Marco de referencia para la recuperación y análisis de vistas arquitectónicas de comportamiento

Marco teórico

Martín Emilio Monroy Ríos, MSc.

Director: José Luis Arciniegas Herrera, PhD.

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Doctorado en Ingeniería Telemática
Línea: Servicios avanzados de telecomunicaciones
Popayán, Octubre de 2016

Contenido

Pág.

1. Introducción 1
2. Arquitectura de software 3
2.1. El concepto 3
2.2. Descripción de la arquitectura 5
2.3. Vistas arquitectónicas 6
2.4. Lenguajes de descripción de arquitectura 8
3. Recuperación de la arquitectura 12
3.1. El proceso de recuperación de la arquitectura 13
3.2. Técnicas de extracción de datos 15
3.3. Técnicas de organización del conocimiento 18
3.4. Técnicas de exploración de la información 19
3.5. Especificación KDM 19
4. Análisis de vistas arquitectónicas 23
4.1. Aspectos relevantes para el análisis arquitectónico 23
4.2. Técnicas de evaluación de la arquitectura 24
5. Herramientas de ingeniería inversa 28
5.1. Caracterización de herramientas 30
5.2. Modelo ontológico para contextos de uso de las herramientas 35
Glosario de Términos 39
Bibliografía 44

Lista de figuras

Pág.

Figura 1. Niveles de abstracción 4
Figura 2. Modelo conceptual de la descripción de la arquitectura. Fuente: ISO/IEC/IEEE 42010:2011 6
Figura 3. Modelo conceptual de un ADL. Fuente: ISO/IEC/IEEE 42010:2011 9
Figura 4. Áreas semánticas de UML. Fuente OMG, 2015... 11
Figura 5. Estructura de los paquetes KDM. Fuente OMG (2011).............................. 20
Figura 6. Modelo ontológico 36

Lista de Tablas

Pág.

Tabla 1. Vistas arquitectónicas. Adaptado de Bass et al. (2013) y Callo Arias et al. (2011) 8
Tabla 2. Estructura de caracterización de herramientas de ingeniería inversa 32
Tabla 3. Relación entre el tipo, el modelo y la representación de la fuente 32
Tabla 4. Propiedades Comunes 34
Tabla 5. Características de los contextos de uso 37

1. Introducción

Este documento forma parte de los resultados obtenidos en la Tesis Doctoral titulada "Marco de referencia para la recuperación y análisis de vistas arquitectónicas de comportamiento". Representa la base teórica y conceptual de ARCo (Architecture Recovery in Context), el esquema interpretativo que simplifica y condensa el campo del conocimiento de la ingeniería inversa, al señalar y codificar selectivamente los objetos, situaciones, acontecimientos, experiencias y las acciones que se producen en un proceso de recuperación y análisis de vistas arquitectónicas de comportamiento.

Tiene como fin presentar, desde la perspectiva epistemológica, la fundamentación teórica que hace posible la representación y explicación del proceso de recuperación y análisis de vistas arquitectónicas de comportamiento, convirtiéndose en un instrumento que facilita la interpretación del Marco de Referencia ARCo. Para lograr este objetivo, el documento se encuentra organizado como se explica a continuación.

En el capítulo dos se hace un análisis sobre el concepto de arquitectura de software, el estándar ISO/IEC/IEEE 42010 para la descripción de arquitecturas de software, las propuestas más relevantes sobre vistas arquitectónicas, y algunos lenguajes de descripción de arquitectura, centrando la atención en el Lenguaje Unificado de Modelado (UML).

En el tercer capítulo se aborda el tema de la recuperación de arquitecturas, analizando la evolución del concepto de ingeniería inversa, explicando el proceso canónico para la recuperación de arquitecturas, y las técnicas utilizadas para cada una de las fases que conforman dicho proceso: La extracción de datos, la organización del conocimiento, y la exploración de la información. Al final del capítulo se explica la especificación KDM (Knowledge Discovery Metamodel), utilizada como meta-modelo para representar los activos que conforman productos software implementados, y que actualmente corresponde al estándar ISO/IEC 19506.

El cuarto capítulo aborda el análisis de arquitecturas, explicando el concepto, los aspecto relevantes para el análisis arquitectónico y las técnicas de evaluación de arquitectura identificadas en la revisión de la literatura. En el último capítulo se centra la atención en las herramientas de ingeniería inversa, presentado el modelo de caracterización de herramientas y el modelo ontológico para contextos de uso propuestos como resultado de la tesis doctoral.

Finalmente se presenta un glosario de términos, como parte constitutiva de la fundamentación teórica del Marco de Referencia ARCo, cuyo fin principal es presentar una base conceptual que garantice una interpretación más acertada del marco de referencia para la recuperación y análisis de vistas arquitectónicas de comportamiento.

2. Arquitectura de software

El instituto de Ingeniería de Software ha identificado más de un centenar de definiciones de arquitectura de software (SEI, 2016), que ha clasificado en tres grupos: Definiciones modernas, definiciones clásicas y definiciones bibliográficas. Esto demuestra que aún existe un gran debate sobre el tema, por eso en esta sección se establecen los conceptos utilizados en el desarrollo de la tesis sobre: arquitectura de software, descripción de la arquitectura, marcos de referencia arquitectónicos y lenguajes de descripción de arquitectura.

2.1. El concepto

Según el estándar ISO/IEC/IEEE 42010:2011 la arquitectura de software es la organización fundamental de un sistema encarnada en sus elementos, las relaciones que existen entre ellos y su entorno, y los principios que rigen su diseño y evolución. En la definición se sustituyó el término componente (utilizado en el estándar ANSI/IEEE 1471:2000) por elemento, porque éste último es más general, mientras que el primero hace referencia específicamente a un elemento que representa una unidad computacional en tiempo de ejecución (Bass et al., 2013).

Esta definición implica que todo producto software tiene una arquitectura, que aborda únicamente decisiones significativas sobre los aspectos estructurales y de comportamiento del sistema. La estructura o parte estática representa la información sobre los elementos del sistema y las relaciones que existen entre ellos, mientras que el comportamiento o parte dinámica comprende la forma como los elementos interactúan en tiempo de ejecución para cumplir las funciones del sistema (Bass et al., 2013).

Una decisión significativa sugiere un proceso de abstracción, en el que se omiten detalles que no afectan la comprensión del sistema, o el entendimiento de la forma como están relacionados los elementos que lo conforman y la manera cómo interactúan entre sí. Un nivel de abstracción es el grado de cercanía existente entre la realidad y su representación, con base en un lenguaje definido sobre el que es posible visualizar, construir y documentar los artefactos de un sistema software (Gannod y Cheng, 1999).

Claudio Riva (2002) establece dos niveles de abstracción: Dominio del problema, especifica desde la perspectiva del usuario qué se supone debe hacer el sistema, y el dominio de la solución, que define cómo el sistema logra lo que debe hacer. Estos

dos niveles se conectan por medio de las características del sistema, que corresponden al conjunto coherente e identificable de funcionalidades del sistema (Turner et al., 1999). Para el desarrollo de la tesis, el dominio de la solución se subdividió en dos partes (Monroy et al., 2013): Diseño, correspondiente a la solución conceptual del sistema, representada por el diseño en bajo nivel y el diseño en alto nivel o arquitectura. La segunda parte hace referencia a la implementación, conformada por el ejecutable del sistema y el código fuente, como se representa en la Figura 1.

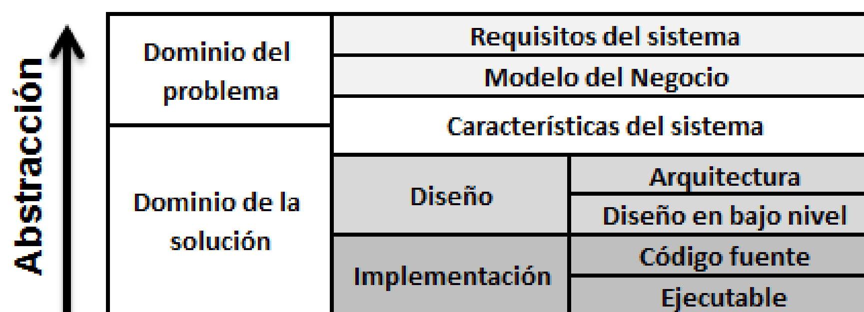

Figura 1. Niveles de abstracción

En el paso de la solución conceptual a la implementación no asegura, por múltiples causas, una coincidencia exacta entre la arquitectura propuesta y la arquitectura realmente obtenida en la implementación (Riva, 2004), además, en el proceso de evolución del producto es posible que se hagan modificaciones en la implementación sin actualizar los modelos del diseño, por eso se establece diferencia entre la arquitectura conceptual y la arquitectura concreta. La arquitectura conceptual hace referencia a la arquitectura que existe en la mente del arquitecto y que posiblemente está documentada (Riva, 2004; Tran y Holt, 1999).

Esta arquitectura también es conocida como arquitectura lógica (Medvidovic y Jakobac, 2006), diseñada (Kazman y Bass, 2005; Tran y Holt, 1999), propuesta (Riva, 2004; Woods et al., 1999), o idealizada (Harris et al., 1995). Por otra parte, la arquitectura concreta corresponde a la que se deriva directamente del código fuente (Riva, 2004; Tran y Holt, 1999), algunos autores la denominan arquitectura física (Medvidovic y Jakobac, 2006), construida (Harris et al., 1995; Tran y Holt, 1999), realizada (Woods et al., 1999) o implementada (Kazman y Bass, 2005; Riva, 2004).

En este orden de ideas, la arquitectura es el punto de referencia para: gestionar el proceso de desarrollo del producto software, validar los requisitos garantizando los atributos arquitectónicos, desplegar el sistema, implementar cambios y entender el producto para múltiples propósitos. Por eso es necesario documentar la arquitectura en forma suficientemente detallada, organizada y sin ambigüedades, para que los interesados puedan encontrar la información pertinente y tomar decisiones (Bass et

al., 2013). Eso se logra por medio de la descripción de la arquitectura, como se explica en la siguiente sección.

2.2. Descripción de la arquitectura

Es el trabajo producido para expresar la arquitectura. No existe un formato estándar para documentar la arquitectura, puede ser un documento, un repositorio o un conjunto de modelos, sin embargo debe contener la identificación y descripción del sistema, de los interesados y sus asuntos de interés; la definición de cada vista, modelo y puntos de vista arquitectónico usados; las reglas de correspondencia aplicadas y los fundamentos para la toma de decisiones arquitectónicas (ISO/IEC/IEEE 42010:2011).

La arquitectura corresponde a una abstracción del sistema, que consiste en conceptos y propiedades, y que se expresa por medio de la descripción de la arquitectura, teniendo en cuenta los asuntos que preocupan a los interesados. La descripción está conformada por: puntos de vista, vistas arquitectónicas, correspondencias, reglas de correspondencia y fundamentos para la toma de decisiones arquitectónicas, como se observa en la Figura 2 y se explica a continuación.

Un punto de vista es un producto de trabajo en el que se establecen los convenios para la construcción, interpretación y utilización de las vistas arquitectónicas, enmarcando los asuntos de interés específicos del sistema. Se debe tener en cuenta, que un asunto de interés puede estar comprendido en uno o varios puntos de vista y que todo punto de vista tiene convenciones para los tipos de modelado que utiliza. Las convenciones de un punto de vista pueden incluir los lenguajes utilizados, las notaciones, los tipos de modelos, las reglas de diseño, los métodos de modelado y las técnicas de análisis de las vistas producidas. (ISO/IEC/IEEE 42010:2011).

Una vista arquitectónica es un producto de trabajo que expresa la arquitectura del sistema software desde la perspectiva de un punto de vista. Puede estar conformada por uno o varios modelos arquitectónicos, que usan convenciones de modelado apropiadas a los asuntos de interés que están dirigidos, y que se especifican en los tipos de modelos que las rigen (ISO/IEC/IEEE 42010:2011).

Por otra parte, una correspondencia define una relación entre los elementos de la descripción de la arquitectura: interesados, asuntos de interés, vistas arquitectónicas, puntos de vista arquitectónicos, tipos de modelos, modelos arquitectónicos, decisiones y fundamentos para la toma de decisiones arquitectónicas. Las correspondencias se rigen bajo reglas de correspondencia, que son usadas para reforzar las relaciones entre la descripción de la arquitectura o entre descripciones de arquitectura (ISO/IEC/IEEE 42010:2011).

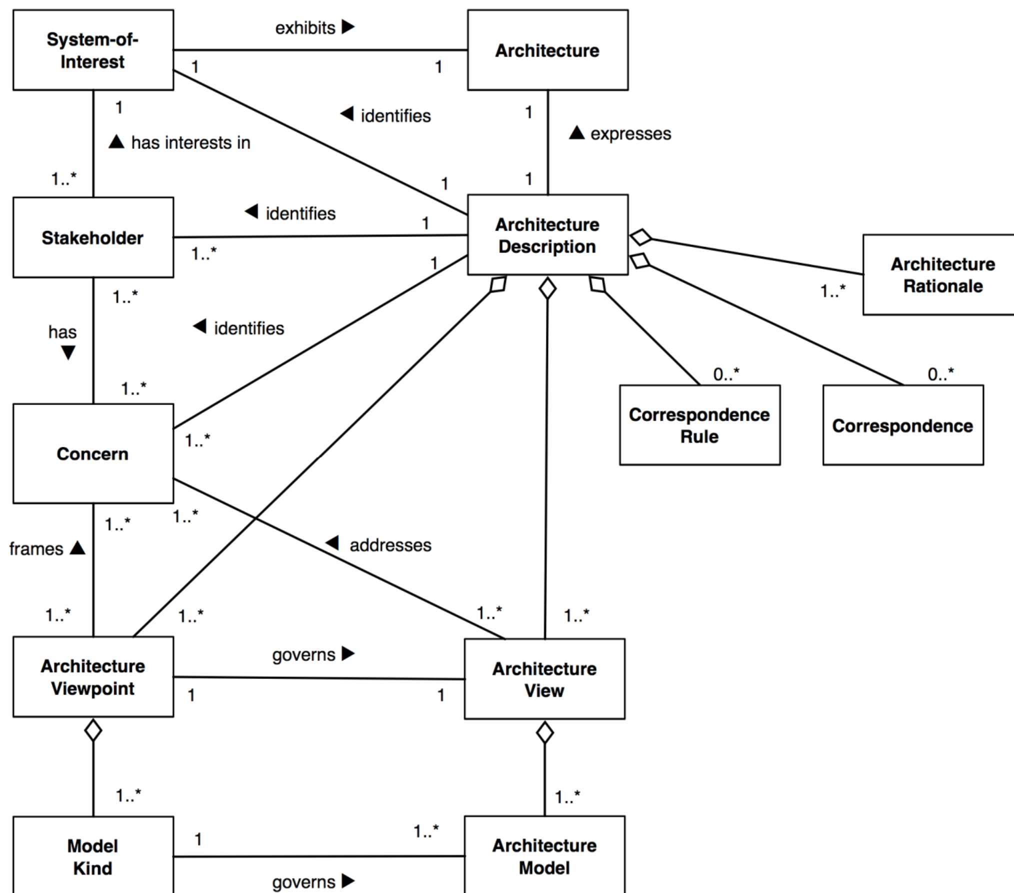

Figura 2. Modelo conceptual de la descripción de la arquitectura. Fuente: ISO/IEC/IEEE 42010:2011

Mientras que un fundamento arquitectónico explica, justifica o razona a cerca de las decisiones que se han tomado. El argumento de una decisión puede incluir su fundamento, las alternativas consideradas con sus posibles ventajas y desventajas, las potenciales consecuencias de la decisión y la citación a las fuentes de información utilizadas.

2.3. Vistas arquitectónicas

La recuperación de la arquitectura implica reconstruir los modelos que representan las vistas del producto software, teniendo en cuenta los puntos de vista que establecen los convenios para su interpretación y análisis, enmarcando los asuntos de interés específicos del sistema, por lo tanto, una de las primeras actividades que se debe realizar en este proceso es identificar las vistas a recuperar (Van Deursen et al., 2004). Existen varias propuestas teóricas que plantean los puntos de vista y las vistas arquitectónicas que describen el producto software.

Una de las más usadas en la de Philip Kruchten (1995), adoptada por el proceso unificado de desarrollo de software (Jacobson et al., 2000), que propone cuatro vistas: La vista lógica, que da soporte a los requerimientos funcionales, representando los servicios que debe proveer para el usuario final. La vista de procesos: orientada a representar aspectos relacionados con la concurrencia, la integridad del sistema y la tolerancia al fallo, que influyen en características como el desempeño y la disponibilidad. La vista de desarrollo, centrada en mostrar la organización del sistema por módulos en su entorno de desarrollo. La vista física, establece un mapeo entre los elementos lógicos del sistema y los nodos físicos que se requieren para su ejecución. Además agrega una quinta vista llamada vista de escenarios, que representa instancias generales de uso del sistema que integran las cuatro vistas anteriores.

Otra propuesta que ha tenido gran aceptación es la presentada por Soni et al. (1995). quienes plantean describir la arquitectura por medio de una vista conceptual, que describe el sistema en términos de sus elementos más representativos de diseño y las relaciones entre ellos. La vista de interconexión entre módulos abarca dos estructuras ortogonales: La descomposición funcional y capas. La vista de ejecución describe la estructura dinámica del sistema y la vista de código que describe cómo se organizan el código fuente, binario y las bibliotecas en el entorno de desarrollo.

En su libro Arquitectura de software en la práctica, Bass et al. (2013) afirman que la arquitectura del producto software se puede describir a partir de tres estructuras arquitectónicas: Módulos o unidades de implementación: representan una forma basada en el código de considerar el sistema, y corresponden a áreas de responsabilidad funcional asignadas. Componente y conector: elementos en tiempo de ejecución, que son las principales unidades de computación, y conectores que son los vehículos de comunicación entre los componentes. Estructuras de asignación: muestran la relación entre los elementos de software y los elementos en uno o más entornos externos en los que se crea y se ejecuta el software.

Para entender la arquitectura en tiempo de ejecución Callo Arias et al. (2011) definen tres clases de vistas: 1) El perfil de ejecución proporciona una visión de la realización de una función del sistema en tiempo de ejecución, indicando los componentes que la implementan y las dependencias en el tiempo. 2) La concurrencia de la ejecución brinda una visión general sobre cómo los elementos en tiempo de ejecución del sistema, se ejecutan concurrentemente en diferentes niveles de abstracción. 3) El uso de recursos provee una visión general sobre cómo los elementos software (componentes, proceso, hilos) usan los recursos hardware en tiempo de ejecución.

Las vistas arquitectónicas que asume el presente trabajo corresponden a la fusión de las propuestas hechas por Bass et al. (2013) y Callo Arias et al. (2011), porque al unirlas comprenden en forma completa todos los aspectos que describen el producto software, pertinentes al proceso de recuperación y análisis. Además, están documentadas según los lineamientos establecidos por el estándar ISO/IEC/IEEE

42010:2011. En la Tabla 1 se muestra las vistas arquitectónicas, resaltando con un asterisco las que propone Callo Arias et al. (2011)

Tabla 1. Vistas arquitectónicas. Adaptado de Bass et al. (2013) y Callo Arias et al. (2011)

Punto de vista	Estructuras
Estructuras modulares	Descomposición
	Usos
	Capas
	Clases
	Modelo de datos
Componentes y conectores	Servicios/Procesos
	Concurrencia
	Ejecución de la concurrencia*
	Perfil de la ejecución*
Estructuras de asignación	Implementación
	Asignación de trabajo
	Despliegue
	Uso de recursos*

Independientemente del punto de vista que se utilice para representar la arquitectura del sistema, básicamente existen dos aspectos que la describen: El primero es la estructura del sistema, representada en los elementos que lo conforman y sus relaciones, también se le conoce como la parte estática. El segundo es el comportamiento, que describe cómo la interacción entre los elementos afecta al sistema en un momento determinado o cuando se encuentra en un estado, también se denomina parte dinámica y revela aspectos relacionados con el orden de la interacción entre los elementos, la concurrencia y la dependencia de la interacción con respecto al tiempo (Clements et al., 2003). La descripción de la arquitectura se documenta utilizando el lenguaje natural y lenguajes especializados para este fin, que se explican en la siguiente sección.

2.4. Lenguajes de descripción de arquitectura

Un lenguaje de descripción de arquitectura (Architecture Description Language ADL) es la forma de expresión que se usa para representar la arquitectura del sistema. Provee uno o más tipos de modelos como medios para enmarcar los asuntos que preocupan a los interesados (puntos de vista). Puede establecer reglas de correspondencia para reforzar las relaciones entre descripciones de arquitectura, como se observa en el modelo conceptual de la Figura 3.

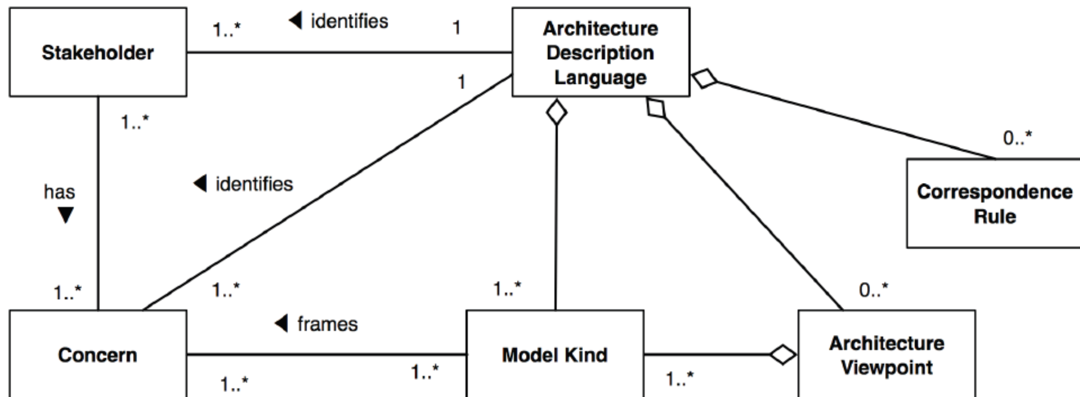

Figura 3. Modelo conceptual de un ADL. Fuente: ISO/IEC/IEEE 42010:2011

El ADL sirve para controlar y soportar el proceso de desarrollo, controlar y garantizar el análisis, la evolución, la reutilización, la flexibilidad e interoperabilidad del sistema (Clements, 1996). También sirve para automatizar los procesos de conversión y como instrumento de comunicación para soportar y comunicar decisiones, para lo cual debe identificar los asuntos de interés que expresa y sus interesados, identificar los tipos de modelos que implementa y cómo se enmarcan en los asuntos de interés, con sus respectivos puntos de vista y reglas de correspondencia (ISO/IEC/IEEE 42010:2011).

Se han presentado varias propuestas de lenguajes de descripción de arquitectura: LILEANNA (Tracz, 1993), Aesop (Garlan et al., 1994), UniCon (Shaw et al., 1995), Rapide (Luckham et al.,1995), MetaH (Binns et al., 1996), ACME (Garlan et al., 1997), SADL (Moriconi y Riemenschneider, 1997), Wright (Allen, 1997), ArchiMate 2.1 (The Open Group, 2012) y UML (Unified Modeling Language) en su versión 2.5 (OMG, 2015).

UML es un lenguaje para visualizar, especificar, construir y documentar los artefactos (modelos) de un sistema software, desde una perspectiva orientada a objetos (Rumbaugh et al., 2000). Es parte fundamental de la iniciativa MDA (Model Driven Architecture) orientada a desarrollar aplicaciones y escribir especificaciones, basadas en modelos independientes de la plataforma (PIM) de la aplicación (OMG, 2005). UML consiste de tres categorías principales de elementos del modelo, cada una de las cuales se pueden utilizar para hacer declaraciones acerca de los diferentes tipos de cosas individuales dentro del sistema que está siendo modelado. Estas categorías son (OMG, 2015):

Clasificadores. Un clasificador describe un conjunto de objetos. Un objeto es un individuo con un estado y relaciones con otros objetos. El estado de un objeto identifica los valores de las propiedades del objeto.

Eventos. Un evento describe un conjunto de posibles ocurrencias. Una ocurrencia es algo que sucede y que tiene alguna consecuencia en relación con el sistema.

Comportamientos. Un comportamiento describe un conjunto de posibles ejecuciones. Una ejecución es una representación de un conjunto de acciones (potencialmente sobre un cierto periodo de tiempo) que pueden generar y responder a las ocurrencias de eventos, incluyendo el acceso y cambio del estado de los objetos.

La sintaxis de UML hace referencia a la forma cómo se pueden construir, representar e intercambiar modelos UML. La especificación UML define la sintaxis de UML, tanto de manera abstracta como concreta. Sin embargo, la sintaxis de UML se especifica con el marco de referencia MOF (Meta Object Facility) (OMG, 2015), y el significado de los modelos sintácticos para los propósitos de conformidad entre herramientas se da en la especificación MOF Core y las especificaciones de definición de diagramas (OMG, 2015) y XMI (OMG, 2015).

Mientras que la semántica de UML tiene que ver con el significado estándar de las declaraciones hechas por un modelo UML sobre el sistema que se está modelando. En general el modelado en UML construye dos categorías semánticas (OMG, 2015):

Semántica estructural: Define el significado de los elementos del modelo estructural UML, con respecto a los individuos del dominio que está siendo modelado, y que puede ser cierto en algún momento específico del tiempo. También es conocida como semántica estática, sin embargo, se recomienda no denominarla así para evitar confusión con las limitaciones de significado que tiene este término en el contexto de los lenguajes de programación. Se representa por medio de los diagramas de clase, de estructuras compuestas, de componentes, de despliegue, de objetos y de paquetes.

Semántica del comportamiento: Define el significado del modelo de comportamiento de los elementos UML, que hacen declaraciones sobre cómo los individuos del dominio que está siendo modelado cambian con el tiempo. También es conocida como semántica dinámica. Se representa por medio de diagramas de actividades, de casos de uso, de máquinas de estado y los diagramas de interacción: secuencia, comunicación y tiempos.

La semántica estructural de UML provee los fundamentos de la semántica del comportamiento, como se observa en la Figura 4. Esto se refleja en la concepción de comportamiento en términos de cambio del estado del sistema especificado a través del modelado estructural. Mientras que los constructos del modelado estructural en UML están construidos sobre la base común de conceptos fundamentales como tipos, nombres de dominio, relaciones y dependencias; y están representados en distintas tipos de clasificadores, tales como: tipos de datos, clases, señales, interfaces, componentes, entre otros (OMG, 2015).

La semántica del comportamiento común está direccionada a la comunicación que puede resultar entre objetos estructurales y su comportamiento asociado. Es importante resaltar que sólo está orientado a comportamientos discretos dirigidos por eventos, sin embargo, la semántica de UML no establece restricciones sobre los

intervalos de tiempo entre eventos, que pueden ser considerados tan pequeños como sea necesario (OMG, 2015).

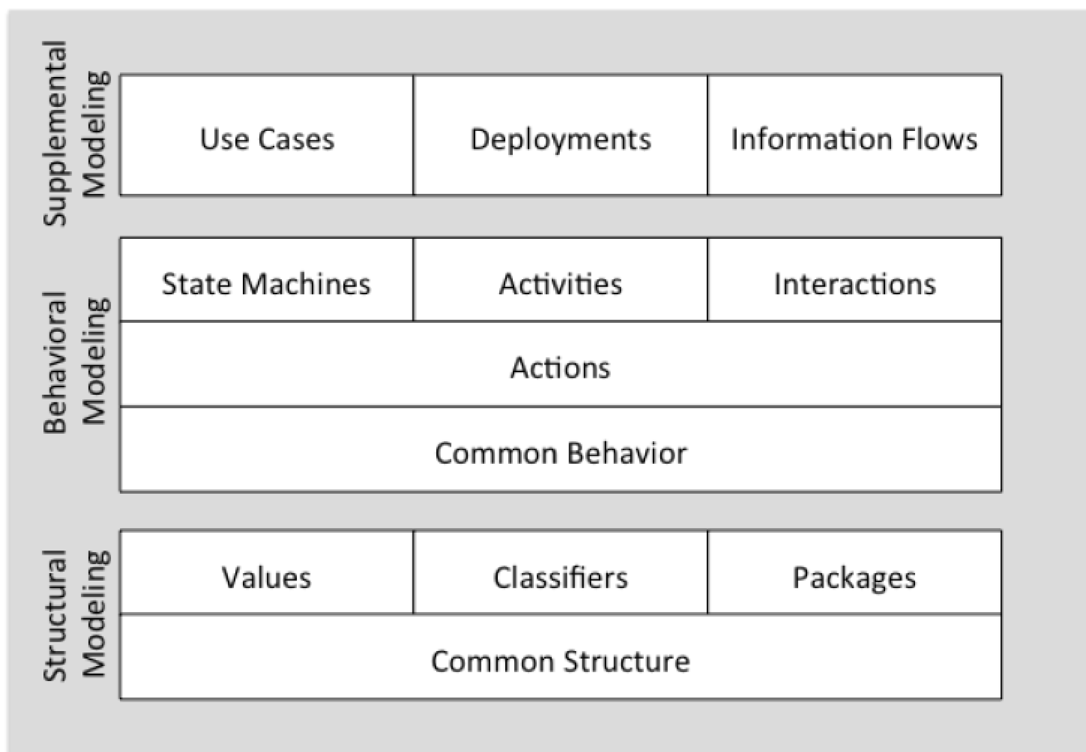

Figura 4. Áreas semánticas de UML. Fuente OMG, 2015

Las acciones son la unidad fundamental del comportamiento en UML, y se usan para definir el comportamiento a nivel de detalle, con una capacidad expresiva y de resolución comparable con las instrucciones ejecutables en los lenguajes de programación. Están disponibles para su uso con cualquiera de los formalismos de mayor nivel que requieran describir comportamientos detallados, como las máquinas de estado, las actividades y las interacciones (OMG, 2015). Además de tener un ADL, es conveniente contar con un marco de referencia arquitectónico para facilitar la creación, interpretación, análisis y uso de las descripciones de la arquitectura.

3. Recuperación de la arquitectura

El concepto de ingeniería inversa en el contexto de la ingeniería de software ha evolucionado desde la formulación planteada por Chikovsfky y Cross en 1990, según la cual es “el proceso de análisis de un sistema para identificar sus componentes y las relaciones entre ellos, y para crear una representación del sistema en otra forma o en otro nivel de abstracción”. Actualmente se entiende como “todo método destinado a la recuperación del conocimiento de un sistema software, en apoyo a la realización de actividades de ingeniería de software” (Tonella et al., 2007).

La recuperación de arquitectura es un enfoque de la ingeniería inversa, cuyo principal objetivo es reconstruir las vistas arquitectónicas de un producto software (Ducasse y Pollet, 2009). En la literatura se encuentran varios términos para hacer referencia a este concepto: Arquitectura inversa; extracción, minería, descubrimiento o reconstrucción de la arquitectura. Para algunos autores, el término descubrimiento está directamente relacionado con procesos "arriba-abajo" (Top-Down), mientras que el término recuperación lo asocian sólo con procesos "abajo-arriba" (Bottom-Up) (Medvidovic et al., 2003). En este documento no se aplica esta distinción.

Existen tres enfoques para la realización del proceso de recuperación de la arquitectura, dependiendo del referente que se utilice para el inicio del proceso (Ducasse y Pollet, 2009). El primero es "Abajo-Arriba" (Bottom-Up), que comienza con el conocimiento en bajo nivel representado en los modelos del código fuente, y continúa haciendo procesos progresivos de abstracción hasta llegar al nivel de arquitectura, siguiendo la propuesta de Tilley et al. (1996) conocida como extracción, abstracción y presentación.

El segundo enfoque es "Arriba-Abajo" (Top-Down), comienza con conocimiento en alto nivel representado en requerimientos o estilos arquitectónicos, que sirven para formular una hipótesis de la arquitectura, que finalmente es comparada con la representación obtenida directamente a partir de código fuente. El tercer tipo de enfoque combina los dos anteriores, por eso es conocido como híbrido; porque por una parte, el conocimiento es abstraído utilizando varias técnicas (Bottom-up), mientras que por otra, el conocimiento en alto nivel es refinado (Top-Down) y confrontado con el extraído usando el enfoque Abajo-Arriba.

En esta sección se describe el proceso de recuperación de la arquitectura, prestando especial atención a las actividades que lo conforman y, a las técnicas de extracción

de datos y organización del conocimiento, que hacen posible la recuperación de la estructura y el comportamiento de los productos software.

3.1. El proceso de recuperación de la arquitectura

Para recuperar las vistas arquitectónicas de un sistema software se requiere realizar un conjunto de pasos, orientados bajo una serie de lineamientos. Actualmente existen múltiples propuestas que describen la forma cómo debe realizarse el proceso de recuperación de arquitectura, sin embargo en esencia todas ellas incluyen la propuesta de Tilley et al. (1996) que comprenden las siguientes actividades:

Extracción de datos: También conocida como recolección, consiste en identificar a partir de los datos en bruto del sistema objeto de estudio, los elementos que lo conforman y las relaciones entre ellos, usando mecanismos apropiados de extracción.

Organización del conocimiento: En esta actividad los datos obtenidos en la etapa de extracción se representan en una forma que permita su almacenamiento y recuperación de manera fácil y eficiente, haciendo posible el análisis de los artefactos y sus relaciones, por eso también se le conoce como abstracción.

Exploración de la información: brinda los mecanismos para navegar, analizar y presentar los resultados del proceso de ingeniería inversa. La navegación permite al arquitecto desplazarse por las estructuras de información generadas en la fase de organización del conocimiento. Para hacer posible el análisis se deriva y extrae información que no está disponible en forma explícita, generando vistas que pueden ayudar a la comprensión del sistema objeto de estudio. La presentación de los resultados debe utilizar estrategias que faciliten su comprensión, haciendo posible la navegación y el análisis.

Las técnicas usadas comúnmente en cada una de estas actividades se explican más adelante en esta sección. Estas actividades corresponden al modelo genérico del proceso de ingeniería inversa y hacen parte fundamental del proceso de recuperación de arquitectura. Las propuesta más utilizadas como procesos para recuperar la arquitectura son Symphony (Van Deursen et al., 2004), QADSAR (Stoermer et al., 2003) y Focus (Medvidovic y Jakobac, 2006) descritas a continuación.

Symphony: Es un proceso dirigido por vistas que combina patrones comunes y buenas prácticas de ingeniería inversa para recuperar la arquitectura de un producto software utilizando vistas y puntos de vistas arquitectónicos. Comprende dos fases, la primera es la reconstrucción del diseño, donde se analiza el problema, se seleccionan los puntos de vista que se pretenden reconstruir, se definen las vistas de las fuentes, y se designan las reglas de mapeo de las fuentes a las vistas que se quieren lograr.

En la fase de ejecución reconstrucción se analiza el sistema, se extraen las vistas fuente, y se aplican las reglas de mapeo para obtener las vistas que se quieren reconstruir. Estas fases son iterativas, cada ejecución de la reconstrucción revela nuevas oportunidades que permiten refinar el entendimiento del problema y la reconstrucción del diseño. Este proceso permite un doble resultado, la reconstrucción del diseño y la reconstrucción de la arquitectura.

QADSAR: Es un proceso de reconstrucción de la arquitectura de software dirigido por atributos de calidad (Quality Attribute Driven Software Architecture Reconstruction). Comprende los siguientes pasos: Identificación del ámbito, extracción de modelos a partir de las fuentes de información, abstracción de los modelos, instanciación de elementos y propiedades y finalmente evaluación de atributos de calidad.

En el primer paso se identifican los tipos de vistas arquitectónicas y las partes del sistema que se pretenden reconstruir. La identificación depende de los escenarios de atributos de calidad, de los modelos de atributos de calidad relacionados y del tipo de sistema. En el segundo paso se extraen los elementos y las relaciones (estáticas y dinámicas). En el tercer paso se identifican y aplican las estrategias de abstracción para detallar las vistas obtenidas a partir de la información fuente.

En el cuarto paso se especifican los tipos de elementos para cada tipo de vista que se quiere recuperar. A estos elementos, representados en capas, tareas, relaciones, etc., se les debe asignar las propiedades requeridas, tales como rendimiento, tiempos de las tareas, etc. El último paso es realizado con escenarios de atributos de calidad.

Focus: Es un proceso ligero de recuperación de arquitectura que tiene tres facetas: La primera usa los requerimientos de evolución de un sistema para aislarlo y recuperar incrementalmente sólo el fragmento de la arquitectura del sistema afectado por la evolución. La segunda implica que se recuperan las nociones arquitectónicas de los estilos arquitectónicos y conectores de software, además de los componentes. Finalmente, no sólo permite la reconstrucción de la arquitectura, también hace posible su evolución.

El proceso de recuperación de la arquitectura está conformado por seis actividades divididas en dos categorías: Recuperación de la arquitectura lógica y recuperación de la arquitectura física. La primera actividad corresponde a la identificación de componentes a partir de código fuente, la segunda consiste en proponer el modelo arquitectónico idealizado, en la tercera se mapean los componentes identificados en el modelo arquitectónico idealizado.

En la cuarta actividad se identifican los casos de uso claves. En la quinta actividad se analiza la interacción de los componentes teniendo en cuenta los casos de uso clave y el modelo arquitectónico idealizado. En la última actividad se genera la arquitectura refinada, logrando el objetivo propuesto para el proceso.

Otra propuesta genérica para el proceso de recuperación de arquitectura es la de Claudio Riva (2002), en la que define cuatro pasos: recuperar los conceptos arquitectónicos, obtener el modelo, hacer abstracción y presentar los resultados. Por otra parte, Pinzger et al. (2003) proponen PuLSE-DSSA para recuperar la arquitectura de familias de programas; Favre (2004) plantea CacOphoNy, un proceso dirigido por metamodelos; Vasconcelos y Werner (2011) se basan en minería de datos, mientras que Pashov y Riebisch (2004) proponen un proceso basado en modelado de características.

Todo proceso de recuperación de la arquitectura involucra la aplicación de técnicas de extracción de datos, técnicas de organización del conocimiento y técnicas de exploración de la información, que se explican a continuación.

3.2. Técnicas de extracción de datos

El objetivo de las técnicas de extracción de datos consiste en recuperar los elementos que constituyen el sistema, a partir de los artefactos disponibles del producto software, identificando sus relaciones en bajo nivel. El estándar ISO/IEC 19506, más conocido como la especificación KDM (Knowledge Discovery Metamodel) de la OMG, establece un inventario de los elementos que conforman el sistema software y define un formato para su representación, como se explica más adelante. Entre los artefactos que conforman el producto software están los archivos ejecutables, de código fuente, de configuración, descriptores de servicios, bases de datos, modelos del sistema, entre otros.

Las técnicas de extracción de datos se clasifican en técnicas de análisis estático y técnicas de análisis dinámico (Van Deursen et al., 2004). Las primeras recuperan información que es válida para todas las posibles ejecuciones del sistema, mientras que las segundas recuperan información sobre un escenario específico de ejecución. Las técnicas de análisis estático recuperan los aspectos estructurales del sistema a partir de la información textual disponible en los artefactos del producto, lo cual se puede lograr usando alguno de los siguientes métodos:

1. *Inspección manual*: Consiste en examinar directamente los artefactos, registrando aspectos físicos que se puedan distinguir, como por ejemplo directorios de archivos, nombres de clases a partir de la exploración de código fuente, etc.; o aspectos de comportamiento a partir de la interacción directa con la aplicación (Moonen, 2003). También incluye técnicas informales de extracción de datos, como entrevistas a expertos, usuarios y al equipo de desarrollo.

2. *Análisis de léxico*: Radica en identificar los componentes de léxico a partir del texto contenido en los artefactos del producto software, separando aquellos que representan elementos constituyentes del sistema. La herramienta más usada en el contexto de la ingeniería inversa para este fin es grep, que permite hacer búsquedas a partir de cadenas de texto que coincidan con una expresión regular. También hay disponibles lenguajes que permiten el procesamiento de texto como awk, perl y lex,

que facilitan al usuario la ejecución de ciertas acciones cuando se está buscando una cadena (Callo Arias et al., 2011).

Murphy y Notkin (1997), proponen un Modelo extractor de la fuente léxica (Lexical Source Model Extractor LSME), que utiliza un conjunto de expresiones regulares jerárquicamente relacionadas, para describir las construcciones del lenguaje que tienen que ser asignadas a la vista de origen. El uso de patrones jerárquicos evita algunas de las dificultades de los patrones léxicos naturales, manteniendo la flexibilidad y robustez de este enfoque.

3. *Análisis sintáctico*: Consiste en verificar la relación de concordancia y jerarquía de las cadenas que conforman los artefactos que representan el producto software. Se utilizan para aumentar la precisión y el nivel de detalle. Generalmente crean un árbol sintáctico a partir de la entrada y permiten a los usuarios recorrerlo, consultarlo o buscar coincidencias de ciertos patrones, lo cual evita tener que manejar todos los aspectos de cada lenguaje y permite centrarse en partes relevantes del proceso de recuperación.

4. *Análisis difuso*: Esta técnica puede ser vista como un híbrido entre el análisis léxico y el análisis sintáctico (Van Deursen et al., 2004). Hace posible reconocer solamente ciertas partes de un lenguaje de programación ignorando algunos componentes léxicos (Koppler, 1997). Se centran principalmente en el análisis de C y C ++ para facilitar la consulta del programa. Generalmente, esto implica la extracción de información sobre las referencias a un símbolo, las definiciones globales, funciones de llamadas, archivo incluidos, etc.

5. *Islas gramaticales*: Combinan las posibilidades de la especificación detallada de las gramáticas con el comportamiento tolerante de los enfoques léxicos. Los analizadores robustos generados a partir de isla gramaticales combinan la exactitud del análisis sintáctico con la velocidad, la flexibilidad y la tolerancia que, por lo general, sólo se encuentran en el análisis léxico, lo cual hace que este enfoque sea muy adecuado para el desarrollo de extractores del modelo origen, incluso si el extractor resultante se utiliza sólo para un único proyecto (Moonen, 2003).

6. *Analizadores semánticos*: Consiste en identificar el significado de los componentes de léxico, generalmente apoyándose en los resultados del análisis sintáctico. Algunas estrategias que se utilizan son: el uso de nombres y tipos, el análisis de flujo de datos, y la definición de puntos de análisis. Otra estrategia que se utiliza en el análisis semántico es la indexación latente semántica, que consiste identificar patrones en las relaciones entre los términos contenidos en los artefactos software de textos no estructurados (Kuhn et al., 2005; De Boer y van Vliet, 2008; Risi et al., 2010)

Por otra parte, el análisis dinámico se realiza aplicando técnicas de instrumentación, que consisten en supervisar la ejecución del producto software, para analizar patrones, secuencias y dependencias (Schmerl et al., 2006). Esto se logra mediante

el uso de depuradores, contadores de rendimiento, el registro de eventos, el trazado de los caminos de ejecución y la conexión a un entorno de ejecución preparado.

Los depuradores son aplicaciones que facilitan la prueba y refinación de programas, permitiendo la ejecución de un conjunto de instrucciones, mientras se hace seguimiento al comportamiento del sistema. Actualmente todos los entornos de desarrollo incorporan este tipo de herramientas en sus funcionalidades. En el caso de java existe la posibilidad de utilizar la interfaz JPDA (Java Platform Debugger Architecture), que permite el paso de solicitudes entre el proceso depurado y el depurador (Oracle, 2015).

Los contadores de rendimiento, también conocidos como profiling o monitores de rendimiento, son componentes que permiten supervisar el comportamiento de una aplicación en tiempo de ejecución. Algunas herramientas que brindan esta funcionalidad son: HPROF (Oracle, 2016), gprof (Sourceware, 2016), YourKit (YourKit, 2016), AQTime (Automatedq, 2016), entre otras.

El trazado de los caminos de ejecución radica en la capacidad de recibir mensajes sobre la ejecución del sistema durante su propia ejecución. Implica la realización de tres fases. La primera consiste en incluir el código de traza en la aplicación, lo que exige identificar previamente las partes del sistema que se van a monitorear y los aspectos que registra la traza. La segunda fase consiste en ejecutar el escenario de evaluación, para que se registre la traza de información en el destino especificado. Finalmente se hace el análisis de la traza para extraer los elementos del sistema que se han planeado recuperar (Callo Arias et al, 2011). El registro de eventos es similar al registro de trazas de ejecución, consiste en observar los eventos en la ejecución de la aplicación, guardando el trazado de aquellos que se consideran importantes.

La conexión a un entorno de ejecución preparado se fundamenta en la capacidad de establecer comunicación entre procesos, con el fin de pasar como mensaje las características de ejecución bajo observación, desde la aplicación objeto de estudio hacia el entorno que hace el análisis. Esta estrategia es utilizada por Yan et al. (2004) en la herramienta Discotect, y por Mei y Huang (2004) en la plataforma PKUAS.

Adicionalmente, para la recuperación del comportamiento simulando la ejecución de la aplicación se pueden utilizar técnicas estáticas, que implican la instrumentación del código fuente, generalmente por medio de anotaciones (Vanciu y Abi-Antoun, 2013; Abi-Antoun y Barnes, 2010; Abi-Antoun y Aldrich, 2009). Esta estrategia involucra dos limitaciones que se deben tener en cuenta al momento de utilizarla, la primera radica en la sobrecarga que implican las anotaciones en el código fuente, y la segunda está asociada con el tiempo que representa la modificación del código fuente al agregar las anotaciones, teniendo en cuenta que se hace manualmente (Abi-Antoun y Barnes, 2010).

3.3. Técnicas de organización del conocimiento

El resultado del proceso de extracción de los elementos del sistema genera gran cantidad de información, que debe ser representada en una forma que permita su almacenamiento y recuperación de manera fácil y eficiente, para que sea posible el análisis de los artefactos y sus relaciones. La principal estrategia que se utiliza para lograr este objetivo es la abstracción, combinada con el análisis del dominio del problema para determinar la forma como se organizan los elementos extraídos (Tilley et al., 1996).

El objetivo principal de las técnicas de organización del conocimiento es obtener las vistas objetivo a partir de las vistas origen (Van Deursen et al., 2004). Esto se logra estableciendo reglas de mapeo y el conocimiento del dominio del problema, para relacionar los elementos básicos extraídos con los modelos que representan las vistas objetivo. Dependiendo del grado de automatización, las técnicas de organización del conocimiento se pueden clasificar en (Van Deursen et al., 2004): manuales, semiautomáticas y automáticas.

Las técnicas manuales generalmente usan herramientas de propósito general para elaborar manualmente las vistas objetivo a partir de la inspección directa del sistema. Herramientas como SHRiMP (Storey et al., 1996), Rigi (Kienle y Muller, 2010) y Bauhaus (Raza et al., 2006) se utilizan para ayudar a la visualización de resultados intermedios.

Las técnicas semiautomáticas permiten la generación de las vistas objetivo por medio de la interacción con herramientas, generalmente estableciendo el mapeo en forma manual. Utilizan estrategias como el álgebra relacional (Holt, 1998) y consultas SQL (Kazman y Carrière, 1999), para crear la vista jerárquica de módulos agrupándolos a partir del cálculo de dependencias; el álgebra relacional de particiones (Krikhaar, 1999) para verificar la conformidad entre la arquitectura recuperada y la arquitectura hipotética, así como también para calcular dependencias entre módulos (Postma, 2003); y el álgebra proposicional para inferir relaciones (Van Deursen et al., 2004).

Mens (2000) usa el lenguaje Prolog para mapear elementos del diseño hacia representaciones arquitectónicas y verificar la conformidad, mientras que Claudio Riva (2002) combina el álgebra relacional y el lenguaje Polog para definir un método que infiere información arquitectónica. También está la propuesta del lenguaje de consulta GReQL (Kullbachm y Winter, 1999) y el Modelo de reflexión (Murphy et al., 2001), este último para utilizado para verificar conformidad a nivel de arquitectura.

Finalmente, las técnicas automáticas están basadas en distintas clases de algoritmos de agrupamiento, que utilizan estrategias como el acoplamiento (Mancoridis et al., 1999; Tzerpos y Holt, 2000), los nombres de archivos (Anquetil y Lethbridge, 1999), el análisis de conceptos (Eisenbarth et al., 2003; Lundberg y Löwe, 2003; Bojic y Velasevic, 2000) y el uso de reglas de diseño (Cai et al., 2013).

3.4. Técnicas de exploración de la información

Estas técnicas tienen como fin hacer accesibles las vistas objetivo obtenidas en la fase de organización del conocimiento, para que se puedan inspeccionar e interpretar, brindando mecanismos que permitan visualizar, navegar y analizar resultados del proceso de ingeniería inversa. Para cada una de estas funciones existen técnicas que se explican a continuación.

Las técnicas de visualización hacen posible la presentación de los resultados en forma visual a través de gráficos, texto y otras formas de percepción humana incluyendo la interacción (Van Deursen et al., 2004). La visualización implica la representación de la información del sistema por medio de modelos correspondientes a las vistas objetivo, lo cual se logra utilizando reglas de mapeo, aplicando el concepto de metáfora (Panas et al., 2003).

La navegación permite al arquitecto desplazarse por las estructuras de información generadas en la fase de organización del conocimiento. Las estrategias usadas para lograr este objetivo son los hipervínculos, el plegado (folding) y colapso (collapsing) de información (Panas, 2003) y la técnica de ojo de pescado (Tilley, 1998). Por otra parte, el análisis de los resultados se logra con el cálculo de métricas, que brindan herramientas como Imagix 4D (Imagix Corporation, 2016) y JDepend (Clarkware Consulting, 2016).

Algunas herramientas que permiten la exploración de la información, brindando funcionalidades de visualización y navegación son: VizzAnalyzer (Löwe et al., 2003), VANESSA (Pacione et al., 2005), SHriMP (Storey et al., 1995), CodeCrawler (Lanza, 2003), iCIA (Interactive Change Impact Analysis Tool) (Kim et al., 2010), CCVISU (Beyer, 2008), ArchView (Pinzger, 2005) y CoCA-Ex (Holy et al., 2013).

3.5. Especificación KDM

La especificación KDM (Knowledge Discovery Metamodel) (OMG, 2011) define un meta-modelo para representar productos software existentes, sus elementos, asociaciones y entornos de operación, proporcionando una ontología común y un formato de intercambio, que hace posible la interoperabilidad entre herramientas de recuperación y análisis del conocimiento. KDM se define a través de MOF y determina el formato de intercambio a través de XML mediante la aplicación de la norma de mapeo MOF XMI al modelo KDM. El formato de intercambio definido por KDM se llama esquema KDM XMI. Esta especificación ha sido adoptada por la Organización Internacional de estándares como el estándar ISO/IEC 19506 y está estructurada en cuatro capas y doce paquetes que se representan en la Figura 5 y se explican a continuación.

Las capas corresponden al nivel de abstracción de la representación, comenzando por capa de infraestructura que hace referencia a los elementos físicos, hasta llegar a la capa de abstracción que representa elementos lógicos del sistema. Cada capa

del meta-modelo se base en la capa previa y está estructurada en paquetes que definen un conjunto de elementos del meta-modelo, cuyo propósito es representar un aspecto específico e independiente del conocimiento del sistema objeto de estudio.

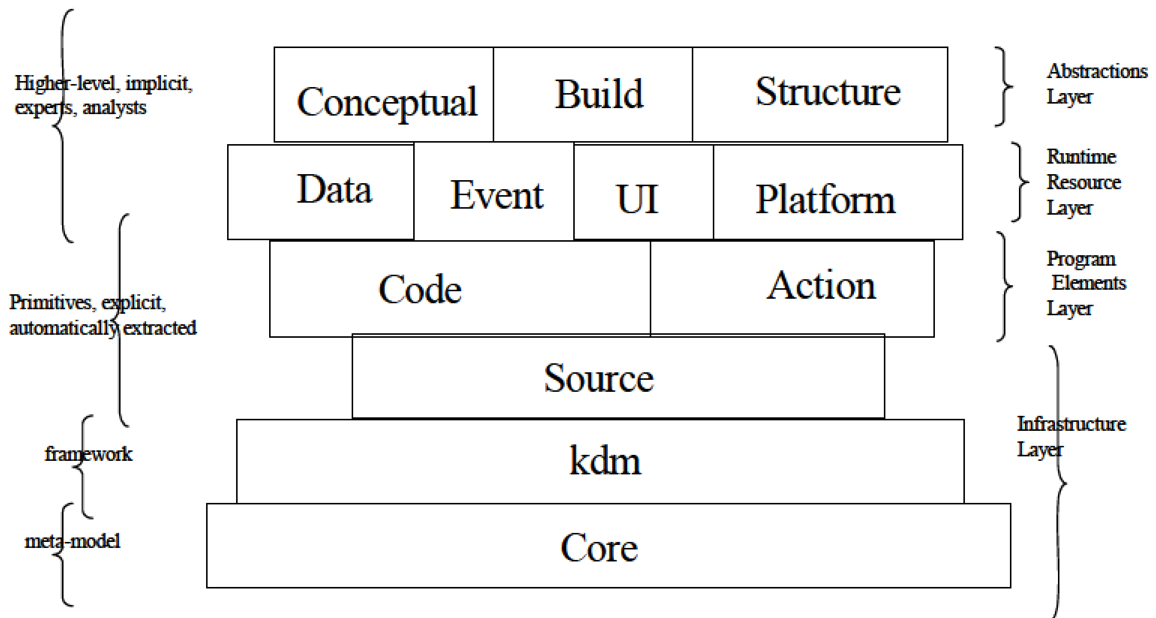

Figura 5. Estructura de los paquetes KDM. Fuente OMG (2011)

La capa de infraestructura corresponde al nivel más bajo de abstracción, define un conjunto de elementos usados sistemáticamente a través de toda la especificación KDM y está conformada por tres paquetes: Núcleo (Core), KDM y Fuente (source). El primero define los elementos básicos para crear y representar los elementos del meta-modelo usados en todos los paquetes, determinando la estructura del modelo KDM y definiendo los patrones fundamentales, restricciones y tipos de datos usados en la especificación.

El paquete KDM también define elementos comunes del meta-modelo que constituyen la infraestructura de los demás paquetes y el marco de referencia de cada representación KDM, determinando su estructura física que consiste en uno o más segmentos que poseen los modelos. También define algunos mecanismos de extensión básicos, para agregar nuevos elementos al meta-modelo con el fin de extender la semántica de la especificación.

El paquete fuente define los elementos del meta-modelo que representan el inventario de los artefactos físicos que existen en el producto software, y el mecanismo clave de trazabilidad que consiste en definir cómo un elemento KDM referencia los artefactos del sistema original. Esta es una parte importante de la infraestructura de KDM, porque los otros paquetes utilizan este mecanismo para hacer referencia al código fuente y los artefactos físicos del producto software,

permitiendo identificar cuáles son los artefactos del sistema, cuál es el rol de cada uno de ellos, cómo están organizados y qué dependencias existen entre ellos.

La capa de elementos del programa define un conjunto de elementos del meta-modelo, cuyo principal fin es proveer una representación intermedia independiente del lenguaje para varios constructos determinados por los lenguajes de programación. Esto facilita la identificación de los elementos computacionales y no computacionales del sistema, de los módulos, de los tipos de datos usados y de las unidades de comportamiento; así como también permite identificación de la organización en bajo nivel de los elementos computacionales, y de las relaciones entre los elementos físicos del sistema, los elementos computacionales y los módulos. De igual manera, hace posible la identificación de las relaciones en bajo nivel entre los elementos de código fuente, específicamente las relaciones de control de flujo y flujos de datos.

Esta capa está conformada por dos paquetes: Código y acción. El paquete código define un conjunto de elementos del meta-modelo cuyo propósito es representar los elementos del programa y sus relaciones a nivel de implementación. El paquete acción extiende del paquete Código y define los elementos del meta-modelo que representan la descripción del comportamiento a nivel de implementación, determinado por los lenguajes de programación. Por regla general en una instancia KDM, cada instancia de un elemento de acción representa un constructo del lenguaje de programación.

La capa de recursos en tiempo de ejecución está conformada por cuatro paquetes: Plataforma, interfaz de usuario, eventos y datos. Describe los patrones comunes para la representación del entorno operativo del sistema software, para lo cual suministra los elementos de modelado que representan los recursos y las abstracciones de las acciones de los recursos para que puedan ser gestionados. El paquete Plataforma define los elementos del meta-modelo que hacen posible la representación de los entornos de operación del sistema, facilitando la identificación de los recursos que usa el sistema, los elementos de la plataforma en tiempo de ejecución usados por el sistema y su comportamiento asociado, los flujos de control iniciados por los eventos sobre los recursos, entre otros aspectos.

El paquete de la interfaz de usuario define un conjunto de elementos del meta-modelo que representan aspectos relacionados con la información de la interfaz de usuario, incluyendo su composición, la secuencia de operaciones y sus relaciones con el producto software. Permite distinguir los elementos de la interfaz de usuario, cómo están organizados, los flujos de datos originado a partir de la interacción del usuario y los flujos de control iniciados por el usuario.

El paquete de eventos define los elementos del meta-modelo que representan el comportamiento en alto nivel de la representación, particularmente las transiciones de estados dirigidas por eventos. Permite establecer cuáles son las distintas etapas que intervienen en el comportamiento del sistema de software, cuáles son los

eventos que causan las transiciones entre estados y qué elementos de acción se realizan en un estado determinado.

El paquete de datos define los elementos del meta-modelo cuyo propósito es representar la organización de los datos en el producto software, haciendo posible conocer la organización de los datos persistentes, el modelo de información que los soporta, los elementos de acción para la lectura y la escritura de los datos persistentes, y el control de flujo determinado por los eventos correspondientes a la persistencia de datos.

Adicionalmente, la capa de abstracción define el conjunto de elementos que representan el dominio del problema y las abstracciones específicas de la aplicación, como los artefactos relacionados con el proceso de construcción del producto software. Esta capa está conformada por los paquetes: Estructura, conceptual y construcción. El primero define los elementos del meta-modelo que representan los componentes arquitectónicos del sistema y la trazabilidad de estos elementos hacia otros aspectos del software, por lo que hace posible identificar los elementos estructurales, sus interfaces y su organización.

El paquete conceptual brinda los constructos para crear el modelo conceptual durante la fase de análisis del descubrimiento del conocimiento a partir del código fuente. Permite definir los elementos de comportamiento y los escenarios soportados por el sistema, además de los términos del dominio y las reglas de negocio implementados en el sistema. El paquete construcción define los elementos del meta-modelo que representan los aspectos relacionados con el proceso de construcción del software, haciendo posible identificar las entradas del proceso, los artefactos generados, las herramientas usadas, los flujos de trabajo u los proveedores de los artefactos fuente.

4. Análisis de vistas arquitectónicas

El análisis de la arquitectura es la actividad que consiste en descubrir propiedades importantes del sistema a partir de sus modelos arquitectónicos (Taylor et al., 2009). Su principal propósito es predecir la calidad de un sistema antes de que sea construido, no para establecer estimaciones precisas, sino para definir los principales efectos de la arquitectura, por eso se utiliza para evaluar los riesgos potenciales y verificar que los requisitos de calidad se han abordado en el diseño (Dobrica y Niemelä, 2002).

El propósito de esta sección es presentar los aspectos relevantes para el análisis arquitectónico, con el ánimo de entender los objetivos que pretende, su ámbito, los tipos de análisis que existen y el nivel de automatización. También se relacionan las principales técnicas de evaluación de la arquitectura, indicando el fin que pretenden lograr y las estrategias que utilizan.

4.1. Aspectos relevantes para el análisis arquitectónico

El proceso de análisis de la arquitectura deriva información no explicita, brindando vistas útiles que contribuye al entendimiento del sistema objeto de estudio, siendo mejores los resultados cuando se utilizan mecanismos lingüísticos para lograr este propósito (Tilley et al., 1996), aprovechando las capacidades descriptivas disponibles en lenguajes de modelado como UML, que registran la estructura, el comportamiento y demás propiedades del sistema (Clements et al., 2002).

El objetivo del análisis arquitectónico consiste en garantizar la completitud, la consistencia, la compatibilidad y la corrección (Taylor et al., 2009). La completitud aborda aspectos internos y externos. Los aspectos internos están relacionados con la intención de la arquitectura y la notación del modelado, asegurando que todos los elementos hayan sido modelados en forma completa usando la notación seleccionada, y que todas las decisiones hayan sido aprehendidas. Los aspectos externos hacen referencia al cumplimiento de los requisitos del sistema, ante el desafío que representa su complejidad y las múltiples notaciones utilizadas para capturarlos.

La consistencia es una propiedad interna del modelo arquitectónico, que radica en asegurar que los diferentes elementos del modelo no se contradicen unos con otros y comprende las siguientes dimensiones: Nombre, interfaz, comportamiento, interacción y refinamiento. Existen dos tipos de consistencia: sintáctica y semántica.

La primera comprende los aspectos relacionados con el cumplimento de las reglas de sintaxis del lenguaje de modelado, y la segunda hace referencia a la compatibilidad entre lo especificado y lo que expresa el modelo (Clarke et al., 2003).

La compatibilidad es una propiedad externa que asegura que el modelo arquitectónico se adhiere a las guías y restricciones del estilo arquitectónico, de la arquitectura de referencia y de los estándares arquitectónicos (Taylor et al., 2009). De igual forma, la corrección es una propiedad externa del modelo arquitectónico, que asegura que el modelo cumple en forma completa las especificaciones del sistema, y que la implementación del sistema cumple totalmente con lo expresado por la arquitectura (Taylor et al., 2009).

Por otra parte, el análisis arquitectónico comprende aspectos estructurales, de comportamiento, de interacción y características no funcionales, se hace a nivel de componentes y conectores, o de sistemas y subsistemas, estableciendo relaciones de intercambio de datos, a nivel de estructuras de datos, flujos de datos o propiedades de los datos de intercambio. Se puede hacer en diferentes niveles de abstracción, y sirve para comparar dos o más arquitecturas, con respecto a sus propiedades no funcionales, datos, procesos, formas de interacción y configuración (Taylor et al., 2009).

Existen diferentes criterios para clasificar los tipos de análisis arquitectónicos. Dependiendo el nivel de intervención humana en la realización del análisis, puede ser: Manual, parcialmente automático y completamente automático. Otro criterio es el nivel de formalidad de los modelos sobre los cuales se hace al análisis, según esto, el análisis puede ser: formal, semi-formal e informal. Finalmente, según el aspecto que se analiza puede ser estático (estructura) o dinámico (comportamiento) (Taylor et al., 2009).

4.2. Técnicas de evaluación de la arquitectura

Hay dos clases de técnicas principales de evaluación de arquitecturas (Bass et al., 2013; Abowd et al., 1997): Revisión basada en preguntas y métricas. Las primeras generan preguntas cualitativas que se formulan a la arquitectura, y que se pueden aplicar para cualquier atributo de calidad. En esta categoría se encuentran las técnicas basadas en escenarios, en cuestionarios y en listas de verificación. Por otra parte, las técnicas basadas en métricas proponen medidas cuantitativas que se aplican a la arquitectura, y se utilizan para responder a preguntas específicas, por lo cual están orientadas a atributos de calidad concretos, lo que ha influido en el hecho de que no sean tan ampliamente aplicadas como las técnicas de revisión basada en preguntas (Dobrica y Niemelä, 2002). En este grupo se incluyen las técnicas de generación de prototipos, simulación, experimentos y cálculo de métricas.

Bass et al. (2013) establecen cuatro dimensiones que caracterizan a las técnicas de evaluación de arquitecturas: generalidad, nivel de detalle, la fase y qué aspecto evalúan. Con respecto a la generalidad, las técnicas pueden ser: generales

(cuestionario de preguntas), específicas de dominio (lista de chequeo y creación de prototipos), o específicas del sistema (escenarios). El nivel de detalle indica la cantidad de la información (poca, intermedia o mucha) que se requiere sobre la arquitectura para realizar la evaluación.

Existen tres fases o momentos de interés para la evaluación de la arquitectura: antes, durante y después del desarrollo (Abowd et al., 1997): La evaluación en la fase temprana se produce después de haber tomado las decisiones arquitectónicas iniciales de alto nivel y de alta prioridad, pero sin que exista la arquitectura concreta. En este punto, se pueden evaluar las decisiones preliminares y detectar posibles errores usando cuestionarios y prototipos. La evaluación de la fase intermedia se realiza durante la elaboración del diseño arquitectónico, teniendo en cuenta que esta actividad es un proceso iterativo. Esto implica que una parte de la arquitectura concreta está implementada, dependiendo el estado de avance del desarrollo. En esta etapa se pueden identificar problemas con el diseño arquitectónico establecido, aplicando técnicas basadas en escenarios y listas de verificación.

La evaluación de la fase posterior a la implementación se produce después de que el sistema ha sido diseñado por completo, implementado y desplegado (Bass et al., 2013). En esta etapa, existen tanto la arquitectura como el sistema, por lo tanto se puede responder a preguntas adicionales, tales como si la arquitectura concreta coincide con la arquitectura conceptual, y si el producto ha estado en funcionamiento por un tiempo, también se puede comprobar el desvío entre la arquitectura conceptual y la arquitectura concreta.

La mayoría de los métodos de análisis de la arquitectura usan escenarios, porque es una práctica sistematizadas (Kazman et al., 2000) y una buena manera de sintetizar las interpretaciones individuales sobre un atributo de calidad del software desde una perspectiva común. Esta perspectiva incorpora las características específicas del sistema, siendo más sensible al contexto (Bass et al., 2013). Uno de los primeros métodos de evaluación de arquitectura es SAAM (Software Architecture Analysis Method) (Kazman et al., 1994), que guía la inspección de la arquitectura centrando la atención en problemas potenciales como la especificación incompleta de requisitos o conflicto entre ellos.

SAAM tiene como objetivo verificar si los principios y supuestos arquitectónicos básicos documentados corresponden con los atributos deseados para la aplicación, contribuyendo a la evaluación de los riesgos inherentes a la arquitectura. De este método se extienden: SAAM basado en escenarios complejos (SAAM Founded on Complex Scenarios SAAMCS) (Lassing et al., 1999), la extensión de SAAM a partir de la integración en el dominio (Extending SAAM by Integration in the Domain ESAAMI) (Molter, 1999) y SAAM para la evolución y reutilización (Software Architecture Analysis Method for Evolution and Reusability SAAMER (Lung et al., 1997).

El método de análisis de arquitectura por compensación (Architecture Trade-Off Analysis Method ATAM) (Kazman et al., 2000) tiene como principal objetivo proveer una forma de entender las capacidades de la arquitectura de software, con respecto a múltiples atributos de calidad que compiten entre sí, reconociendo la necesidad de equilibrarlos cuando se especifica la arquitectura del sistema y después de desarrollarlo. ATAM es considerado un marco de referencia para diferentes técnicas de evaluación en función de los atributos de calidad. Integra de una manera eficiente y práctica el mejor modelo teórico individual de cada atributo considerado (Dobrica y Niemelä, 2002).

Utiliza tres tipos de escenario de prueba del sistema desde diferentes puntos de vista arquitectónico: 1) casos de uso, se aplica para la obtención de información e implica usos típicos del sistema; 2) escenarios de crecimiento, que cubren los cambios esperados; y 3) escenarios exploratorios, que cubren los cambios extremos que se esperan en el sistema bajo estrés. Los escenarios cumplen un rol triple en este método, porque son una técnica que ayuda a establecer en términos concretos los requisitos y restricciones que son no claros ni medibles. Además, facilitan la comunicación entre las partes interesadas, ya que los obligan a ponerse de acuerdo sobre su percepción de los requisitos. Por último, los escenarios exploran el espacio definido por el modelo de atributos, ayudando a poner en términos concretos los parámetros del modelo que no forman parte de la arquitectura de software (Dobrica y Niemelä, 2002).

ALMA (Architecture-level modifiability analysis) es otro método para hacer análisis a nivel de arquitectura, sobre la capacidad de un sistema software para ser modificado (Bengtsson et al., 2004), desde la perspectiva de su diseño arquitectónico. Este método está basado en escenarios y, también compara arquitecturas candidatas, evalúa el riesgo asociado a las modificaciones de la arquitectura, y predice el esfuerzo necesario para implementar las modificaciones previstas. Los resultados de la predicción sobre la modificación se evidencian en tres valores: El primero es la predicción del esfuerzo de modificación, el segundo es el mejor y peor caso de esfuerzo previsto para el sistema, y el tercero es el escenario de perfil de cambio. De esta manera, el método de predicción proporciona un marco de referencia que ayuda al arquitecto, en la decisión sobre el nivel de aceptación de la capacidad para ser modificado que tiene el software.

Adicionalmente existen otras técnicas como SACAM (Software Architecture Comparison Analysis Method) (Stoermer et al., 2003), cuyo objetivo es brindar un referente base para el proceso de elección de arquitecturas, mediante la comparación de las capacidades de las arquitecturas candidatas, lo cual la hace muy útil cuando se exploran diseños arquitectónicos para familias de productos o cuando se requiere reutilizar activos software. SBAR (Scenario-Based Architecture Reengineering) (Bengtsson y Bosch, 1998), para estimar el potencial de la arquitectura diseñada para alcanzar los requisitos de calidad del software. ALPSM (Architecture Level Prediction of Software Maintenance) (Bengtsson y Bosch, 1999), para analizar la capacidad de mantenimiento del producto software, a partir de la

observación de su impacto a nivel de la arquitectura en distintos escenarios. SAEM (Software Architecture Evaluation Model) (Dueñas et al., 1998), establece la base para la evaluación de la calidad de la arquitectura del producto software, y la predicción de la calidad del sistema final.

5. Herramientas de ingeniería inversa

Una herramienta de ingeniería inversa es un instrumento que facilita el proceso de recuperación del conocimiento de un sistema software, recreando abstracciones de diseño y arquitectura a partir de código fuente, documentación existente, experiencia de los expertos, y el conocimiento en general a cerca del problema y el dominio de la aplicación (Guéhéneuc et al., 2006). En las dos últimas décadas la ingeniería inversa ha evolucionado representativamente, posibilitando un continuo mejoramiento de los procesos de construcción de software, permitiendo a arquitectos y desarrolladores tener una clara imagen del sistema que están construyendo, disminuyendo la posibilidad de error al facilitar la verificación de la coherencia entre el código desarrollado y la arquitectura propuesta, facilitando a su vez el mantenimiento y la adquisición de conocimiento de sistemas heredados (Canfora y Di Penta, 2007).

Esto se evidencia en el surgimiento de un número cada vez más amplio de herramientas de ingeniería inversa que ofrecen múltiples funcionalidades. Una revisión detallada del estado del arte ha permitido identificar que son múltiples los trabajos que se ha realizado para caracterizar las herramientas de ingeniería inversa. Canfora presenta en 1992 uno de los primeros estudios tendiente a analizar las dificultades que surgen en el uso de documentos producidos por herramientas de Ingeniería Inversa, identificando tres problemas: 1. El nivel de detalle (son demasiado detalladas o son demasiado generales); 2. No producen información para los encargados del mantenimiento; 3. La inflexibilidad de las herramientas (Canfora y Cimitile, 1992).

Storey et al. (1996) documentan un experimento para evaluar dos enfoques opuestos disponibles para la visualización de las estructuras de software en el editor gráfico de la herramienta Rigi, resaltando la importancia que tiene la facilidad de navegación visual entre los documentos generados por las herramientas de ingeniería inversa. Por otra parte, Brown y Wallnau (1996) definen un marco de referencia para evaluar software basado en dos objetivos: Entender las diferencias existentes entre tecnologías y entender cómo esta diferencia se orienta a las necesidades específicas de un contexto determinado. Para lograr dichos objetivos establecen un proceso de tres fases.

La primera es la descripción del modelo, la cual se puede hacer a través de la genealogía de la tecnología o del hábitat del dominio del problema. La segunda es el diseño del modelo, que comprende las actividades de análisis comparativo de características, formulación de la hipótesis y diseño del experimento. Finalmente la

tercera fase corresponde a la evaluación del experimento, para confirmar o refutar la hipótesis. Esta propuesta es utilizada por Gannod y Cheng (1999) para definir un marco de referencia basado en cualidades semánticas, para clasificar y comparar software de ingeniería inversa y técnicas de recuperación de diseño, en el cual la cualidad semántica mide la habilidad del producto de transmitir certeramente información de comportamiento.

Más concretamente, para el caso de la recuperación de arquitecturas y patrones de diseño, Jing Dong et al. (2007) realizan un estudio comparativo entre distintos enfoques, teniendo en cuenta los siguientes aspectos: lo que describe (estructura, comportamiento, semántica), las representaciones intermedias, el grado de coincidencia con el patrón, la representación del resultado final, el grado de automatización (automático, semiautomático) y los patrones que soporta. Por otra parte, Ghulam Rasool et al. (2010) recomiendan directrices basadas en la observación y en la evaluación de diferentes herramientas, que pueden ser utilizadas para comparar las características de los instrumentos existentes y para desarrollar nuevas herramientas de recuperación de patrones de diseño.

Ninguno de los trabajos anteriormente mencionados, analiza de manera integral las características de las herramientas de ingeniería inversa, sólo se limitan a un aspecto determinado, como la salida que generan, las formas de visualización, las diferencias entre tecnologías y la recuperación de arquitecturas y patrones. Desde el punto de vista de caracterización, uno de los trabajos más detallados es el realizado por Bellay y Gall (1997), quienes basados en su experiencia definen cuatro categorías funcionales para evaluar herramientas de ingeniería inversa: Análisis, representación, edición/navegación y capacidades generales, cada una de las cuales a su vez está conformada por un conjunto de criterios específicos.

En la Décima Conferencia Europea sobre Mantenimiento de Software y Reingeniería, Guéhéneuc et al. (2006) plantean la necesidad de contar con un marco de referencia comparativo común bien definido y fácil de comprender, que haga menos complejo el trabajo de evaluación de herramientas de recuperación de diseño. Los autores proponen un marco de referencia comparativo común, construido a partir de su experiencia en calidad de extensión de otros marcos comparativos existente, analizando las siguientes características: contexto en el cual se aplica, intención, tipos de usuarios, tipos de entradas, técnicas utilizadas, tipos de salidas que provee, como está implementada y grado de madurez de la herramienta. Aunque estos trabajos sirven como referentes para caracterizar herramientas, no contemplan nuevos requerimientos que han surgido en los últimos años.

Luego, en la Décimo Quinta Conferencia de Trabajo sobre ingeniería inversa, Jeno Fülöp et al. (2008) presentan su trabajo sobre la aplicación de un enfoque comparativo llamado BEFRIEND (BEnchmark For Reverse engInEering tools workiNg on source coDe), que posteriormente sustenta en calidad de tesis doctoral (Fülöp, 2011), con el que se puede evaluar y comparar con facilidad y eficiencia las salidas de las herramientas de ingeniería inversa, tales como, herramientas de

minería de patrones de diseño, detectores de duplicación de código y verificadores de violación de las reglas. BEFRIEND soporta diferentes tipos de familias de herramientas, lenguajes de programación y sistemas de software, permitiendo a los usuarios definir sus propios criterios de evaluación. Este trabajo se limita a presentar la herramienta, pero no definen un instrumento de caracterización.

En consecuencia, no se ha encontrado en la literatura un instrumento que permita a usuarios, productores e investigadores caracterizar las herramientas de ingeniería inversa, centrando su atención exclusivamente en aquellos criterios particulares que le interesan, con el propósito de facilitar el análisis, la comprensión y la selección de la herramienta adecuada para la toma de decisiones al momento de adquirir, construir o mejorar un producto de este tipo, ocasionando pérdida de tiempo, esfuerzos y recursos.

En este orden de ideas, surge la necesidad de un instrumento, bien definido y fácil de comprender, que sirva a usuarios, productores e investigadores para caracterizarlas en forma clara y completa (Guéhéneuc et al., 2006; Fülöp et al., 2008; Canfora et al., 2011) y que permita a los diseñadores, ingenieros y administradores de proyecto, analizar las tecnologías necesarias que posibiliten ejecutar proyectos de diseño y desarrollo de una manera más avanzada, eficiente y cooperativa (Riba y Molina, 2006; Muñoz y Muñoz, 2010).

5.1. Caracterización de herramientas

En un escenario de recuperación de arquitectura, es necesario identificar las herramientas más adecuadas para lograr este objetivo, por lo cual, los productores de herramientas de ingeniería inversa deben identificar nuevas funcionalidades y características para ofrecer valor agregado a los usuarios; mientras que los investigadores en este campo del conocimiento, desean proponer nuevos métodos, técnicas y herramientas, que representen mejoras en los procesos de recuperación de conocimiento de sistemas software, para lo cual requieren identificar en forma precisa las características de las herramientas disponibles.

Por lo tanto, usuarios de herramientas de ingeniería inversa, productores e investigadores, requieren de un instrumento que facilite la toma de decisión ante cada una de las situaciones planteadas. Como respuesta a esta necesidad se propone un modelo de caracterización de herramientas de ingeniería inversa (Monroy et al., 2012), basado en las propuestas de Guéhéneuc et al. (2006), Bellay y Gall (1997).

La caracterización de las herramientas de ingeniería inversa se hace teniendo en cuenta dos criterios: el primero corresponde al aspecto estructural, centrando la atención en los elementos básicos que conforman la arquitectura genérica de este tipo de herramientas: Analizador e Interfaz (Chikofsky y Cross, 1990), mientras que el segundo se centra en las propiedades comunes entre ellas.

Aspecto estructural: Para caracterizar las herramientas de ingeniería inversa, tomando como principal referente su arquitectura genérica, se establece la siguiente estructura: elemento, función, aspecto y característica, como se observa en la Tabla 2 se han definido dos tipos de elementos: Analizador e Interfaz. El primero es el componente responsable de la transformación del software objeto de estudio a un mayor nivel de abstracción, mientras que la interfaz se encarga de permitir la interacción entre el usuario y la herramienta.

Una función corresponde a las tareas que realiza cada elemento de la arquitectura, por ejemplo el analizador cumple con tareas de entrada, proceso y salida. El aspecto indica un matiz o rasgo diferenciador para cada función que cumplen los elementos, como es el caso de la función entrada que cumple el analizador, para la cual se definen cinco aspectos: los supuestos de la entrada, la fuente, la precisión, la automatización y el manejo de versiones. Por último se encuentra la característica, entendida como una cualidad que sirve para distinguir a cada uno de los aspectos identificados para las funciones que cumplen los elementos.

En la Tabla 2 se presenta la estructura de caracterización de las herramientas de ingeniería inversa, tomando como referente principal los aspectos estructurales. Las características marcadas con un signo más (+) han sido establecidas por Guéhéneuc et al. (2006) y las que tienen un asterisco (*) han sido definidas por Bellay y Gall (1997). El principal aporte del este trabajo fue definir la estructura de caracterización para facilitar el análisis, la comprensión y selección de las herramientas, de tal manera que se puedan identificar sus ventajas y desventajas. A continuación sólo se explican aquellas características que se han incluido.

Para la función de entrada del analizador se incluyeron dos características nuevas asociadas a la fuente. La primera hace referencia al tipo de fuente, asumiendo que en la actualidad el proceso de ingeniería inversa incluye, además del código fuente, colecciones de datos y documentación. Adicionalmente, se ha definido la relación entre las características: tipo, modelo y representación de la fuente, teniendo en cuenta que para cada tipo de fuente existe un modelo que la describe y una forma de representación, como se muestra en la Tabla 3.

La segunda característica corresponde al tipo de aplicación, los posibles tipos son: aplicación de escritorio, aplicación web, aplicación RIA (Aplicación Rica de Internet), servicio, aplicación móvil y aplicación para televisión digital. Se utiliza para definir la naturaleza de la tecnología de interacción, según la experiencia de usuario y la tecnología con la que está implementada la aplicación objeto de estudio, con el ánimo de facilitar la identificación del uso de patrones arquitectónicos.

Elemento	Función	Aspecto	Características
Analizador	Entrada	Supuestos (+)	Estructurales, Semánticos
		Fuente	Tipo, Modelo (+), Representación (+), Origen de datos, Tipos de datos(+) , Tipos de aplicación
		Precisión (+)	Semántica, Sintáctica
		Automatización (+)	Automática, Semiautomática, Manual
		Versiones (+)	Múltiples, Singular
	Proceso	Técnica (+)	Adaptabilidad, Automatización, Complejidad, Determinismo, Explicación, Tratamiento de aspectos indefinidos, Nivel de detalle, Incremental, Iterativo, Independiente del lenguaje, Madurez, Método, Modelo, Escalabilidad, Semántica
		Tipo de analizador	Dinámicos, Estáticos, Híbridos, Históricos
		Aspectos generales	Reconocimiento de patrones, Artefactos que recupera, Análisis incremental (*), Reparsing (*), Preprocesador de comandos configurable (*), Soporte para conmutadores de compilador(*), Capacidad para analizar Funciones/Variables externas (*), Capacidad para trabajar con información incompleta, Capacidad para trabajar con caja negra, Identificación de clonación, Ingeniería inversa de Documentos WSDL en UML
	Salida	Tipo	Modelo, Representación
		Reporte (+)	Legibilidad, Nivel de detalle, Modelo, Calidad, Representación, Tipo de datos
Interfaz	Visualización	Trazabilidad	Horizontal, Vertical
		Tipo (*)	Estática, dinámica
		Tipo de Navegación (*)	Capas, Ojo de pescado, SHriMP: Simple Hierarchical Multi Perspective view
		Calidad (+)	Sintáctica, Semántica
		Niveles de abstracción	Código, Diseño, Arquitectura, Requerimientos, Modelo del negocio
		Análisis de los resultados	Capacidades de hipertexto (*), Capacidad para analizar diferencias, Mecanismos de consulta, Capacidad para agrupar cambios relacionados, Capacidad para clasificar y combinar información desde repositorios de software diferentes y heterogéneos
	Edición	Adaptabilidad del reporte generado	Anotaciones de ayuda (*), Permite organizar el informe (*), Capacidad para mejorar el diseño del reporte de salida (*), Facilidades de edición
		Capacidad de salida (*)	Generación automática de documentación, Formatos de exportación, Impresión de informes
		Generales	Definición de proyectos, Función de búsqueda, Capacidad para seleccionar entradas representativas

Tabla 2. Estructura de caracterización de herramientas de ingeniería inversa

Fuente		
Tipo	**Modelo**	**Representación**
Código Fuente	Paradigma de programación	Lenguajes de programación
Colección de datos	Relacional, jerárquico, etc.	SQL, XML, etc.
Documentación	UML, ADL, Lenguaje natural	Gráficos, texto

Tabla 3. Relación entre el tipo, el modelo y la representación de la fuente

Adicionalmente, en los aspectos generales de la función proceso se han incluido características, que corresponden a los retos actuales que deben atender las herramientas de ingeniería inversa, tales como la capacidad para trabajar con información incompleta, la capacidad para trabajar con caja negra para el caso concreto de los sistemas orientados a servicios, la capacidad para identificar clonación de código facilitando el trabajo de reutilización y la comprobación del manejo de derechos de autor; así como también la capacidad para realizar tareas de ingeniería inversa a partir de documentos WSDL.

También se agregaron a los aspectos generales características que permiten calificar de manera más detallada al analizador de una herramienta de ingeniería inversa: Reconocimiento de patrones y Artefactos recuperados. El primero indica si la herramienta reconoce o no la existencia del uso de patrones a partir de la entrada que se le suministra, en caso afirmativo se debe especificar el tipo de patrón reconocido (de diseño, arquitectónico, etc.) y los nombres de los patrones concretos que identifica según el tipo (singleton, Abstract factory, etc).

Los artefactos recuperados corresponden a las piezas de información que recupera el analizador, está directamente relacionado con su tipo, el modelo y la representación de la salida. Para el caso de un analizador estático que utilice como modelo UML, la representación de la salida podría ser a través de diagramas de clase, estructuras compuestas, de paquetes, de componentes o de despliegue; mientras que para un analizador dinámico que utilice el mismo modelo, la representación de la salida podría ser diagramas de casos de uso, de secuencia, comunicación, actividades, máquinas de estado, general de interacción, procesos, estado o tiempos.

Para la función que representa la salida del analizador se ha incluido el tipo de salida, que es determinado por el tipo de analizador, por ejemplo, para un analizador estático el tipo de salida puede corresponder a una vista estática, cuyo modelo en UML está representado a por medio de un diagrama de clases.

El segundo componente representativo de cualquier herramienta de ingeniería inversa es la interfaz de usuario, la cual permite realizar funciones de visualización y edición. En el caso de la visualización, se presenta con frecuencia la necesidad de rastrear los modelos obtenidos como resultado de la ingeniería inversa, desde el código fuente hasta el modelo del dominio del problema, pasando por el diseño y los requerimientos, lo que se conoce como trazabilidad horizontal. Del mismo modo, es necesario rastrear los distintos artefactos obtenidos para cada una de estas actividades (trazabilidad vertical), por ejemplo cuando a nivel de diseño se hace el seguimiento de la estructura de un componente por medio de su diagrama de clases y su comportamiento a través de un diagrama de secuencia (Pfleeger, 2010).

También se incluye un nuevo aspecto denominado nivel de abstracción, que hace referencia a la capacidad que tiene la herramienta para permitir visualizar los resultados que arroja, desde el código fuente hasta el modelo del negocio, pasando

por el diseño, las vistas arquitectónicas y los requerimientos del sistema. Además, se incorporan nuevas características que permiten establecer la capacidad de análisis de resultados de la herramienta, a través de mecanismos de consulta, de análisis de diferencias, de agrupación de cambios relacionados y de clasificación y combinación de información desde repositorios de software diferentes y heterogéneos.

Con respecto a las capacidades de edición de la herramienta se incluyeron características como la facilidad de edición, que implica acciones como cortar, copiar, pegar y deshacer, entre otras que permiten la adaptabilidad del reporte. Igualmente se agregaron aspectos generales de edición, tales como: la posibilidad de definir proyectos para agrupar resultados según la necesidad del usuario; la capacidad para realizar búsquedas en los artefactos recuperados y la capacidad para seleccionar entradas representativas, dándole mayor libertad al usuario y haciendo más eficiente su trabajo.

Propiedades comunes: Para el segundo criterio que se utilizó en la caracterización se establece un factor y una propiedad, como se observa en la Tabla 4. El factor corresponde a un matiz o rasgo diferenciador de la herramienta, mientras que la propiedad hace referencia a una cualidad que distingue a cada uno de los factores identificados para las herramientas. Todas las propiedades identificadas han sido definidas previamente por Guéhéneuc et al. (2006) y aquellas marcadas con un asterisco (*) en la Tabla 4 han sido descritas por Bellay y Gall (1997). En este trabajo se distinguen cuatro aspectos: Contexto, intención, perfil del usuario y madurez de la herramienta.

Factor	Propiedades
Contexto	Tiempo de vida, Metodología, Campo de acción, Escenario, Universo del discurso
Intención	Objetivos a largo plazo, Objetivos a corto plazo
Perfil del Usuario	Conocimiento, Experiencia, Prerrequisitos, Usuario objeto, Tipo de usuario
Madurez de la Herramienta	Calidad, Tipo de licencia, Independencia del lenguaje, Documentación, Plataformas soportadas (*), Editor Integrado o externo (*), Navegador Integrado o externo (*), Capacidad de almacenamiento (*), Base de usuarios, Soporte a múltiples usuarios

Tabla 4. Propiedades Comunes

El contexto define el entorno y las restricciones externas de la herramienta, la intención establece los objetivos para los cuales ha sido desarrollada la herramienta, el perfil de usuario indica las habilidades y características que debe cumplir la persona que utilice la herramienta, mientras que la madurez de la herramienta hace referencia al conjunto propiedades que determinan el grado con el cual la herramienta cumple con los requerimientos técnicos del usuario.

El instrumento propuesto además de facilitar el análisis y entendimiento de las herramientas de ingeniería inversa, permite la valoración independiente y conjunta de cada uno de sus componentes. No pretende establecer un modelo terminado de caracterización, sino un instrumento dinámico que debe ser permanentemente actualizado para que atienda a las necesidades del momento. Se recomienda utilizarlo cuando se requiera caracterizar herramientas de ingeniería inversa, teniendo en cuenta que no es indispensable valorar todas las características identificadas, más bien, se sugiere establecer claramente cuáles son pertinentes a partir del objetivo propuesto para la valoración.

Es de utilidad para usuarios de herramientas de ingeniería inversa, porque ofrece una plantilla de características que brindan este tipo de herramientas, de tal manera que puede seleccionar aquellas de su particular interés para establecer un comparativo entre las herramientas existentes, sirviéndole de referente y como instrumento de apoyo para la toma de decisión al momento de escoger la más apropiada. Del mismo modo, este instrumento puede ser utilizado por los productores, para caracterizar la herramienta que desarrollan, identificando sus fortalezas para mostrarlas como estrategia de mercadeo y las debilidades para establecer los aspectos a mejorar y futuros desarrollos.

5.2. Modelo ontológico para contextos de uso de las herramientas

El número representativo de herramientas de ingeniería inversa identificadas en la revisión de la literatura, evidencia la dificultad para seleccionar las herramientas más adecuadas ante una situación específica, como lo plantea la metodología. Por eso se definió el modelo ontológico que se explica a continuación (Monroy et al., 2016), usando como referente el modelo de caracterización de herramientas expuesto. La implementación del modelo se realizó en Protégé 5.0 (Stanford, 2015)

El dominio de la ontología está determinado por los contextos de uso de las herramientas de ingeniería inversa en el ámbito de la ingeniería de software. Esta ontología se utiliza para facilitar las tareas de toma de decisión, con respecto al uso de herramientas de ingeniería inversa, a partir de las circunstancias que describen el contexto de uso, por lo tanto, está orientada para ser utilizada por ingenieros de software, docentes y estudiantes en este campo del conocimiento, además de los profesionales relacionados con los contextos de uso establecidos.

En consecuencia, la ontología ha sido diseñada principalmente para responder la pregunta ¿Qué herramientas de ingeniería inversa se pueden utilizar ante un contexto de uso determinado?. Además, puede responder entre otras a las siguientes preguntas: ¿Qué herramientas cumplen con una función específica?, ¿Qué herramientas cumplen con una característica determinada?, ¿Qué funciones se tienen en cuenta en un contexto específico? ¿Qué características se tienen en cuenta en un contexto específico?.

El modelo ontológico se representa en la Figura 6 usando la notación de UML por medio de un modelo de dominio, presentando las clases, su jerarquía y las relaciones entre ellas, manifestando la semántica descrita a continuación. La educación, la producción de software, la seguridad informática y la computación forense son contextos de uso que requieren de herramientas de ingeniería inversa, teniendo en cuenta que la producción de software a su vez tiene procesos como el mantenimiento, la documentación, la verificación de software y la reutilización.

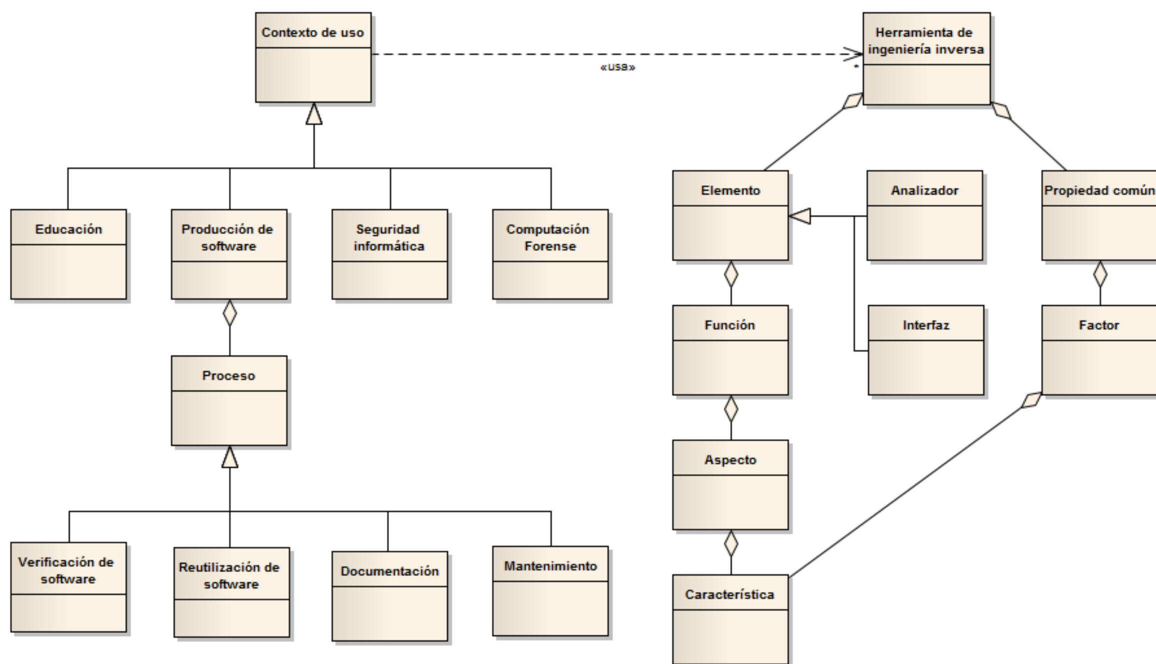

Figura 6. Modelo ontológico

En su estructura, según la caracterización hecha por Monroy et al. (2012), las herramientas de ingeniería inversa tienen dos elementos: el analizador, responsable de la conversión del código fuente a un nivel mayor de abstracción, y la interfaz que permite la interacción entre el usuario y la herramienta. Una función corresponde a las tareas que realiza cada elemento de la arquitectura, por ejemplo el analizador cumple con tareas de entrada, proceso y salida. El aspecto indica un matiz o rasgo diferenciador para cada función que cumplen los elementos. La característica, entendida como una cualidad que sirve para distinguir a cada uno de los aspectos identificados para las funciones que cumplen los elementos, y los factores que distinguen las propiedades comunes a cualquier producto software, como es el caso de las herramientas de ingeniería inversa.

Las características de los contextos de uso se establecen a partir de la propuesta de caracterización de herramientas de ingeniería inversa (Monroy et al., 2012), según la

cual se definen 82 características clasificadas teniendo en cuenta el elemento, la función y el aspecto. Debido a que no todas las características son determinantes del contexto de uso, para cada una se evaluó su nivel de pertinencia para cada contexto de uso (M: Mantenimiento, V: Verificación de software, Dc: Documentación, R: Reutilización, S: Seguridad informática, E: Educación y C: Computación forense), obtenido los resultados relacionados en la Tabla 5, donde se omiten las características valoradas como no determinantes.

Esto se logró estableciendo la siguiente escala de valoración: La característica es obligatoria (O) para el contexto de uso si al no cumplirse no se le logra el propósito del contexto de uso; la característica es necesaria (N), si contribuye al logro del propósito, pero si no se cumple no implica que no se logre el objetivo del contexto de uso; la característica es deseada (D) si ayuda a mejorar los resultados del propósito del contexto de uso; y finalmente, la característica se valora como no determinante (ND) si no afecta para nada el cumplimiento del propósito del contexto de uso.

Tabla 5. Características de los contextos de uso

	Producción de Sw				***S***	***E***	***C***
Características	***M***	***V***	***Dc***	***R***			
Explicación de la técnica	D	D	D	ND	ND	N	ND
Tratamiento de aspectos indefinidos	D	ND	D	ND	N	ND	O
Tipo de analizador estático	N	N	N	D	N	N	O
Tipo de analizador dinámico	D	D	D	D	D	D	N
Tipo de analizador Híbrido	D	D	D	D	D	D	N
Tipo de analizador Histórico	D	ND	D	D	ND	D	D
Reconocimiento de Patrones	N	N	D	O	D	D	D
Capacidad para analizar Funciones/Variables Externas	D	ND	ND	D	O	ND	D
Capacidad para trabajar con información incompleta	D	D	D	D	O	D	O
Capacidad para trabajar con caja negra	D	D	D	D	D	D	O
Identificación de clonación	N	D	ND	N	D	D	O
Ingeniería Inversa de documentos WSDL en UML	D	D	D	ND	ND	D	D
Legibilidad del reporte	N	N	O	N	N	O	D
Calidad del reporte	N	N	N	N	D	N	N
Trazabilidad Horizontal	O	O	D	ND	N	D	D
Trazabilidad Vertical	O	O	N	D	N	D	D
Visualización Estática	N	N	N	N	N	N	N
Visualización Dinámica	N	N	N	D	N	N	N
Niveles de Abstracción: Código	O	O	D	N	N	O	D
Niveles de Abstracción: Diseño	O	O	O	N	N	O	D
Niveles de Abstracción: Arquitectura	N	N	N	D	N	D	D
Niveles de Abstracción: Requerimientos	D	D	D	D	D	D	D
Niveles de Abstracción: Modelo del Negocio	D	D	D	D	D	D	D
Capacidades de Hipertexto	N	N	N	D	D	N	D
Capacidad para analizar diferencias	N	N	D	D	ND	D	N
Mecanismos de consulta	N	N	D	D	D	D	N
Capacidad para agrupar cambios relacionados	N	N	N	D	ND	N	D

Características	Producción de Sw				S	E	C
	M	V	Dc	R			
Anotaciones de ayuda	D	D	D	ND	D	D	ND
Permite organizar el informe	N	N	N	ND	D	N	ND
Generación automática de documentación	N	N	O	ND	D	N	ND
Función de Búsqueda	N	N	N	N	N	N	N
Capacidad para seleccionar entradas representativas	N	N	N	D	D	D	D

Glosario de Términos

Actividad: Comprende un conjunto de operaciones o tareas propias de una persona o entidad

Ámbito: Espacio ideal configurado por las cuestiones y los problemas de una o varias actividades o disciplinas relacionadas entre sí.

Analizador: Mecanismo que permita la separación de las partes de algo para conocer su composición.

Analizador dinámico: Mecanismo que permite conocer las partes que describen el comportamiento de un producto software

Analizador estático: Mecanismo que permite conocer las partes que describen la estructura de un producto software

Analizador híbrido: Mecanismo que permite conocer las partes que describen la estructura y el comportamiento de un producto software

Analizador histórico: Mecanismo que permite conocer las partes que describen la evolución de producto software a partir de sus versiones.

Analizador lingüístico: Responsable de la verificación de la sintaxis y la gramática de las consultas escritas en lenguaje natural que se registran en el mecanismo de consulta.

Arquitectura: Es la organización fundamental de un sistema encarnada en sus componentes, las relaciones que existen entre ellos y su entorno, y los principios que rigen su diseño y evolución (ISO/IEC/IEEE42010).

Arquitectura conceptual: Arquitectura que existe en la mente del arquitecto y que posiblemente está documentada. También es conocida como arquitectura lógica, diseñada o idealizada.

Arquitectura concreta: Corresponde a la arquitectura que se deriva directamente del código fuente. Algunos autores la denominan arquitectura física, construida, realizada o implementada.

Arquitectura hipotética: Es la arquitectura que se define al comienzo del proceso de recuperación, a partir del conocimiento previo que se tiene del sistema, cuando se utilizan enfoques "Arriba-abajo" (de lo abstracto a lo concreto) o enfoques híbridos.

Artefacto: Es la especificación de una pieza física de información que es usada o producida en el proceso de desarrollo de software, o en el despliegue y operación del sistema.

Componente: Elemento que representa una unidad computacional en tiempo de ejecución.

Comportamiento: Descripción sobre cómo la interacción entre los elementos afecta al sistema en un momento determinado o cuando se encuentra en un estado, también se denomina parte dinámica y revela aspectos relacionados con el orden de la interacción entre los elementos, la concurrencia y la dependencia de la interacción con respecto al tiempo.

Conector: Sinónimo de contrato.

Consulta original: Pregunta escrita en lenguaje natural.

Consulta procesada: Expresión XQuery que representa la consulta original.

Consulta compuesta: Pregunta en lenguaje Natural que involucra propiedades cualitativas individuales y colectivas del sistema, que puede ser respondida a partir de la semántica del modelo, pero que no consiguen ser resueltas solamente con consultas XQuery.

Consulta simple: Pregunta en lenguaje natural que pueden ser respondidas a partir de las propiedades estructurales de los modelos representados en XMI.

Contexto: Entorno físico o de situación de cualquier índole en el que se considera un hecho.

Contrato: Es un conjunto de requisitos y limitaciones de uno o más componentes que definen un comportamiento colectivo y derivado de las interfaces participantes.

Descripción de la arquitectura: Producto de trabajo usado para expresar una arquitectura.

Dominio de la aplicación: Campo del conocimiento para el cual ha sido construido el software.

Elemento arquitectónico: Parte constitutiva de la arquitectura.

Escenario: Es una configuración dinámica de los elementos arquitectónicos. Los escenarios permiten encadenar muchos componentes y contratos/conectores entre

sí para formar estructuras de ejecución completas que modelan los aspectos dinámicos del sistema.

Estructura arquitectónica: Conjunto de elementos arquitectónicos que pueden ser software o hardware.

Instrumento: Elemento de apoyo que sirven para guiar o registrar las actividades y los resultados obtenidos en cada una de ellas.

Interesado (Stakeholder): Individuo, grupo, organización (o clases de ellos) que tienen un interés sobre el sistema.

Interés: Asunto sobre el sistema relevante para uno o más interesados.

Mecanismo: Conjunto de partes de una máquina en su disposición adecuada para el cumplimiento de una función específica.

Mecanismo de consulta: Estructura que organiza sus partes constitutivas para permitir la formulación de preguntas, con la respectiva representación de las respuestas.

Modelo conceptual: Es una representación externa, precisa, completa y consistente con el conocimiento científicamente compartido, que facilita la comprensión o enseñanza de sistemas o estados de las cosas del mundo

Motor de consulta: Mecanismo responsable de la ejecución de la consulta procesada sobre el repositorio que contiene los modelos.

Nivel de abstracción: Es el grado de cercanía existente entre la realidad y su representación, con base en un lenguaje definido sobre el que es posible visualizar, construir y documentar los artefactos de un sistema software.

Punto de vista arquitectónico: Conjunto de convenciones para construir, interpretar, usar y analizar un tipo de vista arquitectónica, que enmarca las preocupaciones específicas del sistema. Incluye clases de modelos, lenguajes y notaciones, métodos de modelado y técnicas de análisis para enmarcar un conjunto específico de intereses.

Problema: Conjunto de hechos o circunstancias que dificultan la consecución de algún fin.

Propósito: Conjunto de intenciones que tienen los interesados con respecto al contexto.

Recurso: Elemento disponible para resolver una necesidad o cumplir con un propósito.

Situación: Conjunto de factores o circunstancias que afectan a alguien o algo en un determinado momento.

Unidad computacional: es una parte ejecutable del sistema. Ejemplos: instrucciones, bloques básicos, rutinas, objetos, unidades de compilación, componentes, módulos y subsistemas.

Vista arquitectónica: Producto de trabajo que expresa la arquitectura desde la perspectiva de un conjunto específico de intereses/preocupaciones. Representa un conjunto de elementos arquitectónicos y las relaciones entre ellos

Vista objetivo: Vista arquitectónica que se desea recuperar.

Bibliografía

Abi-Antoun, M., & Barnes, J. M. (2010). Analyzing security architectures. Paper presented at the ASE'10 - Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, 3-12.

Abowd, G., Bass, L., Clements, P., Kazman, R., & Northrop, L. (1997). Recommended Best Industrial Practice for Software Architecture Evaluation (No. CMU/SEI-96-TR-025). Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Allen R., A Formal Approach to Software Architecture, Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMUCS-97-144, May 1997.

Anquetil, N., & Lethbridge, T. C. (1999, October). Experiments with clustering as a software remodularization method. In Reverse Engineering, 1999. Proceedings. Sixth Working Conference on (pp. 235-255). IEEE.

Automatedq, Memory and Performance Profiling AQtime Pro, recuperada 31 de enero de 2016, disponible en: http://www.automatedqa.com/products/aqtime/.

Bass, L.; Clements, P. & Kazman, R. (2013), Software Architecture in Practice. Third edition. Addison-Wesley

Bengtsson, P., & Bosch, J. (1998), Scenario-Based Architecture Reengineering, Proc. Fifth Int'l Conf. Software Reuse (ICSR 5).

Bengtsson, P., & Bosch, J. (1999). Architecture level prediction of software maintenance. In Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on (pp. 139-147). IEEE.

Bengtsson, P., Lassing, N., Bosch, J., & van Vliet, H. (2004). Architecture-level modifiability analysis (ALMA). Journal of Systems and Software, 69(1), 129-147.

Binns, P., Englehart, M., Jackson, M., & Vestal, S. (1996). Domain-specific software architectures for guidance, navigation and control. International Journal of Software Engineering and Knowledge Engineering, 6(02), 201-227.

Bojic, D., & Velasevic, D. (2000, February). A use-case driven method of architecture recovery for program understanding and reuse reengineering. In csmr (p. 23). IEEE.

Cai, Y., Wang, H., Wong, S., & Wang, L. (2013, June). Leveraging design rules to improve software architecture recovery. In Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures(pp. 133-142). ACM.

Callo Arias, T. B., Avgeriou, P., America, P., Blom, K., & Bachynskyy, S. (2011). A top-down strategy to reverse architecting execution views for a large and complex software-intensive system: An experience report. Science of Computer Programming, 76(12), 1098-1112.

Canfora G. y M. Di Penta, New Frontiers of Reverse Engineering, IEEE Future of Software Engineering, 326 – 341 (2007).

Canfora, G., Di Penta, M., & Cerulo, L. (2011). Achievements and challenges in software reverse engineering. Communications of the ACM, 54(4), 142-151.

Clarke, E., Kroening, D., & Yorav, K. (2003, June). Behavioral consistency of C and Verilog programs using bounded model checking. In Design Automation Conference, 2003. Proceedings (pp. 368-371). IEEE.

Clements, P., A survey of architecture description languages, Proceedings of the 8th international workshop on software specification and design. IEEE Computer Society, 1996.

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., & Little, R. (2002). Documenting software architectures: views and beyond. Pearson Education.

Clements, P., Garlan, D., Little, R., Nord, R., & Stafford, J. (2003, May). Documenting software architectures: views and beyond. In ICSE (Vol. 3, pp. 740-741).

De Boer, R. C., & van Vliet, H. (2008). Architectural knowledge discovery with latent semantic analysis: Constructing a reading guide for software product audits. Journal of Systems and Software, 81(9), 1456-1469.

Dobrica, L., & Niemelä, E. (2002). A survey on software architecture analysis methods. Software Engineering, IEEE Transactions on, 28(7), 638-653.

Ducasse, S., & Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. IEEE Transactions on Software Engineering, 35(4), 573-591.

Dueñas, J. C., de Oliveira, W. L., & Juan, A. (1998). A software architecture evaluation model. In Development and Evolution of Software Architectures for Product Families (pp. 148-157). Springer Berlin Heidelberg.

Eisenbarth, T., Koschke, R., & Simon, D. (2003). Locating features in source code. IEEE Transactions on Software Engineering, 29(3), 210-224.

Favre, J. M. (2004, November). Cacophony: Metamodel-driven software architecture reconstruction. In Reverse Engineering, 2004. Proceedings. 11th Working Conference on (pp. 204-213). IEEE.

Gannod G. y B. Cheng, A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques, 6th Working Conference on Reverse Engineering, 77 – 88, Atlanta, Georgia USA, 6 al 8 de octubre (1999)

Garlan, D., Allen, R., & Ockerbloom, J. (1994). Exploiting style in architectural design environments. ACM SIGSOFT Software Engineering Notes, 19(5), 175-188.

Garlan, D., Monroe R. y D. Wile, "Acme: an architecture description interchange language." CASCON, noviembre 1997.

Guéhéneuc Y., Mens K. y R. Wuyts, A Comparative Framework for Design Recovery Tools, Proceedings of the 10th European Conference on Software Maintenance and Reengineering, 134 – 143, Bari, Italia, 22 al 24 de marzo (2006).

Harris, D.R.; Reubenstein, H.B. y A.S. Yeh, Reverse Engineering to the Architectural Level, Proc. Int'l Conf. Software Eng. (ICSE), 1995.

Holt, R. C. (1998, October). Structural manipulations of software architecture using Tarski relational algebra. In Reverse Engineering, 1998. Proceedings. Fifth Working Conference on (pp. 210-219). IEEE.

Holy, L., Snajberk, J., Brada, P., & Jezek, K. (2013, September). A Visualization Tool for Reverse-Engineering of Complex Component Applications. In 2013 IEEE International Conference on Software Maintenance (pp. 500-503). IEEE.

Jacobson, I., Booch, G. & Rumbaugh, J., (2000), The Rational Unified Process. Addison Wesley.

Kazman, R. , & Carriere, S. J. (1998). Perils of reconstructing architectures. International Software Architecture Workshop, Proceedings, ISAW, , 13-16.

Kazman, R. y L. Bass, Categorizing Business Goals for Software Architectures, technical report, Carnegie Mellon Univ., SEI, 2005.

Kazman, R., Bass, L., Webb, M., & Abowd, G. (1994, May). SAAM: A method for analyzing the properties of software architectures. In Proceedings of the 16th international conference on Software engineering (pp. 81-90). IEEE Computer Society Press.

Kazman, R., Carrière, S. J., & Woods, S. G. (2000). Toward a discipline of scenario-based architectural engineering. Annals of software engineering, 9(1-2), 5-33.

Kazman, R., Klein, M., & Clements, P. (2000). ATAM: Method for architecture evaluation (No. CMU/SEI-2000-TR-004). Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

Kienle, H. M., & Müller, H. A. (2010). Rigi—An environment for software reverse engineering, exploration, visualization, and redocumentation. Science of Computer Programming, 75(4), 247-263.

Kim, T. H., Kim, K., & Kim, W. (2010, July). An interactive change impact analysis based on an architectural reflexion model approach. In Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual(pp. 297-302). IEEE.

Koppler, R. (1997). A systematic approach to fuzzy parsing. Software-Practice and Experience, 27(6), 637-650.

Kruchten, P., The 4+1 View Model of Architecture, IEEE Software, 12 (6), Nov.1995, IEEE, pp.42-50.

Kuhn, A., Ducasse, S., & Girba, T. (2005, November). Enriching reverse engineering with semantic clustering. In Reverse Engineering, 12th Working Conference on (pp. 10-pp). IEEE.

Lanza, M. (2003, March). Codecrawler-lessons learned in building a software visualization tool. In Software Maintenance and Reengineering, 2003. Proceedings. Seventh European Conference on (pp. 409-418). IEEE.

Lassing, N., Rijsenbrij, D., & van Vliet, H. (1999, August). On software architecture analysis of flexibility, Complexity of changes: Size isn't everything. In Proc. Second Nordic Software Architecture Workshop NOSA (Vol. 99, pp. 1103-1581).

Löwe, W., Ericsson, M., Lundberg, J., Panas, T., & Pettersson, N. (2003). Vizzanalyzer-a software comprehension framework. Software Engineering Research and Practice-SERPS, 3.

Luckham, D.C., J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan and W. Mann, “Specification and analysis of system architecture using RAPIDE”, IEEE Transactions on Software Engineering, 21(4), 336–355, April 1995

Lundberg, J., & Löwe, W. (2003). Architecture recovery by semi-automatic component identification. Electronic Notes in Theoretical Computer Science,82(5), 98-114.

Lung, C. H., Bot, S., Kalaichelvan, K., & Kazman, R. (1997, November). An approach to software architecture analysis for evolution and reusability. In Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research (p. 15). IBM Press.

Mancoridis, S., Mitchell, B. S., Chen, Y., & Gansner, E. R. (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. In Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on (pp. 50-59). IEEE.

Medvidovic, N. y V. Jakobac, Using Software Evolution to Focus Architectural Recovery, Automated Software Eng., vol. 13, no. 2, pp. 225-256, 2006, doi:10.1007/s10515-006-7737-5.

Medvidovic, N., Egyed, A. y P. Gruenbacher, Stemming Architectural Erosion by Architectural Discovery and Recovery, Proc. Int'l Workshop from Software Requirements to Architectures, 2003.

Mei, H., & Huang, G. (2004, May). PKUAS: an architecture-based reflective component operating platform. In Distributed Computing Systems, 2004. FTDCS 2004. Proceedings. 10th IEEE International Workshop on Future Trends of (pp. 163-169). IEEE.

Mens, K. (2000). Automating architectural conformance checking by means of logic meta programming (Doctoral dissertation, PhD thesis, Vrije Universiteit Brussel).

Molter, G. (1999, August). Integrating SAAM in domain-centric and reuse-based development processes. In Proceedings of the 2nd Nordic Workshop on Software Architecture, Ronneby (pp. 1-10).

Moonen, L. (2003, September). Exploring software systems. In Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on (pp. 276-280). IEEE.

Moriconi, M. y R. A. Riemenschneider, Introduction to SADL 1.0: A Language for Specifying Software Architecture Hierarchies. Technical Report SRI-CSL-97-01, SRI International, March 1997.

Muñoz, D. F., & Muñoz, D. F. (2010). Planeación y control de proyectos con diferentes tipos de precedencias utilizando simulación estocástica.Información tecnológica, 21(4), 25-33.

Murphy, G. C., & Notkin, D. (1997). Reengineering with reflexion models: A case study. Computer, 30(8), 29-36.

Murphy, G. C., Notkin, D., & Sullivan, K. J. (2001). Software reflexion models: Bridging the gap between design and implementation. Software Engineering, IEEE Transactions on, 27(4), 364-380.

OMG, Diagram Definition, Versión 1.1., 2015. Recuperado 13 de enero 2016, disponible en http://www.omg.org/spec/DD/1.1/

OMG, Introduction to omg's unified modeling language® (UML®), 2005. Recuperado 1 de noviembre de 2015, diponible en: http://www.uml.org/what-is-uml.htm

OMG, OMG Meta Object Facility (MOF) Core Specification, versión 2.5, 2015, recuperado 13 de enero 2016, disponible: http://www.omg.org/spec/MOF/2.5/

OMG, Unified Modeling Language. Versión 2.5. 2015. Recuperado 13 de enero 2016, disponible en: http://www.omg.org/spec/UML/2.5/

Oracle, HPROF: A Heap/CPU Profiling, recuperada 2 de febrero de 2016, disponible en: Toolhttps://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html

Oracle, Platform Debugger Architecture (JPDA), recuperado junio 13 de 2015, disponible en: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/

Pacione, M. J., Roper, M., & Wood, M. (2004, November). A novel software visualisation model to support software comprehension. In Reverse Engineering, 2004. Proceedings. 11th Working Conference on (pp. 70-79). IEEE.

Panas, T., Löwe, W., & Aßmann, U. (2003, June). Towards the Unified Recovery Architecture for Reverse Engineering. In Software Engineering Research and Practice (pp. 854-860).

Pashov, I., y Riebisch, M. (2004, May). Using feature modeling for program comprehension and software architecture recovery. In Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the (pp. 406-417). IEEE.

Pinzger, M., "ArchView—Analyzing Evolutionary Aspects of Complex Software Systems," PhD thesis, Vienna Univ. Of Technology, 2005.

Pinzger, M., Gall, H., Girard, J. F., Knodel, J., Riva, C., Pasman, W., y Wijnstra, J. G. (2003). Architecture recovery for product families. In Software Product-Family Engineering (pp. 332-351). Springer Berlin Heidelberg.

Postma, A. (2003). A method for module architecture verification and its application on a large component-based system. Information and Software Technology, 45(4), 171-194.

Raza, A., Vogel, G., & Plödereder, E. (2006). Bauhaus–a tool suite for program analysis and reverse engineering. In Reliable Software Technologies–Ada-Europe 2006 (pp. 71-82). Springer Berlin HeidelbergMurphy, G. C. y Notkin, D. (1996).

Risi, M., Scanniello, G., & Tortora, G. (2010, September). Architecture recovery using latent semantic indexing and k-means: an empirical evaluation. In Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on (pp. 103-112). IEEE.

Riva, C. (2002). Architecture reconstruction in practice. In Software Architecture (pp. 159-173). Springer US.

Riva, C., View-Based Software Architecture Reconstruction, PhD thesis, Technical Univ. of Vienna, 2004.

Rumbaugh, J., JACOBSON, G., Rumbaugh, I., Jacobson, I., & Booch, G. (2000). El lenguaje unificado de modelado: manual de referencia. Addison Wesley,.

Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., & Yan, H. (2006). Discovering architectures from running systems. Software Engineering, IEEE Transactions on, 32(7), 454-466.

SEI Software Engenieering Institute, (2016), What is your definition of software architecture?, recuperado: marzo 13 de 2016, disponible en http://www.sei.cmu.edu/architecture/start/glossary/definition-form.cfm.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., & Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. Software Engineering, IEEE Transactions on, 21(4), 314-335.

Soni, D., Nord, R. L., & Hofmeister, C. (1995, April). Software architecture in industrial applications. In Software Engineering, 1995. ICSE 1995. 17th International Conference on (pp. 196-196). IEEE.

Sourceware, GNU gprof, recuprada el 31 de enero de 2016, disponible en: https://sourceware.org/binutils/docs/gprof/

Stoermer, C., Bachmann, F., & Verhoef, C. (2003). SACAM: The software architecture comparison analysis method (No. CMU/SEI-2003-TR-006). CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

Stoermer, C., O'Brien, L., y Verhoef, C. (2003). Moving towards quality attribute driven software architecture reconstruction. In null (p. 46). IEEE.

Storey, M. A. D., Müller, H., & Wong, K. (1996). Manipulating and documenting software structures. Software Visualization, 7, 244-263.

Storey, M. A. D., y Müller, H. A. (1995, October). Manipulating and documenting software structures using SHriMP views. In Software Maintenance, 1995. Proceedings., International Conference on (pp. 275-284). IEEE.

Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2009). Software architecture: foundations, theory, and practice. Wiley Publishing.

The Open Group, ArchiMate 2.1 Specification, 2012, recuperado marzo 13 de 2016, disponible en: http://www.archimate.org/.

Tilley, S. (1998). A reverse-engineering environment framework (No. CMU/SEI-98-TR-005). Carnegie-Mellon Univ Pittsburgh pa software engineering inst.

Tilley, S.R., Smith, D.B. y S. Paul, Towards a Framework for Program Understanding, Proc. Int'l Workshop Program Comprehension, p. 19, 1996, doi:10.1109/WPC.1996.501117.

Tonella, P., Torchiano, M., Du Bois, B., & Systä, T. (2007). Empirical studies in reverse engineering: state of the art and future trends. Empirical Software Engineering, 12(5), 551-571.

Tracz, W., Lileanna: A Parameterized Programming Language. In Proceedings of the Second International Workshop on Software Reuse, pages 66-78, Lucca, Italy, March 1993.

Tran, J. y R. Holt, Forward and Reverse Repair of Software Architecture, Proc. Conf. Centre for Advanced Studies on Collaborative Research, 1999.

Turner C. R., Fuggetta A., Lavazza L., Wolf A. L., A conceptual basis for feature engineering. The Journal of Systems and Software, 49, Elsevier, 1999.

Tzerpos, V., & Holt, R. C. (2000, November). ACDC: An algorithm for comprehension-driven clustering. In wcre (p. 258). IEEE.

Van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., & Riva, C. (2004, June). Symphony: View-driven software architecture reconstruction. In Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on (pp. 122-132). IEEE.

Vanciu, R., & Abi-Antoun, M. (2013, November). Finding architectural flaws using constraints. In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on (pp. 334-344). IEEE.

Vasconcelos, A., y Werner, C. (2011). Evaluating reuse and program understanding in ArchMine architecture recovery approach. Information Sciences, 181(13), 2761-2786.

Woods, S.G.; Jercarrie`re, S. y R. Kazman, The Perils and Joys of Reconstructing Architectures, SEI Interactive, The Architect, vol. 2, no. 3, 1999.

Yan, H., Garlan, D., Schmerl, B., Aldrich, J., & Kazman, R. (2004, May). Discotect: A system for discovering architectures from running systems. InProceedings of the 26th International Conference on Software Engineering(pp. 470-479). IEEE Computer Society.

YourKit, YourKit .NET & Java Profiling, recuperada 31 de enero de 2016, disponible en: http://www.yourkit.com/