MDD+

Mejorando el Proceso de Desarrollo de Software: Propuesta basada en MDD

Nro. Referencia: 14-INV-056

Investigador Principal: Luca Cernuzzi, Universidad Católica “Nuestra Señora de la Asunción”

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

Informe sobre la Etapa de Diseño

Autor del Documento: Linda Riquelme, Universidad Católica “Nuestra Señora de la Asunción”

Noviembre – 2016

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.

Tabla de contenido

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]. 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*[1], 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*[2] 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*)[3] 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[4]. ***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.

[1] 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 [10].
[2] 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*.
[3] 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.
[4] 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

TDD es una práctica definida en el *Agile Manifesto*[5] 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 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[6] 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[7]).

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 [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[8] para la representación de *tests* aplicables a sistemas de plataforma Web.

[5] 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/.

[6] 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

[7] **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). [11]

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

- 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 utilización 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 [6] más formales).

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.

Este informe presenta la etapa de diseño del trabajo. En la misma se lleva a cabo la definición de perfiles UML que permitirán modelar *tests* según los aspectos considerados en nuestra propuesta.

2. Diseño de la Propuesta

El desarrollo de nuestra propuesta está dividido en tres etapas principales:

a. Definición de un proceso de desarrollo de *software*.
b. Desarrollo de una herramienta de modelado de *tests.*
c. Implementación de una herramienta de generación de *tests* a partir del modelado previo.

En las siguientes subsecciones del documento se presentan los avances de las etapas mencionadas, correspondientes al diseño de la propuesta.

2.1. MoFQA: proceso de desarrollo de *software* basado en TDD

En este trabajo definimos un método para el desarrollo de *software* (lo denominamos **MoFQA** o **Model-First Quality Assurance**) siguiendo el proceso TDD. Dicho método incluye 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*.

La *Figura 1* 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.

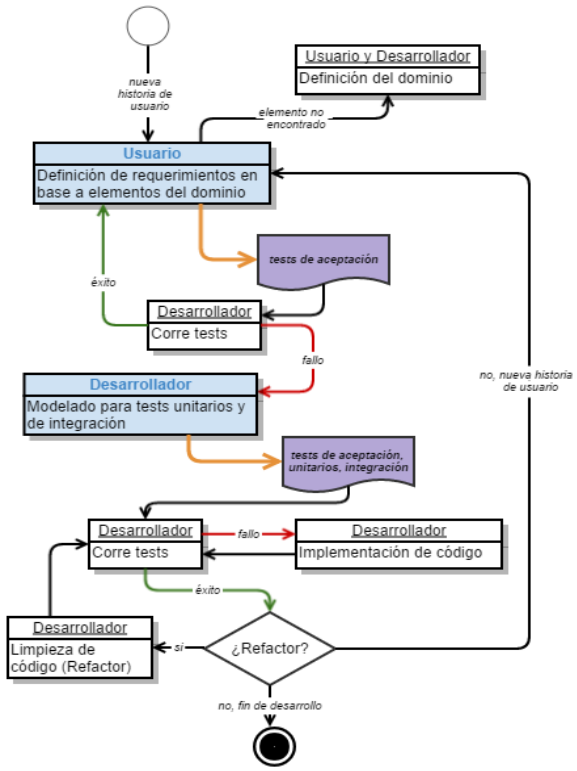

Figura 1: Proceso de desarrollo de software definido en la propuesta

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*)[9] buscando mejorar la comunicación entre el usuario y desarrollador en la transmisión de conocimientos del dominio del *software* a implementar. Como se menciona en [7]: "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

[9] 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 [7].

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."

2.2. Modelos definidos en la propuesta

Como etapa previa a la implementación de herramientas de modelado y generación de código se llevó a cabo la definición de un perfil UML que permitirá el modelado de *tests* por parte del usuario final y el desarrollador (ambos actores considerados en el proceso de desarrollo del *software* definido por **MoFQA**). 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 2* muestra el perfil UML[10] (***Acceptance Criteria***[11]) que permitirá modelar los *tests* para este trabajo. El perfil 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 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 cumplirse antes y después de su ejecución. Estas condiciones son definidas utilizando los elementos de este grupo.

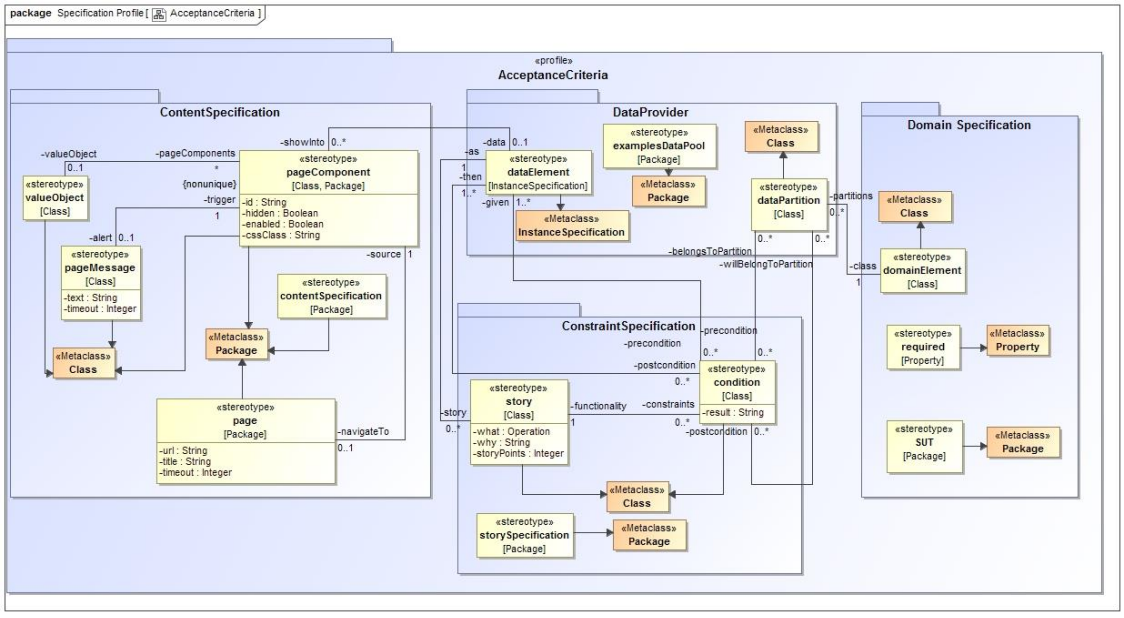

Figura 2: Perfil UML definido en la propuesta

[10] 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.
[11] Los modelos que pueden ser definidos utilizando este perfil están orientados a la descripción y generación de *tests* de aceptación en su gran mayoría.

La *Figura 3* muestra los elementos definidos en el grupo ***Data Provider***. Éstos tienen la finalidad de representar los datos de prueba 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: mediante la utilización de elementos <<dataElement>>.
- Como clasificaciones o particiones de datos: cada partición es definida por un elemento <<dataPartition>> que especifica el valor que debe tener uno o más atributos de una clase del dominio[12].

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*.

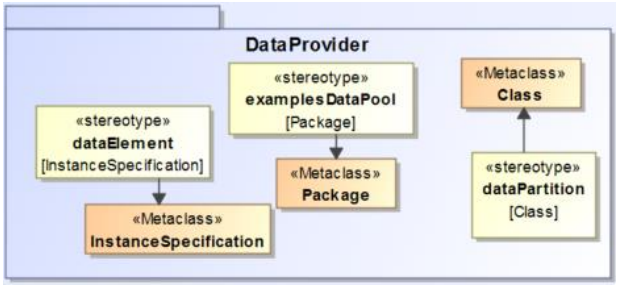

Figura 3: Elementos definidos en el grupo Data Provider

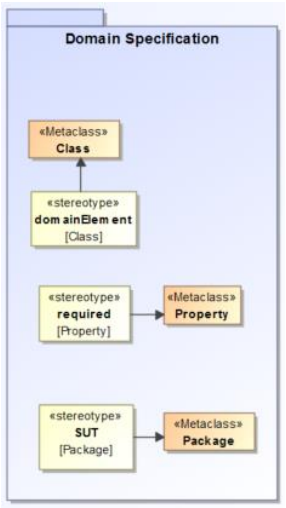

Figura 4: Elementos del grupo Domain Specification.

Para modelar los elementos del dominio se definió el grupo ***Domain Specification***, en la

Figura 5 puede verse sus elementos. Los elementos del dominio pueden ser definidos dentro de paquetes <<SUT[13]>>, el cual puede contener clases y enumeraciones.

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 ***¡Error! No se encuentra el origen de la referencia.***.

[12] El concepto de *dataPartition* fue inspirado en el elemento <<dataPartition>> definido en el perfil *TestData* de UTP (*UML Testing Profile*) [9].
[13] SUT: siglas del inglés *System Under Test*.

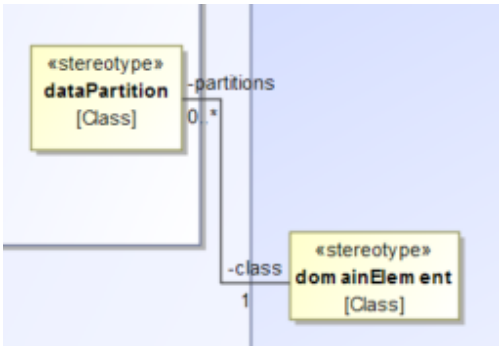

Figura 5: Relación entre <<dataPartition>> y <<domainElement>>

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. La *Figura 6* muestra los elementos definidos en el grupo ***Constraint Specification***.

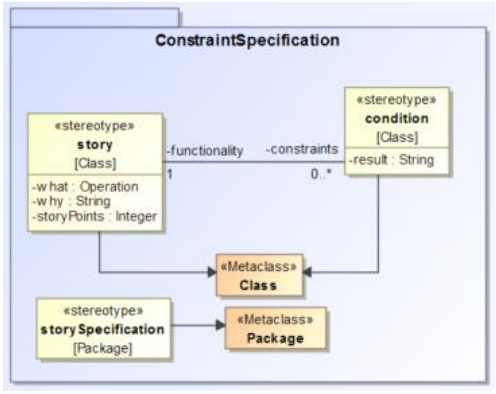

Figura 6: Elementos del grupo Constraint Specification

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

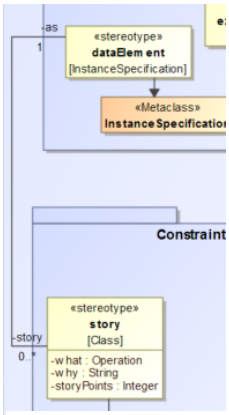

Figura 7: Un <<story>> se define a partir de un <<dataElement>> del dominio.

Cada funcionalidad se define como clase de tipo <<story>> con valores etiquetados que definirán sus características. Las etiquetas *as* (surge a partir de la relación entre <<story>> y <<dataElement>> graficada en la *Figura 7*), *what* (hace referencia a una operación definida en el <<dataElement>> especificado para la etiqueta *as*), *why* (cadena de caracteres descriptiva) y *storyPoints* (valor numérico) permiten especificar las funcionalidades en forma de historia de usuario con la siguiente estructura[14] *"As a as, I want what, 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>>.

[14] Estructura de historia de usuario basada en http://www.yodiz.com/blog/writing-user-stories-examples-and-templates-in-agile-methodologies/

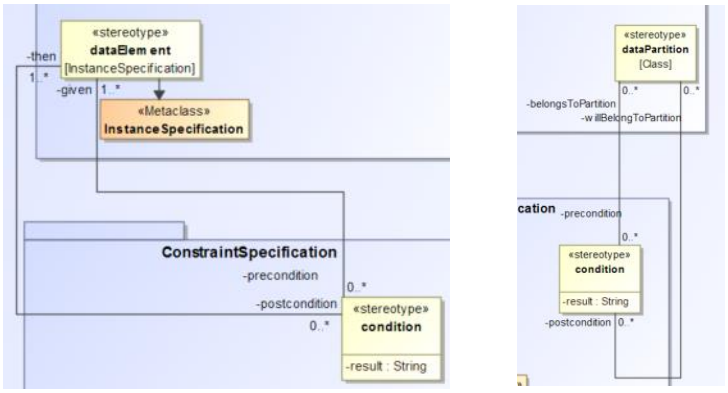

*Figura 8: Relaciones del elemento <<condition>> con los elementos del grupo **Data Provider**.*

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 8*.

El último grupo de elementos definidos en el perfil es ***Content Specification***. El mismo 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 9*.

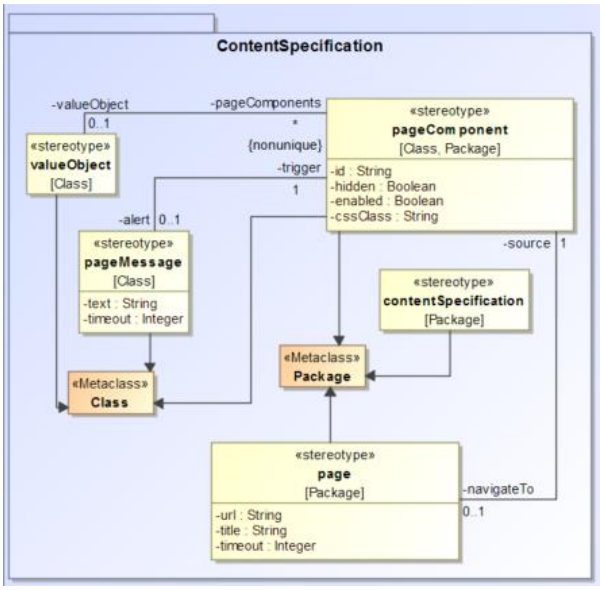

*Figura 9: Elementos del grupo **Content Specification***

El contenido para las diferentes páginas a probar debe ser especificado en paquetes de tipo <<contentSpecification>>.

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: *id* (código identificador del 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).

Las acciones sobre un elemento <<pageComponent>> también pueden ser descritas mediante relaciones con otros elementos:

- Al hacer *click* sobre un <<pageComponent>> puede esperarse que se cargue una nueva página, esta transición se define con la etiqueta *navigateTo* del <<pageComponent>>.

- 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>>.

Una clase <<pageMessage>> representa mensajes de alerta que pueden aparecer en la página como consecuencia de un *click* realizado sobre un <<pageComponent>>. Tiene las propiedades: *text* (texto que aparecerá en el mensaje) y *timeout* (máximo tiempo de espera para su aparición, en segundos).

Figura 10: Relación entre un <<pageComponent>> y un <<dataElement>>

Es posible además especificar elementos de datos para cada <<pageComponent>>, relacionándolo con un <<dataElement>> previamente definido (la *Figura 10* muestra la asociación entre <<pageComponent>> y <<dataElement>> definida en el perfil). Esta relación es realizada asignando un valor a la etiqueta *data* del <<pageComponent>>. Mientras un <<dataElement>> puede 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>>.

3. Ejemplo Ilustrativo

Con el objetivo de ilustrar el perfil presentado en la sección anterior, en esta sección se presenta un pequeño modelo de ejemplo. En este ejemplo, modelaremos un sistema de Biblioteca Virtual mediante el cual usuarios registrados pueden visualizar ítems (libros, revistas, CDs y DVDs) disponibles (o no) para su préstamo y, en caso de estar disponibles, solicitar su préstamo.

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

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

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 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).

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.

La *Figura 11* ilustra un diagrama del dominio para el sistema de Biblioteca Virtual descrito.

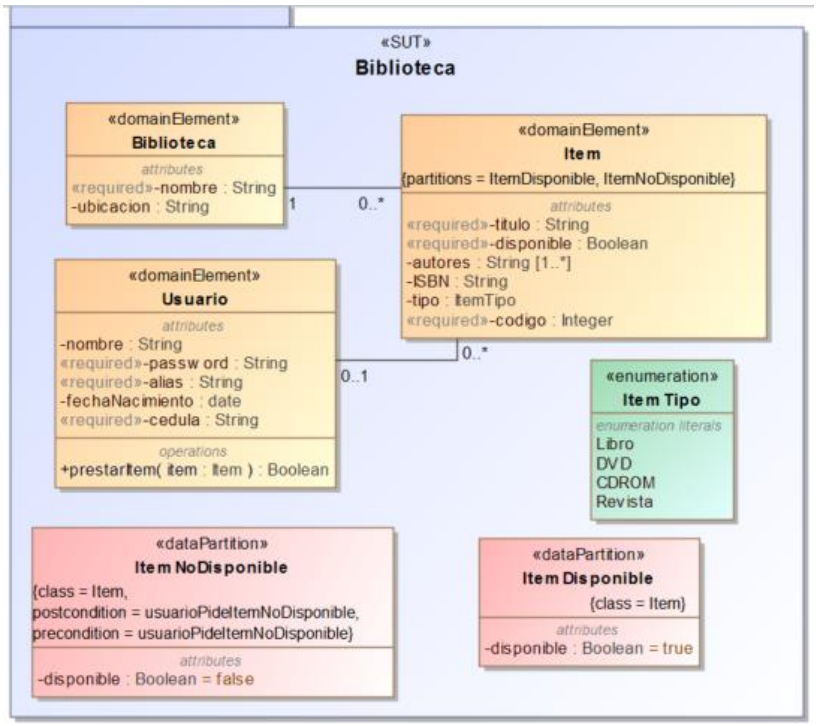

Figura 11: Diagrama de dominio para el sistema Biblioteca Virtual

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 12*), se presentan algunas instancias o elementos <<dataElement>> de las entidades Usuario e Item.

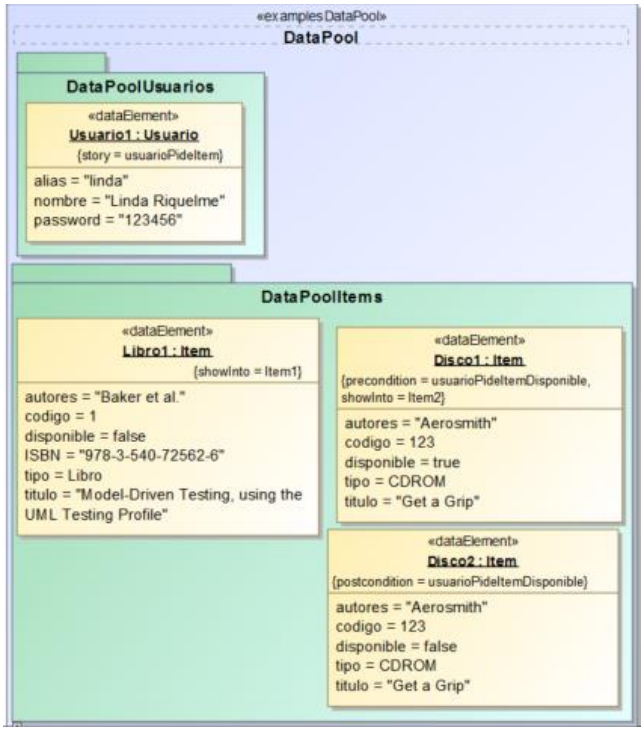

Figura 12: Instancias de elementos del dominio definidas como <<dataElement>>.

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 Usuario1, verificaremos las dos condiciones:

1. Que no pueda prestarse un Item no disponible.
2. Un Item disponible, en cambio, sí puede ser prestado.

La *Figura 13* representa, por un lado, la historia de usuario mediante una clase <<story>> (se asignó el valor 10, como puntos de historia). Usuario1 es un elemento <<dataElement>> definido previamente, que implementa la operación prestarItem. Los elementos <<condition>> representan 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), la ejecución de la operación prestarItem tendrá 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, el Item a prestar corresponde al <<dataElement>> Disco1. 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 valores de los atributos de la instancia Disco1, éstos deben coincidir con los valores definidos en la instancia Disco2 (disponible=false).

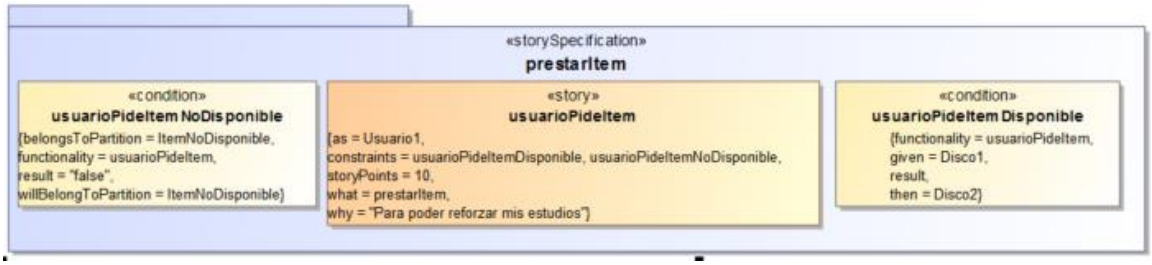

Figura 13: Modelado de historia de usuario "As Usuario1, I want prestarItem, so that Para poder reforzar mis estudios" con dos pre y post condiciones.

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, la *Figura 14* muestra dos páginas de la Biblioteca Virtual donde es posible solicitar el préstamo de ítems. La primera página (webItemsDisponibles) tiene asociada una URL para su acceso, un título a ser verificado y un tiempo máximo de tolerancia para su carga. 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) con el class CSS itemListaCss asignado.

Uno de los ítems que se desea ver en la lista está representado por el <<pageComponent>> Item1. Éste debe mostrar los autores y el título del <<dataElement>> Libro1. Como el ítem no está disponible, el elemento debe estar deshabilitado y un *click* sobre él debe generar la aparición de un mensaje de alerta: "Libro No Disponible".

El otro elemento a buscar en la lista tiene los atributos autores y título definidos para el <<dataElement>> Disco1. A diferencia del anterior, éste se encuentra disponible, por lo cual el elemento HTML 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!".

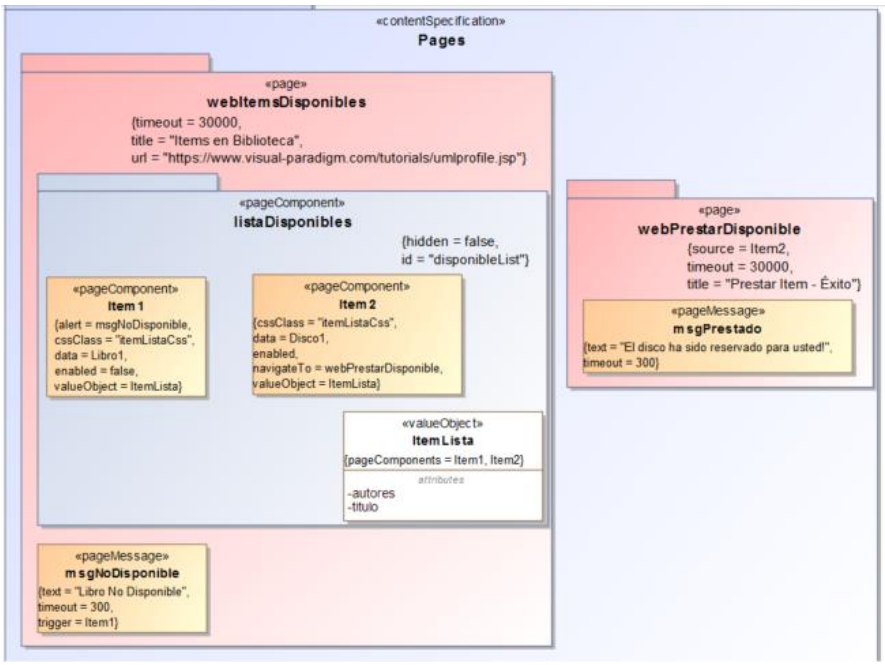

Figura 14: Definición de los componentes de las páginas que se crearán para la nueva historia de usuario.

4. 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 y 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] A. Avram, Domain-driven design Quickly, Lulu. com, 2007.

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

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

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

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