Casa abierta al tiempo **División de Ciencias Básicas e Ingeniería.**

Modelado de tácticas de atributos de calidad para la generación de esqueletos de arquitecturas ejecutables.

Que presenta:
Lic. Pedro Antonio Marcial Palafox.

Para obtener el grado de
Maestro en Ciencias
(Ciencias y Tecnologías de la Información).

Asesor

Dr. Humberto Cervantes Maceda.

Sinodales

Presidente: Dra. Perla I. Velasco Elizondo.
Secretario: Dr. Humberto Cervantes Maceda.
Vocal: M. C. Alfonso Martínez Martínez.

11 de Noviembre de 2009

Contenido

Resumen.

En este documento se presenta la propuesta del trabajo realizado en el proyecto de investigación *"Modelado de tácticas de atributos de calidad para la generación de esqueletos de arquitecturas ejecutables"*. La propuesta planteada en este trabajo consiste en generar una herramienta y definir un proceso que permitan modelar un subconjunto de tácticas de atributos de calidad en un modelo arquitectural con el fin de generar esqueletos de arquitecturas ejecutables, siguiendo un enfoque de desarrollo dirigido por modelos (MDA). Dentro del modelo arquitectural, se identifican componentes y puntos de configuración sobre los cuales se puede especificar las tácticas de atributos de calidad del sub conjunto. Se definen también reglas de transformación para posteriormente, mediante el uso de una herramienta, generar código que represente el esqueleto de una arquitectura ejecutable. El trabajo realizado en este proyecto se enfoca en el modelado de las reglas de transformación y el código generado para el subconjunto seleccionado de tácticas de atributos de calidad. Se propone que se siga el mismo proceso establecido en este proyecto para definir el modelado y reglas de transformación para el resto de tácticas de atributos de calidad.

Palabras clave: Arquitectura ejecutable, atributos de calidad, patrones arquitecturales, *Spring*, modelo específico de plataforma.

1 Introducción.

1.1 Contexto.

El desarrollo de un producto de software tiene generalmente como objetivo dar una solución a un problema dentro de una organización. Durante el proceso de desarrollo de aplicaciones de software, se realiza una actividad de definición de requerimientos. En esta actividad el cliente específica sus necesidades de negocio; estas necesidades son requerimientos que especifican tareas que podrá realizar el usuario que interactuará con el sistema, tomando en cuenta restricciones del ambiente en donde se pondrá en ejecución la aplicación. Estas especificaciones son conocidas como requerimientos de software [1]. Estos requerimientos son modelados para posteriormente con cada actividad y tarea del proceso de desarrollo, obtener un modelo más detallado y concreto que se acerque a una solución del problema, para finalmente plasmar este modelo en un producto codificado por el equipo de desarrollo (la aplicación) [6].

Los requerimientos de software son clasificados en requerimientos funcionales y requerimientos no funcionales. Los requerimientos funcionales especifican actividades y comportamientos que la aplicación debe proporcionar al usuario final. Por lo tanto los requerimientos funcionales son considerados para realizar el diseño detallado de la aplicación, es decir, la manera en que se realizará cada actividad y funcionalidad dentro del sistema. Por otra parte los requerimientos no funcionales son especificaciones, restricciones y atributos que la aplicación debe tener, y que por lo tanto deben ser considerados para su desarrollo. Dentro de los requerimientos no funcionales existe un grupo específico de estos, los atributos de calidad, que son esenciales en el desarrollo de este proyecto. Los atributos de calidad son características que permiten verificar y medir, entre otras cosas, el grado de satisfacción de los usuarios y/o diseñadores con respecto al sistema. Con la especificación de los requerimientos no funcionales (atributos de calidad) se realiza el diseño de una arquitectura específica para la aplicación que se desarrollará [2] [6].

De acuerdo al SEI (Software Engineering Institute) [32] la arquitectura de software de un programa o sistema de cómputo es la estructura o estructuras del sistema, las cuales comprenden elementos de software, propiedades externas de esos elementos y las relaciones entre ellos [2]. La arquitectura puede ser vista de manera simplificada como los cimientos sobre los cuales se realizará la construcción de la aplicación. Por lo tanto, en el proceso de desarrollo se realizan actividades concretas durante fases tempranas de desarrollo para poder establecer y validar la arquitectura, pues de esta dependen las actividades siguientes de desarrollo.

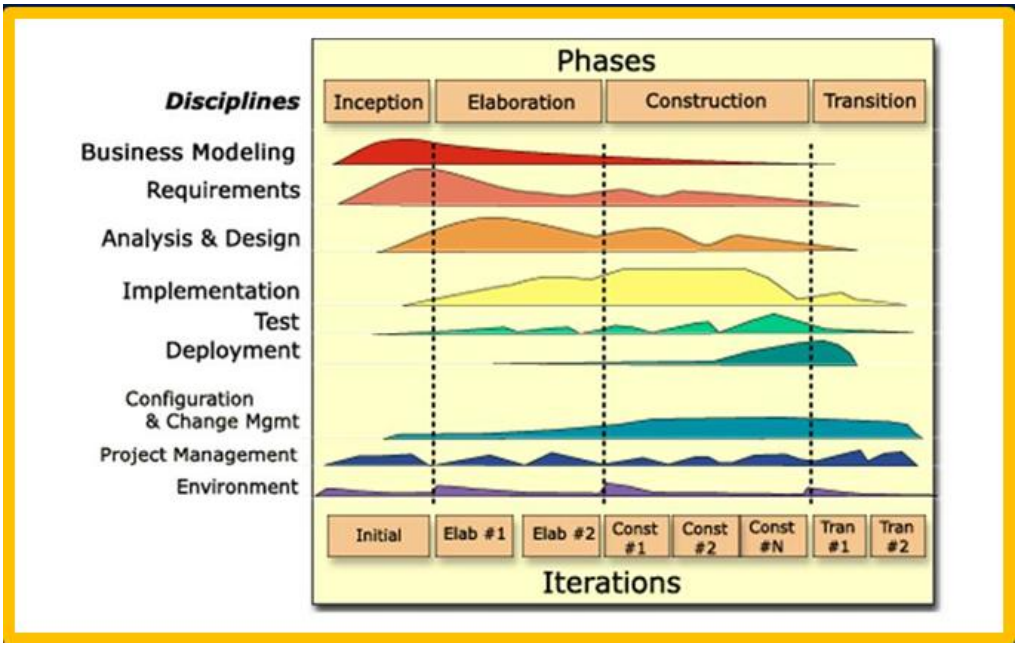

Figura 1. Proceso de desarrollo Rational Unified Process [1].

RUP (Rational Unified Process) es un marco de procesos de desarrollo de software muy utilizado en la industria. Este marco de procesos define cuatro fases para el desarrollo de software las cuales son: Incepción, Elaboración, Construcción y Transición [Figura 1]. Justamente, durante toda la fase de Elaboración, entre otras cosas, se realiza el diseño de una arquitectura obteniendo como producto final de esta fase una arquitectura ejecutable que será utilizada durante la fase de construcción para continuar desarrollando la aplicación [1].

La arquitectura ejecutable es una implementación parcial del sistema, en la cual principalmente se implementan características no funcionales junto con un sub conjunto de características funcionales, que representan las funcionalidades más importantes priorizadas por los desarrolladores y los clientes, con el objetivo de validar que la arquitectura sea la adecuada para la aplicación que se está desarrollando. Uno de los objetivos de construir una arquitectura ejecutable es mitigar posibles riesgos tecnológicos [1].

1.2 Problemática.

Dentro del contexto descrito anteriormente, se debe señalar que actividades de diseño y construcción de la arquitectura son actividades que requieren un tiempo considerable y no existe una herramienta que facilite el desarrollo de la arquitectura ejecutable. Con el fin de facilitar el desarrollo de la arquitectura, una herramienta para este propósito debería permitir a los arquitectos y diseñadores trabajar a nivel de modelos y permitirles el generar de manera sencilla un esqueleto de la arquitectura ejecutable (de aquí en adelante se considerará esqueleto de arquitectura ejecutable como una implementación parcial de requerimientos no funcionales, específicamente atributos de calidad).

1.3 Propuesta.

La propuesta del proyecto de investigación "*Modelado de tácticas de atributos de calidad para la generación de arquitecturas ejecutables"* consiste en producir una herramienta, siguiendo un proceso basado en el enfoque MDA (diseño dirigido por modelos), que permita generar esqueletos de arquitecturas ejecutables que satisfacen diversos atributos de calidad. El esqueleto de arquitectura ejecutable está compuesto por un conjunto de componentes de software con interfaces y relacionados entre ellos que forman la parte estructural enfocada a satisfacer los atributos de calidad con respecto a una plataforma específica, esta plataforma puede estar compuesta por uno o más *Frameworks* (herramientas utilizadas para el desarrollo de software). Sin embargo, los componentes del esqueleto de la arquitectura ejecutable no incluyen la lógica que satisface los requerimientos funcionales ya que esto no es el enfoque del trabajo.

Debido que se sigue un enfoque MDA (Diseño Dirigido por Modelos) [22], la herramienta debe permitir representar un modelo de una arquitectura de una aplicación para una plataforma específica. Dicho modelo debe, además, permitir configurar propiedades asociadas a tácticas (decisiones de diseño) que permitan soportar diversos atributos de calidad del sistema asociadas con la plataforma específica que se desea utilizar. A partir de este modelo específico de plataforma, la herramienta debe permitir realizar una transformación hacia código generando, así como el esqueleto de arquitectura ejecutable.

Por otra parte, el proceso definido en este proyecto debe ser extensible hacia el soporte de otras plataformas y otros tipos de atributos de calidad, pues para este proyecto sólo se utiliza un sub conjunto de los definidos por el SEI. Es decir, se espera que el proceso utilizado en este proyecto establezca las bases necesarias para poder implementar un proceso similar para generar arquitecturas ejecutables sobre otras plataformas, lo cual implicaría una diferente implementación que se adecue a los atributos de calidad soportados por cada plataforma.

1.3.1 Objetivos generales y objetivos particulares del proyecto.

1.3.1.1 Objetivos generales.

Los objetivos generales de este proyecto de investigación son:

- Definir un proceso que utilice como base el enfoque MDA para el modelado y generación de un esqueleto de arquitectura ejecutable de una plataforma específica con soporte de un cierto número de atributos de calidad.
- Construir una herramienta que permita generar esqueletos de arquitecturas ejecutables para una plataforma en específico.
- Reducir el tiempo de diseño y construcción, así como facilitar el trabajo en estas fases de desarrollo de sistemas, utilizando dicha herramienta.

1.3.1.2 Objetivos particulares.

Los objetivos particulares de este proyecto de investigación son:

- Realizar un estado del arte sobre arquitecturas y requerimientos no funcionales de software, así como de la manera en que estos son utilizados en el proceso de desarrollo de un sistema de software.
- Identificar cuáles y como es que la plataforma seleccionada (*Framework* de desarrollo) soporta el manejo de atributos de calidad.
- Investigar e implementar la manera de construir un modelo arquitectural para un sistema utilizando la plataforma de desarrollo seleccionada, *Framework* de desarrollo (de ahora en adelante se referirá al conjunto de herramientas de desarrollo seleccionadas como *Framework* de desarrollo).
- Estudiar las posibles herramientas que nos permitan seguir el enfoque MDA para la generación de un esqueleto de arquitectura ejecutable a partir de modelos definidos de la plataforma seleccionada.

1.3.1.3 Limitaciones y Alcances.

La principal limitante del proyecto es el escaso tiempo para su elaboración. Pero existen factores específicos que influyen directamente en la planeación y alcance del proyecto, estos factores específicos son las siguientes:

- Requerimientos Funcionales. El incluir los aspectos funcionales dentro del modelo arquitectural es complejo, debido principalmente a que los requerimientos funcionales y los requerimientos no funcionales son ortogonales (independientes) [2], así como también que los requerimientos funcionales son específicos en cada aplicación. Por tal motivo no se incluye las características funcionales en el desarrollo de este proyecto (véase sección 4.3).
- Requerimientos no Funcionales. Debido a que los requerimientos no funcionales y los requerimientos funcionales son ortogonales, así como también que el manejo de atributos de calidad depende del *Framework* utilizado para el desarrollo, en este proyecto se considera únicamente la implementación de aspectos no funcionales.
- Proceso MDA. Debido a que es un proceso muy extenso que permite la utilización de modelos para la representación de la aplicación, modelos que permiten especificar características funcionales y no funcionales, para este proyecto solo se consideran ciertas fases de desarrollo de esta metodología enfocadas principalmente en el manejo de requerimientos no funcionales, debido a limitantes de tiempo principalmente.
- Complejidad en manejo del *Framework* de desarrollo. Debido a que existe una complejidad considerable en el manejo del *Framework.* Por tal razón solo se escogieron un número limitado de herramientas de desarrollo las cuales, a su vez, limita el número de tácticas.

Como resultado del análisis de las limitantes mencionadas anteriormente el alcance del proyecto es el siguiente:

- La especificación del modelo así como el código generado únicamente contemplarán el soporte de un conjunto de atributos de calidad soportado por la plataforma seleccionada.
- La selección del *Framework* de desarrollo deberá considerar emplear un *Framework* que permita implementar el soporte de tácticas de calidad de manera independiente de los aspectos funcionales.
- Este proyecto se enfocará únicamente en el modelo, de la metodología MDA, que permita definir el soporte de atributos de calidad en una plataforma específica.
- Únicamente se implementará un subconjunto de las tácticas soportadas por el *Framework* seleccionados, debido a la complejidad para poder hacer uso de este en su totalidad, así como el poco tiempo que se tiene para implementar un conjunto mayor de tácticas.

1.4 Metodología y Calendarización.

1.4.1 Metodología.

En esta sección se describe la metodología empleada en el desarrollo de este proyecto de investigación. Es decir, la manera en que se hizo la investigación; específicamente, investigación previa (antecedentes), hipótesis, variables, tipo de estudio, procesos utilizados y evaluación. El detalle de la metodología empleada para el desarrollo de este proyecto de investigación esta descrita continuación:

- **Investigación previa.** Se realiza la investigación sobre arquitecturas y atributos de calidad, así como también de cómo es que varios Framework de desarrollo implementan un conjunto de tácticas para el soporte de atributos de calidad (el término de tácticas de atributos de calidad será definido más adelante en este trabajo).
- **Hipótesis.** La hipótesis de la que se parte en este proyecto es que el proceso que se define en este trabajo de tesis, permite construir un modelo específico de plataforma el cual contiene definiciones de soporte de un conjunto de atributos de calidad. El proceso definido permite la construcción de una herramienta que permita realizar la generación de código de manera automática a partir del modelo específico de plataforma, por lo que se estaría reduciendo el tiempo de desarrollo dedicado a la construcción de esqueletos de arquitecturas ejecutables (también conocidos como prototipos). En síntesis, se toma como hipótesis que el proceso definido permite la construcción de una herramienta que minimizaría el tiempo de desarrollo durante la transición entre las fases de elaboración y construcción de RUP.
- **Variables.** Las variables dentro de la metodología, pero más aun dentro del proceso del proyecto son el conjunto de tácticas de atributos de calidad que son soportadas, así como la manera en que son implementadas en cada

plataforma (*Framework*). Sin embargo, por cuestiones de tiempo sólo se utilizará un conjunto de tácticas sobre una única plataforma de desarrollo.

- **Tipo de estudio.** El tipo de estudio para este trabajo de tesis es experimental, ya que mediante la evaluación de la herramienta generada se pretende corroborar que el tiempo de desarrollo disminuye en la transición entre las fases de desarrollo de elaboración y construcción.

- **Procesos utilizados.** Se tomó como base parte del proceso MDA (más adelante se hará énfasis en que partes de este proceso se enfocó este trabajo) para la implementación y construcción de la herramienta que permitirá validar el trabajo realizado en esta tesis.

- **Evaluación.** El método de evaluación del trabajo es realizar comparativas de tiempo en el desarrollo de software empleando la herramienta generada en este proyecto.

1.4.2 Calendarización.

La calendarización para el proyecto de investigación consiste en las siguientes fases que tienen como propósito alcanzar cada uno de los objetivos del proyecto:

- **Investigación.** Durante esta fase se realizarán actividades de investigación sobre arquitecturas de software y el manejo de atributos de calidad en plataformas de desarrollo. Esta fase tiene contemplada una duración de 3 meses.

- **Definición.** Definición del proceso de modelado de atributos de calidad para la generación de un esqueleto de arquitectura ejecutable. Esta fase tiene contemplada una duración de 4 meses.

- **Construcción y evaluación.** Durante esta fase de desarrollo del proyecto están contempladas actividades relacionadas con la construcción de la herramienta, evaluación de la misma y la elaboración de la tesis de este proyecto. Durante esta fase se contempla también el tiempo necesario para estudiar el *Framework* que permitan construir la herramienta de generación de esqueletos, por lo tanto el tiempo empleado en esta fase es de 6 meses.

1.5 Estructura de la Tesis.

El presente documento está estructurado de manera que el lector pueda conocer y familiarizarse con los términos, tecnologías, tácticas, patrones y definiciones necesarios para poder comprender el proyecto.

En el segundo capítulo de este documento se describe la investigación (estado del arte) que fue necesaria realizar para poder comprender y llevar a cabo este proyecto. Dentro de este capítulo se hace mención a las definiciones de arquitectura, así como a los principales aspectos (requerimientos, patrones y tácticas) que deben ser considerados para

poder realizar el modelado o diseño y las distintas maneras en que es posible documentar una arquitectura. Dentro de la investigación realizada se describe cómo es que los requerimientos no funcionales son soportados dentro de plataformas especificas tales como los EJB's (*Enterprise JavaBeans*) y el *Framework Spring*, así como las características específicas de cada una de estas plataformas. En este capítulo se introducen los conceptos necesarios de MDA (Arquitectura Dirigida por Modelos), que es el enfoque utilizado en este proyecto. Dentro de este capítulo también se describen los conceptos necesarios de este enfoque que son utilizados para el desarrollo de este proyecto de investigación para especificar cómo se realiza la transformación de un modelo de arquitectura hacia código.

En el tercer capítulo se describe la teoría a detalle de la propuesta definida para este proyecto. En este capítulo se describen los elementos que son considerados necesarios dentro de un modelo que representa la arquitectura de una aplicación sobre una plataforma en específico, para poder realizar la transformación de las propiedades asociadas a las tácticas seleccionadas. Por último dentro de este capítulo se define en qué consisten las reglas de transformación para poder realizar la transformación hacia código para una plataforma específica.

En el capítulo cuatro se describe a detalle el proceso y las actividades que se siguen para desarrollar la herramienta. Dentro de las actividades realizadas se describe la definición de las etiquetas específicas que permitirán la configuración de tácticas para soportar atributos de calidad sobre una plataforma específica. Así como cuales con las reglas de transformación definidas y la adaptación de estas a una herramienta de generación de código.

En los últimos capítulos se presenta la discusión critica de esta tesis así como también las conclusiones obtenidas de la realización del este proyecto y los resultados obtenidos, así como también se citan las referencias consultadas.

2 Estado del arte y antecedentes.

2.1 Arquitectura.

La arquitectura de software es la descripción de las estructuras del sistema, algunos elementos que la conforman son módulos, procesos, componentes de implementación, capas, etc. La arquitectura es el primer artefacto que puede ser analizado para determinar qué tan bien logrados son sus atributos de calidad y esto también sirve como un elemento de diseño. Una arquitectura sirve como vehículo de comunicación entre los diferentes interesados en el desarrollo, es la representación de las decisiones de diseño tempranas y es una especificación reutilizable que puede ser transferida a nuevos sistemas [2].

La arquitectura de software es desarrollada considerando las principales especificaciones de diseño que se hacen sobre un sistema. Las especificaciones de diseño deben considerar las características que el sistema debe proporcionar. Algunos ejemplos de las especificaciones de diseño son: la estructuración de los distintos tipos de elementos y componentes de software que conformarán la aplicación, la organización de los componentes acorde con responsabilidades o funcionalidades comunes, comportamiento específico del sistema acorde con especificaciones de desempeño y facilidad de uso para el usuario final [2]. Por ser un elemento que se enfoca en cubrir las necesidades del usuario, la arquitectura de software es considerada el marco de referencia de desarrollo que es necesario para guiar la construcción del software. El diseño de una arquitectura adecuada afecta todo el ciclo de vida del proceso de desarrollo, pues al determinar las dimensiones de todos los componentes necesarios para construir la aplicación es posible realizar una mejor estimación del tiempo requerido para el desarrollo del sistema. Principalmente por estas razones se le considera una tarea de gran importancia en el desarrollo e ingeniería de software [3] [4] [5].

A las necesidades y especificaciones que el usuario desea que la aplicación presente se les conoce como requerimientos de software. En la siguiente sub sección se presenta una definición de los requerimientos de software, así como también se hace la distinción de cuales requerimientos influyen en el desarrollo de la arquitectura.

2.1.1 Requerimientos.

Los requerimientos de software son las especificaciones funcionales y técnicas de la aplicación proporcionadas por el cliente. Los requerimientos de software se pueden catalogar en dos tipos: requerimientos funcionales (RF) y requerimientos no funcionales (RNF) [Figura 2] [6].

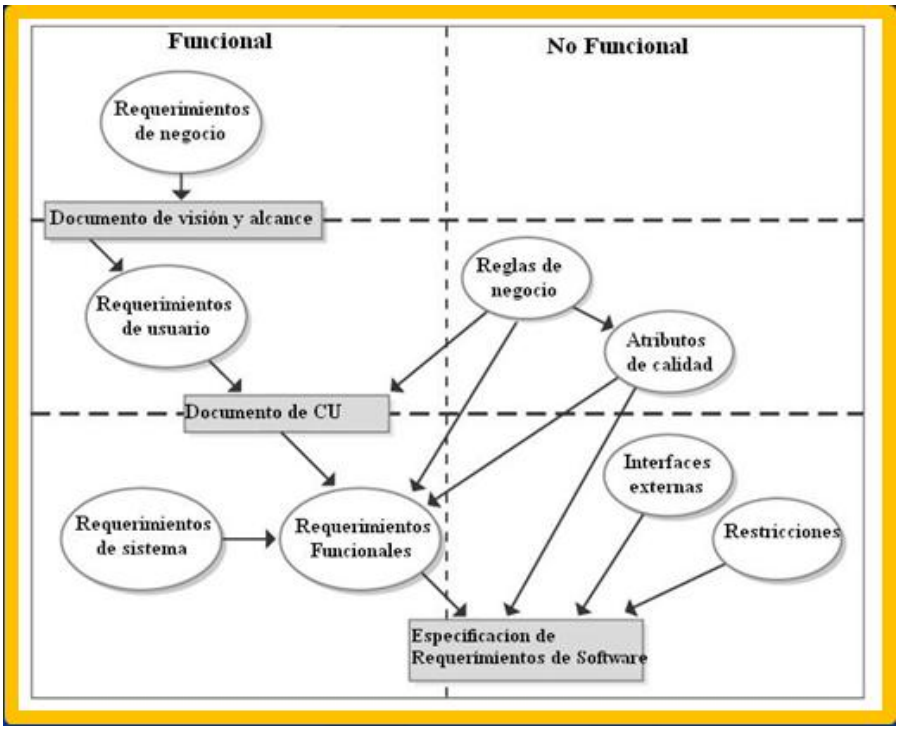

Figura 2. Tipos de Requerimientos de Software [6].

Los requerimientos de software son muy importantes pues son las especificaciones explícitas e implícitas de lo que el usuario espera del producto de software, es necesario tener una definición concisa de los distintos tipos de requerimientos que existen. En las siguientes sub secciones se describen los distintos tipos de requerimientos de software con más detalle.

2.1.1.1 Requerimientos funcionales.

Los requerimientos funcionales especifican las actividades de negocio del cliente y la funcionalidad que la aplicación permitirá realizar al usuario final. Estas actividades definen el comportamiento específico de cada aplicación [2]. Retomando la Figura 2 es posible observar que dentro de los requerimientos funcionales de software se encuentran los requerimientos de negocio, estos consisten en la manera en que la empresa lleva a cabo las actividades necesarias para que esta funcione dentro del rubro en el que se desenvuelve, es decir todas las actividades y procesos propios de la empresa que los empleados utilizan para realizar su trabajo. También dentro de los requerimientos funcionales se consideran necesidades específicas de los usuarios finales (personas que operan el sistema) y especificaciones de cómo debe operar el sistema, llamadas casos de uso (CU). El análisis de estos requerimientos permite desarrollar la implementación detallada del funcionamiento de la aplicación.

Las características funcionales no son las únicas que deben ser cubiertas durante el desarrollo de aplicaciones de software. Por otra parte, también deben ser consideradas características y especificaciones que no tienen que ver de

forma directa con la funcionalidad del sistema, estas especificaciones son llamadas requerimientos no funcionales y son descritos a continuación [2].

2.1.1.2 Requerimientos no funcionales.

Los requerimientos no funcionales son las especificaciones, características, objetivos de desempeño y atributos que el cliente proporciona, no siempre de manera explícita, y desea que tenga la aplicación que se construirá. Estos requerimientos deben ser considerados al momento de realizar el diseño e implementación pues imponen restricciones en estas actividades [6] [7]. Como se puede observar de la Figura 2, los requerimientos no funcionales son catalogados de manera más específica en:

- **Reglas de negocio**. Definen o limitan como se realizan algunos aspectos de negocio.
- **Atributos de calidad.** Son propiedades que especifican características para poder verificar y medir el grado de satisfacción de los usuarios y/o diseñadores con respectos al sistema de software.
- **Interfaces Externas.** Describen las conexiones del sistema con otras herramientas o productos.
- **Restricciones.** Describen o limitan aspectos de diseño e implementación.

De entre estas categorías de requerimientos no funcionales se encuentran los atributos de calidad y restricciones, los cuales influyen directamente en el diseño de la arquitectura de software y en la calidad del sistema [6], por lo que son descritos con más detalle en las siguientes sub secciones.

2.1.1.2.1 Atributos de calidad.

Los atributos de calidad han sido de interés para la comunidad del software desde la década de los 70s. Dentro de la comunidad del software se han publicado varias taxonomías y definiciones de los atributos de calidad (Modelo de calidad de McCall, ISO 9126, FURPS, SEI). Sin embargo el SEI (*Software Engineering Institute*) identifica tres problemas que se presentan en las discusiones de atributos de calidad vistos desde una perspectiva arquitectural, estos tres problemas son los siguientes [2]:

- Las definiciones de atributos de calidad no son operaciones. Es decir, algunas de las definiciones de atributos de calidad especifican cualidades o características que todo sistema tiene en poco o mayor grado sobre ciertos componentes dentro del sistema. Por ejemplo, modificabilidad, cada sistema es modificable con respecto a un conjunto de cambios y no así con respecto a otro.
- Existe un problema al definir cual cualidad pertenece a un aspecto de calidad particular (traslapo de cualidades en atributos de calidad). Por ejemplo, una falla de sistema es un aspecto de disponibilidad, de seguridad o usabilidad.

- Cada atributo de calidad ha desarrollado su propio vocabulario. Por ejemplo: el desempeño tiene “eventos” arribando a un sistema, la seguridad tiene “ataques” arribando a un sistema, la disponibilidad tiene “fallas” de un sistema. Y la usabilidad tiene “entradas de usuario”.

El SEI propone el uso de escenarios de atributos de calidad, este concepto se describirá a detalle en las siguientes secciones, como una medida de caracterizar los atributos de calidad y así dar una solución a los primeros dos problemas antes mencionados. Una solución al tercer problema es proveer los conceptos que son fundamentales para cada atributo de calidad (definir que vocabulario será empleado en el manejo de atributos de calidad) [2].

Para realizar esta tesis se ha seleccionado la categorización y manejo de atributos de calidad que propone el SEI, principalmente debido a la solución que plantean a los tres problemas antes mencionados. El SEI clasifica los atributos de calidad por disponibilidad, modificabilidad, desempeño, seguridad, facilidad de pruebas y usabilidad (estas cualidades serán descritas más adelante) [2]. La Figura 3 muestra la clasificación que el SEI hace de los atributos de calidad.

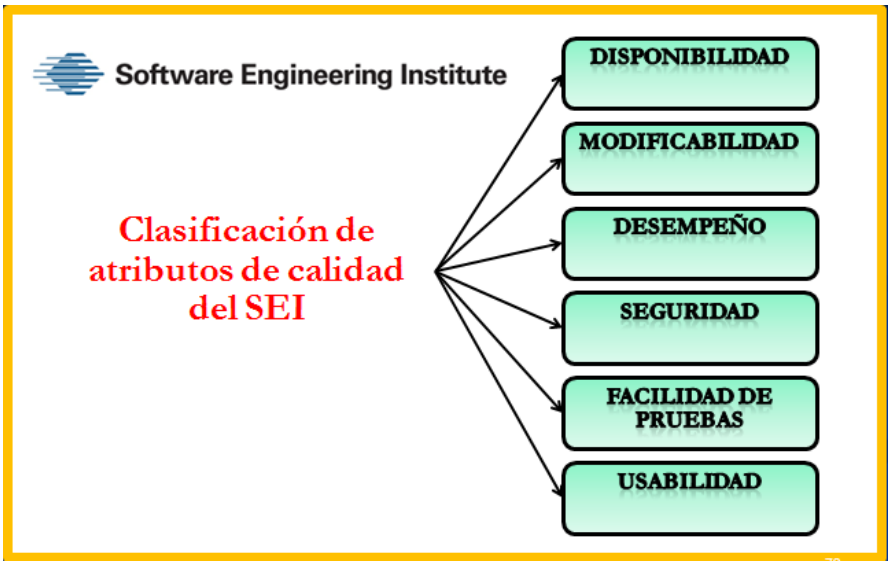

Figura 3. Clasificación de Atributos de Calidad del SEI [2].

Los atributos de calidad, como su nombre lo indica, permiten medir y controlar la calidad de un sistema. Los atributos de calidad pueden ser vistos como cualidades que el sistema presenta acorde con las especificaciones que el usuario solicita.

El SEI también propone una manera de especificar los atributos de calidad, para que estos puedan ser manejados por todos los involucrados del proyecto durante el desarrollo de este. Este tipo de especificación de los atributos de calidad son los escenarios y se describen en la siguiente sección. Durante el desarrollo de este proyecto de investigación se adoptará el uso de la definición y especificación de los atributos de calidad propuestos por el SEI, debido principalmente a que existen muy pocas alternativas que permitan definir y documentar los atributos de calidad.

2.1.1.2.1.1 Escenarios de atributos de calidad.

Un escenario de un atributo de calidad es una representación de un requerimiento específico de un atributo de calidad que el sistema debe presentar. Un escenario debe especificar las características y elementos necesarios para que el atributo de calidad pueda ser manejado en todas las actividades de desarrollo que lo requieran. Un escenario de acuerdo al SEI debe estar compuesto por seis partes, las cuales son [2]:

- **Fuente de estímulo.** Esta es alguna entidad (una persona, un sistema de cómputo, o cualquier otra entidad que interactúa con el sistema) que genere estímulos hacia el sistema.
- **Estímulos.** Los estímulos son condiciones que deben ser consideradas cuando afectan o estimulan algún comportamiento específico del sistema.
- **Ambiente.** Los estímulos ocurren bajo determinadas condiciones. Es decir el sistema presenta alguna configuración en específico o se encuentra en algún estado determinado por la carga de trabajo.
- **Artefacto.** Los artefactos son los componentes o conjunto de elementos del sistema que son afectados por los estímulos.
- **Respuesta.** La respuesta son las actividades que se derivan como resultado de algún estimulo hacia un componente.
- **Medida de la respuesta.** Cuando la respuesta ocurre, esta debe ser medida de alguna manera tal que el requerimiento pueda ser probado.

Se pueden diferenciar dos tipos de escenarios; estos son los escenarios generales y los escenarios específicos. Los escenarios generales son independientes del sistema y por lo tanto pueden ser reutilizados para construir los escenarios específicos para diferentes sistemas. Por otra parte los escenarios específicos describen las características detalladas de un atributo de calidad dentro de un sistema particular. Uno de los usos de los escenarios generales es permitir a los interesados comunicarse, sabiendo que cada atributo tiene su propio vocabulario para describir sus conceptos básicos y que diferentes términos puedan representar el mismo evento.

Una colección de escenarios concretos puede ser usada como requerimientos de atributos de calidad para el desarrollo de un sistema. Cada escenario es suficientemente concreto para ser significativo para la arquitectura, y los detalles de la respuesta son suficientemente significativos de manera que es posible probar si el sistema ha logrado la respuesta esperada.

En las siguientes secciones se presentan las definiciones adoptadas de los atributos de calidad así como un ejemplo de un escenario para cada uno de ellos.

2.1.1.2.1.1.1 Disponibilidad.

La disponibilidad es la probabilidad de que un sistema sea capaz de operar cuando le sea requerido, bajo circunstancias especificadas. La disponibilidad de un sistema tiene que ver con las fallas de este y las consecuencias asociadas. Una falla de sistema ocurre cuando el sistema no proporciona un servicio en el tiempo especificado, una falla como tal es observada por los usuarios del sistema [2].

Para poder controlar este atributo de calidad es necesario dar un manejo apropiado a las fallas de disponibilidad que puedan presentarse en el sistema. Para esto los diseñadores deben considerar lo siguiente: ¿Cómo es detectada la falla del sistema?, ¿Cuán frecuente puede ocurrir una falla de sistema?, ¿Qué pasa cuando la falla ocurre?, ¿A partir de qué momento un sistema está permitido para estar fuera de operación?, ¿Cuándo pueden ocurrir las fallas sin ningún daño?, ¿Cómo pueden ser prevenidas las fallas?, y ¿Qué clase de notificaciones son requeridas cuando ocurre la falla?

Considerando lo anterior es necesario diferenciar las fallas de los defectos. Un defecto puede llegar a ser una falla si no se corrige u oculta. Esto es, una falla es observable para el usuario del sistema y el defecto no. Cuando un defecto llega a ser observado, este llega a ser una falla. Por ejemplo, un defecto puede ser seleccionar un mal algoritmo para realizar un cómputo, resolviendo un cálculo equivocado que resulta en una falla del sistema. Una vez que un sistema falla, un concepto importante relacionado con la disponibilidad llega a ser el tiempo que toma reparar esta falla desde que esta es observada por los usuarios [2].

Un ejemplo de un escenario específico del atributo de calidad de disponibilidad se muestra en la Tabla 1:

Elemento del escenario	Valores
Fuente de estimulo	Sistema externo que interactuará con el sistema en desarrollo.
Estimulo	Mensaje imprevisto relacionado con la interoperabilidad, el sistema externo realiza una operación de consulta solicitando información.
Artefacto	Proceso encargado de la interoperabilidad con otro sistema.
Ambiente	El sistema se encuentra en operación normal.
Respuesta	Informar al usuario de problemas con la interoperabilidad y continuar con la operación normal del sistema.
Medida de la respuesta	El sistema presenta un tiempo de inactividad menor a 2 segundos.

Tabla 1. Escenario de disponibilidad.

En la Tabla 1 se expone un escenario de disponibilidad; la posible falla de disponibilidad se debe a la interoperabilidad que puede tener un sistema con otros. Considerando que la falla de interoperabilidad con otro sistema específico no puede ser controlada por el equipo de desarrollo y no afecta considerablemente el funcionamiento de la aplicación, esta falla puede ser permitida siempre y cuando el sistema no se bloqueé esperando respuesta de la otra aplicación. Como se

ejemplifica en el escenario anterior, la métrica esperada como resultado podría ser que el tiempo de inactividad sea menor a 2 segundos.

2.1.1.2.1.1.2 Modificabilidad.

La modificabilidad de un sistema es referente al costo de realizar cambios a este, es decir cuán fácil es poder sustituir componentes de software o cambiar la implementación de un componente sin modificar los componentes que se encuentran relacionados con este. Para realizar un cambio es necesario considerar dos aspectos muy importantes [2].

- **¿Qué elementos pueden cambiarse?** Cualquier elemento dentro del sistema puede ser cambiado, ejemplos de esto son: la plataforma donde es ejecutado el sistema (hardware, sistema operativo, etc.), el ambiente en el cual el sistema se ejecuta (sistemas con los cuales se puede realizar interoperación, los protocolos usados para comunicar con el resto del mundo, etc.), las cualidades que el sistema presenta (desempeño, confiabilidad, y considerar posibles modificaciones futuras), y su capacidad (número de usuarios soportados, número de operaciones simultaneas, etc.). Pero algunas partes del sistema, tales como la interfaz de usuario o la plataforma, son lo suficientemente distinguibles y sujetas a cambios que pueden ser consideradas por separado. Por lo que es posible considerar los cambios de plataforma dentro del una sub categoría del atributo de calidad que es conocida como portabilidad [2]. Como consecuencia de un cambio sobre un elemento deben ser analizados las consecuencias de dicho cambio, es decir que elementos son afectados de manera directa e indirecta por este cambio y cuáles de ellos deben ser modificados para poder ajustarse a la nueva configuración del sistema y permitirle a este desempeñarse correctamente.

- **¿Cuándo se debe realizar el cambio y por quien?** Anteriormente un cambio era realizado directamente en el código fuente. Esta actividad era realizada por un desarrollador, el cual posteriormente se encargaba de realizar las pruebas necesarias para la validación del componente modificado y la integración de este con el sistema completo. Actualmente se puede considerar que los cambios pueden ser realizados durante diferentes etapas del desarrollo o por diferentes interesados. Los cambios pueden ser hechos a la implementación (código fuente), durante la compilación (usando interruptores durante la fase de compilación), durante la construcción (selección de librerías), durante la configuración (por un rango de técnicas, incluyendo ajuste de parámetros) o durante la ejecución (conjunto de parámetros). Un cambio puede ser realizado por un desarrollador, un usuario final o un sistema de administración [2].

Una vez que un cambio ha sido especificado, la nueva implementación debe ser diseñada, implementada, probada y puesta en práctica. Todas estas acciones influyen en el desarrollo de una aplicación pues repercuten en el tiempo de desarrollo y el costo de la aplicación, justamente el costo y tiempo pueden ser considerados para realizar la medición de la modificabilidad.

Como se mencionó anteriormente un aspecto a considerar en el momento de realizar cambios es el número de elementos afectados por un cambio, ya que este trabajo implica más tiempo en la modificación de componentes. Justamente esto se ve representado en el siguiente escenario de modificabilidad.

Elemento del escenario	Valores
Fuente de estimulo	Desarrollador.
Estimulo	Se desea cambiar la interfaz de usuario.
Artefacto	Código o componentes de software relacionados con la interfaz de usuario.
Ambiente	Etapas de diseño del sistema en desarrollo.
Respuesta	Modificaciones realizadas sin afectar otros componentes del sistema.
Medida de la respuesta	Los cambios deben ser realizados en un plazo no mayor de 1 día.

Tabla 2. Escenario de modificabilidad.

En la Tabla 2 se expone un escenario de modificabilidad, el cual consiste en realizar un cambio a la interfaz de usuario. Para este escenario la respuesta y medida de la respuesta pueden ser controladas al hacer que los componentes de la interfaz no estén fuertemente acoplados a otro tipo de componentes del sistema.

2.1.1.2.1.1.3 Desempeño.

El desempeño es referente a los eventos (interrupciones, mensajes, solicitudes de usuario, o tiempo transcurrido) que ocurren y a los cuales el sistema debe responder. Básicamente el desempeño considera cuánto tiempo le toma al sistema responder a un evento [2].

El desempeño se vuelve complicado por los diferentes tipos de eventos que pueden existir durante la ejecución de una aplicación y las distintas fuentes de estos eventos. Los eventos pueden arribar de solicitudes de usuario, de otros sistemas, o del sistema mismo. Por ejemplo, para un sistema financiero basado en web, los eventos son generados de las solicitudes de sus usuarios, como tal el desempeño podría medirse por el número de transacciones que pueden ser procesadas en un minuto.

Otro factor importante a considerar en el desempeño es la administración de recursos, pues de esta depende, en gran parte, el tiempo y la conducta que el sistema tome para responder una petición. El siguiente escenario de desempeño representa aspectos a considerar de la administración de recursos durante las transacciones del sistema.

Elemento del escenario	Valores
Fuente de estimulo	Usuarios que operan el sistema.
Estimulo	Inicio de transacción, se realiza una solicitud de información.
Artefacto	Sistema administrador de transacciones.
Ambiente	El sistema se encuentra operando de forma normal con un número de 50 usuarios accediendo simultáneamente.
Respuesta	Procesar las transacciones.
Medida de la respuesta	Tiempo de espera promedio no mayor a 2 segundos.

Tabla 3. Escenario de desempeño.

En la Tabla 3 se expone un escenario de desempeño, el cual consiste en implementar un adecuado manejo de transacciones para controlar el tiempo de espera en que una operación solicitada por el usuario (transacción) será procesada.

2.1.1.2.1.1.4 Seguridad.

La seguridad es una manera de juzgar la habilidad de un sistema para resistir a peticiones de usuarios no autorizados mientras éste continua proporcionando servicios a los usuarios registrados. Un intento para traspasar la seguridad es llamado ataque o amenaza y este puede tomar un gran número de formas con distintos objetivos específicos. Algunos de estos objetivos pueden ser: intentar de manera no autorizada acceder a datos o servicios, modificar información restringida, denegar el acceso o servicios a usuarios legítimos (registrados con distintos permisos de operar el sistema) [2].

La seguridad puede ser caracterizada como un sistema que provee cualidades principalmente de no-rechazo, confiabilidad, integridad, confianza, disponibilidad y verificación [2].

- **No-rechazo.** Es la propiedad de un sistema que especifica que una transacción ejecutada dentro de este no puede ser negada por ninguna otra transacción en ejecución. Esto significa que un usuario, que no cuente con los privilegios necesarios, no puede denegarle los servicios de sistema al resto de los usuarios.

- **Confidencialidad.** Es la propiedad de que los datos o servicios son protegidos de accesos no autorizados. De acuerdo con el tipo de usuario o sus privilegios dentro del sistema, un usuario solo puede utilizar los servicios y acceder a información que le sean permitidos.

- **Integridad.** Esta es la propiedad que hace referencia a que los datos o servicios proporcionados no son alterados o corrompidos durante su envió o funcionamiento.

- **Garantía o confianza.** Es la propiedad de que las partes de una transacción que fueron recibidas contienen lo que la fuente envió.

- **Disponibilidad.** Esta propiedad especifica las condiciones en las cuales el sistema estará disponible para los usuarios.

- **Verificación de transacciones.** Es la propiedad que especifica los niveles de verificación que permitirán reconstruir alguna transacción o actividad que haya sido interrumpida durante su ejecución.

Un escenario de seguridad se presenta en la Tabla 4:

Elemento del escenario	Valores
Fuente de estimulo	Usuario identificado correctamente.
Estimulo	Operación de modificar información.
Artefacto	Datos dentro del sistema.
Ambiente	El sistema se encuentra operando de manera normal.
Respuesta	El sistema mantiene registros de los usuarios identificados.
Medida de la respuesta	Las sesiones de los usuarios pueden ser recuperados durante las siguientes 2 horas después de ser identificado por primera vez.

Tabla 4. Escenario de seguridad.

En la Tabla 4 se presenta un escenario de seguridad, en el cual se especifica que el sistema debe implementar un mecanismo para poder almacenar las sesiones de usuarios registrados durante un periodo de tiempo.

2.1.1.2.1.1.5 Facilidad de pruebas.

Un software fácil de probar se refiere a la facilidad con la cual el software puede ser construido para posteriormente encontrar defectos o problemas directamente probando sus componentes [2].

Para que un sistema sea apropiadamente fácil de probar, este debe ser capaz de controlar cada estado interno de los componentes y las entradas para así observar las salidas. Frecuentemente esto se hace directamente usando un software específico para realizar pruebas. Algunas de las posibles medidas utilizadas en los escenarios de facilidad de pruebas son: el tiempo en realizar una prueba (tiempo promedio registrado en realizar una prueba a un componente de un tamaño específico) y el número de errores encontrados durante la ejecución de la prueba (se realiza una estimación de la probabilidad de encontrar más fallas después de realizar las pruebas a un componente).

Las pruebas son hechas por varios desarrolladores, probadores, verificadores, o usuarios y es el último paso realizado en varias partes del ciclo de vida del software.

Elemento del escenario	Valores
Fuente de estimulo	Probador
Estimulo	Ejecución de la prueba unitaria sobre un componente de software específico.
Artefacto	Componente del sistema.
Ambiente	Momento de realización de pruebas unitarias.
Respuesta	El componente responde a los eventos programados en la prueba, mediante el uso de sus interfaces y salidas.
Medida de la respuesta	Tiempo de realización de pruebas unitarias no mayor a 3 horas por componente.

Tabla 5. Escenario de facilidad de pruebas.

El escenario descrito en la Tabla 5, consiste en minimizar el tiempo de realización de pruebas, realizando un diseño de componentes que permita controlar a los componentes por separado.

2.1.1.2.1.1.6 Usabilidad.

El atributo de calidad de usabilidad es concerniente a que tan fácil un sistema puede ser usado por el usuario para realizar una tarea deseada. Algunos de los enfoques que puede tomar la facilidad de uso pueden ser catalogados dentro de las siguientes áreas [2]:

- **Uso eficiente de un sistema.** Que características debe tener el sistema para lograr que los usuarios sean más eficientes en las operaciones que realizan con el sistema.
- **Minimizando el impacto de error.** Que puede hacer el sistema para que un error de usuario tenga un impacto mínimo.
- **Incrementando la confianza y la satisfacción.** Que hace el sistema para dar la confianza al usuario que la acción correcta está siendo tomada.

Elemento del escenario	Valores
Fuente de estimulo	Usuarios.
Estimulo	Minimizar el impacto de errores.
Artefacto	Sistema.
Ambiente	Tiempo de ejecución del sistema.
Respuesta	Cancelar operaciones actuales.
Medida de la respuesta	Tiempo de respuesta del sistema (cancelación) menor a 1 segundo.

Tabla 6. Escenario de usabilidad.

El escenario de usabilidad descrito en la Tabla 6, el cual especifica como mediante el uso de mecanismos internos para controlar el tiempo de respuesta el sistema puede incrementar la confianza y satisfacción que percibe el usuario.

Hasta ahora solo se ha hecho mención de los atributos de calidad y cómo es posible controlar aspectos en cada uno de ellos para desarrollar un sistema de acuerdo a las especificaciones del usuario. Sin embargo existe otro requerimiento no funcional que también debe ser considerado para el diseño de la arquitectura de un sistema de software, este requerimiento no funcional son las restricciones especificadas por el cliente. En la siguiente sección se mencionará cual es la definición que se ha adquirido y la cual será utilizada en el desarrollo de este proyecto de investigación.

2.1.1.2.2 Restricciones.

Las restricciones son limitaciones de diseño e implementación que deben ser consideradas por el desarrollador para el diseño y construcción del producto. Este tipo de requerimiento no funcional determina la estructuración de componentes (diseño) y la tecnología (implementación) para la puesta en ejecución de la aplicación. Ejemplos de restricciones son: la utilización de un manejador de base de datos en específico, la distribución de los servicios y recursos en distintos huéspedes (computadoras), la utilización de un determinado protocolo de comunicación, o que el sistema sea accedido de manera remota por un navegador web [6].

Las restricciones de diseño deben ser consideradas para la selección del patrón arquitectónico, ya que este patrón se debe adecuar a la aplicación que se esté desarrollando. Este concepto junto con la definición de tácticas de atributos de calidad será definido en la siguiente sección dedicada a definir el diseño de una aplicación, específicamente de una arquitectura de software.

2.1.2 Diseño de arquitecturas de software.

El diseño de una arquitectura es una tarea muy importante durante el proceso de desarrollo de software y consiste de actividades que permitan definir la arquitectura adecuada para la aplicación que se está desarrollando. Para esto se deben considerar las necesidades del usuario, para así tomar las decisiones adecuadas para desarrollar la arquitectura. Durante

el diseño de la arquitectura se consideran los requerimientos de software, funcionales y no funcionales. Para esto, de acuerdo al SEI, se hace uso de los escenarios correspondientes a los atributos de calidad que el sistema soportará. Posteriormente de acuerdo con las restricciones que el usuario proporciona se realiza la selección y adaptación de uno o varios patrones arquitectónicos (distintas formas de organizar componentes de software) que serán usados para estructurar al sistema. Una vez seleccionados los patrones arquitectónicos se realiza la descomposición de las grandes estructuras de los patrones en elementos con responsabilidades específicas. Por último se define el comportamiento y funcionalidad de cada uno de los componentes del sistema, parte de esto se realiza mediante la asignación de tácticas de atributos de calidad y más patrones.

La sección siguiente presenta mayor detalle relativo a las tácticas de atributos de calidad, específicamente aquellas que serán implementadas en el contexto de este proyecto.

2.1.2.1 Tácticas para el control de atributos de calidad.

Una táctica es la manera en que los diseñadores y arquitectos deciden organizar los componentes del sistema, así como asignarle características específicas a cada uno de ellos para controlar sus comportamientos y salidas (posibles respuestas a eventos) para así soportar uno varios atributos de calidad específicos. Es decir, las tácticas son decisiones de diseño que permiten controlar la respuesta del sistema con respecto a un atributo de calidad. Un ejemplo de esto, es que al realizar la selección de una organización de componentes específica se adoptan varias tácticas que permiten responder a estímulos de acuerdo al tipo de sistema que se esté desarrollando [2].

En el catálogo del SEI se definen 52 tácticas dentro de las 6 categorías de atributos de calidad, 13 de disponibilidad, 14 de modificabilidad, 8 de desempeño, 9 de seguridad, 4 de facilidad de pruebas y 4 de usabilidad. En el contexto de ésta tesis no se hace uso del catálogo entero, sino de un sub-conjunto de tácticas que están soportadas por el *Framework* de desarrollo seleccionado.

En las siguientes secciones se definen un sub conjunto de tácticas de los atributos de calidad que se presentaron hasta ahora.

2.1.2.1.1 Tácticas de disponibilidad.

Las tácticas de control de disponibilidad están principalmente clasificadas en tres categorías: detección de fallas, recuperación y prevención. Cada una de ellas se enfoca en implementar acciones preventivas o reactivas, aunque si los requerimientos así lo especifican es posible utilizar un subconjunto de distintas tácticas para el control del comportamiento del sistema.

2.1.2.1.1.1 Detección de fallas.

Dentro de las tácticas reactivas de disponibilidad se encuentra la detección de fallas en el desarrollo de software. Una de las tácticas empleadas para la detección de fallas en los lenguajes de programación abstractos (orientados a objetos) es el uso de manejo de excepciones. Este método consiste en el reconocimiento de fallas el cual consiste en encontrar una falla (una excepción) dentro de la ejecución de un componente del sistema, este componente indica el tipo de falla que ha ocurrido a un componente específico que se encarga de tratar la falla [2].

2.1.2.1.1.2 Recuperación de fallas.

Otro tipo de tácticas reactivas de disponibilidad son las tácticas de recuperación, estas consisten en implementar componentes dentro de una configuración de software que permita recuperar y reparar el sistema. Usualmente las tácticas de recuperación más utilizadas en el desarrollo de software son aquellas que se implementan sobre sistemas cliente-servidor; ejemplos de sistemas que implementan este tipo de configuración son los manejadores de bases de datos [2]. Una de las tácticas de recuperación de fallas adoptada por los sistemas actuales es el uso de *checkpoints*, frecuentemente utilizada para reparar el sistema o regresarlo a un estado coherente. Un *checkpoint* es un registro de un estado consistente del sistema creado periódicamente o en respuesta a eventos específicos. Algunas veces un sistema falla de una manera inusual, dejando al sistema en un estado de inconsistencia. En este caso, el sistema debería ser restaurado usando un *checkpoint* previo de un estado consistente y actualizando al sistema utilizando una bitácora de transacciones en la cual están registradas todas las transacciones que han ocurrido desde el momento de la falla.

2.1.2.1.1.3 Prevención de fallas.

Las tácticas de prevención de fallas son medidas preventivas considerando un conjunto de posibles fallas a ocurrir. Una de las tácticas más utilizadas en sistemas que realizan persistencia o manejo de información de manera concurrente es la táctica conocida como manejo de transacciones. Esta táctica consiste en realizar una transacción, que es un conjunto de pasos secuenciales de manejo de datos, como una sola operación de manipulación de datos. Este manejo transaccional permite deshacer de un solo golpe todos los pasos secuenciales que conforman la transacción, si es que llegase a ocurrir una falla. Normalmente las transacciones son utilizadas para evitar cualquier pérdida de datos cuando el sistema falla o colapsa, así como también para evitar colisiones entre hilos simultáneos de ejecución que estén accediendo a recursos compartidos [2].

2.1.2.1.2 Tácticas de modificabilidad.

Las tácticas de modificabilidad tienen como objetivo controlar el tiempo y costo de cambios en implementación, pruebas, y puesta en ejecución. Los distintos tipos de tácticas están organizadas dentro de conjuntos acorde con sus metas específicas. Uno de estos conjuntos tiene como meta la reducción del número de módulos que son directamente afectados por un cambio, este conjunto de tácticas son conocidas como “Localización de modificaciones”. Otro de los conjuntos tiene como meta limitar las modificaciones en los módulos localizados y son conocidas como “Prevención de

efecto de onda". Un tercer grupo tiene como meta controlar el tiempo de desarrollo y costo "Retraso de tiempo de ligado" [2]. A continuación se describe más a detalle el manejo que se hace de cada una de estos tipos de tácticas.

2.1.2.1.2.1 Localización de modificaciones.

El principal objetivo de este conjunto de tácticas de modificabilidad es minimizar el número de módulos que necesitarán ser modificados en iteraciones posteriores del desarrollo. Cada una de estas tácticas agrega estrategias de diseño que minimizan el costo de posibles modificaciones. Las siguientes tácticas de localización de modificaciones son empleadas durante la fase de diseño e implementación del proceso de desarrollo [2]:

- **Mantener coherencia semántica.** Esto se refiere a la cohesión y acoplamiento entre componentes de software de la aplicación. El objetivo es asegurar que un componente con responsabilidades comunes (alta cohesión de responsabilidades) no tengan que depender demasiado de componentes con distintas responsabilidades (bajo acoplamiento entre componentes).

- **Anticipar cambios esperados.** Esta es otra táctica que hace uso de asignación de responsabilidades comunes a los módulos, pues al tener componentes débilmente dependientes se evita modificar demasiados módulos.

- **Generalización de módulos.** Tomando en cuenta posibles cambios, el módulo es configurado para poder recibir distintos tipos parámetros e inter operar de manera general con varios módulos.

- **Limitar opciones posibles.** Las modificaciones pueden afectar muchos módulos. Esto se puede limitar mediante la restricción de posibles opciones. Un ejemplo de estas tácticas son los *plugins* de un navegador o el limitar el tipo de entrada aceptada en una caja de verificación de información.

2.1.2.1.2.2 Prevención de efecto de onda.

El efecto de onda de una modificación se refiere a la necesidad de realizar cambios a módulos que no son afectados directamente por el cambio. Es decir, si es necesario modificar un componente de software pero existen componentes que dependen de este, entonces esos componentes deberían ser modificados para poder operar con la nueva implementación del componente del cual dependen, este tipo de tácticas intenta minimizar el número de módulos que necesitarán ser modificados por consecuencia de modificar el primer componente [2].

En algunos lenguajes de programación orientados a objetos, como Java, permiten el uso de interfaces de un componente particular; estas interfaces permiten interactuar con el componente pero a su vez ocultan la implementación del mismo. Con el uso de interfaces se logra romper la dependencia entre componentes, ya que las interfaces sirven como intermediario. El manejo de interfaces es una de las tácticas de prevención de efecto de onda, sin embargo no es la única ya que actualmente en el desarrollo de software se hace uso de varias tácticas de este tipo. A continuación se enlistan algunas de las tácticas más utilizadas hoy en día en los lenguajes de programación de última generación.

- **Encapsulamiento.** Esta táctica consiste en la descomposición de responsabilidades y elementos de una entidad o componente de software, en donde parte de estas responsabilidades y elementos son ocultados para el resto de los componentes, permitiendo únicamente manipular aquellos que únicamente el componente publica al resto de los componentes.

- **Uso de intermediario.** Esta táctica consiste en introducir un componente específico entre dos módulos para deshacer la dependencia entre ellos. Si existen cualquier tipo de dependencia entre dos componentes es posible utilizar un intermediario entre ellos que maneje actividades asociadas con las dependencias. Ejemplos de intermediarios son:

    - **Datos.** Los repositorios o medios de almacenamiento actúan como intermediarios entre entidades productoras y consumidoras.

    - **Servicios.** Los patrones de fachada, puente, mediador, estrategia, proxy y fábrica todos proveen intermediarios que convierten la sintaxis de un servicio para que puede ser comprendida y utilizada por otros servicios. Por lo tanto todos ellos pueden ser usados para prevenir cambios que sean propagados entre componentes [25].

    - **Manejo de recursos.** Un manejador de recursos es un intermediario que es responsable de realizar la localización de los recursos. Ciertos manejadores de recursos pueden garantizar satisfacer las solicitudes sin algún tipo de restricciones. Los módulos o componentes ceden el control del recurso al manejador de recursos.

    - **Fabricas.** Esta táctica consiste en la implementación de un componente que tiene la habilidad de crear las instancias de los componentes que sean necesarias, con esto la dependencia entre componentes existentes es satisfecha por las acciones de la fábrica.

2.1.2.1.2.3 Retraso de tiempo de ligado.

Este tipo de tácticas de modificabilidad tienen como objetivo retrasar el momento en que se realiza el ligado entre componentes, es decir la configuración del sistema es especificada hasta el momento de su ejecución. Con el uso de esta táctica es posible que los desarrolladores puedan modificar o sustituir componentes durante el proceso de desarrollo para la realización de pruebas de los componentes o de módulos del sistema. Algunas de las tácticas más utilizadas son [2]:

- **Registro en tiempo de ejecución.** Consiste en realizar procesos de registro de módulos o componentes, utilizando operaciones *plug-and-play* (enchufar y listo) al momento de la ejecución.

- **Archivos de configuración.** Consiste en utilizar archivos de configuración que permitan especificar los parámetros de entrada necesarios al momento en el que el sistema arranca.

- **Polimorfismo.** Es un principio de programación que es utilizado para poder realizar enlaces tardíos (en tiempo de ejecución) en llamadas a métodos. Esta táctica es utilizada cuando un método tiene varias implementaciones.

2.1.2.1.3 Tácticas de desempeño.

El objetivo principal de las tácticas de desempeño es configurar el sistema de software para generar una respuesta a un evento con algunas restricciones de tiempo y recursos. Las tácticas de desempeño controlan el tiempo con el cual una respuesta es generada durante la ejecución del sistema. Para poder realizar la configuración adecuada de desempeño del sistema se debe considerar los dos principales contribuidores del tiempo de respuesta [2]:

- **Consumo de recursos.** Los recursos manejados por los sistemas de software son: tiempo de CPU, datos almacenados, ancho de banda consumido en las comunicaciones de red, memoria utilizada y el uso de entidades definidas para un sistema particular.

- **Tiempo de bloqueo.** Durante la ejecución de un proceso de cómputo, este puede ser bloqueado por requerir de un recurso que no se encuentra disponible, o porque el proceso depende del resultado de otro, que aun no se encuentra disponible.

Para configurar el manejo adecuado de los factores que afectan el desempeño de un sistema, existen tres categorías en las cuales las tácticas de desempeño se encuentran clasificadas de acuerdo a la manera en que se manejan los recursos en cada una de ellas. Las tres categorías son: demanda de recursos, administración de recursos y arbitraje de recursos.

2.1.2.1.3.1 Demanda de recursos.

Las tácticas que se encargan de controlar la demanda de los recursos consideran varios factores para esto, estos factores son: tiempo entre eventos para utilizar un recurso (tiempo de espera), porcentaje del recurso consumido por cada evento y frecuencia con la que el evento ocurre [2].

Las tácticas que se encargan de controlar el tiempo de espera se enfocan en reducir los recursos requeridos o reducir el número de eventos procesados. Las tácticas que se encargan de reducir los recursos son:

- **Incremento de la eficiencia computacional.** Esta táctica consiste en mejorar el algoritmo utilizado en procesamientos para disminuir el tiempo de ejecución y por lo tanto reducir el tiempo de espera entre eventos.

- **Reducir la sobrecarga de cómputo.** Consiste en reducir el procesamiento realizado para eventos específicos, es decir si existen eventos que son no muy frecuentes se debe reducir los recursos que se requieran para generar las respuestas a esos eventos. Por ejemplo, el uso de intermediarios incrementa los recursos consumidos en procesar un evento, y el remover estos disminuye el tiempo de espera. Sin embargo, para la implementación de esta táctica se pierde cualidades de modificabilidad, este es un clásico compromiso entre modificabilidad/desempeño.

Como se mencionó la otra forma de reducir el tiempo de espera es reducir el número de eventos procesados. Esto puede ser realizado de una de las siguientes formas:

- **Administrar la cantidad de eventos.** Consiste en reducir la frecuencia de muestreo en la cual las variables de ambiente son monitoreadas, la demanda puede ser reducida.

- **Control de la frecuencia de muestreo.** Consiste en un bajo monitoreo de las solicitudes encoladas del sistema.

Las tácticas mencionadas anteriormente, frecuentemente resultan en la pérdida de solicitudes y por tal razón no son usadas en el desarrollo de sistemas críticos. Otras tácticas para reducir o manejar la demanda involucran controlar el uso de los recursos, tales como:

- **Limitar el tiempo de ejecución**. Consiste en colocar un límite en el tiempo de ejecución que es usado para responder a un evento.

- **Limitar el tamaño de las colas.** Esta tácticas controlan el número máximo de solicitudes en la cola y consecuentemente los recursos usados para procesar las solicitudes.

2.1.2.1.3.2 Administración de recursos.

Las tácticas de administración de recursos se encargan de controlar el uso de estos y por lo tanto reducir los tiempos de respuesta. Algunas tácticas de manejo de recursos son [2]:

- **Introducir concurrencia.** Consiste en introducir mecanismos que permitan manejar varias solicitudes en paralelo, con esto se logra reducir el tiempo de bloqueo. La concurrencia puede ser introducida para procesar diferentes eventos en diferentes hilos o para crear hilos adicionales para procesar diferentes conjuntos de actividades.

- **Mantener múltiples copias de datos o cálculos.** El propósito de manejar replicas es reducir la carga de los cálculos computacionales o accesos a datos realizados en un servidor. Sin embargo, al utilizar esta técnica se debe tener cuidado con la sincronización en el manejo de los recursos.

- **Aumentar recursos disponibles.** Consiste en aumentar los recursos que permitan reducir el tiempo de espera, algunos de estos recursos son: procesadores, memoria, redes, etc. En la mayoría de los casos también es posible realizar la sustitución por recursos con mayor velocidad de procesamiento, almacenamiento o transferencia de datos. Sin embargo el uso de estas tácticas definitivamente incrementan el costo del sistema.

2.1.2.1.3.3 Arbitraje de recursos.

Las tácticas de arbitraje de recursos se enfocan en el manejo de solicitudes o eventos dentro de un sistema. Estas tácticas consisten en políticas de calendarización de solicitudes. Una política de calendarización conceptualmente tiene dos partes: asignación de prioridad y despacho. Todas las políticas de calendarización asignan prioridades. Los criterios de

competición por la calendarización incluyen optimización de recursos usados, importancia de la solicitud, minimizar el número de recursos usados, minimizar de tiempo de espera, maximizar el rendimiento, etc. [2].

2.1.2.1.4 Tácticas de seguridad.

Las tácticas de seguridad tienen como propósito mantener a los componentes del sistema y los datos de este en un estado coherente. Las tácticas de seguridad pueden ser divididas de acuerdo a sus objetivos principales: restringir ataques, detectar ataques y recuperarse de ataques. Hoy en día las aplicaciones hacen uso de diferentes tácticas de seguridad, sin embargo el arquitecto debe considerar el conjunto total de tácticas a implementar en una aplicación para encontrar un balance de acuerdo a las necesidades del usuario y así no afectar el control sobre otras tácticas de atributos de calidad [2].

2.1.2.1.4.1 Resistencia a los ataques.

El principal objetivo de estas tácticas es la impedir el acceso de usuarios no deseados al sistema, así como también proporcionar mecanismo de protección y confidencialidad de información. Las principales tácticas implementadas en aplicaciones de software son las siguientes [2]:

- **Autentificación de usuarios.** Son mecanismos que garantizan que un usuario o una computadora remota tengan acceso a funciones del sistema. Algunos de estos mecanismos son: contraseñas, certificados digitales.

- **Autorización de usuarios.** Son mecanismos que garantizan que usuarios autentificados tienen los derechos para acceder y modificar datos o servicios del sistema. El control de acceso puede ser por usuario o por clases de usuarios (grupos, roles, listas).

- **Mantenimiento de confidencialidad de datos.** Son mecanismos que garantizan la protección de datos de usuarios no autorizados. La confidencialidad es usualmente lograda implementando alguna forma de encriptación a los datos y a los enlaces de comunicación.

2.1.2.1.4.2 Detección de ataques.

Otra manera de proteger al sistema es utilizando tácticas de detección de ataques. La detección de un ataque es usualmente a través de un sistema específico de detección de intrusiones. Tales sistemas trabajan para comparar patrones de tráfico de la red hacia una base de datos. Frecuentemente, los paquetes que contiene la información enviada deben ser filtrados para poder realizar la comparación. El filtrado puede ser sobre las bases del protocolo, banderas TCP, tamaños de carga, direcciones de fuente o destino, o por número de puerto [2].

2.1.2.1.4.3 Recuperación de ataques.

Las tácticas de recuperación de ataques se enfocan en realizar las actividades necesarias para poder restablecer el sistema a un estado coherente después de haber recibido un ataque. Estas tácticas pueden ser divididas en las siguientes categorías:

- **Restauración de estado.** Las tácticas usadas en la restauración del sistema o datos son mecanismos que realizan la re estabilización de datos o sistema a un estado coherente.

- **Identificación de ataques.** Consisten en la identificación del atacante para mantener un registro rastreable. Un registro rastreable es una copia de cada transacción aplicada a los datos en el sistema, este registro también incluye información de identificación de la fuente.

2.1.2.1.5 Tácticas de facilidad de pruebas.

El objetivo de las tácticas de facilidad de pruebas es facilitar la realización de pruebas sobre componentes y módulos del sistema. Las dos categorías de tácticas de facilidad de pruebas son: proveer entradas y capturar salidas, y monitoreo interno [2].

- **Proveer entradas y capturar salidas.** Generalmente este tipo de tácticas permiten especificar un tipo de entrada y de acuerdo a esta se espera un tipo de salida. Dentro de este tipo de tácticas se encuentran mecanismos de separación de interfaz e implementación y el uso de componentes específicos para la realización de pruebas.

- **Monitoreo interno.** Esta táctica consiste en utilizar componentes o software especializado para monitorear el comportamiento de los componentes durante su ejecución.

2.1.2.1.6 Tácticas de usabilidad.

Las tácticas de usabilidad se enfocan en facilitarle al usuario las actividades que tenga que realizar con el sistema. Dos de los tipos de tácticas de usabilidad empleadas son [2]:

- **Tácticas de tiempo de ejecución.** Este tipo de tácticas consiste en ayudar al usuario a realizar sus actividades fácilmente. En general consiste en retroalimentar al usuario sobre lo que hace el sistema y permitirle realizar operaciones tales como cancelar, deshacer, etc.

- **Tácticas de tiempo de diseño.** Principalmente este tipo de tácticas consisten en decisiones de diseño que permitan al usuario interactuar con el sistema. Ejemplo de este tipo de tácticas es la separación de la interfaz de usuario del resto de la aplicación. Se implementa con patrones de MVC (Modelo Vista Controlador) [33], mismo que es utilizado en la táctica de coherencia semántica del atributo de calidad de modificabilidad.

2.1.2.2 Patrones arquitectónicos.

Un patrón arquitectónico es la forma de describir, desarrollar, nombrar y capturar técnicas de desarrollo probadas. Un patrón documenta una dupla recurrente de problema-solución dentro de un contexto dado. Cada uno de los patrones está dirigido a cubrir problemas específicos del sistema, generalmente relacionados a restricciones de diseño e implementación. Sin embargo cada uno de los patrones tiene sus ventajas e inconvenientes que deben ser considerados durante el diseño de la arquitectura [37].

Dos ejemplos de patrones arquitectónicos son los patrones cliente-servidor y n-tercios [3] [8]. El primer patrón consiste en separar al sistema en dos estructuras, una de ellas (servidor) debe contener los componentes necesarios para atender peticiones del otro tipo de estructura (cliente) por su parte debe tener los elementos necesarios para poder solicitar y manejar la información que proporcione el servidor. Esta separación de componentes por estructuras se encuentra representada en la Figura 4.

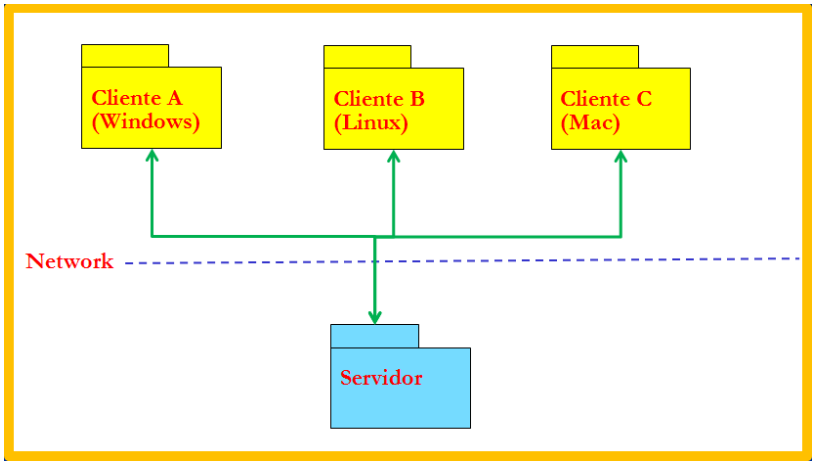

Figura 4. Arquitectura Cliente-Servidor.

Por otra parte el patrón de n-tercios, específicamente tres-tercios, es implementado en la mayoría de las aplicaciones web que atiende peticiones de clientes. Dentro de la estructura del servidor se realiza una separación de componentes por responsabilidades. Principalmente estas responsabilidades se separan en cada uno de los tercios: presentación, negocios, y datos. En cada uno de estos tercios se encuentran componentes o elementos que se encargarán de dar formato a la información que será manejada por las entidades de los clientes (presentación), procesar todas las peticiones de información (negocios) y administrar los datos almacenados por el sistema (datos). La Figura 5 muestra los elementos que pueden encontrarse en el lado del servidor utilizando el patrón tres-tercios.

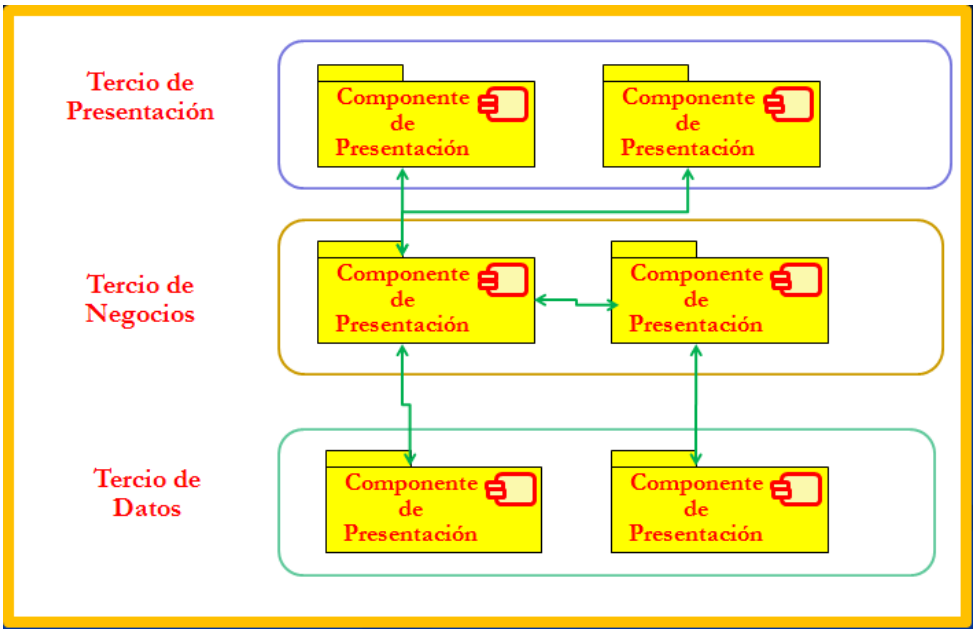

Figura 5. Arquitectura de una aplicación WEB.

Como se puede concluir de la definición de patrón arquitectónico, la selección del o los patrones arquitectónicos es uno de los factores a considerar que definirá la base sobre la cual se modelará la arquitectura de la aplicación. Una vez que la arquitectura de un sistema y cada uno de los componentes de esta han sido especificados, es necesario representar adecuadamente esta arquitectura para que cualquier involucrado dentro del proyecto pueda comprenderla y trabajar sobre el desarrollo del sistema. Justamente en la siguiente sección se describen algunas formas de documentar la arquitectura de aplicaciones.

2.1.3 Documentación de arquitecturas.

Como se ha mencionado en las secciones anteriores, la definición de una arquitectura adecuada es considerada la actividad base del desarrollo de aplicaciones. Sin embargo, el diseño y modelado de la arquitectura de una aplicación no es tarea sencilla, así como también la correcta representación de esta. Una forma de documentar una arquitectura es mediante el uso de diferentes vistas que proporcionen diferentes grados de abstracción para los cada uno de los involucrados en el desarrollo del sistema. Para poder modelar estas diferentes vistas de una arquitectura se utilizan lenguajes de descripción o modelado de sistemas. En la siguiente sección se hace una descripción de las características principales de estos lenguajes y como es que son utilizados para documentar el diseño de una arquitectura.

2.1.3.1 Lenguajes de descripción de arquitectura (ADL).

Un Lenguaje de Descripción de Arquitectura (ADL) es un lenguaje que se utiliza para describir de forma abstracta la arquitectura de software. Estos lenguajes proporcionan mecanismos para modelar la arquitectura conceptual de un sistema, principalmente se centran en la estructura de alto nivel de la aplicación (estructuración de los componentes) y no

en los detalles de implementación de los módulos. La mayoría de estos lenguajes son propuestas académicas de universidades que fueron surgiendo desde comienzos de la década de 1990 hasta la actualidad [9].

En la siguiente sección se detallan cuales son los elementos de alto nivel que los ADLs modelan explícitamente [9] [10].

2.1.3.1.1 Elementos principales de un ADL.

Un ADL describe la estructura de un sistema de software en términos de elementos arquitectónicos de alto nivel. Estos elementos arquitectónicos son los *componentes* y *conectores*, ligados entre ellos con *configuraciones* específicas. Los componentes principales de una arquitectura son los siguientes [3]:

- **Componentes.** Son elementos arquitectónicos que son responsables de describir y representar alguna funcionalidad específica del sistema.
- **Conectores.** Son elementos arquitectónicos utilizados para representar las comunicaciones entre los elementos que componen el sistema.
- **Configuraciones.** Las configuraciones son combinaciones de componentes y conectores en una unidad única y coherente. Las configuraciones pueden ser vistas como conjuntos de componentes que dirigen el comportamiento del sistema de software.

Actualmente los modelos de arquitectura generados con ADLs no permiten especificar el detalle necesario de la aplicación para que los involucrados en el proyecto puedan continuar con el desarrollo de la aplicación. Por lo mismo son muy poco utilizados en el desarrollo de aplicaciones y en algunos casos solo son utilizados como documentación parcial del sistema. Por esta razón los arquitectos y diseñadores utilizan otros mecanismos que les permiten detallar más la estructuración del sistema. En el ambiente empresarial, el diseño y documentación de la arquitectura de software es realizado utilizando diferentes diagramas o modelos de software que representan explícitamente cada una de las partes de la arquitectura. En la siguiente sección se hace mención a otro lenguaje de modelado de sistemas de software, así como sus características y como son utilizados en el proceso de desarrollo de software [9] [10].

2.1.3.2 Lenguaje Unificado de Modelado (UML)

UML [11] es un lenguaje gráfico que permite visualizar, especificar, construir y documentar artefactos de un sistema de software. Es decir, UML es una herramienta que permite generar diseños representativos del sistema que son utilizados por los desarrolladores para comenzar la construcción de una aplicación de software. Estos diseños son diagramas que representan lo que hará el sistema y cada uno de ellos se enfoca en mostrar, a un grupo de involucrados, el nivel de detalle necesario para el desarrollo del sistema, este nivel de detalle va desde la distribución de los componentes de software en entidades físicas hasta funcionalidades que deben implementar cada uno de los componentes de la aplicación.

Dentro de los diagramas UML existe un diagrama que muestra la organización y dependencia entre el conjunto de componentes de software, estos componentes son una representación física (en algunos casos conceptual) de una o más clases o elementos físicos (archivos de configuración, código, componentes de base de datos, etc.) del sistema. Es decir, los componentes de software que pueden ser modelados dentro de este diagrama representan elementos específicos de plataformas de desarrollo.

El diagrama de componentes es utilizado para modelar una vista estática de la implementación del sistema. Esta representación del sistema principalmente soporta la configuración de un conjunto de partes del sistema, componentes que pueden ser ensamblados de distintas formas para producir un sistema en ejecución. Este diagrama normalmente es utilizado para modelar código fuente, ejecutables, bases de datos físicas y sistemas adaptables. La Figura 6 muestra un diagrama de componentes en donde se puede observar componentes como, tablas de la base de datos (<<Subsystem Data Base>>, archivos de presentación de información (Login.jsp), y componentes de negocio (ModificarInfo.java) [11].

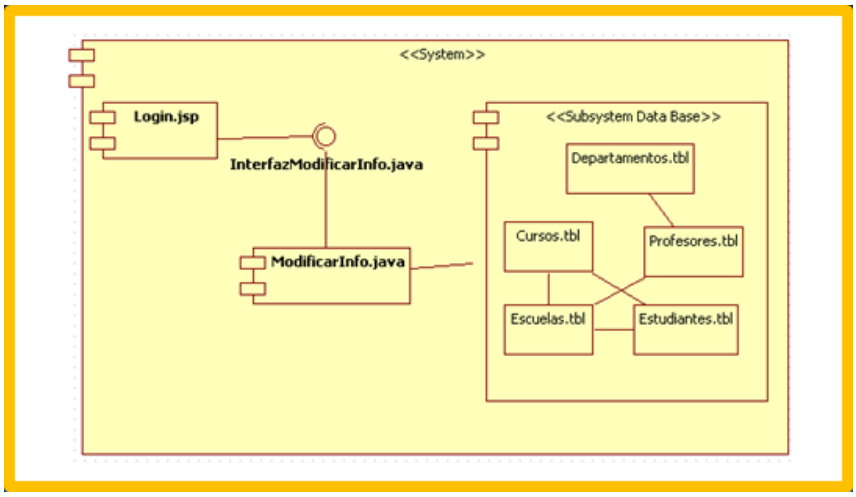

Figura 6. Diagrama de componentes.

Aunque los diagramas UML representativos de la arquitectura son muy utilizados en el proceso de desarrollo, en especial el diagrama de componentes, no permiten especificar características de funcionalidad o comportamiento de la aplicación. Es decir, no permiten hacer anotaciones para especificar las propiedades que soportan tácticas de configuración de atributos de calidad. Por esta razón el diagrama de componentes junto con el resto de los diagramas UML son sólo utilizados como documentación en el desarrollo de software. Por tal motivo los arquitectos utilizan estos diagramas junto con la documentación de escenarios de atributos de calidad para representar la arquitectura del sistema y continuar con la construcción del sistema.

Hasta ahora solo se han presentado dos herramientas utilizadas para la representación de una arquitectura de software, los ADL's y UML. En la siguiente sección se presentarán dos de los *Frameworks* utilizados para desarrollar aplicaciones y como cada uno de ellos permiten implementar la arquitectura junto un conjunto de atributos de calidad.

2.2 Modelos de componentes y Frameworks para aplicaciones empresariales.

Los componentes de software son unidades ejecutables de producción, adquisición e implementación independiente que pueden ser colocadas dentro de un sistema funcional. Para permitir la composición entre componentes, un componente de software agrega un modelo de componentes y los elementos de componentes específicos de una plataforma. Un modelo de componentes puede estar compuesto por abstracciones tales como: procedimientos, clases, módulos o aplicaciones enteras [26].

Hoy en día existen una gran variedad de modelos de componentes que permiten soportar distintos tipos de requerimientos no funcionales. Dos de los modelos de componentes utilizados en el desarrollo de sistemas son los *Enterprise JavaBeans* y el *Framework Spring*. Generalmente estos modelos de componentes se ven acompañados por *Frameworks* que apoyan en la realización de actividades tales como el soporte de la persistencia. En las siguientes sub secciones se describirá cada uno de estos modelos así como también las tácticas de control de atributos de calidad que cada uno de estos modelos implementa.

2.2.1 Enterprise JavaBeans.

Los *Enterprise JavaBeans* (EJB) es una tecnología para desarrollar aplicaciones empresariales con la plataforma J2EE. La plataforma EJB facilita el desarrollo basado en componentes de aplicaciones empresariales con un patrón arquitectónico de 3-tercios. Principalmente los tipos de componentes utilizados en EJB y en los que se implementa la lógica de negocios son [12] [13]:

- **EJB de Entidad.** Estos componentes representan datos de negocio así como también implementan mecanismos de acceso a datos. Por este motivo los EJBs de entidad son vistos como componentes de persistencia.

- **EJB de Sesión.** En estos componentes se modelan los procesos de negocio síncronos de un sistema. Existen dos tipos de estos componentes: con estado (objetos distribuidos que limitan el acceso a un solo cliente) y sin estado (objetos distribuidos que permiten accesos concurrentes).

- **EJB dirigidos por mensajes.** Estos componentes modelan procesos de negocio que son accedidos de manera asíncrona. Para realizar la comunicación con este tipo de componentes se hace uso de un sistema de mensajería específico de la plataforma (*Java Messaging System*).

Los EJB's hacen uso de dos elementos para realizar la administración de los recursos y servicios. Estos dos componentes son el contenedor y el descriptor de despliegue.

2.2.1.1 Contenedor.

El contenedor es una entidad que se encarga de la administración de los recursos y servicios del sistema. Para esto, esta entidad se encarga de proporcionar un conjunto de servicios de administración de recursos. Los servicios más comunes proporcionados por un contenedor son:

- **Manejo de ciclo de vida.** El contenedor se encarga de controlar el ciclo de vida de los objetos de la aplicación.
- **Búsqueda.** El contenedor debe proveer mecanismos para obtener las referencias a los objetos manejados.
- **Configuración de objetos.** El contenedor debe proveer mecanismos para configurar la administración de los objetos que se ejecuten en el.
- **Resolución de dependencias.** El contenedor se encargará de manejar las relaciones entre objetos y datos.

En la Figura 7 se muestra una representación grafica del contenedor. El contenedor es una entidad que, además de manejar el ciclo de vida de los componentes, intercepta peticiones hacia componentes (componentes de vista, negocio o integración entre otros) que requieren la ejecución de alguna funcionalidad que necesite el soporte de servicios específicos (RNF) de acuerdo al tipo de componente, así como también se encarga de manejar el ciclo de vida de los componentes. Esta entidad, el contenedor, actúa como un proxy que intercepta las llamadas a los componentes e implementa el manejo de tácticas de atributos de calidad de acuerdo al componente de software que será ejecutado.

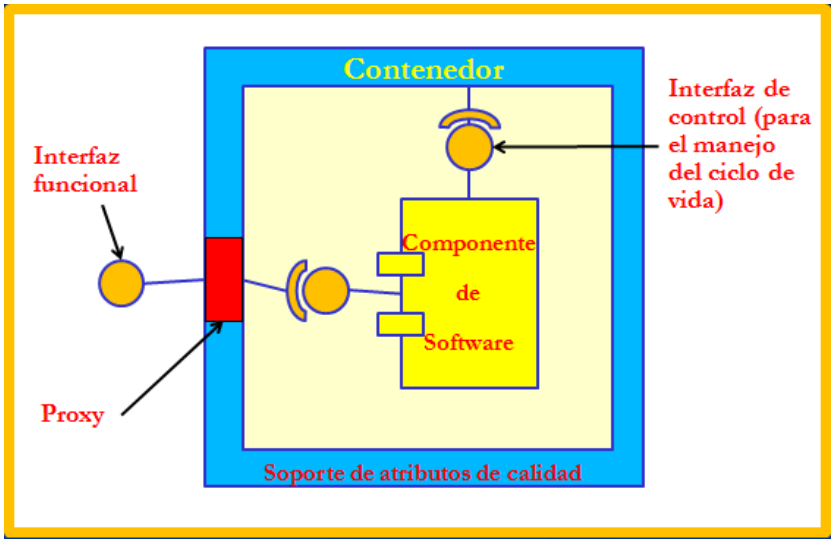

Figura 7. Componente de software dentro de un contenedor.

La configuración del contenedor del sistema es definida mediante un archivo descriptor de despliegue del cual se hablará más a detalle en la siguiente sub sección [13].

2.2.1.2 Descriptor de despliegue.

Un descriptor de despliegue es un archivo XML en el que se configura la administración que el contenedor realiza sobre recursos y atributos de calidad [13].

La utilización del contenedor y el descriptor de despliegue para el manejo de objetos, implica la separación de aspectos no funcionales y la lógica de negocio. Pues, mientras la lógica de negocios es especificada en los componentes de la aplicación, los atributos de calidad pueden ser configurados en el descriptor de despliegue. Esta separación facilita la tarea de los desarrolladores para implementar los requerimientos funcionales y los requerimientos no funcionales de manera independiente [13].

En las siguientes sub secciones se describe como son especificados los atributos de calidad en el descriptor de despliegue del contenedor de EJBs.

2.2.1.3 Principios y tácticas utilizadas en EJB.

Los EJBs son una tecnología que se propone como complemento del lenguaje de programación Java, por lo tanto adquiere algunas de las costumbres utilizadas en este lenguaje, tales como interfaces, agrupar responsabilidades en clases comunes (coherencia semántica), encapsular y aplicar polimorfismo entre otras. Los EJBs permiten administrar objetos y atributos de calidad de dos maneras: programativa y declarativa. Cuando los desarrolladores se encargan de implementar la administración de los recursos en el código se dice que la administración es programativa. Mientras que en la administración declarativa, el contenedor se encarga de la gestión de los recursos [12] [13].

Aunque EJBs, al igual que muchos marcos de trabajo, permite controlar características de los atributos de calidad, en este documento sólo se mencionarán aquellas tácticas de atributos de calidad en los cuales se tiene un control específico de acuerdo a la propuesta de esta tecnología, es decir que son soportados o controlados por el lenguaje, el contenedor y el descriptor de despliegue.

En las siguientes tablas se presenta una lista de algunas de las tácticas de atributos de calidad que son soportadas por EJBs. En la primera columna se encuentra la categoría de la táctica principal que es identificada, en la segunda columna se muestra una de las posibles tácticas específicas que pueden ser empleadas para alcanzar el objetivo de calidad por el *Framework*; finalmente en la última columna se especifica cómo, mediante el uso de elementos determinados, el *Framework* de desarrollo y demás tecnologías empleadas soporta la táctica de atributo de calidad.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Detección de fallas	Manejo de excepciones.	Es soportado por el lenguaje de programación Java.
Recuperación de fallas	Checkpoint/Rollback.	Generalmente es soportado por secciones criticas declaradas en código (try/catch) o por el contenedor, así como por el manejador de base de datos
Prevención de Fallas	Transacciones.	El manejo de transacciones es controlado por el contenedor y especificado dentro del descriptor de despliegue.

Tabla 7. Tácticas de disponibilidad en los EJB's.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Localización de modificaciones	• Mantener coherencia semántica. • Anticipar cambios esperados. • Generalización de módulos.	Estas tácticas son soportadas por el lengua de programación Java (programación orientada a objetos) y por el uso de interfaces.
Prevención de efecto de onda	• Encapsulamiento. • Conservación de interfaces existentes. • Uso de intermediarios.	Tácticas soportadas por técnicas de programación del lenguaje de programación Java.
Retraso en tiempo de ligado	• Registro en tiempo de ejecución. • Archivos de configuración. • Polimorfismo.	Tácticas soportadas por técnicas de programación del lenguaje Java, así como por el contenedor que se encarga de administrar los servicios y ligar los módulos mediante el uso del archivo de configuración del descriptor de despliegue.

Tabla 8. Tácticas de modificabilidad en los EJB's.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Demanda de recursos	• Reducir sobrecarga de cómputo. • Administración de eventos. • Limitar el tamaño de las colas. • Limitar el tiempo de ejecución.	Tácticas soportadas por decisiones de implementación en el código, así como por el contenedor, el cual administra las colas y tiempo de ejecución, entre otras cosas, de acuerdo a la especificación definida en el descriptor de despliegue.
Administración de recursos	Introducción de concurrencia.	Tácticas soportadas por el contenedor dentro de las propiedades del manejador de base de datos.

Tabla 9. Tácticas de desempeño en los EJB's.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Resistencia a los ataques	• Autentificación. • Autorización. • Mantenibilidad de confidencialidad de datos e integridad.	Estas tácticas son soportadas por el contenedor, el cual realiza la implementación especificada en la configuración definida en el descriptor de despliegue.
Recuperación de ataques	Restauración de estado.	Estas tácticas son soportadas por el contenedor dentro de las propiedades del manejador de base de datos.

Tabla 10. Tácticas de seguridad en los EJB's.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Proveer entradas y captura de salidas	• Separación de interfaz e implementación. • Interfaces de prueba especializadas.	Son soportadas por técnicas de programación.

Tabla 11. Tácticas de facilidad de pruebas en los EJB's.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Soporte intuitivo de sistema	Tácticas de tiempo de diseño.	Estas tácticas son soportadas utilizando técnicas y patrones de programación tales como MVC.

Tabla 12. Tácticas de usabilidad en los EJB's.

En las tablas anteriores se puede observar un sub conjunto de las tácticas que son soportadas por los EJB's, este conjunto de tácticas es suficiente para mostrar que la mayoría de las tácticas son soportadas de manera programativa y muy pocas de ellas son controladas por el contenedor. Algunas de las tácticas mencionadas en secciones anteriores no se presentan en estas tablas pues no son soportadas directamente por el *Framework* de desarrollo EJB y tiene que ver más con otros factores, como es el caso de la táctica de aumentar recursos disponibles, en la cual influyen directamente el procesador y la memoria de la máquina que realizará el procesamiento de la aplicación.

Los EJB's son una de las tecnologías utilizadas hoy en día en el desarrollo de aplicaciones web, las cuales son aplicaciones con una gran cantidad de módulos que resuelven problemas de negocios electrónicos. Otro *Framework* utilizado para el desarrollo de aplicaciones web es el *Framework Spring* del cual se hablará en la siguiente sección.

2.2.2 Framework Spring.

Spring [34] es un *Framework* de desarrollo, de código abierto, utilizado para desarrollar aplicaciones sobre plataformas como Java y .NET. Surge como alternativa a los *Enterprise JavaBeans*, la diferencia principal es que este marco puede ser utilizado para desarrollar cualquier tipo de aplicaciones de tal manera que solo necesita la carga de módulos

especializados acorde con el tipo de aplicación (contenedor ligero), por lo tanto no se necesita de una API completa con varios módulos de soporte como en el caso del contenedor de EJB [14] [15].

Las características principales que hacen de *Spring* uno de los marcos de trabajo más utilizados hoy en día son: la Inversión de Control (IoC) o también conocida como inyección de dependencias y la Programación Orientada a Aspectos (AOP). En *Spring* estos principios son manejados por el contenedor ligero, el cual es configurado mediante un archivo XML conocido a como *application context*, cada uno de estos conceptos son explicados en las siguientes secciones.

2.2.2.1 Contenedor ligero y el Application Context.

El contenedor de *Spring* es un contenedor ligero porque a diferencia del contenedor de EJB, el cual para realizar la administración de los recursos necesita del conjunto de APIs de J2EE, el contenedor de *Spring* permite utilizar las herramientas o *Frameworks* de desarrollo especializados para realizar la administración de los recursos que serán utilizados por el tipo de aplicación. Este contenedor se encarga de administrar los objetos (es considerado la fábrica de objetos), permitiendo al desarrollador implementar las funcionalidades de la aplicación sin la necesidad de preocuparse por la creación y conexión a nivel del código de los objetos mediante el principio de Inversión de Control (IoC). Permite el uso de distintas herramientas para realizar la administración de los recursos utilizando los principios de Programación Orientada a Aspectos (AOP). Los conceptos de IoC y AOP son descritos más adelante.

El contenedor ligero de *Spring* es más rápido al momento de iniciar, y su tiempo de ejecución varía dependiendo de las herramientas o *Framework* de desarrollo especializados que se usen en el desarrollo de cada aplicación. La configuración de los recursos y atributos de calidad son definidos en un archivo XML (*application context*), similar al descriptor de despliegue de EJB. Este archivo de configuración permite definir el manejo de atributos de calidad aun si se utilizan diferentes *Frameworks* especializados para el manejo de estos (AOP), también permite especificar el manejo de las relaciones entre componentes de software (IoC) [17].

2.2.2.2 Inversión de Control (IoC).

La Inversión de control (IoC) es un principio de diseño utilizado para controlar las relaciones entre los componentes u objetos de la aplicación. Este principio permite que una entidad externa de los componentes de software sea la encargada de crear las dependencias (inyectar dependencias) entre los componentes en el momento que sea necesario de acuerdo a las características de configuración del *Framework* de desarrollo. Es decir, en la inversión de control se definen y configuran estas dependencias en una entidad exterior, en *Spring* esta entidad es el contenedor ligero. Es decir, el contenedor ligero de *Spring* se encarga de crear las instancias de los objetos que serán utilizados, así como también se encarga de realizar la inyección de dependencias entre componentes en tiempo de ejecución. En la Figura 8 se muestra el principio de inversión de control dentro de *Spring*, esta figura ejemplifica como es que el contenedor (cuadro azul) inyecta las dependencias entre los componentes de software de acuerdo a la configuración especificada dentro del *applicaction*

context, una de las ventajas obtenidas de utilizar este principio es el bajo acoplamiento entre componentes al evitar que estos tomen la iniciativa de conectarse entre ellos, facilitando así la sustituibilidad [17].

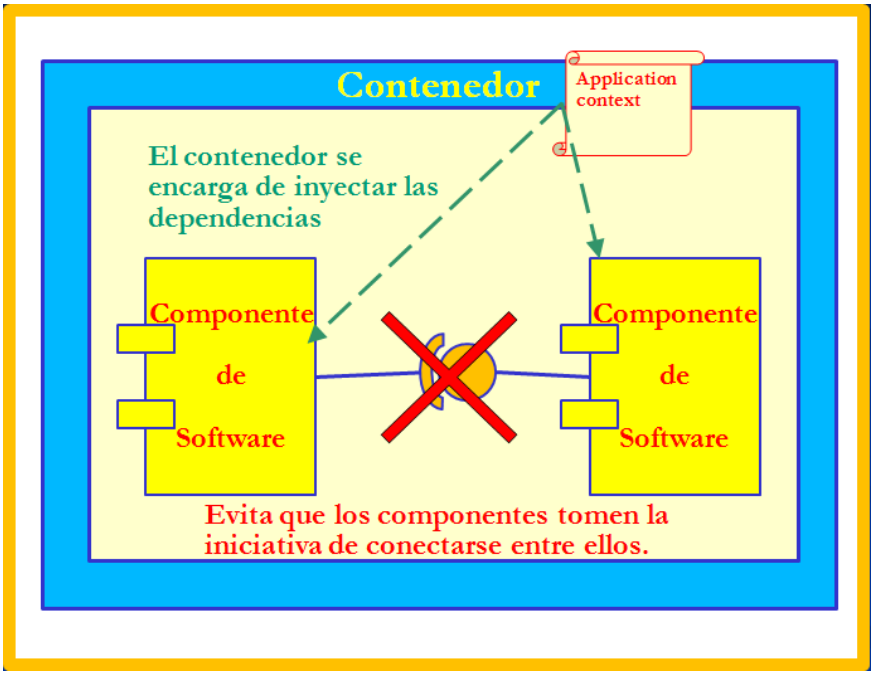

Figura 8. Inversión de control.

Empleando este concepto durante el desarrollo, se disminuye el impacto debido a modificaciones realizadas en posteriores entregas del sistema (reducción del efecto de onda), facilita la comprensión del código, así como también facilita la realización de pruebas ya que permite sustituir fácilmente los componentes del sistema por componentes específicos de pruebas [17].

2.2.2.3 Programación Orientada a Aspectos (AOP).

En el desarrollo de aplicaciones de software existen principalmente dos diferentes tipos de lógicas a desarrollar en un sistema. Una de ellas es la lógica de negocios, la cual involucra todas las funcionalidades principales que definen el dominio del problema o tipo de aplicación que se desarrollará así como también los módulos que la componen (requerimientos funcionales). Esta lógica de negocios es definida dentro de componentes de software con funcionalidades específicas. Por otra parte existen la lógica no funcional o de soporte de RNF (requerimientos no funcionales), esta es considerada lógica secundaria, pues involucra características y atributos que deben estar definidos en un sistema para que esté presente un comportamiento específico bajo ciertas condiciones y durante la ejecución de una funcionalidad específica. La lógica no funcional es definida como servicios o aspectos que son utilizados y definidos dentro de varios componentes de lógica funcional. Es decir, a diferencia de la lógica de negocios, la lógica no funcional es transversal en el sentido de que esta esparcida en varios de los componentes de la aplicación. Algunos ejemplos de lógica no funcional definida como servicios son: manejo transaccional, registro de acceso y manejo de seguridad, por mencionar algunos.

Normalmente, estos módulos son declarados o llamados a lo largo de varios objetos, de lógica de negocios, haciendo difícil el manejo y mantenimiento de estos. La Figura 9 muestra una forma de ver la lógica de soporte de RNF declarada a lo largo de varios componentes de lógica funcional dentro en una aplicación.

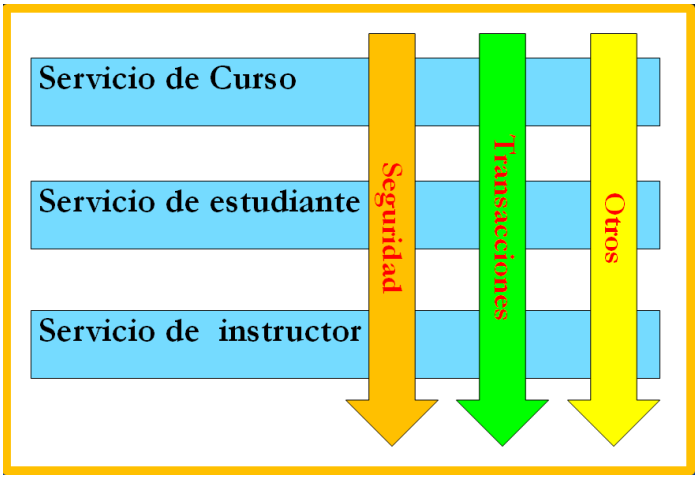

Figura 9. Lógica de soporte de RNF declarada a lo largo de componentes de lógica funcional [24].

La Programación Orientada a Aspectos (AOP) es un paradigma de programación que intenta separar los componentes (lógica de negocios o funcionalidad) del manejo de los aspectos (requerimientos no funcionales) de una aplicación. AOP proporciona mecanismos de abstracción y composición para formar todo el sistema, permitiendo eliminar código disperso en varios componentes, facilitando la comprensión, promoviendo la adaptación y reutilización de componentes de la aplicación [16] [17]. Con AOP es posible definir los aspectos en un lugar específico (*application context*) y no dentro de los componentes que contienen la lógica funcional, pero también es posible definir como y donde estos aspectos son aplicados dentro de los componentes de lógica negocios.

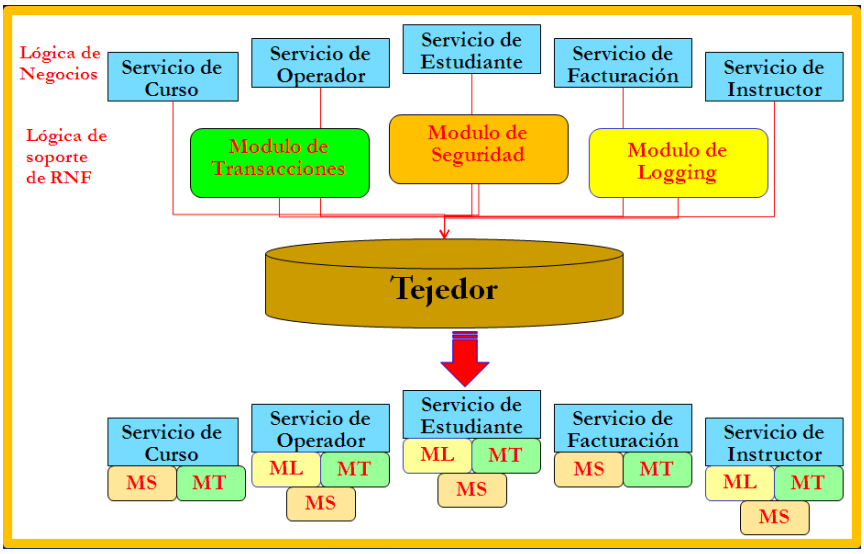

Figura 10. Unión de aspectos y componentes realizada por el tejedor para la ejecución de la aplicación [24].

Una vez que los aspectos y componentes son separados para realizar la implementación de cada uno de ellos, al final deben mezclar funcionalidades para ejecutar la aplicación. Para realizar la mezcla deben ser especificados *puntos de enlace* en el código funcional para especificar comportamientos adicionales (aspectos no funcionales). Después de esto una entidad configurada en el contenedor de *Spring* conocida como el tejedor se encarga de mezclar [Figura 10], mediante los puntos de enlace, los diferentes mecanismos de abstracción y composición en tiempo de ejecución o compilación, lo anterior es válido de manera general para AOP y no solo para *Spring* [16]. El manejo de AOP en *Spring* permite la definición y manejo de aspectos dentro de los componentes de software que los necesiten, esto es realizado utilizando *proxies* creados por el contenedor, esta función es conocida como auto proxy de AOP en *Spring*. En el contexto de *Spring* el tejedor no mezcla la lógica funcional y no funcional, sino que el contenedor introduce *proxies* que manejan aspectos tales como la seguridad, a nivel de las conexiones entre componentes. Este enfoque tiene el mismo resultado que otros enfoques de AOP con la ventaja de que la introducción de manejadores de aspectos se puede hacer al tiempo en que se conectan los componentes.

Spring puede manejar distintas tácticas a través de la introducción de diversos *Frameworks.* Cabe señalar que Spring provee soporte nativo para la introducción de dichos *Frameworks*. Ejemplos de estos *Frameworks* son: *Hibernate* [35] para el manejo de la persistencia de información y *Acegi* [20] para el manejo de seguridad del sistema. A continuación se presentan un poco más a detalle estos 2 *Frameworks*.

2.2.3 Framework Hibernate.

La manera en que el contenedor de *Spring* maneja la persistencia es mediante el uso de herramientas, una de ellas es *Hibernate* [35]. Este es un *Framework* que se encarga, junto con Spring, de realizar la persistencia de información dentro de una base de datos relacional. *Hibernate* es un *Framework* de Mapeo Relacional de Objetos (ORM). Este *Framework*

reduce drásticamente el tiempo en que los desarrolladores codifican las líneas necesarias para implementar los componentes de persistencia, ya que este *Framework* ,entre otras cosas, realiza la persistencia de información mediante el uso de archivos de mapeo, que permiten realizar la transformación de la información de objetos a entidades [14] [15]. Esta característica de *Hibernate* es representada en la Figura 11, la cual muestra la transformación que realiza este *Framework* de una tabla relacional hacia un objeto de entidad, el cual contiene la información almacenada en un renglón de la tabla dentro de sus propiedades.

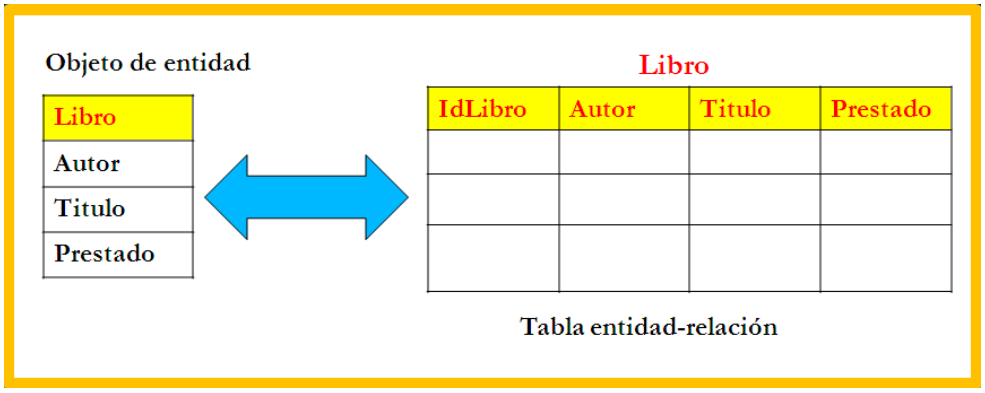

Figura 11. Ejemplo de mapeo de objetos de entidad hacia tablas entidad-relación.

Hibernate también genera de manera automática los mecanismos necesarios para realizar la persistencia de información de los objetos de entidad (consultas como save, update y delete) de acuerdo a los archivos de mapeo especificados, liberando al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias [14] [15] [17]. La Figura 12 muestra la definición un archivo de mapeo de objeto a entidad relación. En este ejemplo se puede observar cómo se realiza la persistencia especificando que objeto Java será mapeado a una tabla específica, así como también la relación entre propiedades del objeto y columnas de la tabla.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
            "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
        <class name="mx.com.uam.proto.dominio.Libro" table="Libro">
                <!-- Identificador unico de la clase -->
                <id name="idLibro" column="idLibro" type="int">
                        <!-- El valor de increment siginifica que el valor inicial
                es 1 y se incrementa en 1 cada vez que se agrega un registro -->
                        <generator class="increment"/>
                </id>
                <!-- Nombre del autor -->
                <property name="autor" type="string"/>
                <!-- Nombre del libro -->
                <property name="titulo" type="string"/>
                <!-- Bandera para saber si esta prestado o no -->
                <property name="prestado" type="boolean"/>
        </class>
</hibernate-mapping>
```

Figura 12. Manejo de persistencia en Spring [14].

2.2.4 Framework Acegi.

Acegi es una librería de código abierto que proporciona una capa de seguridad a aplicaciones desarrolladas con *Spring*. Esta librería permite integrar mecanismos que se encargan de configurar aspectos de las tácticas de seguridad [20]. Para realizar este control es necesario definir filtros dentro de un archivo XML nombrado web.xml, estos filtros agregan capas de seguridad a la aplicación. Algunos de los filtros de seguridad que pueden ser agregados son los siguientes:

- **Filtro de integración.** Este filtro se encarga de recordar la autentificación de un usuario entre solicitudes de este. Al final de cada solicitud, este filtro almacena la información de autentificación de un usuario dentro de una sesión para que esta esté disponible para siguientes solicitudes.

- **Filtro de proceso de autentificación.** Filtro que se encarga de identificar solicitudes de autentificación. Recupera información de usuario de la solicitud para poder determinar si el usuario es identificado por el sistema.

- **Filtro de interpretación de excepciones.** Este filtro se encarga de enviar las respuestas adecuadas a las posibles excepciones que el sistema de seguridad puede mandar. Las excepciones que este sistema puede lanzar son referentes a solicitar la autentificación de un usuario o denegarle el acceso a los recursos.

- **Filtro interceptor de seguridad.** Este filtro se encarga de examinar las solicitudes y determinar si el usuario tiene los privilegios necesarios para acceder a los recursos.

Los filtros descritos anteriormente son solo algunos de filtros que pueden ser definidos para el manejo de seguridad de una aplicación, para conocer la lista completa de filtros que *Acegi* permite implementar en la seguridad de una aplicación,

véase la última versión del manual de referencia de *Acegi*. Como se mencionó anteriormente, los filtros pueden ser vistos como capas de seguridad, en donde las solicitudes de los usuarios tienen que pasar por todas y cada una de ellas para poder acceder al recurso de la aplicación solicitado. La Figura 13 representa el flujo de una solicitud a través de estos filtros [20] [24].

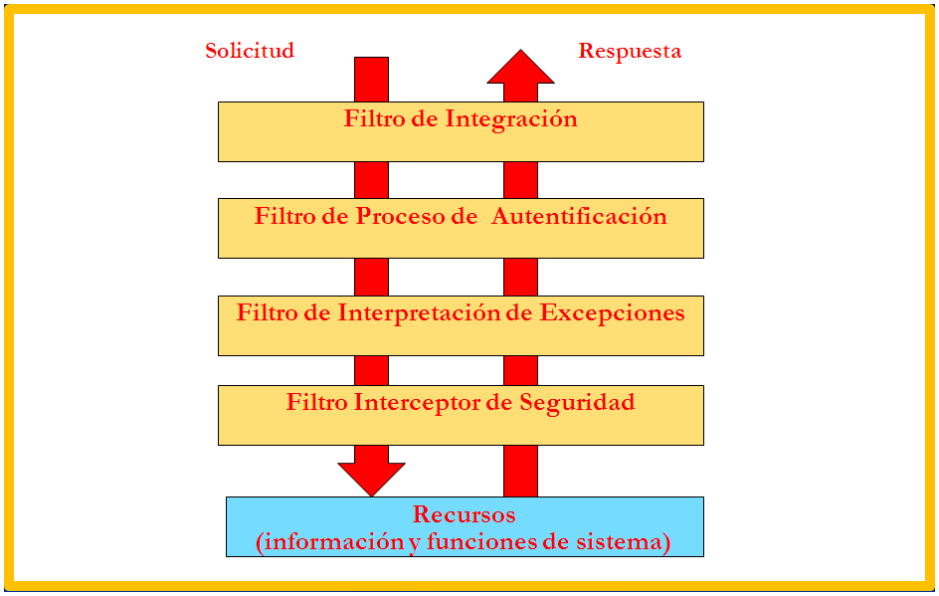

Figura 13. Manejo de seguridad por Acegi [24].

Sin embargo, los filtros no se encargan directamente del manejo de aspectos de seguridad, esta función es delegada de los filtros hacia manejadores que se encargan de manejar dichos aspectos. Las entidades que implementan las tácticas de los aspectos de seguridad son las implementaciones específicas de los manejadores, estas entidades son conocidas como proveedores. Es decir, los proveedores son las entidades que se encargan de realizar los procesos necesarios para cubrir con cada una de las tácticas de seguridad de una manera determinada. Tanto los manejadores y proveedores de aspectos de seguridad de *Acegi* son declarados y configurados en el *application context*, cada una de estos aspectos son definidos en beans de *Spring* (componentes de software con funcionalidades variadas, tales como transportar información entre tercios o manejo de funcionalidad acorde con el negocio.).

2.2.4.1 Principios y tácticas empleadas en el Framework Spring.

Principalmente el manejo de tácticas de atributos de calidad que realiza *Spring* es utilizando un archivo de configuración del contenedor ligero, este archivo es el *application context* (archivo XML), y algunos otros *Frameworks* que operan en conjunto con *Spring*. Las siguientes tablas presentan un subconjunto de las tácticas que pueden ser configuradas por los componentes de *Spring*, técnicas de programación y otros marcos de trabajo que trabajan en conjunto con *Spring*.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Detección de fallas	Manejo de excepciones	Es soportado por el lenguaje de programación Java.
Recuperación de fallas	Checkpoint/Rollback	Esta táctica es soportada utilizando intermediarios de *Frameworks* de desarrollo específicos como es el caso de *Hibernate*.
Prevención de Fallas	Transacciones	Esta táctica es soportada utilizando el *Framework* de desarrollo *Hibernate* en conjunto con el contenedor de *Spring*. Los niveles de transacción para los componentes de negocio son especificados en el *application context*.

Tabla 13. Tácticas de disponibilidad en Spring.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Localización de modificaciones	• Mantener coherencia semántica. • Anticipar cambios esperados. • Generalización de módulos.	Estas tácticas son soportadas por el lengua de programación Java (programación orientada a objetos) y por el uso de interfaces. Estas tácticas son configuradas en el contenedor con IoC.
Prevención de efecto de onda	• Encapsulamiento. • Conservación de interfaces existentes. • Uso de intermediarios.	Estas tácticas so soportadas por técnicas de programación del lenguaje y patrones de diseño. Son configuradas en el contenedor y hacen uso de IoC para ser implementadas.
Retraso en tiempo de ligado	• Registro en tiempo de ejecución. • Archivos de configuración. • Polimorfismo.	Tácticas soportadas por técnicas del lenguaje de programación, archivos de configuración tales como el *application context*, así como también por aspectos de IoC que se encarga del ligado de componentes en tiempo de ejecución.

Tabla 14. Tácticas de modificabilidad en Spring.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Demanda de recursos	• Reducir sobrecarga de cómputo. • Administración de eventos. • Limitar el tamaño de las colas. • Limitar el tiempo de ejecución.	Esta táctica es soportada por el contenedor ligero de *Spring* así como por *Frameworks* específicos, tales como *Hibernate*, pues en la configuración de este *Framework*, dentro del contenedor, se puede especificar el manejo de colas y tiempo de ejecución de las transacciones, así como también mediante el uso de AOP se puede especificar el manejo de eventos y carga de computo de la aplicación.
Administración de recursos	Introducción de concurrencia.	Esta táctica es soportada configurando la concurrencia en diferentes niveles de ejecución por *Frameworks* de desarrollo específicos que trabajan en conjunto con *Spring*. Algunos de estos Framworks son: *Acegi* (manejo de sesiones) y *Hibernate* (acceso a datos concurrentes). También es posible especificar el acceso a dispositivos u otros medios utilizando AOP.

Tabla 15. Tácticas de desempeño en Spring.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Resistencia a los ataques	• Autentificación. • Autorización. • Mantenibilidad de confidencialidad de datos e integridad.	Estas tácticas son soportadas por el *Framework* de desarrollo de *Acegi* y *Spring*. Las propiedades para soportar estas características son configuradas en archivos de filtros (*Acegi*) y en el *application context* (*Spring*).
Recuperación de ataques	Restauración de estado.	Estas tácticas son soportadas por el *Framework Hibernate* en conjunto con *Spring*.

Tabla 16. Tácticas de seguridad en Spring.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Proveer entradas y captura de salidas	• Separación de interfaz e implementación. • Interfaces de prueba especializadas.	Son tácticas soportadas por técnicas de programación así como por características de IoC.

Tabla 17. Tácticas de facilidad de pruebas en Spring.

Categoría de la táctica	Táctica	Elementos que soportan dicha táctica.
Soporte intuitivo de sistema	Tácticas de tiempo de diseño.	Soportado por patrones de diseño como MVC.

Tabla 18. Tácticas de usabilidad en los Spring.

Las tablas anteriores muestran un conjunto de las tácticas que son soportadas por los *Frameworks* de desarrollo *Hibernate* y *Acegi* en conjunto con *Spring*. De estas tablas se puede observar que la mayoría de las tácticas de atributos de calidad son soportadas utilizando los conceptos de IoC y AOP, las cuales son implementadas de forma declarativa en el *application context*. La IoC proporciona a las aplicaciones un manejo y configuración de los componentes de software por una entidad independiente (el contenedor), beneficiando así el uso de tácticas como modificabilidad y facilidad de uso de pruebas. En el caso de AOP, permite realizar la administración de servicios (atributos de calidad) ya sea utilizando una implementación definida por el desarrollador o en otro caso utilizando uno de los tantos *Framework* de desarrollo especializados en el manejo de ciertos servicios (para este proyecto se opto por utilizar *Hibernate* y *Acegi*). Retomando un de los objetivos principales de este proyecto de investigación, el cual es definir un proceso que pueda ser empleado para modelar tácticas de calidad sobre otras plataformas, al realizar la selección de tácticas es posible cambiar la selección de los *Framework* que trabajan en conjunto con él y seguir el mismo proceso para realizar el manejo de servicios. Debido que *Spring* permite implementar características como IoC y AOP de manera declarativa, *Spring* (junto con *Hibernate* y *Acegi*) fue elegido como *Framework* de desarrollo para la implementación de este proyecto de investigación.

Hasta ahora solo se ha mencionado la importancia de la arquitectura de software durante el proceso de desarrollo, y como son implementados los atributos de calidad por el *Framework* de desarrollo durante la fase de construcción. Uno de los problemas que se identificaron y se han presentado en esta tesis hasta ahora es el problema de modelado o representación de las tácticas de atributos de calidad en el diseño y documentación de la arquitectura. Por tal motivo y como uno de los objetivos de este proyecto, el modelar tácticas de atributos de calidad, en la siguiente sección se presentan los conceptos fundamentales de un método de desarrollo que se enfoca en facilitar el desarrollo a través de la transformación de modelos.

2.3 Arquitectura Dirigida por Modelos (MDA).

L a Arquitectura Dirigida por Modelos (MDA) es una propuesta elaborada por OMG (*Object Management Group*). MDA se enfoca en la utilización de modelos, normalmente utilizados como documentación en el desarrollo, para especificar las características de una solución a un proceso de software y mediante la transformación de modelos obtener una solución final (el sistema) [21].

2.3.1 Visión general.

La utilización de modelos para entender un problema y proporcionar una solución es una de las estrategias utilizadas en el proceso de desarrollo. En el enfoque MDA cada uno de los modelos definidos puede ser visto como un artefacto generado en una etapa de desarrollo (análisis, diseño, codificación y pruebas), pero utilizando el enfoque MDA estos modelos son generados con ayuda de mecanismos que garanticen la consistencia con los modelos anteriores, esto permite reducir el re trabajo realizado si los modelos fueran creados manualmente. Los modelos definidos por el desarrollo MDA son [27]:

- **Modelo Independiente de Computación (CIM).** Representa los modelos que caracterizan el dominio del problema. Este tipo de modelos definen características de negocio donde se define la funcionalidad de un sistema en particular.

- **Modelo Independiente de Plataforma (PIM).** Consiste en la representación de los modelos que describen una solución del problema desapegada de la tecnología. La descripción de estos modelos es resultado del análisis de las necesidades del sistema

- **Modelo Específico de Plataforma (PSM).** Consiste en la representación de los modelos resultantes del diseño de la aplicación sobre una plataforma específica.

- **Código.** Son los artefactos generados después de las etapas de codificación y pruebas.

La Figura 14 muestra como los modelos definidos por MDA son resultado de actividades realizadas en las distintas etapas de desarrollo de software.

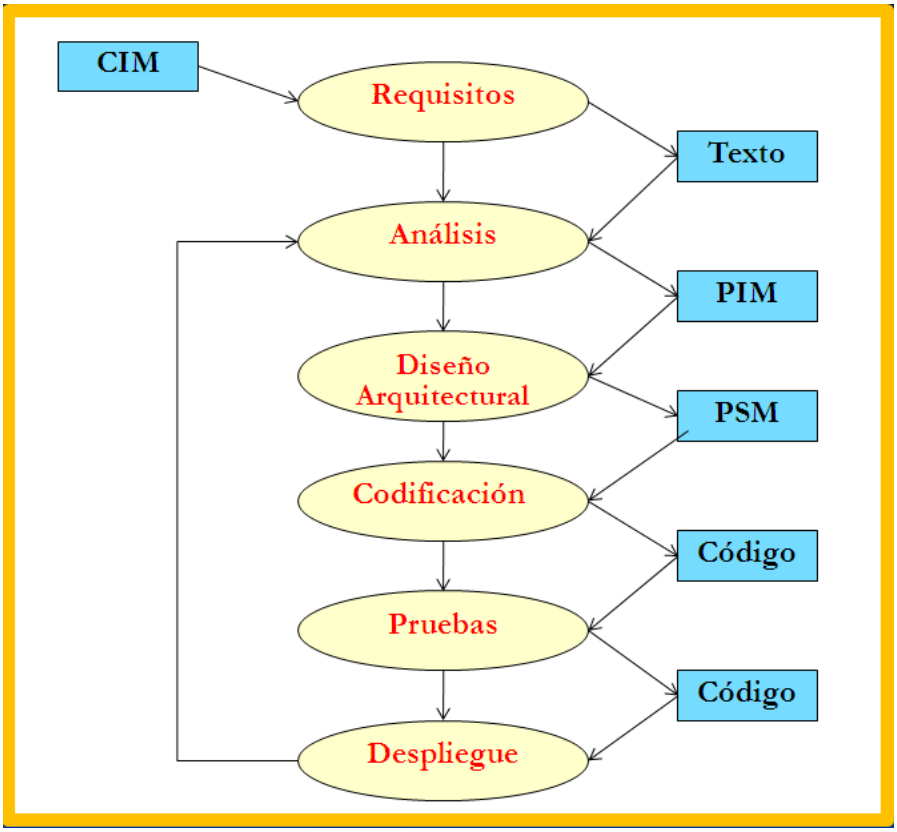

Figura 14. Proceso de desarrollo con modelos MDA [27].

Como se muestra en la figura anterior el proceso de desarrollo de software puede ser visto como una transformación de modelos hasta obtener una solución (código). Justamente en uno de los modelos manejados durante el desarrollo de

software y en el desarrollo con MDA, es posible establecer los atributos de calidad que la aplicación soportará; este modelo es el PSM y de acuerdo a la selección de plataforma que se realiza en este modelo son las tácticas de atributos de calidad que pueden ser configuradas para la aplicación. Es decir, la selección de tecnología establece las tácticas de calidad que pueden ser configuradas para el sistema, así como también la manera en que serán implementadas por dicha tecnología.

Debido a que el PSM es un modelo específico de plataforma (selección de tecnologías de software y hardware empleadas para el desarrollo de software) y que justamente la selección de la plataforma influye directamente en las posibles tácticas que puedan ser empleadas en el desarrollo de una aplicación en específico, es que este modelo es tan importante, ya que en este es posible definir el soporte de tácticas de atributos de calidad. En la siguiente sección se describen los conceptos importantes de este modelo específico de plataforma.

2.3.2 Modelo Específico de Plataforma (PSM).

Las partes de un modelo PSM deben permitir representar los elementos, características y funcionalidad de los componentes del sistema de acuerdo a la plataforma a la que pertenecen. Principalmente los elementos que pueden ser modelados en un PSM son:

- **Componentes de software.** Estos elementos de modelado representan los componentes de software específicos de tecnologías utilizadas para implementar alguna funcionalidad del sistema.

- **Relaciones entre componentes.** Las relaciones son los elementos de modelado que representan la interacción entre componentes de software y como se comunican entre ellos para realizar alguna funcionalidad de la aplicación.

- **Propiedades de componentes.** Son los elementos de modelado que permiten especificar características y comportamiento de los componentes. Con respecto al comportamiento del sistema (la calidad), este puede ser especificado mediante propiedades de varios componentes de software.

El modelo específico de plataforma permite el modelado de componentes y por lo tanto es muy parecido al diagrama de componentes de UML, sin embargo, considerando que la selección tecnológica impone las tácticas de atributos de calidad es necesario modelar componentes de software de la selección tecnológica realizada. Para poder especificar las tácticas que podrán ser modeladas en un determinado PSM es necesario poder modelar componentes y propiedades particulares, debido principalmente a que este modelo es específico de plataforma. Es por esto que algunas veces UML estándar no es suficiente para modelar un PSM y por ello es necesario definir un perfil UML. Este perfil UML debe estar adaptado a la selección de plataforma que se requiera acorde con el dominio del problema.

2.3.3 Perfil UML.

Un perfil UML permite definir lenguajes de modelado derivados de UML, utilizando mecanismos de extensión que permitan adaptar los elementos de un meta-modelo al dominio de negocio del software. Los mecanismos de extensión utilizados en un perfil UML son [27]:

- **Estereotipos (Stereotypes).** Es un mecanismo que permite representar los distintos tipos de elementos que pueden ser agregados a un modelo. Estos mecanismos son utilizados para representar nuevos y distintos tipos de componentes o elementos de software para nuevas plataformas.

- **Valores etiquetados (Tagged Value).** Estos mecanismos de extensión permiten agregar propiedades a las especificaciones de los elementos o componentes a modelar.

- **Restricciones (Constraint).** Son los mecanismos que permiten precisar la especificación de los modelos y sus elementos. Es decir, como y bajo qué condiciones pueden modelarse los componentes y sus relaciones dentro de los modelos.

Después de que un modelo PSM ha sido especificado con ayuda de un perfil UML, el enfoque MDA propone la generación de código a partir de este modelo. Para realizar esto es necesario definir las reglas de transformación para la generación de archivos de código y configuración. En la siguiente sección se describen el concepto de transformación hacia código.

2.3.4 Transformación a código.

Una de las transformaciones que propone el enfoque MDA es la generación de código a partir de un modelo específico de plataforma, este proceso de generación de código es realizado utilizando herramientas de apoyo que permiten traducir componentes del modelo a archivos con código específico de la plataforma, en capítulos posteriores se describirán algunas de las herramientas utilizadas para la transformación a código. Para lograr la transformación a código, como proceso general se deben definir reglas textuales que establezcan aquellos archivos que deben ser generados a partir de uno o más componentes. De manera general, se deben trabajar sobre los siguientes elementos para la definición de las reglas de transformación:

- **Identificación de archivos por componentes de software.** Se realiza un análisis de la correspondencia entre componentes de software modelados en el PSM y los archivos de código fuente que estos generaran.

- **Identificación de relaciones.** Se define como serán traducidas todas las relaciones existentes dentro del PSM para crear las dependencias entre objetos y el manejo de la comunicación entre estos.

- **Identificación de propiedades de componentes.** Se identifican propiedades de los componentes que permiten establecer las propiedades de uno o más archivos de código y configuración.

Una vez identificado la manera en que cada uno de los elementos del modelo influye en la generación de código de una plataforma específica, se deben representar las reglas de manera textual de tal manera que cualquier persona involucrada en el desarrollo pueda entenderlas. Una vez que las reglas de transformación son definidas, se deben adaptar estas reglas de transformación para que herramientas MDA especializadas se encarguen de la generación de código.

3 Propuesta Teórica.

En el capítulo anterior se describieron todos los conceptos necesarios para el planteamiento y comprensión del proyecto. Dentro de este capítulo se explicarán las restricciones y bases sobre las cuales está desarrollada la propuesta de éste proyecto de investigación. El objetivo principal de este proyecto es poder modelar tácticas de atributos de calidad en componentes de software de un modelo arquitectónico específico de una plataforma (PSM) y a partir de este modelo, con ayuda de una herramienta de transformación de modelos, poder generar una implementación parcial del sistema (esqueleto de arquitectura ejecutable) de manera automática y así reducir el tiempo en el desarrollo de software durante las fases de elaboración y construcción. En el desarrollo de este proyecto se utilizará un enfoque MDA, específicamente este proyecto toma como punto de partida el modelo PSM generado en una de las transformaciones propuestas por este enfoque. Sin embargo para poder considerar un modelo específico de plataforma es necesario realizar la selección del patrón arquitectónico que será utilizado para comenzar la construcción de la aplicación, así como también es necesario realizar la selección de la plataforma que será utilizada durante la construcción del sistema. Estos dos puntos a considerar son planteados como restricciones de la aplicación (RNF), pues limitan o restringen las decisiones que deben ser tomadas para comenzar la construcción del sistema.

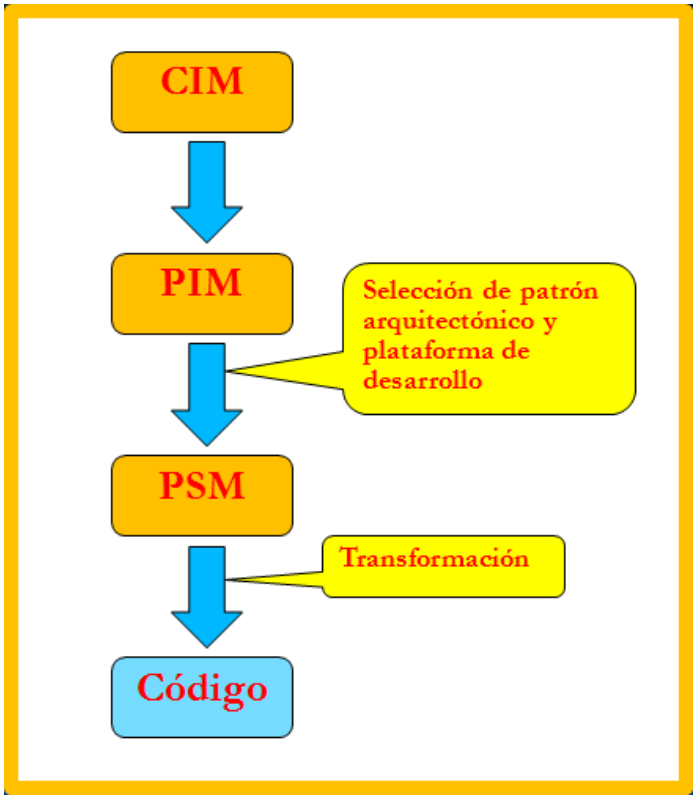

Figura 15. Contexto del Proyecto dentro de la Metodología MDA.

La Figura 15 muestra los modelos de transformación de la metodología MDA, haciendo énfasis en el paso del modelo PIM a PSM y de PSM a Código. Este proyecto toma como base la transformación de PIM a PSM, ya que dentro de esta transformación se definen la plataforma y patrones de diseño que serán utilizados para el desarrollo del sistema. Ya que este proyecto está enfocado en modelar arquitecturas de aplicaciones empresariales genéricas, accesibles vía web; los patrones de diseño que son considerados en este proyecto son los más representativos y utilizados para el desarrollo de sistemas de este tipo, estos patrones serán descritos más adelante. Tomando como base lo anterior el contexto de este proyecto de investigación se enfoca en definir un proceso de construcción de un modelo especifico de plataforma y la transformación a código a partir de ese modelo.

3.1 Proceso.

La especificación detallada del proceso definido para el modelado de tácticas de atributos de calidad y generación de una herramienta que permita automatizar la construcción de esqueletos de arquitecturas ejecutables es descrito en la Figura 16.

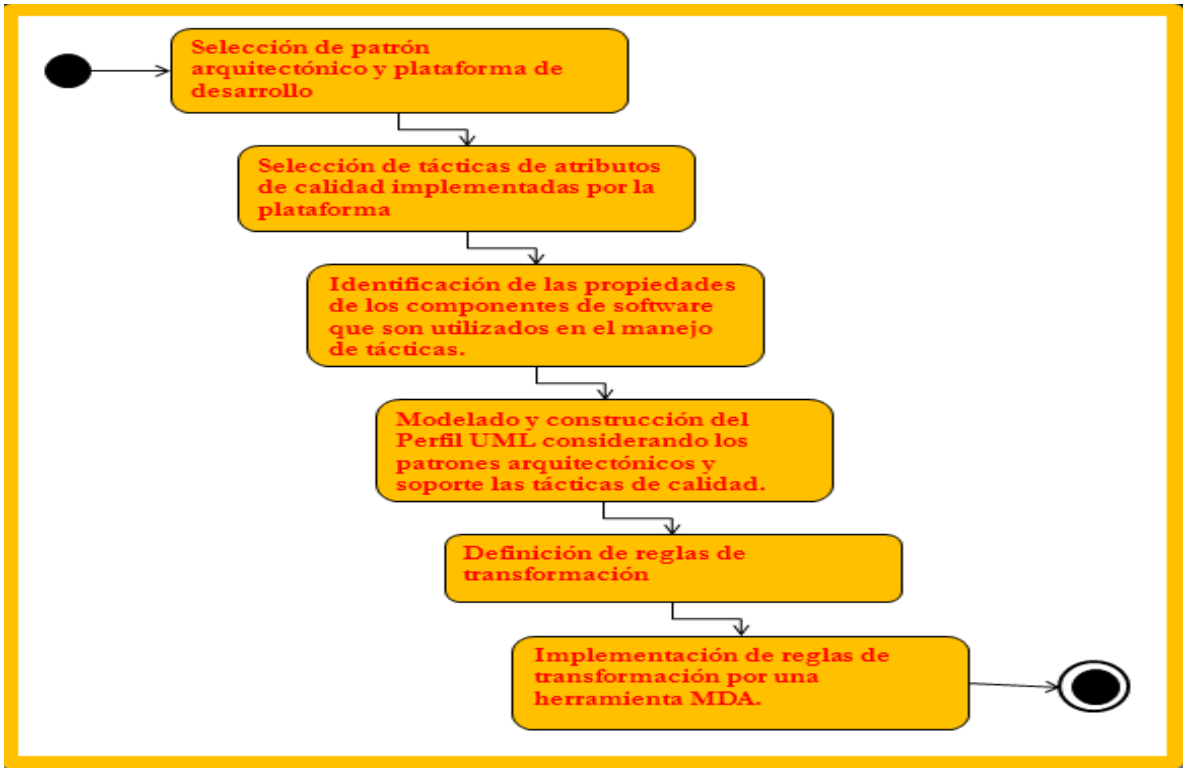

Figura 16.Diagrama de actividades del proceso empleado en este proyecto.

La primera actividad a realizar en el proceso definido, es la selección del conjunto de patrones arquitectónicos y la plataforma de desarrollo que será utilizada para en el diseño y construcción del sistema. Como segunda actividad se realiza la identificación y selección de tácticas que son soportadas por la plataforma seleccionada. La tercera actividad consiste en realiza la identificación de las propiedades de los componentes de software que son utilizadas para la

configuración de las tácticas de atributos de calidad. En la cuarta actividad se selecciona la herramienta que permitirá realizar la construcción y modelado del PSM para el sistema a desarrollar. En la quinta actividad se realiza la definición de reglas de transformación, en las cuales se identifica los componentes que serán utilizados para configurar las tácticas y de los cuales se generarán los archivos de código fuente con la especificación de soporte de tácticas. Por último, se realiza la selección de la herramienta de generación de código que será utilizada para implementar las reglas de transformación anteriormente identificadas. El proceso detallado se describe a lo largo de este capítulo.

3.2 Restricciones en el desarrollo del proyecto.

A continuación se presentan las consideraciones relacionadas a las restricciones dentro del contexto de este proyecto.

3.2.1 Patrones seleccionados.

Los patrones arquitectónicos que se han adoptado como base del modelo de arquitectura son los algunos de los más utilizados en el desarrollo de aplicaciones empresariales que generalmente son accedidas vía web. Específicamente los patrones de diseño utilizados para el modelado de la arquitectura son los siguientes:

- **Cliente-Servidor.** Este patrón es adoptado para el modelo pues el tipo de aplicación que se pretende utilizar en el modelado es una aplicación web.

- **Tres-tercios.** Este patrón es adoptado en la parte del servidor; principalmente el modelo que se adoptará representará la estructura completa del servidor y contendrá todos los componentes de software que se modelarán, por lo tanto también contendrá los elementos necesarios para realizar el soporte de tácticas de atributos de calidad.

- **Capa de Acceso a la Base de Datos.** Este patrón introduce una capa de acceso a la base datos entre los componentes de la aplicación y la base de datos relacional. Esta capa desacopla una aplicación orientada a objetos de los detalles de la base datos. Este patrón estará implementado dentro del tercio de datos (persistencia) [37].

- **MVC.** La implementación de este patrón, también puede ser considerada como respuesta a tácticas de diseño dentro de la categoría de usabilidad y modificabilidad de los atributos de calidad que propone el SEI. Este patrón desacopla el modelo del dominio del problema de la interfaz, así como también el comportamiento de la interfaz de esta mismo mediante el uso de un controlador. Este patrón se implementa mediante el uso de componentes en los tercios de presentación y negocio [33].

3.2.2 Plataforma seleccionada.

Una vez que se ha definido las grandes estructuras en las cuales se organizarán los componentes del modelo que nos permitirá representar un sistema, es necesario seleccionar la plataforma o tecnología que será utilizada para el desarrollo

de los componentes de software en cada una de las estructuras. De los dos *Frameworks* mencionados anteriormente, EJB y *Spring*, se seleccionó el *Framework* de desarrollo Spring por sus características, tales como el manejo de aspectos de manera independiente de la funcionalidad (AOP), la administración de los recursos por una entidad independiente de los componentes de software (IoC), así como también que Spring permite realizar el manejo de la parte no funcional de manera declarativa, ya que esto facilita la transformación del PSM a código fuente. Sin embargo, siguiendo el proceso definido en esta tesis es posible cambiar la plataforma que implementa el manejo de características no funcionales de manera programática.

3.2.3 Tácticas que serán modeladas para la plataforma seleccionada.

Las tácticas que se presentan en las siguientes secciones son un sub conjunto de las tácticas soportadas por la plataforma seleccionada (Spring), por cuestiones de tiempo únicamente se seleccionó e implementó un sub conjunto de tácticas. Estas tácticas pueden ser implementadas en el desarrollo por elementos del *Framework Spring* o por elementos específicos de los *Frameworks* especializados para el manejo de algún servicio o aspecto del desarrollo. *Acegi* y *Hibernate* son los *Frameworks* que serán utilizados para poder soportar atributos de calidad tales como seguridad, desempeño y disponibilidad. En la Tabla 19 se describe el sub conjunto de tácticas y componentes de soporte de los *Frameworks* que serán considerados para poder modelar las tácticas en la representación de arquitectura que se ha estado definiendo en las secciones anteriores (PSM).

Atributo de Calidad	Táctica específica	Framework y componentes de soporte.
Disponibilidad	• Prevención de Fallas: Transacciones (Manejo Transaccional). • Recuperación de Fallas: Checkpoint/rollback.	Tácticas soportadas y configuradas por *Hibernate* y características AOP en *application context* de Spring.
Modificabilidad	• Localización de modificaciones: mantener coherencia semántica. • Prevención de efecto de onda: encapsulamiento, conservación de interfaces existentes y uso de intermediarios. • Retraso en tiempo de ligado: polimorfismo.	Tácticas soportadas y configuradas con IoC de Spring dentro de los componentes y el *application context*
Desempeño	Demanda de recursos: limitar el tiempo de ejecución y el tamaño de las colas.	Tácticas soportadas y configuradas por *Hibernate* en el *application context* de Spring
Seguridad	Resistencia a los ataques: autentificación, autorización, mantenimiento de confidenciabilidad de datos.	Táctica soportada y configuradas en archivos de control de *Acegi* (filtros) y dentro del *application context* de *Spring*.
Facilidad de pruebas	Proveer entradas y capturar salidas: separación de interfaz e implementación.	Táctica soportada con el uso de IoC en componentes y declaración de dependencias en *application context*
Usabilidad	Soporte intuitivo de sistema: tácticas de tiempo de diseño (separación de componentes por responsabilidades).	Táctica soportada con el uso de IoC en la separación de componentes por responsabilidades (uso del patrón MVC).

Tabla 19. Sub conjunto de tácticas y Frameworks que las soportan.

La Tabla 19 muestra un subconjunto de tácticas soportadas por los *Framework* seleccionados y que serán consideradas para realizar el modelado del sistema en el PSM. La selección de tácticas a modelar se baso principalmente en el tiempo que se dispone para el proyecto de investigación, considerando que algunas de las tácticas tales como administración de recursos requieren un mejor manejo de los *Frameworks*. También se busco modelar tácticas de cada categoría de atributos de calidad que permitieran utilizar los conceptos de IoC y AOP. En las siguientes sub secciones se describen a detalle los elementos o propiedades, de los *Frameworks* de desarrollo, que permiten manejar las tácticas de calidad de la tabla anterior.

3.2.3.1 Manejo de tácticas de atributos de calidad en Spring.

En las siguientes secciones se describen los conceptos necesarios de elementos o propiedades sobre los cuales se pueden realizar configuraciones que permitan el soporte de atributos de calidad.

3.2.3.1.1 Manejo de tácticas de disponibilidad.

3.2.3.1.1.1 Manejo transaccional.

El manejo de tácticas de disponibilidad, específicamente la prevención de fallas mediante el manejo de transacciones de manera declarativa, es realizado por el contenedor ligero y configurado en el *application context*. Este contendor implementa características de AOP las cuales separan el manejo de transacciones de los demás componentes. Dentro del archivo de configuración del contenedor (*application context*) se especifican las propiedades de manejo transaccional para cada uno de los componentes del sistema que requieran utilizar este tipo de servicio. Estas propiedades son: nivel de propagación y nivel de aislamiento.

Nivel de propagación.

La propiedad de nivel de propagación controla el ciclo de vida de una transacción de un componente. Es decir, el momento de creación, propagación y límite de existencia. Los posibles valores para esta propiedad son [12][13][14]:

- **Required.** El contenedor fuerza la ejecución de los *beans* (componentes de negocio) con este valor dentro de una transacción. Si una transacción ya está ejecutándose, el *bean* se une a esa transacción. Si no se encuentra ejecutándose una transacción el contenedor puede comenzar una transacción para ejecutar el *bean*.

- **RequiresNew.** El contenedor crea una nueva transacción cuando un *bean* con este valor es llamado. Si una transacción ya se encuentra en ejecución cuando el *bean* es llamado, esa transacción es suspendida durante la invocación del *bean*.

- **Supports.** Con este atributo el contendor puede ejecutar el *bean*, exista o no una transacción ejecutándose. Es decir, los componentes con este valor pueden ser ejecutados dentro o fuera de una transacción sin problema alguno.

- **Mandatory.** Obliga a que una transacción deba estar ya corriendo cuando el *bean* es llamado. Garantiza que el *bean* deba ser ejecutado en una transacción.

- **NotSupported.** El *bean* no podrá ser invocado en una transacción. Ejemplo de estos son los *beans* que ejecutan operaciones no críticas.

- **Never.** Significa que el *bean* no puede ser invocado en una transacción. Además, si el cliente llama al *bean* en una transacción, el contenedor lanza una excepción de regreso al cliente.

Nivel de Aislamiento.

La propiedad de nivel de aislamiento especifica el comportamiento de una transacción dentro de una ejecución concurrente de varias transacciones. Es decir, de qué manera impacta una transacción a otra. Este nivel de aislamiento puede ser declarado en clases completas o únicamente en métodos específicos de las mismas. Dependiendo del manejador de base de datos que se esté utilizando son los niveles de aislamiento que pueden ser utilizados [24]. Algunos de los niveles utilizados por la mayoría de los manejadores de base de datos son:

- **Read Uncommited.** Los componentes definidos con este valor pueden leer cambios de transacciones en progreso y que no han sido finalizadas.
- **Read Commited.** El contenedor permite a los componentes con este valor leer los cambios efectuados por transacciones concurrentes finalizadas.
- **Repeatable Read.** Múltiples lecturas del mismo campo pueden producir el mismo resultado, a menos que la misma transacción realice el cambio. Sin embargo se pueden presentar lecturas fantasmas (lecturas de datos que nunca fueron finalizadas).
- **Serializable.** Las transacciones definidas con este valor pueden operar con los campos mientras no existan otras transacciones concurrentes que estén accediendo a ellos. Este es el nivel de aislamiento máximo, pero con el menor desempeño.

La Figura 17 muestra la configuración del manejo de transacción de un *bean* (usuarioDAO), el cual tiene configurado un manejo transaccional con nivel de propagación con valor *PROPAGATION_REQUIRED* sobre los métodos que contengan el prefijo *save* y *delete*, así como un nivel de aislamiento de *ISOLATION_REPETEABLE_READ* e *ISOLATION_SERIALIZABLE*, respectivamente [18] [19].

```
<bean id="usuarioDAO"
    clas="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name"transactionManager"> <ref local="transactionManager"/></property>
        <property name"target"> <ref local="usuarioDAOTarget"/></property>
        <property name"transactionAttributes">
            <!-- Definición de las propiedades de Manejo transaccional sobre los métodos de la clase -->
            <props>
                <prop key="save*">PROPAGACION_REQUIRED, ISOLATION_REPEATABLE_READ</prop>
                <prop key="delete*">PROPAGACION_REQUIRED, ISOLATION_SERIALIZABLE</prop>
            </prop>
        </property>
        <property name="preInterceptors"><ref local="interceptor" /> </property>
</bean>
```

Figura 17. Especificación de manejo de transacciones en Spring dentro del application context [18].

3.2.3.1.1.2 Checkpoint/rollback.

La otra táctica de disponibilidad que será implementada en esta tesis es la recuperación de fallas mediante el uso únicamente de *rollback*, esta táctica es implementada por el manejador de transacciones de *Hibernate* y definida dentro del *application context*. Por default, un *rollback* se ejecuta en una transacción únicamente si se presenta una excepción en tiempo de ejecución y la manera en que el *rollback* es implementado está definida por el manejador de transacciones seleccionado (*Hibernate*). La Figura 18 muestra la definición del manejador de transacciones de *Hibernate*.

```
<bean id="transactionManager" clas="org.springframework.
    orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Figura 18. Definición del Manejador de Transacciones de Hibernate [18].

3.2.3.1.2 Manejo de tácticas de modificabilidad.

La mayoría de las tácticas de modificabilidad son soportadas por técnicas o conceptos de programación (separación de interfaz e implementación, encapsulamiento, polimorfismo, etc.), uso de intermediarios (patrón de fábrica, *proxys*). *Spring* permite implementar tácticas de modificabilidad mediante el uso de Inversión de control (IoC), esto significa que el contenedor ligero se encarga de conectar (crear dependencias) a los componentes. A su vez *Spring* también permite la definición de intermediaros para el manejo de servicios en específicos, estos intermediarios pueden ser obtenidos de otros *Frameworks*. La IoC y la utilización de intermediarios son algunos de las características de *Spring* que permiten la sustitución o modificación sencilla de un componente en específico sin la necesidad de realizar cambios en varios componentes que estén relacionados, esto permite minimizar el impacto o efecto de onda de los cambios [17].

3.2.3.1.3 Manejo de tácticas de desempeño.

La táctica de desempeño que fue identificada en el desarrollo utilizando *Spring* es la demanda de recursos, específicamente limitar el tiempo de ejecución y el tamaño de las colas que son utilizadas para realiza el acceso a la información almacenada (persistencia de información). La definición de los componentes que permiten acceder a esa información está relacionada con la configuración necesaria para utilizar un manejador de base de datos. En *Spring* la configuración del manejador de bases de datos se especifica dentro del *application context* en el apartado de propiedades de del *Framework* de persistencia [14][15].

El desempeño dentro de la capa de persistencia puede ser controlado por el *Framework* de *Hibernate*, y es configurado dentro del *bean* de *hibernateProperties* del *application context*. Dentro de las propiedades especificadas de este *bean* se encuentran propiedades relacionadas al comportamiento de la aplicación, es decir la manera en que se realizará el acceso a los datos. Dos de las propiedades que permiten especificar el desempeño son *timeout* (tiempo de espera para la

ejecución de un transacción) y *maxPoolSize* (número máximo de transacciones que pueden ejecutarse de manera concurrente). La Figura 19 muestra la definición del bean de *hibernateProperties*.

```
<bean id="hibernateProperties" class="org.springframework.beans.factory.config.PropertiesFactoryBean">
    <property name="properties">
        <props>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
            <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
            <prop key="hibernate.query.substitutions">true 'T', false 'F'</prop>
            <prop key="hibernate.show_sql">false</prop>
            <prop key="hibernate.c3p0.minPoolSize">5</prop>
            <prop key="hibernate.c3p0.maxPoolSize">60</prop>
            <prop key="hibernate.c3p0.timeout">600</prop>
            <prop key="hibernate.c3p0.max_statement">50</prop>
            <prop key="hibernate.c3p0.testConnectionOnCheckout">false</prop>
        </props>
    </property>
</bean>
```

Figura 19. Definición de características de desempeño en Data Source.

3.2.3.1.4 Manejo de tácticas de seguridad.

La categoría de tácticas de seguridad que serán implementadas en este proyecto son referentes a la resistencia a ataques; específicamente tácticas como autentificación, autorización, mantenimiento de confidencialidad e integridad de datos o recursos, tales como información o funciones del sistema. Este tipo de tácticas son declarados dentro del *application context* y dentro de archivos de filtrado del *Framework Acegi* (*Framework* que se agrega a *Spring* y que permite controlar aspectos de seguridad).

El manejo de autentificación es definido en un *bean* para toda la aplicación, sin embargo, el manejo de autorización para el acceso a recursos, puede ser definido con el uso de un auto proxy (proxy creado por Spring para el manejo de aspectos) que se encarga de corroborar los datos de autentificación y autorización del usuario para cada uno de los servicios o funcionalidades del sistema. Es decir este interceptor de seguridad corrobora los servicios de la aplicación a los cuales el usuario tiene acceso.

3.2.3.1.5 Manejo de tácticas de facilidad de pruebas.

Al igual que las tácticas de modificabilidad, las tácticas de facilidad de pruebas son soportadas con el uso de IoC, específicamente en la declaración de interfaces, intermediarios y coherencia semántica de los componentes. Ejemplo de esto, es la separación de interfaz-implementación en donde el contenedor se encarga del ligado de estos en tiempo de ejecución, por lo que es más sencillo realizar la sustitución de nuevas implementaciones u otro tipo de componentes específicos para la realización de pruebas (*stubs* o *mock objects*) [17].

3.2.3.1.6 Manejo de tácticas de usabilidad.

Spring propone el uso del patrón de diseño MVC para desarrollar sistemas separando los componentes de la interfaz de usuario, la lógica de negocio y el acceso a recursos. Al realizar esta separación de responsabilidades el usuario puede disponer únicamente, en su computadora personal, de un pequeño sistema que implemente la interfaz de usuario (aplicación ligera), dejando todos los componentes pesados a la maquina que contenga tanto los componentes de negocios como los datos. Por otra parte, al implementar este patrón los desarrolladores se ven beneficiados, pues les es más fácil desarrollar el sistema por módulos con responsabilidades específicas por lo tanto esta táctica también se puede considerarse dentro de las tácticas de modificabilidad.

Esta separación de componentes es muy similar a la propuesta por el patrón arquitectónico de tres-tercios, sin embargo este patrón MVC es considerado un patrón de diseño ya que propone la composición y funcionamiento entre ciertos componentes. MVC dentro de Spring propone un flujo específico, a través componentes, para el manejo de solicitudes de usuarios [24].

En las siguientes secciones se describirá los elementos del PSM que permitirán representar una arquitectura con propiedades que definen la configuración de tácticas de calidad.

3.3 Elementos del PSM sobre Spring.

Como se mencionó anteriormente, el perfil del PSM es una definición de los componentes y sus propiedades que nos permitirán construir modelos PSM para aplicaciones en desarrollo. Los distintos tipos de elementos, relaciones y propiedades que se definieron en el perfil PSM que permitirá el soporte de atributos de calidad son los siguientes:

3.3.1 Tipos de Componentes.

Son elementos del modelo que representan componentes con funcionalidades específicas, acordes con su tipo y al paquete que pertenecen. En general este tipo de elemento contiene propiedades que permiten especificar características o funcionalidades. Los distintos tipos de componentes son:

- **JSP.** Es un componente que implementa funcionalidades de interacción con el usuario, como solicitar y desplegar información entre otras. Es modelado dentro del paquete de presentación, por lo que es parte de la implementación del patrón MVC (Modelo Vista Controlador).

- **Autentificador.** Este componente implementa funcionalidad de verificación y validación de información de usuarios del sistema. Es modelado dentro del paquete de presentación y tiene una asociación con el componente de vista de JSP.

- **Controlador.** Es el componente de control (MVC) que permite la interacción con los componentes del tercio de negocios.

- **Validador.** Es el componente que realiza validación de algunas operaciones de negocios realizadas por los componentes de manejadores de casos de uso. Este tipo de componentes son modelados dentro del paquete de negocios.

- **Manejador.** Este tipo de componente implementa la lógica de negocio específica de un sistema, es decir CU (casos de uso) de una aplicación.

- **DAO.** Es el componente que permite acceder y manipular información almacenada en una base de datos. Este componente es modelado dentro del paquete de datos.

- **Entidad.** Este componente representa los objetos que la aplicación utilizará para manipular información durante su ejecución (modelo de objetos de dominio), pero a su vez también representa las tablas de una base datos entidad relación. Para esto último en sus propiedades permite especificar elementos llave para el acceso de información así como el tipo de elemento. Este componente es modelado dentro del elemento de base de datos que a su vez este es modelado dentro del paquete de datos.

3.3.2 Interfaces.

Este tipo de elementos son utilizados para tener un bajo acoplamiento entre componentes. En este tipo de elementos se puede agregar información de los métodos que son implementados por los componentes que contiene la funcionalidad específica.

- **Interfaz Manejador de CU.** Es el tipo de interfaz utilizada para acceder a la funcionalidad de un Manejador de CU. Permite especificar información de los métodos que implementa los Manejadores de CU. Es modelado dentro del paquete de negocios.

- **Interfaz DAO.** Este tipo de interfaz permite acceder a la implementación para acceder a la información almacenada en una base de datos. Permite agregar información de los métodos utilizados en los DAOs. Es modelado dentro del paquete de datos.

3.3.3 Paquetes.

Son la representación lógica de los tercios de una aplicación. Los distintos tipos de paquetes que pueden ser modelados en este PSM son:

- **Presentación.** Es el paquete que contiene todos los componentes del tercio de presentación. Es decir, principalmente los componentes que permiten la interacción usuario-sistema.

- **Negocios.** Es el paquete que contiene todos los componentes que realizan lógica de negocios así como los componentes que permiten acceder a una funcionalidad.

- **Datos.** Es el paquete de contiene los elementos que permiten el acceso a la información almacenada en una base de datos.

3.3.4 Otros elementos y propiedades.

- **Base de Datos.** Elemento de modelo que representa la entidad de base de datos, permite configurar características propias del manejador de base de datos que se vaya utilizar.

- **Métodos.** Representan la especificación de los métodos que un componente de interfaz puede implementar. Dentro de la descripción que permiten realizar se encuentran los parámetros y valor de retorno.

- **Atributos.** Son elementos que representan propiedades de un componente (nombre, tipo).

3.3.5 Valores de atributos.

Este tipo de elementos del modelo representados por enumeraciones de los distintos valores que puede tomar un atributo específico. Algunos de las enumeraciones definidas dentro del perfil son:

- **JavaTypes.** Este tipo de enumeración enlista los posibles valores de tipos de datos java que un atributo de clase o componente de entidad puede tomar. Los valores son: *String, Long, Double, Float, Integer, Byte, Char, Boolean, Short* y *void*.

- **VisibilidadMetodos.** En esta enumeración se enlistan las visibilidades que pueden ser especificadas para métodos y atributos java. Los posibles valores son: *public, prívate, friend* y *package*.

- **TransaccionAislamiento.** Este tipo de enumeración enlista los valores que pueden tomar los atributos definidos para soportar el manejo transaccional para el nivel de aislamiento. Los posibles valores para este atributo son: *READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ* y *SERIALIZABLE*.

- **TransaccionPropagacion.** Este tipo de enumeración enlista los valores que permiten configurar el nivel de propagación para el manejo transaccional de un componente de negocios. Los posibles valores para este atributo son: *MANDATORY, NESTED, NEVER, NOT_SUPPORTED, REQUIRED, REQUIRES_NEW* y *SUPPORT*.

La Figura 20 muestra las enumeraciones definidas dentro del perfil PSM.

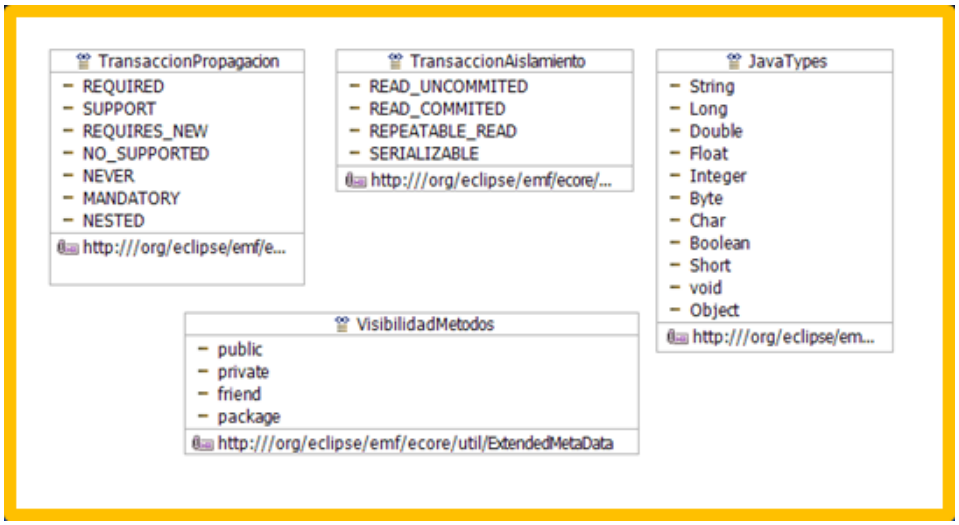

Figura 20. Valores de las enumeraciones.

Al momento de la definición del perfil PSM se agregan valores etiquetados para poder configurar en un modelo las tácticas de atributos de calidad. En el capítulo anterior se describieron las tácticas que pueden ser implementadas por los tres marcos de trabajo, sin embargo no todas ellas agregan marcas al perfil PSM, pues algunas son soportadas con la inclusión de componentes como interfaces. A continuación se describen los componentes y los valores etiquetados que son necesarios agregar al perfil para el conjunto de tácticas identificadas.

3.4 Modelado de tácticas de atributos de calidad por elementos.

En esta sección se describe el modelado de las tácticas de calidad en cada uno de los elementos del Perfil PSM. La estructuración de las siguientes secciones va desde la estructuración de la arquitectura en los patrones arquitectónicos hasta las propiedades de los elementos del modelo.

3.4.1 Modelado del patrón arquitectónico.

Uno de los patrones arquitectónicos que se está tomando como base del modelado con el perfil PSM es el patrón cliente-servidor [ver sección 2.1.2.2]. De este patrón arquitectónico, la estructura del cliente no está representada por componentes del modelo, pues se considera un cliente ligero en el cual los elementos de interacción son construidos dentro del servidor y enviados a un navegador para su presentación al usuario. Del lado del servidor, este está estructurado por todos los componentes que realizan operaciones de algún tipo de procesamiento en el sistema. Otro de los patrones arquitectónicos que son tomados como base de este proyecto es el patrón de tres-tercios. Este patrón es representado dentro del perfil PSM con los elementos definidos como paquetes, estos elementos son utilizados para representar la estructuración del sistema que se esté modelando. En este caso los paquetes representan cada uno de los tercios del patrón arquitectónico: presentación, negocios y datos. En cada uno de estos tercios únicamente pueden ser modelados elementos que tengan responsabilidades comunes con el tercio. Es decir, que en el patrón de presentación

únicamente podrán ser modelados elementos del perfil que estén involucrados con el manejo y presentación de la información que el usuario puede operar.

3.4.2 Modelado del tercio de Presentación.

Dentro de esta estructura de diseño se encuentran los elementos de modelado de componentes de software que permiten representar componentes para validar y manejar la información acorde con el dominio del problema. Entre los componentes que se modelan dentro de este tercio se encuentran los siguientes:

- **Componente de Presentación (JSP).** Este elemento representa a un componente con la responsabilidad específica de visualización e interacción de la información. Por lo tanto se puede decir que este elemento está diseñado para implementar tácticas de usabilidad (este componente es parte de la implementación del patrón MVC, el cual es utilizado como repuesta a la táctica de tiempo de diseño) y modificabilidad (este componente es definido dentro del *application context* utilizando el principio de IoC).

- **Componente de Seguridad de Autentificación (Autentificador).** En este elemento de modelado se representa una entidad encargada de realizar la verificación de información de acceso de los usuarios que requieran servicios del sistema. Es decir, al utilizar este elemento en el modelado de una arquitectura se están implementando tácticas de seguridad, específicamente de autentificación de usuarios.

- **Componente Vista Controlador (Controlador).** El modelado de este componente permite especificar la separación de los componentes de manejo de presentación de los componentes de control de negocios. Este elemento de modelado es la implementación del modelo vista controlador que es utilizado para soportar tácticas de usabilidad, ya que al igual que el componente de presentación este forma parte de la implementación del patrón de MVC.

En la Figura 21 se muestran los elementos del tercio de presentación en un diagrama UML del perfil.

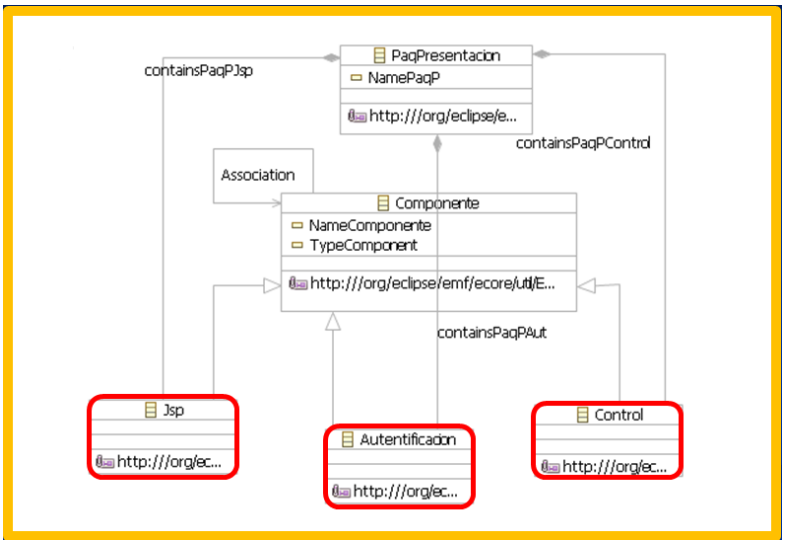

Figura 21. Elementos de modelado para Presentación.

3.4.3 Modelado del tercio de Negocios.

Dentro de esta estructura de diseño y de acuerdo con las especificaciones realizadas en el perfil, es posible modelar componentes de software encargados de realizar funcionalidades específicas del dominio del problema sobre el cual se elabora la aplicación. Es decir componentes de negocio y un componente de validación de información manejada en operaciones del sistema. Específicamente los componentes que pueden modelarse en este tercio de la arquitectura son:

- **Componente Manejador de Caso de Uso (Manejador).** Este elemento debe ser modelado por cada uno de los casos de uso que este compuesto el sistema. Como cada uno de estos componentes representa un modulo específico del sistema, con una funcionalidad determinada, estos componentes deben de tener características de manejo seguro sobre la información. Para poder realizar esto se agregaron los siguientes valores etiquetados:
    - **TransaccionPropagacion.** Valor etiquetado que permiten especificar el nivel de propagación de la transacción que sea manejada por el componente. Es decir, especifica si el componente pueden ser ejecutado dentro de un manejo transaccional.
    - **TransaccionAislamiento.** Valor etiquetado que permite especificar el nivel de asilamiento que requiera la transacción durante su ejecución. Es decir, especifica si el componente requiere ejecutarse dentro de una transacción específica para el componente o puede ser ejecutada dentro de otras transacciones existentes.
- **Interfaz de Manejador de Caso de Uso.** Este componente es la representación de la interfaz (componente intermediario) que realiza la separación de la implementación (Manejador de Caso de Uso). Es decir, entre ambos componentes permiten el soporte de tácticas de modificabilidad y facilidad de pruebas. También es necesario

mencionar, que al ser un elemento de interfaz, es posible especificar los métodos de funciones que el manejador tendrá que implementar.

- **Componente de Validación (Validador).** Este componente es utilizado para representar componentes necesarios para realizar validación de algún tipo de información, con esto se separa aún más las responsabilidades que el manejador de caso de uso tenga que realizar. Por lo tanto, con el modelado de este elemento se implementan tácticas de modificabilidad y facilidad de pruebas.

En la Figura 22 se muestran los elementos del perfil representados en un diagrama UML.

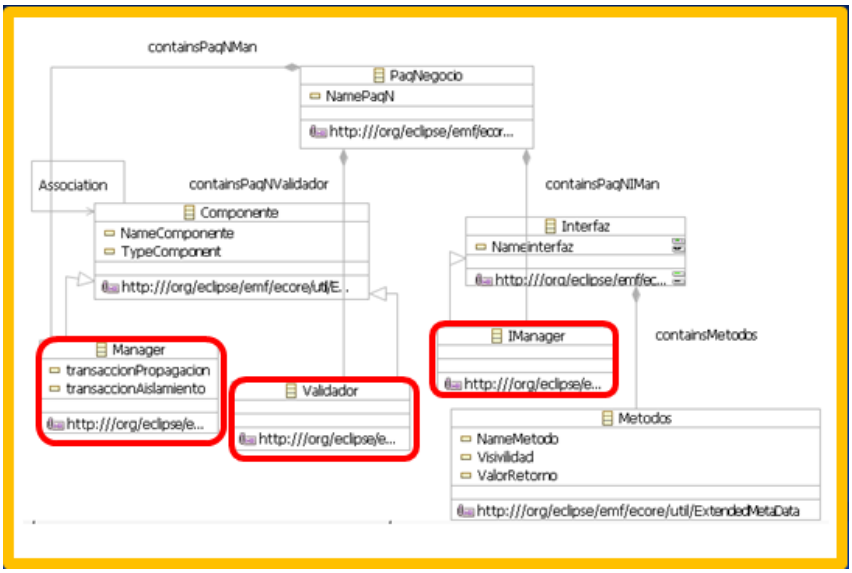

Figura 22. Elementos de modelado para Negocios.

3.4.4 Modelado del tercio de Datos.

Dentro de esta estructura de modelado se encuentran elementos que se encargan del manejo de persistencia de la información. Dentro de los tipos de elementos que pueden modelarse en este tercio se encuentran:

- **Componente DAO.** Este elemento de modelado representa el componente encargado de ejecutar las operaciones necesarias para realizar la persistencia de la información en la base de datos.

- **Interfaz de DAO.** Es el elemento que permite la interacción entre componentes de Manejo de casos de uso y los componentes que realizan la persistencia. Este elemento es un intermediario que separa la implementación de la interfaz. Con el modelado de este tipo de componentes se implementan tácticas de modificabilidad y facilidad de pruebas con el uso de intermediarios.

- **Elemento de Base de Datos.** Este elemento de modelado permite representar la entidad física de la base de datos. Por tal motivo este elemento tiene valores etiquetados que permiten configurar los elementos de acceso a la base

de datos. Sin embargo algunos de los valores etiquetados permiten configurar el soporte de tácticas de desempeño a través de propiedades del *Framework Hibernate*. Los valores etiquetados que pueden ser configurados son:

- **NameBD.** Representa el nombre con el cual será identificada la base de datos.
- **DriverClassName.** Este valor etiquetado representa el driver necesario de cada manejador de base de datos. Esta es una de las propiedades que deben ser definidas para poder establecer una conexión con la base de datos, es especificada en el *dataSource* dentro del *application context* de *Spring*.
- **Password.** Esta propiedad contiene la especificación de la contraseña de acceso a la base de datos (propiedad del *dataSource*).
- **UserName.** Esta propiedad contiene el nombre de usuario que será utilizado para tener acceso a operar la base de datos (propiedad del *dataSource*).
- **URL.** Define la ubicación de la base de datos dentro del servidor (propiedad del *dataSource*).
- **Dialect.** Así como el *DriverClassName* esta propiedad define el manejador de base de datos que será utilizado para acceder a la base de datos. Esta propiedad puede ser definida dentro del archivo de propiedades de configuración de hibérnate o en un *bean* dentro del *application context*. Este *bean* es llamado *hibernateProperties*.
- **minPoolSize.** Esta propiedad permite definir el tamaño mínimo de la cola de solicitudes que la base de datos puede atender. Al configurar esta propiedad es posible mejorar el desempeño del sistema. Esta propiedad es especificada dentro del *bean hibernateProperties*.
- **maxPoolSize.** Esta propiedad permite definir el tamaño máximo de la cola de solicitudes que la base de datos puede atender. Este valor permite configurar tácticas de desempeño. También es especificada dentro del *bean hibernateProperties*.
- **Timeout.** Esta propiedad permite definir el tiempo de inactividad en el cual una sesión con la base de datos estará activa, esto significa que después del tiempo especificado la sesión tendrá que restablecerse para volver a atender solicitudes hacia la base de datos.

- **Componente de Entidad.** El objetivo de este elemento es representar los componentes de entidad de la aplicación, es decir los elementos que almacenarán la información durante las operaciones del sistema, pero a su vez representa las tablas de la base de datos en las cuales se realiza la persistencia de la información. Este tipo de elemento permite especificar los atributos de las entidades, el tipo del atributo, así como el elemento que será

considerado como llave primaria dentro de la representación de la tabla de la base de datos. Este tipo de elementos permiten el soporte de tácticas de desempeño en conjunto con el componente de base de datos.

Los elementos del perfil, entidad y atributo, así como los valores etiquetados para cada uno de ellos son mostrados en la Figura 23.

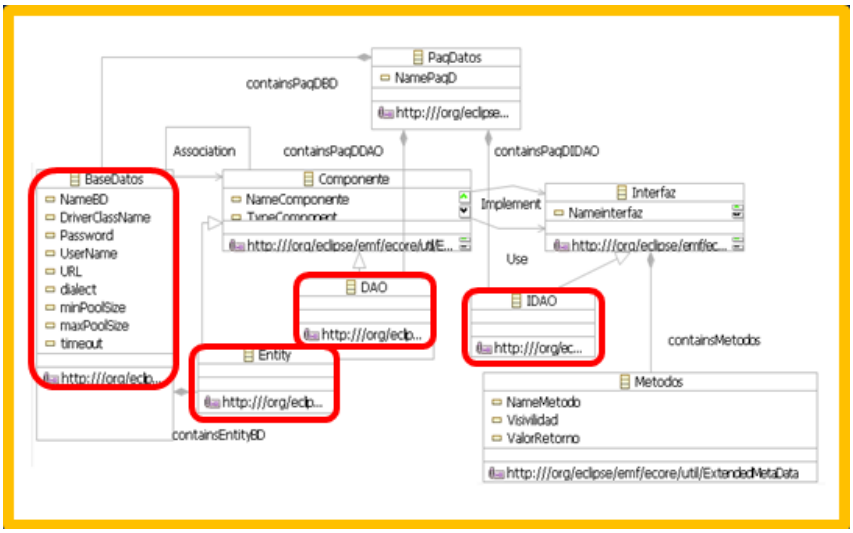

Figura 23. Elementos de modelado de Desempeño.

En la Figura 24 se muestra la definición dentro del perfil del componente de base de datos y los valores etiquetados que permiten la configuración el desempeño de las colas de solicitudes, así como los atributos necesarios para la conexión a la base de datos.

Figura 24. Elemento de configuración del componente de Base de Datos.

Hasta ahora solo se han descrito los elementos, componentes y propiedades definidas en el perfil que permiten modelar las tácticas de atributos de calidad. En la siguiente sub sección se describen el conjunto de reglas de transformación propuestas para la generación de código que den soporte a cada atributo de calidad mencionado.

3.5 Reglas de Transformación sobre Spring.

Las reglas de transformación descritas a continuación en la Tabla 20 están definidas textualmente, considerando los elementos planteados en las secciones anteriores, estos elementos permiten especificar características que dan soporte al sub conjunto de atributos de calidad previamente definido. Estas reglas describen los archivos y a grandes rasgos las secciones de código específico que deben ser generados a partir de cada uno de los elementos del perfil PSM.

Núm. de Regla.	Elementos del modelo	Reglas de transformación para los archivos generados
1	**Sistema o PSM**	Para el sistema que se esté modelando se debe generar un archivo descriptor XML (*application context*).
2	**JSP(Login)**	Para el sistema modelado se debe generar un archivo JSP que permita solicitar información de acceso (usuario y contraseña).
3	**JSP**	Por cada componente JSP se debe definir el mapeo correspondiente del componente JSP en el *application context*.
4	**Autentificador**	Para el único componente Autentificador modelado en el PSM se debe de generar un archivo *DispatcherServlet*, que debe contener las definiciones de los filtros de seguridad. Estos filtros son los definidos por *Acegi* y son: integración, autentificación, interpretación de excepciones y por ultimo el interceptor de seguridad.
5	**Autentificador**	Para el único componente Autentificador del sistema se deben definir los *beans* que permitan el manejo de seguridad dentro del *application context*, estos *beans* son: el filtro de procesamiento de autentificación, el manejador de autentificación y autentificación usando DAO.
6	**Controlador (MVC)**	Por cada componente Controlador modelado se debe generar un archivo Java encargado de separar y conectar las relaciones entre los componentes de presentación y negocio.
7	**Controlador (MVC)**	Por cada componente de control se debe especificar las relaciones entre los componentes de control con los JSP correspondientes en el *application context*.
8	**Manejador de CU**	Por cada componente manejador de caso de uso se debe generar un archivo Java con las propiedades del componente como atributos de la clase.
9	**Manejador de CU**	Por cada manejador de caso de uso se debe definir un bean dentro del *application context* que este referenciado al componente manejador de CU, así como el manejo transaccional correspondiente definido en los valores etiquetados del manejador.
10	***Valor etiquetado* TransaccionPropagacion.**	Para el valor etiquetado de *TransaccionPropagacion* de cada componente de manejador de CU se debe definir el nivel de propagación dentro del *bean* correspondiente del componente manejador de CU.
11	***Valor etiquetado* TransaccionAislamiento.**	Para el valor etiquetado de *TransaccionAislamiento* de cada componente de manejador de CU se debe definir el nivel de aislamiento dentro del *bean* correspondiente del componente manejador de CU.
12	**Interfaz Manejador de CU**	Para cada elemento de Interfaz de Manejador de CU se debe generar un archivo de Interfaz Java con los métodos correspondientes que contiene la representación del componente.
13	**Interfaz Manejador de CU**	Para cada elemento de Interfaz de Manejador de CU se debe definir la interfaz dentro del

		application context, así como la relación al correspondiente manejador de CU.
14	**Validador**	Para cada componente Validador modelado se debe generar un archivo Java encargado de la validación de información y datos.
15	**Validador**	Para cada componente Validador se debe definir el correspondiente *bean* dentro del *application context*.
16	**DAO (componente de integración)**	Por cada componente DAO modelado se debe generar un archivo Java que extienda de *HibernateDaoSupport*. Este archivo, entre otros, será utilizado para realizar la persistencia de información de una entidad.
17	**DAO (componente de integración)**	Por cada componente DAO se debe definir el un *bean* dentro del *application context*, así como sus propiedades y relaciones.
18	**Interfaz DAO**	Por cada elemento de Interfaz DAO modelado se debe generar un archivo Interfaz Java que contenga la declaración de los métodos definidos en el elemento del modelo.
19	**Interfaz DAO**	Por cada elemento de Interfaz DAO se debe definir un *bean* de interfaz de integración en el *application context* con sus relaciones.
20	**Base de Datos**	Por cada elemento de Base de Datos modelado se debe definir los *beans* de configuración dentro del *application context* de Spring. Estos *beans* son: *sessionFactory*, *dataSource* y *hibernateProperties*. Estos *beans* estas configurados con los valores etiquetados especificados en el elemento de Base de datos modelado.
21	***Valor etiquetado* NameBD**	Para el valor etiquetado de *NameBD* del componente de Base de Datos se debe definir el nombre de esta dentro de la configuración de propiedades de la base de datos en el archivo de configuración de *Hibernate*.
22	***Valor etiquetado* DriverClassName**	Para el valor etiquetado de *DriverClassName* del componente de Base de Datos se debe definir el controlador que se utilizara para acceder a la base de datos, este controlador debe ser definido dentro del *bean* de *dataSource* del *application context*.
23	***Valor etiquetado* Password**	Para el valor etiquetado de *Password* del componente de Base de Datos se debe especificar la contraseña de acceso para acceder a la base de datos, esta contraseña debe ser definida dentro del *bean* de *dataSource* del *application context*.
24	***Valor etiquetado* UserName**	Para el valor etiquetado de *UserName* del componente de Base de Datos se debe definir el nombre de usuario para acceder a la base de datos, este nombre de usuario debe ser definido dentro del *bean* de *dataSource* del *application context*.
25	***Valor etiquetado* URL**	Para el valor etiquetado de URL del componente de Base de Datos se debe definir la ubicación de la base de datos, esta ubicación debe ser definida dentro de la configuración de propiedades del *bean* de *dataSource* del *application context*.
26	***Valor etiquetado* dialect**	Para el valor etiquetado de *Dialect* del componente de Base de Datos también define el manejador de base de datos que será utilizado, esta propiedad debe ser definida dentro de la configuración de propiedades del *bean* de *hibernateProperties* del *application context*.
27	***Valor etiquetado* minPoolSize**	Para el valor etiquetado de *minPoolsize* del componente de Base de Datos se debe definir el tamaño mínimo de la cola de solicitudes que pueden ser atendidas para acceder a información de la base de datos, este tamaño debe ser definido dentro de la configuración de propiedades del *bean* de *hibernateProperties* del *application context*.
28	***Valor etiquetado* maxPoolSize**	Para el valor etiquetado de *maxPoolsize* del componente de Base de Datos se debe definir el tamaño máximo de la cola de solicitudes que pueden ser atendidas para acceder a información de la base de datos, este tamaño debe ser definido dentro de la configuración

		de propiedades del *bean* de *hibernateProperties* del *application context*.
29	***Valor etiquetado* timeout**	Para el valor etiquetado de *timeout* del componente de Base de Datos se debe definir el tiempo de tolerancia para que una sección con la base de datos este activa, esta propiedad debe ser definida dentro del *bean* de *hibernateProperties* del *application context*.
30	**Componentes de entidad**	Por cada componente de entidad modelado se debe generar un archivo Java correspondiente a la clase de dominio de la entidad.
31	**Componentes de entidad**	Por cada componente de entidad se debe definir una entrada de tipo *bean* correspondiente a la clase de dominio en el *application context*.
32	**Componentes de entidad**	Por cada componente de entidad se debe generar un archivo de mapeo objeto-entidad relación con extensión hbm.xml.
33	**Componentes de entidad**	Por cada componente de entidad se debe definir los objetos de dominio de mapeo en el *application context*, específicamente dentro de la propiedad de *mappingResources* del *bean* de *sessionFactory*.
34	**Atributo del componente de Entidad**	Por cada atributo del componente de entidad se debe definir el atributo, dentro de la clase de dominio Java, correspondiente junto con los métodos *set* y *get* para el manejo de información.
35	**Atributo del componente de Entidad**	Por cada atributo del componente de entidad se debe definir la propiedad correspondiente en el archivo de mapeo de *Hibernate*.

Tabla 20. Reglas textuales de transformación de las tácticas de atributos de calidad.

Las reglas anteriormente mencionadas son reglas de transformación definidas de manera textual. En el siguiente capítulo se describe el proceso realizado para desarrollar una herramienta que implemente las reglas de transformación textuales definidas. Es decir, agregación de valores etiquetados dentro del perfil PSM y la implementación de las reglas dentro del *Framework Acceleo*.

4 Desarrollo y evaluación de la herramienta.

En este capítulo se muestra el trabajo de desarrollo realizado durante la elaboración de este proyecto, así como también se presentan los resultados obtenidos de la evaluación realizada a la herramienta construida en este proyecto de investigación. En la siguiente sub sección se describen los procesos seguidos para poder desarrollar la herramienta de modelado de tácticas de atributos de calidad en un modelo arquitectónico.

4.1 Desarrollo de la herramienta.

Para el desarrollo de la herramienta que permita modelar las tácticas de atributos de calidad sobre un modelo específico de plataforma, es necesario primero crear un perfil PSM que nos permita modelar la arquitectura de una aplicación así como también el subconjunto de tácticas que se han seleccionado, acorde con la plataforma elegida.

4.1.1 Proceso de creación del Perfil PSM.

El proceso seguido para la creación del perfil consiste en utilizar una herramienta que permita definir el perfil PSM y poder generar modelos utilizando ese perfil. El *Framework* utilizado para la definición del perfil, así como el editor que permita crear modelos a partir de este perfil, es GMF (*Graphical Modeling Framework*) [28]. Principalmente, el *Framework* GMF está compuesto por otros dos *Frameworks*: EMF (*Eclipse Modeling Framework*) y GEF (*Graphical Editing Framework*) [29] [30]. EMF permite diseñar el dominio de un problema dentro de un meta-modelo (elementos y propiedades de un modelo estructurado que serán utilizadas para modelar soluciones de un dominio específico). GEF es un *Framework* que permite especificar elementos visuales para modelar soluciones específicas utilizando el meta-modelo definido con EMF. Es decir, ambas herramientas permiten editar y modificar la definición del perfil y el editor grafico de este. Por último GMF permite obtener un generador que produce los editores gráficos necesarios para crear modelos del perfil. El detalle de la definición del perfil utilizando esta herramienta es descrito en un trabajo previo, pues el objetivo principal de este proyecto es la implementación de las tácticas de calidad en un perfil PSM. Sin embargo, es necesario mencionar algunos detalles de ese trabajo para poder comprender la implementación realizada.

4.1.1.1 Elementos de definición del Perfil.

Los elementos que son definidos en el perfil se plantearon en secciones anteriores como los componentes necesarios para modelar una aplicación utilizando el conjunto de tres *Frameworks* (*Spring*, *Hibernate* y *Acegi*). Estos elementos son definidos en el perfil utilizando la herramienta GMF, la cual permite clasificarlos. Es decir, utilizando métodos de extensión del perfil UML como estereotipos y valores etiquetados. La definición de elementos que GMF permite deben realizarse utilizando los siguientes elementos de especificación:

- **EClass.** Es un elemento de definición que permite representar componentes o entidades abstractas. Para la definición del perfil PSM de este proyecto este elemento es utilizado para representar componentes de software

como clases, interfaces, paquetes de tercios, componentes de la base de datos, atributos específicos de componentes, métodos de interfaz, así como también las posibles relaciones entre los componentes de software.

- **EAttribute.** Son elementos de definición que pueden ser especificados dentro de una *EClass* y permiten especificar características o atributos propios del componente que se esté representando. Este elemento puede tomar tipos de valores estándar definidos por UML, como *string, float*, etc, pero a su vez también pueden ser tomados los valores definidos en *EEnum* definidas dentro del mismo perfil.

- **EEnum.** Es un elemento de definición que contiene los posibles valores que un *EAttribute* de una *EClass* puede tomar. Es decir, este elemento contiene los tipos de valores específicos que un atributo de un componente puede tener. Para la definición del perfil PSM este elemento es utilizado para definir los posibles valores de algunos de los valores etiquetados que son necesarios para especificar el manejo de tácticas de calidad sobre un componente.

- **EEnumLiteral.** Son elementos que solo pueden especificar un único valor etiquetado dentro de una *EEnum* para un atributo en específico. Por ejemplo, en la definición de los valores etiquetados para el nivel de propagación de la táctica de manejo transaccional, este elemento con el valor de *REQUIRED* está contenido dentro de la numeración que contiene todos los posibles valores para especificar el nivel de propagación de un componente de manejador de CU.

- **EReference.** Es el tipo de elemento utilizado para representar las relaciones entre los elementos que se estén representando dentro del perfil. Existen tres tipos diferentes de relaciones que pueden ser utilizadas al momento de crear un perfil y estas son:

    - **Association.** Este tipo de relación entre elementos del perfil es utilizada para definir todas las posibles relaciones que existan entre los componentes de modelado que se estén representando. En la definición del perfil PSM es utilizada para especificar las relaciones de *use, implement, association.*

    - **Aggregation.** Este tipo de relación es utilizado para especificar si un elemento definido (*EClass*) está contenido dentro de otro. En este proyecto se utilizo esta relación para especificar que componentes de modelado están contenidos dentro de cada tercio de la arquitectura.

    - **Generalization.** Este tipo de relación es utilizado para especificar si un elemento (*EClass*) hereda propiedades de otro elemento abstracto. La generalización fue utilizada para poder definir distintos tipos de componentes que a su vez contienen características diferentes entre ellos, mismas que heredan dentro de su misma categoría.

La Figura 25 muestra la representación del perfil en un esquema de árbol. En esta representación del perfil se puede observar como los componentes son definidos dentro de la entidad *PSMProfile* que es la representación esquemática del perfil.

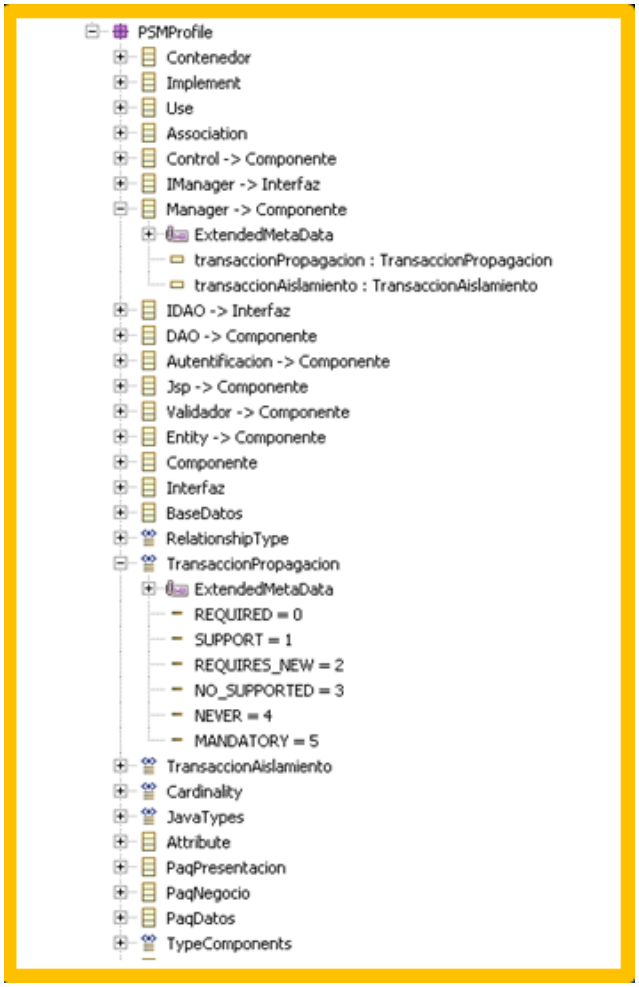

Figura 25. Esquema de árbol del Perfil PSM.

Sin embargo, esta no es la única representación que los plugins de GMF permiten visualizar o construir, ya que también es posible crear el perfil con un editor gráfico que la herramienta integra, así como también exportar el perfil en formatos de transferencia, tales como XMI, para que otras herramientas puedan interpretarlo y trabajar con él.

Retomando la definición del perfil mostrada en la Figura 25, la definición de los componentes que así lo requieran contendrán dentro de ellos los valores etiquetados para el manejo de tácticas de calidad, como es el caso del Manager -> Componente (Manejador de CU). Dentro de los componentes definidos en el perfil, existen algunos componentes que no pueden ser modelados gráficamente pues son representaciones abstractas, como por ejemplo las relaciones que existen entre los componentes. Esto de alguna manera limita las posibles relaciones creadas dentro de un modelo. Cabe mencionar que para la definición de este perfil no se utilizaron restricciones OCL (*Object Constraint Language*) entre sus

componentes. Esto es debido principalmente a que las restricciones que existen entre los componentes y sus propiedades pueden ser configuradas y especificadas dentro de la definición grafica del perfil mismo (utilizando GMF), así como también en las propiedades de los elementos gráficos que son utilizados para modelar un sistema.

Al crear el perfil con la herramienta GMF, el perfil y su editor gráfico son definidos dentro de tres proyectos que siguen el estándar de nombramiento de paquetes Java, agregando al final la fecha del último entregable. En la Figura 26 se muestra los tres proyectos que componen el perfil y el editor. Dentro del proyecto del perfil se encuentra la carpeta de *model* en la que están los archivos de configuración y definición del perfil. El archivo *PSMProfile.ecore_diagram* contiene la representación grafica del perfil (esquema); al abrir este archivo podemos observar la representación de cada uno de los elementos que componen el perfil así como un conjunto de barras de herramientas que permiten modificar o agregar nuevos elementos y relaciones entre ellos.

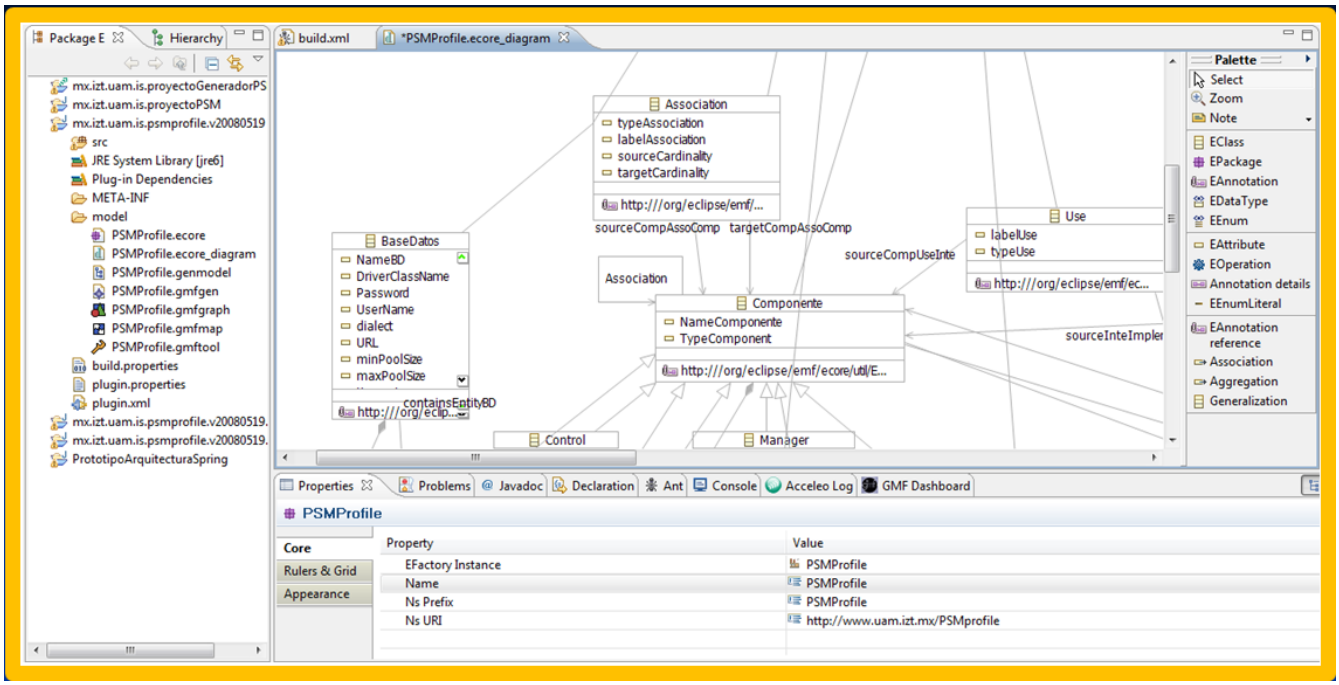

Figura 26. Definición del Perfil en tres proyectos de Eclipse.

En la siguiente sección se describe la manera en que fueron agregados elementos al perfil para introducir algunas de tácticas de atributos de calidad para así poder modelar sistemas con soporte de estas tácticas.

4.1.1.2 Introducción del soporte de tácticas en el perfil.

En la sub sección anterior se introdujeron los elementos que permiten crear un perfil UML; estos conceptos serán necesarios para poder seguir el proceso de introducción de algunas de las tácticas que se modelarán dentro del perfil PSM. En las siguientes sub secciones se describe la manera en que se definieron los elementos necesarios del perfil que permiten crear modelos con soporte de algunas tácticas de calidad. Algunas de estas tácticas son especificadas en el perfil

con la definición de valores etiquetados y otras con la definición de componentes específicos. Justamente en las siguientes sub secciones se describen dos de las tácticas identificadas y que son descritas utilizando valores etiquetados y componentes específicos en el perfil. Pro cuestiones de tiempo y espacio, únicamente se describe la implementación de dos de las tácticas.

4.1.1.2.1 Táctica de disponibilidad en el perfil.

La táctica de disponibilidad dentro del perfil es modelada definiendo valores etiquetados dentro del componente de Manejador de CU. Específicamente, para poder describir el soporte de disponibilidad en el perfil es necesario definir el componente de Manejador de CU y dentro de este las propiedades de disponibilidad, estas propiedades deben hacer referencia a los posibles valores o tipos que pueden tomar. Las acciones detalladas que se deben realizar para especificar el soporte de esta táctica son descritas a continuación:

1. Dentro del perfil se debe seleccionar de la paleta de herramientas el elemento *EClass* y agregar un elemento de este tipo al perfil (el perfil es el área de trabajo principal en el *Framework*).

2. Al elemento recientemente agregado se le debe dar el nombre *Manager* para identificar al tipo de componente correspondiente al Manejador de CU.

3. Dentro de las propiedades de este elemento se debe especificar la super *EClass* a la que pertenece el *Manager*, en este caso es Componente (este elemento es otra *EClass* previamente agregada al perfil). De esta *EClass* heredará las propiedades comunes de *NameComponente* y *TypeComponent*.

4. Dentro del componente Manager se debe agregar un elemento *EAttribute*, de la paleta de Herramientas, para cada uno de los valores etiquetados que se desean especificar como parte del soporte de la táctica de disponibilidad. Es decir, un *EAttribute* para el nivel de propagación y otro para el nivel de aislamiento. La Figura 27 muestra la parte del perfil referente a los componentes mencionados.

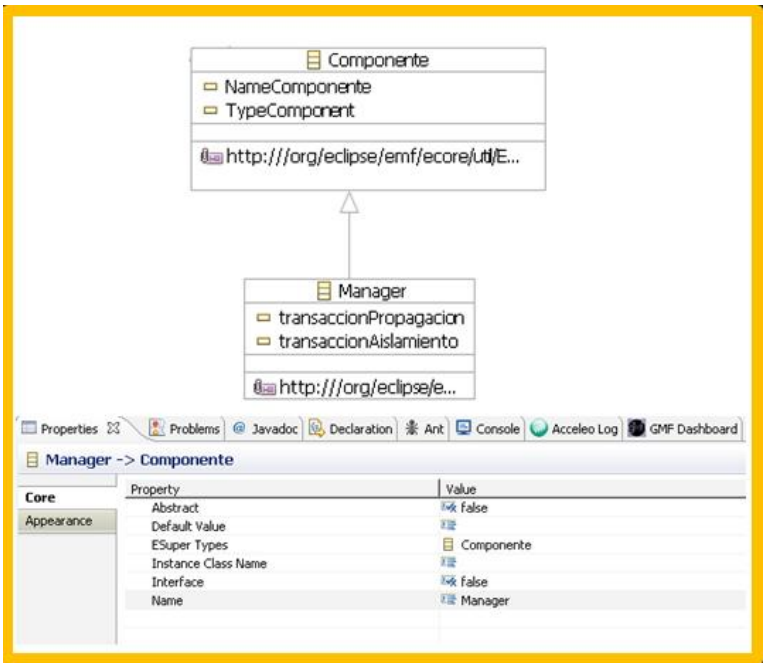

Figura 27. Definición del componente de Manager y sus propiedades en el perfil.

5. Dentro del perfil se debe agregar una *EEnum* por cada *EAttribute* agregado al elemento Manager. Las dos *EEnum* son: *TransaccionPropagacion* y *TransaccionAislamiento*.

6. Para cada una de las *EEnum* agregadas al perfil se debe agregar un *EEnumLiteral* por cada valor posible que pueda tomar cada una de los atributos de disponibilidad definidos dentro del *Manager*. La Figura 28 muestra los valores definidos de las *EEnumLiteral* dentro de las *EEnum* correspondientes.

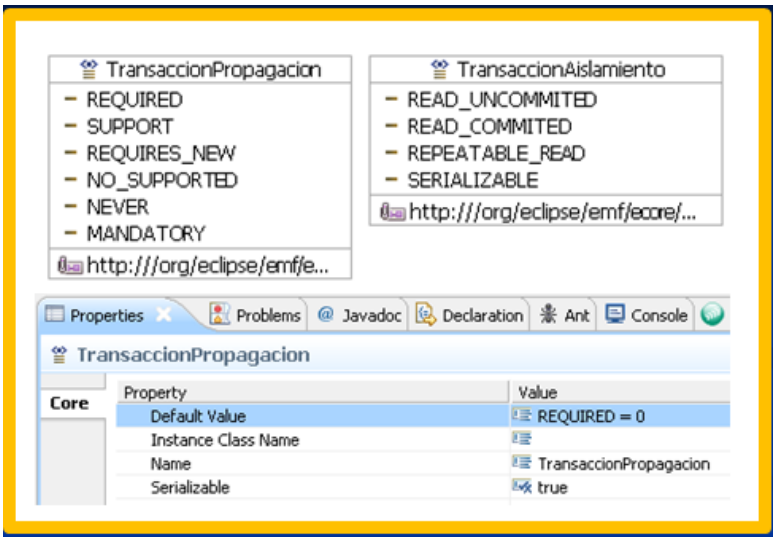

Figura 28. Definición de las EEnum para los valores etiquetados de Transacciones.

7. Dentro de las propiedades de los *EAttribute* de *transaccionPropagacion* y *transaccionAislamiento* definidos en el componente de *Manager* se especifica el *EAttribute Type* referenciando a la *EEnum* correspondiente; esto se especifica de esta forma para que esta propiedad pueda tomar los valores definidos dentro de la *EEnum*. La Figura 29 muestra la vista de propiedades de uno de los *EAttributos* definidos dentro del *Manager*.

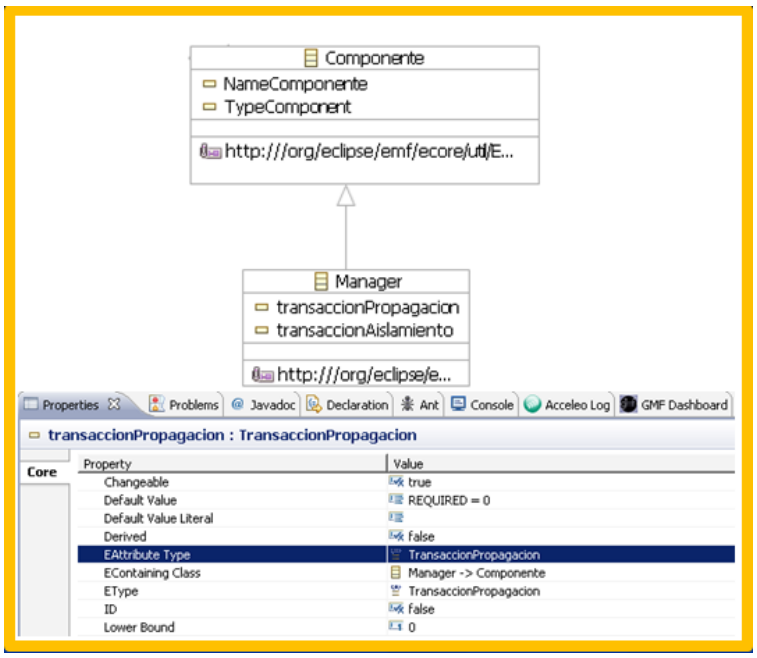

Figura 29. Estableciendo la relación de los valores etiquetados hacia las EEnum.

4.1.1.2.2 Táctica de desempeño en el perfil.

Para poder definir la táctica de desempeño, es necesario considerar los elementos del perfil de Base de datos y entidad así como sus principales relaciones y propiedades. El procedimiento para definir estos componentes en el perfil está descrito a continuación:

1. Dentro del perfil se debe seleccionar de la paleta de herramientas el elemento *EClass* y agregar un elemento de este tipo al perfil.

2. A la *EClass* agregada se debe introducir el nombre de *BaseDatos* dentro de la vista de propiedades de este elemento.

3. De la paleta de herramientas, para el elemento *BaseDatos* se debe agregar un *EAttribute* por cada uno de los siguientes valores etiquetados: *NameBD*, *DriverClassName*, *Password*, *UserName*, *URL*, *dialect*, *minPoolSize*, *maxPoolSize* y *timeout*.

4. Para cada valor etiquetado agregado se debe especificar el valor del *EAttribute Type* correspondiente a cada uno. Para la mayoría de estos se específico el valor de *String*, con la excepción de *minPoolSize*, *maxPoolSize* y *timeout* que son definidos como *Integer*.

5. Dentro del perfil se debe seleccionar y agregar otro *EClass* y nombrarlo como *Entity*. Este elemento debe tener una relación de agregación desde el componente *BaseDatos*. Para realizar esto es posible especificar la *ESuper Types* de *BaseDatos* dentro de las propiedades del *Entity*, o también es posible dibujar una relación de *Aggregation* desde *BaseDatos* hacia *Entity*.

6. Dentro del perfil se debe seleccionar y agregar una *EClass* y nombrarla *Attribute*.

7. Se debe agregar un *EAttribute*, dentro de la *EClass* del *Attribute*, para cada uno de los siguientes valores etiquetados: *NameAtt*, *TypeAtt* y *keyAtt*. Los primeros dos valores etiquetados permiten definir atributos de clase así como también columnas de una tabla, pues el elemento de *Entity* permite representar componentes de dominio y tablas de una base de datos. Sin embargo para poder realizar una representación adecuada de una tabla se necesita de elementos que sirvan de llave principal de la tabla, justamente para esto es definido el *EAttribute keyAtt* con un *EType Boolean*.

8. Se debe definir una relación de agregación desde la *EClass Entity* hacia *Attribute*. La Figura 30 muestra la definición de estos elementos dentro del perfil.

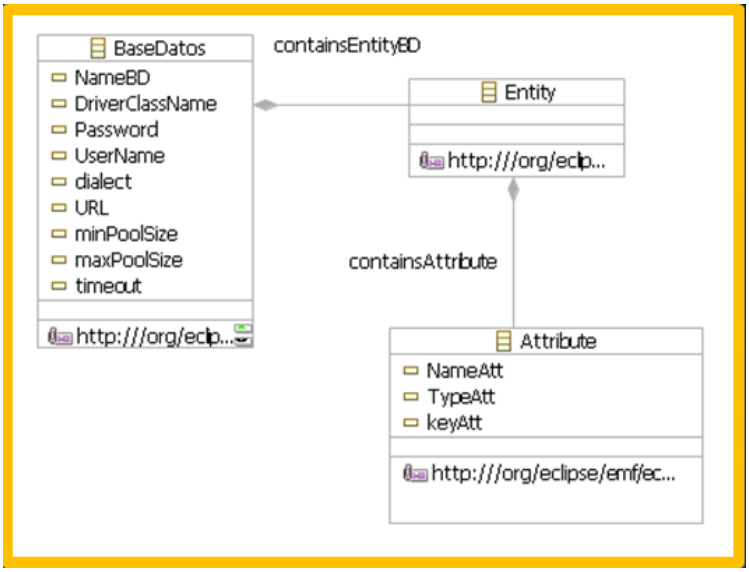

Figura 30. Definición de los elementos de desempeño en el perfil PSM.

Ya que se ha mencionado como realizar la definición de algunas tácticas de calidad dentro del perfil, es posible comenzar a describir cómo es que estas tácticas son traducidas a código para generar un esqueleto de arquitectura ejecutable a partir de un modelo de una arquitectura de una aplicación. En la siguiente sección se describe el proceso

realizado para definir las reglas de transformación definidas en el capítulo anterior y las cuales indican que archivos son generados a partir de elementos del modelo.

4.1.2 Proceso de definición de reglas de transformación.

Para poder comenzar a realizar la definición de reglas de transformación es necesario primero determinar cual herramienta será utilizada para realizar esta actividad. En la siguiente sub sección se muestra el análisis realizado a varias herramientas, para finalmente seleccionar la herramienta que mejor se adecue para realizar el soporte de tácticas en este proyecto.

4.1.2.1 Herramientas de generación de código.

Existen varias herramientas que permiten realizar la generación de código; algunos de los *Framework* que permiten realizar esta labor son: *JET*, *AndroMDA* y *Acceleo*. Estas herramientas permiten considerar la generación de código siguiendo un enfoque MDA, es decir a partir de un modelo poder generar código ejecutable. Para la implementación de este proyecto se realizó un estudio de comparación entre estas tres herramientas.

Este estudio tiene como puntos de comparación las siguientes características:

- **Integración con Eclipse.** Se debe considerar que la herramienta permita una integración directa con el entorno de desarrollo Eclipse. Esto se debe principalmente a que se pretende minimizar el riesgo de aprendizaje de un ambiente nuevo. Así como también considerar que tanto los *Framework*s de desarrollo (*Hibernate*, *Acegi* y *Spring*) y la herramienta de creación del perfil permiten la integración con Eclipse.

- **Implementación de reglas de transformación para los atributos de calidad.** Se debe considerar que la herramienta que se utilizara permita implementar todas las reglas de transformación, así como futuras reglas. Con esto se disminuirá el riesgo de sufrir retrasos en la implementación de reglas.

- **Soporte de modelos generados por herramientas GMF.** Se debe considerar que la herramienta a seleccionar pueda trabajar con las definiciones del perfil realizadas con GMF.

- **Generación de código de diferentes tipos de archivos.** Se debe considerar que la herramienta permita generar código para distintas plataformas, tales como: *Spring*, *Java*, *Hibérnate*, *XML*, *JSP*, etc.

- **Curva de aprendizaje.** Se debe considerar que la herramienta seleccionada tenga una curva de aprendizaje corta. Es decir, que el tiempo de dedicación para aprender el manejo de la herramienta y la manera de construir los elementos que permitan realizar la generación de código no retrase el desarrollo del proyecto.

- **Facilidad de uso.** Se debe considerar que la herramienta no presenta un nivel de complejidad demasiado elevado en su uso. Es decir, que el manejo de la herramienta sea intuitivo y no requiera de un número excesivo de

elementos de configuración por parte de los elementos manejados. Por consecuencia con esto se pretende minimizar el tiempo de implementación de las reglas de transformación.

El resultado del análisis realizado a estos *Framework* se presenta en la Tabla 21:

	Acceleo	AndroMDA	JET
Integración con Eclipse.	**Si es factible.** Es inmediata y directa.	**No se encuentra disponible aun.**	**Si se encuentra disponible.** Sin embargo se requiere de un alto nivel de programación para lograrlo.
Implementación de reglas de transformación para los atributos de calidad.	**Si es factible.** Cualquier regla de transformación puede ser implementada en este, dependiendo directamente de lo explicito del modelo.	**Si es factible.** Cualquier regla pude ser implementada.	**Si es factible.** Cualquier regla de transformación puede ser implementada de manera fácil.
Soporte de modelos generados por herramientas GMF.	**Si es factible.** Soporta el manejo de modelos creados por perfiles específicos.	**Si es factible.** Soporta el manejo de modelos PSM para distintas plataformas.	**Si es factible.** Soporta el manejo de modelos, pero es muy complicada la navegación que se proporciona de este.
Generación de código de diferentes tipos de archivos.	**Si es factible.** Permite generar cualquier tipo de archivo de texto.	**Si es factible.** Con el uso de cartuchos permite generar varios tipos de archivos de distintas tecnologías, y permite crear un cartucho propio para la transformación.	**Si es factible.** Permite generar cualquier tipo de archivo de texto.
Facilidad de uso.	**Si es posible.** Requiere elementos muy básicos de programación, el manejo del ambiente es muy intuitivo.	**No factible.** Facilidad de navegación en el ambiente, pero el proceso de configuración es extenso.	**No factible.** Se requiere leer demasiada información y es necesario tener buenas bases de programación en Java.
Curva de aprendizaje.	**Media**, debido a la documentación concisa que existe, depende en parte del dominio del problema.	**Alta**, debido al manejo de todas las tecnologías que se utilizan en conjunto para la ejecución.	**Alta**, se requiere leer demasiada documentación y tener conocimientos previos de programación.

Tabla 21. Comparativa de herramientas de generación de código.

Por el tiempo de desarrollo de este proyecto se tomo como base de estudio el estado del arte realizado en el proyecto de investigación de *"Un enfoque MDA para el desarrollo de aplicaciones basadas en un modelo de componentes orientados a servicios"* [23], así como los tutoriales de implementación de cada una de las herramientas. Debido a que *Acceleo* es una herramienta fácil de utilizar y el único problema identificado durante el análisis fue en que la facilidad de implementación de reglas depende del dominio del problema y de la correcta definición del modelo, *Acceleo* es la herramienta seleccionada para la realización de la implementación de las reglas de transformación. En la siguiente sub sección se describen algunos conceptos fundamentales sobre el funcionamiento de esta herramienta de generación de código.

4.1.2.1.1 Acceleo.

Acceleo es una herramienta que permite la generación de archivos usando modelos creados en UML, MOF y EMF. *Acceleo* permite generar distintos tipos de archivos: XML, Java, Descriptores de Servicios, JSP, y en general cualquier archivo de texto. Este *Framework* principalmente requiere de tres componentes para la generación de código, estos componentes son:

- **Meta-modelo (Perfil PSM).** Este meta-modelo (perfil) contiene todos los elementos y propiedades de los componentes que pueden ser representados dentro de un PSM.

- **Modelo PSM de una aplicación.** Es un modelo que representa la arquitectura de una aplicación y los componentes de este modelo contienen valores etiquetados con una configuración específica para soportar alguna táctica de atributo de calidad.

- **Plantillas de trasformación por cada tipo de archivo a generar.** Este tipo de componente contiene las reglas de transformación por cada componente para generar los archivos de código correspondientes.

Estos tres tipos de elementos que *Acceleo* requiere son configurados dentro de cadenas de generación. Estas cadenas de generación contienen todas las plantillas que generarán los archivos, el meta-modelo, el modelo PSM, así como la ruta de generación de los archivos y un archivo de registro de errores.

La Figura 31 muestra los elementos que *Acceleo* requiere para la generación de código siguiendo un enfoque MDA.

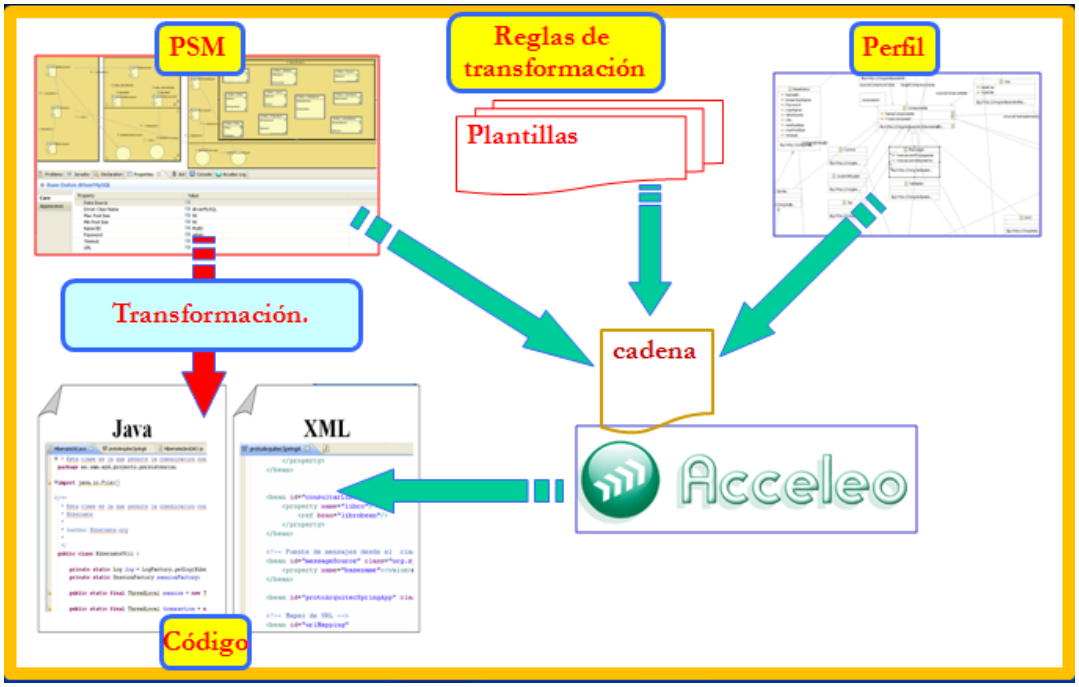

Figura 31. Transformación de modelo a código por Acceleo..

Ya que se describió cual es la herramienta de generación y cuáles son los elementos que esta considera, se puede proceder con la especificación del proceso seguido para la definición de algunas reglas de transformación definidas que implican el soporte de tácticas de atributos de calidad.

Siguiendo con el enfoque realizado hasta ahora en el proceso de creación del perfil, en donde se describió la manera de especificar los elementos necesarios para soportar tácticas de disponibilidad y desempeño, en las siguientes secciones se describen la forma en que fueron implementadas las reglas de transformación referentes a estas dos tácticas.

4.1.2.2 Definición de reglas de transformación de disponibilidad.

Para la definición de las reglas de transformación de disponibilidad se retomará la tabla 8 de reglas de transformación y se considerarán los elementos del perfil que se definieron en el proceso de creación de este. La definición de las reglas de transformación se definen sobre los componentes que principalmente permiten el soporte de disponibilidad, es decir, el componente de *Manager*, sus valores etiquetados y la utilización de las *EEnum* para la configuración de estos últimos. Con respecto a las reglas de transformación también se consideran aquellas reglas generales que son necesarias por englobar los componentes necesarios, tales como la definición de un *application context* por cada sistema (regla 1 de la tabla 8). El procedimiento para la definición de las reglas de transformación de disponibilidad es presentado a continuación:

Para la regla de transformación 1 en donde se define la estructura principal del archivo *application context* se debe especificar en una plantilla de transformación, llamada *webApplicationContext-Contenedor.mt*, que es el esqueleto básico de este archivo de configuración. Este esqueleto debe contener todas las llamadas al resto de templetes o plantillas, por lo que deben ser especificadas cada una de ellas en la sección de *import*, que permitirán construir por completo este archivo. Esto se debe a que el resto de estas plantillas configuran una parte específica de componentes particulares y que han sido definidos por separado para su mejor comprensión y funcionamiento por componente. En la Tabla 22 se muestra el código que debe contener esta plantilla de transformación.

Descripción de la regla de transformación	Archivo generador de código
1. Para el sistema que se esté modelando se debe generar un archivo descriptor XML **(webApplicationContext-Contenedor.mt).** Este archivo generador principalmente contiene la estructura básica para generar un *application context*, dentro de este templete existen llamadas a otros templetes que se encargan de generar secciones específicas de código referentes a otro tipo de componentes del modelo que deben ser especificados dentro del	<% **metamodel** /mx.izt.uam.is.psmprofile.v20080519/model/PSMProfile.ecore **import** mx.izt.uam.is.generador.configuration.descriptor.webApplicationContext-BaseDatos **import** mx.izt.uam.is.generador.configuration.descriptor.webApplicationContext-BeanNegocioIManager **import** mx.izt.uam.is.generador.configuration.descriptor.webApplicationContext-BeanNegocioManager **import** mx.izt.uam.is.generador.configuration.descriptor.webApplicationContext-BeansDominio **import** mx.izt.uam.is.generador.configuration.descriptor.webApplicationContext-DAO **import** mx.izt.uam.is.generador.configuration.descriptor.webApplicationContext-JSP **import** mx.izt.uam.is.generador.configuration.descriptor.webApplicationContext-MVC **import** mx.izt.uam.is.generador.configuration.descriptor.webApplicationContext-SessionFactory **import** mx.izt.uam.is.generador.configuration.descriptor.webApplicationContext-Validador %> **<%script type="**PSMProfile.Contenedor**" name="**Contenedor**" file="**web/WEB-

application context.

```
INF/webApplicationContext-servlet.xml"%>
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
        <!--
        ######################### CONFIGURACION DEL DATASOURCE
############################
        -->
        <%eAllContents("PSMProfile.BaseDatos").BaseDeDatos%>
        <!--
        ############################## BEANS DE DOMINIO
###################################
        -->
        <%eAllContents("PSMProfile.Entity").BeansDominio%>
        <!--
        ################################# BEANS DE MVC
#######################################
        -->
        <%eAllContents("PSMProfile.Control").MVC%>
        <!--
        ################################ BEANS DE VALIDACION
##############################
        -->
        <%eAllContents("PSMProfile.Validador").Validador%>
        <!--
        ##################### MANEJO DE MENSAJES DE messages.properties
####################
        -->
        <bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename"><value>messages</value></property>
    </bean>
    <!--
        ################################## MAPEO URL
#######################################
        -->
        <bean id="urlMapping"

        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
                <property name="mappings">
                        <props>
                                <%eAllContents("Jsp").JSP%>
                        </props>
                </property>
        </bean>
    <!--
        ####### Resolucion de Vistas eliminacion del path e dependencia con cotrolador
######
        -->
        <bean id="viewResolver"
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property
name="viewClass"><value>org.springframework.web.servlet.view.JstlView</value></property>
        <property name="prefix"><value>/WEB-INF/jsp/</value></property>
        <property name="suffix"><value>.jsp</value></property>
    </bean>
    <!--
        ######################### CONFIGURACION DE SESSION FACTORY
#######################
        -->
        <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean"
        depends-on="dataSource,hibernateProperties">
                <property name="hibernateProperties">
                        <ref bean="hibernateProperties" />
                </property>
                <property name="dataSource">
                        <ref local="dataSource"/>
                </property>
                <!--  OR mapping files. -->
                <property name="mappingResources">
                        <list>
                                <%eAllContents("Entity").SessionFactory%>
                        </list>
                </property>
        </bean>

        <!--
        ############################### BEANS DE DAOS
```

	`` ################################### -->` `<%eAllContents("DAO").DAO%>` <!-- ############################ DEFINICION DEL MANEJADOR DE TRANSACCIONES ################################### -->` `<bean id="transactionManager"` `  class="org.springframework.orm.hibernate.HibernateTransactionManager">` `  <property name="sessionFactory">` `  <ref bean="sessionFactory"/>` `  </property>` `</bean>` <!-- ############################ BEANS DE NEGOCIO ################################### -->` `<%eAllContents("Manager").BeanNegocioManager%>` `<%eAllContents("IManager").BeanNegocioIManager%>` </beans>```

Tabla 22. Definición de regla de generación del application context en código.

En las reglas de transformación 8, 9, 10 y 11 se especifica el código y archivos que deben ser generados para cada uno de los componentes de *Manager* de CU especificados en el modelo. Específicamente, por cada componente de *Manager* se deben crear el archivo Java en donde se especificara la lógica de negocios y también se debe realizar la definición de los *beans* del componente *Manager* y los *beans* que permitan realizar el manejo transaccional dentro del *application context*. La Tabla 23 muestra el código generado de estas reglas de transformación.

Descripción de la regla de transformación	Archivo generador de código
8. Por cada componente manejador (*Manager*) de caso de uso se debe generar un **archivo Java** con las propiedades del componente como atributos de la clase **(managerJAVA.mt).** Este archivo contiene principalmente las sentencias necesarias para generar los métodos definidos en la Interfaz correspondiente del *Manager* así como los atributos definidos dentro del componente *Manager* y los métodos *getter* y *setter* correspondientes a cada uno de los atributos.	``` <% metamodel /mx.izt.uam.is.psmprofile.v20080519/model/PSMProfile.ecore %> <%script type="PSMProfile.Manager" name="managerJAVA" file="src/mx/izt/uam/is/proyecto/aplicacion/manager/Manager<%NameComponente%>Impl.java"%> package <%paqueteaplicacion%>.manager; import java.io.Serializable; import org.apache.commons.logging.Log; import org.apache.commons.logging.LogFactory; import mx.izt.uam.is.proyecto.dominio.*; import mx.izt.uam.is.proyecto.persistencia.*; import mx.izt.uam.is.proyecto.aplicacion.interfaces.*; /** * <%NameComponente%> * * @author PM * */ public class Manager<%NameComponente%>Impl implements Serializable, Interfaz<%Implement.Nameinterfaz%> { /** * */ private static final long serialVersionUID = 2610240320311496060L; private Interfaz<%Use.Nameinterfaz%> <%Use.Nameinterfaz.toLowerCase()%>; <%for (containsAttComp){%> private <%TypeAtt%> <%NameAtt%>; <%}%> protected final Log logger = LogFactory.getLog(getClass()); <%for (Implement.containsMetodos){%>```

	<%Visivilidad%> <%ValorRetorno%> <%NameMetodo%>(<%for (containsParametros){%><%if (i+1==nSize()||i==nSize()-1){%><%}else{%>, <%}%><%TypeAtt%> <%NameAtt%><%}%>){ <%ValorRetorno%> returnValue = null; return returnValue; } <%}%> <%for (containsAttComp){%> /** * Proporciona el <%NameAtt.toUpperCase()%> * * @return <%NameAtt.toUpperCase()%> */ public <%TypeAtt%> get<%NameAtt.toU1Case()%>() { return <%NameAtt%>; } /** * Permite asignar <%NameAtt.toUpperCase()%> * * @param <%NameAtt.toUpperCase()%> */ public void set<%NameAtt.toU1Case()%>(<%TypeAtt%> <%NameAtt%>) { logger.info("\n ** set <%NameAtt.toUpperCase()%> "+ <%NameAtt%>); this.<%NameAtt%> = <%NameAtt%>; } <%}%> } <%script type="PSMProfile.Manager" name="paqueteaplicacion"%> mx.izt.uam.is.proyecto.aplicacion
9. Por cada manejador (*Manager*) de caso de uso se debe definir una entrada de tipo ***bean*** dentro del ***application context*** que este referenciado al componente manejador de CU, así como el manejo transaccional correspondiente definido en los valores etiquetados del manejador **(webApplicationContext-BeanNegocioManager.mt).** Este archivo generador contiene la definición de un bean de servicio, el cual hace referencia a los *beans* con la definición del *Manager*, del Proxy que implementará el manejo transaccional y las propiedades para dicho manejo.	<% metamodel /mx.izt.uam.is.psmprofile.v20080519/model/PSMProfile.ecore %> <%script type="PSMProfile.Manager" name="BeanNegocioManager"%> <!-- ******************Proxing CU <%NameComponente%> ********** --> <bean id="<%NameComponente%>Service" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"> <property name="target"> <ref bean="manager<%NameComponente%>Target"/> </property> <property name="proxyInterfaces"> <ref bean="mx.izt.uam.is.proyecto.aplicacion.Proxy<%Implement.Nameinterfaz%>"/> </property> <property name="transactionManager"> <ref bean="transactionManager"/> </property> <property name="transactionAttributes"> <props> <prop key="*">PROPAGATION_<%transaccionPropagacion%>, ISOLATION_<%transaccionAislamiento%>, readOnly</prop> </props> </property> </bean> <!-- ******************CU <%NameComponente%> ********** --> <bean id="manager<%NameComponente%>Target" class="mx.izt.uam.is.proyecto.aplicacion.<%NameComponente.toLowerCase()%>.Manager<%NameComponente%>Impl"> <%for (containsAttComp){%> <constructor-arg index="<%i%>"> <ref local="<%NameAtt.toLowerCase()%>" /> </constructor-arg> <%}%> </bean>
10. Para el valor etiquetado de *TransaccionPropagacion* de cada componente de manejador de CU se debe	**Esta especificado en el bean dentro del property nombrado transactionAttributes** <property name="transactionAttributes"> <props>

definir el nivel de propagación dentro del *bean* correspondiente del componente manejador de CU **(webApplicationContext-BeanNegocioManager.mt).**	<prop key="*">**PROPAGATION_<%transaccionPropagacion%>**, ISOLATION_<%transaccionAislamiento%>, readOnly</prop> </props> </property>
11. Para el valor etiquetado de *TransaccionAislamiento* de cada componente de manejador de CU se debe definir el nivel de aislamiento dentro del *bean* correspondiente del componente manejador de CU **(webApplicationContext-BeanNegocioManager.mt).**	**Esta especificado en el bean dentro del property nombrado transactionAttributes** <property name="transactionAttributes"> <props> <prop key="*">PROPAGATION_<%transaccionPropagacion%>, **ISOLATION_<%transaccionAislamiento%>**, readOnly</prop> </props> </property>

Tabla 23. Definición de reglas de Disponibilidad en código.

Estas son algunas de las reglas principales para el soporte de tácticas de disponibilidad. En la siguiente sección se describen las reglas de transformación seleccionadas para ejemplificar el soporte de tácticas de desempeño por componentes del modelo.

4.1.2.3 Definición de reglas de transformación de desempeño.

Al igual que en la definición de reglas de transformación de disponibilidad, se retomará la tabla 8 de reglas de transformación y se considerarán los elementos del perfil que se definieron en el proceso de creación del perfil para la definición de las reglas de transformación de desempeño. Para la definición de estas reglas son considerados los siguientes elementos del perfil con sus propiedades: el componente *BaseDatos*, el componente *Entity* y el elemento *Attribute*. En el caso del componente *BaseDatos*, cada una de sus propiedades permiten configurar características específicas de configuración de la base datos, pero principalmente las propiedades que permiten modelar el desempeño son: *timeout*, *maxPoolSize* y *minPoolSize*.

Las reglas de transformación 20, 22, 23, 24, 25, 26, 27 ,28 y 29 involucran al componente *BaseDatos* y sus propiedades, las cuales son plasmadas en el *bean* de definición de componentes dentro del *application context*. Los archivos de generación de código que permiten generar los *beans* de definición dentro del *application context* se presentan en la Tabla 24:

Descripción de la regla de transformación	Archivo generador de código
20. Por cada elemento de Base de Datos modelado se debe definir los *Beans* de configuración dentro del *application context* de *Spring*. Estos *Beans* son: *sessionFactory*, *dataSource* y *hibernatePropertiese*. Estos *beans* estas configurados con	<% **metamodel** /mx.izt.uam.is.psmprofile.v20080519/model/PSMProfile.ecore %> **<%script type="PSMProfile.BaseDatos" name="BaseDeDatos"%>** <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"> <property name="driverClassName"> <value><%DriverClassName%></value> </property>

los valores etiquetados especificados en el elemento de Base de datos modelado. **NOTA.** La sección del *bean* del *sessionFactory* está definida dentro del templete del contenedor, descrito en la regla de transformación 1 **(webApplicationContext-BaseDatos.mt).** Este archivo generador contiene la definición de los *beans* de *dataSource* y *hibernateProperties*. Cada uno de estos *beans* contiene valores etiquetados del componente *BaseDatos* que son necesarios para la configuración de la conexión hacia la base de datos.	`<property name="url">` `<value><%URL%>;create=true</value>` `</property>` `<property name="username">` `<value><%UserName%></value>` `</property>` `<property name="password">` `<value><%Password%></value>` `</property>` `</bean>` `<!--` `######### CONFIGURACION DE LAS PROPIEDADES DE LA BASE DE DATOS CON HIBERNARTE ############` `-->` `<bean id="hibernateProperties"` `class="org.springframework.beans.factory.config.PropertiesFactoryBean">` `<property name="properties">` `<props>` `<prop key="hibernate.hbm2ddl.auto">update</prop>` `<prop key="hibernate.dialect"><%dialect%></prop>` `<%-- org.hibernate.dialect.DerbyDialect --%>` `<prop key="hibernate.query.substitutions">true 'T', false 'F'</prop>` `<prop key="hibernate.show sql">false</prop>` `<prop key="hibernate.c3p0.minPoolSize"><%maxPoolSize%></prop>` `<prop key="hibernate.c3p0.maxPoolSize"><%minPoolSize%></prop>` `<prop key="hibernate.c3p0.timeout"><%timeout%>600</prop>` `<prop key="hibernate.c3p0.max_statement">50</prop>` `<prop key="hibernate.c3p0.testConnectionOnCheckout">false</prop>` `</props>` `</property>` `</bean>`
22. Para el valor etiquetado de *DriverClassName* del componente de Base de Datos se debe definir el controlador que se utilizará para acceder a la base de datos, este controlador debe ser definido dentro del *bean* de *dataSource* del *application context* **(webApplicationContext-BaseDatos.mt).**	**Esta especificado en el bean de dataSource en la property nombrada driverClassName** `<property name="driverClassName">` `<value><%DriverClassName%></value>` `</property>`
23. Para el valor etiquetado de *Password* del componente de Base de Datos se debe especificar la contraseña de acceso para acceder a la base de datos, esta contraseña debe ser definida dentro del *bean* de *dataSource* del *application context* **(webApplicationContext-BaseDatos.mt).**	**Esta especificado en el bean de dataSource en la property nombrada password** `<property name="password">` `<value><%Password%></value>` `</property>`
24. Para el valor etiquetado de *UserName* del componente de Base de Datos se debe definir el nombre de usuario para acceder a la base de datos, este nombre de usuario debe ser definido dentro del *bean* de *dataSource* del *application context* **(webApplicationContext-BaseDatos.mt).**	**Esta especificado en el bean de dataSource en la property nombrada username** `<property name="username">` `<value><%UserName%></value>` `</property>`
25. Para el valor etiquetado de URL del componente de Base de Datos se debe definir la	**Esta especificado en el bean de dataSource en la property nombrada URL** `<property name="url">` `<value><%URL%>;create=true</value>`

ubicación de la base de datos, esta ubicación debe ser definida dentro de la configuración de propiedades del *bean* de *dataSource* del *application context* **(webApplicationContext-BaseDatos.mt).**	`</property>`
26. Para el valor etiquetado de *dialect* del componente de Base de Datos también define el manejador de base de datos que será utilizado, esta propiedad debe ser definida dentro de la configuración de propiedades del *bean* de hibernateProperties del *application context* **(webApplicationContext-BaseDatos.mt).**	**Esta especificado en el bean de hibernateProperties en la lista de propiedades** `<prop key="hibernate.dialect"><%dialect%></prop>`
27. Para el valor etiquetado de *minPoolsize* del componente de Base de Datos se debe definir mínimo tamaño de la cola de solicitudes que pueden ser atendidas para acceder a información de la base de datos, este tamaño debe ser definido dentro de la configuración de propiedades del *bean* de hibernateProperties del *application context* **(webApplicationContext-BaseDatos.mt).**	**Esta especificado en el bean de hibernateProperties en la lista de propiedades** `<prop key="hibernate.c3p0.maxPoolSize"><%minPoolSize%></prop>`
28. Para el valor etiquetado de *maxPoolsize* del componente de Base de Datos se debe definir máximo tamaño de la cola de solicitudes que pueden ser atendidas para acceder a información de la base de datos, este tamaño debe ser definido dentro de la configuración de propiedades del *bean* de *hibernateProperties* del *application context* **(webApplicationContext-BaseDatos.mt).**	**Esta especificado en el bean de hibernateProperties en la lista de propiedades** `<prop key="hibernate.c3p0.minPoolSize"><%maxPoolSize%></prop>`
29. Para el valor etiquetado de *timeout* del componente de Base de Datos se debe definir el tiempo de tolerancia para que una sección con la base de datos este activa, esta propiedad debe ser definida dentro del *bean* de *hibernateProperties* del *application context* **(webApplicationContext-**	**Esta especificado en el bean de hibernateProperties en la lista de propiedades** `<prop key="hibernate.c3p0.timeout"><%timeout%>600</prop>`

BaseDatos.mt).	

Tabla 24. Definición de reglas de transformación del componente de BaseDatos en código.

Por otra parte, existen otros elementos definidos dentro del perfil que son utilizados para la definición de las reglas de transformación de desempeño, estos son el elemento *Entity* y los atributos que pueden ser definidos dentro de estos. Las reglas de transformación 30, 31, 32, 33, 34 y 35, hacen mención a las secciones de código que deben ser generadas para el soporte de esta táctica de desempeño. En la Tabla 25 se muestra la definición de las reglas de transformación para el desempeño respecto a los elementos *Entity* y *Attribute*.

Descripción de la regla de transformación	Archivo generador de código
30. Por cada componente de entidad modelado se debe generar un archivo Java correspondiente a la clase de dominio de la entidad **(dominioJAVA.mt).** Este archivo permite realizar la generación del archivo Java de dominio correspondiente por cada elemento *Entity* definido dentro del modelo. Dentro de este archivo se encuentran las definiciones de los atributos y sus correspondientes métodos *getter* y *setter*.	<% **metamodel** /mx.izt.uam.is.psmprofile.v20080519/model/PSMProfile.ecore %> **<%script type="**PSMProfile.Entity**" name="**dominioJAVA**"** **file="**src/mx/izt/uam/is/proyecto/dominio/<%NameComponente%>.java**"%>** package <%paquetedominio%>; import java.io.Serializable; import org.apache.commons.logging.Log; import org.apache.commons.logging.LogFactory; /** * <%NameComponente%> * @author PM * */ public class <%NameComponente.toUlCase()%> implements Serializable{ /** Logger for this class and subclasses */ protected final Log logger = LogFactory.getLog(getClass()); <%-- GENERACION DE LOS ATRIBUTOS DEL COMPONENTE ENTITY ASI COMO EL IDENTIFICADOR PRINCIPAL --%> /*---- Atributos de la clase ----*/ /** * Identificador unico de la clase */ /** * Atributo de <%NameComponente.toUpperCase()%> */ **<%for (containsAttribute){%>** private <%TypeAtt%> <%NameAtt.toLowerCase()%>; **<%}%>** /*--------------------------------*/ /** * Constructor de la clase * */ public <%NameComponente.toUlCase()%>(){ } **<%for (containsAttribute){%>** /** * Proporciona el <%NameAtt.toUpperCase()%> * * @return <%NameAtt.toUpperCase()%> */ public <%TypeAtt%> get<%NameAtt.toUlCase()%>() { return <%NameAtt.toLowerCase()%>; } /** * Permite asignar <%NameAtt.toUpperCase()%>

	`*` `* @param <%NameAtt.toUpperCase()%>` `*/` `public void set<%NameAtt.toU1Case()%>(<%TypeAtt%> <%NameAtt.toLowerCase()%>) {` `logger.info("\n ** set <%NameAtt.toUpperCase()%> "+ <%NameAtt.toLowerCase()%>);` `this.<%NameAtt.toLowerCase()%> = <%NameAtt.toLowerCase()%>;` `}` `<%if (NameAtt.startsWith("Id")&&TypeAtt.equalsIgnoreCase("Integer")){%>` `/**` `* Sobreescritura del metodo equals para poder asignar nuestro criterio de igualdad. `* En nuestro caso el criterio de igualdad sera si los identificadores son iguales` `*` `* @return true si son iguales, false si no` `*` `*/` `public boolean equals(Object obj){` `if(obj instanceof <%parent().NameComponente.toU1Case()%>){` `<%parent().NameComponente.toU1Case()%> aux = (<%parent().NameComponente.toU1Case()%>)obj;` `if(aux.get<%NameAtt.toU1Case()%>() == this.<%NameAtt.toLowerCase()%>){` `return true;` `}else{` `return false;` `}` `}else{` `return false;` `}` `}` `<%}else{%>` `<%}%>` `<%}%>` `}`  `<%script type="PSMProfile.Entity" name="paquetedominio"%>` `mx.izt.uam.is.proyecto.dominio`
31. Por cada componente de entidad se debe definir un *bean* correspondiente a la clase de dominio en el *application context* **(webApplicationContext-BeansDominio.mt).**	`<%` `metamodel /mx.izt.uam.is.psmprofile.v20080519/model/PSMProfile.ecore` `%>` `<%script type="PSMProfile.Entity" name="BeansDominio"%>` `<bean id="<%NameComponente%>"` `class="mx.izt.uam.is.proyecto.dominio.<%NameComponente%>"></bean>`
32. Por cada componente de entidad se debe generar un archivo de mapeo objeto-entidad relación con extensión hbm.xml **(entidadHBM.mt).** Dentro de este archivo generador se realiza la definición del mapeo entre atributos de un elemento de dominio y las columnas de una tabla de la base de datos. Dentro de esta definición se realiza la identificación del atributo que servirá como llave primaria en la tabla de base de datos correspondiente.	`<%` `metamodel /mx.izt.uam.is.psmprofile.v20080519/model/PSMProfile.ecore` `%>` `<%script type="PSMProfile.Entity" name="entidadHBM"` `file="src/mx/izt/uam/is/proyecto/hbm/<%NameComponente%>.hbm.xml"%>` `<?xml version="1.0"?>` `<!DOCTYPE hibernate-mapping PUBLIC` `"-//Hibernate/Hibernate Mapping DTD 3.0//EN"` `"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">` `<hibernate-mapping>` `<class name="mx.izt.uam.is.proyecto.dominio.<%NameComponente%>" table="<%NameComponente%>">` `<%for (containsAttribute){%>` `<%if (keyAtt.equalsIgnoreCase("true")){%>` `<!-- Identificador unico de la clase -->` `<id name="<%NameAtt.toLowerCase()%>" column="<%NameAtt.toLowerCase()%>" type="int">` `<!-- El valor de increment siginifica que el valor inicial es 1 y se incrementa en 1 cada vez que se agrega un registro -->` `<generator class="increment"/>` `</id>` `<%}else{%>` `<!-- <%NameAtt.toUpperCase()%> -->` `<property name="<%NameAtt.toLowerCase()%>" type="<%TypeAtt.toLowerCase()%>"/>`

	<%}%> <%}%> </class> </hibernate-mapping>
33. Por cada componente de entidad se debe definir los objetos de dominio de mapeo en el *application context*, específicamente dentro de la propiedad de *mappingResources* del *bean* de *sessionFactory* **(webApplicationContext-SessionFactory.mt).**	<% **metamodel** /mx.izt.uam.is.psmprofile.v20080519/model/PSMProfile.ecore %> <%script type="PSMProfile.Entity" name="SessionFactory"%> <value>mx/izt/uam/is/proyecto//hbm/<%NameComponente%>.hbm.xml</value>
34. Por cada atributo del componente de entidad se debe definir el atributo, dentro de la clase de dominio Java, correspondiente junto con los métodos *set* y *get* para el manejo de información **(dominioJAVA.mt).**	**Esta especificado dentro del archivo generador de dominio** /** * Atributo de <%NameComponente.toUpperCase()%> */ <%for (containsAttribute){%> private <%TypeAtt%> <%NameAtt.toLowerCase()%>; <%}%> <%for (containsAttribute){%> /** * Proporciona el <%NameAtt.toUpperCase()%> * * @return <%NameAtt.toUpperCase()%> */ public <%TypeAtt%> get<%NameAtt.toU1Case()%>() { return <%NameAtt.toLowerCase()%>; } /** * Permite asignar <%NameAtt.toUpperCase()%> * * @param <%NameAtt.toUpperCase()%> */ public void set<%NameAtt.toU1Case()%>(<%TypeAtt%> <%NameAtt.toLowerCase()%>) { logger.info("\n ** set <%NameAtt.toUpperCase()%> "+ <%NameAtt.toLowerCase()%>); this.<%NameAtt.toLowerCase()%> = <%NameAtt.toLowerCase()%>; }
35. Por cada atributo del componente de entidad se debe definir la propiedad correspondiente en el archivo de mapeo de hibérnate **(entidadHBM.mt).**	**Esta especificado dentro del archivo generador de archivos de mapeo hbm.xml** <hibernate-mapping> <class name="mx.izt.uam.is.proyecto.dominio.<%NameComponente%>" table="<%NameComponente%>"> <%for (containsAttribute){%> <%if (keyAtt.equalsIgnoreCase("true")){%> <!-- Identificador unico de la clase --> <id name="<%NameAtt.toLowerCase()%>" column="<%NameAtt.toLowerCase()%>" type="int"> <!-- El valor de increment siginifica que el valor inicial es 1 y se incrementa en 1 cada vez que se agrega un registro --> <generator class="increment"/> </id> <%}else{%> <!-- <%NameAtt.toUpperCase()%> --> <property name="<%NameAtt.toLowerCase()%>" type="<%TypeAtt.toLowerCase()%>"/> <%}%> <%}%> </class> </hibernate-mapping>

Tabla 25. Definición de reglas de transformación del componente Entity y sus atributos en código.

De la misma manera en que fueron definidas las reglas de disponibilidad y desempeño, es posible definir el resto de las reglas de transformación para las tácticas de atributos de calidad identificadas en este proyecto sobre la plataforma de Spring. El sub conjunto de reglas de transformación definidas y la herramienta de generación construida, es posible

identificar si los objetivos de este proyecto han sido alcanzados. En la siguiente sección se describe la validación de la herramienta generada hasta ahora. Es decir, mediante un proceso de validación se obtiene mediciones del comportamiento de la herramienta.

4.2 Validación de la herramienta.

La herramienta construida en este proyecto permite obtener, a partir de un modelo arquitectónico, un esqueleto de arquitectura ejecutable. Este es justamente el objetivo de este proyecto de investigación. Sin embargo, otro objetivo implícito es el facilitar y reducir el tiempo de desarrollo de una arquitectura. Por lo tanto, la validación de la herramienta construida en este proyecto de investigación tiene como objetivo corroborar que la herramienta permite agilizar el desarrollo de un esqueleto de una arquitectura ejecutable. Es decir, se busca cuantificar, bajo ciertos parámetros, la manera en que la herramienta reduce el tiempo de desarrollo del esqueleto arquitectónico. Para lograr esto se consideran tres factores de medición del trabajo de generación de código de la herramienta. Estos tres factores son:

- **Número de archivos generados.** Este dato permitirá tener un registro del número de archivos generados a partir del modelo PSM y sus elementos.

- **Porcentaje de código generado de un componente de software (archivo).** Al ser una herramienta que genera esqueletos, no se espera que genere la implementación específica de cada aplicación. En este caso se busca identificar y clasificar los distintos tipos de archivos que son generados así como un porcentaje aproximado de código generado en cada uno de ellos.

- **Tiempo de codificación manual.** Con este factor de medición se busca cuantificar el tiempo de desarrollo que puede ahorrarse si se utiliza la herramienta para la construcción del esqueleto arquitectónico en vez de realizar la codificación de este de manera manual. Como factor de medición implícito dentro del tiempo de codificación, se encuentra el número de líneas físicas (líneas codificadas de acuerdo a los estándares de Java [36], sin contabilizar líneas en blanco) en cada archivo. Con este factor de medición es posible registrar el tiempo de codificación en cada uno de los archivos generados a partir del modelo.

Una vez que los parámetros de validación de la herramienta han sido establecidos, es posible aplicar dicha validación mediante la construcción de una aplicación de ejemplo, de la cual se obtendrán los valores de estos parámetros para así llegar a una conclusión con respecto a la herramienta desarrollada. El ejemplo que se utilizó se presenta en la siguiente sección.

4.2.1 Ejemplo de uso de herramienta.

La aplicación que servirá para ejemplificar y poder obtener las mediciones necesarias del proceso de validación, consiste en un único Caso de Uso (CU). El dominio del problema de la aplicación concierne al manejo administrativo de una biblioteca. Específicamente, el caso de uso que será implementado es el alta de un libro. Los componentes de

software que son necesarios para construir el CU de la aplicación son pocos en comparación de una aplicación completa, pero son suficientes para obtener las mediciones necesarias para la validación. La Figura 32 muestra el PSM que es elaborado para representar la estructuración de la aplicación para el caso de uso alta de libro.

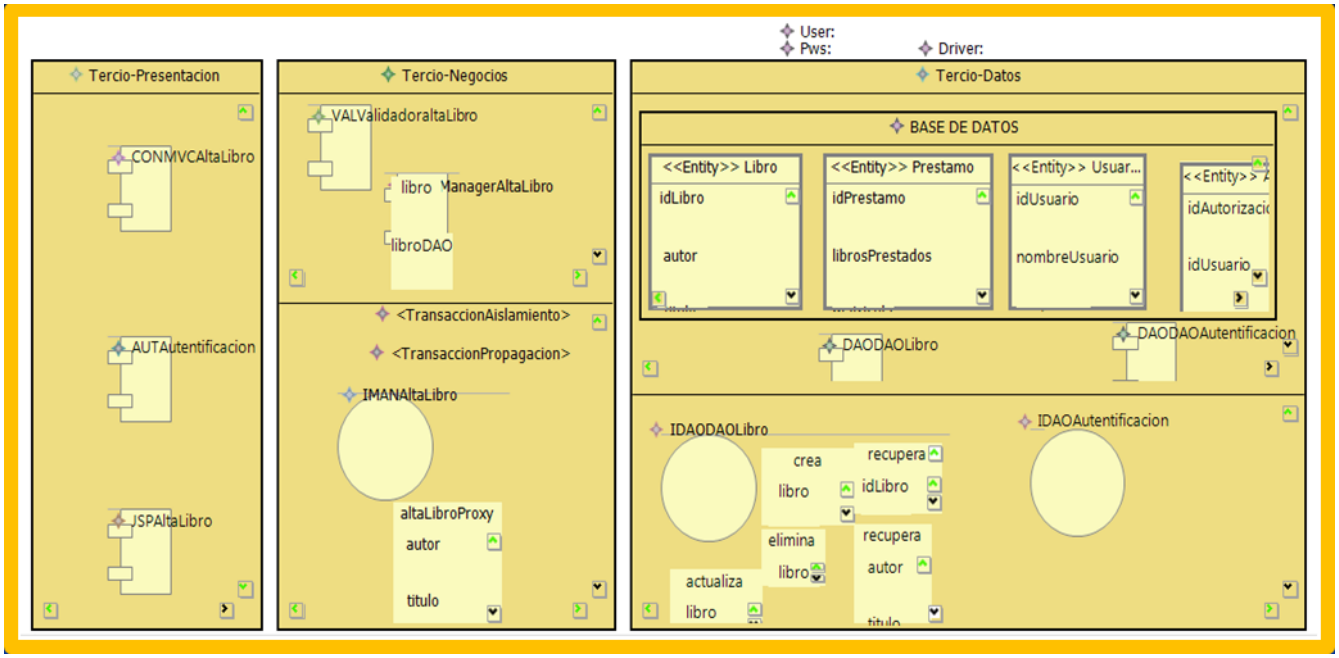

Figura 32. PSM para el CU de Alta de Libro.

Como se puede observar del diagrama de la Figura 32, este contiene los elementos de modelado de presentación (JSP, Autentificación, Controlador), negocios (Validador, Manager del CU de Alta de Libro, Interfaz del Manager) y datos (Base de datos con elementos de dominio-entidad, DAOs de acceso y sus interfaces). Este diagrama no muestra las relaciones entre los componentes de software para hacerlo más comprensible. Esto es posible ya que el editor permite especificar estas relaciones en las propiedades de los componentes.

Como se puede observar del modelo, los elementos de dominio que fueron considerados únicamente corresponden a los elementos necesarios para el CU, en este caso la entidad de libro, así como los elementos de dominio que son utilizados para la autentificación de usuarios del sistema.

4.2.2 Evaluación y resultados.

El proceso de evaluación consiste principalmente en obtener las mediciones de los indicadores establecidos previamente, obteniendo como resultado el tiempo de desarrollo y trabajo de codificación que puede ser ahorrado, a partir de un modelo específico de plataforma con un número determinado de elementos, al utilizar la herramienta desarrollada en este proyecto para la construcción de esqueletos arquitecturales.

Aunque el proceso en el cual se elabora un modelo específico de plataforma puede ser considerado equivalente a actividades del modelado y diseño de la arquitectura de software para una aplicación. En este proceso de evaluación se registrará el tiempo de elaboración del PSM para un CU específico con un número de componentes determinado. En la Tabla 26 se enlistan los componentes utilizados para modelar el caso de uso de alta libro, así como el número de componentes de cada tipo utilizado y sus propiedades principales que describen a cada uno (métodos y atributos).

Elemento del modelo.	Número de componentes modelados.	Propiedades configurables del componente del modelo.
Paquete Presentación	1	1 propiedad configurable (nombre del paquete).
Paquete Negocios	1	1 propiedad configurable (nombre del paquete).
Paquete Datos	1	1 propiedad configurable (nombre del paquete).
JSP	1	2 propiedades configurables (nombre y componente con el que se asocia).
Autentificador	1	3 propiedades configurables (nombre, componentes asociados e interfaz de DAO que implementa para el acceso a la BD para la autentificación)
Controlador	1	3 propiedades configurables (nombre, componentes asociados e interfaz de Manager que utiliza para realizar lógica de negocios).
Validador	1	2 propiedades configurables (nombre y componentes asociados).
Interfaz de Manager	1	2 propiedades configurables (nombre y componente que implementa la interfaz). 1 definición de método declarado dentro de la interfaz de manager Alta Libro.
Manager	1	5 propiedades configurables (nombre, interfaz que implementa, interfaz de DAO que usa, nivel de propagación y nivel de aislamiento). 2 atributos declarados dentro del componente para el manejo del dominio del problema.
Interfaz de DAO	2	2 propiedades configurables (nombre y componente que implementa la interfaz). Interfaz DAOLibro. 5 definiciones de métodos. Interfaz DAOAutentificacion**.** 1 definición de método.
DAO	2	3 propiedades configurables (nombre, interfaz que implementa y componente de entidad con el cual está asociado).
Base de datos	1	9 propiedades configurables (véase valores etiquetados del componente Base Datos).
Entidad	4	2 propiedades configurables (nombre y componentes de entidad asociados). Cada componente tiene un número diferente de atributos definidos, esto depende directamente de la entidad que represente.

Tabla 26. Componentes y propiedades para modelar el CU Alta Libro.

Como se puede observar de la Tabla 26 y de la Figura 31, para modelar el caso de uso de Alta Libro se utilizaron 18 componentes. El tiempo registrado para elaborar el modelo con todos los componentes y cada una de sus propiedades del CU fue de aproximadamente 40 minutos (para alguien con experiencia en el uso de la herramienta). Sin embargo, el proceso de modelado de un CU puede ser realizado utilizando otras herramientas y metodologías de desarrollo. Es decir, aunque no se utilice la herramienta desarrollada en este proyecto de investigación, en el proceso de desarrollo de software se realizan tareas con objetivos similares, documentar la arquitectura de software y modelar un CU. Por tal motivo el tiempo registrado para elaborar el modelo de un CU únicamente es medido con el objetivo de registrar el tiempo de dicha actividad, ya que incluso este tiempo puede variar dependiendo directamente de la complejidad del CU.

En las siguientes secciones se reportan los resultados de las mediciones registradas de los parámetros de medición establecidos: porcentaje código generado de un componente, número de componentes generados de una aplicación y tiempo de codificación manual.

4.2.2.1 Número de archivos generados.

Este factor de medición contabiliza el número de archivos que son generados a partir de un modelo específico de plataforma para un CU de una aplicación. En la Tabla 27 se enlistan los archivos generados, así como los elementos del modelo que son utilizados para la generación de estos.

Elemento del modelo	Elemento del PSM del CU.	Archivos generados	Definiciones generadas.
JSP	JSPAltaLibro	JSPAltalibro.jsp	Entrada en el Bean urlMapping dentro del *application context*.
Autentificador	AUTAutentificacion		Beans de autentificación y su manejo
Controlador	CONMVCAltaLibro	MVCAltaLibro.java	Bean mvcAltalibro dentro del *application context*
Validador	VALValidadorAltaLibro	ValidadorAltaLibro.java	Bean validadorAltaLibro dentro del *application context*.
Interfaz de Manager	IMANAltaLibro	InterfazAltaLibro.java ProxyAltaLibro.java	Bean proxyAltalibro
Manager	MANAltaLibro	ManagerAltaLibroImpl.java	• Bean ManagerAltaLibroService • Bean ManagerAltaLibroTarget • Bean TransactionManager
Interfaz de DAO	IDAOLibro	InterfazDAOLibro.java	

	IDAOAutentificacion	InterfazDAOAutentificacion.java	
DAO	DAOLibro	DAOLibro.java	Bean daoLibro
	DAOAutentificacion	DAOAutentificacion.java	• Beans daoAuthentificationProvider • Bean AuthentificationDAO • Bean daoAutentificacion.
Base Datos	Base de datos		• Bean hibernateProperties • Bean dataSource
Entidad	Libro	• Libro.java • Libro.hbm.xml	• Bean Libro de dominio • Definición del archivo de mapeo en el bean de sessionFactory
	Prestamo	• Prestamo.java • Prestamo.hbm.xml	• Bean Prestamo de dominio • Definición del archivo de mapeo en el bean de sessionFactory
	Usuario	• Usuario.java • Usuario. hbm.xml	• Bean Usuario de dominio • Definición del archivo de mapeo en el bean de sessionFactory
	Autorizacion	• Autorizacion.java • Autorizacion.hbm.xml	• Bean Autorizacion de dominio • Definición del archivo de mapeo en el bean de sessionFactory

Tabla 27. Archivos y definiciones generados a partir del PSM del CU Alta Libro.

De la Tabla 27 se puede observar que son 19 (18 enlistados en la tercera columna y el *application context* donde están las definiciones de los beans) los archivos generados directamente de los componentes modelados en el PSM, estos archivos están enlistados en la última columna, así como también dentro de esta columna se encuentran enlistadas las definiciones de los beans correspondientes que deben ser definidos dentro del *application context*, contando este ultimo como un único archivo que es generado a partir de la mayoría de los elementos del PSM.

4.2.2.2 Porcentaje de código generado de un componente de software.

Este factor de medición puede variar considerablemente, ya que la complejidad y funcionalidad en cada CU puede variar de acuerdo al tipo de la aplicación que se esté desarrollando. Sin embargo, considerando las restricciones de que la herramienta únicamente nos proporciona un esqueleto arquitectónico, se han identificado tres categorías en las cuales se generaliza el porcentaje de código generado. Estas categorías, principalmente, engloban los archivos que son generados

por completo, aquellos en los cuales únicamente se generan las definiciones de métodos y atributos, y por último aquellos en los cuales únicamente se genera el cuerpo principal del archivo. La Tabla 28 muestra los tres rangos de porcentaje que serán considerados para evaluar la generación de archivos.

Porcentaje de código.	Descripción del archivo o elemento.
10%	Este tipo de archivos o definiciones de elementos están muy ligados a la implementación de la aplicación y por tal motivo únicamente contiene el cuerpo de la definición del archivo de código. Ejemplo de esto son las clases de negocio y archivos de presentación.
40%	Este tipo de archivo contiene únicamente la definición de los métodos, pero no así la implementación del mismo, y atributos especificados dentro del elemento de modelado correspondiente. Ejemplo de esto son las interfaces.
100%	Este tipo de archivos o definiciones de elementos contienen la generación completa, en la mayoría de los casos, de código de acuerdo con lo modelado en el PSM. Por el tipo de archivo es posible generar métodos con implementación. Ejemplo de esto son las clases de dominio.

Tabla 28. Generalización de porcentajes de código a considerar.

Una vez establecidos los porcentajes que serán considerados para catalogar los archivos y secciones de código generados, para seguir con el proceso de evaluación de la herramienta, es necesario cuantificar los archivos y secciones de código generados para el PSM del caso de uso de Alta Libro. La Tabla 29 muestra el porcentaje de código generado por cada elemento del PSM.

Elemento del modelo específico de plataforma.	Archivos generados partir de un o mas elementos del PSM.	Porcentaje de código generado (observaciones).
JSPAltaLibro	JSPAltalibro.jsp	10% Se considera que es generado el 10% de código debido a que únicamente se genera las etiquetas del cuerpo de un JSP, debido a que la estructuración de los elementos dentro de un componente de este tipo es diferente en cada uno de ellos.
	Entrada en el Bean urlMapping dentro del *application context*	10%. Únicamente es definido el bean que contiene la especificación de cada JSP sin detalles de manejo de eventos o enlaces con otros JSP.
AUTAutentificacion	Beans de autentificación y su manejo.	100%. Es generado el bean que contiene la definición de los elementos de autentificación.
CONMVCAltaLibro	MVCAltaLibro.java	10%. Únicamente se genera la clase java con el método de manejo de solicitudes, este método no contiene ninguna implementación.

	Bean mvcAltalibro dentro del *application context.*	100%. Se genera la definición del bean del controlador dentro del *application context*.
VALValidadorAltaLibro	ValidadorAltaLibro.java	10 %. Únicamente se genera el archivo con la definición de la clase principal, debido a que esta clase realiza procesos de validación de información o datos de acuerdo con la funcionalidad del CU en específico.
	Bean validadorAltaLibro dentro del *application context.*	100%. Se realiza la definición del bean dentro del *application context.*
IMANAltaLibro	InterfazAltaLibro.java	100%. Se genera la interfaz Java con la definición de todos los métodos modelados en el elemento del PSM.
	ProxyAltaLibro.java	40%. Únicamente son generados lo métodos y atributos definidos en la interfaz.
	Bean proxyAltalibro	100%. Se genera la definición del bean proxy por cada elemento interfaz modelado en el PSM.
MANAltaLibro	ManagerAltaLibroImpl.java	40%. Contiene la definición de los métodos declarados en el elemento de modelado de interfaz, así como también los atributos declarados en el manager y los métodos correspondientes para operar sobre dichos atributos y mantener así la encapsulación.
	Bean ManagerAltaLibroService	100%. Es generada la definición del bean correspondiente.
	Bean managerAltaLibroTarget	100%. Es generada la definición del bean correspondiente.
	Bean transactionManager	100%. Es generada la definición del bean correspondiente.
IDAOLibro	InterfazDAOLibro.java	100%. Se genera la definición de la interfaz DAO para cada uno de los elementos definidos dentro del modelo de este tipo.
IDAOAutentificacion	InterfazDAOAutentificacion.java	100%. Se genera la definición de la interfaz DAO para cada uno de los elementos definidos dentro del modelo de este tipo.
DAOLibro	DAOLibro.java	40%. Se genera la definición de la clase y sus métodos, pero no así la implementación de los mismos.
	Bean daoLibro	100%. Se genera la definición del bean correspondiente dentro del *application context.*
DAOAutentificacion	DAOAutentificacion.java	40%. Se genera la definición de la clase y sus métodos, pero no así la implementación de los mismos.
	Bean daoAuthentificationProvider	100%. Es generada la definición del bean correspondiente.
	Bean authentificationDAO	100%. Es generada la definición del bean correspondiente.
	Bean daoAutentificacion	100%. Es generada la definición del bean correspondiente.

Base de datos	Bean hibernateProperties	100%. Es generada la definición del bean correspondiente con las propiedades correspondientes de la base de datos.
	Bean dataSource	100%. Es generada la definición del bean correspondiente con las propiedades correspondientes de la base de datos.
Libro	Libro.java	100%. Es generado el archivo por completo con atributos y métodos para utilizar el componente de dominio.
	Libro.hbm.xml	100%. Se genera por completo el archivo de mapeo de acuerdo a las propiedades del elemento del PSM.
	Bean Libro de dominio	100%. Es generada la definición del bean correspondiente.
	Definición del archivo de mapeo en el bean de sessionFactory	100%. Se genera la entrada correspondiente de cada archivo de mapeo dentro del bean de sessionFactory.
Prestamo	Prestamo.java	100%. Es generado el archivo por completo con atributos y métodos para utilizar el componente de dominio.
	Prestamo.hbm.xml	100%. Se genera por completo el archivo de mapeo de acuerdo a las propiedades del elemento del PSM.
	Bean Prestamo de dominio	100%. Es generada la definición del bean correspondiente.
	Definición del archivo de mapeo en el bean de sessionFactory	100%. Se genera la entrada correspondiente de cada archivo de mapeo dentro del bean de sessionFactory.
Usuario	Usuario.java	100%. Es generado el archivo por completo con atributos y métodos para utilizar el componente de dominio.
	Usuario. hbm.xml	100%. Se genera por completo el archivo de mapeo de acuerdo a las propiedades del elemento del PSM.
	Bean Usuario de dominio	100%. Es generada la definición del bean correspondiente.
	Definición del archivo de mapeo en el bean de sessionFactory	100%. Se genera la entrada correspondiente de cada archivo de mapeo dentro del bean de sessionFactory.
Autorizacion	Autorizacion.java	100%. Es generado el archivo por completo con atributos y métodos para utilizar el componente de dominio.
	Autorizacion.hbm.xml	100%. Se genera por completo el archivo de mapeo de acuerdo a las propiedades del elemento del PSM.
	Bean Autorizacion de dominio	100%. Es generada la definición del bean correspondiente.
	Definición del archivo de mapeo en el bean de sessionFactory	100%. Se genera la entrada correspondiente de cada archivo de mapeo dentro del bean de sessionFactory.

Tabla 29. Porcentaje de código generado en cada archivo y sección de código.

De los porcentajes mostrados en la Tabla 29 se puede observar que los archivos generados a partir de elementos de presentación, tales como el JSP y Controlador, únicamente contiene un 10% de código generado, esto se debe a que son elementos de modelado en los cuales no se agregaron valores etiquetados para el manejo de tácticas de atributos de

calidad. Por otra parte estos elementos generan componentes de vista que puede ser implementado de varias formas y no es considerado dentro del alcance de este proyecto la generación de código de este tipo (únicamente parte del esqueleto de arquitectura ejecutable).

En cuanto a los componentes de software de negocios: Validador, Proxy, Manager, y DAOs; en la mayoría de los casos, estos componentes contienen un aproximado de 40% de código generado. Esto se debe principalmente a que este tipo de elementos contiene valores etiquetados que permiten dar soporte a tácticas de calidad así como a propiedades características de este tipo de componentes, entre otras. Es decir, con las propiedades configurables definidas dentro de cada elemento es posible generar un mayor porcentaje de código, pero no así detalles de implementación.

Por último los componentes generados a partir de los elementos de Base de Datos y entidad, son generados al 100% aproximadamente, esto se debe principalmente a que estos componentes del modelo contiene los valores etiquetados necesarios para la configuración de las propiedades de cada uno de los componentes de software generados, tales como archivos de configuración de la base de datos (beans dentro del *application context*) y archivos de dominio y mapeo.

De la medición de este factor de evaluación, se llega a la conclusión que los archivos que contienen un porcentaje mayor de código generado son aquellos que son generados a partir de elementos del modelo que contiene un mayor número de valores etiquetados definidos dentro de las propiedades de los elementos de modelo, esto se debe también a que estos archivos no contienen implementación específica de la aplicación (lógica de negocio). Por otra parte los archivos generados que presentan un porcentaje de código menor son generados a partir de elementos del modelo que representan componentes que deben implementar funcionalidades específicas dentro del sistema que se está desarrollando. Debido principalmente a que este proyecto se enfoca en modelar tácticas de calidad, que están clasificadas como requerimientos no funcionales, la agregación de valores etiquetados para el soporte de requerimientos funcionales esta fuera de los objetivos de este proyecto.

4.2.2.3 Tiempo de codificación manual.

Para cuantificar este factor de medición establecido se realizó la codificación manual de los archivos generados por la herramienta, es decir, se realizó la codificación del mismo número de líneas en cada archivo. Con esto se obtuvo un tiempo de codificación manual para construir el mismo esqueleto arquitectónico que la herramienta proporciona de manera automática. Este tiempo de codificación puede ser minimizado durante el proceso de desarrollo, específicamente durante la elaboración de la arquitectura ejecutable, si se utiliza la herramienta construida en este proyecto de investigación. Para poder considerar la medición de este factor, se necesita contabilizar el número de líneas físicas de cada archivo y a su vez el tiempo de codificación de cada uno de los archivos. La Tabla 30 justamente muestra estos factores de medición, número de líneas de cada archivo y el tiempo de codificación de cada archivo.

Tipo de Archivo	Archivos	Numero de líneas físicas	Tiempo de codificación (minutos)
java	MVCAltaLibro.java	15	4
	ValidadorAltaLibro.java	9	3
	InterfazAltaLibro.java	6	1
	ProxyAltaLibro.java	60	15
	ManagerAltaLibroImpl.java	32	7
	Autorizacion.java	32	5
	Libro.java	41	9
	Prestamo.java	47	10
	Usuarios.java	41	9
	DAOAutentificacion.java	7	2
	DAOLibro.java	26	5
	FabricaDAO.java	19	4
	HibernateUtil.java	72	18
	InterfazDAOAutentificacion.java	5	1
	InterfazDAOLibro.java	10	2
xml	webApplicationContext-servlet.xml	212	60
hbm.xml	Autorización.hbm.xml	18	4
	Libro.hbm.xml	20	4
	Prestamo.hbm.xml	22	4
	Usuarios.hbm.xml	20	4
jsp	JSPAltaLibro.jsp	11	2
	index.jsp	3	1

Tabla 30. Número de líneas y tiempos de codificación por archivo.

Como se puede observar, de la Tabla 30, el número de archivos generados es de 23, al codificar todos estos archivos se contabilizaron 728 líneas físicas y el tiempo de codificación de estos archivos fue de 174 minutos. Estos resultados son mostrados en la Tabla 31.

Numero total de archivos.	Numero total de líneas físicas.	Tiempo total de codificación (minutos).
23	728	174

Tabla 31. Tiempo de codificación.

Por lo tanto se puede concluir que durante el proceso de desarrollo del CU de Alta Libro, utilizando esta herramienta para la generación de un esqueleto de arquitectura ejecutable se puede generar 23 archivos con aproximadamente 728 líneas físicas de código de manera automática, minimizando el tiempo de codificación de estos archivos en aproximadamente 174 minutos (2 horas 54 minutos). Aunque para realizar una buena evaluación de la herramienta se deberían de realizar varias mediciones en el desarrollo de diferentes CU, esta única medición nos dá una idea de la ventaja de utilizar la herramienta generada en este proyecto. El principal factor por el cual únicamente se realizo una medición fue la limitante de tiempo de desarrollo del proyecto de investigación, el cual no debe exceder de tres trimestres.

4.3 Integración con el proyecto "Herramienta para el desarrollo de arquitecturas de software usando un enfoque MDA".

Como se mencionó anteriormente el enfoque MDA consiste en la utilización de modelos para especificar una solución a un proceso de software y a través de transformaciones hacia modelos más detalladas y específicos obtener una solución a dicho proceso. Este enfoque se caracteriza por la generación de cuatro modelos: CIM, PIM, PSM y código. Este proyecto de investigación está enfocado en modelar el soporte de tácticas de calidad y como se ha explicado en secciones anteriores, esto solo es posible al nivel del PSM, ya que este modelo permite especificar tácticas para una plataforma específica. Por otra parte, se encuentra en desarrollo el proyecto de tesis: "Herramientas para el desarrollo de arquitecturas de software usando un enfoque MDA" [31]. Este proyecto, como su nombre lo indica, utiliza el enfoque MDA para el desarrollo de arquitecturas, específicamente, se enfoca en el trabajo de modelado de los requerimientos funcionales sobre los modelos CIM y PIM. Estos modelos permiten representar una solución del dominio del problema con características de negocio, sin apegarse aun a una tecnología en específico (véase sección 2.3.1).

Retomando ambos proyectos, implícitamente cada uno de ellos se enfoca en el modelado de requerimientos de software (funcionales y no funcionales) para la generación de una arquitectura ejecutable parcial, con el objetivo de agilizar las actividades del proceso de desarrollo en las cuales se tiene que generar una arquitectura ejecutable, así como validar que dicha arquitectura es la adecuada para el sistema que se esté desarrollando. Es decir, ambos proyectos sientan las bases para la generación de dos herramientas que en conjunto agilizarían actividades de documentación, diseño y construcción de una arquitectura ejecutable.

5 Discusión crítica.

En este capítulo se presenta la discusión crítica de este proyecto, la cual consiste en explicar y justificar el presente trabajo de tesis. Específicamente se justificarán y analizarán las decisiones de implementación, así como también se describirán los factores que limitaron aspectos en desarrollo del proyecto de investigación.

- **Propósito.** Como punto de partida de esta discusión se consideran los objetivos generales de este proyecto, el porqué de este trabajo de investigación. Dentro de la industria del software ha existido una tendencia a desarrollar herramientas que agilicen y faciliten diversas tareas dentro del desarrollo de sistemas. Una de las arreas que han sido poco explotadas por dichas herramientas es la actividad en la cual se construye una arquitectura ejecutable (un prototipo), con la cual se puede probar las funcionalidades del sistema para validar que la arquitectura diseñada es la adecuada para continuar la construcción del sistema. Este proyecto de investigación define un proceso el cual puede ser utilizado para desarrollar una herramienta con tal propósito, sin embargo por cuestiones de tiempo y considerando el grado de complejidad que requiere generar una arquitectura ejecutable completa, requerimientos funcionales y no funcionales, esta tesis solo se enfoca en generar un esqueleto de arquitectura ejecutable con soporte para la parte no funcional (restricciones y atributos de calidad). Este proceso fue empleado para la construcción de una herramienta la cual emplea un conjunto de patrones y tácticas de atributos de calidad. Sin embargo, siguiendo el mismo proceso puede ser empleado otro conjunto diferente de patrones y tácticas de atributos de calidad, así como distintos *Frameworks* de desarrollo que implementan las tácticas de atributos de calidad.

- **Selección de patrones.** La selección de patrones arquitectónicos fue realizada considerando los patrones más utilizados y representativos en el desarrollo de aplicaciones empresariales. Los patrones cliente servidor, n-tercios, capa de acceso a la base de datos y MVC son algunos de los más empleados en el ámbito laboral y académico. Como se mencionó anteriormente esta selección de patrones puede cambiar de acuerdo las especificaciones del sistema.

- **Selección de plataforma.** Aunque ambos *Frameworks* presentados en esta tesis permiten el soporte de las mismas tácticas de atributos de calidad, la selección de *Spring* se debe principalmente a la manera que este *Framework* permite soportar dichas tácticas. Es decir, este *Framework* permite realizar la declaración de tácticas de atributos de calidad de manera declarativa, permitiendo así facilitar la definición de las reglas de transformación para la generación de código fuente.

- **Selección de atributos de calidad.** Se optó por tomar la definición y manejo de atributos de calidad que propone el SEI, ya que realmente existen muy pocas alternativas que especifiquen explícitamente tanto la definición de los atributos de calidad como la manera de controlar dichos atributos. El SEI al definir una manera de explícita del manejo de atributos de calidad mediante tácticas y documentar dichas tácticas utilizando escenarios, facilita la

labor de identificar como es que los *Frameworks* de desarrollo de hoy en día implementan dichas tácticas para controlar la respuesta de un atributo de calidad especifico.

- **Selección de tácticas de atributos de calidad.** La selección de tácticas realizada es un sub conjunto de las tácticas soportadas por el *Framework* de desarrollo seleccionado (*Spring*). El principal factor por el cual se redujo el conjunto de tácticas a implementar en este trabajo de tesis, es el tiempo con el que se dispone para la elaboración del proyecto. El segundo factor que se consideró en la selección de tácticas, son los atributos de calidad. Es decir, se seleccionaron tácticas de cada uno de los atributos de calidad, definidos por el SEI, que pudieran ser implementadas por el *Framework*, en su mayoría de manera declarativa.

- **Atributos de calidad que se contraponen (modificabilidad/desempeño).** El manejo y decisión de implementar un conjunto especifico de tácticas de atributos de calidad es un tema muy importante en el desarrollo de software; ya que es una decisión que afectará el trabajo de todos los involucrados en el desarrollo, pero más importante aún de la aplicación. Ya que existen factores que deben considerarse al implementar algunas de las tácticas de atributos de calidad, uno de estos factores es la medida en que afecta o dificulta el soporte de los otros atributos de calidad. Un ejemplo de esto es la relación que existe en implementar sistemas de prevención de fallas (disponibilidad) o sistemas específicos de detección de intrusos (seguridad) con el desempeño del sistema, ya que al implementar estas dos tácticas el desempeño del sistema se puede ver afectado drásticamente. Este trabajo de tesis muestra la implementación de tácticas que afectan en poca medida el desempeño, pero la decisión de que tácticas implementar puede variar de acuerdo a las necesidades de cada sistema a desarrollar.

- **Tácticas empleadas que tienen traslapes a través de los mecanismos empleados.** Algunas de las tácticas modeladas en el PSM pueden ser consideradas tácticas relacionadas a más de una categoría de atributo de calidad, ejemplo de estas tácticas son las referentes a los atributos de calidad de modificabilidad, facilidad de pruebas y usabilidad. Ya que estas tácticas comparten soluciones a problemas muy similares, utilizando mecanismos de programación tales como separación de responsabilidades y congruencia semántica. Es decir, estas tácticas son modeladas en el PSM como una solución a tres diferentes atributos de calidad. La selección de estas tácticas fue considerando que son soluciones estructurales básicas que la mayoría de los sistemas de hoy en día presenta.

- **Justificar el uso de la herramienta.** Para justificar el uso del proceso y a su vez de la herramienta generada se debe corroborar que la herramienta minimiza el tiempo en construir un esqueleto de arquitectura ejecutable. Únicamente se utiliza un ejemplo para cuantificar tiempos, ya que se considera que no es necesario realizar el mismo estudio varias veces, ya que el objetivo de esta evaluación era demostrar que la herramienta generada facilita el desarrollo de sistemas durante la transición entre las fases de elaboración y construcción de RUP. El factor principal por el que no se considero necesario elaborar una evaluación con más ejemplos es debido a que en el desarrollo de aplicación, incluso cada uno de los casos de de uso definidos para la construcción de un sistema

tiene un grado de complejidad diferente y por ende un tiempo de desarrollo diferente. Es decir, el realizar un adecuado estudio sobre lo factible que es implementar el uso de la herramienta generada durante el desarrollo de un sistema, es muy complejo y por cuestiones de tiempo queda fuera de los objetivos de este trabajo de tesis. Por lo tanto, únicamente se verificó que el uso de la herramienta generada ayuda y facilita en cierta medida el desarrollo de sistemas.

- **Aportación del trabajo.** La aportación principal de este trabajo son las bases para la generación de una herramienta que permite modelar una arquitectura para una plataforma específica, considerando una selección previa de patrones de diseño así como de tácticas de atributos de calidad que el sistema soportará. Este mismo proceso puede ser utilizado como base para la generación de una herramienta que permita modelar una arquitectura de sistema completa y así generar una arquitectura ejecutable que considere tanto la parte funcional como la no funcional.

Considerando principalmente los objetivos generales de este proyecto, el trabajo realizado a lo largo del desarrollo del mismo y de los resultados obtenidos se identificaron los siguientes inconvenientes:

- La curva de aprendizaje de los *Frameworks* necesarios para la construcción requirió de mayor tiempo del considerado inicialmente, lo cual propició que el proyecto se prolongara a más de tres trimestres.

- Faltaron por implementar reglas de transformación de las tácticas de seguridad definidas. Esto se debe principalmente a las limitantes de tiempo que tiene el proyecto así como también que los *Frameworks* requieren de mayor tiempo para su aprendizaje.

- Aunque la herramienta generada puede ser utilizada para diseñar un modelo de arquitectura, también se es consciente que dentro de las organizaciones se hace uso de metodologías y herramientas específicas para el diseño de una aplicación y que el uso de esta herramienta pudiera llegar a incrementar el tiempo de desarrollo en esta fase. Aun considerando este factor, el uso de la herramienta es justificado por la notoria reducción de tiempo en la fase de construcción.

En la sección de conclusiones y perspectivas se hace mención del análisis realizado al finalizar el proyecto y se identifica cuales de los objetivos del proyecto fueron alcanzados así como el rubro en el cual se le ve futuro a este proyecto de investigación.

6 Conclusiones y perspectivas.

6.1 Conclusiones.

Realizando un análisis del los objetivos generales de este proyecto de investigación, así como del trabajo realizado a lo largo del mismo y los resultados obtenidos, se llega a las siguientes conclusiones:

- Se logró definir un proceso que sigue el enfoque MDA, el cual permite modelar y generar esqueletos de arquitecturas ejecutables con soporte de tácticas de calidad sobre una plataforma específica (Spring).

- Se construyó una herramienta que permite la generación de esqueletos de arquitecturas ejecutables con soporte de un conjunto de tácticas de calidad.

- La herramienta construida fue probada como herramienta de apoyo en el desarrollo de un Caso de Uso (CU), reduciendo el tiempo en la fase de construcción, pues esta herramienta genera gran parte del código concerniente a requerimientos no funcionales, específicamente tácticas de atributos de calidad.

- Se logró adquirir una mayor visión y conocimientos de la importancia de la arquitectura de software, de cada uno de los artefactos que son necesarios para representarla y poder así comenzar la construcción de un sistema.

6.2 Perspectivas.

Dentro del trabajo realizado en este proyecto de investigación se establece el proceso que sienta las bases a seguir para incorporar un conjunto mayor de tácticas de calidad (definidas por el SEI) del mismo *Framework* de desarrollo seleccionado o cualquier otro *Framework* que sea utilizado en el desarrollo de sistemas de software. Este proceso toma como punto de partida el modelo específico de plataforma (PSM) definido dentro del enfoque MDA y únicamente considera la generación del código específico de soporte de un conjunto de tácticas de calidad. Pero este mismo proceso puede ser empleado para modelar características funcionales en alguna de las otras etapas de la metodología MDA y así poder generar un esqueleto más completo de la arquitectura ejecutable.

Por lo tanto con la generación de herramientas de este tipo (MDA) se busca agilizar el proceso de desarrollo de las aplicaciones de software en las fases de diseño y construcción, para poder cumplir con la entrega de sistemas con calidad en los plazos acordados con el cliente.

7 Referencias.

[1] Philippe Kruchten. *"The Rational Unified Process: An Introduction, Third Edition"*. Tercera Edición, Editorial Addison Wesley, Diciembre 19 de 2003.

[2] Len Bass, Paul Clements, Rick Kazman. *"Software Architecture in Practice, Second Edition"*. Segunda Edición. Editorial Addison Wesley, Abril 11 de 2003.

[3] Jonh C. Georgas, Eric M.Dashofy, y Richard N. Taylor. *"Desarrollo Centrado en la Arquitectura: Un acercamiento diferente a la Ingeniería de Software"*. The ACM Student Magazine. [En línea]. Disponible en: http://www.acm.org/crossroads/español/xrds12.4/arqcentric.html.

[4] Raquel Anaya (Septiembre 27 a Octubre 01 de 2005). *"Especificación y Modelado de Arquitecturas de Software"*. Presentado en XXV salón de Informática "Arquitecturas Empresariales de Software". Bogotá, Colombia. [En línea]. Disponible en: http://www.acis.org.co/fileadmin/Base_de_Conocimiento/XXV_Salon_de_Informatica/ArquitecturasSoftware-RaquelAnaya.ppt.

[5] Carlos Billy Reynoso (Marzo de 2004). *"Introducción a la arquitectura de Software"*. Universidad de Buenos Aires, Argentina. [En línea]. Disponible en: http://www.microsoft.com/spanish/msdn/arquitectura/roadmap_arq/intro.asp.

[6] Karl E. Wiegers (2003). *"Software Requirements, Second Edition"*. Editorial Microsoft Press 2003. [En línea]. *Disponible en:* http://library.books24x7.com

[7] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, Charles B. Weinstock (Diciembre 1995). *"Quality Attributes"*. Software Engineering Institute Carnegie Mellon University Pittsburgh, Pennsylvania 15213. [En línea]. Disponible en: http://www.sei.cmu.edu/pub/documents/95.reports/pdf/tr021.95.pdf.

[8] Rikard Land (Febrero 2002). *"A Brief Survey of Software Architecture. Malardalen University Departament of Computer Engineering"*. Department of Computer Engineering, Malardalen University, Vasteras, Sweden. [En linea]. Disponible en: http://www.mrtc.mdh.se/publications/0381.pdf.

[9] Carlos Reynoso, Nicolás Kicillof (Marzo de 2004). *"De Lenguajes de descripción arquitectónica de software ADL"*. Universidad de Buenos Aires, Argentina. [En línea]. Disponible en: http://www.willydev.net/InsiteCreation/v1.0/descargas/prev/adl.pdf.

[10] Octavio Martín Díaz (Octubre 23 de 2001). *"Una Visión General a los Lenguajes de Descripción Arquitectónica Informe Técnico LSI.2001-01"*. Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, España. [En línea]. Disponible en: http://74.125.95.132/search?q=cache:CTT7Jp6J8BwJ:www.lsi.us.es/docs/informes/LDA-InfoT.ps+Una+Visión+General+a+los+Lenguajes+de+Descripción+Arquitectónica&cd=1&hl=es&ct=clnk&gl=mx.

[11] Grady Booch James Rumbaugh, Ivar Jacobson. "The Unified Modeling Language User Guide". Primera Edición. Editorial Addison Wesley. Octubre 20 de 1998.

[12] Ed Roman, Rima Patel Sriganesh, Gerald Brose. *"Mastering Enterprise JavaBeans: Third Edition".* Tercera Edición. Editorial Wiley Publishing Inc. Enero de 2005.

[13] Inderjeet singh, Beth Stearns, Mark Johnson, and Enterprise Team. *"Designing Enterprise Applications with the J2EE Plataform, Second Edition"*. Segunda Edicion. Editorial Addison Wesley. 2005.

[14] Rod Johnson, Juergen Hoeller. *"Expert One-on-One J2EE Development without EJB"*. Editorial Wiley Publishing Inc. 2004.

[15] Anil Hemrajani. *"Agile Java Development with Spring, Hibernate and Eclipse"*. Editorial Sams. Mayo 9 de 2006.

[16] Antonia María Reina Quintero (Diciembre de 2000). *"Visión General de la Programación Orientada a Aspectos"*. Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, España. [En línea]. Disponible en: http://www.lsi.us.es/docs/informes/aopv3.pdf.

[17] Rob Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu. *"Professional Java Development with the Spring Framework"*. Editorial Wiley Publishing Inc. 2005.

[18] Christopher Van Eenoo, Osama Hylooz, Khaled M. Khan. *"Addressing Non-Functional Properties in Software Architecture using ADL"*. School of Computing and Information Technology University of Western, Australia. [En linea]. Disponible en: http://mercury.it.swin.edu.au/ctg/AWSA05/Papers/vaneenoo-et-al.pdf.

[19] J. Andrés Díaz Pace. *"Arquitectura Cliente Servidor"*, *"Especificación de Arquitecturas"* y *"Lenguajes de Descripción de Arquitecturas (ADLs)"*. ISISTAN- Facultad de Cs. Exactas, UNICEN, Diciembre de 2005.

[20] Ben Alex(2004, 2005, 2006). *"Acegi Security: Reference Documentation 1.0.5"*. Documento de Referencia del *Framework Acegi*. [En línea]. Disponible en: http://www.acegisecurity.orghttp://www.java2s.com/Code/Jar/acegi-security/Downloadacegisecurity105jar.htm/guide/springsecurity.pdf.

[21] Luis Enrique Corredera de Colsa. *"Arquitectura dirigida por modelos para J2ME".* Universidad Pontifica de Salamanca en Madrid, 2007, en línea http://personal.telefonica.terra.es/web/lencorredera/mda_j2me.pdf.

[22] Juaquin Millar and Jishnu Mukerji. *"MDA Guide Version 1.0.1".* Documento de Referencia de MDA. [En línea]. Disponible en: http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[23] Néstor A. Riba Zárate. *"Un enfoque MDA para el desarrollo de aplicaciones basadas en un modelo de componentes orientados a servicios"*. Trabajo de investigación de Tesis. Universidad Autónoma Metropolitana Unidad Iztapalapa, Ciencias Básicas e Ingeniería, Maestría en Ciencias y Tecnologías de la Información. Julio 2007.

[24] Craig Walls, Ryan Breidenbach. *"Spring in Action, Second Edition".* Segunda Edición. Editorial Manning Publications Co. 2005.

[25] Erich Gamma, Richard helm, Ralph Johnson, John Vlissides. *"Design Patterns: Elements of Reusable Object-Oriented Software"*. Primera Edición. Editorial Addison Wesley. Enero 15 de 1995.

[26] Clemens Szyperski, Dominik Gruntz, Stephan Murer. *"Component Software, Beyond Object-Oriented Programming"*. Segunda Edición. Editorial Addison Wesley. 2002.

[27] Juan Bernardo Quintero, Raquel Anaya (Diciembre 2007). *"MDA y el papel de los modelos en el proceso de desarrollo de software"*. Revista EIA, ISSN 1794-1237 Número 8, p.161-146. Escuela de Ingeniería de Antioquia, Medellín , Colombia. [En línea]. Disponible en: http://revista.eia.edu.co/articulos8/Art.10.pdf

[28] Página web del *Framework* GMF http://www.eclipse.org/modeling/gmf/.

[29] Página web del *Framework* EMF http://www.eclipse.org/modeling/emf/.

[30] Página web del *Framework* GET http://www.eclipse.org/gef/.

[31] Gustavo Basurto Páez. *"Herramienta para el desarrollo de arquitecturas de software usando un enfoque MDA"*. Trabajo de investigación de Tesis. Universidad Autónoma Metropolitana Unidad Iztapalapa, Ciencias Básicas e Ingeniería, Maestría en ciencias y Tecnologías de la Información. Sin Publicar.

[32] Página web del SEI http://www.sei.cmu.edu/.

[33] William Crawford, Jonathan Kaplan. *"J2EE Design Patterns".* Editorial O'Reilly. Septiembre 2003.

[34] Rob Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, DArren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Rick Evans (2004, 2005, 2006). *"The Spring Framework - Reference Documentation 2.0.8"*. Documento de Referencia del *Framework Spring*. [En línea]. Disponible en: http://static.springframework.org/spring/docs/2.0.x/reference/index.html.

[35] Christian Bauer, Gavin King. *"Hibernate in Action".* Primera Edición. Editorial Manning Publications Co. 2005.

[36] Scott W. Ambler. *"Writing Robust Java Code – The Ambysoft Inc. Coding Standards for Java"*. Version 17.01d. Enero del 2000.

[37] Frank Buschmann, Kevin Henney, Douglas C. Schmidt. *"Pattern-Oriented, Software Architecture. A Pattern Language for Distributed Computing. Volumen 4"*. Editorial John Wiley & Sons, Ltd. 2007.