio primario. Se presenta una técnica práctica para aplicar MBT, reutilizando modelos de desarrollo para la generación de *tests*.	Discusión sobre las ventajas y desventajas de la reutilización de modelos para generación de *tests*. Se presentan además detalles de *web testing*. Finalmente, se implementa una técnica para la aplicación de MBT mediante un ejemplo práctico.
Model-Based Testing of Community-Driven Open-Source GUI Applications [22]	Artículo de Conferencia	2006	Estudio primario. Descripción de un proceso ideado para llevar a cabo el *testing* de aplicaciones elaboradas por una comunidad de desarrollo.	Se describen los problemas de *testing* en aplicaciones GUI, especialmente en comunidades de desarrollo. Se propone una técnica para llevar a cabo el *testing* de comunidades *Open Source* de desarrolladores, interconectados por la WWW.
Model Based Testing in Web Applications [23]	Artículo de *Journal*	2014	Documento informativo, recopila varios trabajos y se analiza los modelos que utilizan para generar *tests* de aplicaciones *web*.	Discusión de conceptos principales y modelos utilizados para generar *tests* para aplicaciones *web*, se acompaña la discusión con un caso de estudio ilustrativo.

Tabla 1: Trabajos recopilados para el análisis del Estado del Arte.

Las limitaciones halladas en los trabajos seleccionados, según los criterios de análisis definidos, se detallan en las siguientes secciones.

Adaptación del desarrollo de software a los pasos definidos por TDD

La naturaleza *test-first* de TDD no es adoptada por la mayoría de las herramientas de generación de *tests*. Éstas se concentran, sin embargo, en un código de producción ya implementado [16].

Relación de *tests* con requerimientos del *software*

En [17] se presenta un mapeo sistemático de trabajos con el objetivo de verificar qué tan vinculados se encuentran el *testing* y los requerimientos del *software* (funcionales y no funcionales). Destaca la importancia de la relación entre la definición de requerimientos y el *testing*: (1) un fuerte vínculo entre ellos mejorará las salidas del proceso de desarrollo de *software*; (2) ayuda a encontrar posibles errores de forma anticipada; (3) desde la perspectiva de la dirección del proyecto, ayudaría a obtener un plan de *testing* más acertado, consecuentemente reduciendo los costos del proyecto y las estimaciones de tiempo/tareas.

En varios trabajos se lleva a cabo la generación de *tests* para programas ya implementados. Ante esta situación, [16] menciona que el *testing* de código ya existente (o de modelos derivados de código ya existente) puede crear desconexiones entre requerimientos y los *tests*.

Una minoría de los trabajos encontrados incluye aspectos no funcionales del *software* en el proceso de *testing*. Este punto es destacado por el estudio sistemático presentado en [18], se mencionan algunas razones: (1) los usuarios no siguen un mismo patrón de comportamiento para la ejecución de una funcionalidad de *software* por lo que se complica la definición de *tests* para estos aspectos; (2) algunas características tales como flujo en red, usabilidad y seguridad no poseen una forma clara de definición. De los trabajos analizados en [18], solo el 8% tiene en cuenta algunos aspectos no funcionales (ejemplo: control de acceso, medidas de *performance*, seguridad, usabilidad, confiabilidad). En [17] se vuelve a verificar que el enfoque principal de las propuestas y soluciones MBT está sobre los requerimientos funcionales, mientras que una notable minoría incluye algunos aspectos no funcionales.

Además, es importante destacar que entre los trabajos hechos sobre la relación de requerimientos y *tests*, se ha depositado mucha atención a la trazabilidad. Esta permite que se pueda determinar qué requerimientos se han cubierto (por qué *tests*) y de qué forma los *tests* generados cubren dichos requerimientos [17].

Soporte de herramientas

Se debe utilizar varias herramientas para llevar a cabo el proceso de desarrollo de *software* de la mano de MBT. Se necesita: herramienta de modelado para definición de *tests* (y, en algunos casos, para la definición del sistema a implementar), generador de *tests*, entorno de desarrollo, herramienta para la ejecución y validación de *tests*. En la mayoría de los casos observados, se tienen herramientas diferentes para cada tarea y la integración entre dichas herramientas no es un trabajo sencillo.

En [18] se destaca la falta de integración de técnicas MBT al proceso de desarrollo de *software*, mencionando que puede deberse en gran parte a la falta de integración de herramientas que soporten todos los pasos envueltos en el desarrollo de *software* (incluyendo herramientas MBT para definición de *tests*).

Complejidad de definición de *tests*

En cuanto a la dificultad que conlleva la aplicación de MBT, [18] destaca que existe aún la necesidad de reducir la complejidad: el *tester* hoy en día debe tener conocimientos sobre los lenguajes de modelado, la definición de los criterios de cobertura, formatos de salida generados, entre otros.

Además, [19] recalca que el modelado de *tests* sigue siendo un desafío; al respecto menciona:

- Para sistemas complejos, los modelos necesitan abstraer una gran cantidad de detalles, de otra manera los modelos resultarían inmanejables.
- Las habilidades necesarias para el modelado de *tests* son mucho mayores que las requeridas para la escritura de procedimientos de *tests*.

En cuanto a lenguajes de modelado, la mayoría utiliza notaciones no-UML. Los resultados publicados por [18] demuestran: el 46% de los trabajos analizados usan notaciones distintas de UML y cubren *tests* basados en requerimientos, el 31% de los trabajos usan notaciones no UML y cubren *tests* estructurales (basados en la arquitectura, componentes, interfaces y módulos del sistema). Finalmente, solo un 23% de los trabajos analizados utiliza notaciones UML para el modelado.

Niveles de automatización

En cuanto a la facilidad otorgada por las herramientas actuales de *testing*, [20] recalca que aún se requiere mucho trabajo manual. En la práctica, se realiza de forma manual: la inferencia de modelos intermedios, la generación de secuencias de *tests*, la selección de datos de *test*, priorizaciones, la predicción de valores esperados (definición de oráculos[22]), análisis y pre-selección de *test cases*.

Independencia entre modelos de *test* e implementación

Es necesario independizar los modelos utilizados para la generación de *tests* de los modelos o código de implementación para evitar replicar los errores existentes en la implementación [10]. Por ello se recomienda no reutilizar los modelos de desarrollo para llevar a cabo MBT. Pese a esto, algunos trabajos revisados utilizan el mismo modelo para generación de *tests* y del sistema final con la finalidad de reducir los costos en la definición de *test cases*.

Consideramos que los trabajos que realizan ingeniería inversa presentan la misma problemática: si los *tests* se definen a partir del código de implementación, no sería posible detectar ciertos tipos de errores (por ejemplo, falta de adecuación a los requerimientos del *software*, incompletitud en la solución, etc.).

Otros autores agregan que en algunos casos, sin embargo, la reutilización de modelos puede resultar beneficiosa: para verificar la correctitud de la herramienta de generación de código [21].

[22] Los oráculos (*test oracles*) es el nombre que se suele dar a las especificaciones y ejemplos de resultados esperados luego de llevar a cabo un *test*. [25]

Adecuadas para *web testing*

El *testing* de aplicaciones *web* es un proceso muy importante ya que es un área de gran crecimiento en ingeniería de *software* [23]. Las aplicaciones *web* están basadas en una arquitectura de múltiples capas y es necesario que el proceso de *testing* sea aplicado en cada una de las fases, pero de forma automática para obtener resultados más acertados.

Las aplicaciones *web* usualmente son de naturaleza heterogénea y sus características principales son [23]: interoperabilidad, tolerancia a fallos, escalabilidad, confiabilidad y transparencia ya que soportan un entorno distribuido. Debido a la variedad de usos y a la constante evolución de aplicaciones *web*, en [20] se anticipan algunas características: "En el futuro estas aplicaciones serán interconexiones complejas entre servicios, aplicaciones, contenido, multimedia; todos ellos con mayor información semántica. La adaptabilidad y autonomía de estas aplicaciones mejorará la experiencia del usuario permitiendo cambios dinámicos del lado cliente y servidor. Algunas tecnologías claves que contribuyen al desarrollo de estas aplicaciones son: *web* semántica, *web* 2.0 y aplicaciones RIA".

En [20] se analiza las características de aplicaciones web futuras y las técnicas de *testing* que serán necesarias para afrontar la complejidad que dichas aplicaciones presentarán. Se espera que las aplicaciones *web* del futuro cuenten con las siguientes características: capacidad de auto-modificación, comportamiento autónomo, observabilidad baja[23], comunicación asíncrona, comportamiento dependiente del tiempo y la carga, espacios de configuraciones muy amplios y escalas ultra largas. La utilización de técnicas de *testing* actuales, no puede asegurar la calidad de las aplicaciones respecto a las características mencionadas. Además, la cantidad de recursos requerida es muy grande.

Con respecto a la interfaz de usuario (GUI, por sus siglas en inglés, *Graphical User Interface*), incluimos los aspectos mencionados en [22], sobre las complicaciones de realizar *testing* de aspectos GUI:

- La cantidad de interacciones posibles con un GUI es enorme. La gran cantidad de estados posibles resulta en una cantidad grande de permutaciones de entrada, requiriendo *testing* extensivos. Un problema derivado es la determinación de la cobertura de un conjunto de *test cases*.
- Un aspecto importante es la necesidad de realizar verificaciones en el estado del GUI con cada paso de la ejecución del *test*. La ejecución de un *test case* debe ser finalizada apenas se detecte un error puesto que un estado incorrecto del GUI puede llevar a una ventana/pantalla no esperada, por ejemplo.
- Si no se introducen verificaciones en cada uno de los pasos, puede resultar difícil la identificación de la causa real del error.
- En cuanto al *regression testing*: el mapeo de entradas y salidas no permanece constante luego de ejecuciones sucesivas (y varias versiones del *software*).

MBT resulta muy efectivo con aplicaciones muy dependientes del estado interno o de la persistencia de datos [20]. Como las aplicaciones *web* tienden a ser muy dependientes de su estado, el entorno, contexto y usuarios, MBT aparece como una opción apropiada para ellas.

[23] Los resultados generados por una aplicación web puede tener varias formas (páginas HTML, datos almacenados en una base de datos, mensajes enviados a otras aplicaciones y servicios).

Sin embargo, es complicado definir un modelo genérico para todos los posibles entornos de ejecución (para cada configuración diferente, en cualquier momento y con requerimientos variables de carga). Como consecuencia, los modelos para estas aplicaciones deben ser automáticamente actualizados y refinados [20].

Otros datos recopilados, que pueden resultar de interés:

En [18] se encontró una mayor cantidad de trabajos dirigidos a *tests black-box* (basados en los requerimientos de entrada/salida, aproximadamente un 59%), mientras que una menor cantidad de trabajos se enfoca en *tests white-box* (representando un 41% de los trabajos analizados).

En cuanto a niveles de *testing*, [18] revela: el 62% de los trabajos analizados permiten realizar *system testing*, el 22% *integration testing*, el 10% *unit testing*, mientras que la minoría restante permite realizar *regression testing*. El trabajo concluye, además, que el *testing* a nivel de módulos (*tests* unitarios) no es un nivel de abstracción muy usual en MBT, puesto que a ese nivel, son mejores otras técnicas que soportan *testing* estructural.

Bases para la propuesta

Para los propósitos de este trabajo, de las limitaciones mencionadas en la sección anterior, se destacan las siguientes:

- Las implementaciones existentes de MBT no toman ventaja completa de TDD. Se observan las problemáticas:
    - La mayoría de las herramientas no refleja la naturaleza *test-first* de TDD. Se generan los *tests*, en cambio, para un *software* (o parte de él) ya existente.
    - En algunos trabajos, sin embargo, se recurre a la generación de *test cases* en conjunto con el sistema mismo a verificar (de manera a aplicar de forma eficiente las técnicas de TDD). Si bien esta técnica aprovecha las ventajas ofrecidas por TDD, la implementación del *software* final está totalmente ligada a la generación de sus *tests*. Nos interesa independizar ambos procesos (generación del sistema final y generación de *tests*) de forma a otorgar mayor flexibilidad al proceso de desarrollo del *software* y obtener mayor redundancia en los requerimientos del sistema a implementar.
- Soporte de herramientas: como un obstáculo para la implementación de MBT en la práctica, aparece la falta de herramientas integradas que faciliten los pasos envueltos en el proceso de *testing* y desarrollo.
- Se requiere minimizar los conocimientos previos necesarios para la implementación de técnicas actuales de MBT: lenguaje de modelado, criterios de cobertura de *tests*, herramientas de soporte. Con esto se destaca la necesidad de minimizar la complejidad existente para la aplicación de las soluciones actuales.
- Es necesario generar *tests* más acordes a los requerimientos del sistema y no solo en base al funcionamiento del mismo.

En el siguiente capítulo se define la propuesta de este trabajo para hacer frente a las limitaciones destacadas en el párrafo anterior.

Capítulo 3. MoFQA: Una Propuesta para el Desarrollo de *Software* basada en TDD

En la investigación realizada a lo largo de este trabajo, no hemos hallado una propuesta que defina prácticas para el uso de técnicas MBT a lo largo del proceso TDD para el desarrollo de *software*. En este capítulo presentamos primeramente una propuesta que define prácticas para el desarrollo integrando ambos enfoques: TDD utilizando herramientas MBT para la generación automática de *tests*. Más adelante, presentamos una herramienta desarrollada, como parte de este trabajo, para la generación de *tests* unitarios y de aceptación.

Así, la propuesta de este trabajo, a la cual denominamos **MoFQA** o ***Model-First Quality Assurance***, se compone de dos elementos principales:

1- Un método propuesto (serie de pasos y prácticas definidas) para el desarrollo de *software* teniendo en cuenta el proceso de *testing*.

2- Herramientas desarrolladas como parte de este trabajo: (i) para el modelado de requerimientos por parte de usuarios finales; (ii) perfiles UML para la definición de *tests* de aceptación y unitarios abstractos; (iii) reglas de transformación para convertir los *tests* abstractos a ejecutables. Las herramientas propuestas se limitan a un dominio específico de aplicación: desarrollo de aplicaciones Web 2.0 donde la utilización de metodologías ágiles para el desarrollo resulta factible.

Es importante aclarar que ambos componentes de MoFQA son independientes: no se requiere la utilización de las herramientas que presentaremos para seguir el método de desarrollo propuesto (y viceversa). El método propuesto puede ser aplicado a otros dominios (no limitándose solo a aquellos orientados al desarrollo *web*), en los cuales sea aplicable el desarrollo basado en metodologías ágiles.

MoFQA como propuesta para el desarrollo de *software*

A continuación, definimos un método para el desarrollo de *software* siguiendo el proceso TDD. Dicho método propone la incorporación de técnicas y herramientas MBT a este proceso pues se desea facilitar la tarea de *testing* mediante el uso de modelos que permitan la generación automática de código de *tests*. La **Figura 9** describe el flujo de pasos a seguir de acuerdo al modelo propuesto. Es posible verificar que el método exige seguir el enfoque *test-first* de TDD, a partir de la utilización de modelos definidos tanto por el usuario final como por el desarrollador del *software*.

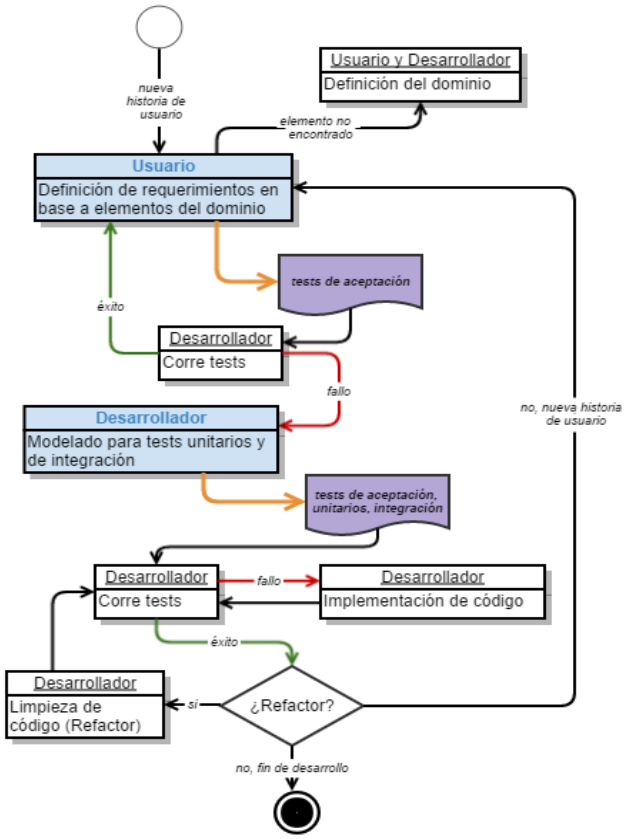


Figura 9: Proceso de desarrollo de software definido en la propuesta

En conjunto, el usuario y el desarrollador definen los elementos del dominio del software a implementar, agregando cada elemento en la medida que sea necesario. Ante una nueva funcionalidad (que puede ser descrita a partir de una historia de usuario), el usuario primeramente define los requisitos para su implementación (que derivarán a los tests de aceptación). Dicha definición puede ser realizada utilizando los modelos proveídos por MoFQA o por otra herramienta de modelado que permita la generación automática (o semi-automática) de tests. A partir de ahí, es posible derivar los tests de aceptación de forma automática, los cuales son ejecutados y validados por el desarrollador. El éxito en la ejecución puede significar que la funcionalidad ya fue implementada o que hubo algún error en la definición de los tests. Ante un fallo, el desarrollador pasa a ampliar el modelado, definiendo tests unitarios para la funcionalidad actual (y posiblemente algunos tests de integración para verificar la interacción entre los módulos implementados y a implementar). A partir de ahí, se derivan los tests ejecutables de forma automática y se sigue el proceso TDD convencional hasta pasar todos los tests.

Es importante destacar que la definición de todos los tipos de tests se realiza a partir del mismo conjunto de elementos del dominio. Además, se requerirá que la implementación misma del sistema a verificar esté basada en el mismo dominio (caso contrario no podrá pasar los tests). Intentamos así poner en práctica los principios de DDD (Domain Driven Design)[24] buscando mejorar la comunicación entre el usuario y desarrollador en la transmisión de conocimientos del dominio del software a implementar. Como se menciona en [24]: “Un

[24] DDD (*Domain-Driven Design*) consiste en una serie de patrones para la construcción de *software* prestando especial atención a su dominio y a la lógica de negocios, a partir de la definición de un modelado común definido en continua colaboración entre desarrolladores y expertos del dominio. Más información en [24].

proyecto se enfrenta a problemas serios cuando los miembros del equipo no hablan un lenguaje común a la hora de discutir sobre el dominio del software. Es por dicha razón que el modelo debe centralizar el lenguaje a ser utilizado. En todas las comunicaciones deberá utilizarse el vocabulario definido en los modelos de forma consistente, inclusive en el código final."

La propuesta se basa en las prácticas de desarrollo ágil, en la que se promueve la participación activa y constante del usuario final en la implementación de cada nueva funcionalidad.

Herramientas MoFQA

Con el análisis del estado del arte vimos que uno de los problemas en la aplicación de las prácticas MBT en la industria es la falta de integración entre las herramientas involucradas en el proceso de *testing*. Es necesario utilizar herramientas diferentes para el: modelado, generación y ejecución de *tests*. Esta motivación nos llevó a desarrollar una serie de herramientas integradas orientadas al *testing* de sistemas basados en plataformas web. Inicialmente, nos hemos enfocado en la definición y generación de *tests* de aceptación y unitarios.

La definición y generación de *tests* utilizando las herramientas proveídas por nuestra propuesta conlleva los pasos ilustrados en la **Figura 10**.

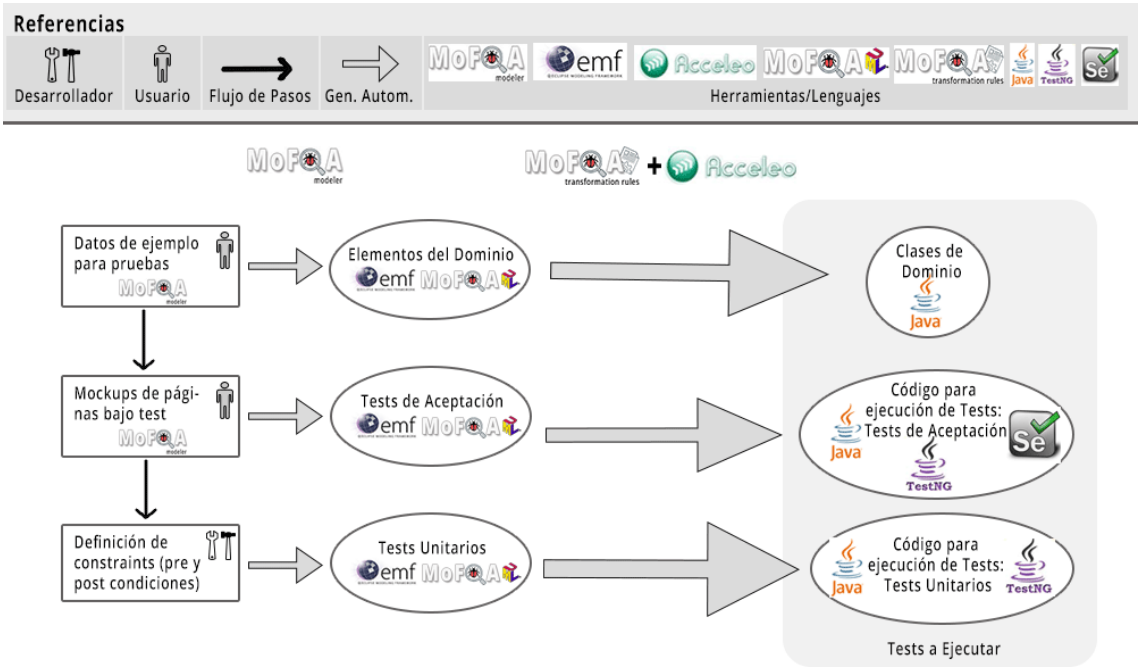


Figura 10: Flujo de pasos para la definición de *tests* con MoFQA.

Los principales actores involucrados en este proceso son el desarrollador y el usuario final. El usuario final dispone de una herramienta a la cual denominamos **MoFQA Modeler** que le permitiría definir los requerimientos que deberán ser satisfechos en los *tests* de aceptación. **MoFQA Modeler** genera los *tests* abstractos a partir de las definiciones del usuario y los representa como modelos EMF[25]. El desarrollador puede enriquecer los modelos que representan los *tests* abstractos (para agregar *tests* unitarios especificando pre y post

[25] http://www.eclipse.org/modeling/emf/

condiciones para la ejecución de cada método a implementar o implementado en el código del sistema a verificar) utilizando los perfiles UML (exportados en formato EMF) definidos como parte de esta propuesta.

Los modelos generados pueden ser importados a *Eclipse*[26] donde, en conjunto con la herramienta *Acceleo*[27] y las reglas de transformación, se generarían los *tests* ejecutables (unitarios y de aceptación). Los *tests* resultantes están definidos en el lenguaje *Java*[28], utilizando los *frameworks* de *testing TestNG*[29] y *Selenium*[30]. Como agregado, las reglas también definen transformaciones de los elementos del dominio (entidades y relaciones) del sistema a implementar, generando una clase *Java* para cada entidad.

En esta sección, presentamos las herramientas propuestas y ejemplificaremos con dos casos de aplicación.

Casos de Ejemplo

A modo de ejemplificar la utilización de las herramientas que presentaremos, utilizaremos dos casos de ejemplo. Supondremos que deseamos llevar a cabo el desarrollo de:

1- Un sistema de Biblioteca Virtual

2- El portal web existente: Amazon.es

A continuación, se listan los requerimientos definidos para cada ejemplo. En las siguientes secciones veremos cómo podemos utilizar las herramientas **MoFQA** para definir los requerimientos de ambos ejemplos y generar los *tests cases* relacionados.

Ejemplo 1: Biblioteca Virtual

Se desea desarrollar un sistema de Biblioteca Virtual con plataforma web mediante el cual usuarios registrados pueden visualizar ítems (libros, revistas, CDs y DVDs) disponibles (o no) para su préstamo. Definiremos los *tests* para las siguientes funcionalidades:

1- La página principal debe cargar con una tolerancia máxima de 10 segundos y debe tener el título “Items en Biblioteca”.

2- Se debe desplegar una lista de ítems disponibles para el préstamo y una lista de ítems no disponible. Para cada ítem se muestra los datos: autor/es, título.

3- Si se solicita el préstamo de un ítem no disponible, se debe mostrar uno de los mensajes “Libro no disponible”, “Disco no disponible” o “Revista no disponible” según corresponda.

4- La solicitud de préstamo de un ítem disponible, redirecciona a una página que realiza el préstamo y emite el mensaje “¡El disco/libro/revista ha sido reservado para usted!”.

5- Se debe permitir agregar nuevos ítems a la biblioteca.

---

[26] https://www.eclipse.org/
[27] https://www.eclipse.org/acceleo/
[28] https://www.java.com/es/
[29] http://testng.org/doc/
[30] http://www.seleniumhq.org/

Ejemplo 2: Funcionalidades de Amazon.es

Definiremos los *tests* para una plataforma web existente: Amazon.es. Supondremos que deseamos verificar los siguientes requerimientos:

1- Un usuario no autenticado debe ver los siguientes elementos (resaltados en la **Figura 11**) en la *Home*:

   a. Enlace con el texto "Todos los departamentos".

   b. Botón/enlace para inicio de sesión.

   c. Texto "Bienvenido" en algún lugar de la página principal.

   d. Mensaje "Hola, Identifícate".

Figura 11: Visualización de elementos definidos en el requerimiento 1 del ejemplo Amazon.es.

2- Un *click* sobre el enlace "Todos los departamentos" en la *Home* nos lleva a una nueva página con la lista de todos los departamentos y sub-departamentos de la tienda. En la **Figura 12** se muestra la lista tal y como aparece hoy en Amazon.es.

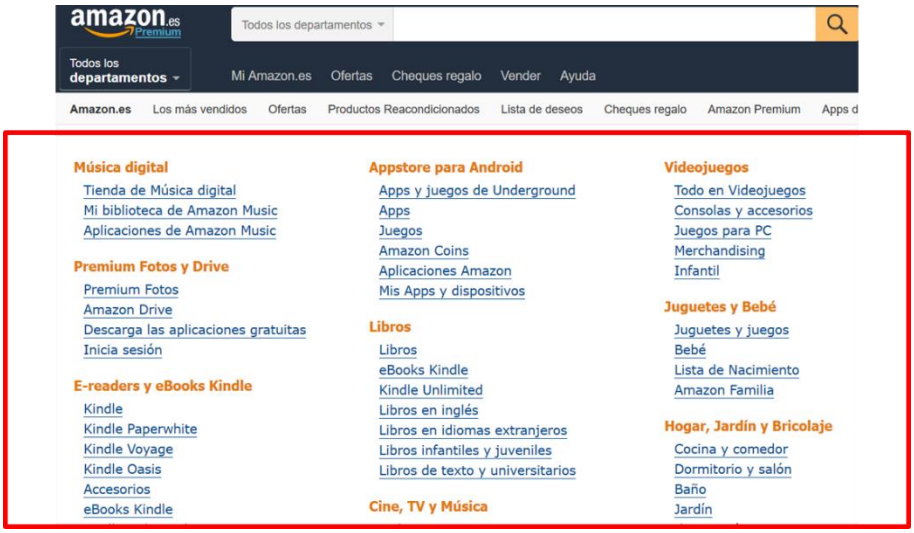


Figura 12: Lista de departamentos y sub-departamentos en Amazon.es

3- Un *click* sobre la opción de inicio de sesión en la *Home*, redirecciona a una página con el formulario que solicita los datos de *login*. La página debe contar con los elementos (resaltados en la **Figura 13**):

a. Texto “Iniciar sesión”.

b. Formulario con campos: e-mail y contraseña.

c. Botón con el texto “Iniciar sesión”.

Figura 13: Formulario de Inicio de Sesión con los elementos del requerimiento 3 del ejemplo Amazon.es.

4- En el formulario anterior, un usuario no válido que intente iniciar sesión debe visualizar el mensaje de error “Datos incorrectos”. El texto “Iniciar Sesión” cambia por “Inténtalo nuevamente”.

5- Un usuario válido debe ser redireccionado a la *Home*, en la cual se visualizará el mensaje “Hola” seguido por el nombre del usuario.

Perfiles UML para la definición de *tests* abstractos

MoFQA propone un perfil UML para el modelado de *tests* abstractos por parte del usuario final y el desarrollador (ambos actores considerados en el proceso de desarrollo del *software* definido previamente). Cabe destacar, que este perfil está orientado directamente a la definición de *tests* para aplicaciones Web. A continuación, presentaremos los elementos definidos en el perfil UML.

La **Figura 14** muestra el perfil UML[31] (***Acceptance Criteria***[32]) definido. Está dividido en cuatro grupos principales:

1- Data Provider: los datos de entrada para las pruebas pueden ser modelados utilizando estos elementos.

2- Domain Specification: permite describir elementos del dominio (entidades y relaciones) de la aplicación a probar.

3- Content Specification: para la definición de componentes que deberán estar visibles en los navegadores Web.

4- Constraint Specification: cada unidad funcional a ser codificada por el desarrollador puede estar asociada a una serie de pre y post condiciones que deben

[31] En la figura, las clases en amarillo son estereotipos definidos en el perfil, mientras que las clases de color naranja son meta-clases extendidas por los estereotipos. Las enumeraciones aparecen en color verde.

[32] Los modelos que pueden ser definidos utilizando este perfil están orientados a la descripción y generación de *tests* de aceptación en su gran mayoría. Se definió un pequeño subconjunto (grupo *Constraint Specification*) que puede ser utilizado para modelar *tests* unitarios abstractos (mediante la definición de pre y post condiciones para la ejecución de métodos).

cumplirse antes y después de su ejecución. Estas condiciones son definidas utilizando los elementos de este grupo.

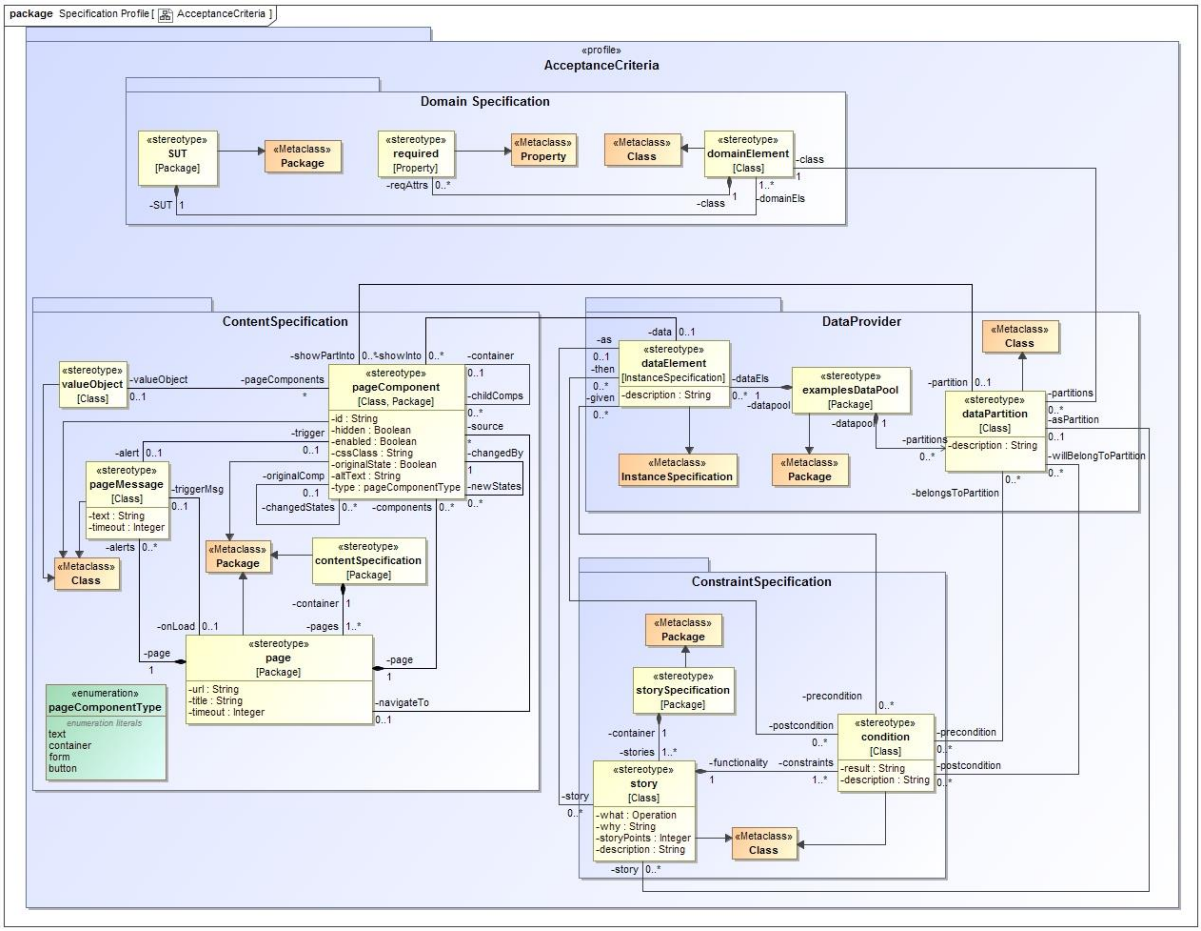


Figura 14: Perfil UML definido en la propuesta

La **Figura 15** muestra los elementos definidos en el grupo ***Data Provider***. Éstos tienen la finalidad de representar los datos de prueba (o ejemplos) para la ejecución de *tests cases*.

Los datos de prueba se definen en paquetes estereotipados como *<<examplesDataPool>>* y pueden representarse de dos formas:

- A través de instancias de los elementos del dominio (*dataEls*): mediante la utilización de elementos *<<dataElement>>*.
- Como clasificaciones o particiones de datos (*partitions*): cada partición es definida por un elemento *<<dataPartition>>* que especifica el valor que debe tener uno o más atributos de una clase del dominio[33].

Los elementos definidos en un paquete *<<examplesDataPool>>* pueden ser referenciados por otros elementos del modelo: de esta forma se definirán los datos de prueba a utilizar en los *tests*.

[33] El concepto de *dataPartition* fue inspirado en el elemento <<dataPartition>> definido en el perfil *TestData* de UTP (*UML Testing Profile*) [4].

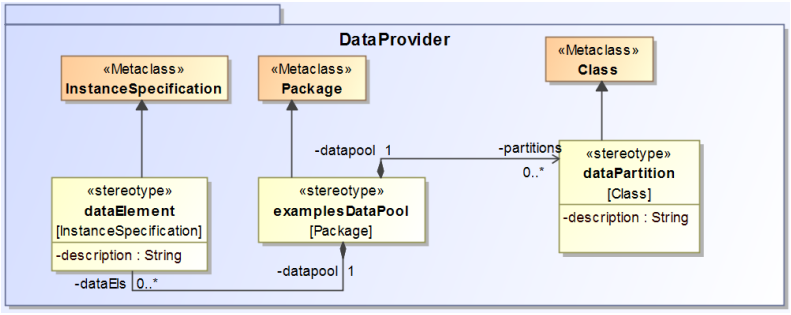


Figura 15: Elementos definidos en el grupo *Data Provider*

Para modelar los elementos del dominio (entidades y relaciones) se definió el grupo ***Domain Specification***, en la **Figura 16** puede verse sus elementos. Los elementos del dominio pueden ser definidos dentro de paquetes <<*SUT*>>[34], el cual puede contener clases y enumeraciones.

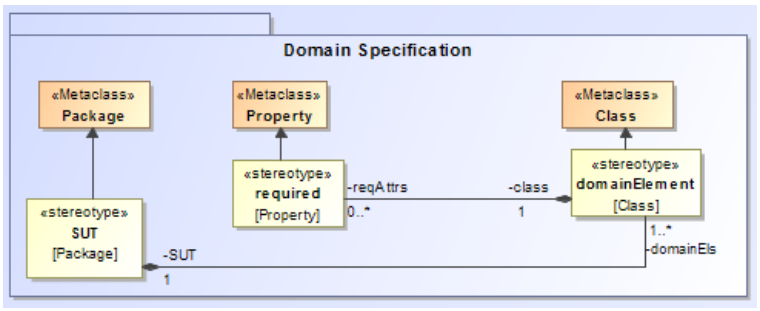


Figura 16: Elementos del grupo *Domain Specification*.

Las clases de tipo <<*domainElement*>> constituirán las entidades que forman parte de la lógica de negocio del sistema a verificar. Estos elementos pueden tener asociaciones entre sí, operaciones y atributos. A su vez, un atributo de cada <<*domainElement*>> puede ser de tipo <<*required*>> (requeridos u obligatorios) o no.

Además, dentro de un paquete <<*SUT*>>, como parte de especificación del dominio del sistema, es posible definir particiones de datos para cada <<*domainElement*>>. Esta relación puede verse en la **asigna** valores a uno o más de sus atributos. De esa forma es posible formar grupos de datos de acuerdo a los valores que tengan ciertos atributos.

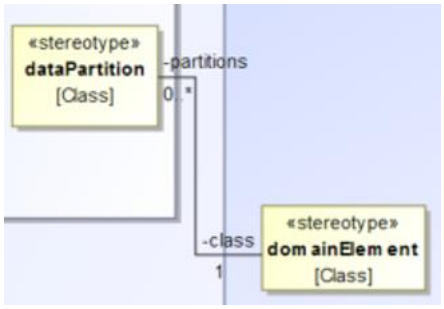


Figura 17. Una partición de datos definida en un <<*domainElement*>> asigna valores a uno o más de sus atributos. De esa forma es posible formar grupos de datos de acuerdo a los valores que tengan ciertos atributos.

[34] **SUT:** siglas del inglés *System Under Test*, se refiere al sistema sobre el cual se ejecutarán los *tests*.

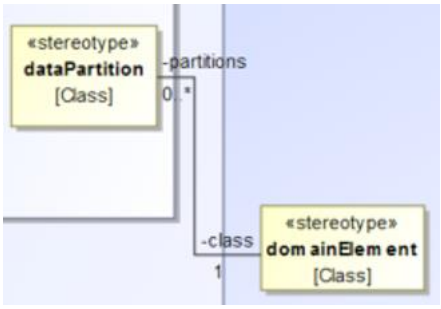


Figura 17: Relación entre <<dataPartition>> y <<domainElement>>

El grupo de elementos ***Content Specification*** tiene la finalidad de permitir la descripción de los componentes que deberán estar visibles en el navegador Web como resultado de ciertas acciones. Los elementos definidos pueden verse en la **Figura 18**. Las páginas a verificar y el contenido que se buscará en ellas debe ser especificado en paquetes de tipo *<<contentSpecification>>*.

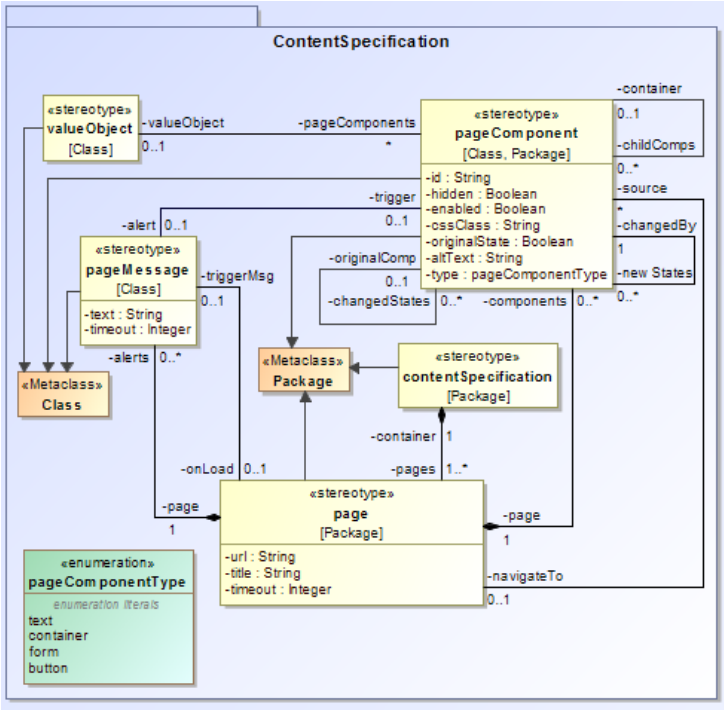


Figura 18: Elementos del grupo Content Specification

Cada página Web a verificar es modelada mediante paquetes de tipo *<<page>>* con los valores etiquetados *url* (URL correspondiente a la página en cuestión), *title* (título de la página, que aparece en el tag <title> HTML) y *timeout* (cantidad de tiempo máximo de espera para la carga de la página, en segundos).

Los diferentes elementos que pueden definirse dentro de un tag <body> HTML pueden ser modelados mediante elementos de tipo *<<pageComponent>>* que pueden tener las propiedades principales: *id* (código identificador del elemento), *type* (identifica el tipo de elemento), *cssClass* (*class* CSS que debe tener el elemento), *hidden* (valor booleano que indica si el elemento se encuentra escondido o no), *enabled* (valor booleano que indica si el elemento se encuentra habilitado o no), *altText* (si el elemento es un texto o un botón, esta propiedad define el texto que debe mostrar), *originalState* (valor booleano que indica si el elemento

representa a un componente de la página en su estado original o, en caso de ser falso, el componente es consecuencia de una acción efectuada sobre otro elemento).

Un <<*pageComponent*>> puede ser de tipo texto (*text*), contenedor (*container*), formulario (*form*) o botón (*button*):

- Un elemento de tipo texto, como su nombre lo indica, representa a un texto visible en la página a verificar en un momento dado; la cadena de texto que despliega puede representarse mediante la propiedad *altText* del <<*pageComponent*>> o mediante datos definidos dentro de un paquete <<*examplesDataPool*>>.
- Un elemento contenedor representa a un elemento HTML que puede estar compuesto por otros sub-componentes. Esta jerarquía puede ser representada mediante el atributo *chilComps* del contenedor.
- Un elemento de tipo formulario representa a un <form> HTML y debe tener asociado un elemento de datos definido en un paquete <<*examplesDataPool*>>.
- Un elemento de tipo botón representa a un vínculo o enlace, un *click* sobre él desencadena una acción. La propiedad *altText* en un elemento de este tipo define el texto que se mostrará en el botón.

Una clase <<*pageMessage*>> representa mensajes de alerta que pueden aparecer en la página en un momento determinado. Tiene las propiedades: *text* (texto que aparecerá en el mensaje) y *timeout* (máximo tiempo de espera para su aparición, en segundos). Puede aparecer al cargarse la página o como consecuencia de una acción sobre algún componente de la página. Un elemento <<*page*>> debe especificar un valor en la etiqueta *triggerMsg* en caso de que se espere la aparición de un mensaje de alerta tan pronto como se carga la página. Esta etiqueta relaciona un <<*pageMessage*>> con el elemento <<*page*>>.

Es posible además especificar elementos de datos para cada <<*pageComponent*>> relacionándolo con un <<*dataElement*>> o con un <<*dataPartition*>> previamente definido (la **Figura 19** muestra la asociación entre <<*pageComponent*>> y los elementos de un *Data Provider* definida en el perfil). Esta relación es realizada asignando un valor a la etiqueta *data* o a la etiqueta *partition* del <<*pageComponent*>>.

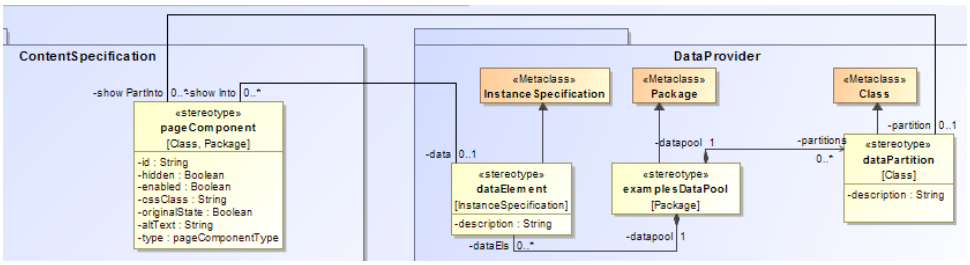


Figura 19: Relación entre un <<pageComponent>> y los elementos del grupo *Data Provider*

La relación entre un <<*pageComponent*>> y un elemento de datos permite especificar los datos a ser utilizados, según el tipo de <<*pageComponent*>>. Mientras un <<*dataElement*>> o un <<*dataPartition*>> pueden tener varios atributos, con valores asignados a cada uno de ellos, un <<*pageComponent*>> puede limitarse a mostrar solo los valores de algunos atributos. Esta limitación es definida mediante clases <<*valueObject*>>, que listan solo los atributos que serán utilizados por el/los elemento/s <<*pageComponent*>>. Un <<*valueObject*>> es una clase cuyos atributos identifican a los atributos a ser utilizados para un elemento de datos dado.

Las acciones[35] sobre un elemento <<*pageComponent*>> pueden ser descritas mediante sus relaciones:

- Al hacer *click* sobre un <<*pageComponent*>> puede esperarse que se cargue una nueva página, esta transición se define con la etiqueta *navigateTo* del <<*pageComponent*>>. Este atributo hace referencia a otra página (elemento <<*page*>>) definida en el modelo.
- Otra posibilidad sería que se espere la aparición de un mensaje de alerta luego de hacer *click* sobre un <<*pageComponent*>>. Esto es especificado mediante la etiqueta *alert* del <<*pageComponent*>>, que lo relaciona con un elemento <<*pageMessage*>>.
- También es posible que otros componentes (elementos <<*pageComponent*>>) de la página se vean afectados como consecuencia de la acción. Esto se especifica asociando todos los elementos <<*pageComponent*>> que se ven afectados, utilizando la etiqueta *newStates* del <<*pageComponent*>> sobre el cual se ejecuta la acción.

El último grupo de elementos definidos en el perfil es ***Constraint Specification***, la **Figura 20** muestra los elementos definidos en él. Cada unidad funcional de código implementada por el desarrollador, puede tener asociada una serie de pre y post condiciones que deben cumplirse. Es posible especificar condiciones de entrada para una prueba en particular y sus salidas o condiciones finales esperadas.

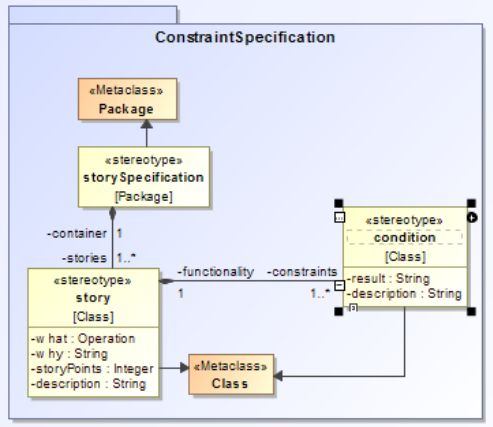


Figura 20: Elementos del grupo *Constraint Specification*

Las funcionalidades con pre y post condiciones serán definidas dentro de paquetes de tipo <<*storySpecification*>>.

Cada funcionalidad se define como clase de tipo <<*story*>> con valores etiquetados que definirán sus características. Las etiquetas *as* y *asPartition* (surgen a partir de las relaciones entre <<*story*>> y los elementos del *Data Provider*, graficadas en la **Figura 21**), *what* (hace referencia a una operación definida en: (i) el <<*dataElement*>> especificado para la etiqueta *as* o; (ii) el <<*dataPartition*>> especificado para la etiqueta *asPartition*), *why* (cadena de caracteres descriptiva) y *storyPoints* (valor numérico) permiten especificar las funcionalidades en forma de historia de usuario con la siguiente estructura[36] *"As a as/asPartition, I want what,*

[35] Se consideró la acción de efectuar un *click* sobre un elemento/componente de la página.
[36] Estructura de historia de usuario basada en http://www.yodiz.com/blog/writing-user-stories-examples-and-templates-in-agile-methodologies/

so that why" y asignar una cantidad de puntos a la historia (fijando un valor a la etiqueta *storyPoints*).

Finalmente, las pre y post condiciones para cada <<*story*>> se definen a partir de clases estereotipadas como <<*condition*>>.

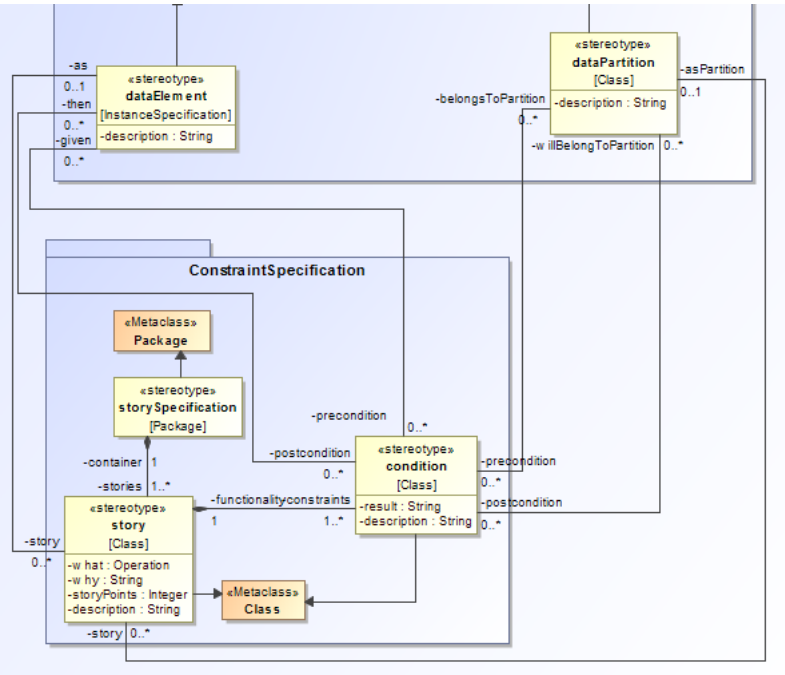


Figura 21: Relaciones entre elementos del grupo Constraint Specification y los elementos del grupo Data Provider.

Una clase <<*condition*>> puede tener un valor etiquetado como *result* que indica (en forma de cadena de caracteres) el valor retornado por la operación ejecutada en el *test*. Las pre y post condiciones se indican mediante diferentes estados en los que se encuentran las instancias de datos especificadas en el ***Data Provider***. Dichos estados pueden especificarse usando elementos de tipo <<*dataElement*>> (usando las etiquetas *given* para pre-condiciones y *then* para post-condiciones) o <<*dataPartition*>> (usando las etiquetas *belongsToPartition* para pre-condiciones y *willBelongToPartition* para post-condiciones). Las relaciones entre estos elementos se muestran en la **Figura 21**.

Utilización de los Perfiles UML para el Modelado de Requerimientos de la Biblioteca Virtual

Utilizando los elementos de modelado definidos en el perfil UML podemos especificar:

- Elementos del dominio del sistema Biblioteca Virtual.
- Algunos datos de ejemplo a utilizar para las pruebas.
- Contenido de las páginas que forman parte de la plataforma *online*.
- Diferentes funcionalidades a implementar por desarrolladores, cada una asociada a pre y post condiciones.

El dominio es definido dentro de un paquete <<*SUT*>> y contiene las entidades (clases <<*domainElement*>>) principales del dominio. A continuación, se mencionan las entidades que consideraremos con sus atributos[37] principales:

- Biblioteca: con atributos nombre (requerido) y ubicación, puede tener varios elementos de tipo Item.
- Usuario: con atributos nombre, *password* (requerido), alias (requerido), fecha de nacimiento, cédula; puede prestar algunos elementos de tipo Item.
- Item: con atributos título (requerido), disponibilidad (requerido), lista de autores, ISBN, tipo de Item (libro, DVD, CD, revista), código identificador (requerido). Una operación asociada a una entidad de este tipo puede ser la verificación de disponibilidad del Item.

Para clasificar un Item por tipo, se define una enumeración con los tipos posibles. Además, puede ser interesante distinguir dos clasificaciones diferentes para un elemento Item: a) el Item está disponible para su préstamo; b) el Item no está disponible. Las clasificaciones o grupos de datos son representadas usando clases <<*dataPartition*>> con los atributos relevantes y los valores que deben tomar para pertenecer a él. La clase a la cual pertenece cada grupo de datos se define asignando un valor a la etiqueta *class* de cada <<*dataPartition*>>. La **Figura 22** ilustra un diagrama del dominio para el sistema de Biblioteca Virtual descrito.

[37] Los atributos requeridos son marcados con el estereotipo <<*required*>>.

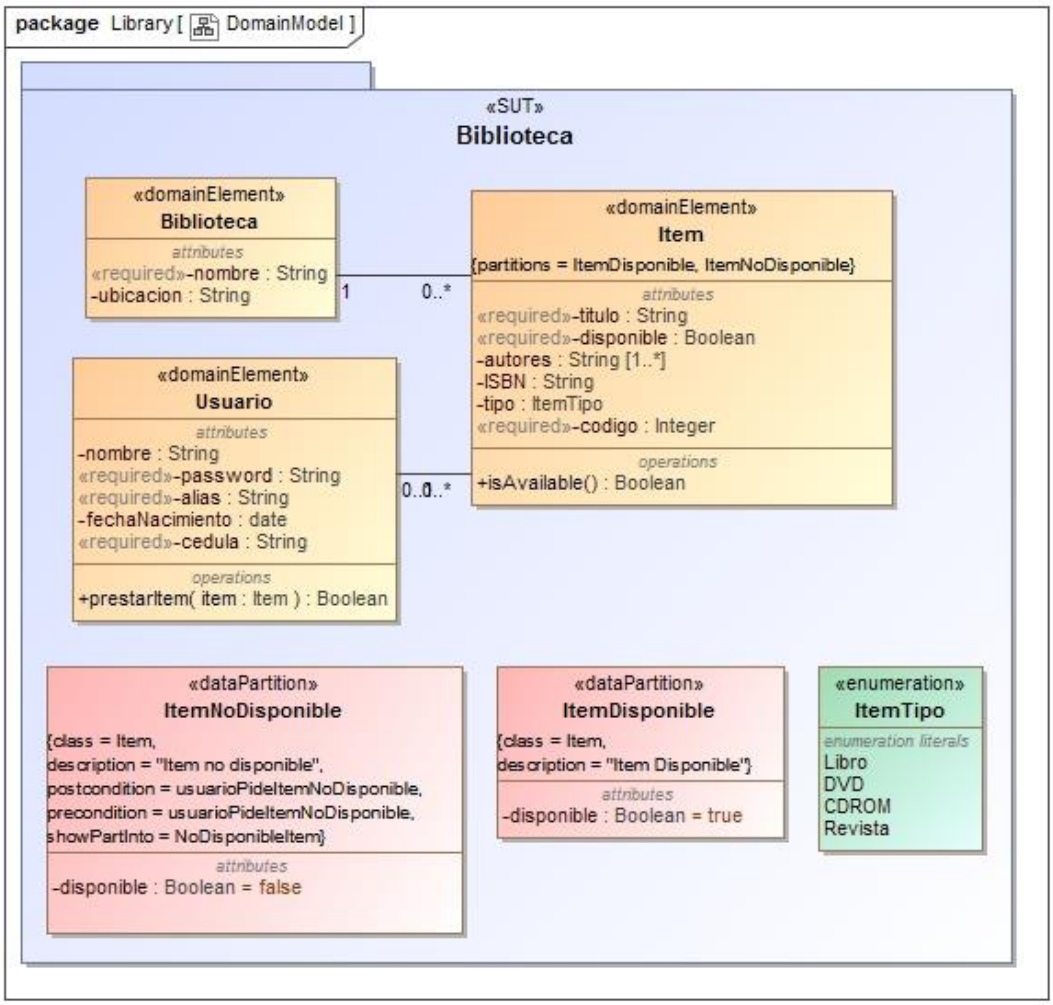


Figura 22: Diagrama de dominio para el sistema Biblioteca Virtual

Una forma de especificar datos de ejemplo para las pruebas ya fue definida en el dominio presentado (a partir de las particiones ItemNoDisponible e ItemDisponible). Sin embargo, para algunas pruebas puede ser importante la definición de ejemplos específicos de elementos <<*domainElement*>> que forman parte del dominio.

En el siguiente diagrama (**Figura 23**), se presentan algunas instancias (elementos <<*dataElement*>>) de las entidades Usuario e Item. Como ejemplo hemos creado un usuario (se supone válido y registrado en el sistema) de ejemplo y tres instancias de Item (un libro no disponible para préstamo y un disco expresado en dos estados diferentes: cuando está disponible y cuando no lo está).

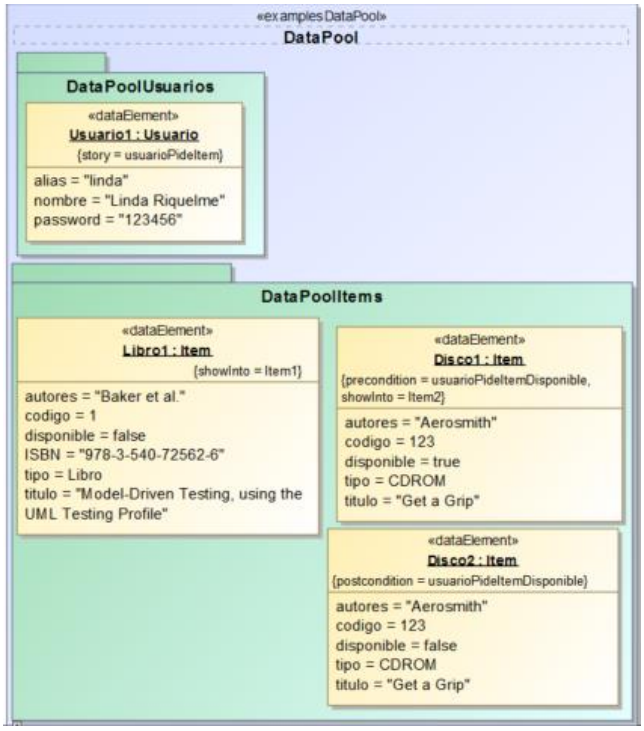


Figura 23: Instancias de elementos del dominio definidas como <<*dataElement*>>.

Supongamos que el desarrollador debe trabajar sobre una nueva funcionalidad: que los usuarios registrados puedan prestar ítems, en caso de estar disponibles. Así, se crea la siguiente historia de usuario *"As Usuario1, I want prestarItem, so that Para poder reforzar mis estudios"*. Siendo prestarItem una operación definida para un <<*domainElement*>> Usuario y Usuario1 una instancia de Usuario definida en un <<*examplesDataPool*>> del modelo; verificaremos las dos condiciones:

1. Que no pueda prestarse un Item no disponible (condición usuarioPideItemNoDisponible, más adelante).
2. Un Item disponible, en cambio, sí puede ser prestado (condición usuarioPideItemDisponible, más adelante).

En la **Figura 24** se puede observar el modelo definido para las condiciones a verificar. La historia de usuario se representa mediante una clase <<*story*>> (se asignará el valor 10, como puntos de historia o *storyPoints*). Usuario1 es un elemento <<*dataElement*>> definido previamente, que implementa la operación prestarItem. Con elementos <<*condition*>> representamos las condiciones que deseamos verificar antes y después de la ejecución de dicha operación:

- Condición usuarioPideItemNoDisponible: dado un Item perteneciente a la partición ItemNoDisponible (definida en el dominio para clases Item con el atributo *disponible* = false), la ejecución de la operación prestarItem deberá dar el resultado "false". Luego de ejecutarse la operación, se vuelve a verificar el estado del Item: éste debe seguir perteneciendo a la partición ItemNoDisponible.
- Condición usuarioPideItemDisponible: en este caso, utilizaremos como dato de prueba un Item con el valor *disponible* = true. El <<dataElement>> Disco1 cumple con esta condición. Como éste se encuentra disponible, el resultado de la ejecución de prestarItem debería ser "true". Finalizada la ejecución de la operación, es necesario volver a verificar los nuevos valores de los atributos de la instancia

Disco1, éstos deben coincidir con los valores definidos en la instancia Disco2 (*disponible* = false ya que el disco deja de estar disponible).

Figura 24: Modelado de historia de usuario "As Usuario1, I want prestarItem, so that Para poder reforzar mis estudios" con dos pre y post condiciones.

Además de definir los diferentes estados del sistema por medio de pre y post condiciones para las operaciones implementadas, el usuario puede definir mediante algunos elementos del grupo ***Content Specification***, la forma en que interactuará con las diferentes páginas del sistema.

Por ejemplo, en la **Figura 25** se modela los requerimientos (definidos en la sección "Ejemplo 1: Biblioteca Virtual" anterior) a verificar:

- **Requerimiento 1:** La primera página (webItemsDisponibles) tiene asociada una URL para su acceso, un título a ser verificado ("Items en Biblioteca") y un tiempo máximo de tolerancia para su carga (10 segundos).
- **Requerimientos 2, 3 y 4:**
  - **Lista de elementos disponibles:** Esta página deberá tener una lista (elemento identificado por el id disponibleList) visible de ítems (cada uno de ellos mostrando solo sus atributos: autores y título). Para verificar que los elementos que aparecen en ellas están disponible, se buscará un elemento con valor *disponible* = false, definido anteriormente entre los datos de ejemplo. Disco1 cumple con esta condición.

    Uno de los ítems que se desea ver en la lista está representado por el <<*pageComponent*>> ItemDisponible. Éste debe mostrar los autores y el título (se establece la relación entre ItemDisponible y el <<*valueObject*>> ItemLista) del <<*dataElement*>> Disco1. Como el item se encuentra disponible, el elemento HTML que lo representa debe estar habilitado. El evento *click* sobre él debe cargar una nueva página (webPrestarDisponible) en la cual debe mostrarse el mensaje de alerta: "¡El disco ha sido reservado para usted!".
  - **Lista de elementos no disponibles:** En la misma página debe visualizarse una segunda lista (identificada por el id unavailableList) con ítems (cada uno de ellos mostrando solo los atributos autores y título, tal como se especifican en el <<*valueObject*>> ItemLista) no disponibles para el préstamo. Se verifica que los datos sean válidos buscando dos elementos con los datos de ejemplo: (i) un <<*pageComponent*>> Item1 con los datos representados en el <<*dataElement*>> Libro1; (ii) un <<*pageComponent*>> NoDisponibleItem con los datos representados en la partición ItemNoDisponible (en este caso, solo interesa validar que el atributo *disponible* tenga el valor "false"). Ninguno de los ítems está disponible por lo que deben estar deshabilitados y un *click* sobre cualquiera

de ellos debe generar la aparición de un mensaje de alerta: “Libro No Disponible”.

- **Requerimiento 5:** La opción de añadir nuevo ítem a la biblioteca se representa mediante el <<*pageComponent*>> newItemLink que redirecciona a la página identificada como addItemForm.

  En esta nueva página debe aparecer un formulario (<<*pageComponent*>> newItemForm) en cuyos campos ingresaremos los datos del ejemplo definido en el <<*dataElement*>> Libro1. En el modelo se especifica que los campos de Item que se ingresarán en el formulario para la prueba son: autores, ISBN, disponible, tipo, titulo. Esto se representa a través del <<*valueObject*>> ItemForm.

  Además, debe existir un elemento de ID “errorMsg” que sirva para mostrar los detalles del ítem recientemente agregado (en caso de que haya sido agregado con éxito); inicialmente no debe ser visualizado (este elemento está representado mediante el <<*pageComponent*>> submitHiddenMsg. Un botón submitBtn define la consecuencia de realizar un *click* sobre él luego de insertar los datos de Libro1 (mediante su valor etiquetado *newStates*): el elemento inicialmente dado por submitHiddenMsg cambiará de estado (pasará a ser visible y se insertarán en el los datos de autores y título de Libro1). Este cambio de estado es representado por el <<*pageComponent*>> submitShownMsg.

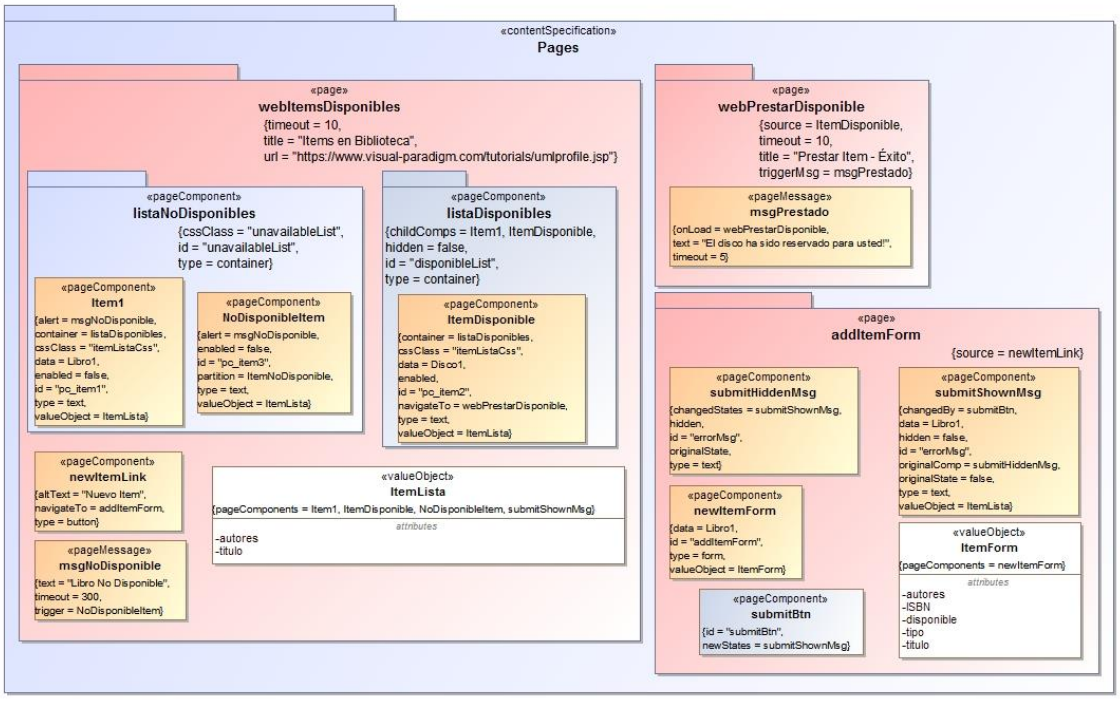


Figura 25: Definición de los componentes de las páginas que se crearán para la nueva historia de usuario.

Utilización de los Perfiles UML para el Modelado de Requerimientos de Amazon.es

A modo de ampliar los ejemplos, en esta subsección representaremos[38] (utilizando los perfiles UML propuestos) los requerimientos definidos anteriormente para el ejemplo de Amazon.es.

1- Visualización de la Home cuando para un usuario no autenticado (**Figura 26**). Todos los componentes se representan mediante elementos <<*pageComponent*>>:

a. <<*pageComponent*>> EnlaceDepartamentos con el texto "Todos los departamentos" de tipo *button*.

b. <<*pageComponent*>> Bienvenida que desplegará el texto "Bienvenido" de tipo *text*.

c. <<*pageComponent*>> SolicitudIdentificacion conteniendo el texto "Hola, Identifícate" de tipo *text*.

d. <<*pageComponent*>> IniciarSesion con el texto "Inicia sesión de manera segura" de tipo *button*.

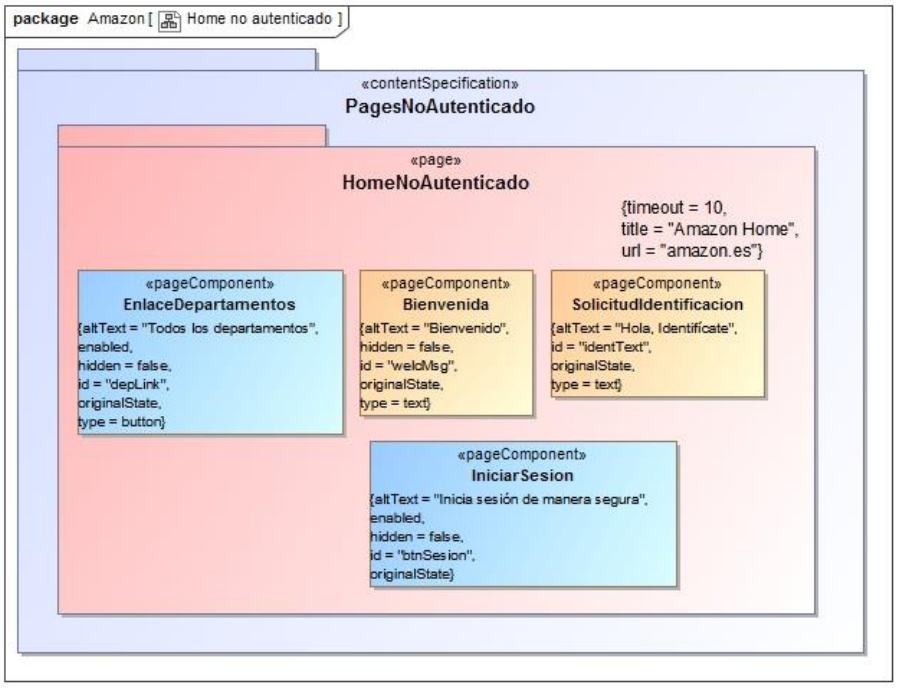


Figura 26: Modelado de elementos que debe visualizar un usuario en la Home no autenticado en el ejemplo Amazon.es (usando perfiles UML propuesto).

2- Función del enlace "Todos los departamentos": Se desea visualizar la lista de los departamentos de la plataforma por lo que se definen 3 datos de ejemplo (<<*dataElement*>> Dep1, <<*dataElement*>> Dep1.1 y <<*dataElement*>> Dep1.2) con los atributos nombre y descripción. La definición de la entidad <<*domainElement*>> Departamento (perteneciente al dominio del sistema) y sus correspondientes instancias (correspondientes a datos de ejemplo a usar en los *tests*) se representan en la **Figura 27**.

[38] En los diagramas, resaltamos en azul los enlaces, en naranja los elementos de tipo texto y en verde los formularios.

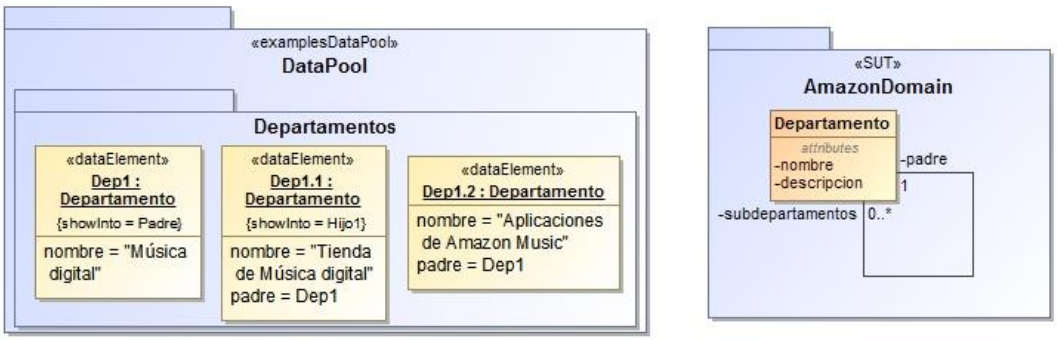


Figura 27: Modelado de Entidades y Datos de Ejemplo para representar Departamentos de Amazon.es

Además, se modela la nueva página (iremos a ella haciendo click sobre el <<*pageComponent*>> antes definido EnlaceDepartamentos). La nueva página contará con dos elementos principales: <<*pageComponent*>> Padre y <<*pageComponent*>> Hijo1. Ambos representan enlaces cuyos datos derivan de los ejemplos definidos por Dep1 y Dep1.1. Solo se verá el nombre de cada departamento en pantalla, esto es definido por el <<*valueObject*>> DepartamentoVO. La **Figura 28** muestra los elementos descritos.

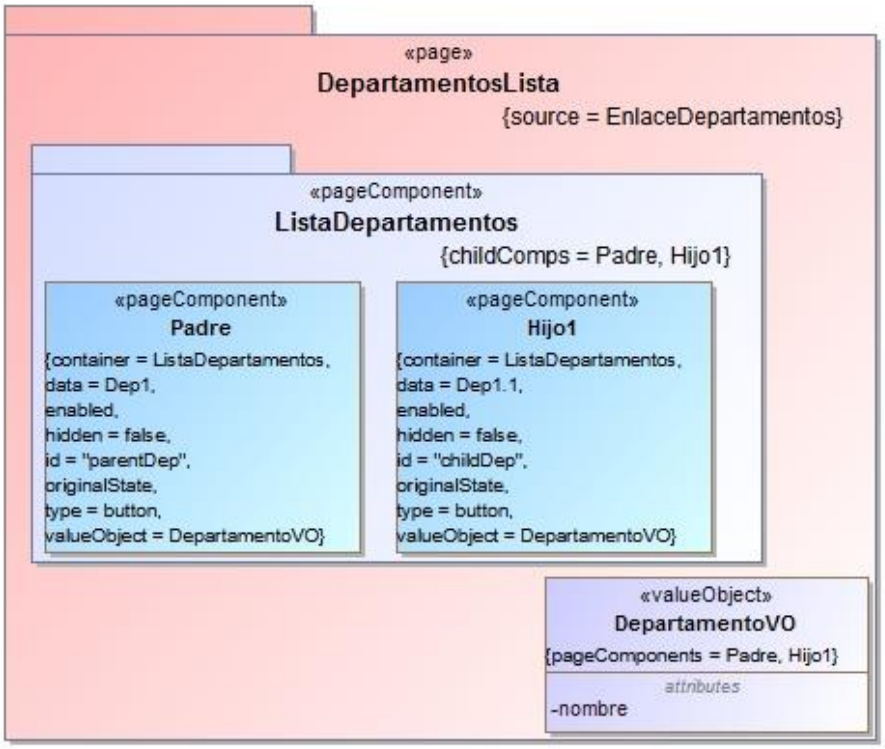


Figura 28: Modelado de página con lista de departamentos de Amazon.es.

3- Inicio de sesión: Para el inicio de sesión necesitamos definir una nueva entidad en el dominio (<<*domainElement*>> Usuario) a partir de los cuales definiremos datos de prueba (un usuario con datos válidos dado por el <<*dataElement*>> UsVal y otro con datos no válidos dado por el <<*dataElement*>> UsNoVal). Estos elementos aparecen ilustrados en la **Figura 29**.

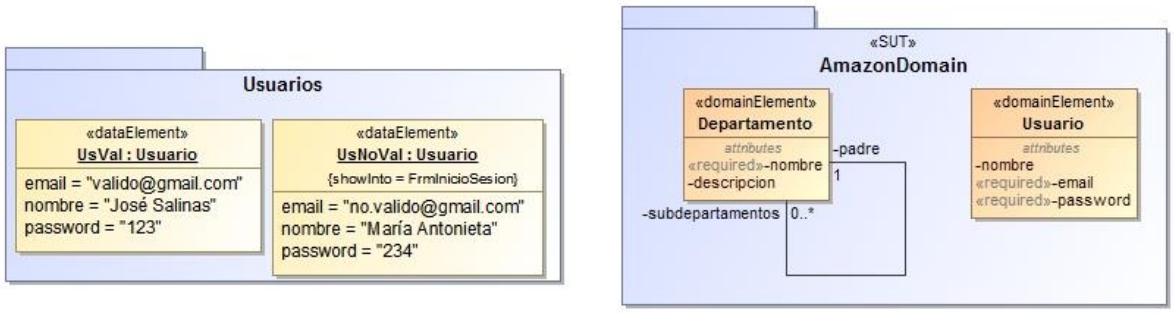


Figura 29: Entidad Usuario agregada al dominio Amazon.es y datos de prueba definidos.

La página de inicio de sesión (<<*page*>> InicioSesion) es modelada utilizando un elemento formulario (<<*pageComponent*>> FrmInicioSesion), un elemento de texto (<<*pageComponent*>> Titulo) y un botón (<<*pageComponent*>> BtnInicio). Los campos del formulario se definen mediante el <<*valueObject*>> UsuarioLoginVO. Se puede acceder a esta página haciendo *click* sobre el elemento <<*pageComponent*>> IniciarSesion, representado en la Home (requerimiento 1). La **Figura 30** ilustra los elementos mencionados.

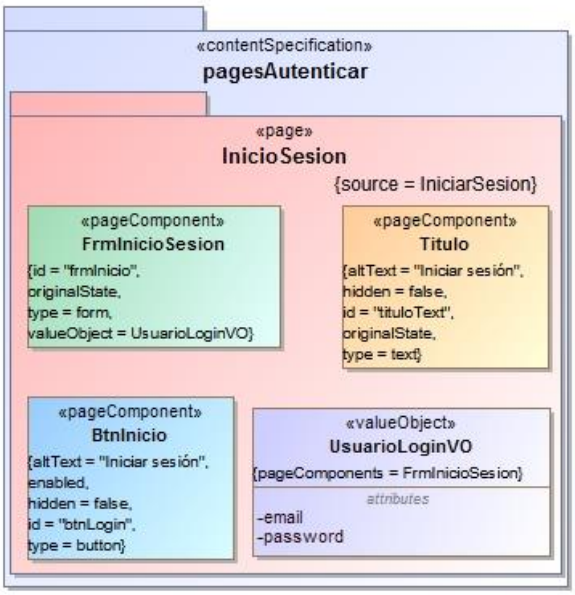


Figura 30: Modelado de página de Login Amazon.es.

4- Inicio de sesión incorrecto: Para simular un error en el inicio de sesión, volvemos a modelar la página de inicio de sesión, indicando esta vez el dato de prueba a utilizar en los campos del formulario (atributos del <<*dataElement*>> UsNoVal que aparecen en el <<*valueObject*>> UsuarioLoginVO). Un *click* sobre el botón <<*pageComponent*>> BtnInicioNoValido provocará el cambio de estado del <<*pageComponent*>> Titulo1, el cual pasará a mostrar el texto "Inténtalo nuevamente" (el elemento <<*pageComponent*>> Titulo2 representa el siguiente estado para Titulo1).

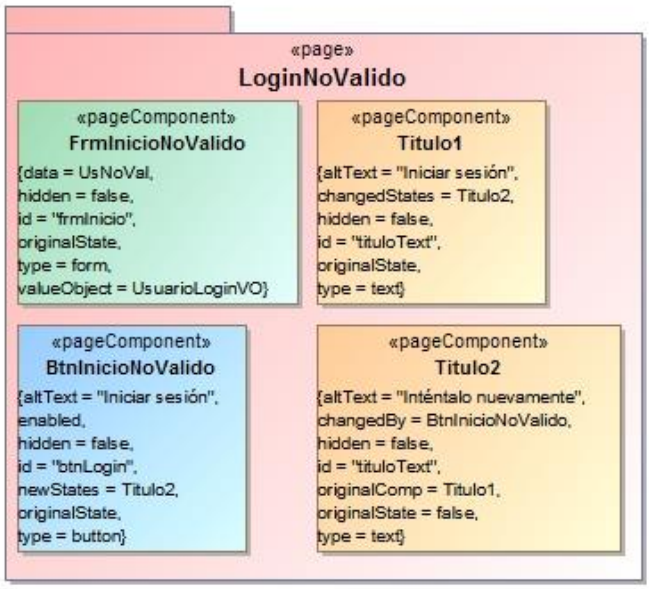


Figura 31: Modelado de Página de Inicio de Sesión para Amazon.es e intento de autenticación por parte de un usuario no válido.

5- Actualización de la Home para un usuario autenticado: Volvemos a modelar la página de inicio de sesión, esta vez, indicando que el usuario utilizado para las pruebas será el dado por el <<*dataElement*>> UsVal (especificado en el formulario <<*pageComponent*>> FrmInicioValido) pues cuenta con datos válidos para la autenticación. Un *click* sobre el botón <<*pageComponent*>> BtnInicioValido deberá redireccionar a otra página (acción expresada mediante el valor *navigateTo* del <<*pageComponent*>> BtnInicioValido). El modelo de la página descrita aparece en la **Figura 32**.

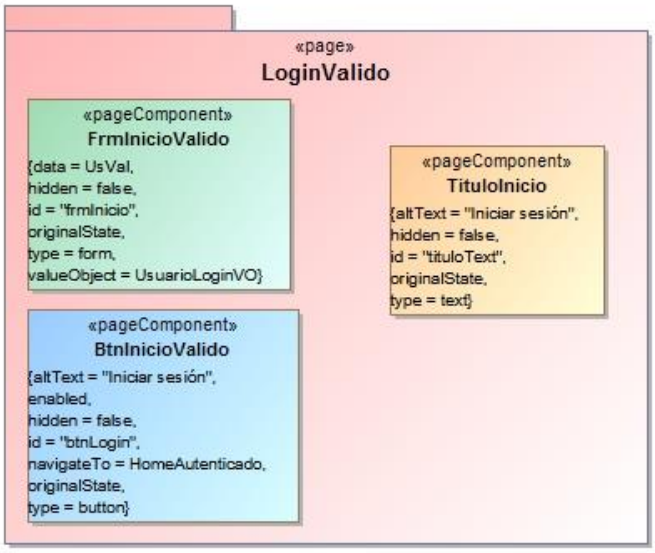


Figura 32: Modelado de Autenticación Exitosa en Amazon.es.

La Home para un usuario autenticado está representada en la **Figura 33**. Se accede a ella luego de una autenticación exitosa y cuenta con un elemento de texto <<*pageComponent*>> Saludo que despliega el texto "Hola", seguido por el nombre del

usuario autenticado (<<*dataElement*>> UsVal con atributos dados por el <<*valueObject*>> SaludosUsuarioVO).

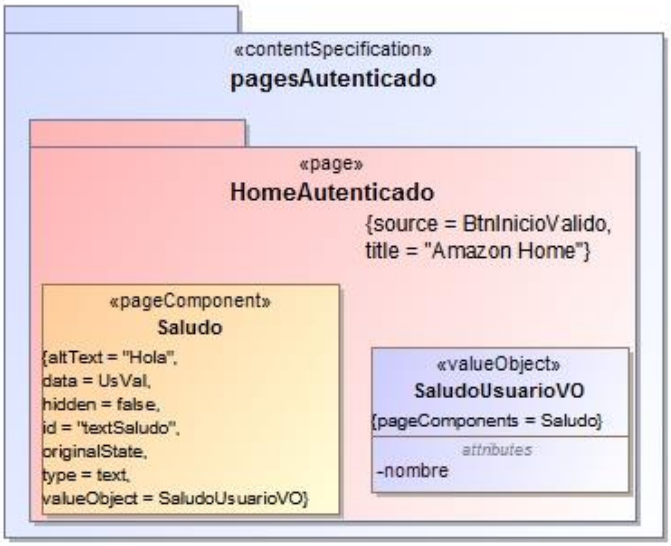


Figura 33: La autenticación exitosa redirecciona a la Home, donde se visualiza un saludo de bienvenida al usuario.

Generación de Código de Test a partir de los Modelos

Los modelos de usuario y desarrollador representan *tests* (aceptación y unitarios, respectivamente) abstractos. La transformación de *tests* abstractos a concretos se realiza utilizando la herramienta Acceleo[39], en conjunto con las reglas de transformación definidas por MoFQA (MoFQA *transformation rules*). Estas reglas permiten generar código en Java[40] (*frameworks* TestNG[41] y Selenium[42]).

El código generado puede ser utilizado para ejecutar los tests de aceptación y tests unitarios sobre el software bajo verificación, en un entorno de desarrollo integrado como Eclipse43. Es importante mencionar, que una vez configurado todo el entorno de desarrollo en Eclipse, será posible centralizar la ejecución de todos los pasos que el desarrollador debe llevar a cabo para el proceso de testing:

- Importando los perfiles UML, en formato EMF, podrá llevar a cabo el modelado principalmente de tests unitarios. Como se menciona en la siguiente sección, los modelos generados por la herramienta de modelado de tests de aceptación, también se generan en formato EMF por lo que pueden ser importados y adjuntados a los modelos definidos por el desarrollador.

- La instalación de la herramienta Acceleo y la importación de las reglas de transformación, le permitirá generar los tests ejecutables a partir del modelo que representa los tests abstractos del sistema a verificar.

---

[39] **Acceleo:** http://www.eclipse.org/acceleo/
[40] **Java:** https://www.java.com/es/
[41] **TestNG:** http://testng.org/doc/index.html
[42] **Selenium:** http://www.seleniumhq.org/
[43] **Eclipse:** http://www.eclipse.org/

- Para ejecutar los tests generados y visualizar los resultados, es necesario instalar al proyecto las herramientas TestNG y Selenium. Una vez instalados, la ejecución del archivo XML generado ejecuta de forma automática todos los tests definidos por los modelos.

- Además, si el código de implementación del sistema web a verificar está escrito en Java, es posible integrar los ambientes de testing y desarrollo en un solo proyecto. Esto permitiría además la definición de tests que requieran acceder a métodos de las clases definidas en el código de implementación (este es el caso de los tests unitarios).

MoFQA Modeler: Una propuesta para definición de *tests* de aceptación

En el esfuerzo de facilitar el modelado de requerimientos (que derivarán a los *tests* de aceptación) de parte de los usuarios finales del sistema, hemos desarrollado **MoFQA Modeler**[44]. Consiste en una herramienta de modelado simple que permite definir los elementos:

- Páginas con sus componentes (texto, campos de formularios, botones, mensajes de alerta).
- Datos de ejemplo que deberán visualizarse en los componentes de las páginas durante la ejecución de los *tests* de aceptación.
- Resultados de las interacciones con los diferentes componentes de las páginas.

MoFQA Modeler toma los elementos definidos por el usuario y los transforma en modelos (EMF versión 5, enriquecido con los perfiles UML definidos especialmente para nuestra propuesta). Estos modelos representan los elementos del dominio del *software* bajo verificación y *tests* de aceptación abstractos. Por el formato en el cual son exportados los modelos, también es posible integrarlos directamente al entorno de trabajo configurado por el desarrollador (propuesto en la sección anterior).

En el **Apéndice A** incluimos un manual de usuario donde se presentan las características de esta herramienta.

Con esta herramienta se pretende otorgar un nivel de abstracción más para el usuario final del sistema para el cual generaremos los *tests*. De esta forma, el usuario no necesita tener conocimientos de modelado. Además, como la herramienta genera automáticamente varios elementos del modelo, el usuario debe invertir menos tiempo en el modelado de requerimientos.

[44] Disponible en http://www.dei.uc.edu.py/proyectos/mddplus/herramientas/mofqa/modeler/.

Capítulo 4. Validación y Análisis de Resultados

Este trabajo ha propuesto, primeramente, un modelo para el desarrollo de *software* basado en conceptos teóricos y evidencias halladas en la literatura sobre las ventajas que presentan tanto el proceso TDD como MBT, utilizándose de forma aislada. La propuesta integra ambos enfoques, principalmente, para reforzar (y automatizar) la generación de *tests* utilizando técnicas MBT. Hemos planteado la hipótesis de que, entre otras ventajas, esta integración permitiría que el proceso de desarrollo resulte más productivo respecto al menor tiempo dedicado a la generación de *tests*, la mayor cantidad de *tests* asociados a cada funcionalidad y, mediante la incorporación del usuario final en la definición de *tests* de aceptación, un mayor acercamiento a los requerimientos reales para la implementación de cada funcionalidad. Esta hipótesis, sin embargo, queda como propuesta teórica a ser evaluada por trabajos futuros.

Por otra parte, hemos desarrollado herramientas MBT (que pueden o no ser utilizadas siguiendo el modelo de desarrollo propuesto por **MoFQA**) como propuestas para la definición y generación de *tests* unitarios y de aceptación para sistemas de plataforma web. Efectuamos una primera evaluación a las herramientas desarrolladas, los resultados serán presentados en este capítulo.

Método de Validación

Para poder llevar a cabo un análisis inicial sobre la utilidad de las herramientas presentadas, llevamos a cabo dos pequeñas experiencias:

1- Definición de los requerimientos especificados en uno de los casos de ejemplo presentados y generación de *tests* correspondientes. Como para la ejecución de los *tests* necesitábamos un sitio web existente, optamos por el Ejemplo 2: Amazon.es.

2- Se llevó a cabo un pequeño taller de laboratorio con alumnos de semestres iniciales de Ing. Informática para obtener una idea aproximada de la usabilidad de la herramienta **MoFQA Modeler**.

Es importante mencionar nuevamente que con estas experiencias no se desea validar el modelo de desarrollo propuesto por **MoFQA**. En ambos casos se trabajó con sistemas de plataforma web ya implementados, por lo que no se siguió el proceso TDD para la implementación de las funcionalidades en prueba.

Métricas de Interés

Como se menciona en [25], una métrica importante para validar el proceso de *testing* está basada en la cobertura de requerimientos por parte de *tests*: “La calidad de una porción de *software* es frecuentemente definida por su habilidad de cubrir los requerimientos definidos”. Esta métrica obtiene una medida de la cantidad de requerimientos que son verificados por los *tests* definidos. Esta medida puede presentarse como el porcentaje de requerimientos cubiertos por *tests*:

$$Cobertura = \left(\frac{cant.de\ requerimientos\ cubiertos}{total\ de\ requerimientos}\right) * 100$$

Fórmula 1: Porcentaje de cobertura de requerimientos [25]

Para realizar un mapeo entre los requerimientos y *test cases* definidos, utilizaremos una matriz de trazabilidad (RTM, *Requirements Traceability Matrix*)[45].

Si bien la cantidad de líneas de código es una métrica que se recomienda evitar [7], nos puede dar una idea aproximada del nivel de automatización en la generación de *tests* por lo que utilizaremos también esta métrica. Asociada a esta métrica, contabilizaremos los pasos intermedios generados para cada *test case*: un *test case* puede estar constituido por varios pasos, cada uno de los cuales realiza verificaciones intermedias.

Finalmente, deseamos obtener una medida de usabilidad para la herramienta propuesta **MoFQA Modeler**. Para ello utilizamos el cuestionario SUS (*System Usability Scale*), tal como se menciona en [26]: "La usabilidad de un sistema no es una cualidad que pueda medirse de forma absoluta, sin embargo, puede ser aproximada midiendo qué tan apropiado resulta el sistema para los propósitos de los usuarios y el entorno en el cual será usado (apropiado para el contexto en el cual será implementado). SUS consiste en una escala simple de 10 ítems que otorga una visión global de las visiones subjetivas de la usabilidad de un sistema.". En la **Figura 34** se listan los 10 ítems definidos en SUS.

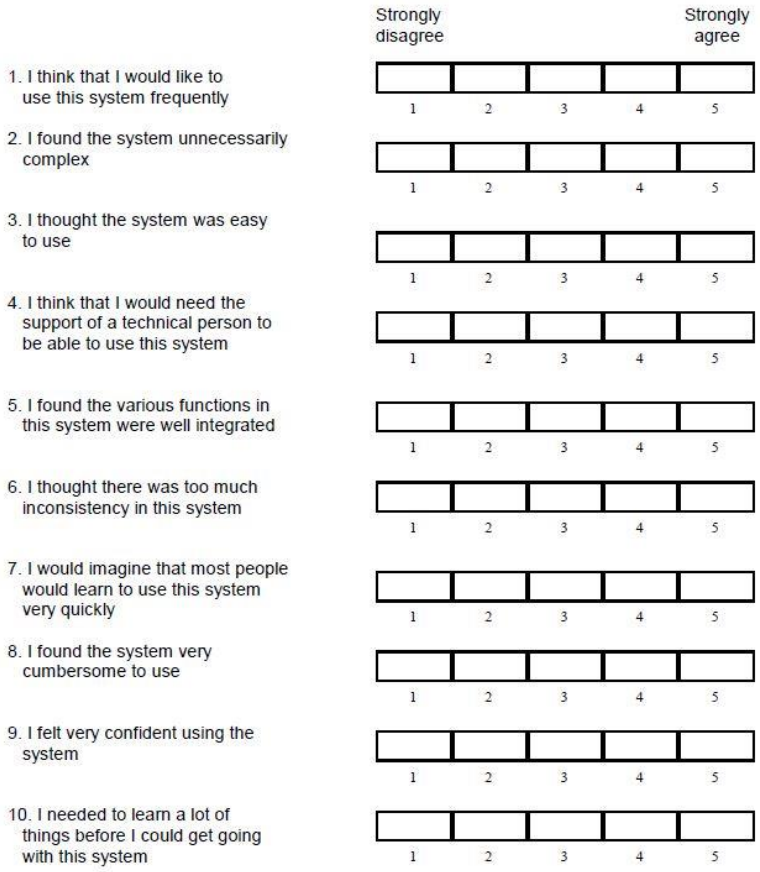


Figura 34: Escala para medición de la usabilidad (SUS) [26].

[45] How to Create Requirements Traceability Matrix – Exact Process with Sample TM Template: http://www.softwaretestinghelp.com/requirements-traceability-matrix/

El cuestionario SUS debe ser respondido por los encuestados al finalizar la experiencia con la herramienta. Cada ítem de la escala tiene asociado un valor numérico que va de "Fuertemente en desacuerdo" (1) a "Fuertemente de acuerdo" (5). Al finalizar la encuesta, es posible obtener una medida de la usabilidad que va de 0 a 100 (más detalles sobre el cálculo en [26]). Para el análisis de dicho valor numérico, adoptaremos las pautas dadas en [27]:

- Un valor de 68 en SUS es calificado por debajo del promedio, mientras que un valor mayor a 68 está por encima del promedio.
- El valor dado por SUS no está relacionado de forma lineal al porcentaje de usabilidad, como se mencionó, un puntaje 68 en SUS representa aproximadamente el 50% en la medida de usabilidad. La **Figura 35** grafica esta relación.

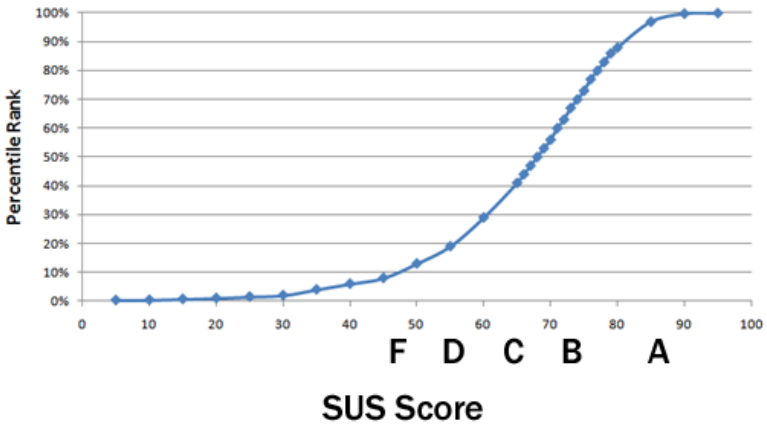


Figura 35: Relación entre puntajes SUS (con una escala de letras: de A+ a F) y escala percentil de la medida de Usabilidad. [27]

Finalmente, tomamos la escala definida en [28] para clasificar las puntuaciones obtenidas en SUS por cada encuestado, clasificando dicho valor de "Lo peor imaginable" a "Lo mejor imaginable". En la **Figura 36** se relaciona la escala de adjetivos respecto a los valores SUS.

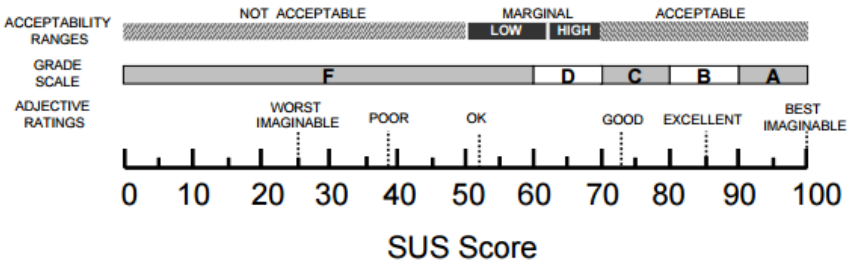


Figura 36: Relación valores SUS, rangos de aceptabilidad y escala de adjetivos. [28]

Validación 1: Ejemplo Amazon.es

La ilustración consistió en la definición de 5 requerimientos para el portal web Amazon.es:

R1- Algunos elementos a ser visualizados en la Home por un usuario no autenticado.

R2- Lista de departamentos debe desplegar algunos departamentos significativos (expresados como datos de ejemplo).

R3- Definición de elementos para la página de inicio de sesión.

R4- Inicio de sesión no válido.

R5- Inicio de sesión con datos válidos de usuario debe redireccionar a la Home y desplegar nuevos datos, de acuerdo al usuario autenticado.

Se prosiguió a modelar los requerimientos de dos formas: a. Utilizando los perfiles UML y el editor UML MagicDraw[46]; b. Utilizando la herramienta MoFQA Modeler. Como resultado se realizó una comparación entre los tiempos para la definición de *tests*.

Análisis de Resultados

La ilustración permitió verificar el ahorro de tiempo que supone el modelado de *tests* de aceptación (según las definiciones de los perfiles MoFQA) utilizando MoFQA Modeler, en comparación a hacerlo utilizando directamente los perfiles UML y una herramienta de edición de UML. Creemos que MoFQA Modeler lleva ventaja porque abstrae varios detalles del modelo, que se presentan transparentes para el usuario. Su uso no solo permite que usuarios sin experiencia en modelado definan *tests* de aceptación sino que, además, permite que usuarios más técnicos puedan modelar dichos *tests* en menor cantidad de tiempo. En la Tabla 2 se visualizan los tiempos obtenidos.

	Modelado en MagicDraw	**Modelado en MoFQA Modeler**
R1	18 min.	5 min.
R2	19 min.	6 min.
R3	20 min.	8 min.
R4	14 min.	3 min.
R5	14 min.	3 min.

Tabla 2: Tiempo (en minutos) para el modelado de cada requerimiento definido en la Experiencia 2 utilizando (i) Perfiles UML y MagicDraw; (ii) MoFQA Modeler.

Si bien es posible visualizar que el tiempo de modelado es mucho menor en todos los casos (un promedio de 12 minutos de ventaja para MoFQA Modeler en todos los requisitos), es importante mencionar que la herramienta tiene limitaciones [10] y no es posible generar todos los elementos disponibles en el perfil UML de MoFQA. Sin embargo, para la mayoría de los casos, las funcionalidades ofrecidas son suficientes para generar los *tests* requeridos.

Validación 2: Experiencia con Alumnos

Se llevó a cabo una segunda experiencia, esta vez en una clase de laboratorio con alumnos del segundo semestre de la carrera de Ing. Informática. En total, participaron en la

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

experiencia 24 alumnos. Principalmente, se deseaba con esta experiencia obtener una medida aproximada sobre la usabilidad de la herramienta **MoFQA Modeler** según usuarios sin experiencia en modelado o programación.

Procedimientos

En una sesión de 90 minutos, se presentó la herramienta **MoFQA Modeler** con sus principales características y se solicitó a los alumnos modelar un requerimiento para una plataforma en línea, bastante utilizada y conocida por ellos: Aula Virtual[47].

Al iniciar la sesión se dio una breve introducción sobre el concepto de *tests* de aceptación, su importancia y los métodos que se utilizan para su definición (manual y herramientas de automatización para la ejecución de *tests*). La idea de esta introducción fue motivar a los alumnos sobre la importancia de estos tipos de *tests* e introducirlos sobre la importancia de contar con una herramienta que permita al usuario generarlos automáticamente durante la etapa de definición de los requerimientos del *software*.

El siguiente paso fue mostrar la utilización de la herramienta mediante la definición de una historia en la que se representaría el requerimiento 1 del ejemplo, presentado anteriormente en este documento, Amazon.es.

Posteriormente, se llevó a cabo un breve trabajo (descrito en la planilla presentada en el **Apéndice B**). En resumen, el trabajo consistía en:

- Seleccionar al menos un requerimiento (de una lista predefinida) para ser verificado en la plataforma Aula Virtual.
- Definir *tests* a realizar de forma manual para la verificación de los requerimientos.
- Modelar los requerimientos utilizando **MoFQA Modeler**, capturar los datos: tiempo de modelado con la herramienta, dudas/complicaciones con la utilización de la herramienta.
- Completar el cuestionario SUS adaptado para la experiencia.

Adaptación de SUS para la experiencia

Como se desea obtener una medida de la usabilidad de **MoFQA Modeler** para un grupo de usuarios con perfil no técnico, al finalizar la experiencia solicitamos a los alumnos que completen el cuestionario SUS adaptado para los fines de nuestra evaluación.

Las preguntas se responden asignando valores que van de 1 (Fuertemente en desacuerdo) a 5 (Fuertemente de acuerdo). Éstas son presentadas a continuación:

1. Pienso que me gustaría utilizar **MoFQA** frecuentemente para definir *tests* de aceptación para una aplicación web.

2. La herramienta **MoFQA** me parece innecesariamente complicada.

3. **MoFQA** me resultó fácil de utilizar.

4. Pienso que necesitaría el soporte técnico de una persona para poder utilizar la herramienta.

[47] https://aulavirtual.uc.edu.py/aulas/

5. Creo que las funcionalidades de la herramienta están bien integradas.

6. Me parece que hay mucha inconsistencia en la herramienta.

7. Me parece que la mayoría de las personas aprenderían muy rápido a utilizar **MoFQA**.

8. Me resultó muy incómodo utilizar **MoFQA** para definir los *tests* de aceptación.

9. Me siento seguro utilizando **MoFQA** para definir *tests* de aceptación.

10. Necesité aprender muchas cosas antes de poder utilizar **MoFQA**.

Complicaciones durante la Experiencia

Durante el taller surgieron algunos inconvenientes que pudieron afectar negativamente a la experiencia de los usuarios utilizando la herramienta. A continuación, mencionamos las más relevantes:

- Problemas de conexión a internet y acceso concurrente a la herramienta impidieron que algunos alumnos pudieran terminar el modelado de requisitos. En algunos casos, se dio por finalizado el ejercicio pero no pudieron guardar los resultados.
- En el momento de la experiencia, la herramienta presentaba un pequeño *bug*: no se podía seleccionar las páginas de destino para los enlaces modelados. Luego de la experiencia, este *bug* ha sido subsanado.
- La limitación en el tiempo solo permitió que cada alumno trabaje por un único requerimiento. Los modelos definidos resultaron finalmente bastante acotados respecto a todas las funcionalidades que ofrece **MoFQA Modeler**.

Recopilación de Datos

Para analizar los resultados de la experiencia se tomaron los siguientes datos:

- La planilla de experiencia (ver **Apéndice B**) completada por cada alumno.
- Los modelos generados a partir de las definiciones de requerimientos, mediante **MoFQA Modeler**.

De cada planilla se obtuvo primeramente los valores asignados por los alumnos a cada ítem del cuestionario SUS. A partir de ahí se obtuvo una primera medida de usabilidad. De los 24 alumnos encuestados, 23 completaron las 10 preguntas del cuestionario (el valor SUS del alumno restante fue descartado por estar incompleto).

El valor SUS de cada participante es calculado de la siguiente forma [26]: los valores de las preguntas impares son disminuidos en 1 y, en el caso de las preguntas pares, restamos sus valores de 5. Sumamos los resultados obtenidos y calculamos el producto por 2,5. Este cálculo es representado por la **Fórmula 2**.

$$SUS_i = \left(\sum_{j=1}^{5} (5 - P_{2j}) + \sum_{j=1}^{5} (P_{2j-1} - 1) \right) * 2.5$$

Fórmula 2: Cálculo del valor SUS para participante i, donde P_k corresponde al valor asignada a la pregunta número k del cuestionario.

Para obtener la primera medida de usabilidad, calculamos el promedio de estos valores individuales. Para 23 encuestados, obtuvimos un promedio de SUS igual a 55,43 (por debajo del promedio, según el análisis presentado en [27]). En la **Tabla 3** presentamos el valor SUS obtenido para cada participante de la encuesta y el promedio final.

ALUMNO	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	SUS
1	5	2	4	2	4	5	4	1	4	4	**67,5**
2	3	2	2	5	4	2	2	5	3	2	**45**
3	4	2	4	3	4	2	4	2	1	2	**65**
4	4	2	5	2	4	1	5	1	5	1	**90**
5	3	2	3	3	5	3	4	3	4	3	**62,5**
6	3	4	3	5	3	4	4	3	3	4	**40**
7	3	2	2	4	4	1	4	2	4	1	**67,5**
8	3	3	3	3	3	3	3	3	3	3	**50**
9	3	3	3	4	4	2	2	1	3	2	**57,5**
10	3	1	5	3	3	3	3	3	3	3	**60**
11	4	5	3	4	3	4	3	5	3	3	**37,5**
12	1	1	4	3	2	3	4	3	2	4	**47,5**
13	3	3	4	5	2	5	3	5	5	1	**45**
14	2	3	2	3	3	4	2	3	3	2	**42,5**
15	3	1	4	5	2	5	5	3	2	2	**50**
16	4	1	4	2	2	3	5	1	5	4	**72,5**
17	2	3	2	1	2	3	5	3	3	1	**57,5**
18	2	3	2	4	3	3	1	4	1	3	**30**
19	4	2	4	2	3	2	4	2	4	3	**70**
20	3	4	2	5	3	3	2	4	2	4	**30**
21	3	3	3	4	4	2	3	2	4	3	**57,5**
22	4	1	4	2	4	2	4	1	4	1	**82,5**
23	3	3	3	4	3	3	4	3	2	3	**47,5**
									SUS PROMEDIO		**55,43**

Tabla 3: Valor SUS para cada participante de la encuesta[48] y promedio final.

Clasificamos además los valores SUS individuales según la escala de adjetivos dada por [28]: 2 encuestados calificaron la herramienta como "Lo peor imaginable", 9 le dio un puntaje de "Pobre", 10 le dio un puntaje de "Aprobado", mientras que 1 la calificó como "Buena" y 1 como "Excelente". Estos resultados pueden observarse en la **Tabla 4** y Figura 37: Clasificación de valores SUS por Escala de Adjetivos.**Figura 37**. Tenemos así un 47,83% de los encuestados que asignó un puntaje por debajo a "Aceptable" (Adjetivo OK) a la herramienta, mientras que un 52,17% lo calificó como "Aceptable" o mejor.

A partir de los modelos generados (*tests* abstractos) por cada alumno mediante la herramienta, se verificó que muy pocos lograron generar modelos significativos para los requerimientos que seleccionaron en el trabajo. Durante la experiencia, pudimos verificar que gran parte de los problemas se debieron a los inconvenientes mencionados en la sección anterior. Por esta razón, se seleccionaron solamente los 4 modelos más completos para generar los *tests* de aceptación.

[48] Cada columna P_i indica el valor dado por el participante a la pregunta número i.

SUS	Adjetivo
30	WORST
30	WORST
37,5	POOR
40	POOR
42,5	POOR
45	POOR
45	POOR
47,5	POOR
47,5	POOR
50	POOR
50	POOR
57,5	OK
57,5	OK
57,5	OK
60	OK
62,5	OK
65	OK
67,5	OK
67,5	OK
70	OK
72,5	OK
82,5	GOOD
90	EXCELLENT

Tabla 4: Clasificación de valores SUS por Escala de Adjetivos.

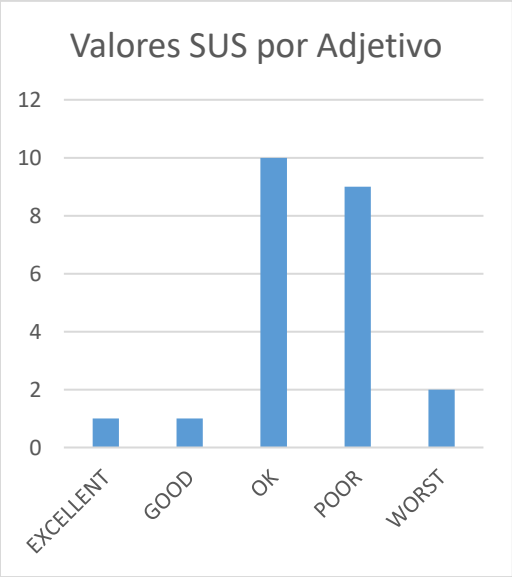


Figura 37: Clasificación de valores SUS por Escala de Adjetivos.

Para las 4 muestras seleccionadas, realizamos una comparación entre los *tests* manuales definidos por los alumnos y los *tests* generados a partir de los modelos resultantes. Las comparaciones que interesaron se presentan a continuación:

- Cantidad de pasos y verificaciones intermedias definidas para *tests* manuales con respecto a la cantidad de pasos y verificaciones intermedias realizadas por los *tests* generados: Se verificó que para cada requerimiento definido, las reglas de transformación permitieron generar varios pasos intermedios de *test*, con verificaciones intermedias. Con esto concluimos que cada requerimiento está acompañado por al menos un *test* que, a su vez, está compuesto por varios pasos intermedios que realizan pequeñas verificaciones por el camino. La ¡Error! No se encuentra el origen de la referencia. muestra los *tests* intermedios generados para uno de los trabajos seleccionados. El requerimiento modelado en el trabajo seleccionado tenía asociado solamente 3 pasos de test: 1- ingresar a la Home; 2- Buscar 3 enlaces; 3- Probar que los enlaces direcciones a páginas correctas.

Pasos intermedios de *tests* generados	**Sub-pasos**	**Criterios aceptación/rechazo**
Carga URL https://aulavirtual.uc.edu.py/aulas/index.php		Pasa si la página carga y tiene el título "Index"
Busca el elemento "Cursos de la plataforma"		Solo pasa si existe el elemento ID="Cursos de la plataforma"
Busca el elemento "Darse de baja de Curso"		Solo pasa si existe el elemento ID="Darse de baja de Curso"
Busca el elemento "Inscribirse a nuevo Curso"		Solo pasa si existe el elemento ID="Inscribirse a nuevo Curso"
Verificar componente "Cursos de la plataforma"	Carga URL https://aulavirtual.uc.edu.py/aulas/index.php	
	Busca el elemento "Cursos de la plataforma"	Solo pasa si existe el elemento ID="Cursos de la plataforma"
	Verifica que esté visible	Método isDisplayed() = true
	Verifica que esté habilitado	Método isEnabled() = true
	Verifica el texto del enlace	Texto = "Todos los cursos de la plataforma"
	Click sobre el elemento	Se carga la página con URL "https://aulavirtual.uc.edu.py/aulas/claroline/course/platform_courses.php" y tiene el título "Cursos de la plataforma"
Verificar componente "Darse de baja de Curso"	Carga URL https://aulavirtual.uc.edu.py/aulas/index.php	
	Busca el elemento "Darse de baja de Curso"	Solo pasa si existe el elemento ID="Darse de baja de Curso"
	Verifica que esté visible	Método isDisplayed() = true
	Verifica que esté habilitado	Método isEnabled() = true
	Verifica el texto del enlace	Texto = "Darse de baja del curso"
	Click sobre el elemento	Se carga la página con URL "https://aulavirtual.uc.edu.py/aulas/claroline/auth/courses.php?cmd=rqUnreg" y tiene el título "Darse de baja"
Verificar componente "Inscribirse a nuevo Curso"	Carga URL https://aulavirtual.uc.edu.py/aulas/index.php	
	Busca el elemento "Inscribirse a nuevo Curso"	Solo pasa si existe el elemento ID="Inscribirse a nuevo Curso
	Verifica que esté visible	Método isDisplayed() = true
	Verifica que esté habilitado	Método isEnabled() = true
	Verifica el texto del enlace	Texto = "Inscribirse a Curso"
	Click sobre el elemento	Se carga la página con URL "https://aulavirtual.uc.edu.py/aulas/claroline/auth/courses.php?cmd=rqReg" y tiene el título "Inscribirse a curso"

Mientras el *test* manual definido durante la experiencia (según planilla, la definición y ejecución manual necesitó una dedicación de 4 minutos) se componía de solo 3 simples pasos, un modelado de 5 minutos con **MoFQA Modeler**, generó un código de *test* (90 líneas de código propias del *test* más código del *framework* de *test*) que ejecuta 22 pasos intermedios de *test*, asociando cada paso a una comprobación individual.

- Cantidad de tiempo empleadas para la definición y ejecución de *tests* manuales con respecto a la cantidad de tiempo empleada para el modelado, generación y ejecución de *tests* generados.

Análisis de Resultados

La ilustración permitió verificar el ahorro de tiempo que supone el modelado de *tests* de aceptación (según las definiciones de los perfiles MoFQA) utilizando MoFQA Modeler, en comparación a hacerlo utilizando directamente los perfiles UML y una herramienta de edición de UML. Creemos que MoFQA Modeler lleva ventaja porque abstrae varios detalles del modelo, que se presentan transparentes para el usuario. Su uso no solo permite que usuarios sin experiencia en modelado definan *tests* de aceptación sino que, además, permite que usuarios más técnicos puedan modelar dichos *tests* en menor cantidad de tiempo. En la Tabla 2 se visualizan los tiempos obtenidos.

	Modelado en MagicDraw	**Modelado en MoFQA Modeler**
R1	18 min.	5 min.
R2	19 min.	6 min.
R3	20 min.	8 min.
R4	14 min.	3 min.
R5	14 min.	3 min.

Tabla 5: Tiempo (en minutos) para el modelado de cada requerimiento definido en la Experiencia 2 utilizando (i) Perfiles UML y MagicDraw; (ii) MoFQA Modeler.

Si bien es posible visualizar que el tiempo de modelado es mucho menor en todos los casos (un promedio de 12 minutos de ventaja para MoFQA Modeler en todos los requisitos), es importante mencionar que la herramienta tiene limitaciones [10] y no es posible generar todos los elementos disponibles en el perfil UML de MoFQA. Sin embargo, para la mayoría de los casos, las funcionalidades ofrecidas son suficientes para generar los *tests* requeridos.

Conclusión

Este trabajo ha propuesto, primeramente, un modelo para el desarrollo de *software* basado en conceptos teóricos y evidencias halladas en la literatura sobre las ventajas que presentan tanto el proceso TDD como MBT, utilizándose de forma aislada. La propuesta integra ambos enfoques, principalmente, para reforzar (y automatizar) la generación de *tests* utilizando técnicas MBT. Hemos planteado la hipótesis de que, entre otras ventajas, esta integración permitiría que el proceso de desarrollo resulte más productivo respecto al menor tiempo dedicado a la generación de *tests*, la mayor cantidad de *tests* asociados a cada funcionalidad y, mediante la incorporación del usuario final en la definición de *tests* de aceptación, un mayor acercamiento a los requerimientos reales para la implementación de cada funcionalidad. Esta hipótesis, sin embargo, queda como propuesta teórica a ser evaluada por trabajos futuros.

Por otra parte, hemos desarrollado herramientas MBT (que pueden o no ser utilizadas siguiendo el modelo de desarrollo propuesto por **MoFQA**) como propuestas para la definición y generación de *tests* unitarios y de aceptación para sistemas de plataforma web. Para poder llevar a cabo un análisis inicial sobre la utilidad de las herramientas desarrolladas, efectuamos dos pequeñas experiencias: (i) una ilustración comparando los tiempos de definición de *tests* usando los perfiles UML y la herramienta de modelado; (ii) un taller de 90 minutos con alumnos del segundo semestre de Ing. Informática e Ing. Electrónica de la Universidad para obtener una idea inicial sobre la usabilidad del sistema.

Con las experiencias realizadas hemos obtenido un puntaje por debajo del promedio, según las escalas definidas por SUS, en cuanto a la usabilidad de la herramienta de modelado. Creemos, sin embargo, que la experiencia se vio afectada negativamente por inconvenientes técnicos que se presentaron y una limitación respecto al tiempo disponible para la explicación de la herramienta. Por ello nos parece importante (y proponemos como trabajo futuro) la definición de nuevas experiencias, tal vez más formales, para obtener una medida más significativa sobre la usabilidad de la herramienta.

Por otra parte, hemos constatado mediante una ilustración, que el tiempo de definición de *tests* utilizando la herramienta de modelado para *tests* de aceptación que proponemos, es menor al tiempo que lleva su modelado utilizando algún editor UML, como por ejemplo, *MagicDraw*.

Finalmente, vimos que los *tests* generados realizan un trabajo importante de verificación. Relacionamos los requerimientos definidos con respecto a los *tests* generados y verificamos que todos son cubiertos con una serie de pasos intermedios (cada uno realizando una verificación adicional).

Para concluir, creemos que el *testing* de *software* es un área de investigación muy importante que necesita seguir en auge por la necesidad vigente de su implementación y optimización en la industria. Necesitamos acercarnos a las necesidades que surgen en este ámbito y vemos que un impedimento para su utilización sigue siendo el trabajo y tiempo que conlleva su ejecución. Herramientas y métodos que busquen hacer del *testing* un proceso más fluido pueden causar un alto impacto en la productividad de los equipos de desarrollo y en la calidad de los productos de *software*. Creemos que el trabajo realizado en esta tesis fue diseñado teniendo en cuenta éstos y otros requerimientos por lo que vale la pena darle continuidad.

A partir de los trabajos iniciados durante el desarrollo de esta tesis y de los resultados obtenidos, se proponen como trabajos futuros para continuar con la investigación:

- Definición y ejecución de un experimento o caso de estudio para validar el modelo de desarrollo de *software*, basado en TDD y prácticas MBT, propuesto. Creemos que son factores importantes para la validación de resultados: experiencia en TDD con que cuenta el equipo de desarrollo, dominio de desarrollo de *software* y plataforma a la cual está orientado, herramientas MBT utilizadas.
- Generalización de las reglas de transformación de forma a que los tests no estén únicamente orientados a la plataforma destino seleccionada en este trabajo.
- Definición y ejecución de un experimento o caso de estudio para validar la usabilidad de **MoFQA Modeler**.
- Ampliación de los perfiles UML y las reglas de transformación para la generación de *tests* más enriquecidos que aprovechen más funcionalidades que pueden ser implementadas mediante los *frameworks* de *test*.
- Extensión de perfiles UML para la definición de *tests* unitarios más enriquecidos.
- Inclusión de nuevos elementos en los perfiles y reglas de transformación para la definición de *tests* de integración y análisis de la utilidad que representan para el proceso global de *testing*.
- Extensión de herramientas de modelado y reglas de transformación para otros dominios de desarrollo, no necesariamente de plataformas web.

Referencias

[1] G. J. Myers, The art of software testing, Word Association, Inc., 2004.

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

[3] B. Kent, Test driven development: by example, 2002.

[4] B. A. Shappee, "Test First Model-Driven Development," 2012.

[5] N. Rafique, N. Rashid, S. Awan and Z. Nayyar, "Model Based Testing in Web Applications," *International Journal of Scientific Engineering and Research (IJSER), vol,* vol. 2, no. 1, 2014.

[6] J. M. Wing, "A specifier's introduction to formal methods," *Computer,* vol. 23, no. 9, pp. 8-22, 1990.

[7] C. Jones, P. O'Hearn and J. Woodcock, "Verified software: A grand challenge," *IEEE Computer,* vol. 39, no. 4, pp. 93-95, 2006.

[8] P. C. Jorgensen, Software testing: a craftsman's approach, CRC press, 2013.

[9] P. Baker, Z. R. Dai, J. Grabowski, I. Schieferdecker and C. Williams, Model-driven testing: Using the UML testing profile, Springer Science \& Business Media, 2007.

[10] S. Kumar and S. Bansal, "Comparative study of test driven development with traditional techniques," *Int J Soft Comput Eng (IJSCE),* vol. 3, no. 1, pp. 2231-2307, 2013.

[11] J. Canós, P. Letelier and M. C. Penadés, "Metodologías Ágiles en el desarrollo de Software," *Universidad Politécnica de Valencia, Valencia,* 2003.

[12] J. Shore and others, The art of agile development, " O'Reilly Media, Inc.", 2007.

[13] S. W. Ambler, "Introduction to test driven development (TDD)," *Agile Data,[En línea]. Available: http://www. agiledata. org/essays/tdd. html.,* 2006.

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

[15] M. Brambilla, J. Cabot and M. Wimmer, Model-driven software engineering in practice, vol. 1, Morgan \& Claypool Publishers, 2012, pp. 1-182.

[16] J. HOFSTADER, "Model-Driven Development Applied to Microsoft Visual Studio, 2006.[cited October 2009]," *Available by Internet:< http://msdn. microsoft. com/en-us/library/aa964145. aspx.*

[17] S. Yenduri and A. L. Perkins, "Impact of Using Test-Driven Development: A Case Study.," in *Software Engineering Research and Practice*, 2006.

[18] S. Keele, "Guidelines for performing systematic literature reviews in software engineering," in *Technical report, Ver. 2.3 EBSE Technical Report. EBSE*, 2007.

[19] Z. A. Barmi, A. H. Ebrahimi and R. Feldt, "Alignment of requirements specification and testing: A systematic mapping study," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, 2011.

[20] 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*, 2007.

[21] J. Peleska, "Industrial-strength model-based testing-state of the art and current challenges," *arXiv preprint arXiv:1303.1006*, 2013.

[22] B. Marin, T. Vos, G. Giachetti, A. Baars and P. Tonella, "Towards testing future web applications," in *Research Challenges in Information Science (RCIS), 2011 Fifth International Conference on*, 2011.

[23] 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*, 2011.

[24] Q. Xie and A. M. Memon, "Model-based testing of community-driven open-source GUI applications," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, 2006.

[25] A. Avram, Domain-driven design Quickly, Lulu. com, 2007.

[26] Zephyr: Real-Time Test Management, "QA Metrics: The Value of Testing Metrics Within Software Development," 2015.

[27] J. Brooke and others, "SUS-A quick and dirty usability scale," *Usability evaluation in industry,* vol. 189, no. 194, pp. 4-7, 1996.

[28] J. Sauro, *Measuring usability with the system usability scale (SUS),* 2011.

[29] A. Bangor, P. Kortum and J. Miller, "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale," *J. Usability Studies,* vol. 4, no. 3, pp. 114-123, #may# 2009.

[30] L. Luo, "Software Testing Techniques: Technology Maturation and Research Strategy," *Class Report for,* 2001.

[31] P. Bourque, R. E. Fairley and others, Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0, IEEE Computer Society Press, 2014.

Apéndice A. Manual de Usuario de MoFQA Modeler

A continuación se explicará cómo utilizar la herramienta **MoFQA Modeler**[49] para la de requerimientos o criterios de aceptación que derivarán a los *tests* de aceptación. Cada explicación será acompañada por un gráfico para su mayor comprensión.

Cómo iniciar el modelado

Si MoFQA Modeler fue utilizado previamente para modelar requerimientos del sitio, busque el sitio deseado en la lista. Selecciónelo y click en "Modelar".

Si va a utilizar MoFQA Modeler por primera vez para el sitio deseado, click en "Nuevo Sitio". La siguiente pantalla le solicitará un nombre para identificar al sitio a modelar:

Definición de Requerimientos

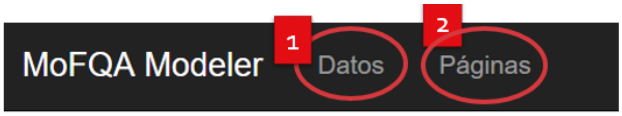
El usuario de **MoFQA Modeler** puede definir los requerimientos de una aplicación web referentes a:

- Contenido de las diferentes páginas de la aplicación.
- Interacción con dichos contenidos.

[49] Disponible en http://www.dei.uc.edu.py/proyectos/mddplus/herramientas/mofqa/modeler/.

Para la definición de requerimientos puede definir datos de prueba que luego se utilizarán para verificar dichos requerimientos en los tests de aceptación.

Así, **MoFQA Modeler** cuenta con dos opciones principales para la definición de requerimientos: Datos y Páginas.



Datos de Prueba

Todos los datos de prueba a utilizarse en los requerimientos deben ser definidos antes en **MoFQA 1.0**, utilizando la opción "Datos" del menú principal.

Los datos de prueba pueden definirse mediante la creación de planillas de datos. Cada planilla permite almacenar datos relacionados.

Una planilla en **MoFQA Modeler** cumple la misma función que una tabla en Excel: almacenar registros (filas de la tabla) que tienen las mismas características (columnas).

Ejemplo: planilla con datos de clientes de una empresa.

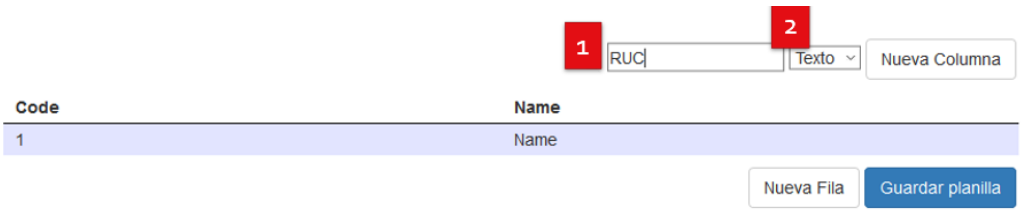
Nombre	Dirección	Teléfono	RUC	¿Activo?
Vanessa Martínez	Carretera de López 232	903-365	2.353.125-5	Sí
Daysi López	Ygatymi 566	559-545	353.155-0	No
Ernesto Bareiro	Jejuí c/ O'leary 211	568-344	1.594.244-8	Sí

Para definir plantillas de datos de prueba: Ingresar a "Datos" y seleccionar "Nueva Planilla".

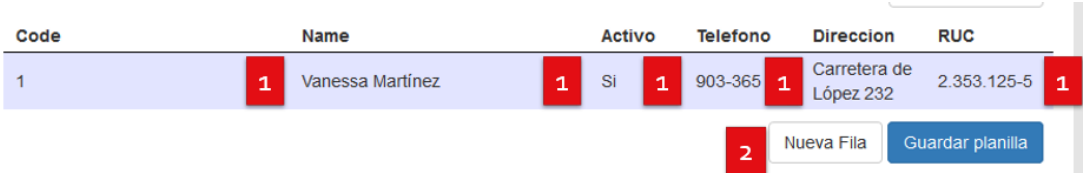
Una planilla se compone de filas (datos de prueba) y columnas (propiedades de los registros). Toda planilla debe tener las columnas Código y Nombre (creadas automáticamente).

Una columna puede ser de tipo texto (el valor de la columna se expresa en texto) o puede hacer referencia a un registro de otra planilla.

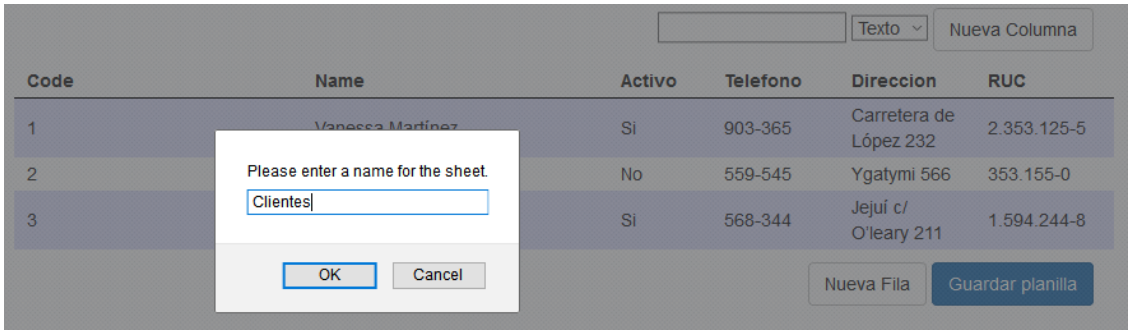
Agregar columnas a la plantilla: (1) ingresar el nombre de la columna; (2) tipo de datos de la columna.



Al hacer doble click sobre cada columna de una fila de la planilla, es posible editar su valor. De esa forma vamos llenando de valores las columnas de cada fila de la planilla (1). Para agregar nuevas filas, seleccionar la opción "Nueva Fila" (2).



Una vez cargados todos los datos de la planilla, se procede a guardar la planilla:



Las planillas de datos pueden almacenar registros que comparten el mismo valor para una o varias columnas. Es posible crear agrupaciones de datos a las cuales hacer referencia cuando no son relevantes todos los campos de un registro de planilla.

Ejemplo para la planilla de clientes:

Grupo	**¿Activo?**
Clientes Activos	Sí
Clientes Inactivos	No

Para cada plantilla creada, es posible definir grupos de datos: (1) seleccionar la opción “Nueva clasificación/grupo de datos para la plantilla”; (2) asignar un nombre al grupo y dar valor a los campos de interés. Ambos pasos se ilustran en las dos imágenes siguientes:

Requerimientos en función al Contenido

MoFQA Modeler permite definir los componentes para las páginas de un sitio web. Las páginas son elementos con una URL única que pueden estar compuestas por uno o más:

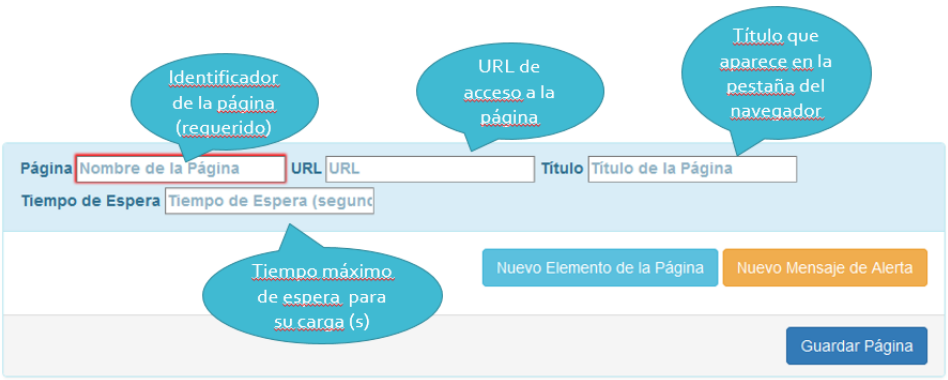
- Componentes de páginas: textos, campos de formulario, botones o vínculos y DIVs contenedores.
- Mensajes de alerta: si la interacción con algún componente

Las páginas son agrupadas en historias de usuario (funcionalidad requerida para el sitio web).

En la opción “Páginas”, crear una nueva “Historia de Usuario” con el nombre del requerimiento actual:

Se debe asignar un nombre a la historia y al pulsar *Enter* es creada la historia. Luego de hacer click sobre la historia creada, seleccionar la opción “Nueva Página” para agregar las páginas relacionadas al requerimiento a definir:

Una página tiene las siguientes propiedades:

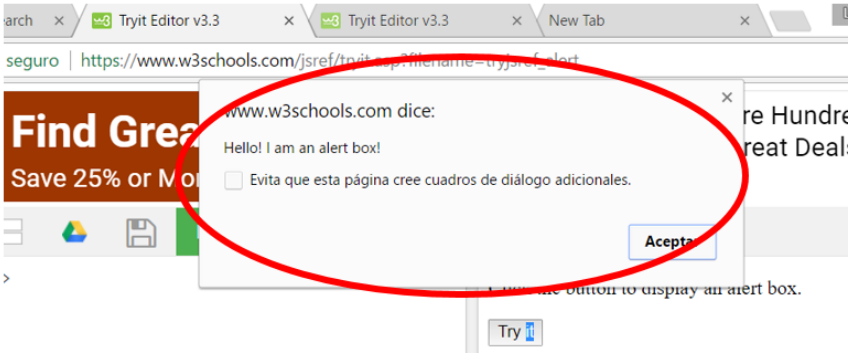


Una página puede tener elementos/componentes y mensajes de alerta. Se crea un elemento/componente (datos, texto, campos de formulario, DIVs contenedores) como parte de los requerimientos de contenido. Para los mensajes de alerta esperados como resultado de las interacciones del usuario con los elementos/componentes, se definen mensajes de alerta para la página actual.

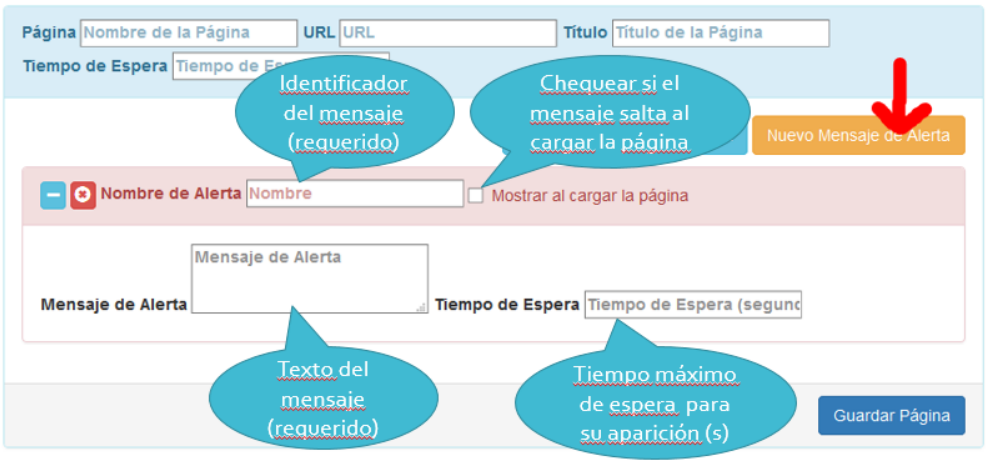
Una vez creados todos los elementos de la página (mensajes de alerta y componentes), es necesario guardar página.

Mensajes de Alerta

El navegador web puede lanzar mensajes de alerta en caso de error, solicitar datos al usuario o simplemente para desplegar un mensaje importante. Un ejemplo de mensaje de alerta puede verse en la siguiente imagen:



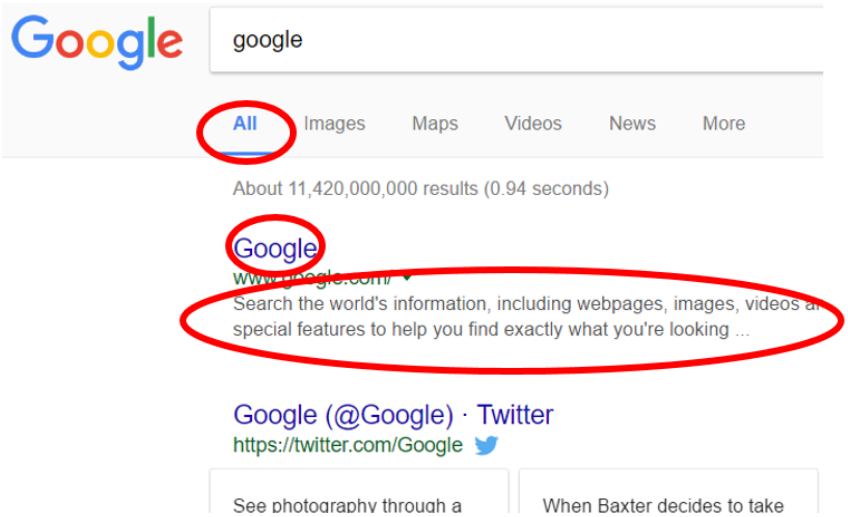
MoFQA Modeler permite definir la aparición de mensajes de alerta para cada página, seleccionando la opción "Nuevo Mensaje de Alerta" de la página donde se desea que aparezca. Un mensaje de alerta tiene las propiedades:



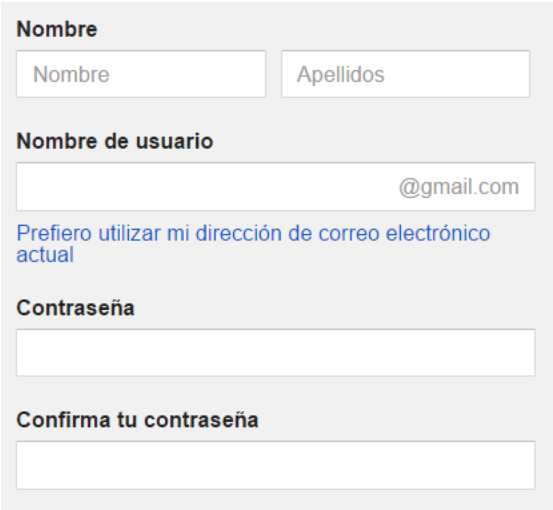
Elementos o Componentes de Páginas

Un componente/elemento de página puede ser de tipo: texto, campo de formulario, DIV contenedor o botón (enlace).

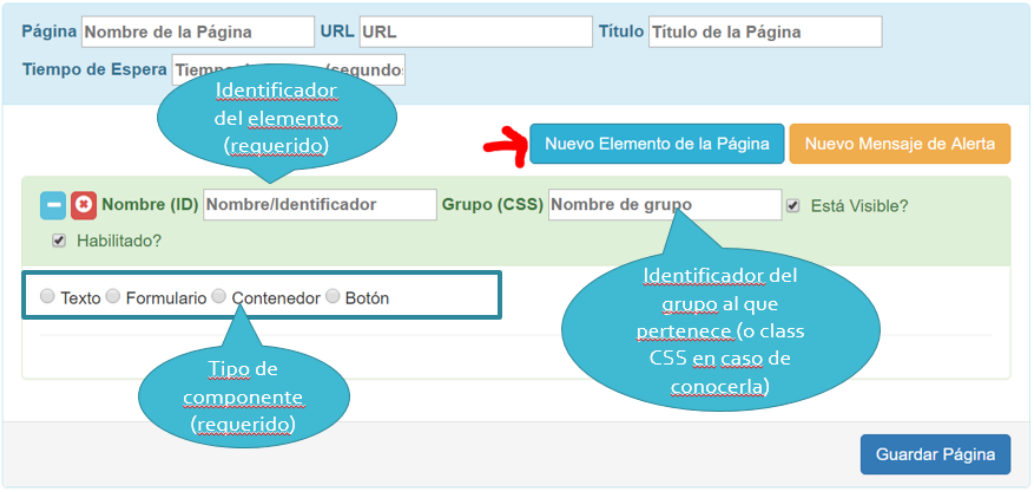
Como su nombre lo indica, si un elemento es de tipo texto, éste identifica a una cadena de caracteres en una página web. Un ejemplo de elemento de tipo texto es presentado a continuación:



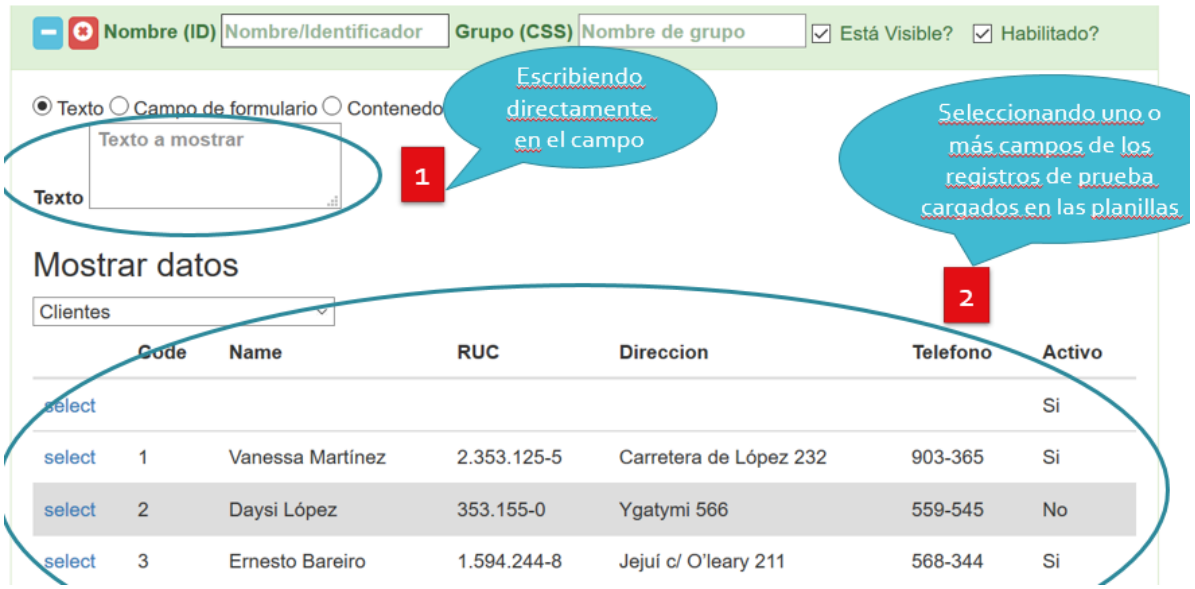
Un elemento de tipo formulario representa a un elemento <form> HTML con campos de texto, en **MoFQA Modeler** el elemento hace referencia a un conjunto de campos de tipo texto de un formulario. Por ejemplo:



Un elemento o componente puede ser creado seleccionando la opción "Nuevo Elemento de la Página" y tiene las propiedades:



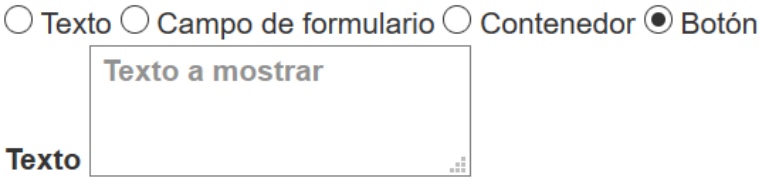
Un elemento de tipo texto hace referencia al texto que debe visualizarse en la página que se está definiendo. El texto a visualizarse puede especificarse de dos formas:



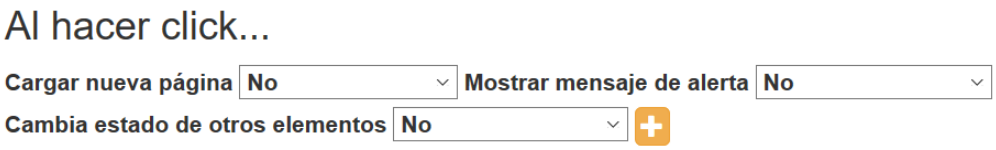
Si el elemento que se desea crear es de tipo formulario, se debe seleccionar un registro definido en una de las planillas. El registro (con sus campos seleccionados) es el que se usará como dato de prueba.

Un elemento de tipo contenedor permite definir elementos HTML contenedores de otros elementos:

Un elemento de tipo botón permite definir botones/enlaces dentro de las páginas. Se debe indicar el texto que se visualizará en el botón/enlace:



Los elementos de tipo texto y botón pueden desencadenar una acción tras hacer click sobre él. Si un elemento de tipo texto desencadena una acción luego de hacer click sobre él, se comporta como un enlace. Las acciones serán verificadas en las pruebas. Acciones posibles: navegar a otra página, mostrar un mensaje de alerta, cambiar el estado (datos que se visualizan, habilitar/deshabilitar, hacer visible/quitar visibilidad, grupo CSS al que pertenece). En la imagen se visualiza la opción que permite configurar las acciones sobre cada elemento creado.



Generación de *Tests* Abstractos

Luego de la definición de todos los elementos del sitio, es posible generar los *tests* abstractos. Para ello, se debe seleccionar la opción “Generar Modelos” en la cabecera. Nos dará la opción de descargar un archivo comprimido donde se encuentran los perfiles UML definidos para los *tests* en **MoFQA** y un archivo testModels.uml que representa los *tests* abstractos en formato EMF, versión 5.

En conjunto con las reglas de transformación, estos archivos pueden ser importados a un proyecto Eclipse (con la herramienta Acceleo instalada) para, posiblemente, enriquecer el modelo y agregarles nuevos elementos a los *tests* y, finalmente, generar los *tests* ejecutables.

Limitaciones de MoFQA Modeler

En esta sección presentamos algunos puntos que consideramos podrían mejorar la usabilidad de la herramienta y permitir la generación de *tests* más enriquecidos:

- No es posible definir campos <<*required*>> para los elementos <<*domainElement*>>: para ello se necesitaría habilitar una opción para identificar cada columna como requerida o no, durante la creación de planillas.
- Tampoco es posible definir operaciones para los elementos <<*domainElement*>>.
- Las columnas de las planillas (atributos de elementos <<*domainElement*>>) solo pueden pertenecer a los tipos String y otros <<*domainElement*>>.
- Cuando una columna es del tipo de otra planilla, debe utilizarse el ID generado para el registro deseado. Esto puede resultar confuso y tedioso para el usuario.
- Las asociaciones que se generan entre elementos <<*domainElement*>> solo son 1 a 1.
- En general, ningún elemento creado con **MoFQA Modeler** (plantilla, particiones/grupos de datos, páginas y componentes de páginas) puede ser modificado luego de ser grabados. Si hubo algún tipo de error, se debe eliminar el elemento para volver a crearlo.

Apéndice B. Planilla de Experiencia de Usuario

En este apéndice, se describe el trabajo que realizaron los alumnos durante la experiencia de validación. Los mismos debían seguir los pasos indicados y completar las tablas presentadas en la planilla que aparece en la siguiente sección.

MoFQA Modeler: Experiencia Usuario

1. Seleccione al menos un requerimiento de la siguiente lista para el sitio web https://aulavirtual.uc.edu.py/aulas/ para luego validar su cumplimiento:

Requerimiento	**Descripción (opcional)**
Texto bienvenida (usuario no autenticado)	Al cargar la página, aparece el texto: "Bienvenido a este sitio".
Enlace a UC Abierta	Al cargar la página, aparece el enlace: Universidad Católica "Nuestra Señora de la Asunción". El enlace redirecciona a la página http://www.uc-abierta.uc.edu.py/.
Login no válido	Se visualiza un formulario para inicio de sesión. Los campos son: Nombre de usuario, Contraseña. Además tiene un botón "Entrar". Al probar con el usuario: Nombre: pepito Contraseña: 12345 Debe saltar un mensaje de alerta con el mensaje "Los datos no son válidos"
Login válido	Se visualiza un formulario para inicio de sesión. Los campos son: Nombre de usuario, Contraseña. Además tiene un botón "Entrar". Al probar con el usuario: Nombre: juana Contraseña: 12345 Debe cargar la página https://aulavirtual.uc.edu.py/aulas/, donde se visualiza el mensaje "MI ESCRITORIO".
Mi Escritorio	Se visualizan los enlaces: "Inscribirse en un nuevo curso": https://aulavirtual.uc.edu.py/aulas/claroline/auth/courses.php?cmd=rqReg&categoryId=0. "Darse de baja del curso": https://aulavirtual.uc.edu.py/aulas/claroline/auth/courses.php?cmd=rqUnreg "Todos los cursos de la plataforma": https://aulavirtual.uc.edu.py/aulas/claroline/course/platform_courses.php
Cursos de la plataforma	Al acceder a la página https://aulavirtual.uc.edu.py/aulas/claroline/course/platform_courses.php, se ve la lista de todos los cursos cargados en la plataforma Claroline.

2. Para los requerimientos seleccionados, ¿cómo definiría los *tests* de aceptación? Para completar la tabla, verifique el tiempo que le llevó definir los pasos de cada *test*.

Requerimiento	Pasos a seguir para la prueba	Cómo saber si tuvo éxito o no	Tiempo de definición de *tests*

3. Ejecute cada uno de los *tests* definidos y complete la tabla a continuación:

Requerimiento	Tiempo de ejecución de *tests*	¿Fue exitosa la prueba?

4. A continuación, utilizaremos MoFQA Modeler para definir los requerimientos seleccionados. Para ello, siga los pasos:

   a. Ir al enlace: http://solucionesintegral.com/mofqa/register.php.

   b. Crear un nuevo sitio con su nombre y apellido, click en Agregar.

   c. Modelar los requerimientos seleccionados.

   d. Completar la tabla:

Requerimiento	Dudas/Complicaciones/Comentarios sobre la herramienta	Tiempo de modelado

5. Finalmente, complete el cuestionario:

a. Pienso que me gustaría utilizar MoFQA frecuentemente para definir tests de aceptación para una aplicación web.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**

b. La herramienta MoFQA me parece innecesariamente complicada.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**

c. MoFQA me resultó fácil de utilizar.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**

d. Pienso que necesitaría el soporte técnico de una persona para poder utilizar la herramienta.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**

e. Creo que las funcionalidades de la herramienta están bien integradas.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**

f. Me parece que hay mucha inconsistencia en la herramienta.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**

g. Me parece que la mayoría de las personas aprenderían muy rápido a utilizar MoFQA.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**

h. Me resultó muy incómodo utilizar MoFQA para definir los tests de aceptación.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**

i. Me siento seguro utilizando MoFQA para definir tests de aceptación.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**

j. Necesité aprender muchas cosas antes de poder utilizar MoFQA.

Fuertemente en desacuerdo				**Fuertemente de acuerdo**
1	**2**	**3**	**4**	**5**