MoFQA: Una propuesta para la generación automática de *tests* a partir de modelos siguiendo el proceso TDD[1]

Primera Presentación

Alumna: Linda Riquelme **Co-Tutor:** Dr. Luca Cernuzzi

Tutor: Ing. Magalí González **Colaborador:** Ing. Nathalie Aquino

1. 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]. La esencia del *software testing* 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 [2].

Realizar pruebas sobre la ejecución de un programa utilizando todas las combinaciones y condiciones de entrada posibles no resulta factible (ni siquiera si hablamos de un producto pequeño y simple). La cantidad de defectos, además, puede ser muy grande; muchos de ellos pueden ocurrir de forma tan infrecuente que son difíciles de detectar. Otro punto a considerar es la dimensión no funcional de la calidad del *software*[2], ésta puede llegar a ser bastante subjetiva, lo que dificulta su verificación.

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 *test cases* más óptimos (es decir, con la menor cantidad posible de *tests*, encontramos un mayor número de errores) así como algunas asunciones o abstracciones que los *testers*[3] 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). Creemos que de esta forma podemos garantizar la obtención de un producto de mejor calidad. Con este objetivo, nos centraremos en dos enfoques principales: *Model-Based Testing* y *Test-Driven Development*.

Model-Based Testing **(MBT)** [3] aparece como una propuesta MDD (*Model-Driven Development*)[4] 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[5]. ***Test-Driven Development*** **(TDD)** [4], 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.

TDD es una práctica definida en el *Agile Manifesto*[6] para el desarrollo de *software* con el objetivo de obtener una mayor cantidad de *tests* asociados al producto final que permitan determinar la calidad del mismo. Sin embargo, la definición de *tests* es un paso que se realiza, generalmente, de forma manual: se elaboran *scripts* utilizando *frameworks* y herramientas específicas de *testing*, pudiendo tener finalmente 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 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

---

[1] Este trabajo está siendo desarrollado en el marco del proyecto "Mejorando el proceso del desarrollo de *software*: Una propuesta basada en MDD" (14-INV-056), financiado por el Consejo Nacional de Ciencias y Tecnología (CONACYT).

[2] 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 [24].

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

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

[5] 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

[6] El *Agile Manifesto* es una declaración formal, redactada en febrero del 2001, en la cual se promulgan 4 principios básicos y 12 prácticas a tener en cuenta en la aplicación de las metodologías ágiles para el desarrollo de *software*. Se encuentra disponible en: http://agilemanifesto.org/.

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 [5]) 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[7] 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. Con ese objetivo, dedicaremos especial atención a reducir la dificultad en la definición de *tests*, a partir de la utilización de técnicas ampliamente adoptadas en el desarrollo ágil (en lo que respecta a la definición de requerimientos por parte del usuario, a partir de historias de usuario[8]).

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 *test cases* 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 [6]: "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.
- Enriquecimiento de perfiles UML[9] para la definición de *tests*[10] aplicables a sistemas de plataforma Web.
- Desarrollo de una herramienta integrada para el modelado, generación y ejecución de *tests* orientada a la plataforma definida para los propósitos de este trabajo.
- 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. Además, siguiendo el proceso TDD, se requiere que cada funcionalidad tenga una serie de *tests* relacionados en la medida que se avanza con la implementación. Finalmente, la reutilización de perfiles UML para definir los modelos que soportará la herramienta a implementar, tiene la finalidad de permitir la representación de *tests* a partir de modelos conocidos ampliamente (y simplificados en relación a otras notaciones matemáticas [7] más formales). Algunas notaciones candidatas para la definición de modelos en nuestra propuesta son: casos de uso, historias de usuario, diagramas de clase, diagramas de estado UML.

Se promoverá además la utilización de técnicas y prácticas de la metodología ágil que permitan: obtener productos entregables en menor plazo de tiempo (y consecuentemente, la retroalimentación por parte del cliente se produce de forma más temprana), anticiparse a la posibilidad de cambios de requerimientos del *software* (lo que implica la posibilidad de reducción de costos de desarrollo). Son prácticas que destacamos: la participación activa del usuario final en el proceso, la definición de *tests* para cada funcionalidad a implementar, el desarrollo iterativo del producto basado en historias de usuario.

En las siguientes secciones de este documento se presentarán algunos conceptos importantes (todos ellos considerados en la definición de la propuesta) sobre *software testing*, TDD y MBT. Mencionaremos algunas características de las herramientas existentes y las limitaciones encontradas en ellas. Finalmente, incluimos una descripción de nuestra propuesta y un plan de implementación.

2. SOFTWARE TESTING

Con el objetivo de asegurar la calidad del *software*, bajo ciertos criterios definidos previamente, el *software testing* es un proceso que permite encontrar errores en el producto final. En esta sección, presentamos los aspectos que consideramos se deben tener en cuenta para la definición de un plan de *testing*.

2.1. Estrategias de testing

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

a- ***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.

[7] 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

[8] **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). [23]

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

[10] Se definirán los modelos en base a UTP (*UML Testing Profile*) [8] que consiste en una extensión UML con conceptos para *testing* de *software*.

b- ***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 [1]. 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: (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 [1] en el capítulo 4 *"Test-Case Design"*). Según [1], 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 [2], 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*.

2.2. 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. 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. Se destacan por ejemplo: *tests* de aceptación (verifican que el programa cumpla con los requerimientos definidos por el usuario), *tests* de módulos individuales, *tests* de módulos integrados y las interfaces entre ellos.

La literatura distingue distintos procesos de desarrollo de *software* [8], entre ellos: el modelo en cascada, el modelo espiral, el proceso unificado, el *V-model* y el *W-model*. 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*:

1- Se generan los requerimientos (metas para el producto final) a partir de las necesidades del usuario.
2- 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.
3- Se procede el 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.
4- Se especifica la función de cada uno de los módulos.
5- 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* [8]:

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[11], 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* [1]: *facility testing*, *stress testing*, *usability testing*, *security testing*, *performance testing*, *storage testing*, *configuration testing*, *compatibility 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*.

[11] La integración de módulos puede realizarse usando técnicas *top-down* o *bottom-up* [1].

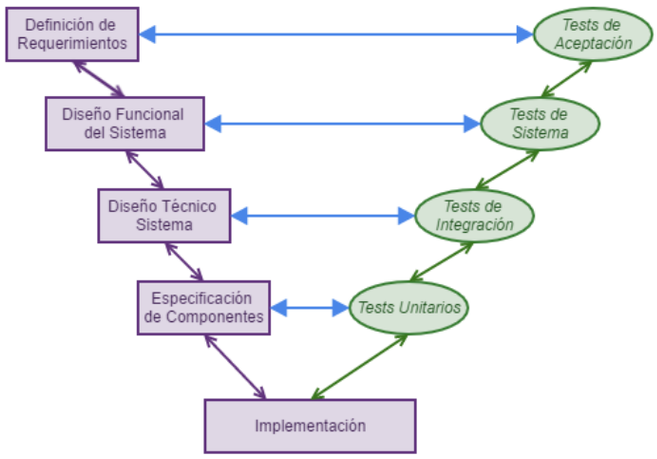

Figura 1: Modelo V para el desarrollo de *software* [8]

2.3. Test-Driven Development

Test-Driven Development (TDD) es un proceso de desarrollo de *software* fundado en los conceptos de programación de *test-first* usados en *extreme programming*[12]. La filosofía usada en TDD es *Red-Green-Refactor*, consistente en un desarrollo iterativo tanto de los *tests* como del producto final.

En TDD, el desarrollador inicia el proceso escribiendo un *test case* ejecutable que será utilizado para verificar una funcionalidad a ser implementada. Se espera que el primer *test* falle (*Red*) puesto que, inicialmente, la funcionalidad no está incluida. Cuando un *test* falla, el programador escribe el código necesario para pasarlo y, 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, permite que el código final resulte más claro y simple, manteniendo la seguridad de que todos los *tests* anteriores vuelvan a pasarse exitosamente. La **Figura 2** grafica el proceso descripto.

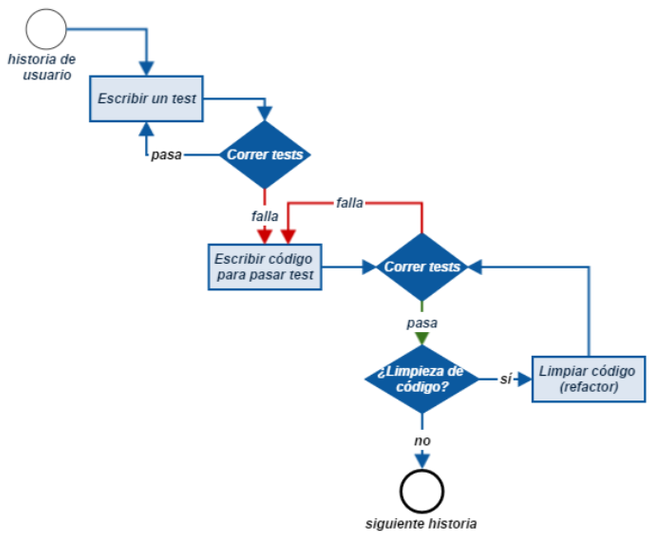

Figura 2: Proceso TDD

2.3.1. Ventajas de TDD

Kent Beck, uno de los mayores contribuyentes en el área de metodologías ágiles y TDD, menciona en [4] 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.
- 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.

[12] ***Extreme Programming*:** es una metodología ágil de 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 realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. [23]

- 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.

2.3.2. Limitaciones de TDD

En cuanto a las limitaciones, [4] menciona los puntos:

- El *testing* basado en *unit tests* no resulta suficiente para situaciones donde es requerido un *testing* funcional completo.
- 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.
- 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 dejan el *testing* como actividad secundaria.

2.3.3. 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.

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

Acceptance TDD

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 *unit tests* o los *integration tests*) 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 es una herramienta para los desarrolladores que los ayuda a crear unidades de código bien escritas para realizar una serie de operaciones. ATDD, por su parte, es una herramienta de comunicación entre el cliente, el desarrollador y el *tester* para asegurar que los requerimientos estén bien definidos. TDD requiere automatización[14] mientras que ATDD no (pero su automatización ayudaría a la realización de un *regression testing*).

En la **Figura 3** se diagrama la forma en que ATDD y TDD pueden ser utilizados en conjunto para el desarrollo de *software* [9]. Este diagrama sirvió como base para la definición del método, presentado en las siguientes secciones de este documento. Como nos parece clave la intervención del usuario en la definición de requisitos (y consecuentemente, de los *tests* de aceptación), incorporaremos los conceptos de ATDD a nuestra propuesta.

[13] Uno de los principios del *software testing* incluidos por Myers en [1], 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.

[14] En este contexto, la automatización se refiere a la ejecución automática de *tests* (definidos a partir de códigos ejecutables) y la validación de resultados. No se incluye la generación automática de *tests* a partir de modelos u otras abstracciones.

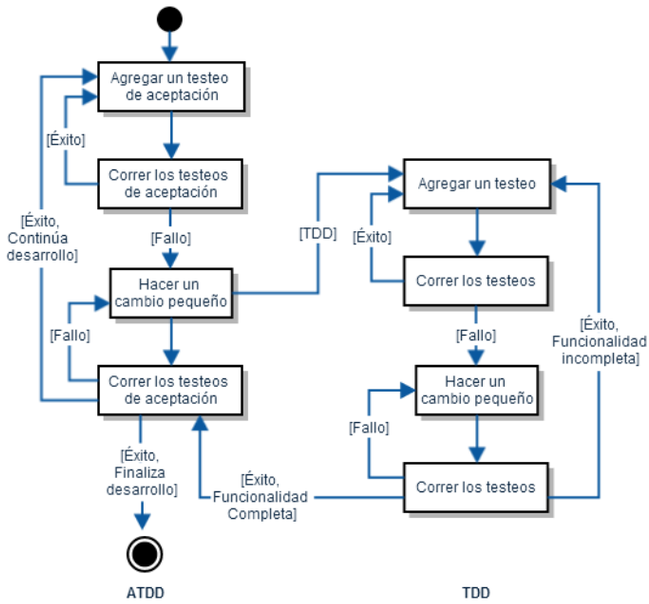

Figura 4: Implementación de TDD y ATDD en conjunto [9]

2.4. Testing Automatizado

En esta sección se mencionarán varias técnicas de *testing* utilizadas en la industria (según la clasificación presentada en [10]). La comparación a realizar nos permitirá introducir las ventajas ofrecidas por los enfoques que deseamos resaltar en este trabajo.

1. ***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*.
2. ***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.
3. ***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*. 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. Para superar dicho inconveniente se requiere altos niveles de abstracción en la definición de *scripts*.
4. ***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).

Para *testing* automatizado se usan herramientas como *Fitnesse*[15] o *RSpec*[16] (orientadas a la definición de *tests* de aceptación por parte de los usuarios), mientras que los desarrolladores que utilizan metodologías ágiles comúnmente usan la familia de herramientas *open source xUnit* como *JUnit*[17] o *VBUnit*[18]. 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***, el diseño 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.

3. Model-Driven Development

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

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

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, 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 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 de una herramienta que permita el diseño y desarrollo de la solución (que soporte la generación automática de código) facilitaría la sincronización entre los diagramas y el código, resultando en que los diagramas terminan siendo representaciones reales de la solución implementada.

3.1. Model-Based Testing

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 [3]:

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 en el modelo, podrán ser detectados muy probablemente cuando los *tests* sean ejecutados.
3- Generación de *tests* abstractos[19] 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.

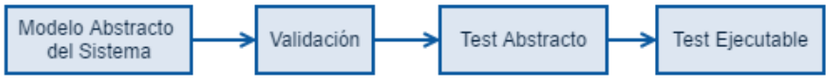

Figura 5: Fases de *Model-Based Testing*

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*).

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 [3]:

¿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.

[19] 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.

- 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.

3.2. Integración de MBT y TDD

Como hemos mencionado en las secciones anteriores, TDD consiste en un conjunto de prácticas para llevar a cabo el proceso del desarrollo de *software* usando *tests* como base. La generación de dichos *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 trabajo de 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[20].

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]**:

- 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 y *test cases.*

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* en base a *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 las siguientes secciones, se verá con mayor detalle la definición de la propuesta.

4. Definición de la Propuesta

En esta sección, presentaremos inicialmente un análisis de trabajos realizados, en las áreas de interés, a la fecha de redacción de este documento. Dicho análisis nos llevó a encontrar algunas debilidades y/o limitaciones de MBT en la actualidad; mencionaremos las más relevantes para los fines de este trabajo y, finalmente, destacaremos aquellas que constituirán la base de nuestra propuesta. Para concluir, se planteará una hipótesis en base a determinados argumentos teóricos, y se propondrá un método, a implementar como parte de este trabajo, para obtener algunos resultados representativos.

4.1. Estado del Arte

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

Para la búsqueda de trabajos, generamos una cadena de búsqueda adaptando las técnicas propuestas en [13]:

1- Teniendo en cuenta los objetivos de este estudio, se identifican los siguientes aspectos:
   a. Población: trabajos realizados a la fecha de MBT y TDD orientados o no al entorno Web.
   b. 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.
   c. 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**.
2- Se enumeran las palabras claves, abreviaciones y sinónimos que identifican a:

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

a. los enfoques que deseamos unificar (MBT y TDD);
b. las metodologías de desarrollo de *software* relacionadas;
c. 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.
4- La cadena de búsqueda obtenida resulta a partir del enlace de las siguientes listas, utilizando el conector lógico **AND**:
   a. 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
   b. ((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 existentes. La **Tabla 1** 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 existentes 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*: Además de las herramientas, se encontró evidencia de que en la práctica aún resulta complejo definir *tests* (a pesar de la abstracción otorgada por los modelos), razón por la cual MBT 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 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.

Las limitaciones halladas en los trabajos seleccionados, según los criterios de análisis definidos, a continuación:

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 base ya existente [5].

Relación de *tests* con requerimientos del *software*

En [14] 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, [5] 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 [15], se mencionan algunas razones: (1) los usuarios no siguen un mismo patrón de comportamiento para la ejecución de una aplicación por lo que se complica la definición de *tests* para estos aspectos; (2) algunas

[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.

características tales como flujo en red, usabilidad y seguridad no poseen una forma clara de definición. De los trabajos analizados en [15], solo el 8% tiene en cuenta algunos aspectos no funcionales (ejemplo: control de acceso, medidas de *performance*, seguridad, usabilidad, confiabilidad). En [14] 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 [14].

Título	Tipo de Informe	Año	Tipo de Estudio	Objeto de Estudio
Test First Model-Driven Development [5]	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 [14]	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 [15]	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 [16]	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 [17]	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 [18]	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 [19]	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 [6]	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.

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 [15] se destaca la falta de integración de técnicas MBT al proceso de desarrollo de *software*, esto 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, [15] 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, [16] recalca que el modelado de *tests* sigue siendo un desafío; al respecto menciona:

a. Para sistemas complejos, los modelos necesitan abstraer una gran cantidad de detalles, de otra manera los modelos resultarían inmanejables.

b. 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 [15] demuestran: el 46% de los trabajos analizados usan notaciones distintas de UML y cubren *tests* basados en requerimientos (*functional tests*), 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*, [17] 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, muchos 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 lo mismo se aplica a trabajos que realizan ingeniería inversa: 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 [18].

Adecuadas para *web testing*

El *testing* de aplicaciones *web* es un proceso muy importante ya que es el área de mayor crecimiento en ingeniería de *software* [6]. 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 [6]: 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 [17] 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 [17] 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 [19], 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 [17]. 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. 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 [17].

[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*. [22]

[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).

Otros datos recopilados, que pueden resultar de interés:

- En [15] 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*, [15] 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.

4.2. 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 los casos de prueba 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.
- Se requiere minimizar los conocimientos previos necesarios para la implementación de técnicas actuales de MBT: lenguaje de modelado, criterios de cobertura de *tests*, formatos de salida generados, 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 la siguiente sección se define la propuesta de este trabajo para hacer frente a las limitaciones destacadas en el párrafo anterior. La propuesta se limita a un dominio específico de aplicación: desarrollo de aplicaciones Web 2.0 utilizando técnicas propuestas por metodologías ágiles; los *tests* que la herramienta a implementar permitirá generar estarán basados en las necesidades propias del ambiente *web*.

4.3. Alcance de la Propuesta

En este trabajo se desea destacar la importancia del *testing*, tal como es destacada en el artículo citado en [20]: “Es necesario que la informática justifique la confianza que la sociedad deposita 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”.

Hemos visto las ventajas de MBT en lo que respecta a la abstracción y generación automática de *tests*, por lo que apuntamos inicialmente a la implementación de una herramienta MBT que permita generar *tests* en base a modelos sencillos tanto para el usuario final como para el programador. En particular, nos centraremos en sistemas de plataforma Web 2.0. En este sentido, debemos considerar los aspectos particulares para posibilitar la definición de *tests* en dicha plataforma de destino. Por otra parte, creemos que el desarrollo de *software* puede producir resultados de forma más temprana y verificada, si se realiza en base a las prácticas definidas por TDD. Por lo tanto, en este trabajo se propone un método que integra MBT y TDD al proceso de desarrollo, que será posteriormente implementado en una herramienta con las características mencionadas.

Así, 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* con la posibilidad de mejorar, consecuentemente, la calidad del producto final.”

La formulación de la hipótesis mencionada se llevó a cabo sobre las siguientes bases teóricas:

- Cuanto antes detectemos fallas en el sistema, menos costoso será solucionar los problemas, ante la menor complejidad existente en el código [20]. 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 *software 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 parámetros y *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 que unifique los enfoques MBT y TDD, elaboraremos un entorno de desarrollo integrado que implemente el método definido y, posteriormente, realizaremos algunas validaciones sobre la herramienta implementada. Creemos que el análisis final que pensamos aportar puede servir como ilustración que ayude, en trabajos posteriores, a probar la hipótesis propuesta.

4.4. Detalles de la Propuesta

En este trabajo definiremos un método para el desarrollo de *software* (en adelante, lo denominaremos **MoFQA** o ***Model-First Quality Assurance***) orientados a plataformas Web 2.0, siguiendo el proceso TDD. Dicho método incluirá técnicas y herramientas MBT pues se desea facilitar la tarea de *testing* mediante el uso de modelos que permitan la generación automática de código de *tests*. A continuación, el método será implementado en una herramienta que desarrollaremos como parte de este trabajo.

El *software* a verificar, sin embargo, no será generado a partir de los modelos proveídos por **MoFQA**; la generación automática de código se limitará a los *tests*. De esta forma se busca, por un lado, independizar los procesos de generación de *tests* y de desarrollo del *software* a verificar. Por otra parte, la redundancia en las especificaciones de la solución, otorgará una mayor posibilidad de encontrar errores en las distintas etapas del desarrollo.

Deseamos, además, que el usuario final tenga una alta participación en la definición de requerimientos y *tests* de aceptación por lo que nuestra herramienta deberá permitir la definición de modelos simples (sin requerir muchos conocimientos técnicos por parte del usuario). Además, el programador podrá definir *unit* e *integration tests* que, en conjunto con los *tests* de aceptación, guiarán el desarrollo mediante la aplicación de los pasos definidos por TDD.

La **Figura 6** describe el flujo de pasos a seguir en el proceso de desarrollo de *software* utilizando **MoFQA**. Es posible verificar que el método exige seguir el enfoque *test-first* propuesto por TDD, a partir de la utilización de modelos definidos tanto por el usuario final como por el desarrollador del *software*. La propuesta sigue 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.

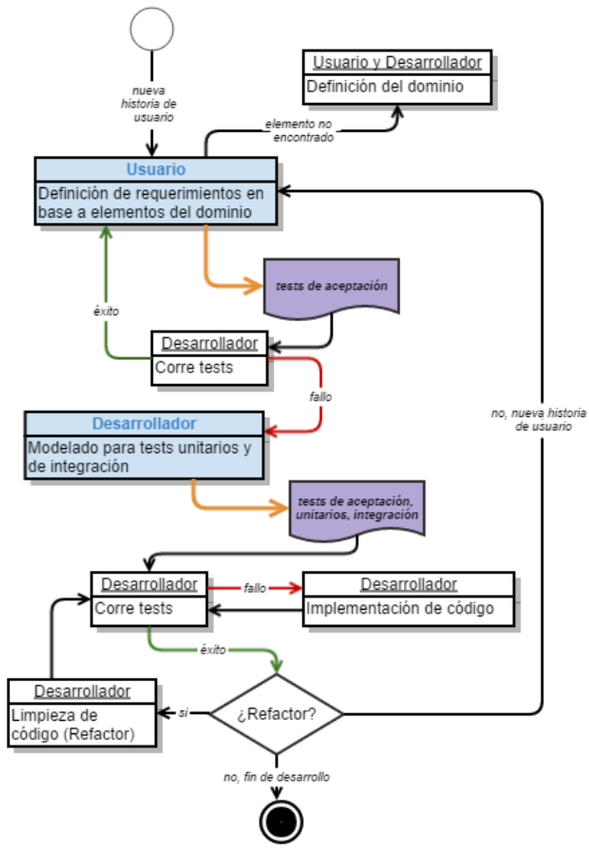

Figura 6: Flujo de desarrollo de *software* en MoFQA

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 *tests* de aceptación. Dicha definición es realizada utilizando los modelos proveídos por **MoFQA**. A partir de ahí, es posible derivar los *tests* de aceptación de forma automática, los cuales son ejecutados y validados por el desarrollador con ayuda de **MoFQA**. 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 *test* 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 [21]: "Un 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.".

El prototipo que construiremos en este trabajo será desarrollado como un *plugin* del entorno Eclipse[25] de manera a integrar varias herramientas existentes para llevar a cabo el proceso de desarrollo de *software*. Se implementará, como primer paso, un editor de modelos utilizando ***Sirius***[26]. Para ello, definiremos un metamodelo compatible con **EMF (*Eclipse Modeling Framework*)**[27], que representará los modelos que se podrán utilizar en **MoFQA**. El siguiente paso consiste en la definición de reglas de transformación de modelos a códigos de *test*.

Hoy en día, Java es un lenguaje muy utilizado para la generación de aplicaciones *web*, debido a la potencia del lenguaje y los componentes y *frameworks* otorgados por la plataforma Java EE para el desarrollo *web*[28]. Con esta base, nos enfocaremos en el *testing* de aplicaciones escritas en dicho lenguaje, y los *tests* estarán expresados en ***TestNG***[29]. *TestNG* permite la definición de *tests* en los niveles que necesitamos (*tests* unitarios y de integración, además de otras posibilidades) además de la ejecución de los mismos.

Finalizada la implementación de la herramienta, se iniciará un proceso de validación de resultados a partir de una ilustración que compare la utilización de la herramienta proveída, respecto a técnicas tradicionales, para el desarrollo de aplicaciones *web* de ejemplo, con sus respectivos *tests*.

5. Estado Actual del Proyecto

La **Tabla 2** lista las actividades definidas para el avance de nuestra propuesta y el estado de cada uno.

Actividad	Estado
Recopilación de trabajos relacionados / Análisis del Estado del Arte	Finalizado
Estudio de notaciones de modelado utilizadas para MBT	En Progreso
Definición del metamodelo y/o perfiles UML	En Progreso
Definición del Método MoFQA	En Progreso
Estudio de las herramientas a utilizar para la implementación	En Progreso
Definición de reglas de transformación	No Iniciado
Desarrollo de herramienta	No Iniciado
Validaciones	No Iniciado

Tabla 2: Estado de las actividades definidas para este trabajo

En la siguiente lista, describimos cada una de las actividades definidas y el estado en el que se encuentra:

- **Recopilación de trabajos relacionados / Análisis del Estado del Arte:** En este trabajo se presentan las conclusiones obtenidas luego de la recopilación de trabajos relacionados y análisis anteriores sobre el estado del arte. Esta etapa se encuentra concluida.
- **Estudio de notaciones de modelado utilizadas para MBT**: Se llevó a cabo un estudio de varias notaciones utilizadas para definir modelos en MBT. Entre ellas: diagramas de estado finito, notaciones lógicas, diagramas UML, lenguajes de pre y post condiciones. Se encontró que existe un paquete de perfiles UML específicos para *testing* (UTP [8]), que permite utilizar modelos UML para la definición del comportamiento de programas, otorgando elementos útiles para el *testing* (por ejemplo: temporizadores, contenedores de datos a utilizar, etc.). Por la utilidad de estos perfiles, se redireccionó el estudio hacia la forma de utilización de dichos perfiles.
- **Definición del metamodelo y/o perfiles UML:** En paralelo al estudio de UTP, hemos iniciado el proceso de definición del metamodelo para permitir el diseño de modelos de historias de usuario además de la reutilización de modelos UML (casos de uso, diagramas de clases, diagramas de estado UML, diagramas de interacción/actividades). Se desea utilizar como base los perfiles UTP pero esta decisión se encuentra aún bajo consideración.

[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 [21].
[25] http://www.eclipse.org/
[26] *Sirius* es una herramienta de modelado basada en EMF (*Eclipse Modeling Framework*), que permite la generación de editores de modelados a partir de la definición de un DSM (Domain Specific Model). Más información en: https://www.obeo.fr/en/products/eclipse-sirius.
[27] EMF (*Eclipse Modeling Framework*) consiste en un *framework* para la definición y generación de herramientas y aplicaciones en base a modelos. Más información en: https://eclipse.org/modeling/emf/.
[28] Más detalles en: https://netbeans.org/kb/trails/java-ee.html
[29] Framework para la definición y ejecución de *tests*, basado en JUnit y NUnit (*frameworks*, pertenecientes a la familia *xUnit*, para la definición de *tests* para programas escritos en Java y C#, respectivamente). Referencia TestNG: http://testng.org/doc/index.html.

- **Definición del Método MoFQA:** Se encuentra definido el proceso que deberá seguir toda herramienta que implemente el método (se presentó en la sección anterior). La definición total del método concluirá con la conclusión de la actividad presentada en el ítem anterior, la cual proveerá a **MoFQA** los modelos necesarios para su implementación.
- **Estudio de las herramientas a utilizar para la implementación:** Hasta el momento se han probado varias herramientas de tipo CASE[30] para la generación de editores de modelos, entre ellos: *Graphiti*[31], *Sirius*[32], *Obeo Designer*[33]. Queda pendiente estudiar los lenguajes utilizados en las herramientas para la definición de restricciones varias, validaciones, etc. Con una de estas herramientas se generará un *plugin* (para Eclipse) que proveerá un editor de modelos. Finalmente, investigaremos la forma de integrar dicho *plugin* a otros ya existentes (*Acceleo*[34] para la transformación de modelo a código, *TestNG* para la ejecución de *tests*).
- **Definición de reglas de transformación:** Las reglas de transformación son necesarias para convertir los modelos a código de *test*. Esta etapa aún no fue iniciada.
- **Desarrollo de herramienta:** Implica aplicar los conocimientos adquiridos en el estudio de herramientas para la generación del *plugin* de modelado y su correspondiente integración con los demás elementos.
- **Validaciones:** Se definirá uno o varios ejemplos de aplicaciones *web* a desarrollar a modo de ilustración. Para sacar resultados, se realizará comparaciones entre la utilización del método propuesto, contra el empleo de técnicas tradicionales, para llevar a cabo el proceso de desarrollo. La definición de los casos de ejemplo y de las métricas a utilizar para la comparación, son tareas pendientes. El trabajo finaliza con la ilustración y captación de resultados a partir de las comparaciones.

6. REFERENCIAS

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

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

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

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

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

[6] 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.

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

[8] 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.

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

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

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

[12] 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.*

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

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

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

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

[24] 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.

[30] Las herramientas CASE (Computer Aided Software Engineering, Ingeniería de Software Asistida por Computadora) son diversas aplicaciones informáticas o programas informáticos destinadas a aumentar la productividad en el desarrollo de software reduciendo el costo de las mismas en términos de tiempo y de dinero. Referencia: https://es.wikipedia.org/wiki/Herramienta_CASE

[31] https://eclipse.org/graphiti/

[32] https://www.eclipse.org/sirius/

[33] http://www.obeodesigner.com/

[34] https://eclipse.org/acceleo/