# ARQUITECTURA DE LÍNEA BASE PARA EL FRAMEWORK ONTOCONCEPT.


**CARLOS ALBERTO OCAMPO SEPÚLVEDA**

**Ingeniero de Sistemas**

**Director:**

**PhD JULIO CESAR CHAVARRO PORRAS**

**UNIVERSIDAD TECNOLÓGICA DE PEREIRA**

**FACULTAD DE INGENIERÍAS**

**MAESTRÍA EN INGENIERÍA DE SISTEMAS**


PEREIRA, noviembre 2017

## Tabla de contenido

Tabla de contenido ..... 2
GLOSARIO ..... 7
RESUMEN ..... 13
CAPITULO 1. GENERALIDADES ..... 14
PROBLEMA OBJETO DE ESTUDIO ..... 14
JUSTIFICACIÓN ..... 16
OBJETIVO GENERAL ..... 16
OBJETIVOS ESPECÍFICOS ..... 17
DELIMITACIÓN DEL ALCANCE ..... 17
CAPITULO 2. MARCO TEÓRICO: LAS ARQUITECTURAS DE SOFTWARE Y SOA ..... 18
2.1 DEFINICIÓN DE ARQUITECTURA Y ALGUNOS CONCEPTOS RELACIONADOS ..... 18
2.2 ATRIBUTOS DE CALIDAD ..... 19
2.2.1 ARQUITECTURA Y ATRIBUTOS DE CALIDAD ..... 20
2.3 ESTILOS ARQUITECTURALES ..... 24
2.3.1 ARQUITECTURAS CENTRADAS EN DATOS ..... 25
2.3.2 ARQUITECTURAS DE FLUJOS DE DATOS ..... 25
2.3.3 ARQUITECTURAS DE MÁQUINAS VIRTUALES ..... 26
2.3.4 ARQUITECTURAS DE LLAMADA Y RETORNO ..... 26
2.3.5 ARQUITECTURAS DE COMPONENTES INDEPENDIENTES ..... 27
2.3.6 ESTILOS HETEROGÉNEOS ..... 27
2.4 ORGANIZACIÓN DE ESTILOS ARQUITECTURALES ..... 28
2.5 LENGUAJES DE DESCRIPCIÓN ARQUITECTURAL ..... 32
2.5.1 Principales Características de los LDA ..... 33
2.5.2. Elementos Arquitectónicos que modelan los ADL ..... 33
2.5.3 Relación Lenguajes de Descripción Arquitectural ..... 35
2.5.4 UML como LDA ..... 40
CAPITULO 3. ONTOLOGÍAS Y FRAMEWORKS ONTOLOGICOS ..... 43
3.2 FRAMEWORK ONTOLÓGICO ..... 45
3.2.1. ONTOCONCEPT ..... 45

3.2.2 Protegé ........ 47
3.2.3 NeOn Toolkit ........ 47
CAPITULO 4. CARACTERÍSTICAS DE SOA Y ROA PARA WEB ........ 49
4.1 ARQUITECTURAS ORIENTADAS A SERVICIOS (SOA) ........ 49
4.2 PRINCIPIOS FUNDAMENTALES DE SOA ........ 52
4.3 CARACTERÍSTICAS DE SOA ........ 55
4.4 COMPONENTES DE SOA ........ 55
¿Por qué SOA?. ........ 56
4.5 ARQUITECTURAS ORIENTADAS A RECURSOS (ROA) ........ 57
4.5.1 PROPIEDADES DE ROA ........ 58
4.5.2 REST ........ 59
CAPITULO 5. RECOMENDACIÓN ARQUITECTURAL PARA FRAMEWORK ONTOLOGICO ........ 61
5.1 Principales Módulos de Ontoconcept ........ 62
5.2 Módulos relacionados con la edición Ontológica ........ 63
5.3 Propuesta Arquitectural ........ 64
5.3.1 UML 2.0 ........ 64
5.3.2 Arquitectura Recomendada usando UML como ADL ........ 65
5.3.3 Especificación de Componentes y Conectores ........ 67
5.4 Aspectos No funcionales ........ 70
5.4.1. ¿Por qué los atributos de calidad son usados para el análisis? ........ 70
5.4.2. ¿Entonces cómo especificar los atributos de calidad? ........ 72
5.5 ESCENARIOS POR ATRIBUTOS DE CALIDAD ........ 73
Objetivos de Calidad ........ 78
Esenciales ........ 78
Esperados ........ 79
Deseados ........ 79
5.6 ¿Cuáles son las salidas de una evaluación arquitectónica? ........ 79
5.6.1 Lista priorizada de los atributos de calidad requeridos ........ 79
5.6.2 Riesgos y no riesgos ........ 79
5.7. ¿Cuáles son los costos y beneficios de realizar una evaluación arquitectónica? ........ 80
5.7.1 Reúne a los stakeholders ........ 80
5.7.2 Fuerza una articulación en las metas específicas de calidad ........ 80

5.7.3 Fuerza una explicación clara de la arquitectura ..... 80
5.7.4 Mejora la calidad de la documentación de la arquitectura ..... 80
5.7.5 Descubre oportunidades de reuso ..... 80
5.7.6 Resultan mejoras en las arquitecturas ..... 81
5.8 Propuesta de aplicación No funcional para Ontoconcept. ..... 81
Plantilla de Lista de Escenarios (QAW) ..... 86
Anexo III: Plantilla para especificación de escenarios (QAW) ..... 88
El cuarto Método, análisis de intercambio de arquitectura ..... 97
Desafíos ..... 97
CAPITULO 6. COMPARATIVA DE FRAMEWORKS ..... 110
6.1 Algunos framework lógicos que existen en la actualidad son: ..... 110
CAPITULO 7. CONCLUSIONES, RECOMENDACIONES Y TRABAJO FUTURO ..... 112
7.1 CONCLUSIONES ..... 112
7.2 RECOMENDACIONES ..... 112
7.3 TRABAJOS FUTUROS ..... 112
8. BIBLIOGRAFÍA ..... 114

## LISTA DE TABLAS

Tabla 1. Atributos de Calidad del Sistema................**¡Error! Marcador no definido.**

Tabla 2. Fuente GARLAN, David y SHAW, Mary. An introduction to software architecture. CMU Software ........................................................3**¡Error! Marcador no definido.**

Tabla 3. Estrategias en Descripción de Arquitectura por UML ([adoptado de [12]) .... 38

Tabla 4. Operaciones de Recursos ........................................................................ 59

Tabla 5. Plantilla Especificación de Componente .................................................. 69

## LISTA DE ILUSTRACIONES

Ilustración 1. Interfaz de usuario de Ontoconcept - Fuente [1] ........ 46
Ilustración 2. Interfaz de Usuario NeOn Toolkit ........ 48
Ilustración 3. Arquitectura de marco de referencia para la gestión conceptual del cambio ontológico ........ 62
Ilustración 4. Módulos relacionados al proceso de edición ontológica ........ 64
Ilustración 5. Diagrama de Componentes - Vista Externa de la Arquitectura ........ 67
Ilustración 6. Vista Interna del Componente Cargador ........ 68
Ilustración 7. Partes para el manejo de escenarios de calidad ........ 72

## LISTA DE ANEXOS

Anexo 1. CASO DE ESTUDIO ESCENARIOS ARQ

Anexo 2. METODO ATAM PASOS

Anexo 3. Descripción UML de la arquitectura propuesta

## GLOSARIO

Se presenta las definiciones de algunos conceptos utilizados, algunos han sido extractados y traducidos de https://www.w3.org/2003/glossary/, también en https://www.sei.cmu.edu/architecture/start/glossary/index.cfm en caso contrario se indica la fuente bibliográfica.

**ADL**: (Architecture Description Language). Lenguaje de Descripción Arquitectural. Es un lenguaje descriptivo de modelado arquitectónico de software que se focaliza en la estructura de alto nivel de la aplicación antes que, en los detalles de implementación de sus módulos concretos, que puede usarse como un formato de intercambio común para herramientas de diseño de arquitectura y/o como base para desarrollar nuevas herramientas de diseño y análisis arquitectónico.

**Arquitectura de Software:** ANSI / IEEE Std 1471-2000, Práctica recomendada para la descripción arquitectónica de sistemas intensivos en software. La arquitectura se define por la práctica recomendada como la organización fundamental de un sistema, incorporada en sus componentes, sus relaciones entre sí y el medio ambiente, y los principios que rigen su diseño y evolución. Esta definición pretende abarcar una variedad de usos del término arquitectura mediante el reconocimiento de sus elementos comunes subyacentes. El principal de ellos es la necesidad de comprender y controlar los elementos del diseño del sistema que capturan la utilidad, el costo y el riesgo del sistema.

En algunos casos, estos elementos son los componentes físicos del sistema y sus relaciones. En otros casos, estos elementos no son físicos, sino componentes lógicos. En otros casos, estos elementos son principios o patrones perdurables que crean estructuras organizacionales duraderas. La definición pretende abarcar estos usos distintos, pero relacionados, al tiempo que fomenta una definición más rigurosa de lo que constituye la organización fundamental de un sistema dentro de dominios particulares.
Tomado de: https://www.sei.cmu.edu/architecture/start/glossary/moderndefs.cfm

**Arquitectura empresarial:** Un medio para describir las estructuras y los procesos comerciales que conectan las estructuras empresariales.

**Arquitectura de referencia:** Un modelo de referencia mapeado en elementos de software que implementa la funcionalidad definida en el modelo de referencia.

**Arquitectura del sistema:** Un medio para describir los elementos e interacciones de un sistema completo, incluidos sus elementos de hardware y sus elementos de software.

**Arquitectura de Integración Empresarial. Entreprise Architecture Integration** (EAI): Son las soluciones que integran las aplicaciones nuevas con las existentes, proporcionando las funcionalidades necesarias para dar soporte a los procesos del negocio.

**Arquitectura orientada a servicios (SOA):** Una forma de diseñar, desarrollar, implementar y administrar sistemas, en la que

- los servicios proporcionan funcionalidad empresarial reutilizable a través de interfaces bien definidas para promover la interoperabilidad
- los consumidores del servicio se crean utilizando la funcionalidad de los servicios disponibles
- la interfaz de servicio y la implementación están claramente separadas para promover la independencia de la plataforma
- una infraestructura SOA permite el descubrimiento, la composición y la invocación de servicios
- los protocolos son predominantemente, pero no exclusivamente, intercambios de documentos basados en mensajes

Los sistemas creados con este paradigma se denominan "sistemas orientados a servicios". Obtenga más información sobre SOA .

**Atributo de calidad:** Una propiedad de un producto de trabajo o bienes por el cual su calidad será juzgada por alguna parte interesada o partes interesadas. Los requisitos de atributos de calidad como los de rendimiento, seguridad, modificabilidad, confiabilidad y usabilidad tienen una influencia significativa en la arquitectura del software de un sistema.

**Business Process Management** (BPM): es la aplicación de técnicas y herramientas de software que se utilizan para analizar y optimizar los procesos de negocios en las empresas; y es también una condición para aumentar la competitividad y mejorar la habilidad de innovación. (Verificar referencia y definición)

**Bus de servicios empresariales** (EBS): Bus al que se conectan servicios. Esto puede hacerse a través de tecnología de servicios web, mediante tecnología clásica EAI (Entreprise Architecture Integration) o a través de conectores estándares Java. El ESB provee las herramientas para facilitar el funcionamiento de las arquitecturas tipo SOA, además provee mecanismos de integración como mensajería, adaptadores hacia repositorios, listener y medios de administración de servicios. (Verificar referencia y definición)

**Calidad de Software:** Es el grado con el que un sistema, componente o proceso cumple con los requisitos especificados y las necesidades o expectativas del cliente o usuario.

**Capa:** Una agrupación de módulos que, en conjunto, ofrecen un conjunto cohesivo de servicios a otras capas. Las capas están relacionadas entre sí por la relación estrictamente ordenada que se permite usar.

**Controladores arquitectónicos:** La combinación de requisitos funcionales, de calidad-atributo y empresariales que dan forma a la arquitectura o al elemento considerado.

**Diseño detallado:** Un término que las personas a veces usan para referirse a un diseño arquitectónico detallado, es decir, un diseño arquitectónico de elementos de grano fino o decisiones de diseño arquitectónico de grano fino, como la interfaz precisa de un elemento, y que en ocasiones significa diseño no arquitectónico.

**Diseño no arquitectónico:** Las decisiones de diseño se llevan a cabo después de la arquitectura y se dejan a los diseñadores e implementadores posteriores porque estas decisiones están suficientemente restringidas por la arquitectura o porque no son críticas para lograr los requisitos del sistema.

**Elemento**: El bloque de construcción arquitectónico (componente, conector o módulo) que es nativo de un estilo.

**Escenarios de Calidad:** Descripciones de una interacción del sistema con respecto a algún atributo de calidad. Típicamente, el enfoque se centraría en tres tipos de escenarios: caso de uso, usos anticipados del sistema; crecimiento, cambios anticipados al sistema; Estreses imprevistos y exploratorios para el sistema (usos y / o cambios).

**Estilo arquitectónico:** Una especialización de los tipos de elementos y relaciones, junto con un conjunto de restricciones sobre cómo se pueden usar.

**Interoperabilidad**: Interoperabilidad es la condición que permite que diferentes sistemas heterogéneos puedan intercambiar datos y procesos sin que se presenten problemas en la comunicación.

**Interfaz de software:** Un límite a través del cual dos entidades independientes se encuentran e interactúan o se comunican entre sí.

**Lenguaje de diseño y análisis de arquitectura (AADL):** Una norma internacional de la Sociedad de Ingenieros Automotrices (SAE) establecida en 2004 que permite el análisis de diseños de sistemas (y sistemas de sistemas) antes del desarrollo y admite un enfoque de desarrollo basado en modelos centrado en la arquitectura a lo largo del ciclo de vida del sistema. Conozca más sobre AADL.

**Lenguaje de descripción de arquitectura (ADL):** Un lenguaje (gráfico, textual o ambos) para describir un sistema de software en términos de sus elementos arquitectónicos y las relaciones entre ellos.

**Marco de arquitectura:** "Convenciones y prácticas comunes para la descripción de la arquitectura establecida dentro de un dominio específico o comunidad de partes interesadas" ( ISO / IEC 42010 ).

**Método de análisis de intercambio de arquitectura (ATAM):** Un método para evaluar arquitecturas de software en relación con los objetivos de atributos de calidad que revela qué tan bien una arquitectura satisface determinados objetivos de calidad y proporciona información sobre cómo esos objetivos de calidad interactúan entre sí. Aprenda más sobre ATAM.

**Método de diseño impulsado por atributos (ADD**): Un método sistemático para diseñar la arquitectura de software de un sistema intensivo en software al basar el proceso de diseño en los requisitos de atributos de calidad de la arquitectura, lo que permite a los diseñadores comprender las compensaciones de atributos de calidad al principio del proceso de diseño. Aprende más sobre ADD.

**Método de análisis de costo beneficio (CBAM):** Un método para analizar los costos, beneficios e implicaciones del cronograma de las decisiones sobre la arquitectura del software; asocia explícitamente los costos, los beneficios y la incertidumbre con las decisiones arquitectónicas como un medio para optimizar la elección de tales decisiones. Marco de referencia: Una abstracción en la cual un código común que proporciona una funcionalidad genérica puede ser anulado o especializado selectivamente por un código de usuario que proporciona una funcionalidad específica.

**Modelo de referencia:** Una división de funcionalidad en elementos junto con el flujo de datos entre esos elementos.

**Ontología:** "Una ontología es una especificación formal y explícita de una conceptualización compartida" [SBF98]. En esta definición se deben destacar los siguientes aspectos: conceptualización, en este contexto, es un modelo abstracto de cómo la gente piensa acerca de cosas en el mundo, usualmente restringidas a un área en particular; especificación explícita, hace referencia a que los conceptos y relaciones del modelo abstracto están dados por definiciones y términos explícitos; en su aspecto formal permiten que sean interpretados por personas ó máquinas, las cuales - a través de una pieza de software (agente)- se encuentran habilitadas para este fin.

**Patrón arquitectónico:** Una descripción de elementos y tipos de relaciones junto con un conjunto de restricciones sobre cómo se usan.

**Punto de sensibilidad:** Una propiedad de uno o más componentes (y / o relaciones de componentes) que es crítica para lograr una respuesta de atributo de calidad particular. Los puntos de sensibilidad son lugares en una arquitectura específica en los que una medida de respuesta específica es particularmente sensible (es decir, es probable que un pequeño cambio tenga un gran efecto). A diferencia de las tácticas, los puntos de sensibilidad son propiedades de un sistema específico.

**Relación:** Una definición de cómo los elementos cooperan para lograr el trabajo de un sistema.

**Restricción:** Un requisito para el cual las decisiones de diseño están pre especificadas. Subsistema: Una parte de un sistema que (1) lleva a cabo un subconjunto funcionalmente cohesivo de la misión general del sistema, (2) se puede ejecutar de forma independiente y (3) se puede desarrollar y desplegar de forma incremental.

**Resource Oriented Architecture:** (ROA). Arquitectura Orientada a Recursos. Es un acercamiento de arquitectura, que considera que los recursos son elementos de la web.

La parte clave, de todas formas, es que pueden ser descubiertos, y una vez que son descubiertos pueden representarse a sí mismos. No hay un requerimiento de conocimiento previo del recurso para establecer una conversación, al contario que las habilidades cognitivas de un servicio en la SOA.

**Servicios de Soporte**: Son servicios que no hacen parte del núcleo del negocio pero que proveen funcionalidad requerida para que todo el ambiente funcione correctamente.

**Servicio de Infraestructura**: Son servicios requeridos o brindados por el departamento de IT para soportar la operación de los servicios del negocio.

**Servicio web**: Un servicio web es un sistema de software diseñado para admitir la interacción interoperable de máquina a máquina a través de una red. Tiene una interfaz descrita en un formato procesable por máquina (específicamente WSDL). Otros sistemas interactúan con el servicio web de una manera prescripta por su descripción utilizando mensajes SOAP, normalmente transmitidos utilizando HTTP con una serialización XML junto con otros estándares relacionados con la web.

**Sistema:** Una colección de entidades (elementos, componentes, modelos, etc.) que están organizadas para un propósito común.

**Sistema de Información**: (SI). Conjunto de elementos físicos, lógicos, de comunicación, datos y personal que interrelacionados, permiten el almacenamiento, transmisión y proceso de la información.

**SOAP**: (Simple Object Access Protocol). El conjunto formal de convenciones que rigen las reglas de formato y procesamiento de un mensaje SOAP. Estas convenciones incluyen las interacciones entre nodos SOAP que generan y aceptan mensajes SOAP con el fin de intercambiar información a lo largo de una ruta de mensaje SOAP.

**Taller de mejora de la arquitectura (AIW):** Un método para evaluar y mejorar arquitecturas de software relativas a los objetivos de atributos de calidad. El SEI evalúa una arquitectura usando el ATAM para exponer los riesgos arquitectónicos que potencialmente inhiben el logro de los objetivos comerciales de una organización. Luego, el SEI trabaja con los arquitectos y los gerentes para determinar en qué objetivos comerciales enfocarse y proponer alternativas para mejorar y evolucionar la arquitectura, clasificando estas alternativas en función de sus costos, beneficios e incertidumbre esperados.

**Taller de atributos de calidad (QAW):** Un método para identificar los atributos críticos de la arquitectura de un sistema, como la disponibilidad, el rendimiento, la seguridad, la interoperabilidad y la capacidad de modificación, que se derivan de objetivos de misión o comerciales. El QAW no supone la existencia de una arquitectura de software; destinado a obtener, recopilar y organizar requisitos de atributos de calidad de software en forma de escenarios y utilizarse antes de que se haya creado la arquitectura del software. Aprenda más sobre QAWs .

**Vistas y más allá (Views and Beyond):** Un enfoque para la documentación que sostiene que documentar una arquitectura de software es una cuestión de documentar las vistas relevantes y luego agregar información que se aplica a más de una vista.

**UML:** (Unified Modeling Lenguaje). Es el lenguaje unificado de modelamiento, es el estándar para escribir los planos del software. Puede ser usado para visualizar, especificar, construir y documentar los artefactos de un sistema software.

**WSDL**: (Web Services Description Language). WSDL 2.0 describe un servicio web en dos etapas fundamentales: una abstracta y otra concreta. Dentro de cada etapa, la descripción utiliza una serie de construcciones para promover la reutilización de la descripción y para separar las preocupaciones de diseño independientes.

En un nivel abstracto, WSDL 2.0 describe un servicio web en términos de los mensajes que envía y recibe; los mensajes se describen independientemente de un formato de cable específico que utiliza un sistema de tipo, típicamente el Esquema XML.

Una operación asocia un patrón de intercambio de mensajes con uno o más mensajes. Un patrón de intercambio de mensajes identifica la secuencia y la cardinalidad de los mensajes enviados y / o recibidos, así como a quién se los envía y / o se los envía lógicamente. Una interfaz agrupa las operaciones sin ningún compromiso con el transporte o el formato de cable.

En un nivel concreto, un enlace especifica detalles de formato de transporte y cable para una o más interfaces. Un punto final asocia una dirección de red con un enlace. Y finalmente, un servicio agrupa los puntos finales que implementan una interfaz común.
**https://www.w3.org/TR/2007/REC-wsdl20-20070626/#intro_ws**

**XML**: (Extensible Markup Language). Extensible Markup Language (XML) es un formato de texto simple y muy flexible derivado de SGML (ISO 8879). Originalmente diseñado para enfrentar los desafíos de la publicación electrónica a gran escala, XML también está desempeñando un papel cada vez más importante en el intercambio de una amplia variedad de datos en la Web y en otros lugares. https://www.w3.org/XML/

**RESUMEN**

La arquitectura de software define la vista abstracta de los componentes y sus interrelaciones, así como sus propiedades visibles en forma externa. Este trabajo se centra en la aplicación de los criterios a la construcción de la herramienta tecnológica Ontoconcept, que gestiona el ciclo de vida de de una ontología o una de las fases que lo componen.

Ontoconcept es un framework ontológico construido desde la perspectiva del modelado conceptual, que facilita la abstracción del lenguaje en que se implementa la ontología. Como herramienta tecnológica, facilita el trabajo de construcción y cambio de la ontología en las diferentes etapas: concepción, edición, cambio, razonamiento, etc. Adicionalmente, brinda soporte al trabajo colaborativo, desarrollado por el ingeniero ontológico y a los expertos que modelan un dominio particular, haciendo uso de ambientes abiertos y distribuidos como Internet.

Este trabajo presenta la línea base arquitectural del framework Ontoconcept. Para lograrlo, presenta una revisión de la literatura en el campo de las arquitecturas de software. se apoya en las características de las arquitecturas orientadas a servicios para dar describir el comportamiento modular del framework, y describe el uso de los métodos para instanciar los atributos no funcionales en la arquitectura de Ontoconcept. Es decir, presenta la guía para el uso de QAW, ADD, VaB y ATAM, en la instanciación arquitectural del framework.

Abstract
The software architecture defines the abstract view of the components and their interrelationships, as well as their externally visible properties. This work focuses on the application of criteria to the construction of the Ontoconcept technological tool, which manages the life cycle of an ontology or one of the phases that comprise it.

Ontoconcept is an ontological framework built from the perspective of conceptual modeling, which facilitates the abstraction of the language in which ontology is implemented. As a technological tool, it facilitates the work of construction and change of the ontology in the different stages: conception, edition, change, reasoning, etc. Additionally, it provides support for collaborative work, developed by the ontological engineer and the experts who model a particular domain, making use of open and distributed environments such as the Internet.

This work presents the architectural baseline of the Ontoconcept framework. To achieve this, it presents a review of the literature in the field of software architectures. it relies on the characteristics of service-oriented architectures to describe the modular behavior of the framework, and describes the use of methods to instantiate non-functional attributes in the Ontoconcept architecture. That is, it presents the guide for the use of QAW, ADD, VaB and ATAM, in the architectural instantiation of the framework.

# CAPITULO 1. GENERALIDADES

Este capítulo presenta y delimita el objeto de estudio en el campo de aplicación de las arquitecturas y tecnologías para la construcción de aplicaciones web en el desarrollo del nuevo framework de trabajo ontológico Ontoconcept.

## PROBLEMA OBJETO DE ESTUDIO

Según [1], bajo el título "MARCO DE REFERENCIA PARA LA GESTIÓN DEL CAMBIO EN ONTOLOGÍAS, BASADOS EN MODELOS CONCEPTUALES", fue desarrollado el framework Ontoconcept utilizando una arquitectura cliente-servidor dos capas soportada en plugins.

El Framework Ontoconcept, en su primera versión, se desarrolló con el ánimo de probar la viabilidad del modelo conceptual planteado en la tesis doctoral, pero carece de funcionalidades necesarias para el desarrollo de un trabajo profesional de ingeniería ontológica, (serán expresadas en capítulo 6). En esta dirección, el grupo de investigación GIA de la Universidad Tecnológica de Pereira (UTP) ha asumido el reto de desarrollar la nueva versión del framework Ontoconcept soportándola en las nuevas tecnologías existentes y definiendo sus características arquitecturales con base en los atributos de calidad esperados para una herramienta de trabajo profesional en el campo de la ingeniería ontológica.

El uso creciente de Internet por parte de todos los individuos en forma transversal en todas las actividades de su accionar cotidiano, ha ayudado a crear una tendencia para que las diferentes tecnologías de la información permitan dar soporte a productos en internet y estos a su vez, utilizan la red para prestar servicios de desarrollo a nuevas herramientas que también la usan. Quizá sea esta una razón por la cual las aplicaciones empresariales se interesan en consumir y explotar las características y los beneficios entregados por Internet, donde se pueden integrar aplicaciones web, aplicaciones móviles y aplicaciones de escritorio. Esta tendencia ha generado una gama amplia de tecnologías y propuestas arquitecturales que se encuentran maduras y que facilitan el desarrollo de aplicaciones que explotan las capacidades abiertas y distribuidas de Internet.

La nueva versión del framework Ontoconcept no es ajena a esta realidad, (uso de internet) por el contrario, la naturaleza de una herramienta de trabajo para la gestión del cambio ontológico en particular, y del ciclo de vida de las ontologías en general, tiene restricciones de ingeniería que son solucionables en ambientes abiertos y distribuidos como es Internet; Tal es el caso de la edición remota y el trabajo colaborativo sobre una ontología.

El trabajo de construcción de una ontología es desarrollado por un equipo de personas conformado por varios roles:

- El ingeniero ontológico, persona encargada de guiar el proceso de construcción de la ontología desde el punto de vista metodológico, debe entenderse que la

ingeniería ontológica es una disciplina del área de la representación del conocimiento.
- Equipo de expertos en el dominio que se modela y quienes están encargados de las definiciones precisas del área de conocimiento que es modelado.
- Curador o grupo de curadores, que son las personas que definen si un cambio en el dominio, la especificación o en la perspectiva del uso de la ontología es procedente aplicar.
- Usuarios de la ontología. Personas o máquinas que han de hacer las consultas pertinentes a la ontología.

De lo anterior se puede inferir fácilmente, que el trabajo desarrollado a lo largo del ciclo de vida de una ontología, y en las diferentes etapas que ésta tiene, es de naturaleza participativa, colaborativa y cuando mínimo simplemente de uso abierto. De aquí la necesidad de su uso en ambientes abiertos y distribuidos como Internet.

La necesidad de plantear una solución de diseño basada en Arquitectura Orientadas a Servicios para instanciar el Framework Ontológico Ontoconcept como Servicio Web, definiendo las estructuras y las características que se deben hacer visibles dentro de los elementos que componen Ontoconcept, así como las restricciones funcionales y no funcionales que deben cumplir dichas estructuras, puede ser englobada en la siguiente pregunta:

¿Es posible definir las características que debe cumplir Ontoconcept con base en los lineamientos de una Arquitectura Orientada a Servicios?
La solución de este interrogante implica abordar las siguientes preguntas inherentes:

- ¿Cómo aplicar los principios rectores de SOA a la definición arquitectural de Ontoconcept?

- ¿Cuáles son los elementos que se deben tener en cuenta dentro de una arquitectura?

- ¿Cuáles son los Principios que rigen SOA?

- ¿Cómo se aplica SOA en la definición de una Arquitectura Empresarial?

- ¿Qué criterios debe contemplar Ontoconcept en sus requerimientos no funcionales y de acoplamiento modular?

- ¿Existen módulos que requieran utilizar un estilo arquitectural diferente a la línea base del framework?

# JUSTIFICACIÓN

En la medida que las arquitecturas de software se han convertido en un área de estudio e investigación, nuevos elementos y conceptos cada vez más complejos han sido incorporados, v.gr, patrones arquitecturales, patrones de diseño, estilos arquitecturales, Lenguajes de Descripción Arquitectural (LDA).

En este mismo sentido, podemos encontrar la *Arquitectura Orientada a Servicios (SOA)*, la cual gana mayor importancia debido al uso extendido de los Servicios Web, siendo esta la forma más frecuente de implementar SOA[1]. Dada la tendencia en el desarrollo de software, a utilizar las tecnologías que extienden a Internet, en particular los servicios web y las aplicaciones móviles, se deriva la necesidad de profundizar en las arquitecturas orientadas a servicios y los Restfull Web Services.

En este contexto, se genera la necesidad de abordar en forma disciplinar el estudio de la arquitectura de una aplicación compleja como lo es un framework ontológico.

Ontoconcept es una herramienta para hacer gestión del ciclo de vida ontológico y por tanto permite gestionar el cambio de Ontologías; se encuentra desarrollado en una aplicación de escritorio y requiere ser actualizado, para satisfacer necesidades de una herramienta de trabajo de ingeniería ontológica. Esto quiere decir, que debe ofrecer funcionalidades para garantizar la gestión de repositorios de ontologías, construcción de nuevas ontologías y dar soporte a sus procesos de cambio.

La cantidad y variedad de módulos que componen un framework ontológico y en particular a Ontoconcept, son caracterizados por su alta modularidad y bajo acoplamiento, estos interactúan entre sí y requieren gestionar actividades que incluyen: administrar archivos, repositorios ontológicos o bases de datos, soportar todas las características de edición, visualización, pruebas y razonamiento basado en diferentes tipos de lógicas, trabajo en ambientes colaborativos, distribuidos y abiertos. En este sentido, la línea base arquitectural debe satisfacer la pregunta ¿Cómo será la forma en que los módulos interactúan entre ellos?

Con base en lo anterior, para la actualización requerida por el framework Ontoconcept se propone definir las características arquitecturales que debe soportar, validando los elementos y componentes de una arquitectura orientada a servicios, y determinando los estilos particulares a sus diferentes componentes modulares.

## OBJETIVOS

### OBJETIVO GENERAL

- Proponer la arquitectura de línea base para el Framework Ontoconcept, soportado en los criterios definidos en las arquitecturas web orientadas a servicios.

[1] https://www.juvo.be/blog/oracle-leader-all-3-gartner-magic-quadrants-soa-and-soa-governance

OBJETIVOS ESPECÍFICOS

- Realizar una revisión de las arquitecturas web determinando SOA Vs ROA.
- Determinar los elementos o componentes que constituyen una arquitectura Web con base en el modelo conceptual en comparación con el modelo lógico.
- Determinar los mecanismos de instanciación arquitectural de los requerimientos no funcionales que debe satisfacer Ontoconcept.

## DELIMITACIÓN DEL ALCANCE

El desarrollo de este trabajo se concentra en plantear las características arquitecturales que deben cumplir el framework Ontoconcept, así como los mecanismos de interacción modular y las funcionalidades generales.

Proponer una arquitectura se ocupa de qué hacer, mientras que en la implementación se ocupa de cómo, por lo tanto, este trabajo no incluye la implementación de la arquitectura para el framework ontológico.

El framework Ontoconcept está orientado a satisfacer las operaciones que intervienen en el ciclo de vida de una ontología. En este trabajo, se realizará únicamente el análisis relacionado con los módulos que intervienen en forma directa con el proceso de la arquitectura del framework Ontoconcept, desde su proceso de creación, siguiendo la trazabilidad del cambio.

Una lista de los módulos que serán analizados es la siguiente:
1. Editor
2. Cargador
3. Exportador
4. Gestor de persistencia
5. Razonador
6. Visualizador
7. Gestor de configuración
8. Gestor de log transaccional
9. Gestión de la Ontología de cambio genérico.

Esta lista es bastante aproximada a la vista de alto nivel de los módulos que interactúan directamente durante un proceso de edición ontológica. Puede pensarse, por esta razón, que se equipara la edición ontológica y el proceso de gestión del cambio.

# CAPITULO 2. MARCO TEÓRICO: LAS ARQUITECTURAS DE SOFTWARE Y SOA

Este capítulo presenta una revisión de los principales conceptos inherentes al tema de las arquitecturas de software, profundizando en las arquitecturas utilizadas para el desarrollo de aplicaciones orientadas a internet, en particular SOA, ROA y REST (Representational State Transfer - Transferencia de Estado Representacional).

## 2.1 DEFINICIÓN DE ARQUITECTURA Y ALGUNOS CONCEPTOS RELACIONADOS

En el campo de las arquitecturas de software se han formalizado múltiples conceptos que facilitan establecer puntos de referencia conceptual para poder construir referentes teóricos. Los siguientes conceptos han sido seleccionados como parte del trabajo de construcción del marco de referencia conceptual del proyecto. En primera instancia las definiciones de Arquitectura de Software, que se han seleccionado corresponden a Paul Clements de la Universidad de Carnegie Mellon, la cual ha sido cuna de los principales movimientos teóricos en cuanto al área de arquitecturas de software en particular y de la ingeniería de software en general, y por otro lado la referencia de la IEEE, que ha realizado un importante esfuerzo de estandarización conceptual en el área, de igual forma la referencia a otra de igual importancia para la comunidad de arquitectura e Ingeniería de Software.

1. La Arquitectura de Software es a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de esos componentes según se la percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema. [2]

2. La Arquitectura de Software es la organización fundamental de un sistema encarnada en sus componentes, las relaciones entre ellos y el ambiente, y los principios que orientan su diseño y evolución. [de la edición original, IEEE 1471 2000]

3. Se puede decir que, la arquitectura de software es la habilidad y la comprensión que permite la creación de sistemas exitosos. Los sistemas creados reflejan esa habilidad y entendimiento, o arquitectura.[2]

Estas definiciones de manera común, señalan el papel de la arquitectura en la definición a alto nivel de la interacción de los diferentes componentes, sus interfaces y posible comportamiento. De donde puede inferirse su importancia, no solo en el comportamiento de los componentes del sistema, sino de la capacidad para representar los requerimientos no funcionales del mismo.

Otras definiciones igualmente importantes en el contexto de este trabajo tienen que ver con la definición de lo que son las arquitecturas de referencia, la línea base arquitectural y las arquitecturas soportadas en modelos. Las definiciones presentadas a continuación,

[2] Carl Zipperle https://www.sei.cmu.edu/architecture/start/glossary/community.cfm

son aportadas por el profesor de la Universidad de Antioquia Juan Bernardo Quintero, en el curso de Arquitecturas de Software [3].

Arquitectura de referencia. Proporciona una plantilla de solución probada para la arquitectura en un dominio particular, y define un vocabulario común con el que se puede discutir los detalles de implementación para un producto de software. Se ocupa de los problemas comúnmente encontrados en los proyectos de software como la escalabilidad, la fiabilidad y la seguridad.

En la propuesta de [41], MDSD (Model Driven Software Development) se precisan las arquitecturas de referencia a partir del concepto de arquitectura de dominio que se define como la agregación de tres conceptos:

- Plataforma,
- Lenguaje Especifico de Dominio
- Transformaciones.

La arquitectura generativa, es la especialización de una arquitectura de domino dentro del contexto de AC- MDSD (Desarrollo de Software Dirigido por Modelos Centrado en la Arquitectura).

La Línea Base de la arquitectura, es el conjunto de los productos que retratan el estado actual de la empresa, sus prácticas comerciales, y su infraestructura técnica, comúnmente se refiere a la descripción de los procesos, es decir -Tal como está (As is)- de la arquitectura. Extrapolando este concepto, se hace referencia al estado actual organizacional que brinda el soporte a los mecanismos de trabajo y su instanciación en los diferentes componentes modulares y de infraestructura a ser utilizada.

Finalmente, al conjunto de los productos que retratan el futuro o estado final de la empresa, en general captura el pensamiento estratégico de la organización y sus planes, comúnmente se refiere al deber ser "to be" de la arquitectura.

Estas definiciones permiten precisar lo que se pretende en este trabajo al definir la línea base arquitectural del framework Ontoconcept.

## 2.2 ATRIBUTOS DE CALIDAD

Los atributos de calidad son las características esperadas de un componente o de un sistema. En [4] los autores definen y especifican un conjunto de atributos de calidad esperados en la definición de una arquitectura de software.

Como punto de partida cabe mencionar que los atributos de calidad tratan sobre la funcionalidad de los sistemas, los cuales son la "línea básica" de las capacidades, servicios y comportamiento del sistema. Además, las funcionalidades y otras cualidades están estrechamente relacionadas. Los sistemas son constantemente rediseñados debido a la dificultad de mantenerlos, portarlos, porque son lentos o han sido hackeados.

Es el mapeo de una funcionalidad del sistema sobre la estructura del software lo que determina el soporte de la arquitectura para las cualidades. La arquitectura es el primer escenario en la creación de un sistema en el cual los requerimientos de calidad podrían ser direccionados.

Existen múltiples propuestas de catalogación para los atributos de calidad, en este trabajo se determinó utilizar la propuesta por el SEI (Instituto de Ingeniería del Software de la Universidad de Carnegie & Mellon), sin desconocer los propuestos por otras empresas e institutos tales Microsoft, Oracle, IEEE entre otros. La decisión se fundamentó en que esta clasificación es consistente con la perspectiva de arquitectura de software que ha sido utilizada para guiar este proyecto.

### 2.2.1 ARQUITECTURA Y ATRIBUTOS DE CALIDAD

En [4] los autores plantean dos categorías de atributos de calidad contra los cuales un sistema se puede medir y particularmente a nivel arquitectural:

> Observable a través de la ejecución: ¿En qué medida el sistema satisface sus requerimientos de comportamiento, durante la ejecución? ¿Proporciona los resultados requeridos? ¿los resultados son proporcionados de una manera suficientemente oportuna? ¿los resultados son correctos o están dentro de la tolerancia de precisión y estabilidad especificadas? ¿la función del sistema es la deseada cuando se conecta a otros sistemas?
>
> No Observable a través de la ejecución: ¿Que tan fácil es para el sistema integrar, probar y modificar? ¿Qué tan costoso fue desarrollarlo? ¿Cuál fue su tiempo de comercialización?
>
> Requerimientos No funcionales: Los requerimientos no funcionales son un establecimiento terminológico dentro del cual son agrupados todas las otras cualidades del sistema, como el rendimiento o la modificabilidad. Las cualidades de software tienen que ser diseñadas desde el comienzo. No se puede “obtener” la funcionalidad correctamente y luego devolverse a poner en ella todas las cualidades.
>
> Cualidades de tiempo de ejecución: Seguridad, Rendimiento, tolerancia a fallos
> Los requerimientos no funcionales implican algo que no existe, un tipo de requerimiento que puede ser especificado independientemente del comportamiento del sistema. Viendo las cualidades de esta forma, se tiende a omitirlas hasta el final de la tarea de especificación, dejándolas añadidas al final y pobremente especificadas para iniciar.
>
> La calidad tiene que ser considerada en todas las fases de diseño, implementación y despliegue, pero las distintas cualidades se manifiestan de manera diferente durante estas fases; Por ejemplo, muchos aspectos de calidad de usabilidad no son

arquitecturales, debido a que casi siempre están encapsulados dentro de un solo componente. Su localización afecta la modificabilidad y no la usabilidad.

Por otra parte, la modificabilidad es en gran medida arquitectural. La modificabilidad está determinada por como la funcionalidad está dividida. Un sistema es modificable si los cambios no incluyen un gran número de distintos componentes. Es la división de la responsabilidad lo que determina la modificabilidad de un sistema.

El rendimiento es un ejemplo de una calidad la cual tiene dependencias arquitecturales y no arquitecturales. El rendimiento depende parcialmente de la cantidad necesaria de comunicación entre los componentes, parcialmente en que funcionalidad ha sido "asignado" cada componente, parcialmente en la elección de algoritmos para implementar la funcionalidad seleccionada, y parcialmente en cómo estos algoritmos son codificados.

1. La arquitectura es fundamental para la realización de muchas de las cualidades de interés en un sistema, estas cualidades deben ser diseñadas y evaluadas en el nivel arquitectural.

2. Algunas cualidades no son arquitecturalmente sensibles y tratar de alcanzar estas cualidades o analizarlas arquitecturalmente no es beneficioso.

Es importante tener en cuenta que los atributos de calidad existen dentro de sistemas complejos, y que su satisfacción nunca puede ser alcanzada de manera aislada. El logro de cualquier atributo de calidad tendrá un efecto, a veces positivo o a veces negativo en el logro de otros atributos de calidad.

La principal técnica para lograr un software portable es aislar las dependencias del sistema (sistemas dependientes). Este aislamiento introduce sobrecarga en la ejecución del sistema, típicamente en forma de procesos o procedimientos límites, las cuales perjudican el rendimiento.

Clases de atributos de calidad

A1. Primero, cualidades que se pueden discernir mediante la observación de la ejecución del sistema (rendimiento, seguridad, disponibilidad, funcionalidad, usabilidad) y cualidades del sistema que no son discernibles en tiempo de ejecución (modificabilidad, portabilidad, reusabilidad, integrabilidad, capacidad de pruebas).

A2. Las cualidades atribuibles directamente al sistema, hay cualidades de negocio (como el tiempo de comercialización) que se ven afectadas por la arquitectura.

A3. Finalmente, hay cualidades sobre la propia arquitectura que son importantes

| **A1.1 ATRIBUTOS DE CALIDAD DEL SISTEMA DISCERNIBLES EN EJECUCIÓN** | |
|---|---|
| **Rendimiento** | Respuesta del sistema, tiempo requerido para responder a un estímulo (evento), o número de eventos procesados en un intervalo de tiempo. |

| | |
|---|---|
| | El rendimiento es a menudo una función de la cantidad de comunicación y la interacción que hay entre los componentes del sistema.<br>El rendimiento ha sido el factor determinante en la arquitectura del sistema, y frecuentemente ha comprometido el logro de todas las demás cualidades |
| **Seguridad** | Medida de la capacidad del sistema para resistir intentos no autorizados y denegación de servicio mientras sigue suministrando sus serviciso a usuarios autorizados |
| **Disponibilidad** | Mide la proporción de tiempo en la que el sistema está en funcionamiento. Se mide por la longitud de tiempo entre los fallos, así como por la rapidez con la que el sistema es capaz de reanudar el funcionamiento en caso de fallo. La disponibilidad de estado constante de un sistema es la proporción de tiempo que el sistema está funcionando correctamente. |
| **Confiabilidad** | (estrechamente relacionado con la disponibilidad) Capacidad del sistema de mantenerse operando en el tiempo. La fiabilidad se mide generalmente por el tiempo medio hasta el fallo. |
| **Funcionalidad** | Capacidad del sistema para hacer el trabajo para el cual fue diseñado. Realizar una tarea requiere que muchos o la mayoría de los componentes del sistema trabajen coordinadamente para completar el trabajo. Si a los componentes no se les ha asignado la responsabilidad correcta o no han sido dotados con las capacidades correctas para coordinarse con otros componentes, el sistema será incapaz de responder a la funcionalidad requerida.<br>La funcionalidad es ortogonal a la arquitectura, lo que significa que en gran parte no es arquitectural. Si el logro de la funcionalidad fuese el único requisito, el sistema podría existir como un solo componente monolítico con ninguna estructura interna. |
| **Usabilidad** | La usabilidad puede estar divido en las siguientes áreas:<br>Facilidad de Aprendizaje: ¿Qué tan rápido y fácil es para el usuario aprender a usar la interfaz del sistema?<br>Eficiencia: ¿El sistema responde con la velocidad adecuada a las solicitudes de un usuario?<br>Capacidad de Recordar (Memorability - Memorizable): ¿El usuario puede recordar cómo hacer las operaciones del sistema entre los usos del sistema?<br>Evitación de Errores: ¿El sistema anticipa y previene errores comunes de los usuarios?<br>Manejador de Errores: ¿El sistema ayuda al usuario a recuperarse de los errores?<br>Satisfacción: ¿El sistema hace fácil el trabajo para los usuarios? |
| **A1.2 ATRIBUTOS DE CALIDAD DEL SISTEMA NO DISCERNIBLES EN TIEMPO DE EJECUCIÓN** | |
| **Modificabilidad** | En todas sus formas, puede ser el atributo de calidad más estrechamente alineado con la arquitectura del sistema. La capacidad de hacer cambios rápidamente y el costo provienen directamente de la arquitectura. La modificabilidad es en gran medida una función de la localización de cualquier cambio. Hacer un cambio generalizado al sistema es más costoso que hacer un cambio a un solo componente. |

| | |
|---|---|
| | Ya que la arquitectura define los componentes y las responsabilidades de cada uno, sino que también define las circunstancias en las que cada componente tendrá que cambiar. Una arquitectura clasifica eficazmente todos los posibles cambios dentro de tres categorías de acuerdo a si el cambio va a provocar una modificación de un solo componente, de más de un componente, o de algo más drástico, tal como un cambio en el estilo arquitectural.<br>Extendiendo o cambiando capacidades (Extensibilidad): Adición de nuevas funcionalidades, mejorando la funcionalidad existente, o la reparación de errores.<br>Eliminación de las capacidades indeseadas: racionalizar o simplificar la funcionalidad de una aplicación existente.<br>Adaptación a nuevos ambientes operativos (portabilidad): La portabilidad hace un producto más flexible en la forma en que puede alinearse, apelando a una amplia base de clientes<br>Reestructuración: racionalización de los servicios del sistema, modularización, optimización, o la creación de componentes reutilizables que pueden servir para dar a la organización una ventaja sobre sistemas futuros<br>La modificabilidad algunas veces es llamada mantenibilidad |
| **Portabilidad** | Es la capacidad del sistema de ejecutarse en diferentes ambientes informáticos. Estos ambientes pueden ser hardware, software o una combinación de los dos. |
| **Reusabilidad** | La reutilización es usualmente usada para referirse al diseño de un sistema, para que la estructura del sistema o algunos de sus componentes puedan ser reutilizados de nuevo en futuras aplicaciones.<br>El diseño para la reutilización significa que el sistema ha sido estructurado de manera que sus componentes pueden ser elegidos de productos construidos previamente, en este caso es un sinónimo de integrabilidad, en otros casos la reusabilidad puede ser concebida como un caso especial de modificabilidad.<br>La reusabilidad está relacionada a la arquitectura de software en que los componentes arquitecturales son las unidades de reutilización, y como un componente reusable depende de que tan fuertemente acoplado este con otros componentes. |
| **Integrabilidad** | Es la capacidad de hacer que los componentes desarrollados separadamente de un sistema trabajen juntos directamente. Esto a su vez depende de la complejidad externa de los componentes, sus mecanismos de interacción y protocolos, y el grado en que las responsabilidades han sido fraccionadas limpiamente.<br>La integrabilidad también depende de que tan bien y que tan completamente se han especificado las interfaces de los componentes.<br>Integrar un componente depende no solo de los mecanismos de interacción usados, sino también de la funcionalidad asignada al componente para ser integrado y como esta funcionalidad está relacionada a la funcionalidad del entorno del nuevo componente.<br>Un tipo especial de la integrabilidad es la interoperabilidad. La integrabilidad mide la capacidad de las partes de un sistema para trabajar juntos. La interoperabilidad mide la capacidad de un grupo de partes para trabajr con otro sistema. |

| **Capacidad de Prueba** | La capacidad de prueba de software se refiere a la facilidad con la que el software puede hacer para demostrar sus defectos a través de pruebas. Para que un sistema sea properly testable tiene que ser posible controlar cada estado interno y las entradas de los componentes y luego observar sus salidas.<br>La capacidad de prueba de un sistema se refiere a varias cuestiones estructurales o arquitectónicas: su nivel de documentación arquitectónica, su separación de intereses, y el grado en que el sistema utiliza la información escondida. El desarrollo incremental también beneficia la capacidad de prueba en la misma manera que mejora la integrabilidad |
|---|---|

**Tabla 1. Atributos de Calidad del Sistema**

A2. CUALIDADES DE NEGOCIO AFECTADAS POR LA ARQUITECTURA:

- Tiempo de Mercado
- Costo
- Vida útil proyectada del sistema
- Mercado Objetivo
- Rollout Schedule
- Uso extensivo de sistemas heredados

A3. ATRIBUTOS PROPIOS DE LA ARQUITECTURA

- Integridad Conceptual: es el tema fundamental o la visión que unifica el diseño del sistema en todos los niveles. La arquitectura debe hacer cosas similares de forma similar. Fred Brooks escribe enfáticamente que la integridad conceptual de un sistema es la consideración más importante en el diseño de un sistema, y que los sistemas sin éste fallan. Es mejor tener un sistema que omita ciertas características y mejoras, pero que refleje un solo conjunto de ideas de diseño, a tener un sistema que tenga muy buenas ideas, pero que son independientes y descoordinadas. Para Brooks la integridad conceptual es central para la calidad de un producto.

- Exactitud e Integridad (correctness and completeness): son esenciales para que la arquitectura permita la reunión de todos los requerimientos del sistema y las restricciones de recursos en tiempo de ejecución.
- Edificabilidad: Es la calidad de la arquitectura que permite al sistema ser completado por el equipo disponible en un tiempo prudencial ser abierto a ciertos cambios mientras el desarrollo del sistema avanza.

## 2.3 ESTILOS ARQUITECTURALES

Un estilo arquitectural consiste de algunas características claves y reglas para combinar esas características tales que se preserve la integridad arquitectural.
Una arquitectura de software está determinada por:

- Un conjunto de tipos de componentes (repositorio de datos, procesos, procedimientos) que realizan alguna función en tiempo de ejecución.
- Un Diseño topológico de estos componentes indicando sus interrelaciones en tiempo de ejecución
- Un conjunto de restricciones semánticas (ejemplo un repositorio de datos no se le permite cambiar sus valores guardados en el)
- Un conjunto de conectores (por ejemplo, llamadas a subrutinas, llamada a procedimientos remotos, flujos de datos y sockets) que median la comunicación, la coordinación, o cooperación entre los componentes.

Un estilo define una clase de arquitectura; equivalentemente es una abstracción para un conjunto de arquitecturas que lo reúnan. Los estilos usualmente son ambiguos acerca del número de componentes implicados. Los estilos pueden ser ambiguos sobre los mecanismos sobre los cuales interactúan los componentes, aunque algunos componentes se unen de forma explícita. Los estilos siempre son ambiguos acerca de la función del sistema.

### 2.3.1 ARQUITECTURAS CENTRADAS EN DATOS

Las arquitecturas centradas en datos tienen el objetivo de lograr la calidad de integrabilidad de datos. El termino arquitecturas centrada en datos se refiere a sistemas en los cuales el acceso y la actualización de un almacén de datos ampliamente accedida es una descripción apta.

Los estilos centrados en datos se están volviendo cada vez más importantes debido a que ofrecen una solución estructural para la integrabilidad. Muchos sistemas especialmente los construidos a partir de componentes existentes están logrando la integración de datos a través del uso de mecanismos de Pizarra. Tienen la ventaja de que los clientes son relativamente independientes entre sí, y el almacén de datos es independiente de los clientes. Así, este estilo es escalable: nuevos clientes pueden ser fácilmente añadidos.

### 2.3.2 ARQUITECTURAS DE FLUJOS DE DATOS

Las arquitecturas de flujos de datos tienen el objetivo de alcanzar las cualidades de la reutilización y la modificabilidad. El estilo de flujo de datos se caracteriza por ver el sistema como una serie de transformaciones en piezas sucesivas de entradas de datos. Este estilo tiene 2 subtipos, Lote Secuencial y tuberías y filtros.

En el estilo por lote secuencial, los pasos de procesamiento, o componentes son programas y la suposición es que cada paso se ejecuta hasta finalizar antes de que comience el siguiente paso.

El estilo tubería y filtro enfatiza la transformación incremental de los datos por componentes sucesivos. Los filtros transforman datos incrementalmente, usan poca información contextual y no retienen información de estado entre las instancias. Las tuberías son sin estado y simplemente existen para mover datos entre los filtros.

Sus ventajas principalmente provienen desde su simplicidad y las formas limitadas en las que pueden interactuar con su ambiente. No hay interacciones de componentes complejos de manejar. Simplifica el mantenimiento y mejora la reutilización por la misma razón de los filtros, además se puede tratarlos como cajas negras. Debido a que un filtro puede procesar su entrada en aislamiento del sistema, un sistema de tuberías y filtros se hace fácilmente paralelo o distribuido, proporcionando oportunidades para mejorar un rendimiento de los sistemas sin modificarlo.

La forma en la que se descompone un problema en este estilo es una de las desventajas del mismo. Las aplicaciones interactivas son difíciles de crear en este estilo. Además, el pedido al filtro puede ser difícil.

### 2.3.3 ARQUITECTURAS DE MÁQUINAS VIRTUALES

Las arquitecturas de máquinas virtuales tienen como objetivo alcanzar la cualidad de portabilidad. Las máquinas virtuales son un estilo de software que simula una funcionalidad que no es nativa en el hardware y/o software en el cual está implementado. Puede ser útil de muchas formas: puede permitir simular (y probar) plataformas aún no se han construido (como nuevo hardware), y puede simular modos “desastre” (como es común en simuladores de vuelos o sistemas críticos de seguridad), que serían demasiado complejo, costosos o peligrosos para probar con un sistema real.

### 2.3.4 ARQUITECTURAS DE LLAMADA Y RETORNO

Las arquitecturas de llamada y retorno tienen el objetivo de alcanzar las cualidades de modificabilidad y escalabilidad. Las arquitecturas de llamadas y retorno han tenido el estilo arquitectural predominante en grandes sistemas de software durante los últimos 30 años. Sin embargo, dentro de este estilo hay un número de subestilos, cada uno tiene características interesantes:

- Arquitecturas de programa principal y subrutinas, el objetivo es descomponer un programa en piezas más pequeñas para ayudar a alcanzar la modificabilidad.
- Sistemas de Llamada a procedimientos remotos son sistemas de programa principal y subrutinas que se descomponen en partes que viven en computadores conectados a través de una red. Su objetivo es incrementar el rendimiento mediante la distribución de cómputos y teniendo las ventajas de múltiples procesadores.
- Sistemas de tipos de datos abstractos u orientados a objetos, son la versión moderna de las arquitecturas de llamada y retorno. El objetivo es alcanzar la cualidad de la modificabilidad. El paradigma orientado a objetos como el paradigma tipo de dato abstracto del cual fue desarrollado, enfatiza en la agrupación de los datos y el conocimiento de cómo manipular y acceder a esos datos. Las principales

características que diferencian el paradigma orientado a objetos del paradigma de tipos de datos abstractos son la herencia y el polimorfismo

- Sistemas de Capas: son aquellos en los que los componentes están asignados a capas para controlar la interacción entre componentes. Cada nivel se comunica solo son su vecino inmediato. El objetivo es alcanzar las cualidades de modificabilidad y portabilidad usualmente.

### 2.3.5 ARQUITECTURAS DE COMPONENTES INDEPENDIENTES

Las arquitecturas de componentes independientes consisten de un número de procesos u objetos que se comunican a través de mensajes. Todas estas arquitecturas tienen el objetivo de lograr la modificabilidad desacoplando diversas partes de los cálculos. Envían datos entre sí, pero por lo general no se controlan entre sí. Los mensajes pueden ser trasmitidos a los participantes nombrados o, en el caso de eventos del sistema usando el paradigma publicación/suscripción, para poder pasar entre los participantes no identificados.

Los sistemas de eventos son un subestilo en el cual el control es parte del modelo. Los componentes individuales anuncian los datos que quieren compartir con su ambiente con un conjunto de componentes sin nombre. Los sistemas de eventos hacen uso de un administrador de mensajes que gestionan la comunicación entre componentes, invocando un componente un componente cuando llega un mensaje para él.

Este paradigma es importante porque desacopla las implementaciones de componentes de conocer los nombres y las locaciones de los otros componentes. Los componentes pueden ejecutarse en paralelo, interactuando solamente a través de intercambio de datos cuando lo requieran. Este desacoplamiento facilita la integración de componentes. Los componentes pueden ejecutarse en paralelo, interactuando solamente a través de intercambio de datos cuando lo requieran. Este desacoplamiento facilita la integración de componentes.

Además de los sistemas de eventos, está el estilo de los procesos de comunicación. Un subtipo conocido de este estilo es Cliente-Servidor. El objetivo es alcanzar la calidad de escalabilidad. Existe un servidor para servir datos a uno o más clientes, los cuales generalmente se encuentran en una red. El cliente origina un llamado al servidor, el cual trabaja síncrona o asíncronamente, para dar servicio a la petición del cliente.

### 2.3.6 ESTILOS HETEROGÉNEOS

Los sistemas raramente son construidos a partir de un solo estilo, y se dice que estos tipos de sistemas son heterogéneos. Hay tres tipos de heterogeneidad:

- Localmente Heterogéneos: significa que un esquema (dibujo) de sus estructuras en tiempo de ejecución revelará patrones de diferentes estilos en diferentes áreas.
- Jerárquicamente Heterogéneos, significa que un componente de un estilo, cuando se descompone, está estructurado de acuerdo a las reglas de los diferentes estilos.

- Simultáneamente heterogéneos, significa que cualesquiera de varios estilos pueden bien ser apto para la descripción del sistema.

Esta última forma de heterogeneidad reconoce que los estilos no dividen las arquitecturas de software dentro. El estilo centrado en datos se compone de clientes de hilo independiente, al igual que una arquitectura de componentes independientes. Las capas en un sistema de capas en comprender objetos o componentes independientes o incluso subrutinas en un programa o subrutina principal de un sistema. Los componentes de un sistema de tubería y filtro son usualmente implementados como procesos que operan independientemente, esperando hasta una entrada en su puerto; de nuevo, esto es similar a los sistemas de componentes independientes cuya orden de ejecución esta predeterminada.

## 2.4 ORGANIZACIÓN DE ESTILOS ARQUITECTURALES

Debido a que una arquitectura casi nunca está construida completamente de un solo estilo, una arquitectura necesita entender las interrelaciones entre estilos.

En general, la comprensión de las ramificaciones de una arquitectura que es el resultado de la combinación de estilos requiere un proceso de análisis arquitectural. Se puede analizar el conjunto de estilos introducidos para ver la forma en que se describen, lo que tienen en común y como pueden ser extendidos para admitir otras posibilidades de diseño.

### Categorías de Características

Los principales ejes de clasificación son el control y la interacción de los datos entre los componentes. Se puede hacer discriminaciones más sutiles dentro de estas dimensiones:

- Cómo el control es compartido, asignado y trasferido entre componentes
- Cómo los datos se comunican a través del sistema
- Cómo los datos y el control interactúan
- El tipo de razonamiento que los estilos permiten
- Los tipos de componentes y conectores que se utilizan en el estilo

Partes Constituyentes: Componentes y Conectores, los tipos permitidos de componentes y conectores con discriminadores importantes entre estilos. Los componentes y conectores son los principales elementos básicos de las arquitecturas. Un componente es una unidad de software que realiza alguna función en tiempo de ejecución (Ej.: programas, objetos, procesos y filtros). Un conector es un mecanismo que media la comunicación, coordinación o cooperación entre los componentes. La implementación de un conector puede ser distribuida sobre muchos componentes (Ej. Representaciones compartidas, llamadas a procedimientos remotos, protocolos de paso de mensajes y flujos de datos).

**Problemas de Control:**

Los problemas de control describen como el control pasa entre los componentes y como los componentes trabajan juntos temporalmente.

- Topología, ¿Qué forma geométrica toma el sistema para el control de flujo?
  - Estilo tubería: topología lineal o de control acíclico
  - Estilo de programa principal y subrutinas cuenta con una topología jerárquica (forma de árbol)
  - Algunos sistemas de servidor tienen una topología de estrella
  - Un estilo que consiste en la comunicación de procesos secuenciales puede tener una tecnología arbitraria.

    La topología puede ser estática o dinámica, esto está determinado por el tiempo de unión de los componentes.
- Sincronicidad, ¿Qué tan dependientes son las acciones de los componentes sobre los estados de control de los demás? En un sistema unísono, el estado de cualquier componente implica el estado de los otros. En sistemas síncronos, los componentes se sincronizan regularmente, pero los estados de las relaciones son impredecibles. Los componentes asíncronos son en gran medida impredecibles en sus interacciones o se sincronizan de vez en cuando; los componentes oportunistas como los agentes autónomos trabajan en paralelo completamente independientes el uno del otro
- Tiempo de unión ¿Cuándo es establecida la identidad de un socio en una operación de transferencia de control? Algunas trasferencias de control están predeterminadas en el programa. Otras son vinculadas dinámicamente mientras el sistema está ejecutándose.

**Problemas de Datos**

Los problemas de datos describen como los datos se mueven en el sistema:

- Topología, la topología de los datos describe la forma geometría del flujo de datos del sistema. Las alternativas son las mismas que la topología de control.
- Continuidad, ¿Qué tan continuo es el flujo de datos a través del sistema? Un sistema de flujo continuo tiene nuevos datos disponibles en todo momento. Un sistema de flujo esporádico tiene nuevos datos generados en momentos discretos. La trasferencia de datos puede ser de alto volumen o debajo volumen.
- Modo, el modo de los datos describe como el dato se pone a disposición a través del sistema. En un estilo de objeto, el dato es pasado de componente a componente, mientras que en cualquier de los sistemas de datos compartidos, el dato es compartido poniéndose disponible en un lugar accesible para todos los participantes.
- Tiempo de Unión, ¿Cuándo es establecida la identidad de un socio en una operación de transferencia de control? Esta es la analogía de datos de los mismos problemas de control.

**Problemas de Interacción de Control/Datos**
Los datos de interacción describen la relación de ciertos problemas de control y datos.

- Forma, ¿El control de flujo y el flujo de datos son topologías sustancialmente isomorfas?

- ¿Direccionalidad, ¿Si las formas son sustancialmente las mismas, el control de flujo en la misma dirección que los datos o la dirección opuesta? En un sistema de flujo de datos como la tubería y los filtros, el control y los datos pasan juntos de componente a componente. Sin embargo, en un estilo cliente servidor, el control tiende a fluir en el servidor y el flujo de datos en los clientes.

**Tipo de Razonamiento:** Diferentes clases de arquitecturas se prestan para diferentes tipos de análisis. Un sistema de componentes opera asíncronamente en rendimiento paralelo a una gran diferencia de enfoques de razonamiento que un sistema que se ejecuta como una secuencia fija de pasos atómicos. Los diferentes estilos arquitecturales son buenas combinaciones para diferentes técnicas de análisis.

**REFINAMIENTO DE ESTILOS**

El refinamiento de estos estilos no es solo posible sino necesario antes de que puedan conducir una arquitectura.

**REFINANDO EL ESTILO DE FLUJO DE DATOS**

La red de flujo de datos describe sistemas cuyos componentes operan en grande, disponiendo continuamente de flujos de datos. Los componentes están organizados en topologías arbitrarias que trasmiten los datos con conectores no transformadores. El uso de una topología arbitraria hace más difícil la modificación de un sistema debido a que la interacción entre los componentes no está limitada. Por lo tanto, es natural imponer restricciones a las topologías y analizar los resultados.

Una definición más general para estilo arquitectónico es la que se ofrece en Taylor, Medvidovic y Oreizy (2009), tomada originalmente de Taylor, Medvidovic y Dashofy (2009) : "(…) colecciones de decisiones de diseño que 1) son aplicables en un contexto de desarrollo dado, 2) restringen las decisiones de diseño arquitectónicas específicas para un sistema dentro de dicho contexto, y 3) persigue cualidades que benefician los sistemas resultantes".

Como se aprecia en la definición anterior, se puede describir el estilo arquitectónico (EA) como una colección de decisiones de diseño (DS). Sin embargo, no todas las DS caracterizan al EA, sino que son las DS principales las que lo hacen (Dashofy

2007). Estas DS son las que afectan horizontalmente al sistema, por lo tanto implican a la mayoría de los elementos que conforman el mismo (Dashofy 2007)[3].

Al hablar de EA (y de AS en general) se considera una buena práctica referirse en un principio a los clásicos. Es por esta razón que se presenta un resumen de dos de los EA más referenciados actualmente como son: Tuberías y Filtros.

Abowd, Allen, y Garlan (1993) analizaron lo que ellos llaman el estilo tubería y filtro por la manera de formalización de la semántica de estilos arquitecturales. Identificaron tres variaciones del estilo tubería y filtro:

1. Sistemas sin bucles de retroalimentación o ciclos (acíclicos)
2. Tuberías (lineal)
3. Sistemas con solo componentes de ventilador de salida

Los estilos por lo general proveen cuatro argumentos (Monroe, Kompanek, Melton, Garlan 1997):[4]

1. Un vocabulario de elementos de diseño: tipos de componentes y conectores, como pueden ser filtros, tuberías, clientes, servidores, etc.
2. Reglas de diseño o restricciones que determinan las composiciones permitidas de esos elementos.
3. Interpretación semántica, por lo que las composiciones de elementos de diseño, debidamente limitados por las reglas de diseño, tienen significados bien definidos.
4. Análisis que pueden ser realizados en los sistemas construidos con el estilo.

## REFINANDO EL ESTILO PROCESOS DE COMUNICACIÓN

Los procesos de comunicación son usados para alcanzar el objetivo de modificabilidad y escalabilidad, pero las limitaciones de rendimiento y configuración afecta como estos objetivos son alcanzados. Andrews identifica ocho variantes que ocurren en la práctica y satisface diferentes objetivos.

Una Vía de flujo de datos a través de redes y filtros, esto es una versión del subestilo red de flujo de datos implementado con procesos de comunicación. En esta versión la implementación con mensajes introduce en el flujo de datos la abstracción. Una pieza de datos entra al sistema y realiza su recorrido a través de series de transformaciones, cada transformación acompañada por un proceso separado. La serie no necesita ser lineal.

Solicitudes y Respuestas entre clientes y servidores, Cliente y Servidor es un estilo popular. Puede ser visto como una especialización del estilo de procesos de

[3] GECONTEC: Revista Internacional de Gestión del Conocimiento y la Tecnología. ISSN 2255-5684 Matos-Arias, Y. y Silega-Martínez, N. Vol. 1(1). 2013 pag. 27

[4] Matos Arias, Y., & Silega Martínez, N. (2013). Estilo arquitectónico para el sistema integrado de gestión Cedrux.

comunicación, en el cual las topologías, la sincronicidad y el modo son restringidos de la forma general. Esta es la forma ingenua, que ignora el requisito usual para mantener el estado de una secuencia continua de las interacciones entre el cliente y el servidor.

Atrás y Adelante Interacción entre Procesos Vecinos, un algoritmo latido de corazón hace que cada nodo en el grafico envíe información y luego reúne la nueva información.

Sondas y Ecos en Gráficos
Difusión entre procesos en gráficos completos
Paso de Token a lo largo de bordes en una grafica
Coordinación entre procesos de servidor descentralizado
Trabajadores replicados compartiendo una bolsa de tareas

**USANDO ESTILOS EN EL DISEÑO DE SISTEMAS**

¿Cuál estilo se debe elegir para diseñar un sistema, entonces, si hay más de uno cual elegir? La respuesta es que depende de las cualidades que más nos interesen.

Un estilo puede servir como la descripción principal de un sistema en un área donde el discurso y el pensamiento son lo más importante para cumplir con la incertidumbre. Otros estilos pueden aplicar bien y ser fructíferos. Comenzar con la estructura de la arquitectura que proporciona la mayor influencia en las cualidades (incluyendo funcionalidad) que se espera ser más problemático. A partir de ahí, considere un estilo apropiado para que la estructura aborde las cualidades. En ese punto, otras estructuras y estilos pueden entrar para ayudar a abordar problemas secundarios.

*Deducción: Los estilos pueden ser descritos por un conjunto de características tales como la naturaleza de los componentes y conectores, sus topologías estáticas y el control dinámico y los datos que pasan los patrones, y el tipo de razonamiento que admiten.*

## 2.5 LENGUAJES DE DESCRIPCIÓN ARQUITECTURAL

La formalización de la arquitectura de software ha sido “formulada” por lenguajes formales para la especificación o descripción de la arquitectura, comúnmente llamados Lenguajes de Descripción de Arquitectura (ADL). [9]. A Continuación, se presentan algunas definiciones dadas durante su estudio:

Un ADL (Lenguaje de Descripción de Arquitectura) es un lenguaje (Gráfico o Textual o ambos) para describir un sistema de software en términos de sus elementos arquitecturales y las relaciones entre ellos. [10]

Un ADL es un lenguaje que provee elementos para modelar la arquitectura conceptual de un sistema de software, distinguiéndola de la implementación del sistema. [11]

Los ADLs permiten modelar una arquitectura mucho antes que se lleve a cabo la programación de las aplicaciones que la componen, analizar su adecuación, determinar sus puntos críticos y eventualmente simular su comportamiento. [Reinoso]

### 2.5.1 Principales Características de los LDA

Los lenguajes de descripción de arquitecturas han jugado un papel importante desde la fundación de la Arquitectura de software. Se ha denominado ADL, todo tipo de lenguajes, desde sistemas puramente formales. Hasta lenguajes de especificación algebraica, de programación concurrente o de definición de interfaces [7]. Además, los ADL deben cumplir con determinadas propiedades para que sean considerados como tal. [8]

1. Composición: El lenguaje debe ser tal, que el arquitecto pueda dividir un sistema complejo en partes más pequeñas de manera jerárquica, o construir un sistema a partir de los elementos que lo constituyen.
2. Abstracción: La arquitectura expresa una abstracción que permite identificar a los distintos elementos en una estructura de alto nivel, así como su papel en la misma. Esta abstracción es específica, y se diferencia de otras utilizadas en el desarrollo.
3. Reutilización: Se debe poder reutilizar componentes, conectores, estilos y arquitecturas, incluso en un contexto diferente a aquél en el que fueron definidos.
4. Configuración: El lenguaje debe separar con claridad la descripción de elementos individuales y de la de las estructuras en que participen. Esta característica es fundamental.
5. Heterogeneidad: El ADL ha de ser independiente del lenguaje en que se implemente cada uno de los componentes que manipula; se deben de poder combinar patrones arquitectónicos diferentes en un único sistema complejo.
6. Análisis: Se debe permitir diversas formas de análisis de la arquitectura y de los sistemas desarrollados a partir de ella, igualmente poder analizar la estructura, de modo que se puedan determinar sus propiedades con independencia de una implementación concreta, así como verificarlos después de cualquier modificación. Esto puede vincularse al uso de métodos formales que permitan la definición de arquitecturas sin ambigüedad semántica.

### 2.5.2. Elementos Arquitectónicos que modelan los ADL

Según [8] los elementos arquitectónicos que modelan los ADL se clasifican en:
Estructurales

**Componentes:** Representan los elementos computacionales primarios de un sistema. Intuitivamente, corresponden a las cajas de las descripciones de caja-y-línea de las arquitecturas de software. Ejemplos típicos serían clientes, servidores, filtros, objetos, pizarras y bases de datos. En la mayoría de los ADLs los componentes pueden exponer varias interfaces, las cuales definen puntos de interacción entre un componente y su entorno.

**Tipo de Componentes**: Modulo, Computación (cálculo - computation), Dato-compartido, Archivo-de-secuencia, filtros, procesos, schedProcess, general

**Conectores** (interacción del componente - protocolo de comunicación): Representan interacciones entre componentes. Corresponden a las líneas de las descripciones de caja-y-línea. Ejemplos típicos podrían ser tuberías (pipes), llamadas a procedimientos, broadcast de eventos, protocolos cliente-servidor, o conexiones entre una aplicación y un servidor de base de datos. Los conectores también tienen una especie de interfaz que define los roles entre los componentes participantes en la interacción.

**Tipo de conectores**: tubería, acceso de datos, archivos IO (fileIO), RPC (llamada a procedimiento remoto, llamada a procedimiento, RTSCheduler)

**Puerto:** No debe confundirse con el conector. Los puertos son los puntos por los que un componente puede realizar cualquier tipo de interacción; dicho de otro modo, es cada uno de los fragmentos en los que se segmenta el interfaz de un componente.

El puerto hace referencia a un punto de entrada o de salida del componente. Para aquellos autores que ven los componentes como procesos, el puerto sería el canal de mensajes. Ha de tenerse en cuenta, sin embargo, que los puertos de un componente no sólo expresan los servicios que éste oferta, sino también los requisitos que precisa; esto es, aquellas condiciones que necesita que cumpla el entorno para funcionar correctamente.

**Configuraciones o sistemas:** Se constituyen como grafos de componentes y conectores. En los ADLs más avanzados la topología del sistema se define independientemente de los componentes y conectores que lo conforman. Los sistemas también pueden ser jerárquicos: componentes y conectores pueden subsumir la representación de lo que en realidad son complejos subsistemas.

**Propiedades:** Representan información semántica sobre un sistema más allá de su estructura. Distintos ADLs ponen énfasis en diferentes clases de propiedades, pero todos tienen alguna forma de definir propiedades no funcionales, o pueden admitir herramientas complementarias para analizarlas y determinar, por ejemplo, el throughput y la latencia probable, o cuestiones de seguridad, escalabilidad, dependencia de bibliotecas o servicios específicos, configuraciones mínimas de hardware y tolerancia a fallas.

**Restricciones:** Representan condiciones de diseño que deben acatarse incluso en el caso que el sistema evolucione en el tiempo. Restricciones típicas serían restricciones en los valores posibles de propiedades o en las configuraciones topológicas admisibles. Por ejemplo, el número de clientes que se puede conectar simultáneamente a un servicio.

**Estilos**: Representan familias de sistemas, un vocabulario de tipos de elementos de diseño y de reglas para componerlos. Ejemplos clásicos serían las arquitecturas de flujo de datos basados en grafos de tuberías (pipes) y filtros, las arquitecturas de pizarras basadas en un espacio de datos compartido, o los sistemas en capas. Algunos estilos prescriben un framework, un estándar de integración de componentes, patrones arquitectónicos o como se lo quiera llamar

**Propiedades No Funcionales:** La especificación de estas propiedades es necesaria para simular la conducta de runtime, analizar la conducta de los componentes, imponer restricciones, mapear implementaciones sobre procesadores determinados, etcétera.

**Evolución:** Los ADLs deberían soportar procesos de evolución permitiendo derivar subtipos a partir de los componentes e implementando refinamiento de sus rasgos. Sólo unos pocos lo hacen efectivamente, dependiendo para ello de lenguajes que ya no son los de diseño arquitectónico sino los de programación.

Comportamiento

Debe resolverse los siguientes interrogantes:
¿Cómo se comportan los componentes y conectores?
¿Cómo se comportan integrados? ¿Cómo se integran?

### 2.5.3 Relación Lenguajes de Descripción Arquitectural

| ADL | FECHA | ORGANISMO INVESTIGADOR | OBSERVACIONES |
|---|---|---|---|
| Acme | 1995 | Monroe & Garlan (CMU), Wile (USC) | Lenguaje de intercambio de ADLs |
| Aesop | 1994 | Garlan (CMU) | ADL de propósito general, énfasis en estilos |
| ArTek | 1994 | Terry, Hayes-Roth, Erman (Teknowledge, DSSA) | Lenguaje específico de dominio - No es ADL |
| Armani | 1998 | Monroe (CMU) | ADL asociado a Acme |
| C2 SADL | 1996 | Taylor/Medvidovic (UCI) | ADL específico de estilo |
| CHAM | 1990 | Berry / Boudol | Lenguaje de especificación |
| Darwin | 1991 | Magee, Dulay, Eisenbach, Kramer | ADL con énfasis en dinámica |
| Jacal | 1997 | Kicillof , Yankelevich (Universidad de Buenos Aires) | Adl - Notación de alto nivel para descripción y prototipado |
| LILEANNA | 1993 | Tracz (Loral Federal) | Lenguaje de conexión de módulos |
| MetaH | 1993 | Binns, Englehart (Honeywell) | ADL específico de dominio |
| Rapide | 1990 | Luckham (Stanford) | ADL & simulación |

| ADL | FECHA | ORGANISMO INVESTIGADOR | OBSERVACIONES |
|---|---|---|---|
| SADL | 1995 | Moriconi, Riemenschneider (SRI) | ADL con énfasis en mapeo de refinamiento |
| UML | 1995 | Rumbaugh, Jacobson, Booch (Rational) | Lenguaje genérico de modelado – No es ADL |
| UniCon | 1995 | Shaw (CMU) | ADL de propósito general, énfasis en conectores y estilos |
| Wright | 1994 | Garlan (CMU) | ADL de propósito general, énfasis en comunicación |
| xADL | 2000 | Medvidovic, Taylor (UCI, UCLA) | ADL basado en XML |

**Tabla 2. Fuente GARLAN, David y SHAW, Mary. An introduction to software architecture. CMU Software**

**Acme – Armi**
Acme se define como una herramienta capaz de soportar el mapeo de especificaciones arquitectónicas entre diferentes ADLs, o en otras palabras, como un lenguaje de intercambio de arquitectura. No es entonces un ADL en sentido estricto, aunque la literatura de referencia acostumbra tratarlo como tal. De hecho, posee numerosas prestaciones que también son propias de los ADLs. Su objetivo principal es la motivación fundamental de Acme es el intercambio entre arquitecturas e integración de ADLs.

**Aesop**
El nombre oficial es Aesop Software Architecture Design Environment Generator. Se ha desarrollado como parte del proyecto ABLE de la Universidad Carnegie Mellon, cuyo objetivo es la exploración de las bases formales de la arquitectura de software, el desarrollo del concepto de estilo arquitectónico y la producción de herramientas útiles a la arquitectura, de las cuales Aesop es precisamente la más relevante.

La definición también oficial de Aesop es “una herramienta para construir ambientes de diseño de software basada en principios de arquitectura”. El ambiente de desarrollo de Aesop System se basa en el estilo de tubería y filtros propio de UNIX.

**ArTek**

ArTek fue desarrollado por Teknowledge. Se le conoce también como ARDEC/Teknowledge Architecture Description Language. En opinión de Medvidovic no es un genuino ADL, por cuanto la configuración es modelada implícitamente mediante información de interconexión que se distribuye entre la definición de los componentes individuales y los conectores. En este sentido, aunque pueda no ser un ADL en sentido estricto, se le reconoce la capacidad de modelar ciertos aspectos de una arquitectura. De todas maneras, es reconocidamente un lenguaje específico

de dominio y siempre fue presentado como un caso testigo de generación de un modelo a partir de una instancia particular de uso.

**C2 SADL**

C2 SADL (Simulation Architecture Description Language) es un ADL que permite describir arquitecturas en estilo C2. C2 o Chiron-2 no es estrictamente un ADL sino un estilo de arquitectura de software que se ha impuesto como estándar en el modelado de sistemas que requieren intensivamente pasaje de mensajes y que suelen poseer una interfaz gráfica dominante.

**CHAM**

CHAM (Chemical Abstract Machine) una técnica de especificación basada en álgebra de procesos que utiliza como fundamento teórico los sistemas de rescritura de términos para capturar la conducta comunicativa de los componentes arquitectónicos.

CHAM no es estrictamente un ADL, aunque algunos autores, en particular Inverardi y Wolf aplicaron CHAM para describir la arquitectura de un compilador. Se argumenta, en efecto, que CHAM proporciona una base útil para la descripción de una arquitectura debido a su capacidad de componer especificaciones para las partes y describir explícitamente las reglas de composición.

**Darwin**

Darwin es un lenguaje de descripción arquitectónica desarrollado por Jeff Magee y Jeff Kramer. Darwin describe un tipo de componente mediante una interfaz consistente en una colección de servicios que son ya sea provistos (declarados por ese componente) o requeridos (es decir, que se espera ocurran en el entorno). Las configuraciones se desarrollan instanciando las declaraciones de componentes y estableciendo vínculos entre ambas clases de servicios.

Darwin soporta la descripción de arquitecturas que se reconfiguran dinámicamente a través de dos construcciones: instanciación tardía [lazy] y construcciones dinámicas explícitas. Utilizando instanciación laxa, se describe una configuración y se instancian componentes sólo en la medida en que los servicios que ellos provean sean utilizados por otros componentes. La estructura dinámica explícita, en cambio, se realiza mediante constructos de configuración imperativos. De este modo, la declaración de configuración deviene un programa que se ejecuta en tiempo de ejecución, antes que una declaración estática de la estructura.

**Jacal**

JACAL es un lenguaje gráfico multipropósito para la descripción de arquitecturas de software. Reúne ventajas de otros lenguajes existentes y tiene la virtud de permitir la ejecución (animación) de las arquitecturas descritas. Mediante este

procedimiento se puede comprobar o refutar propiedades deseables de los diseños y recopilar métricas dinámicas.

El diseño de Jacal contempla otras características deseables en un ADL, como por ejemplo contar con una representación gráfica que permita a simple vista transmitir la arquitectura del sistema, sin necesidad de recurrir a información adicional. Para este fin, se cuenta con un conjunto predefinido (extensible) de conectores, cada uno con una representación distinta.

**LILEANNA**

LILEANNA es, visto como ADL, estructural y sintácticamente distinto a todos los demás. De hecho, es oficialmente un lenguaje de interconexión de módulos (MIL), basado en expresiones de módulo propias de la programación parametrizada. Un MIL se puede utilizar descriptivamente, para especificar y analizar un diseño determinado, o constructivamente, para generar un nuevo sistema con base en módulos preexistentes, ejecutando el diseño.

Al igual que en el caso de ArTek, en opinión de Medvidovic LILEANNA no es un genuino ADL, por cuanto la configuración se modela implícitamente mediante información de interconexión que se distribuye entre la definición de los componentes individuales y los conectores. En este sentido, aunque pueda no ser un ADL en sentido estricto, se le reconoce la capacidad de modelar ciertos aspectos de una arquitectura.

**MetaH**

Así como LILEANNA es un ADL ligado a desarrollos que guardan relación específica con helicópteros, MetaH modela arquitecturas en los dominios de guía, navegación y control (GN&C) y en el diseño aeronáutico. Aunque en su origen estuvo ligado estrechamente a un dominio, los requerimientos imperantes obligaron a implementar recursos susceptibles de extrapolarse productivamente a la tecnología de ADLs de propósito general.

MetaH ha sido diseñado para garantizar la puesta en marcha, la confiabilidad y la seguridad de los sistemas modelados, y también considera la disponibilidad y las propiedades de los recursos de hardware.

**Rapide**

Se puede caracterizar como un lenguaje de descripción de sistemas de propósito general que permite modelar interfaces de componentes y su conducta observable. Sería tanto un ADL como un lenguaje de simulación. La estructura de Rapide es sumamente compleja, y en realidad articula cinco lenguajes:

- el lenguaje de tipos describe las interfaces de los componentes;
- el lenguaje de arquitectura describe el flujo de eventos entre componentes;

- el lenguaje de especificación describe restricciones abstractas para la conducta de los componentes;
- el lenguaje ejecutable describe módulos ejecutables; y
- el lenguaje de patrones describe patrones de los eventos.

Los diversos sub-lenguajes comparten la misma visibilidad, scoping y reglas de denominación, así como un único modelo de ejecución. Su objetivo principal es la Simulación y determinación de la conformidad de una arquitectura.

**UML**

El Lenguaje de Modelado Unificado (UML) es un lenguaje de modelado popular y ampliamente extendido. Fue desarrollado por James Rumbaugh, Grady Booch, and Ivar Jacobson (conocidos como los tres amigos). Técnicamente UML no es un ADL y no estaba destinado a serlo. Sin embargo, UML es bastante adecuado para modelar sistemas. Su principal ventaja es que es un lenguaje ampliamente extendido, y del cual muchos diseñadores tienen conocimiento. La versión de UML 2.0 fue publicada en 2004. El lanzamiento incluyó el Lenguaje de Restricciones de Objeto (OCL), que es un lenguaje declarativo dirigido a describir modelos en texto plano. Este lanzamiento, sin embargo, ha sido criticado por ser grande y complejo de entender con un esfuerzo razonable. [Architecture Description Languages, Stefan Bjornander]

**UniCon**

UniCon (Universal Connector Support) es un ADL desarrollado por Mary Shaw y otros. Proporciona una herramienta de diseño para construir configuraciones ejecutables basadas en tipos de componentes, implementaciones y “conexiones expertas” que soportan tipos particulares de conectores. UniCon se asemeja a Darwin en la medida en que proporciona herramientas para desarrollar configuraciones ejecutables de caja negra y posee un número fijo de tipos de interacción, pero el modelo de conectores de ambos ADLs es distinto.

Oficialmente se define como un ADL cuyo foco apunta a soportar la variedad de partes y estilos que se encuentra en la vida real y en la construcción de sistemas a partir de sus descripciones arquitectónicas. UniCon es el ADL propio del proyecto Vitruvius, cuyo objetivo es elucidar un nivel de abstracción de modo tal que se pueda capturar, organizar y tornar disponible la experiencia colectiva exitosa de los arquitectos de software.
El propósito de UniCon es generar código ejecutable a partir de una descripción, a partir de componentes primitivos adecuados. UniCon se destaca por su capacidad de manejo de métodos de análisis de tiempo real a través de RMA (Rate onotonic Analysis).

**Wright**

Se puede caracterizar brevemente como una herramienta de formalización de conexiones arquitectónicas. Ha sido desarrollado por la Escuela de Ciencias Informáticas de la Universidad Carnegie Mellon, como parte del proyecto mayor ABLE. Este proyecto a su vez se articula en dos iniciativas: la producción de una herramienta de diseño, que ha sido Aesop, y una especificación formal de descripción de arquitecturas, que es propiamente Wright.

Wright es probablemente la herramienta más acorde con criterios académicos de métodos formales. Su objetivo declarado es la integración de una metodología formal con una descripción arquitectónica y la aplicación de procesos formales tales como álgebras de proceso y refinamiento de procesos a una verificación automatizada de las propiedades de las arquitecturas de software.

**xADL**

Técnicamente xADL es lo que se llama una aplicación de una especificación más abstracta y genérica, xArch, que es un lenguaje basado en XML, elaborado en Irvine y Carnegie Mellon para la descripción de arquitecturas. Cada tipo de conector, componente e interfaz de xADL incluye un placeholder de implementación que vendría a ser análogo a una clase virtual o abstracta de un lenguaje orientado a objetos. Éste es reemplazado por las variables correspondientes del modelo de programación y plataforma en cada implementación concreta. Esto permite vincular descripciones arquitectónicas y modelos directamente con cualquier binario, scripting o entidad en cualquier plataforma y en cualquier lenguaje.

### 2.5.4 UML como LDA

En [9] El Lenguaje Unificado de Modelado (UML) es un lenguaje grafico formal considerado un estándar industrial "de facto". Aunque el lenguaje ha sido creado como un lenguaje grafico en primer lugar para apoyar el análisis y diseño de software orientado a objetos, el lenguaje ha sido revisado varias veces, hoy es en general un lenguaje formal capaz de describir un sistema de software. El UML ha definido bien la sintaxis formal y semántica y puede ser comprobado y procesado.

**Características Arquitecturales de UML 2.0**

UML 2.0 enriquecido ayuda a modelar problemas arquitecturales de los sistemas de software. Entre las características más importantes se encuentra interfaces enriquecidas, puertos, componentes y conectores refinados.

Las interfaces pueden ser totalmente definidas en una declaración de protocolo y variables de estado de la interfaz.

El Puerto es un nuevo concepto de UML. Es muy cercano a la interface pero cada puerto es un punto de interacción distinta de su clasificador. De todas formas, el puerto puede tener interfaces definidas.

Los componentes y conectores pueden proporcionar la visión interna con la incorporación incluso de otros componentes y conectores internos. Por lo tanto se puede modelar estructuras inferiores del sistema.

**UML y Arquitectura de Software**

UML simple, nunca ha sido identificado suficientemente como para ser considerado un ADL. La posición simple de UML es justo la posición de un lenguaje de documentación.

En [12] se han discutido 5 posibles estrategias en el uso de UML para la descripción de arquitectura. Básicamente se supone que la arquitectura descrita por tipos y sus instancias. A continuación, se resumen las estrategias de la arquitectura de la transcripción.

| TIPOS | INSTANCIAS | USANDO DIAGRAMA DE NOTACIÓN | OBSERVACIONES |
|---|---|---|---|
| Clases | Objetos | Diagrama de Clase | El más intuitivo mapeo Orientado a Objetos |
| Estereotipos | Clases | Diagrama de Clase | Estereotipos |
| Clases | Clases | Diagrama de Clase | Vista homogénea |
| Estereotipos | Componentes | Diagrama de Componente | La aproximación a ADL |
| Subsistemas/ Paquetes | Paquetes | Diagrama de Componente o Despliegue | |

**Tabla 3. Estrategias en Descripción de Arquitectura por UML ([adoptado de [12]])**

La estrategia más importante es usar un diagrama de componente. De todas formas, los estereotipos sugieren que, como solución para este tipo, trae algunos problemas no triviales a UML. UML solo, no proporciona una técnica formal para la definición de estereotipos. Este problema ha sido identificado en la creación de UML y desde las primeras versiones de UML, es acompañado por OCL.

**UML Integrado con OCL**

Lenguaje de Restricción de Objetos (OCL), es un objeto formal basado en (MOF), el cual es un modelo de OMG para acceder a la estructura del modelo UML, la lógica de primer orden y la teoría de conjuntos. [13] OCL es parte integral de UML. Sin embargo, la mayoría de herramientas industriales de UML todavía no soportan OCL más de lo que permite poner en un texto.

Aunque el lenguaje es sugerido para la especificación de restricción, puede ser utilizado con éxito en muchos otros casos. Se ha demostrado su uso como lenguaje formal para demostrar estructuras de objetos.

La forma de uso más frecuente es la definición de restricciones en el modelado de un sistema. Incorporando OCL proporciona limitaciones sintácticas y semánticas adicionales, haciendo el modelo más claro, pero más complicado. De todos modos el uso del lenguaje es obviamente ventajoso para aclarar el modelo.

En consecuencia, OCL es una forma de definición de los estereotipos. Un estereotipo es una nueva clase de metamodelo. Esto significa que los estereotipos amplían el número de elementos que pueden utilizarse en el modelo, por otra parte proporcionan limitaciones específicas y elementos especializados en semántica y sintaxis.

**Perfiles de UML**

[9] Sugirió perfiles basados en estereotipos definidos en OCL para mapear ADLs particulares a UML. Este enfoque parece ser muy prometedor. El "Perfil" de UML es un conjunto de elementos, usualmente centrados en particular en dominios de negocio.

**UML + OCL = ADL?**

Esta es la pregunta principal. Aunque UML integrado con OCL proporciona todas las características esperadas de un ADL. La respuesta a esta pregunta tiende a No. Los conceptos de UML son claros, pueden ser completamente refinados para ajustarse a la mayoría de ADL. Sin embargo, considerar UML como ADL reduce las capacidades del lenguaje. En consecuencia UML tiende a ser utilizado inapropiadamente como ADL. La mejor practica debe ser adquirida mediante el uso especializado del perfil UML, posiblemente mapeando el perfil particular de ADL a UML (possibly profile mapping particular ADL to UML ). Por otro lado, UML debe ser considerado como una alternativa a los ADL, cuando se modela y se razona en arquitecturas de software. UML es justo la descripción industrial del lenguaje Nº 1 para las arquitecturas de software.

# CAPITULO 3. ONTOLOGÍAS Y FRAMEWORKS ONTOLOGICOS

Este capítulo describe los conceptos relacionados con la ingeniería ontológica y algunos conceptos básicos del trabajo soportado en frameworks ontológicos.

## 3.1 EL CONCEPTO DE ONTOLOGÍA Y SUS OPERACIONES

Smith y Welti [14] señalan que, en las ciencias de la información, John McCarthy en 1980, introdujo el término Ontología tomándolo en préstamo de la filosofía, al proponer que los constructores de sistemas inteligentes, basados en lógica, debían primero, listar todas las cosas que existen, construyendo una ontología del mundo. Igualmente, reporta la propuesta de Sowa en 1984 [Sow84], en el sentido de utilizar una ontología para catalogar todas las cosas que hay en el mundo, la forma cómo están ubicadas y como ellas trabajan.

Las siguientes son algunas definiciones de ontologías:

- “Una ontología es una especificación explícita de una conceptualización” [15].
- “Una ontología es una especificación explícita de una vista abstracta, simplificada de un mundo que hemos decidido representar” [16].
- “Es una teoría axiomática hecha explícita por los significados de un lenguaje formal específico. La Ontología en sistemas de información está diseñada para al menos una aplicación práctica y específica. Consecuentemente, dibuja la estructura de un dominio específico de objetos y da explicación por el significado pretendido de un vocabulario formal o protocolos que son empleados por agentes del dominio que está bajo investigación.” [17]
- “Una ontología es una especificación formal y explícita de una conceptualización compartida” [18].

En esta definición, se deben destacar los siguientes aspectos: conceptualización, en este contexto, es un modelo abstracto de cómo la gente piensa acerca de cosas en el mundo, usualmente, restringidas a un área en particular; especificación explícita, hace referencia a que los conceptos y relaciones del modelo abstracto están dados por definiciones y términos explícitos; en su aspecto formal permiten que sean interpretados por personas ó máquinas, las cuales - a través de una pieza de software (agente)- se encuentran habilitadas para este fin.

### Operaciones de Ontologías

En el caso de las operaciones, la granularidad es el grado de detalle con el cual se especifica una operación. El nivel más detallado se define como el nivel atómico, ya que la operación que está siendo definida, no se podrá dividir en gránulos más pequeños, sin que pierda significado. Los operadores ontológicos, por su granularidad, se clasifican en:

- Atómicos
- Compuestos, y
- Complejos

Operaciones atómicas:

Las operaciones atómicas representan las operaciones de grano más fino que se pueden ejecutar en un modelo ontológico específico. Corresponden a las operaciones de edición básica: Agregar, modificar y borrar, aplicables a conceptos simples o compuestos, relaciones, jerarquías de conceptos y relaciones, y axiomas.

Operaciones compuestas: las operaciones se pueden desagregar en múltiples operaciones atómicas y trabajan sobre una sola ontología. Constituyen una capa superior del conjunto de operadores atómicos, en tanto que se pueden describir en términos de operaciones atómicas.

Operaciones complejas

Procedimientos de cambio que involucran múltiples ontologías y que se pueden describir en términos de operaciones atómicas. Las operaciones donde participan dos o más ontologías, requieren de, al menos, un proceso donde se deben establecer correspondencias entre las entidades que componen las ontologías. Este es un proceso aproximativo y puede requerir de la intervención humana. Es de allí, donde se deriva la naturaleza compleja de estas operaciones.

- Emparejado de ontologías (Ontology Matching), es el proceso de encontrar relaciones o correspondencias entre entidades de diferentes ontologías.

- El proceso de emparejar ontologías y la alineación obtenida, aparece como parte de las tareas de mezcla e integración de ontologías, e incluso pueden ser utilizados en otras actividades como procesamiento de consultas y edición ontológica. Su solución constituye un reto clave en el campo de la ingeniería ontológica y ha sido abordado desde diferentes perspectivas.

- Alineación (Alignment), es el conjunto de correspondencias entre dos o más ontologías.La alineación es la salida de un proceso de matching.

- Mapeado (Mapping): Versión orientada o dirigida de un alignment, mapea las entidades de una ontología a al menos una entidad de otra ontología. Desde el punto de vista matemático se requiere que el objeto mapeado sea igual a su imagen, por ejemplo, que es una relación de equivalencia.

- Mezcla (Merge): Creación de una nueva ontología desde dos ontologías fuentes, posiblemente no disjuntas. Las ontologías fuentes permanecen inalteradas, mientras que la nueva ontología contiene el conocimiento de las ontologías iniciales.

- Integración (Integration): Es la inclusión en una ontología de otra ontología, expresando la afirmación con transparencia entre estas ontologías. La ontología integrada retiene el conocimiento de las dos ontologías integradas. Contrario a la mezcla, la primera ontología permanece inalterada, mientras que la segunda es modificada.

**Búsqueda y Publicación de Ontologías**

En la medida en que se han ido desarrollando ontologías, su búsqueda y publicación es una labor que plantea ciertas características de complejidad. El lenguaje de desarrollo, el modelo de representación del conocimiento utilizado en su proceso de diseño, el dominio que representa, el idioma en el cual ha sido desarrollado y si tiene capacidades de gestionar múltiples idiomas, son algunas de los metadatos que es deseable conocer cuando se desea encontrar ontologías para un dominio o tarea.

Un aspecto más complejo es la búsqueda y determinación de diferencias entre versiones de la misma ontología.

Este problema se ha abordado desde la perspectiva de los repositorios de ontologías, que son lugares donde pueden ser encontradas ontologías de dominio con una buena cantidad de metadatos asociados. Buscadores como Watson, permiten la configuración de repositorios de ontologías para agilizar las búsquedas.

## 3.2 FRAMEWORK ONTOLÓGICO

Se presentan a continuación algunos frameworks ontológicos, sin pretender ser exhaustivos.

### 3.2.1. ONTOCONCEPT

Como resultado de la Tesis doctoral del ingeniero Julio César Chavarro Porras [1], se desarrolló la herramienta OntoConcept como parte del marco de referencia para el modelado conceptual del cambio ontológico, la cual implementa una arquitectura cliente-servidor dos capas soportado en plugins, y utiliza el modelo conceptual basado en el Grafárbol para procesar las sentencias del lenguaje declarativo.

Una de las características del marco de referencia “Ontoconcept” es la capacidad para controlar el ciclo de vida de una ontología desde su creación. Esta característica, facilita la reconstrucción de la ontología a partir del log de cambios.
El framework Ontoconcept en su primera versión está compuesto por las siguientes áreas:

Menú principal, que da acceso a las opciones para seleccionar el espacio de trabajo, las ontologías disponibles en un espacio de trabajo, o los objetos que componen una ontología.

Consola, y Navegador de ontología que son las opciones del menú de segundo nivel que aparece debajo del menú principal.

- Menú de tercer nivel, asociado a la ventana activa. Permite ejecutar uno o varios comandos. Crear o limpiar el área de comandos asociado a una ontología. Cargar un archivo de comandos, o salvar los comandos de la sesión activa.
- Un área para escribir los comandos que se desean ejecutar.
- Una ventana de visualización, la cual permite visualizar el área de trabajo de memoria que se encuentra activo.
- Un área para visualizar mensajes, que se han generado durante la ejecución de los comandos.

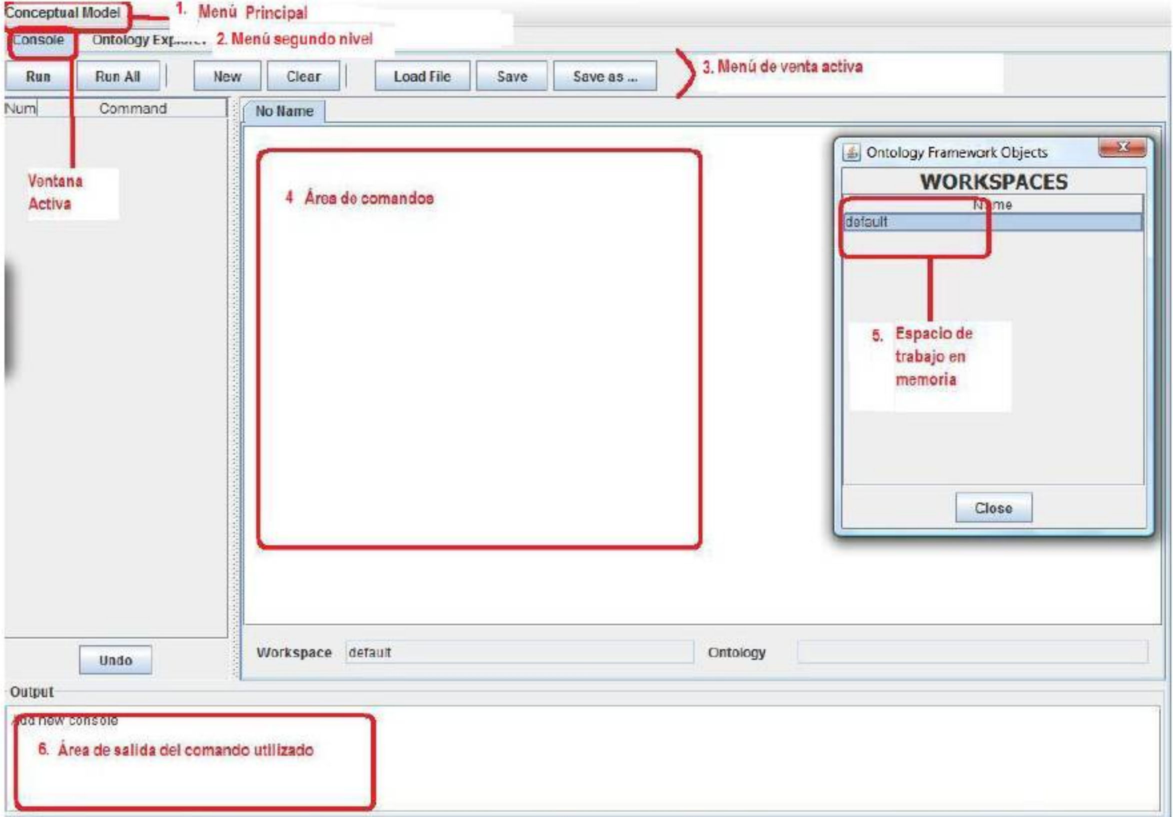



**Ilustración 1. Interfaz de usuario de Ontoconcept - Fuente [1]**

La creación de una ontología en OntoConcept, es independiente del lenguaje en que se va a construir la ontología, porque permite trabajar en la construcción o modificación de la ontología y posteriormente definir el lenguaje de implementación.

Después de creada la ontología, en un espacio de trabajo determinado, se está habilitado para trabajar con las entidades que la componen: conceptos, relaciones, tipos de datos, individuos, o axiomas.

### 3.2.2 Protegé

Protégé es un editor y framework ontológico libre y de código abierto para la construcción de sistemas inteligentes.

Protége fue diseñado y desarrollado inicialmente en la Universidad de Manchester. Protegé utiliza la API de OWL para sustentar todas las tareas de gestión ontológica, desde carga y guardado de ontologías, hasta la manipulación de ontologías durante la edición, para interactuar y ofrecer una opción de razonadores OWL. [19]

### 3.2.3 NeOn Toolkit

NeOn Toolkit [20] es una plataforma de ingeniería ontológica desarrollada por el EC-Funded Neon Project. Está basada en la framework Eclipse y soporta modelado de frame como ontologías extendidas por normas y modelado de DL-ontologías. Como plataforma es un entorno extensible, para el cual los socios de Neon están contribuyendo funcionalidad en forma de plug-ins.

En el marco del proyecto NeOn, se ha desarrollado la herramienta NeOn Toolkit, caracterizada por tener soporte colaborativo y de gestión de cambios.

NeOn Toolkit es un entorno de ingeniería ontológica de estado del arte, código abierto y multiplataforma, el cual proporciona apoyo completo para el ciclo de vida de ingeniería ontológica. La herramienta está basada en la plataforma de eclipse, un ambiente de desarrollo que conduce y proporciona un amplio conjunto de plug-ins, que cubre una variedad de actividades de ingeniería de ontologías.

En la documentación ofrecida en la página oficial de NeOn Toolkit [21], El objetivo de NeOn Toolkit es mejorar la capacidad de manejar múltiples ontologías en red que existen en un contexto particular, creadas en colaboración y que podrían ser altamente dinámicos y en constante evolución.

El editor OWL de NeOn Toolkit es una herramienta de modelado para la creación y mantenimiento de modelos semánticos (a menudo denominado “Ontologías”)
El editor OWL está compuesto por los siguientes paneles:

- Navegador Ontológico: el navegador ontológico muestra el proyecto de la ontología, sus ontologías correspondientes y sus jerarquías de la ontología actual.
- Panel de individuos, el panel de individuos muestra todas las instancias de la clase selecciona más reciente.

- Panel de Propiedades de Entidades, el panel de propiedades de entidades es área de trabajo principal para la definición y modificación de recursos de la ontología seleccionada.

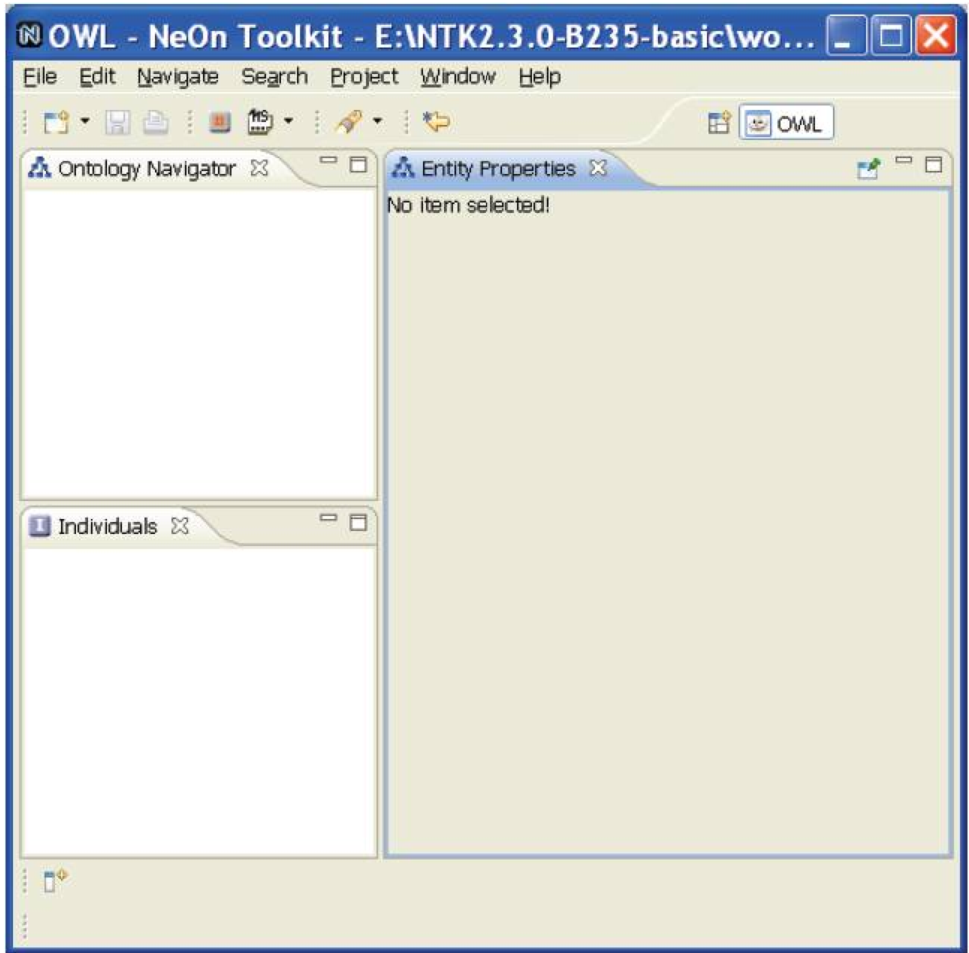



**Ilustración 2. Interfaz de Usuario NeOn Toolkit**

# CAPITULO 4. CARACTERÍSTICAS DE SOA Y ROA PARA WEB

Este capítulo está centrado en presentar el estado del arte de las arquitecturas que pueden ser utilizadas en el desarrollo de aplicaciones orientadas a internet. En este sentido, las arquitecturas distribuidas consisten de componentes que los clientes, así como otros componentes pueden acceder a través de la red mediante una interfaz y mecanismos de interacción que define la arquitectura.

## 4.1 ARQUITECTURAS ORIENTADAS A SERVICIOS (SOA)

Para [ISO 19119] un servicio está definido: "Parte distinta de la funcionalidad proporcionada por una entidad a través de interfaces". En términos computacionales, un servicio es una aplicación que proporciona información y/o funcionalidad a otras aplicaciones. Por lo general los servicios son interacciones no humanas que se ejecutan en servidores e interactúan con aplicaciones mediante una interface. [22]

Según [23], Un servicio es una funcionalidad concreta que puede ser descubierta en la red y que describe tanto lo que puede hacer como el modo de interactuar con ella. Los servicios pueden acoplarse dentro de una aplicación completa que proporciona servicios de alto nivel, con un grado de complejidad superior.

Un servicio es simplemente una aplicación con la que se puede interactuar a través del intercambio de mensajes bien definidos. Lo ideal es que los servicios que se creen tengan un ciclo de vida lo más amplio posible. Sin embargo, la agregación y configuración de estos debería poder cambiarse fácilmente. [25]

La arquitectura orientada a servicios (SOA) es un estilo arquitectónico para la construcción de aplicaciones de software que utilizan los servicios disponibles en una red, como la web, en conjunto con lo expresado por [40]. El W3C la define como un conjunto de componentes que pueden ser invocados, y cuya interfaz de descripciones se puede publicar y descubrir. Su objetivo es permitir que las actividades de negocio puedan ser orquestadas como componentes de aplicaciones.

La SOA en acuerdo con [27], se basa en cuatro abstracciones básicas: servicios, aplicaciones front-end, repositorio de servicios y bus de servicios. Un servicio consiste en una implementación que provee lógica de negocio y datos, un contrato de servicio, las restricciones para el consumidor, y una interfaz que expone físicamente la funcionalidad (ver figura 2). Las aplicaciones front-end consumen los servicios formando procesos de negocios. Un repositorio de servicios almacena los contratos de servicios y el bus de servicios interconecta las aplicaciones front-end y los servicios [42].

Los servicios representan grupos lógicos de operaciones relacionadas con algún concepto del negocio. Por su parte, los procesos del negocio se realizan en servicios orientados a procesos que se componen de secuencias definidas de invocaciones a servicios, mediante una orquestación de los mismos en lo que se conoce como

coreografías de servicios. Un business process management system (BPMS) es el aliado ideal para definir esta orquestación desde dónde invocar los servicios necesarios para realizar el proceso establecido. El término business process management (BPM) se refiere al conjunto de actividades que se realizan para optimizar o adaptar sus procesos de negocio a las nuevas necesidades organizacionales. Como en general están soportadas por herramientas de software, el término BPM es utilizado indistintamente para referirse a las herramientas y a las actividades en acuerdo a los expresado por [42].

En el diseño de una arquitectura SOA la granularidad y los diversos tipos de servicios deben ser definidos. La forma en la que se desarrolla SOA depende concretamente de la plataforma específica y en particular en lo que se refiere a la infraestructura (usualmente basada en TCP/IP) y en la forma en la que la interfaz esta implementada y descrita.

Propiedades de SOA: Adaptabilidad, flexibilidad, agilidad de desarrollo y reutilización de desarrollos existentes.

SOA es un estilo de arquitectura que plantea una serie de guías para la integración de diferentes sistemas informáticos [LN05]. Un objetivo fundamental de SOA es crear sistemas a partir de servicios desarrollados en distintos lenguajes de programación y soportados por plataformas diferentes, con el fin de dar solución completa a los procesos empresariales de una compañía. [Microsoft05] [Utilización de patrones para la construcción de SOA]

Para [25], la arquitectura orientada a servicios, es la más difundida en el mundo de los servicios web, es un estilo de arquitectura de software cuyo propósito es lograr un débil acoplamiento entre los componentes de software que interactúan entre sí.

SOA es el producto de una evolución de las siguientes tecnologías:

- **XML (Extensible Markup Language):** Es uno de los lenguajes más utilizados para el intercambio de datos sobre la Web, El lenguaje XML está concebido para describir objetos de datos llamados Documentos XML y describir de cierta forma los programas que los procesan. Está restringido bajo la norma ISO 8879 el Estándar Generalizated Markup Language. XML es un leguaje etiquetado, característica que le permite definir objetos de datos estructurados en partes bien definidas llamadas elementos.

- **RPC (Remote Procedure Call):** Es un protocolo que permite ejecutar una rutina en un equipo o segmento de red de manera remota.

- **Web Services:** Un Servicio Web es una aplicación de software identificada por un URI (Uniform Resource Identifier), cuyas interfaces se pueden definir, describir y descubrir mediante documentos XML. Los Servicios Web hacen posible la interacción entre agentes de software (aplicaciones) utilizando mensajes XML. Intercambiados mediante protocolos de Internet.[World Wide Web Consortium [W3C])], en general son servicios utilizados para transmitir y recibir datos por aplicaciones heterogéneas vía web, los servicios web aportan interoperabilidad entre las aplicaciones mediante el uso de XML, SOAP, WSDL y UDDI sobre los protocolos de la Internet. XML es usado para describir los datos, SOAP se ocupa para la transferencia de los datos, WSDL se

emplea para describir los servicios disponibles y UDDI se ocupa para conocer cuáles son los servicios disponibles.

SOA proporciona una metodología y un marco de trabajo basado en servicios, el objetivo es proveer la reutilización de un determinado servicio evitando el desarrollo de interfaces, lo cual permite proveer ventaja en la construcción de sistemas distribuidos.

Una Arquitectura Orientada a Servicio está compuesta por elementos funcionales y elementos relacionados con la calidad de servicio [26].

**Elementos funcionales:**

- Transporte: para llevar las peticiones de servicios y respuestas entre el proveedor y el consumidor del servicio.
- Protocolo de comunicación del servicio: se establece entre el proveedor y el consumidor del servicio.
- Descripción del servicio: describe servicio, cómo debe invocarse y que datos son requeridos para la invocación.
- Servicio: Describe un servicio que está disponible para utilizarse.
- Proceso de negocio: conjunto de servicios, invocados de una manera específica, con una determinada secuencia y con ciertas reglas particulares para llevar a cabo la funcionalidad de negocio requerida.
- Registro de Servicio: repositorio de servicios y las descripciones de las mismas.

**Aspectos de la calidad del servicio:**

- Política: reglas o condiciones entre el proveedor y consumidor de los servicios.
- Seguridad: Conjunto de reglas aplicados para la identificación, autorización y el control de acceso de los consumidores de servicios.
- Transacción: Conjunto de atributos que pueden ser aplicados a un grupo de servicios para conseguir un resultado consistente.
- Gestión: Conjunto de atributos que se aplican para manejar a los proveedores del servicio o a los consumidores.

Las definiciones de SOA identifican la utilización de servicios web empleando SOAP y WSDL en su implementación, sin embargo, se puede implementar SOA con cualquier tipo de tecnología que soporte servicios web, en comparación con una arquitectura orientada a objetos, SOA está formada por servicios débilmente acoplados y altamente operables por lo que los componentes de software se vuelven altamente reutilizables.

## 4.2 PRINCIPIOS FUNDAMENTALES DE SOA

Aunque no existe una estandarización de principios de orientación a servicios, algunos autores han proporcionado un conjunto de Principios muy relacionados con la orientación a Servicios.

En [27], se definen cuatro principios de la orientación a servicios:

- **Principio 1: Los límites son explícitos**

  Los servicios interactúan a través de límites explícitos de paso de mensajes bien definidos. Un límite representa el paso entre la interfaz publica del servicio y su implementación interna y privada. El límite de un servicio se publica a través de WSDL y puede incluir aserciones que indican las expectativas de un servicio dado.

  Hay varios principios que están relacionados a este primer principio:

  - Conocer los límites, Los servicios proporcionan un contrato para definir las interfaces públicas que ofrece. Toda la interactuación con el servicio tiene lugar a través de la interfaz pública. La interfaz consta de procesos públicos y representaciones de datos públicos.
  - El consumo de los servicios debe resultar fácil, La interfaz (contrato) del servicio se debe diseñar de modo que su evolución no implique la ruptura de contratos con los usuarios existentes.
  - Evitar interfaces RPC, Pasar mensajes de forma explícita constituye una técnica más adecuada que el uso de modelos RPC. Este enfoque aísla al usuario del mecanismo interno de la implementación del servicio.
  - El área de la superficie del servicio se debe mantener pequeña, Proporcionar pocas interfaces públicas bien definidas al servicio. Estas interfaces deben ser relativamente simples, y estar diseñadas para aceptar un mensaje de entrada bien definido y responder con un mensaje de salida igualmente bien definido.
  - Los detalles de la implementación interna (privada) no deben salir de los límites de un servicio, La entrada de los detalles de implementación en los límites del servicio dará lugar probablemente a un acoplamiento menos flexible entre el servicio y los usuarios del mismo

- **Principio 2**: **Los Servicios son autónomos**

  Los servicios son entidades que se implementan, versionan y administran de forma independiente.
  Aunque los servicios están diseñados para ser autónomos, ningún servicio es completamente independiente. Las soluciones basadas en SOA son fractales. Están formadas por una serie de servicios configurados para una solución específica. Las claves para crear servicios autónomos son el aislamiento y el desacoplamiento. Los

servicios se han diseñado e implementado de forma independiente entre sí y sólo se pueden comunicar a través de mensajes y políticas controlados por contratos.

Principios de diseño básicos que facilitan el cumplimiento del segundo principio:

- Los servicios se deben implementar y versionar de forma independiente al sistema en el que se implementan y consumen
- Los contratos se deben diseñar con la asunción de que una vez que se publiquen, no se podrán modificar. Este enfoque obliga a los desarrolladores a elaborar los diseños de sus esquemas de forma flexible.
- Se debe garantizar la ausencia de errores en los servicios. Para ello, es necesario adoptar un punto de vista pesimista. Desde la perspectiva del usuario, se deben planear niveles no confiables de disponibilidad y rendimiento del servicio. Asimismo, desde el punto de vista del proveedor, se debe esperar que el usuario hará un mal uso del servicio (deliberado o no) y que cometerá errores, tal vez sin notificar al servicio.

- **Principio 3: Los Servicios comparten esquema y contrato, pero no implementación**

La interacción de un servicio se debe basar únicamente en políticas, esquemas y comportamientos basados en contratos. Según [44] el mayor reto de este principio es hacer que estos esquemas y contratos no cambien una vez que se han publicado, evitando así el mínimo impacto en sus usuarios.

La línea que separa los datos internos de los externos es de vital importancia para la implementación y el uso de un servicio determinado. Los datos externos o públicos deben estar definidos por estándares, mientras que los internos o privados deben encapsularse en el interior de la implementación del servicio.

El contrato de un servicio consta de los siguientes elementos:

- Formatos de intercambio de mensajes definidos a través de esquemas XML.
- Patrones de intercambio de mensajes (MEP) definidos a través de WSDL
- Capacidades y requisitos definidos a través de WS-Policy.
- BPEL se puede utilizar como contrato de nivel de proceso empresarial para la agregación de varios servicios

Principios de diseño básicos que facilitan el cumplimiento del tercer principio:

- Garantizar que el servicio permanece estable con el fin de minimizar los efectos en los usuarios del mismo.
- Los contratos se deben diseñar para que sean lo más explícitos posible
- Mantener siempre definida la línea entre las representaciones de datos públicos y privados.
- Versionar los servicios cuando los cambios en el contrato de los mismos sean inevitables.

- **Principio 4: La compatibilidad de los servicios se basa en una directiva**

  El objetivo de este principio es poder configurar el comportamiento y las Expectativas de un servicio en función de los valores de ciertas directivas.
  Las expresiones de directiva ofrecen un conjunto configurable de semánticas interoperables que controlan el comportamiento y las expectativas de un servicio determinado. Una aserción de directiva identifica un comportamiento, el cual es un requisito (o capacidad) del tema de una directiva. [Utilización de patrones para la construcción SOA]

Otro autor que propone un conjunto de principios de la Orientación a Servicio es Thomas ERL [28], definiendo los siguientes:

- **Bajo Acoplamiento:** los servicios mantienen una relación que minimiza las dependencias y solo requiere que conserven una conciencia del uno al otro. Es decir, los servicios tienen que ser independientes, cada vez que se vaya a ejecutar un servicio, se debe acceder a él a través del contrato, logrando así la independencia entre el servicio que se va a ejecutar y el que lo llama. Si se consigue este bajo acoplamiento, los servicios podrán ser totalmente reutilizables.
- **Contrato de Servicios:** Los servicios se adhieren a un acuerdo de comunicaciones, tal como se definen colectivamente por uno o más descripciones de servicios y documentos relacionados. Todo servicio desarrollado, debe proporcionar un contrato en el cual figure: el nombre del servicio, su forma de acceso, las funcionalidades que ofrece, los datos de entrada de las funcionalidades y los datos de salida.
- **Autonomía:** Los servicios tienen el control sobre la lógica que encapsulan. Todo servicio debe tener su entorno de ejecución. De esta manera el servicio es totalmente independiente.
- **Abstracción:** Más allá de lo que se describe en el contrato de servicio, los servicios ocultan la lógica del mundo exterior.
- **Reutilización:** la lógica está dividida en servicios con la intención de promover la reutilización. Todo servicio debe ser diseñado y construido pensando en su reutilización dentro de la misma aplicación, dentro del dominio de aplicaciones o incluso dentro del dominio público para su uso masivo.
- **Componibilidad (Composability):** las colecciones de servicios pueden ser coordinadas y ensambladas para formar servicios compuestos. Todo servicio debe ser construido de tal manera que pueda ser utilizado para construir servicios genéricos de más alto nivel, el cual estará compuesto de servicios de más bajo nivel.
- **Apátrida (Sin estado):** Los servicios minimizan retener información específica para una actividad. Un servicio no debe almacenar ningún tipo de información. Esto se debe a que una aplicación está conformada por un conjunto de servicios, lo que implica que si un servicio almacena algún tipo de información se puede producir problemas de inconsistencia de datos.
- **Descubribilidad (Discoverability):** Los servicios están diseñados para ser descriptivos externamente, para que puedan ser encontrados y evaluados a través de mecanismos de detección disponibles. Todo servicio debe poder ser descubierto de

alguna manera, para que pueda ser utilizado, con el fin de evitar la creación accidental de servicios que proporcionen las mismas funcionalidades.

## 4.3 CARACTERÍSTICAS DE SOA

La arquitectura orientada a servicios contemporánea, contiene las siguientes características concretas:

- SOA está basado en estándares abiertos
- Arquitectónicamente Componible (composable)
- Capaz de mejorar la calidad del servicio

SOA apoya, promueve y fomenta:

- La interoperabilidad intrínseca
- Descubribilidad (discoverability)
- Federación
- Reutilización inherente
- Extensibilidad
- Capas de abstracción
- Débil acoplamiento
- Modelado de negocios orientado a servicios
- Agilidad organizacional

## 4.4 COMPONENTES DE SOA

Los principales componentes que conforman SOA son [29]:

- Sistemas proveedores de servicios
- Sistemas clientes de servicios
- Bus de servicios, el cual canaliza todos los flujos de información entre unos y otros.

### Evolución de SOA

El concepto SOA comenzó con mucho impulso como una tendencia en el área de la integración tecnológica, sin embargo, con el paso del tiempo el modelo tuvo su curva de decepción lo cual se produjo principalmente por la falta de conocimiento de cómo aplicar el concepto, así como comparar SOA y aplicarlo como otros modelos de integración tradicionales.

Aparte de esto, la necesidad de reutilización de funciones asociada a aparición del modelo multicanal debido a la movilidad, ha posicionado a SOA como la mejor opción para poder abrir los aplicativos existentes de una manera reutilizable.

Ha traído como consecuencia que sea necesaria la aplicación de un importante modelo consultivo antes, durante y posterior al proceso de implantación/evolución de una estrategia SOA, ver figura 4, para ello puede crearse servicios para afrontar una implementación SOA y que estaría cubierta con los siguientes bloques de servicios:

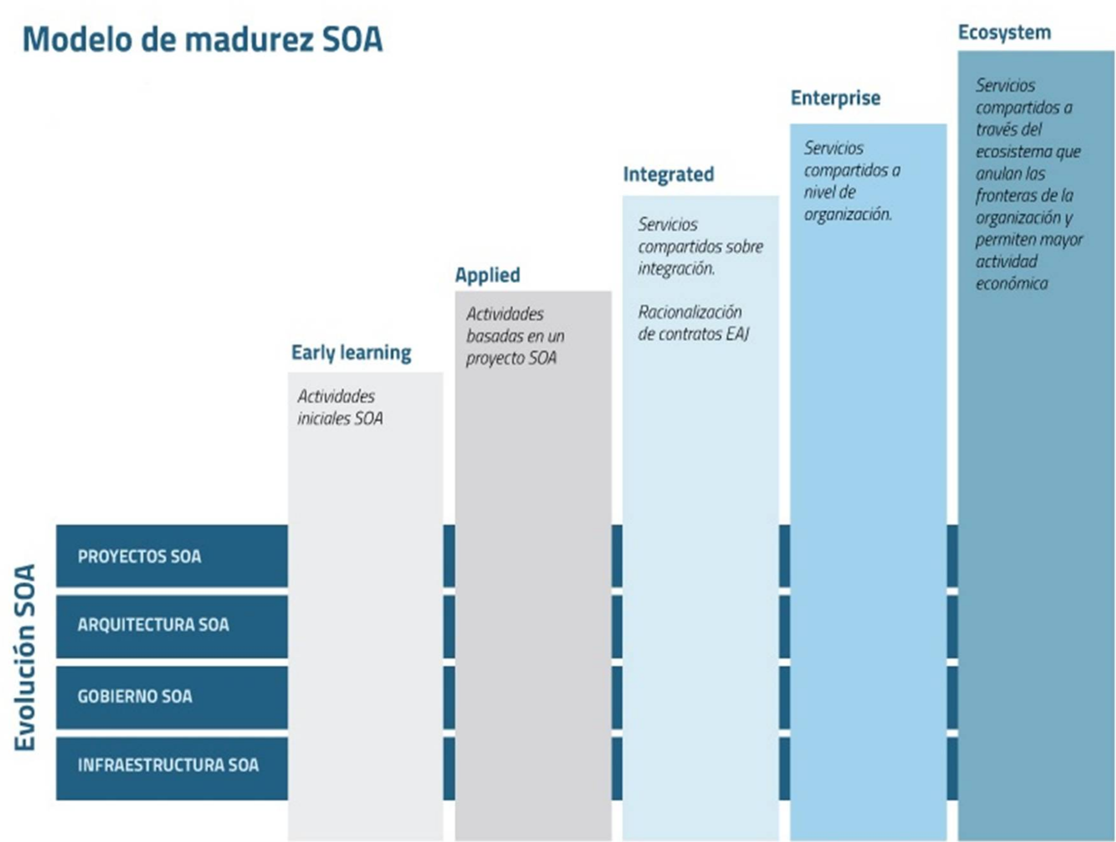



**Figura 4. Evolución de SOA, fuente: http://soaint.com/middleware/arquitectura-orientada-servicios-soa/**

**¿Por qué SOA?.**

Expresado por [37], SOA es un modelo de arquitectura tecnológica que surge de la aplicación del paradigma de orientación a servicios. Dicho paradigma no es una idea revolucionaria, sino que **surge de la influencia de diferentes modelos** como pueden ser: la orientación a objetos, BPM, orientación a aspectos, web services.

La idea subyacente consiste en descomponer la lógica de negocio de una organización (o partes de ella) en pequeñas unidades de funcionalidad. Estas pequeñas unidades son los **servicios**. Con esto conseguimos romper con el concepto de aplicaciones “silo”, donde se creaba una aplicación para resolver una necesidad de negocio concreta, otra

para resolver otra, entre otros. Lo que tendremos será una **plataforma transversal** formada por un **inventario de servicios** (o varios) de forma que no solventaremos las necesidades cambiantes del negocio creando nuevas aplicaciones sino **combinando diferentes servicios** (y creando nuevos servicios cuando corresponda). De esta forma conseguimos que los departamentos de IT y negocio estén alineados de forma que el primero pueda responder de manera ágil a las exigencias del segundo.

Dicho esto, alguien puede pensar: *"Entonces para tener una SOA lo que tengo que hacer es crear web services, a lo bestia"*. Pues no, ojalá fuese tan fácil.

Es cierto que para poder tener una arquitectura SOA necesitamos tener una buena base de servicios. Pero no vale con crearlos de cualquier manera. Estas son algunas consideraciones que debemos tener en cuenta:

- Debemos contar con una buena base de servicios **reutilizables** o muli-propósito. Servicios no diseñados para resolver una necesidad de negocio específica. Los servicios de utilidad y entidad son perfectos en este sentido. De esta forma podremos crear múltiples **composiciones** basadas en ellos conforme las necesidades de negocio van cambiando.
- Nuestros servicios deben contar con un contrato estandarizado (ya sea WSDL, un documento o ambos) con un modelo de datos normalizado dentro de todo nuestro inventario. De esta manera facilitamos la interacción entre servicios. Sustituimos el concepto de integración por **interoperabilidad intrínseca**.
- Debemos **categorizar y registrar** todos nuestros servicios (recursos de plataforma) de forma que, en cualquier momento, podamos consultar con qué recursos contamos, qué funcionalidades encapsulan y cómo interacturar con ellos.
- Debemos evitar el solapamiento de funcionalidades entre servicios como parte de la labor de **gobierno SOA** que debemos ejercer sobre nuestra plataforma.
- Debemos evitar "casarnos" con una tecnología en concreto (Java, .Net, y otros…) a la hora de crear nuestros servicios. Los servicios deben poder implementarse con la tecnología que consideremos necesaria en cada momento sin que esto afecte al resto de la plataforma.

Como se ve, no resulta tan fácil diseñar una arquitectura orientada a servicios. Sin embargo, los servicios web basados tanto en SOAP como REST son **excelentes opciones** a la hora de crear los servicios que conformarán la plataforma ya que, sin lugar a dudas, reúnen las condiciones ideales para poder diseñar servicios que cumplan con los principios de diseño alineados con SOA.

## 4.5 ARQUITECTURAS ORIENTADAS A RECURSOS (ROA)

Las arquitecturas Orientadas a Recursos son un estilo arquitectural y están basadas en el concepto de recurso; cada recurso es un componente distribuido directamente accesible, que es manejado a través de un estándar, una interfaz común que hace posible el manejo de recursos. Las plataformas RESTful basadas en la tecnología de desarrollo REST permiten la creación de ROA. [30]

Es un estilo arquitectural basado en REST, su principal función es exponer recursos, esta arquitectura pretende ser un método sistemático para desarrollar servicios en la Web que se adhieran a los principios de REST.

Los principales conceptos de ROA son los siguientes: [RESTful Web services]

- Recurso: Un recurso es cualquier cosa que sea lo suficientemente importante para ser referenciada.
- Nombre del recurso: URI (Uniform Resource Identifier) la URI es el Nombre y Dirección del Recurso, Identificación única del recurso, cada recurso debe tener al menos un único punto de acceso que lo distinga de otros.
- Representación del Recurso: Información útil sobre el estado actual de un recurso.
- Enlace entre Recursos: Enlace a otra representación del mismo u otro recurso.

### 4.5.1 PROPIEDADES DE ROA

La arquitectura orientada a recursos define cuatro propiedades que un sistema adherido a ella debe tener [31]:

- Direccionabilidad (Addressability): Una aplicación es direccionable si expone los aspectos interesantes de su conjunto de datos a través de recursos. Cada recurso tiene su propia y única URI.

- Apátrida (Statelessness): Cada petición HTTP pasa en completo aislamiento, por lo tanto, incluye toda la información necesaria para que el servidor cumpla con la solicitud. El servidor nunca se basa en información de solicitudes previas.

- Considerando la carencia de estado (statelessness) en términos de la direccionabilidad. La direccionabilidad dice que cada pieza interesante de información que el servidor puede proporcionar debe ser expuesta como un recurso dado su propia URI. La carencia de estado dice que los posibles estados de un servidor también son recursos, y deben recibir sus propias URIs.

- Conectividad (Connectedness): En los RESTful Services las representaciones son hipermedia: documentos que contienen no solo datos, si no links a otros recursos.
  En la disertación de Fielding se presentó el siguiente axioma: “La hipermedia como el motor del estado de la aplicación”, este axioma significa que el estado actual de un HTTP “sesión” no es almacenado en el servidor como un estado de recurso, pero si seguido por el cliente como un estado de aplicación y creado por la ruta que el cliente lleva a través de la web.

  La conectividad exige que los recursos no vivan de manera aislada, sino que establezcan enlaces entre ellos, entregando al cliente los estados vecinos al actual y posibilitando así la navegación

- Una Interfaz uniforme: Todos los recursos en ROA son accedidos a través de la misma interfaz común la cual es HTTP sencillo, cabe señalar que el uso de una interfaz común no necesariamente significa que tola la información necesaria que permita la interoperabilidad y la colaboración entre recursos estén disponibles.

  **ROA y REST**

  Lo importante de REST no es que utiliza la interfaz uniforme específica que define HTTP. REST especifica una interfaz uniforme, pero no especifica qué interfaz uniforme. GET, PUT y el resto no son una interfaz perfecta para todos los tiempos. Lo que es importante es la uniformidad: que cada servicio use la interfaz HTTP de la misma forma.

  El punto no es que GET es el mejor nombre para una operación de lectura, pero GET significa “leer” a través de la web, no importa que recurso lo esté usando. Dado una URI de un recurso, no hay duda de cómo se obtiene una representación: se envía una solicitud HTTP GET a la URI. Sin la interfaz uniforme se tendría que aprender cómo cada servicio espera para recibir y enviar información. Las reglas podrían incluso ser diferentes para diferentes recursos dentro de un solo servicio.

  Sin la interfaz uniforme, se obtiene una multiplicidad de métodos que toman el lugar de GET. Cada servicio habla un lenguaje diferente.

El manejo de los recursos se realiza utilizando las operaciones propias del protocolo HTTP

| **Operación HTTP** | **Descripción** |
|---|---|
| PUT | Crea un nuevo recurso (y el identificador único correspondiente) |
| GET | Recupera la representación de un recurso |
| DELETE | Elimina el recurso |
| POST | Modifica el recurso |
| HEAD | Obtiene información de metadatos sobre el recurso |

**Tabla 4. Operaciones de Recursos**

### 4.5.2 REST

REST es un estilo arquitectural para sistemas hipermedia distribuidos tales como la web. REST queda definido por tres restricciones de interfaz: identificación de recursos, manipulación de recursos mediante representaciones de estos mensajes auto-descriptivos, e hipermedia como motor del estado de la aplicación. Cabe destacar que REST no es una especificación ni un estándar, ya que es tan solo un estilo de arquitectura [25]. Aunque REST no es un estándar, está basado en estándares:

- HTTP
- URL

- Representación de los recursos: XML/HTML/GIF/JPEG/, entre otros.
- Tipos MIME: text/xml, text/html, entre otros.

La finalidad de REST es exponer recursos a través de URIs y HTTP, no servicios a través de interfaces de mensajería. No debe por tanto confundirse con otros protocolos basados en RPC como SOAP O XML-RPC.

Los objetivos de REST son:

- Escalabilidad de la interacción con los componentes: La Web ha crecido exponencialmente sin degradar su rendimiento.
- Generalidad de interfaces: Gracias al protocolo HTTP, cualquier cliente puede interactuar con cualquier servidor HTTP sin ninguna configuración especial.
- Puesta en funcionamiento independiente: HTTP permite la extensibilidad mediante el uso de las cabeceras, a través de las URIs, a través de la habilidad para crear nuevos métodos y tipos de contenido.
- Compatibilidad con componentes intermedios: Los más populares intermediaros son varios tipos de proxys para Web. Algunos de ellos, las caches, se utilizan para mejorar el rendimiento. Otros permiten reforzar las políticas de seguridad: firewalls.

Para manipular los recursos, los componentes de la red (clientes y servidores) se comunican a través de una interfaz estándar (HTTP) e intercambian representaciones de estos recursos (los ficheros que se descargan y se envían). La petición puede ser tramitada por cualquier número de conectores (por ejemplo, clientes, servidores, cachés, túneles, etc.) Así, una aplicación puede interactuar con un recurso conociendo el identificador del recurso y la acción requerida, no necesitando conocer si existen cachés, proxys, cortafuegos, túneles o cualquier otra cosa entre ella y el servidor que guarda la información. La aplicación, sin embargo, debe comprender el formato de la información devuelta (la representación), que es por lo general un documento HTML o XML, aunque también puede ser una imagen o cualquier otro contenido.

# CAPITULO 5. RECOMENDACIÓN ARQUITECTURAL PARA FRAMEWORK ONTOLOGICO

De acuerdo con la arquitectura planteada para el framework Ontoconcept en la tesis doctoral en el capítulo 6 de la misma [1b], se plantea la arquitectura del marco de referencia para la gestión conceptual del cambio ontológico, donde se describen los retos planteados en la construcción de un sistema de gestión, colaborativo y distribuido, del cambio ontológico. La siguiente lista, presenta las características que debe permitir configurar un sistema de gestión ontológica:

- La gestión del cambio en las ontologías es simultáneamente gestión de la evolución y de las versiones.
- Edición sincrónica frente a edición asincrónica.
- Edición constante (continuada) frente a almacenado periódico.
- Con etapa de curación o sin ella
- Cambios registrados frente a cambios no registrados

Los retos planteados, operan como un conjunto de restricciones funcionales, en el estudio del modelo arquitectural necesario para la construcción del marco de referencia planteado. El modelo arquitectural propuesto para un sistema de gestión de cambio, tiene como propósito, servir de base para la construcción de un sistema de gestión de ontologías, con la capacidad de gestionar el cambio de manera declarativa

La ilustración 3, presenta una descripción modular del Marco de Referencia (framework) para la gestión conceptual del cambio en ontologías. Tecnológicamente, el marco de referencia es un servicio Web, extensible, portable, con especificación modular basada en WSDL 2.0, y guiado por los principios del software libre.

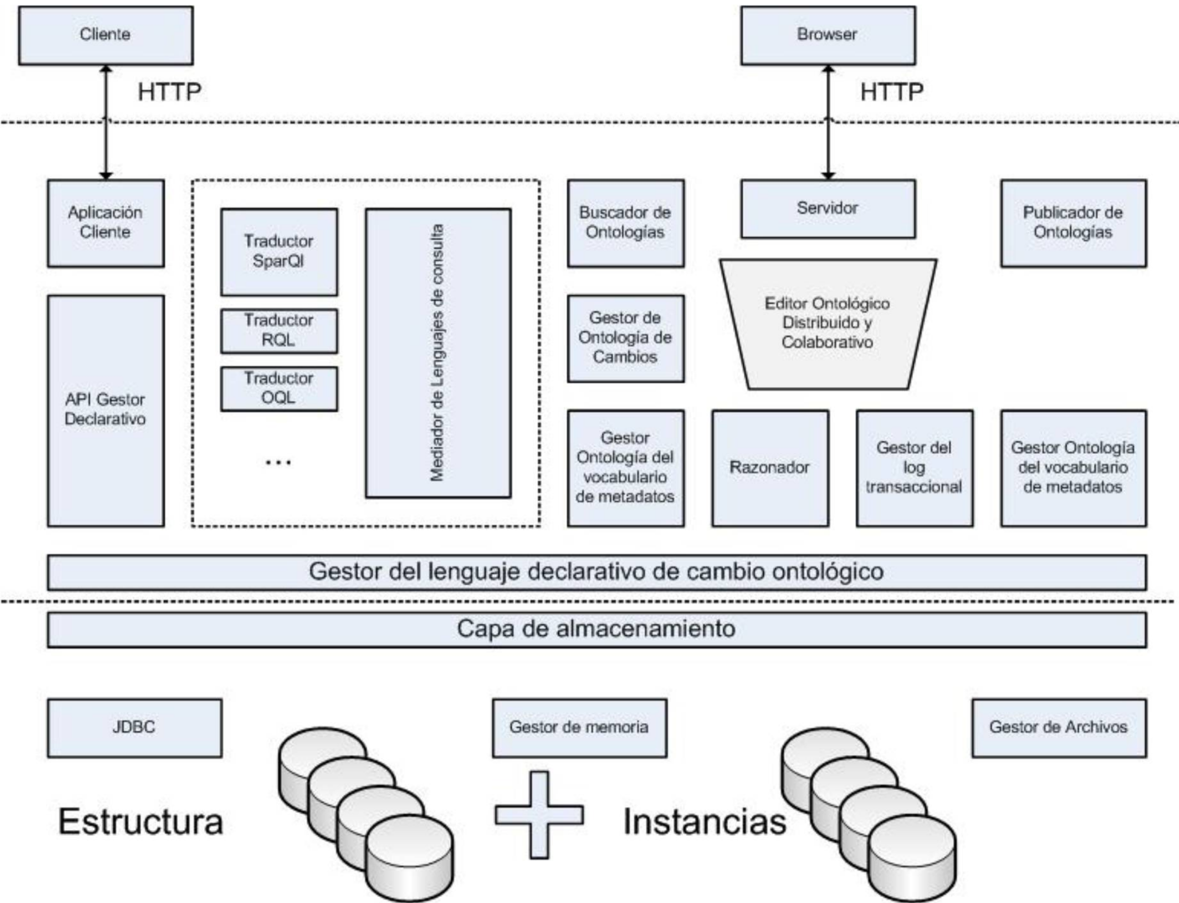



**Ilustración 3. Arquitectura de marco de referencia para la gestión conceptual del cambio ontológico**

## 5.1 Principales Módulos de Ontoconcept

Los principales módulos de la arquitectura propuesta son:

1. Gestor de memoria. Administra las áreas de memoria principal. Entre las características funcionales más importantes se encuentra la de permitir la administración de múltiples espacios de trabajo y cargar múltiples ontologías en cada espacio de trabajo.
2. Gestor del lenguaje declarativo. Contiene el núcleo del lenguaje declarativo que permite gestionar las fases de ejecución de los operadores declarativos que componen el lenguaje.
3. Cargador / Descargador de ontologías. El cargador es el responsable de, evaluar la consistencia del archivo, y cargar la ontología en la memoria. El descargador, exporta la ontología desde la memoria hacia el lenguaje de implementación seleccionado.
4. Gestor del log transaccional. Registra las operaciones de cambio, en términos de operaciones atómicas del lenguaje declarativo. El control de los cambios se registra, cada vez que un conjunto de cambios se registra como permanente, dejando la traza de la sesión de usuario. Se puede exportar el log transaccional al lenguaje de implementación de la ontología.

5. Razonador
6. Gestor de Ontología de vocabulario de metadatos
7. Gestor de Ontología de cambio genérico
8. Editor Ontológico
9. Buscador de ontologías
10. Publicador de ontologías
11. Mediador de lenguajes de consulta
12. Traductores para lenguajes de consulta de ontologías: Sparql, OQL, RQL
13. API del gestor declarativo
14. Aplicación Cliente

## 5.2 Módulos relacionados con la edición Ontológica

Para proponer la arquitectura orientada a servicios, se selecciona un conjunto de módulos de la arquitectura propuesta para el marco de referencia. Dichos módulos se consideran trascendentales en la gestión del cambio ontológico y forman parte del proceso de edición ontológica, por lo tanto, tienen una interacción permanentemente con el módulo principal de edición (Editor Ontológico), durante el ciclo de vida de una ontología, los módulos son los siguientes:

1. Editor, Encargado de crear una nueva ontología o editar una ontología existente.
2. Gestor de Persistencia, encargado de administrar archivos o gestores de repositorios ontológicos (Repositorios Ontológicos y Repositorios de Base de Datos).
3. Cargador o Lector de Ontologías,
4. Exportador o Grabador de ontologías
5. Visualizador de Ontologías, encargado de definir la técnica de visualización, se puede hacer razonamiento sobre DL (lógica de descripción)
6. Razonador, Encargado de realizar el razonamiento sobre las ontologías, como verificar la consistencia de la ontología
7. Gestor de Cambios,
8. Gestor de Operadores Complejos
9. Configurador

La siguiente ilustración presenta una vista externa del sistema, en la cual se puede observar la interacción de los módulos seleccionados.

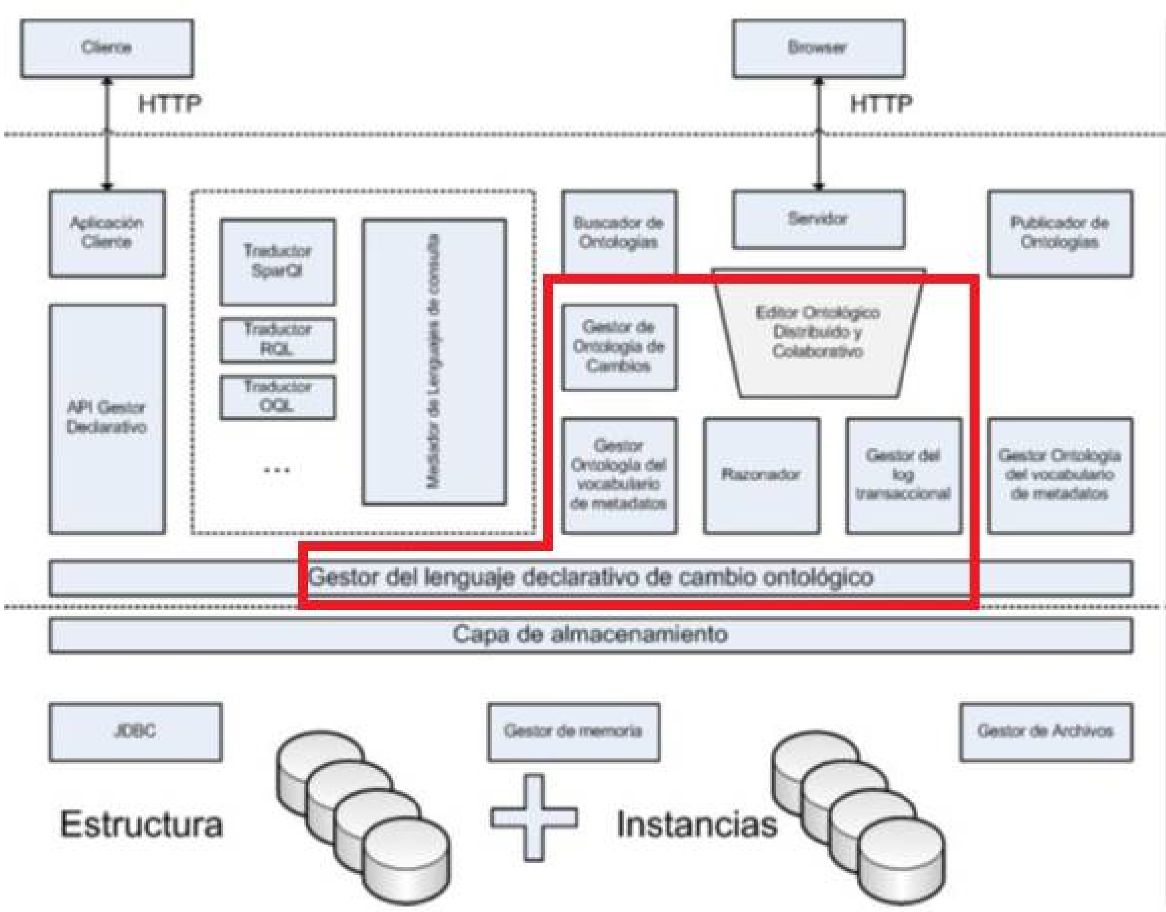



**Ilustración 4. Módulos relacionados al proceso de edición ontológica**

## 5.3 Propuesta Arquitectural

Con el fin cumplir el objetivo de este trabajo, el cual es proponer la línea base arquitectural del framework Ontoconcept, se utilizará el Lenguaje de Modelado Unificado (UML) como Lenguaje de Descripción Arquitectural (ADL), para modelar la arquitectura recomendada, usando mecanismos de extensión, comúnmente llamados estereotipos, los cuales se utilizan para explicitar la especialización de los elementos de los diagramas dando un significado adicional a dichos elementos, enriqueciendo la descripción de la arquitectura. La selección de UML como ADL debido a que como se expresó anteriormente (referencia capitulo LDA), un ADL especifica formalmente una arquitectura de software, modelando componentes, conectores y configuraciones, utilizando una sintaxis simple, entendible y gráfica. Con base en lo anterior UML 2.0 extendido cumple las características y por ende funciones de un ADL.

### 5.3.1 UML 2.0

Según OMG (Object Management Group), UML es el lenguaje de especificación más usado para modelar no solo la estructura de aplicación, comportamiento y arquitectura,

sino también procesos de negocio y estructura de datos. UML está conformado por Modelos y el Lenguaje de Restricción de Objetos (OCL). Tiene la característica de ser extendido por Metamodelos y Perfiles. Adicionalmente los modelos se componen de diagramas y a su vez los diagramas se componen de estereotipos y conjunto de reglas.

UML ofrece varios diagramas para modelar los sistemas, entre los diagramas ofrecidos se encuentra el Diagrama de Componentes, el cual define el comportamiento del sistema, permitiendo modelar la especificación, construcción y visualización de la arquitectura del sistema. Los elementos proporcionados por el diagrama de componentes son:

- Componentes
- Conectores
- Puertos
- Interfaces
- Clasificadores estructurados

Al comparar los elementos ofrecidos por un ADL y los elementos proporcionados por el diagrama de componentes de UML. Se puede inferir que efectivamente UML funciona como un ADL. Para este trabajo se usará UML para especificar la arquitectura propuesta para el framework, haciendo uso de su modelado visual (Diagrama de Componentes), el cual permite construir, especificar y visualizar la arquitectura.

### 5.3.2 Arquitectura Recomendada usando UML como ADL

Teniendo en cuenta el uso de UML como ADL y el diagrama de componentes ofrecido se realiza la especificación de cada uno de los elementos de la línea base arquitectural propuesta.

#### 5.3.2.1 Componentes

Un componente es un grupo de objetos o componentes más pequeños, los cuales interactúan entre sí y se combinan para dar un servicio [32]. Un componente representa una pieza reutilizable del software que proporciona alguna funcionalidad significativa. Un componente es un tipo de clasificador estructurado cuya relación y estructura interna pueden ser mostrados en un diagrama de componentes. Un componente colabora con otros componentes a través de interfaces bien definidas para proporcionar la funcionabilidad del sistema y podría a su vez estar compuesto de componentes que proporcionan su propia funcionalidad.

Los componentes se pueden clasificar en:

- **Vista Externa**: presenta las interfaces que posee y la forma de conexión
- **Vista Interna**: forma en la que el componente está internamente constituido

#### 5.3.2.2 Conectores

Los conectores son los enlaces para comunicar dos o más instancias, son enlaces entre puertos o interfaces. Los conectores no se pueden asociar a descripciones de comportamiento o atributos.

#### 5.3.2.3 Puertos

Los puertos están asociados a interfaces, adicionalmente están asociados a descripciones de comportamiento. Permiten a los componentes comunicarse con el exterior.

#### 5.3.2.4 Interfaces

Las interfaces se clasifican en:

- **Interfaces Provistas:** representan un grupo de mensajes o llamadas que un componente implemente y que otros componentes o sistemas pueden utilizar

- **Interfaces Requeridas:** representa un grupo de llamadas o mensajes que el componente envía a otros componentes o sistemas externos.

#### 5.3.2.5 Clasificadores estructurados

Representan la estructura interna de los componentes.

Siguiendo los elementos planteados por el diagrama de componentes de UML, la siguiente figura representa la vista externa de los componentes (módulos seleccionados) para la recomendación arquitectural del framework Ontoconcept.

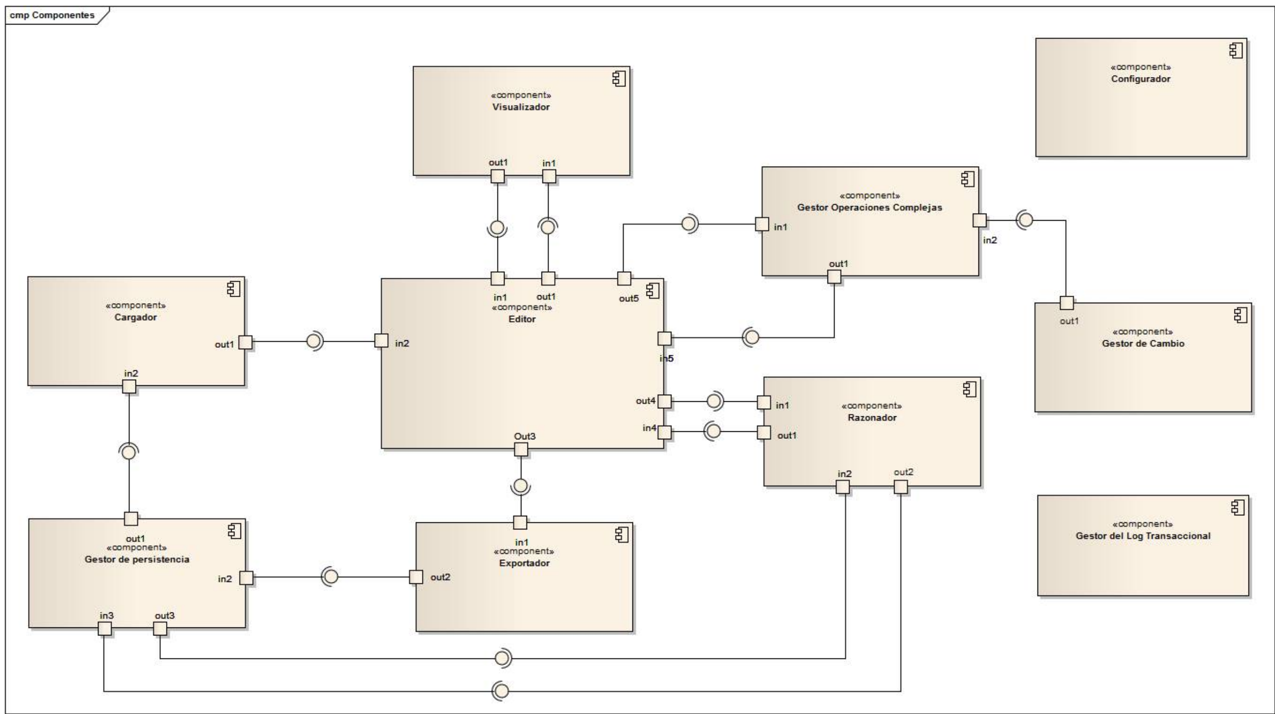



**Ilustración 5. Diagrama de Componentes - Vista Externa de la Arquitectura**

### 5.3.3 Especificación de Componentes y Conectores

A continuación, se presenta un ejemplo de vista interna, correspondiente al componente Cargador desplegado en la vista externa.

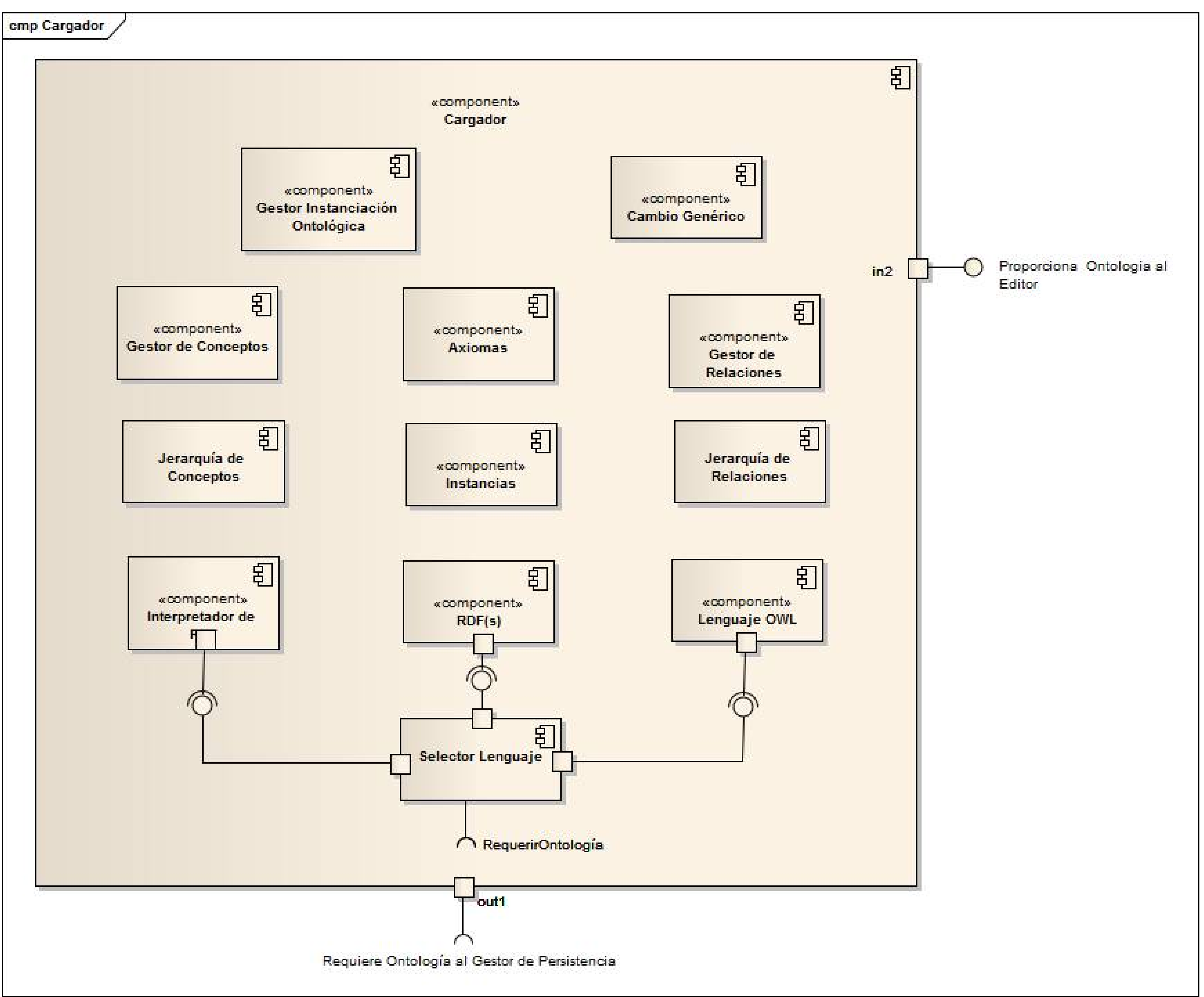



**Ilustración 6. Vista Interna del Componente Cargador**

### 5.3.3.1. Plantilla para la descripción de cada componente

Una plantilla o dispositivo de interfaz, suele proporcionar una separación entre la forma o estructura y el contenido. Es un medio o sistema, que permite guiar, portar, o construir, un diseño o esquema predefinido; agiliza el trabajo de reproducción o de muchas copias idénticas o casi idénticas. Si se quiere un trabajo más refinado, más creativo, la plantilla no es sino un punto de partida, un ejemplo, una idea aproximada de lo que se quiere hacer.

Las plantillas, como norma general, pueden ser utilizadas por personas o por sistemas automatizados.

En relación con los sistemas computacionales, por ejemplo, paquetes de programas basados en la web, utilizan en la actualidad un sistema de plantillas para separar la lógica del programa del formato visualizado y posiblemente unos pocos operadores lógicos para permitir una mejor adaptabilidad de la plantilla.

Se propone entonces una plantilla que servirá de guía para el establecimiento de las partes que conforman las componentes. Con los siguientes elementos:

**Nombre Componente:** se usa para diferenciar los diferentes componentes.

**Característica:** hace referencia al tipo de implementación, ejemplo: UML 2.0.

**Tipo:** las componentes pueden manejar distintos tipos. Se pueden instanciar elementos de los distintos tipos para especificar una arquitectura.

**Restricciones:** mediante los requisitos establecidos se obtiene como el Número de puertos de una componente, atributos y descripciones de comportamiento (protocol state machines) de interfaces. Se puede restringir el número de puertos de una componente. Las interfaces pueden incluir atributos y descripciones de comportamiento.

**Evolución:** tiene relación con permitir el refinamiento de componentes y agregar nuevas componentes. Además, permite especificar familias de aplicaciones al separar los tipos de componentes de las instancias de ellas, clasificadores estructurados y subtipos. El refinamiento se puede lograr utilizando las diferentes vistas provistas por UML.

**Requerimientos Funcionales:** tiene su injerencia con base a las restricciones y reglas asociadas a las componentes.

**Requerimientos No Funcionales:** No, salvo uso de anotaciones en referencias a los de calidad, dado porque UML no posee un mecanismo para modelar propiedades no funcionales.

Para la definición de la propuesta de plantilla se tuvieron en cuenta estas circunstancias:

- ✓ UML 2.0 mejor que sus antecesores
- ✓ UML 2.0 cumple con casi todas las características para ser un ADL
- ✓ Se comporta mejor que la mayoría de los ADLs conocidos
- ✓ La especificación de una Arquitectura se puede hacer mediante la conexión de los diferentes niveles de especificación de UML

Se presenta una plantilla para realizar la descripción y especificación de cada componente planteado en la propuesta arquitectural

| | |
|---|---|
| **Nombre Componente** | |
| **Característica** | |
| **Tipo** | |
| **Restricciones** | |
| **Evolución** | |
| **Requerimientos Funcionales** | |
| **Requerimientos No Funcionales** | |

**Tabla 5. Plantilla Especificación de Componente**

## 5.4 Aspectos No funcionales

De igual manera, serán de particular importancia las *propiedades no funcionales* del sistema de software, pues influyen notoriamente en la calidad del mismo. Estas propiedades tienen un gran impacto en el desarrollo y mantenimiento del sistema, su operabilidad y el uso que éste haga de los recursos (Buschman et al., 1996).

Entre las propiedades no funcionales más importantes se encuentran: modificabilidad, eficiencia, mantenibilidad, interoperabilidad, confiabilidad, reusabilidad y facilidad de ejecución de pruebas (Kazman et al., 2001). Bass et al. (1998) proponen que el término "requerimiento no funcional" es disfuncional, debido a que implica que tal requerimiento no existe, o que es una especie de requerimiento que puede ser especificado independientemente del comportamiento del sistema. En este sentido, Bass indica que debe hacerse referencia a atributos de calidad, en lugar de propiedades no funcionales.

Son los aspectos del sistema, que en general, no afectan directamente a la funcionalidad necesitada, sino que definen la calidad y las características que el sistema debe soportar. Esto tiene una relación directa con la Arquitectura.

Arquitectura como puente el propósito de la arquitectura de software es de ser un puente entre los objetivos de negocio y el sistema. Este puente se logra a través de la satisfacción de los atributos de calidad del sistema, que son parte de los requerimientos del mismo.

### 5.4.1. ¿Por qué los atributos de calidad son usados para el análisis?

Los atributos de calidad son la base para a la evaluación de una arquitectura, pero por si solos no son suficiente para juzgar lo adecuado de la arquitectura. Generalmente, los requerimientos son escritos de la siguiente manera:

- "El sistema debe ser robusto."
- "El sistema debe ser modificable."
- "El sistema debe ser seguro."
- "El sistema debe tener una respuesta aceptable y rápida."

Cada una de las frases anteriores está sujeta a diferentes interpretaciones y malos entendidos. Lo que uno puede considerar robusto, otro puede considerarlo apenas aceptable.

El punto es que los atributos de calidad no son cantidades absolutas, existen en un contexto con metas específicas. En particular:

- Un sistema es modificable (o no) con respecto a un tipo de cambio específico.
- Un sistema es seguro (o no) con respecto a un tipo de amenaza específica.
- Un sistema es confiable (o no) con respecto a la ocurrencia de un tipo de falla específica.
- Un sistema es de alto desempeño (o no) con respecto a un criterio específico del mismo.
- Una arquitectura es construible (o no) con respecto a restricciones de tiempo y presupuesto específicas.

No parece razonable, considerar que un sistema pueda alguna vez, por ejemplo, ser completamente confiable bajo toda circunstancia (pensar en problemas de energía, de natura, meteorológicos, entre otros). Dado esto, es importante que el arquitecto entienda perfectamente bajo qué circunstancias un sistema debe ser confiable para ser considerado aceptable. Por lo tanto, el primer trabajo que debe realizar una evaluación de arquitectura es obtener las metas específicas de calidad ante las cuales la arquitectura será juzgada.

Los atributos de calidad son características que permiten definir de forma objetiva, lo que es un sistema de calidad.

– Ausencia de defectos

– Cualidades del sistema (desempeño, seguridad, etc.)

Los atributos de calidad se derivan de los objetivos de negocio.

– Ejemplo:

- Objetivo de negocio
  - – Ingresar al mercado norteamericano en un periodo de 6 meses
- Atributos de calidad que pueden asociarse:
  - – Modificabilidad: para cambiar el idioma.
  - – Escalabilidad: para soportar la carga de usuarios.

Muchas veces los atributos de calidad no son incluidos dentro de los requerimientos del sistema. Cuando se incluyen atributos de calidad frecuentemente son parecidos a lo que sigue:

– El sistema deberá responder de forma eficiente.

– Las páginas deberán ser ligeras.

– El sistema deberá tener alta disponibilidad.

Los atributos de calidad deben ser expresados de forma cuantitativa, de lo contrario no es posible evaluar si el sistema los satisface o no.

– Una métrica permite establecer un acuerdo sobre un concepto abstracto.
Ejemplo
– El sistema se considera rápido si el tiempo de respuesta entre petición y respuesta es menor a 10 segundos.

### 5.4.2. ¿Entonces cómo especificar los atributos de calidad?

Si algunas de estas metas no son específicas o son ambiguas, se debe pedir a los stakeholders que ayuden al equipo de evaluación a reescribirlas. El mecanismo a utilizar para representar estas metas es el de *escenario*. Un escenario es una pequeña descripción de la interacción de un stakeholder con el sistema. Los escenarios son parecidos a los *casos de uso*.

Un escenario de atributo de calidad es un requerimiento específico a un atributo de calidad, compuesto de 6 partes, según lo expuesto por [4] en segunda y tercera edición, adicional se proponen los siguientes atributos de calidad para ser evaluados mediante el manejo de escenarios: *Disponibilidad, Modificabilidad, Desempeño, Seguridad, Facilidad de prueba, Usabilidad.*

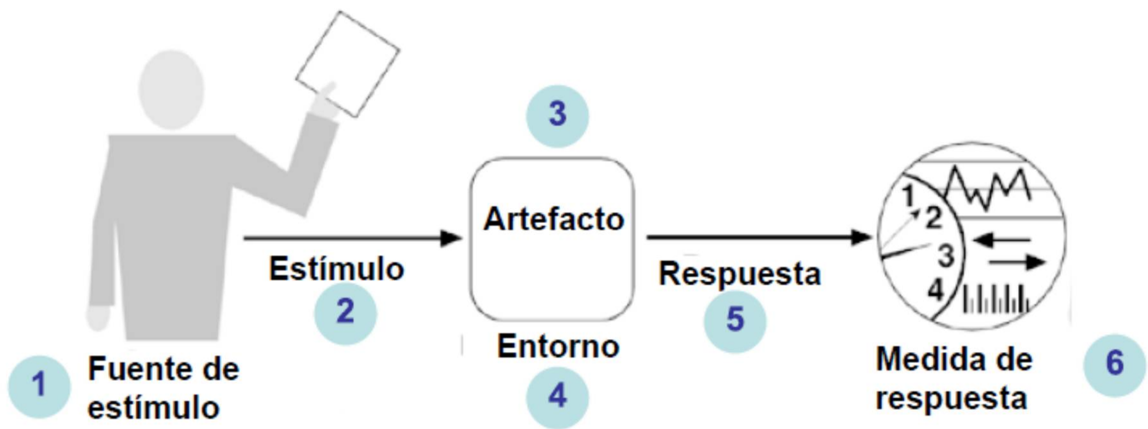



**Ilustración 7. Partes para el manejo de escenarios de calidad.**

**Fuente:** Abowd, G., Bass, L., Clements, P., Kazman, R., & Northrop, L. (2012). *Recommended Best Industrial Practice for Software Architecture Evaluation* (No. CMU/SEI-96-TR-025). Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst. Third edition.

Atributo de calidad afectado: Atributo de calidad relacionado con el escenario

1. Fuente del estímulo: quien o que genera el estímulo.
2. Estímulo: lo que se quiere llevar a cabo.
3. Entorno: condiciones dentro de las cuales se presenta el estímulo.
4. Artefacto: parte del sistema que recibe el estímulo.
5. Respuesta: actividad que ocurre luego de la llegada del estímulo.
6. Medida de la Respuesta: criterio para testear el requerimiento.

## 5.5 ESCENARIOS POR ATRIBUTOS DE CALIDAD

**5.5.1. La disponibilidad**: concierne a las fallas del sistema y a sus consecuencias asociadas.

– Una falla del sistema ocurre cuando el sistema deja de entregar un servicio acorde a su especificación. Dicha falla es observable por los usuarios del sistema (a diferencia de un defecto).

Un concepto importante relacionado es el tiempo que toma reparar el sistema una vez que falla. La distinción entre fallas y defectos permite discutir acerca de estrategias automáticas de reparación: si se ejecuta un código defectuoso y el sistema se logra recuperar del defecto sin que esto sea observable, entonces no hay falla.

La disponibilidad de un sistema es la probabilidad de que se encuentre en un estado de operación cuando se le necesita.

En general se define como:

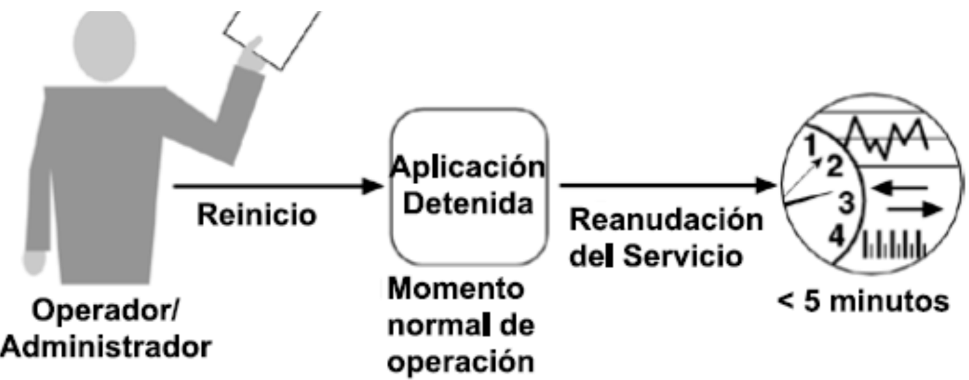
$$\frac{\text{tiempo promedio antes del fallo}}{\text{tiempo promedio antes de fallo} + \text{tiempo promedio de reparación}}$$

De ahí se habla de sistemas con un x% de disponibilidad.

Ejemplo: 364 días al año de operación para 1 día de reparación = 0.997 o 99.7% de disponibilidad

En general, el tiempo en que un sistema se detiene para mantenimiento no se cuenta en la definición.

**Escenario de Disponibilidad:** Ocurre una falla crítica dentro de la aplicación en un momento normal de operación que causa que la aplicación se detenga. La aplicación es reiniciada y el servicio se reanuda en un tiempo no mayor a 5 minutos.



**5.5.2 La Modificabilidad:** se enfoca en el costo de realizar cambios. Al respecto existen dos

Preocupaciones:

**– ¿Qué cosa(s) cambia(n) (artefacto)?**

• Funcionalidad
• Plataforma
• Entorno de operación (sistemas con quien interactúa)
• Protocolos de comunicación
• Atributos de calidad del sistema
• Capacidad (usuarios y operaciones soportadas)

**¿Cuándo se hace el cambio y quién lo hace (entorno)?**

– En el código fuente
– Durante la compilación
– Durante la construcción (selección de librerías)
– Durante la construcción
– Durante la ejecución
– Puede ser hecho por un desarrollador, administrador o usuario

**Escenario de Modificabilidad:** Un desarrollador agrega un nuevo lenguaje al

sistema en tiempo de mantenimiento. El lenguaje es agregado exitosamente sin necesidad de recompilar.

**5.5.3. El desempeño:** tiene que ver con:

– Cuánto tiempo le toma a un sistema responder cuando ocurre un evento o

– Cuántos recursos ocupa o requiere un sistema para procesar un evento.

Se deben considerar

– Número de fuentes de eventos

– Patrones de llegada de los eventos

Un escenario de desempeño inicia con la llegada de alguna petición de servicio.

– Satisfacer la petición involucra consumo de recursos, posiblemente mientras se satisfacen otras peticiones.

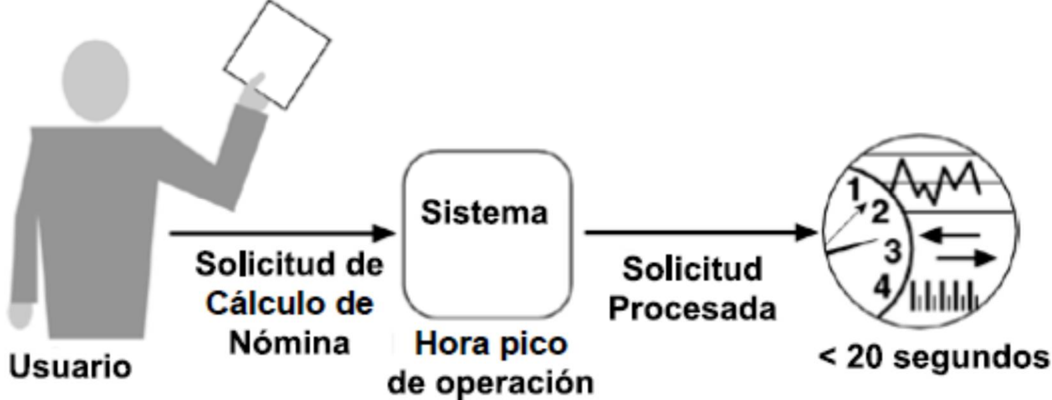
– Se revisan los patrones de llegada de evento periódicos o aleatorios (de acuerdo a distribución probabilística) o bien de forma esporádica.

– La respuesta se caracteriza por:

• **la latencia**: (tiempo entre estímulo y respuesta), deadlines, ancho de banda (transacciones/s), jitter (variación en latencia), numero de eventos no procesados y datos perdidos dado que el sistema estaba ocupado.

• **total de memoria ocupada**: ancho de banda requerido, tamaño ocupado de cachés, buffers, colas o pilas, espacio en disco duro utilizado, porcentaje de procesador utilizado, entre otros.

**Escenario de Desempeño:** Un usuario realiza una solicitud de cálculo de nómina en un momento pico de operación. La solicitud es procesada y la respuesta es devuelta en un tiempo no mayor a 20 segundos.




**5.5.4. La seguridad:** es una medida de la habilidad del sistema de resistir usos no autorizados mientras que sigue proveyendo sus servicios a usuarios legítimos.

Un intento de romper la seguridad se llama ataque.

– Intento no autorizado de acceder a datos o servicios

– Intento no autorizado de modificar datos

– Intento de limitar acceso a usuarios legítimos

La seguridad puede ser caracterizada como un sistema que provee *no-repudiación, confidencialidad, integridad, aseguramiento, disponibilidad y auditoría*.

**No-repudiación:** Propiedad de una transacción (acceso o modificación de datos o servicios) que hace que la participación en ella no pueda ser negada por ninguna por sus participantes.

**Confidencialidad:** Propiedad de que los datos o servicios sean protegidos de accesos no autorizados. Esto significa que un hacker no puede acceder a su saldo de cuenta bancaria.

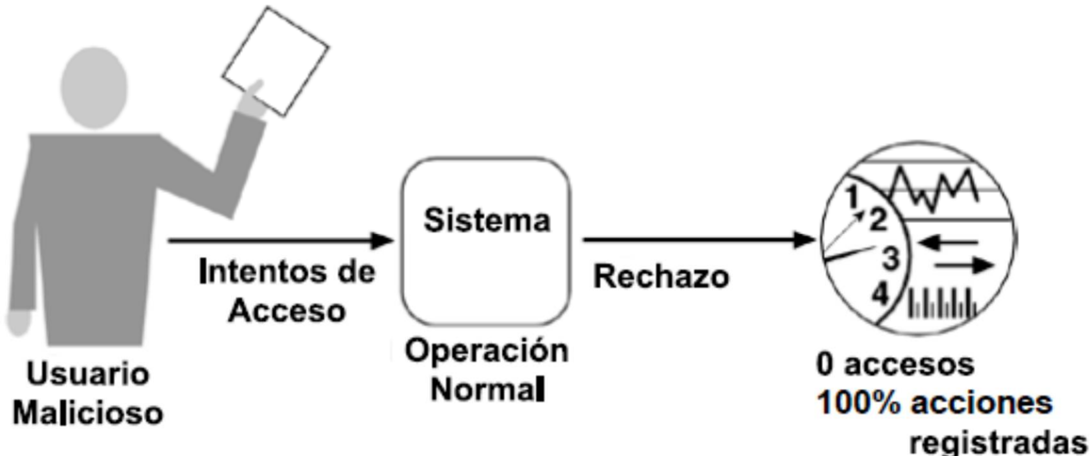
**Integridad:** Propiedad de que los datos o servicios sean entregados como se supone que debían serlo. La calificación en sus cursos no puede ser modificada después que el profesor la registró.

**Aseguramiento:** Propiedad de que los participantes en una transacción sean quienes dicen ser.

**Disponibilidad:** Propiedad de que el sistema siga disponible en caso de ataque para uso legítimo.

**Auditoría:** Propiedad de que el sistema de seguimiento a actividades a nivel suficiente como para reconstruirlas.

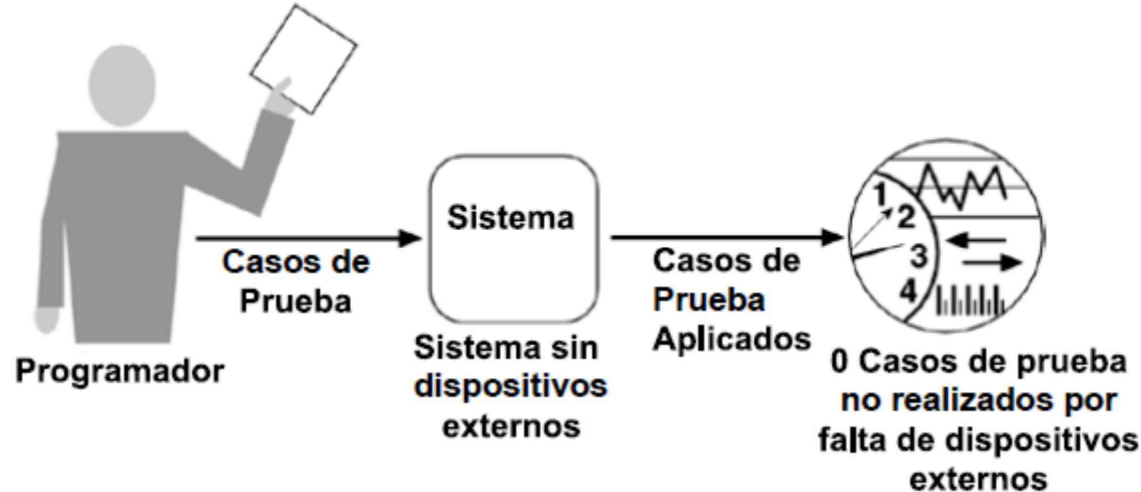
**Escenario de Seguridad:** Un usuario malicioso intenta acceder funciones para las cuales no tiene autorización durante la operación normal del sistema. El usuario no logra acceder a las funciones y el 100% de las acciones del usuario son registradas.




**5.5.5 La facilidad de pruebas:** se refiere a la facilidad con la cual se puede hacer que el sistema demuestre sus defectos a través de pruebas. Para que un sistema sea fácil de probar, debe ser posible controlar el estado interno de los componentes y sus entradas para poder observar sus salidas.

Las medidas de respuesta tienen que ver con qué tan efectivas son las pruebas para descubrir defectos y cuánto tiempo toma realizar pruebas con cierto grado de cobertura.

**Escenario de Facilidad de Pruebas:** Un desarrollador realiza pruebas de sistema cuando aún no se dispone de los dispositivos externos con los cuales se conectará la aplicación. Es posible ejecutar el 100% de los casos de prueba.

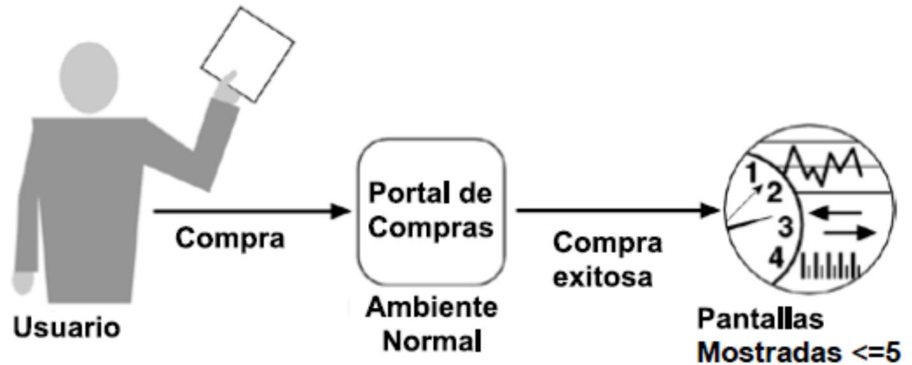



**La usabilidad**: se enfoca en qué tan fácil es para el usuario lograr una tarea deseada y el tipo de soporte que provee el sistema.

Áreas:

– Características de aprendizaje provistas por sistema
– Uso eficiente del sistema
– Minimizar impacto de errores
– Adaptación del sistema a necesidades de usuario
– Aumento de confianza y satisfacción

**Escenario de Facilidad de Uso:** Un usuario realiza la compra de un boleto de concierto a través del portal de compras. La compra es realizada de forma exitosa sin que el sistema muestre más de 5 distintas pantallas.



Los atributos de calidad no son los únicos requerimientos que se consideran para el desarrollo de la arquitectura. Es necesario considerar, además:

– casos de uso primarios y
– restricciones

En conjunto, estos requerimientos son conocidos como los drivers de la arquitectura.

**Casos de uso primarios:** Se refiere a un subconjunto de casos de uso del sistema que son elegidos por:
– Ser de mayor importancia para el negocio.
– Ejercitar un número importante de elementos arquitectónicos.
– Presentar algún riesgo particular.

**Restricciones:** Las restricciones representan condiciones que limitan las decisiones que se pueden tomar dentro del proyecto.

Pueden ser técnicas o administrativas
– El proyecto deberá ser desarrollado en Java.
– La documentación deberá estar redactada en inglés.
– El sistema a desarrollar debe enterar sus operaciones al ERP de la organización.

**Ingeniería de requerimientos:** En la ingeniería de requerimientos se pueden considerar cuatro grupos de actividades, estos grupos se aplican directamente a los atributos de calidad y demás drivers:
– Captura
– Especificación
– Priorización
– Validación

El método QAW (Quality Atributtes Workshop)[5] cubre todas estas etapas, para lo expresado en este aparte de requisitos no funcionales.

**Objetivos de Calidad**
Se pueden añadir a los objetivos del negocio para ajustarlos al proyecto, y pueden ser agrupados por prioridades de acuerdo a los lineamientos del mismo. Se plantea unas categorías que pueden estar ayuda en el momento de elegir.

**Esenciales**

- Funcionalidad > Corrección
- Funcionalidad > Robustez

[5] Disponible en https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=6927

**Esperados**

- Funcionalidad > Exactitud
- Funcionalidad > Compatibilidad
- Funcionalidad > Corrección medible
- Usabilidad > Comprensibilidad y Legibilidad
- Usabilidad > Apoyo para tareas
- Usabilidad > Eficiencia
- Usabilidad > Seguridad
- Usabilidad > Consistencia y Familiaridad
- Usabilidad > Satisfacción Subjetiva

**Deseados**

- Confiabilidad > Consistencia en carga
- Confiabilidad > Consistencia bajo concurrencia
- Confiabilidad > Disponibilidad bajo carga
- Confiabilidad > Longevidad
- Eficiencia
- Escalabilidad
- Escalabilidad > Desempeño bajo carga
- Escalabilidad > Grandes volúmenes de datos
- Operabilidad
- Capacidad de mantenimiento > Comprensibilidad
- Capacidad de mantenimiento > Capacidad de evolución
- Capacidad de mantenimiento > Capacidad de prueba

## 5.6 ¿Cuáles son las salidas de una evaluación arquitectónica?

Las salidas de una evaluación arquitectónica son información e ideas sobre la arquitectura.

### 5.6.1 Lista priorizada de los atributos de calidad requeridos

Una evaluación arquitectónica solo puede proceder si se conoce el criterio de adecuabilidad. Es más, la obtención de los atributos de calidad requeridos contra los cuales la arquitectura será juzgada, constituye la mayor parte del trabajo. Pero ninguna arquitectura puede tener una lista interminable de atributos de calidad, por lo tanto, se utilizan métodos de priorización consensuados.

### 5.6.2 Riesgos y no riesgos

Los riegos son decisiones arquitectónicas potencialmente problemáticas. Los no riesgos son buenas decisiones, que confían en asunciones que con frecuencia son implícitas en la arquitectura.

Documentar riesgos y no riesgos consiste en:

Una decisión arquitectónica (o una decisión que no ha sido tomada).
Una respuesta específica al atributo de calidad que está siendo tratado, junto con las consecuencias del nivel predecible de la respuesta.
Una base lógica por el efecto positivo o negativo que la decisión tuvo en alcanzar el requerimiento del atributo de calidad.

Para que un no riesgo se mantenga como tal, las asunciones no deben cambiar, o al menos si cambian, la designación como no riesgo debe ser re-justificada.

## 5.7. ¿Cuáles son los costos y beneficios de realizar una evaluación arquitectónica?

El mayor beneficio que brinda la evaluación de una arquitectura, es que descubre los problemas que, si se hubiesen dejado sin descubrir, habría sido mucho más costoso corregirlos luego. En breve, una evaluación produce una mejor arquitectura.

Algunos beneficios más que brinda la evaluación se presentan a continuación.

### 5.7.1 Reúne a los stakeholders

En una evaluación arquitectónica es, comúnmente, la primera vez en que la mayoría de los stakeholders se encuentran. Es más, es la primera vez que el arquitecto se encuentra con ellos. Todos comparten un objetivo, lograr un sistema exitoso.

### 5.7.2 Fuerza una articulación en las metas específicas de calidad

El rol del stakeholder es articular las metas de calidad que la arquitectura debería alcanzar para ser considerada exitosa. Estas metas no siempre son capturadas en algún documento de requerimientos.

### 5.7.3 Fuerza una explicación clara de la arquitectura

El arquitecto convoca a un grupo de personas, no en privado, para explicarles la creación de la arquitectura, en detalle y sin ambigüedades. El proyecto se ve beneficiado cuanto antes se realice esta explicación.

### 5.7.4 Mejora la calidad de la documentación de la arquitectura

Generalmente, la evaluación pide documentación que aún no está terminada, entonces se designa alguien del plantel a terminarla. Una vez más, el proyecto se ve beneficiado porque se ingresa al desarrollo mejor preparado al tener los documentos terminados.

### 5.7.5 Descubre oportunidades de reuso

Los stakeholders y el equipo de evaluación provienen de afuera del proyecto de desarrollo, pero trabajan o están familiarizados con otros proyectos. Por lo tanto, ambos

están en una buena posición para detectar componentes que pueden ser reusados en otros proyectos, o conocer componentes que ya existen que pueden ser utilizados en el proyecto actual.

### 5.7.6 Resultan mejoras en las arquitecturas

Las organizaciones que practican la evaluación arquitectónica como un estándar de su proceso de desarrollo, reportan una mejora en la calidad de las arquitecturas que son evaluadas. La evaluación de arquitecturas resulta en mejores arquitecturas no solo después de hecha, pero antes también.

Los costos de la evaluación arquitectónica son todos costos de personal y costos de oportunidad por aquel personal que participa de la evaluación en lugar de hacer otra cosa.

Si otro atributo de calidad, además de los mencionados antes, es importante para determinado proyecto, este también debe formar parte de la evaluación. En particular, el ATAM permite definir nuevos atributos de calidad a ser utilizados en la evaluación.

La Arquitectura de Software condiciona las características del producto final en cuanto a cualidades como desempeño y mantenibilidad. El Architecture Tradeoff Analysis Method (ATAM) es una metodología para evaluar Arquitecturas de Software basada en los atributos de calidad especificados para el sistema, desarrollada por el Software Engineering Institute (SEI).

ATAM se propone su uso en práctica con Ontoconcept, para evaluar el producto obtenido y realizar un segundo ciclo de desarrollo sobre el mismo, mejorando aspectos de riesgo identificados principalmente para el atributo de calidad performance.

## 5.8 Propuesta de aplicación No funcional para Ontoconcept.

### Propuesta de diseño de arquitectura para Ontoconcept

Diseñar una arquitectura se refiere a tomar decisiones de diseño para producir las estructuras asociadas al sistema.

Estas decisiones tienen, sin embargo, un impacto profundo en el sistema, por lo que

– Se deben tomar de manera temprana en el desarrollo.

– Se debe cuidar que estas decisiones sean pertinentes, es decir, están enfocadas a los aspectos realmente importantes del sistema.

¿Qué pasa si no sucede?

El no considerar la arquitectura en el desarrollo de forma temprana puede acarrear problemas importantes. Tales como:

– El sistema no satisface las características, por ejemplo: el desempeño, la disponibilidad o la seguridad de forma adecuada.

– La introducción de cambios en el sistema es excesivamente costosa.

– Se complica el desarrollo del sistema.

El escenario expuesto, es: *Se quiere introducir soporte a disponibilidad previo a la liberación*.

Con esta situación que es común entre algunos desarrolladores de aplicaciones tipo ensayo error, pero un acertado diseño parte de un buen despliegue de diseño de arquitectura, con esta premisa se tiene entonces que es mejor realizar un planeamiento arquitectural con la idea de establecer una buena toma de decisiones en lo que respecta a los frameworks y otros elementos del diseño.

Para realizar un buen diseño de arquitectura es necesario:

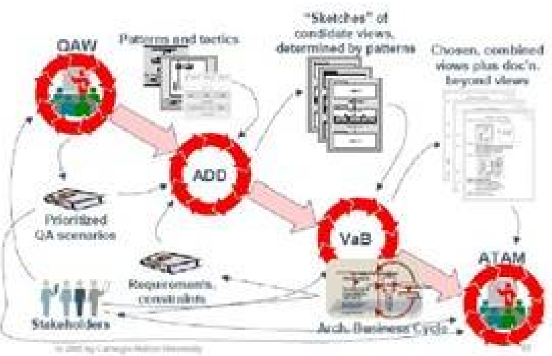
– Tener claridad sobre ciertos requerimientos que influyen con alta prioridad en la estructuración del sistema; los “drivers de la arquitectura”.

– Realizar diseño de forma sistemática.

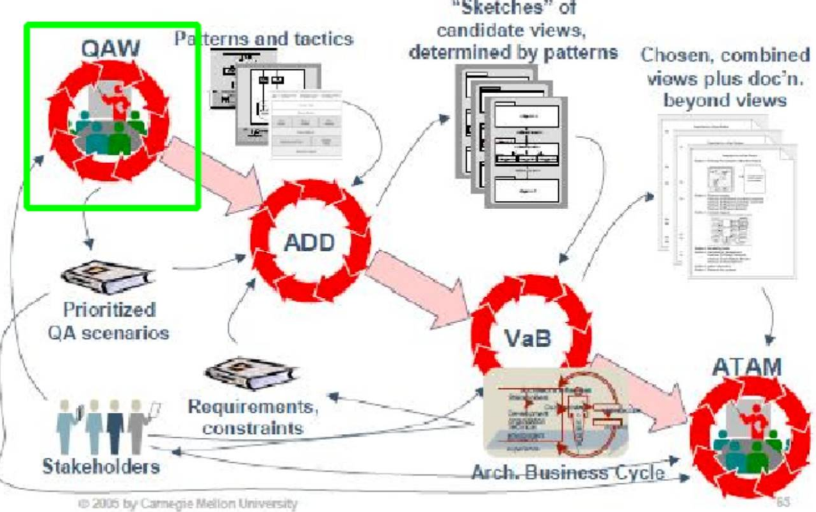
– Poder comunicar el diseño.

Por otro lado, es necesario asegurar que el diseño generado tiene calidad.

El SEI (Software Engineering Institute) propone una serie de métodos enfocados a apoyar en la realización del diseño de la arquitectura de software.



El taller de atributos de calidad (QAW) está enfocado en la captura y priorización de drivers.




Para el caso de Ontoconcept la teoría asociada a los atributos de calidad y drivers arquitectónicos se debe revisar lo planteado en los puntos 5.4 y 5.5 del presente trabajo, y los roles planteados en el capítulo 1. Se recomienda tener en cuenta los siguientes puntos:

**Fase de requerimientos**

La meta de esta fase es tener una lista de drivers arquitectónicos priorizados asociados al FO, caso de estudio con el fin de proceder al diseño de la arquitectura junto con escenarios de los atributos de calidad propuestos y que son considerados más importantes par FO. Inicialmente Disponibilidad.

La Visión se obtiene con el seguimiento de los pasos descritos

El equipo deberá estudiar un documento de visión asociado con el FO, cuidando particularmente los siguientes aspectos:

- ✓ Identificación de los objetivos
- ✓ Identificación de la prioridad de las necesidades
- ✓ Identificación del entorno de operación del sistema
- ✓ El instructor establecerá un tiempo corto para aclaración de dudas
- ✓ El equipo deberá llenar para cada característica que debe tener el sistema, la prioridad que considere debe tener la característica y el o los objetivos con los que se le pueden relacionar.
- ✓ El equipo deberá ejercitar los Pasos 4 al 8 del Taller de Atributos de Calidad.

**Pasos del QAW a cubrir**

El paso 1 se realizó al inicio con la presentación del Taller de Atributos de Calidad.

El paso 2 se realiza con la lectura del Caso de Estudio.

El paso 3 se realiza con la lectura de la parte final del Caso de Estudio.

El paso 4 lo debe realizar el equipo

- ✓ Identificar casos de uso para el sistema derivados de las necesidades expresadas en el documento de visión y deberá elegir un subconjunto de casos de uso que se consideran como relevantes para el diseño de la arquitectura
- ✓ Identificar restricciones a considerar para el diseño de la solución.

El paso 5 lo debe realizar el equipo, en donde deberá identificar escenarios de atributos de calidad usando la lluvia de ideas.

El paso 6 lo debe realizar el equipo, en donde debe consolidar los escenarios.

El paso 7 lo debe realizar el equipo, en donde debe asignar prioridades a los escenarios de atributos de calidad.

El paso 8 lo debe realizar el equipo, en donde debe detallar los escenarios que tuvieron las mayores votaciones del paso 7.

Al final de las actividades de requerimientos se deberá tener un entregable de seguimiento del caso de estudio, con:

Los casos de uso elegidos.

- ✓ Idealmente: usar diagrama de casos de uso e iluminar los casos de uso elegidos.
- ✓ Las restricciones identificadas.
- ✓ Los escenarios de atributos de calidad priorizados.
- ✓ Los 5 escenarios de atributos de calidad de mayor prioridad detallados.

Análisis crítico de la experiencia hasta el momento respondiendo a las siguientes preguntas:

¿Qué dificultades encontraron?

¿Qué salió bien?

¿Qué salió mal?

Observaciones generales

Reporte que incluya: Documento de visión con características del sistema priorizadas y objetivos de Framework Ontoconcept relacionados, Entregas del sistema con las características que incluirán, lista de drivers priorizados, lista de escenarios fusionados y priorizados y escenarios más prioritarios detallados.

Se recomienda el uso de las subsiguientes plantillas para establecer un adecuado trabajo con el método propuesto.

**Plantilla para enlistar los Drivers Arquitectónicos y**

**Plantilla para enlistar escenarios (QAW)**

**Drivers Arquitectónicos**

**AC: atributo de Calidad / CU: Caso de Uso / Rest|: Restricción**

| **Tipo de Driver** | **Descripción del Driver** | **Prioridad** |
|---|---|---|
| AC | Disponibilidad:<br>- 24 horas<br>- Si falla menos de 5 minutos recuperarse | |
| AC | Desempeño: 100 usuarios al tiempo | |
| AC | Modificabilidad:<br>- Ampliar a móviles<br>- Integrar con terceros<br>- Redes sociales | |
| Rest | Fecha de entrega: 31 de diciembre | |
| Rest | Navegadores y Dispositivos soportados | |
| Rest | No tener flash ni applets | |

**Plantilla de Lista de Escenarios (QAW)**

| Id | Escenario | Drivers Relacionados | Fusionado | Prioridad |
|---|---|---|---|---|
| | *Nombre del Escenario* | *Atributo de calidad, Restricción y/o Requerimiento Funcional asociados al escenario* | *Escenario con el que se fusiona* | *Importancia del escenario* |
| **1** | Un usuario hace varios intentos de ingreso fallidos y el tercero bloquea la cuenta | Seguridad | | |
| **2** | Domingo, 2 am de31 de diciembre, un usuario ingresa y lo hace exitosamente. | Disponibilidad | | |
| **3** | Un sistema externo envía la información y si la primera vez falla, hace un segundo intento y debe ser exitoso si todos los parámetros están con formato adecuado. | Disponibilidad | | |
| **4** | Un usuario inicia sesión después de haber 99 usuarios logueados y el sistema debe permitirle ingresar. | Desempeño | | |
| **5** | Usando como navegador IE, firefox , o chrome, el sistema muestra las mismas opciones. | Restricción | | |
| **6** | Se deben revertir las transacciones no finalizadas luego de recuperarse de un fallo o apagado. Informar al usuario de este suceso. | Disponibilidad | | |
| **7** | Se realiza mantenimiento al sistema para una versión o | Disponibilidad | | |

| | | | | |
|---|---|---|---|---|
| | corrección de fallo, y debe estar operativa en 5 minutos. | | | |

**Anexo III: Plantilla para especificación de escenarios (QAW)**

| **Escenario crudo:** | *Un usuario inicia sesión después de haber 99 usuarios logueados y el sistema debe permitirle ingresar.* |
|---|---|
| **Objetivos de negocio correspondientes:** | ON-1 CAR-06 |
| **Atributos de calidad relevantes:** | *Desempeño* |
| **Estímulo:** | *Inicio de sesión* |
| **Fuente de estímulo:** | *Usuario* |
| **Entorno:** | *Operación normal del sistema, con 99 usuarios logueados* |
| **Artefacto (si se conoce):** | *El sistema* |
| **Respuesta:** | *Ingreso permitido* |
| **Medida de la respuesta:** | *El 100% de las veces.* |
| **Preguntas:** | *Son solo usuarios o usuarios y administradores.?*<br><br>*Si hay 100 clientes conectados, que pasa si un administrador requiere entrar?* |
| **Problemas:** | *Si hay mucha demanda de parte de usuarios (se sobrepasan los 100)*<br><br>*La infraestructura soporta esta cantidad de usuarios (ej BD)* |

Algunas recomendaciones a tener en cuenta, en el manejo de escenarios:

a) Todo componente del sistema debe ser testeable por cualquier integrante del equipo de pruebas, al complementarse el desarrollo del mismo.

b) El componente debe poseer una interfaz para controlar su comportamiento y su salida debe poder ser observable.

c) Es necesario que se alcance una cobertura del 85% del código del componente dentro de las 3 horas.

d) Los usuarios deben poder minimizar el impacto de los errores cancelando la operación en curso, siendo el tiempo de cancelación menor a 1 segundo.

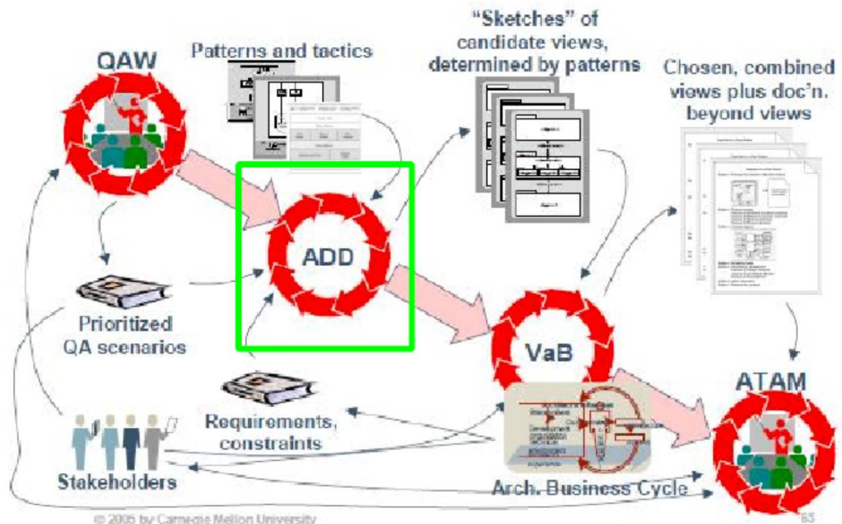
e) Bajo circunstancias normales de operación, el sistema debe mantener información de auditoría sobre los datos que modifique cualquier individuo correctamente identificado.

f) En caso de provocarse un ataque, la imagen correcta de los datos modificados por el usuario debe restaurarse en menos de 1 día.

Bajo condiciones normales de operación, el sistema debe procesar las transacciones de los usuarios con una latencia promedio de 2 segundos.

**El siguiente método es ADD (Attribute Driven Design).**

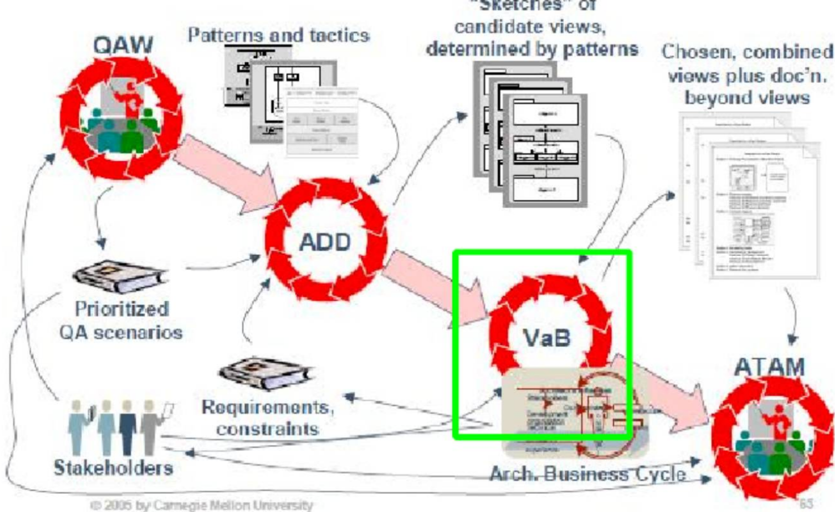
El diseño guiado por atributos (ADD) permite realizar un diseño sistemático de la arquitectura.




En este paso y para la propuesta en curso se recomienda ver el Anexo add.doc, que se encuentra en la carpeta correspondiente.

El tercer método propuesto es VaB (Views and Beyond).

Vistas y más allá (VaB) apoya en la documentación del diseño.




El SEI tiene un enfoque comprobado para documentar la arquitectura de software llamada Views and Beyond.

Estas son algunas publicaciones clave sobre Views and Beyond.

Editorial: Software Engineering Institute

La arquitectura de un sistema de software puede ser su determinante más crucial de éxito o fracaso. Sin una arquitectura adecuada que proporcione la función requerida, así como los atributos de calidad, el proyecto fallará. Pero comunicar una arquitectura a sus grupos de interés es un trabajo tan importante como crearlo en primer lugar. Una arquitectura debe entenderse para que otros -diseñadores de componentes de grano fino, implementadores, probadores, ingenieros de rendimiento, analistas de seguridad, constructores de sistemas de interfaz- puedan construir sistemas a partir de ella, analizarla, mantenerla y aprender de ella.

El SEI tiene un enfoque comprobado para documentar la arquitectura de software llamada Views and Beyond, o V & B. El nombre enfatiza que usamos el concepto de una vista como el principio organizador fundamental para la documentación de la arquitectura. Una vista representa un conjunto de elementos del sistema y las relaciones asociadas a ellos. Las vistas representan las muchas estructuras de sistema que están presentes simultáneamente en los sistemas de software. El principio básico de V & B es que documentar una arquitectura de software implica documentar las vistas relevantes y luego documentar la información que se aplica a más de una vista.

DESAFÍOS

¿Cómo se captura la arquitectura del software para un sistema en un documento que puede servir con éxito a todos los interesados en la arquitectura?

¿Cómo se decide qué vistas arquitectónicas documentar?

¿Qué información registra acerca de una vista arquitectónica más allá del diagrama de cuadro y línea gráfica o "caricatura"?

¿Cómo se especifica la interfaz de software de un elemento arquitectónico? ¿Qué información grabas?

¿Qué información más allá de las vistas debe ser grabada? ¿Qué información se aplica a más de una vista? ¿Cómo se registra la relación entre las vistas?

¿Cómo se especifica el comportamiento de un elemento?

¿Qué notaciones están disponibles para documentar una arquitectura, para documentar una vista, para documentar una interfaz, para documentar el comportamiento?

## DESCRIPCIÓN

V & B es más que un método de documentación de arquitectura. También ayuda al arquitecto a identificar y registrar las decisiones de diseño necesarias durante el desarrollo.

La documentación debe ser el resultado útil de tomar una decisión de arquitectura, no un paso separado en el proceso de arquitectura. Cuanto más se trate la documentación como un seguimiento del diseño, con su propio método separado, es menos probable que se haga.

Documentar una arquitectura de software es una cuestión de documentar las vistas relevantes y luego agregar información que se aplica a todas las vistas. Un primer paso es elegir las vistas relevantes, y esta elección a su vez depende del uso previsto. La documentación se puede utilizar para impulsar el análisis, restringir una implementación, gestionar un proyecto o transmitir una visión general introductoria de un sistema.

La documentación que se aplica a todas las vistas sigue un enfoque simple de cómo y por qué: cómo se organiza la documentación para servir a las partes interesadas, qué es la arquitectura y por qué la arquitectura es como es (es decir, racionalidad del diseño).

V & B funciona tanto en entornos de desarrollo Agile como tradicionales. Es la notación, el lenguaje, el dominio y la tecnología independientes. Y produce documentación que cumple con los estándares ISO / IEC 42010 e IEEE 1471-2000.

## BENEFICIOS

La documentación puede ayudar a una organización de desarrollo de software a evitar las trampas de la documentación inadecuada o incompleta o vaga. Muchos han experimentado la frustración de realizar un gasto masivo de documentación, solo para que los artefactos resultantes acumulen polvo en los estantes. Este es el resultado de la documentación que no se ha planificado adecuadamente y está más orientada a satisfacer estándares que a satisfacer las necesidades de las partes interesadas. El coaching puede evitar esta situación aumentando la documentación y la planificación de la documentación a un nivel acorde con su importancia.

QUIÉN SE BENEFICIARÍA

Los arquitectos de software y las personas a quienes deben comunicar la arquitectura se beneficiarán: diseñadores, implementadores, gerentes técnicos, probadores, analistas, especialistas en calidad y otros. En resumen, cada miembro de un proyecto de desarrollo de software puede ganar.

La documentación del diseño transforma las decisiones de diseño (entradas) en documentos (salidas).

¿Por qué documentar?

La documentación de arquitectura tiene tres propósitos:

– Permite educar, es decir, presentar el sistema a la gente.

– Permite que los involucrados se comuniquen.

– Permite realizar el análisis del sistema.

La documentación de la arquitectura requiere principalmente de la documentación de estructuras creadas en la etapa de diseño.

Estructuras

– De la definición: “La arquitectura de software es la estructura o estructuras del sistema que comprende elementos de software, las propiedades visibles de forma externa de estos elementos, y las relaciones entre los elementos”.

| Estructuras lógicas | Diagrama de componentes |
|---|---|
| – Clases, paquetes. | Diagrama de paquetes y clases |
| Estructuras dinámicas | Diagrama de secuencia |
| – Objetos, hilos, procesos. | |
| Estructuras físicas | Diagrama de implantación |
| – Nodos, Directorios y archivos. | |

## Notaciones

En general las estructuras son representadas por medio de diagramas.

– Es posible usar notaciones “propietarias”

• Sin embargo es, fundamental acompañar los diagramas de una descripción de los elementos gráficos que permita interpretar la representación.

– Es recomendable documentar usando el lenguaje unificado de modelado UML.

• Soportado por herramientas.

• Estándar de facto.

## El concepto de vista

Un diagrama no es suficiente para representar una estructura.

– En general es necesario acompañarlo de información adicional.

Una vista es la representación de un conjunto de elementos del sistema y las relaciones entre ellos, es decir, es la representación de una estructura.

Documentar una arquitectura involucra documentar vistas relevantes y agregar información que aplica a más de una vista.

**Una vista está compuesta por una serie de “paquetes de vista”.**

1. Presentación primaria
2. Catálogo de elementos
   a) Elementos y propiedades
   b) Relaciones y propiedades
   c) Interfaces de elementos
   d) Comportamiento de elementos
3. Diagrama de contexto
4. Guía de variabilidad
5. Antecedentes
   a) Decisiones de diseño
   b) Resultados de análisis
   c) Suposiciones
6. Información adicional
7. Paquetes de vista relacionados

**Tipos de Vista**

Esta componente contiene las vistas de la arquitectura del software. Una vista es una representación de un sistema completo desde la perspectiva de un conjunto relacionado

de preocupaciones [IEEE 1471]. Concretamente, una vista muestra un tipo particular de elementos arquitectónicos de software que ocurren en un sistema, sus propiedades y las relaciones entre ellos. Una vista se ajusta a un punto de vista definitorio.

Las vistas arquitectónicas se pueden dividir en tres grupos, dependiendo de la naturaleza general de los elementos que muestran, que corresponden a los tipos de estructuras:

– Vistas de Módulos (estática)

• Consisten de elementos que son unidades de implementación y asignación de trabajo.

– Vistas de Componente y Conector - C&C (dinámica)

• Consisten de componentes en tiempo de ejecución (unidades de cálculo) y los conectores (canales de comunicación) entre ellos.

– Vistas de Asignación (física)

• Muestran elementos de software y sus relaciones con elementos en entornos externos dentro de los cuales se crea y se ejecuta el software.

• Vistas del módulo. Aquí, los elementos son módulos, que son unidades de implementación. Los módulos representan una forma de considerar el sistema basada en código. Los módulos se asignan áreas de responsabilidad funcional y se asignan a los equipos para su implementación. Hay menos énfasis en cómo el software resultante se manifiesta en tiempo de ejecución. Las estructuras de los módulos nos permiten responder preguntas tales como: ¿Cuál es la responsabilidad funcional principal asignada a cada módulo? ¿Qué otros elementos de software pueden usar un módulo? ¿Qué otro software realmente usa? ¿Qué módulos están relacionados con otros módulos por relaciones de generalización o especialización (es decir, herencia)?

• Vistas de componentes y conectores. Aquí, los elementos son componentes de tiempo de ejecución (que son las unidades principales de computación) y conectores (que son los vehículos de comunicación entre los componentes). Las estructuras de componentes y conectores ayudan a responder preguntas tales como: ¿Cuáles son los principales componentes de ejecución y cómo interactúan? ¿Cuáles son las principales tiendas de datos compartidos? ¿Qué partes del sistema se replican? ¿Cómo progresan los datos a través del sistema? ¿Qué partes del sistema se pueden ejecutar en paralelo? ¿Cómo puede la estructura del sistema cambiar a medida que se ejecuta?

• Vistas de asignación. Estas vistas muestran la relación entre los elementos y elementos de software en uno o más entornos externos en los que se crea y se ejecuta el software. Las estructuras de asignación responden preguntas tales como: ¿En qué procesador se ejecuta cada elemento de software? ¿En qué archivos se almacena cada elemento

durante el desarrollo, las pruebas y la creación del sistema? ¿Cuál es la asignación del elemento de software para los equipos de desarrollo?

Estos tres tipos de estructuras corresponden a los tres tipos generales de decisiones que implica el diseño arquitectónico:

• ¿Cómo se estructura el sistema como un conjunto de unidades de código (módulos)?

• ¿Cómo se estructura el sistema como un conjunto de elementos que tienen comportamiento en tiempo de ejecución (componentes) e interacciones (conectores)?

• ¿Cómo se relaciona el sistema con estructuras ajenas al software en su entorno (como CPU, sistemas de archivos, redes, equipos de desarrollo, entre otros)?

A menudo, una vista muestra información de más de una de estas categorías. Sin embargo, a menos que se elija cuidadosamente, la información en una vista híbrida de este tipo puede ser confusa y poco comprensible.

Las opiniones presentadas en este documento, son las siguientes:

| **Name of view** | **Viewtype that defines this view** | **Types of elements and relations shown** | | **Is this a module view?** | **Is this a component-and-connector view?** | **Is this an allocation view?** |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

**Selección de vistas**

La selección de vistas depende de los involucrados.

– Se deben analizar los involucrados y elegir las vistas pertinentes.

Se debe evitar “documentar por documentar”.

En general se incluye al menos una vista lógica, física y dinámica.

Notas: La documentación es una parte fundamental dentro del desarrollo de arquitecturas.

La documentación de arquitecturas se realiza a través de vistas.

Las vistas se componen de paquetes de vistas relacionados.

Para ampliación del tema se recomienda ver el Anexo Vab.doc.

**El cuarto Método, análisis de intercambio de arquitectura**

El método de análisis de intercambio de arquitectura (ATAM) es un método para evaluar arquitecturas de software relativas a los objetivos de atributos de calidad. Las evaluaciones de ATAM exponen los riesgos arquitectónicos que potencialmente inhiben el logro de los objetivos comerciales de una organización. El ATAM recibe este nombre porque no solo revela cuán bien una arquitectura satisface determinados objetivos de calidad, sino que también proporciona una idea de cómo esos objetivos de calidad interactúan entre sí: cómo se relacionan entre sí.

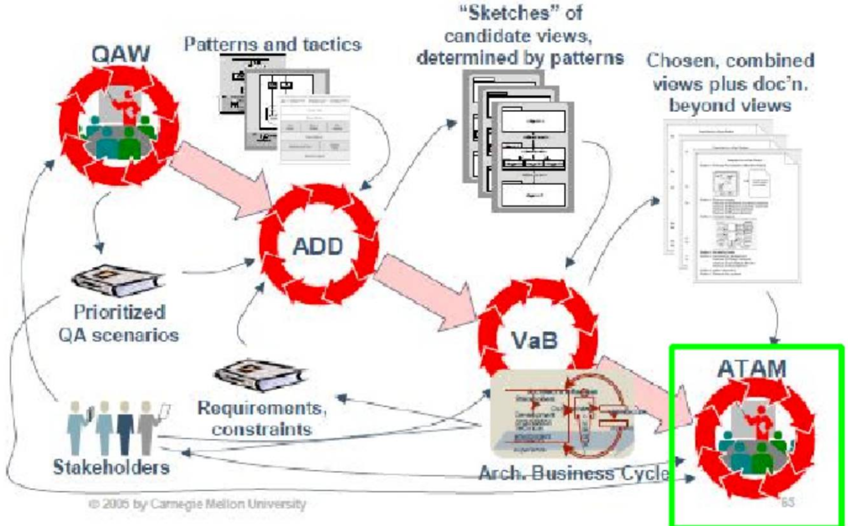
El ATAM es el método líder en el área de evaluación de arquitectura de software. Una evaluación que utiliza el ATAM generalmente demora de tres a cuatro días y reúne un equipo de evaluación capacitado, arquitectos y representantes de los diversos interesados de la arquitectura.

**Desafíos**

Se requiere que la mayoría de los sistemas de software sean modificables y tengan un buen rendimiento. También pueden necesitar ser seguros, interoperables, portátiles y confiables. Pero para cualquier sistema en particular

- ¿Qué significan exactamente estos atributos de calidad como modificabilidad, seguridad, rendimiento y confiabilidad?

- ¿Se puede analizar un sistema para determinar estas cualidades deseadas?
- ¿Qué tan pronto puede ocurrir tal análisis?
- ¿Cómo se puede saber si una arquitectura de software para un sistema es adecuada sin tener que construir primero el sistema?




En las siguientes paginas el enfoque se centra en diversos temas relacionados con el diseño de arquitecturas y los métodos propuestos

– Drivers y QAW.

– Diseño de la AS con ADD y Documentación.

– Aseguramiento de calidad con ATAM.

El enfoque del trabajo es metodológico y no tecnológico.

– Los principios que se promueven son independientes de la tecnología y de la metodología de desarrollo.

Compensación

Decisión de diseño que tiene una influencia positiva en uno o más atributos de calidad, son respuestas que pueden tener una influencia negativa en otro atributo de calidad. Ejemplo: Se toma una decisión de diseño para aumentar el nivel de cifrado de bits para promocionar seguridad (confidencialidad) que a su vez tiene un impacto negativo en el rendimiento (estado latente).

Decisión de diseño que tiene una influencia positiva en una o más cualidades atribuir respuestas y tiene una influencia negativa en otra calidad respuestas de atributos Ejemplo • Se toma una decisión de diseño para aumentar el nivel de cifrado de bits para promocionar seguridad (confidencialidad) que a su vez tiene un impacto negativo en el rendimiento (estado latente)

Categorías de decisiones de diseño
Estas responden a diferentes situaciones del contexto y se encuentran:

**El modelo de coordinación**:

¿Cuáles son los mecanismos de comunicación entre el sistema y entidades externas?
¿Cuáles son los mecanismos de comunicación entre elementos y cuáles son sus propiedades (por ejemplo, acoplamiento síncrono, asíncrono, híbrido)?
¿Cuáles son los mecanismos de comunicación dentro de los elementos?

**El modelo de datos:**

¿Cuál es la estructura -entidades y relaciones, atributos de entidad- en el ¿modelo de datos?
Qué partes del modelo de datos se utilizan por qué elementos de software ¿en qué orden?
¿Cuáles son las reglas de acceso para los elementos de datos?
¿Dónde se crean, modifican y destruyen los elementos de datos?

**Asignación de Funcionalidad:**

¿Cuáles son los principales pasos de procesamiento necesarios para llevar a cabo el trabajo del sistema?
¿Cuál es la división y asignación de funcionalidad a los elementos del software?
¿Cuáles son las abstracciones clave que se pueden utilizar para proporcionar los servicios del sistema? ¿Son los elementos con estado o sin estado?
¿Cuáles son las dependencias de activación y desactivación entre los elementos del software?

**Gestión de recursos de tiempo de ejecución**:

¿Qué estrategias de programación se emplearán?
¿Cuánto saben los elementos del sistema sobre el tiempo?
¿Qué procesos / modelos de hilos serán empleados?
¿Qué recursos deben administrarse y cuáles son sus límites?

T**iempo de enlace de las decisiones en las otras categorías**: las decisiones tomadas para resolver las preguntas en las categorias anteriores pueden estar obligando a que se repitan una variedad de veces, tales como:

- tiempo de diseño (incorporado)
- tiempo de compilación (por ejemplo, modificadores del compilador)
- tiempo de compilación (por ejemplo, reemplazar módulos, seleccionar de la biblioteca)

- tiempo de carga (por ejemplo, bibliotecas de vínculos dinámicos [DLL])
- tiempo de inicialización (por ejemplo, Archivos de recursos)
- tiempo de ejecución (por ejemplo, equilibrio de carga)

Para la siguiente aplicación se sugiere revisar previamente el punto 5.7.1, en el que se menciona: el nombre del método ATAM surge del hecho de que revela la forma en que una arquitectura específica satisface ciertos atributos de calidad, y provee una visión de cómo los atributos de calidad interactúan con otros.

El propósito de ATAM es para evaluar las consecuencias de las decisiones arquitectónicas a la luz de requisitos de atributos de calidad y objetivos comerciales, y continua expresando [46] El ATAM es un método que ayuda a los interesados a formular las preguntas correctas para descubrir decisiones arquitectónicas potencialmente problemáticas Los riesgos descubiertos pueden convertirse en el foco de las actividades de mitigación • por ejemplo, más diseño, más análisis y creación de prototipos Las compensaciones se pueden identificar y documentar explícitamente.

El objetivo de ATAM NO es proporcionar análisis precisos ... el propósito ES descubrir los riesgos creados por las decisiones arquitectónicas evaluadas.

Entre los beneficios de primer nivel que se pueden evidenciar se encuentran:

- riesgos identificados
- requisitos de atributos de calidad clarificados
- documentación de arquitectura mejorada
- base documentada para decisiones arquitectónicas
- mayor comunicación entre los interesados y como resultado final es arquitecturas mejoradas

Flujo conceptual de ATAM

Fuente: Lewis, G. (2010). Basics about cloud computing. *Software Engineering Institute Carnegie Mellon University, Pittsburgh*. Disponible en https://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm

Resumiendo, una arquitectura de software es el artefacto de ciclo de vida más antiguo que incorpora decisiones de diseño significativas, elecciones y compensaciones, las decisiones de diseño se toman en el contexto de la arquitectura con requisitos arquitectónicamente importantes y relevantes, se sugiere identificar patrones y tácticas que apoyan el significado arquitectónico de los requisitos

El ATAM es

- un método para evaluar una arquitectura con respecto a la calidad con múltiples atributos
- una estrategia efectiva para descubrir las consecuencias de la arquitectura decisiones
- un método para identificar tendencias, no para realizar análisis precisos

**Aplicación de los AC en ambientes SOA para Ontoconcept**

A modo de estandarizar y recordar que “La arquitectura del software de un sistema es el conjunto de estructuras necesario razonar sobre el sistema, que comprende software elementos, relaciones entre ellos y propiedades de ambos” [29]

¿Por qué crear o reflexionar en una arquitectura de software?
Porque los atributos de calidad de un sistema pueden predecirse mediante el estudio de su arquitectura, es el caso de la base arquitectural que se propone para Ontoconcept.

Con la puesta de la arquitectura en escenarios, se busca, que ellos en referencia a los atributos de calidad deben ser lo más específicos posible. Un buen escenario deja muy claro qué es el estímulo y qué es lo que se desea. Las respuestas del sistema son, para evitar la ambigüedad.

De los atributos antes mencionados se ha escogido el de DISPONIBILIDAD, para plantear la forma o el modelo de aplicar los demás atributos o No funcionales, planteado como línea base para medir los diferentes escenarios de Ontoconcept.

La disponibilidad se trata de minimizar el tiempo de interrupción del servicio mediante la mitigación de fallas, es mencionado por [29].
Medidas de respuesta típicas

- Intervalo de tiempo cuando el sistema debe estar disponible
- Tiempo de disponibilidad
- Intervalo de tiempo en el que el sistema puede estar en modo degradado
- Tiempo de reparación

La confiabilidad es el grado en que un sistema, producto o componente realiza funciones especificadas bajo condiciones especificadas para un período de tiempo, citado por [11].

Medidas de respuesta típicas

- Tiempo medio entre fallos
- Intervalo de tiempo para la detección de fallas
- Tiempo de recuperación
- Estado del recurso después de la recuperación

Ejemplos de escenarios de disponibilidad planteados para el editor ontológico del framework Ontoconcept

**Escenario A1: Operación después de entrada incorrecta**

**Input:** Un mensaje se recibe durante operación normal del sistema, pero el mensaje esta formateado incorrectamente.
**Output:** El sistema registra el mensaje y continúa operando normalmente sin ningún tiempo de inactividad.

**Escenario A2: Operación degradada durante el mantenimiento**
Se requiere mantenimiento no programado del servidor, en el servidor 'X'. El sistema sigue funcionando en modo degradado durante el tiempo mantenimiento.

Acotaciones sobre la disponibilidad

Las soluciones SOA generalmente se basan en los mecanismos provistos por la ejecución en plataforma o infraestructura

- Detección de fallas: monitor, latido del corazón, marcas de tiempo, etc.
- Recuperación de fallas: redundancia, degradación, reconfiguración, etc.
- Prevención de fallas: desactivación, transacciones, entre otros.

Los acuerdos de nivel de servicio (en inglés *Service Level Agreement* o *SLA*), son un acuerdo escrito entre un proveedor de servicio y su cliente con objeto de fijar el nivel acordado para la calidad de dicho servicio. SLA para servicios típicamente definen los requisitos de disponibilidad como un Porcentaje Anual de tiempo de actividad, como se plantea en el punto 5.5.1.

**Plantilla para el análisis de enfoques arquitectónicos**

| **Análisis de enfoques arquitectónicos** | | | | |
|---|---|---|---|---|
| **Escenario No.** | Escenario: *Texto del escenario del árbol de utilidad* | | | |
| **Atributo(s)** | *Atributos de calidad con los cuales está asociado el escenario* | | | |
| **Entorno** | *Supuestos relevantes sobre el entorno dentro del cual reside el sistema y las condiciones relevantes ante las cuales se lleva a cabo el escenario* | | | |
| **Estímulo** | *Descripción precisa del estímulo del atributo de calidad (ej. Función llamada, fallo, modificación, …) que forma parte del escenario* | | | |
| **Respuesta** | *Descripción precisa de la respuesta del atributo de calidad (ej. Tiempo de respuesta, Medida de la dificultad de modificación)* | | | |
| **Decisiones arquitectónicas** | **Punto de sensibilidad** | **Equilibrio** | **Riesgo** | **No-Riesgo** |
| *Decisiones arquitectónicas relevantes a este escenario que afectan la respuesta del atributo de calidad* | *Punto de sensibilididad #* | *Equilibrio #* | *Riesgo #* | *No-Riesgo #* |
| … | … | … | … | … |
| **Razonamiento** | *Justificación cualitativa y/o cuantitativa sobre por qué la lista de decisiones arquitectónicas contribuyen a satisfacer cada atributo de calidad expresado a través del escenario.* | | | |
| **Diagrama arquitectónico** | *Diagrama o diagramas de vistas anotados con información arquitectónica para soportar el razonamiento, acompañado de texto explicativo, si se considera necesario.* | | | |

**Se muestra ejemplo de aplicación del escenario con Ontoconcept**

| | |
|---|---|
| **Escenario crudo:** | El módulo de edición solicita información de la base, el proceso es exitoso. La solicitud de información a la base, no se logra, el sistema debe reintentar y es exitoso. |
| **Objetivos de negocio correspondientes:** | ON-02 ON-04 |
| **Atributos de calidad relevantes:** | Disponibilidad, Desempeño |
| **Estímulo:** | Solicitud de información |

| Fuente de estímulo: | Módulo de Edicion |
|---|---|
| Entorno: | Sistema en operación |
| Artefacto (si se conoce): | El sistema |
| Respuesta: | Envío exitoso de la información |
| Medida de la respuesta: | 100% de las veces |
| Preguntas: | ¿Si se pierden datos como se puede recuperar el envío de los mismos? |
| Problemas: | Perdida, desacoplamiento del sistema. |

Para un adecuado desarrollo de la evaluación de la base arquitectural de Ontoconcept se propone seguir los siguientes esquemas de medición:

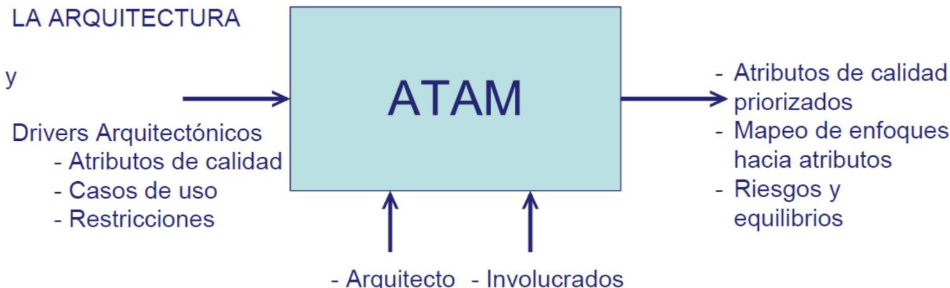
Dada la importancia que tiene la arquitectura en el desarrollo, es indispensable saber si la arquitectura de la que se dispone es adecuada.
– ¿Cómo sabemos si la arquitectura es adecuada?
La arquitectura es adecuada si
– Permite alcanzar los objetivos de los atributos de calidad.
– Es posible construirla con los recursos disponibles.
ATAM toma como entrada información relacionada con el diseño de la arquitectura y produce un análisis de problemas.

La evaluación de arquitectura es una técnica de aseguramiento de calidad enfocada a la identificación de problemas en el diseño de la arquitectura

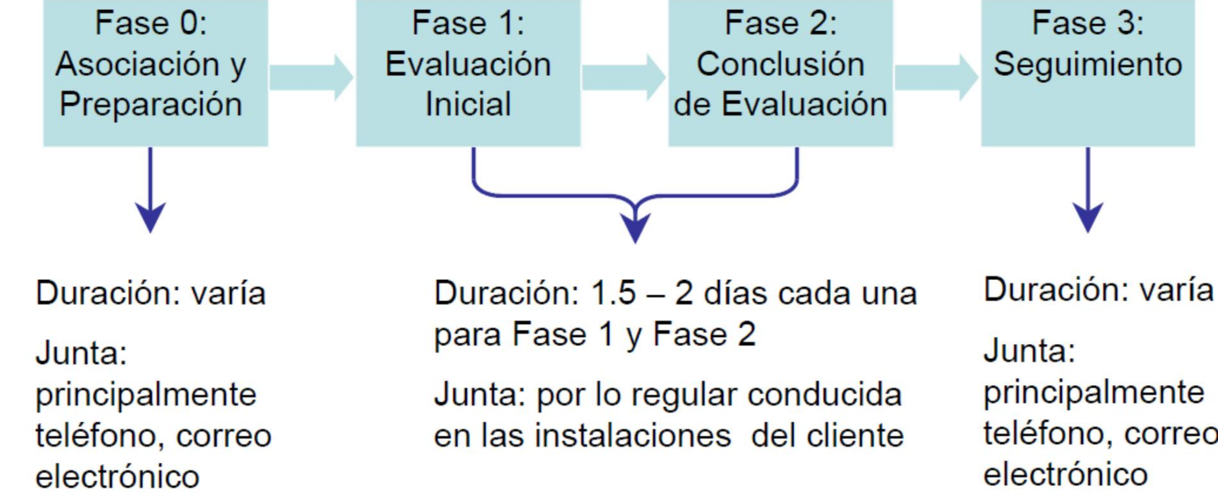
– Permite decir si una arquitectura es apropiada con respecto a un conjunto de objetivos y problemática con respecto a otros objetivos.
– Permite principalmente identificar riesgos con la Arquitectura

El método de evaluación propuesto para Ontoconcept permite realizar una evaluación de la arquitectura basada en escenarios de atributos de calidad.
– Parte del principio que al momento de realizar la evaluación no se conocen todavía cuales son los atributos de calidad del sistema.
– No asume que previamente se realizó QAW, ADD y VaB.

ATAM toma como entrada información relacionada con el diseño de la arquitectura y produce un análisis de problemas como se muestra en la siguiente figura




Las evaluaciones ATAM son conducidas en cuatro fases.




**Roles del ATAM**

Equipo de Evaluación

- – ATAM Lead Evaluator. El moderador del taller, lider de las discusiones.
- – ATAM Evaluator. Apoyo y facilitación del Taller.

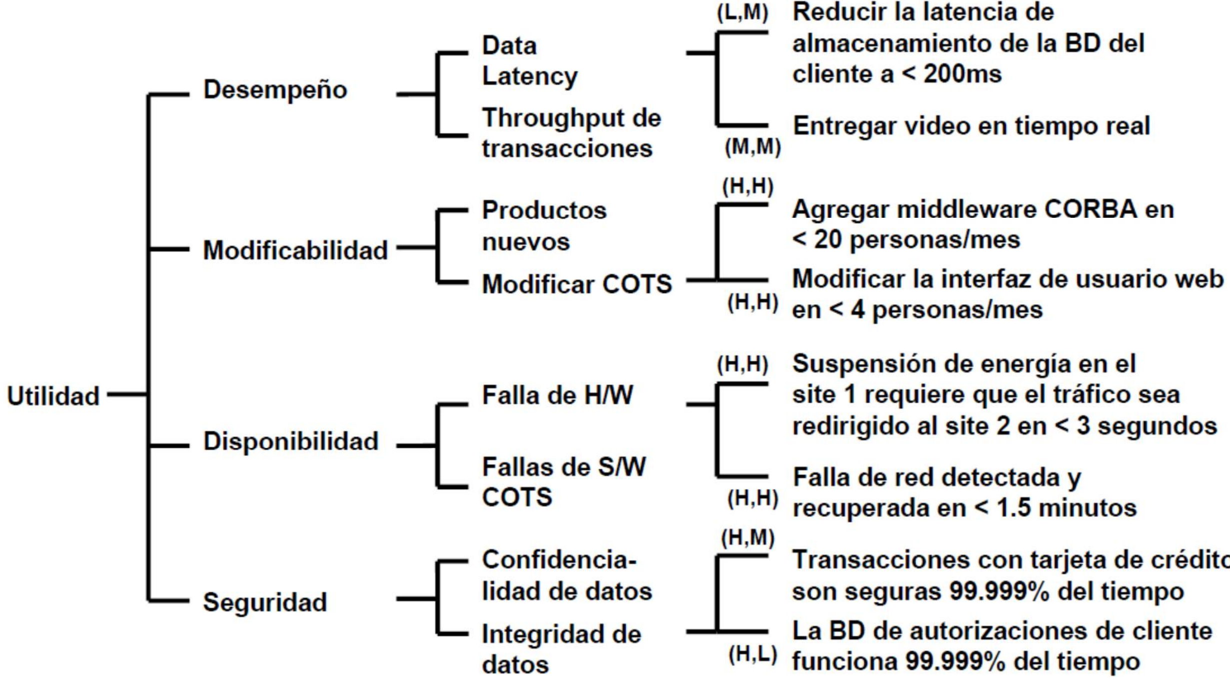
Arquitecto. Presenta y analiza la AS.
Patrocinador o Directivo. Describe las metas de negocio ligadas a la Arquitectura.
Involucrados (Stakeholders). Generan escenarios de análisis

Pasos de ATAM

1. Presentar el ATAM
2. Presentar los drivers del negocio
3. Presentar la arquitectura
4. Identificar los enfoques arquitectónicos
5. Generar el árbol de utilidad de los atributos de calidad
6. Analizar los enfoques arquitectónicos
7. Lluvia de ideas y priorización de escenarios

8. Analizar los enfoques arquitectónicos
9. Presentar los resultados

Como ejemplo del punto 5, "Construcción del árbol de utilidad"




Fuente: Lewis, G. (2010). Basics about cloud computing. *Software engineering institute carniege mellon university, Pittsburgh*. Disponible en https://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm

El propósito es traducir los drivers de negocio en atributos de calidad y priorizarlos.
Los evaluadores identifican, priorizan y refinan los atributos de calidad principales mediante un
árbol de utilidad:
– Se producen escenarios de atributos de calidad.
– La priorización es realizada considerando dos dimensiones: importancia para el éxito del sistema y grado de dificultad en lograr el escenario
• Escenarios (H,H) son candidatos para realizar el análisis
• (M,H) y (H,M) vienen después y, si hay tiempo, los (M,M)

**Como ejemplo del punto 6 se presenta la siguiente**
Plantilla para el análisis de enfoques arquitectónicos

| **Análisis de enfoques arquitectónicos** | |
|---|---|
| **Escenario No.** | Escenario: *Texto del escenario del árbol de utilidad* |
| **Atributo(s)** | *Atributos de calidad con los cuales está asociado el escenario* |

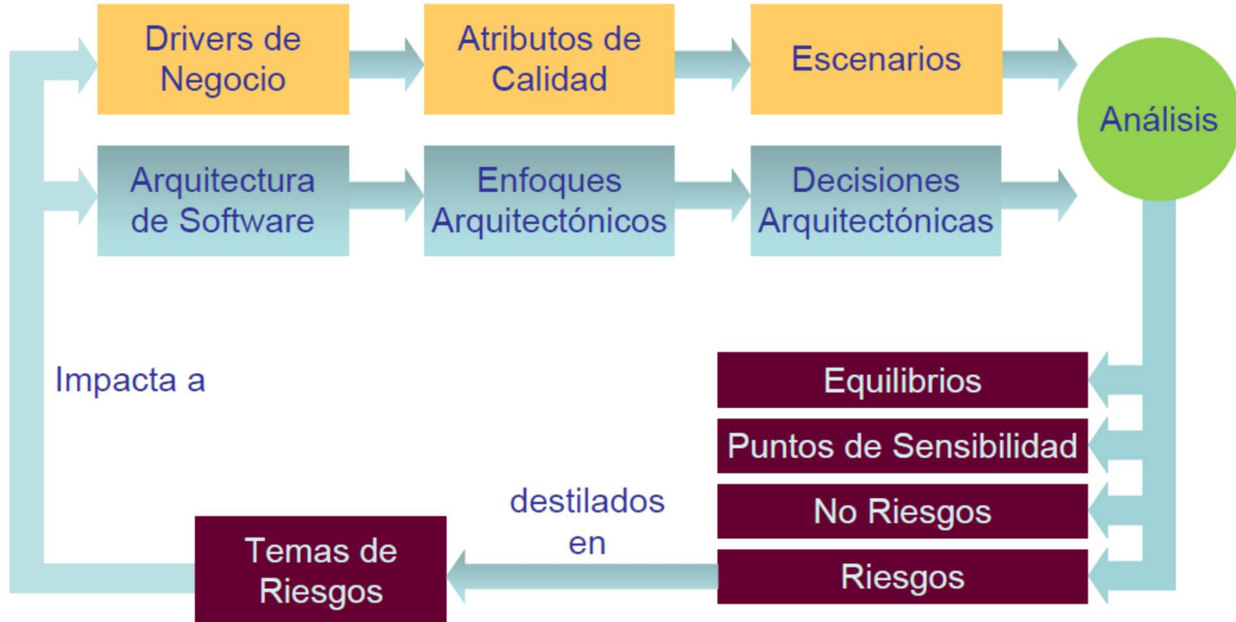
| **Entorno** | *Supuestos relevantes sobre el entorno dentro del cual reside el sistema, y las condiciones relevantes ante las cuales se lleva a cabo el escenario* | | | |
|---|---|---|---|---|
| **Estímulo** | *Descripción precisa del estímulo del atributo de calidad (ej. Función llamada, fallo, modificación, …) que forma parte del escenario* | | | |
| **Respuesta** | *Descripción precisa de la respuesta del atributo de calidad (ej. Tiempo de respuesta, Medida de la dificultad de modificación)* | | | |
| **Decisiones arquitectónicas** | **Punto de sensibilidad** | **Equilibrio** | **Riesgo** | **No-Riesgo** |
| *Decisiones arquitectónicas relevantes a este escenario que afectan la respuesta del atributo de calidad* | *Punto de sensibilidad #* | *Equilibrio #* | *Riesgo #* | *No-Riesgo #* |
| … | … | … | … | … |
| **Razonamiento** | *Justificación cualitativa y/o cuantitativa sobre por qué la lista de decisiones arquitectónicas contribuyen a satisfacer cada atributo de calidad expresado a través del escenario* | | | |
| **Diagrama arquitectónico** | *Diagrama o diagramas de vistas anotados con información arquitectónica para soportar el razonamiento, acompañado de texto explicativo, si se considera necesario.* | | | |

Como ejemplo del Paso 9: Presentar resultados

El propósito es dar retroalimentación a los involucrados.

En esta presentación el líder de evaluación recapitula lo realizado durante la evaluación y presenta:
– Los enfoques arquitectónicos documentados.
– El conjunto de escenarios y su priorización.
– El árbol de utilidad.
– Los RIESGOS y no riesgos descubiertos.
– Los puntos de sensibilidad y equilibrios encontrados.

En resumen, el proceso de la aplicación de ATAM se puede establecer según la figura siguiente:




Notas: Los resultados más importantes son arquitecturas mejoradas. El resultado de un ATAM es una presentación informativa y / o un informe escrito que incluye los principales hallazgos de la evaluación. Estos son típicamente

- un conjunto de enfoques arquitectónicos identificados
- un "árbol de utilidad": un modelo jerárquico de los requisitos arquitectónicos de conducción
- el conjunto de escenarios generados y el subconjunto que fueron mapeados en la arquitectura
- un conjunto de preguntas específicas de atributos de calidad que se aplicaron a la arquitectura y las respuestas a estas preguntas
- un conjunto de riesgos identificados
- un conjunto de no riesgos identificados
- una síntesis de los riesgos en un conjunto de temas de riesgo que amenazan con socavar los objetivos comerciales del sistema

# CAPITULO 6. COMPARATIVA DE FRAMEWORKS

En capítulo 2 se hace referencias con más detalle a los tópicos usados en este, se establece con los conceptos descritos recordar el mismo, no se hace con el ánimo de repetirlos.

### Framework lógicos

Un framework lógico es un metalenguaje para la especificación e implementación de sistemas deductivos, que se utilizan de forma generalizada en la lógica y la teoría de los lenguajes de programación. Básicamente son herramientas para gestionar el ciclo de vida de una ontología o una de las fases que lo componen.

Un framework lógico debe ser tan simple y uniforme como sea posible, pero proporcionar medios intrínsecos para representar conceptos y operaciones comunes en su ámbito de aplicación.

## 6.1 Algunos framework lógicos que existen en la actualidad son:

### Ontoconcept:

Se desarrolló la herramienta Ontoconcept como parte del marco de referencia para el modelado conceptual del cambio ontológico, la cual implementa una arquitectura cliente-servidor dos capas soportado en plugins, y utiliza el modelo conceptual basado en el Grafárbol para procesar las sentencias del lenguaje declarativo. Una de las características del marco de referencia Ontoconcept es la capacidad para controlar el ciclo de vida de una ontología desde su creación. Esta característica, facilita la reconstrucción de la ontología a partir del log de cambios.

### Protegé:

Protege es un editor y framework ontológico libre y de código abierto para la construcción de sistemas inteligentes. Protege fue diseñado y desarrollado inicialmente en la Universidad de Manchester. Protege utiliza la API de OWL para sustentar todas las tareas de gestión ontológica, desde carga y guardado de ontologías, hasta la manipulación de ontologías durante la edición, para interactuar y ofrecer una opción de razonadores OWL.

### NeOn Toolkit

Es una plataforma de ingeniería ontológica desarrollada por el EC-FundedNeon Project. Está basada en la framework Eclipse y soporta modelado de frame como ontologías extendidas por normas y modelado de DLontologías. Como plataforma es un entorno extensible, para el cual los socios de Neon están contribuyendo funcionalidad en forma de plug-ins.

En el marco del proyecto NeOn, se ha desarrollado la herramienta NeOn Toolkit, caracterizada por tener soporte colaborativo y de gestión de cambios. NeOnToolkit es un entorno de ingeniería ontológica de estado del arte, código abierto y multiplataforma, el cual proporciona apoyo completo para el ciclo de vida de ingeniería ontológica.
La herramienta está basada en la plataforma de eclipse, un ambiente de desarrollo que conduce y proporciona un amplio conjunto de plugins, que cubre una variedad de actividades de ingeniería de ontologías.

En la documentación ofrecida en la página oficial de NeOn Toolkit, El objetivo de NeOn Toolkit es mejorar la capacidad de manejar múltiples ontologías en red que existen en un contexto particular, creadas en colaboración y que podrían ser altamente dinámicos y en constante evolución.

El editor OWL de NeOn Toolkit es una herramienta de modelado para la creación y mantenimiento de modelos semánticos (a menudo denominado “Ontologías”)

**Comparativos entre FW y FL**

Se busca establecer diferencias básicas entre los framework mencionados, con la intención de referenciar al lector en posibles puntos de vista que desconoce de cada tipo.

| **framework web** | **frameworks lógicos** |
|---|---|
| Orientado a un modelo llamado vista-controlador (MVC). | Utiliza niveles de objetivos. |
| Son herramientas que usan elementos y códigos reutilizables para poder desarrollar servicios web. | Son conceptos entrelazados que se usan juntos de manera dinámica para desarrollar un proyecto bien diseñado. |
| Abstracción de URL y secciones. | Deben reflejar el interés real del proyecto dándole validez a los aspectos críticos |
| Acceso a datos, ya que tiene herramientas para acceder a fuentes que tengan datos como bases de datos | Son razonables, o sea que deben estar relacionados y representar el núcleo del proyecto, para así darle mayor validez y valor |
| Controladores que se usan para el manejo de eventos. | Tienen que estar enmarcados en cantidad y calidad con los siguientes pasos: Identificar indicador, cuantificar, establecer la calidad y especificar marco de tiempo |
| Autenticación con mecanismos para identificar a los usuarios. | Son independientes, es decir que son enfocados a demostrar un logro en una parte del proyecto. |

Adicional, se plantean unas características propuestas por [39] que muestran algunas situaciones ideales en el uso de los framework lógicos, se aclara que no son todas las deseables, y son sólo para evidenciar puntos de atención sobre los mismos.

# CAPITULO 7. CONCLUSIONES, RECOMENDACIONES Y TRABAJO FUTURO

## 7.1 CONCLUSIONES

Las principales conclusiones que se han alcanzado con el trabajo son:

1. El estilo arquitectural SOA, es completo y suficiente para describir el framework Ontoconcept.
2. UML 2.0 como ADL, tiene la capacidad expresiva para representar las características estructurales y de comportamiento para representar la arquitectura propuesta.
3. El framework completo requiere hacer convivir estilos heterogéneos, dado que sus funcionalidades modulares así lo exigen.
4. Se necesita profundizar y alcanzar mayor rigurosidad en la especificación arquitectural.

## 7.2 RECOMENDACIONES

Se proponen las siguientes:

1. Realizar una extensión de la línea base arquitectural planteada, para que cubra los aspectos de despliegue sobre una plataforma tecnológica compatible con las tecnologías CLOUD que se han popularizado.
2. Popularizar el uso del framework entre los estudiantes y universidades, para que se utilice y familiarice el estudiante de pregrado con el uso de ontologías en los proyectos tecnológicos que se emprendan.

## 7.3 TRABAJOS FUTUROS

1. Se requiere extender el trabajo adelantado para la definición de los componentes de búsqueda, publicación y consulta del framework OntoConcept

2. Realizar una comparación de las funcionalidades ofrecidas por los principales frameworks ontológicos y confrontarlos a las funcionalidades presentadas por Ontoconcept.
3. Aceptar que hay otros factores a considerar al elegir características como las planteadas, y determinar que siempre hay aspectos que son relevantes en cada caso.
4. Aspectos de seguridad: Debido a los pocos mecanismos de seguridad informática en general asociados a Internet, el hecho de colocar una página Web en ella trae determinados riesgos que pueden ser controlados con alguna estrategia adecuada.

La seguridad en una página debe cumplir con tres aspectos: operatividad, integridad y privacidad, al igual que cualquier otro dispositivo de cómputo.

Cada una de las condiciones de seguridad de una página Web: operatividad, integridad y privacidad, tienen sus propios riesgos que actúan dentro o fuera del sistema.

Para establecer una estrategia de seguridad se debe evaluar el grado de importancia de cada uno de ellos y basándose en el establecer normas y procedimientos necesarios para contenerlos:

Para muchas aplicaciones de negocios, como la publicidad y promociones simples, es probable que no se necesite tratar con precauciones de seguridad. Pero si se permite que los usuarios tengan acceso a datos delicados, se deberán tomar medidas para proteger a los datos. Debido a que cada vez son más las personas que desean transferir documentos e información de tarjetas de crédito o cualquier tipo de transmisión de datos en forma segura y sin el temor a los crackers y piratas.

Las medidas de seguridad básicas a tener en cuenta son:

1. La encriptación de Datos
2. Firma Digital
3. Creación de un Sitio Seguro
4. Firewall s, Wrappers y Proxies

## 8. BIBLIOGRAFÍA

[1] Chavarro, Julio Cesar. (2010). *En Marco de Referencia Para la Gestión del Cambio en Ontologías Basados en Modelos Conceptuales* (Doctoral dissertation, Tesis Doctoral. Cali–Colombia. Universidad del Valle).

[1b] Chavarro, Julio Cesar. (2010). *Implementación Tecnológica En Marco de Referencia Para la Gestión del Cambio en Ontologías Basados en Modelos Conceptuales* (Doctoral dissertation, Tesis Doctoral. Cali–Colombia. Universidad del Valle).

[2] Clements, P. C. (1996). "A survey of architecture description languages". *Proceedings of the 8th International Workshop on Software Specification and Design*. pp. 16–00. ISBN 0-8186-7361-3. doi:10.1109/IWSSD.1996.501143, Alemania.

[3] Bernardo-Quintero, J., & Duitama-Muñoz, J. F. (2011). Reflexiones acerca de la adopción de enfoques centrados en modelos en el desarrollo de software. *Ingeniería y Universidad*, *15*(1).

[4] Kazman, R., Abowd, G., Bass, L., & Clements, P. (2003). Scenario-based analysis of software architecture. *IEEE software*, *13*(6), 47-55. “Software Architecture in Practice. Second Edition.

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

[6] Medvidovic, N., & Taylor, R. N. (2010, May). Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2* (pp. 471-472). ACM.

[7] Valencia, A. M., & Ferro González, M. (2011). *Documentación y análisis crítico de algunas arquitecturas de software en aplicaciones empresariales* (Doctoral dissertation, Universidad Tecnológica de Pereira).

[8] Reynoso, C., & Kicillof, N. (2004). Estilos y Patrones en la Estrategia de Arquitectura de Microsoft. *Buenos Aires: Universidad de Buenos Aires*.

[9] Navarčik, M. Using UML with OCL as ADL. In *IIT. SRC 2005: Student Research Conference* (p. 175). Faculty of Informatics and Information Technologies. M. Bieliková (Ed.), IIT.SRC.
[10] Fielding, R. T., & Taylor, R. N. (2000). *Architectural styles and the design of network-based software architectures* (p. 151). Doctoral dissertation: University of California, Irvine..

[11] Medvidovic, N., & Taylor, R. N. (2010, May). Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2* (pp. 471-472). ACM.

[12] Garlan, D., Monroe, R. T., & Wile, D. (2000). Acme: Architectural description of component-based systems. *Foundations of component-based systems*, *68*, 47-68.

[13] Briand, L. C., Labiche, Y., & Cui, J. (2005). Automated support for deriving test requirements from UML statecharts. *Software & Systems Modeling*, *4*(4), 399-423.

[14] Smith, B., & Welty, C. (2001, October). Ontology: Towards a new synthesis. In *Formal Ontology in Information Systems* (Vol. 10, No. 3, pp. 3-9). ACM Press, USA, pp. iii-x.

[15] Gruber, T. R. (1995). A translation approach to portable ontology specifications. *Knowledge acquisition*, *5*(2), 199-220.

[16] Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, *43*(5-6), 907-928.

[17] Zúñiga, G. L. (2001, October). Ontology: its transformation from philosophy to information systems. In *Proceedings of the international conference on Formal Ontology in Information Systems-Volume 2001* (pp. 187-197). ACM.

[18] Studer, R., Benjamins, V. R., & Fensel, D. (1998). Knowledge engineering: principles and methods. *Data & knowledge engineering*, *25*(1-2), 161-197.

[19] Studer, R., Benjamins, V. R., & Fensel, D. (1998). Knowledge engineering: principles and methods. *Data & knowledge engineering*, *25*(1-2), 161-197.

[20] [citado 2017/10/23] Disponible en http://neon-toolkit.org/wiki/Main_Page.html.

[21] OWL Editor. Homepage of NeOn Toolkit. Media:Doku_2.3.pdf - Documentation of NTK 2.3 (pre) release citado 2017/10/23] Disponible en: http://neon-toolkit.org/wiki/Documentation_and_Support.html

[22] Lucchi, R., Millot, M., & Elfers, C. (2008). Resource oriented architecture and REST. *Assessment of impact and advantages on INSPIRE, Ispra: European Communities*.

[23] Montejano, G. A., Testa, O., Garcia, P., Bast, S. G., & Dieste, O. (2013, June). Definición formal de una metodología para la generación de sistemas de software orientados a servicios. In *XV Workshop de Investigadores en Ciencias de la Computación*.

[24] Forero, J. (2013). Consideraciones para una Arquitectura Interoperable de Repositorios de Objetos de Aprendizaje. *Escuela de Posgrado. Facultad Regional de Buenos Aires. Universidad Tecnológica Nacional*.

[25] Ramírez, J. (2009). Arquitectura Orientada a Servicios. Disponible en: http://www.clubinvestigacioncr.com/docs/informesoa.pdf

[26] -- Mauricio Rojas Contreras. ARQUITECTURA DE SOFTWARE PARA INTEGRAR OBJETOS DE APRENDIZAJE BASADA EN SERVICIOS WEB. Disponible en: http://www.unipamplona.edu.co/unipamplona/portalIG/home_40/recursos/03_v13_18/revista_18/03122011/12.pdf

[27] Thomas Erl, Service-Oriented Architecture. Concepts, Technology, and Design. First Printing. Pearson Education, Inc. Rights and Contracts Department One Lake Street. 2005.

[28] Luis Moreno Sánchez. Utilización de Patrones para la Construcción de Arquitecturas Orientadas a Servicios. Tesis de fin de Experto. Universidad Pontificia de Salamanca. 2007.

[29] Fielding, R.T., Architectural Styles and the Design of Network-based Software Architectures, Ph.D. dissertation, in University of California, Irvine. 2000.

[30] Leonard Richardson, Sam Ruby. RESTful Web Services. Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

[31] Popkin Software and Systems. Modelado de Sistemas con UML. Disponible en http://es.tldp.org/Tutoriales/doc-modelado-sistemas-UML/doc-modelado-sistemas-uml.pdf.

[32] Cristian Wilckens, Poder Expresivo de UML 2.0 para especificar arquitecturas de Software [Diapositivas].

[33] Oton, Salvador., Ortiz, A., Hilera, J., *SROA: Sistema de reutilización de objetos de aprendizaje*. ISSN: 1699-4574. Revista Iberoamericana de Informática Educativa. Vol. 5: 7-22. (2007).

[34] Morales, M. I. M. (2011). Acercamiento al desarrollo de arquitecturas orientadas a servicios (SOA) en el dominio de los sistemas de información geográfica. *CUADERNO ACTIVA*, *3*(3), 45-55.

[35] Arlandy, R. M. (2014) SOA vs. SOAP y REST. Disponible en https://www.adictosaltrabajo.com/tutoriales/soavs-soap-rest/

[36] – Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., & Stafford, R. (2003). Catalog of patterns of enterprise application architecture. *URL: http://martinfowler. com/eaaCatalog/index. html*.

[37] Cruz, D., Fontana, J., Rivadeneira Molina, S., & Vilanova, G. (2013, June). Un acercamiento en la integración entre BPMN y SOA. In *XV Workshop de Investigadores en Ciencias de la Computación*.

[38] – Puertas, E., & Vasquez, G. Método de Integración empresarial Orientada a Servicios: Pequeñas y Medianas Empresas. Disponible en: https://www.researchgate.net/profile/Edwin_Puertas2/publication/274963369_Metodo_de_integracion_empresarial_orientada_a_servicios_pequenas_y_medianas_empresas/links/552de0fd0cf21acb092190b6/Metodo-de-integracion-empresarial-orientada-a-servicios-pequenas-y-medianas-empresas.pdf.

[39] Reynoso, C. B. (2004). Introducción a la Arquitectura de Software. *Universidad de Buenos Aires*, *33*.

[40] Diao, L., & Ma, Y. (2008). A Versatile SOA-based E-Business Platform. 2008 International Symposium on Electronic Commerce and Security, 638-641. IEEE. DOI: 0.1109/ISECS.2008.49.

[41] Beydeda, S., & Book, M. (2005). *Model-driven software development* (Vol. 15). V. Gruhn (Ed.). Heidelberg: Springer. Página 19.

[42] Mahmoud, Q. (2005). Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI), Sun Java site.

[43] Salazar Parra, D. (2015). Línea base arquitectural del Framework Ontoconcept (Doctoral dissertation, Universidad Tecnológica de Pereira).

[44] Reynoso, C., & Kicillof, N. (2010). De Lenguajes de descripción arquitectónica de software ADL.

[45] Pérez, J., & Quintero, J. B. (2009). Estrategias para la Definición de una Técnica de Modelado para Arquitecturas de Referencia. In CIbSE (pp. 127-132).