Universidad de las Ciencias Informáticas
Facultad 1

Propuesta de un Lenguaje de Descripción para SOA.

Trabajo de Diploma

Para optar por el título de Ingeniero en Ciencias Informáticas

Autor:

Raymond Abreu Maceo
Lisandra Arevalo Pérez

Tutor:

Ing. Noel Rondán Domínguez

Ciudad Habana, Junio 2008

DEDICATORIA

A mis padres Ramón y Ana, las personas que desde el 25 de Agosto de 1984 no han dejado de vivir para mí, ni de estar conmigo.
Mi segunda madre Alba, que también ha sido parte de mi triunfo.
Mis hermanos Raydel, Michel y Joaquín, que por tratar de servirles de ejemplo, han sido motivo para superarme cada día más.
Mi familia, en especial a mi tíos Nelson y Milca, y mis primos: Laurent, Nelsy, Betty y Nelsitin.
Yanitza, que me ha acompañado estos dos últimos años, y ha vivido lo malo y lo bueno conmigo en ese tiempo.
Dasiel (La Manta), Suha (El Ecoyo) y Noichel, más que compañeros, hermanos…
Tito, Melga…puedo llegar hasta 100.
Yoel que cuando lo necesitaba, estaba disponible…
Otros que no están cerca de mí físicamente, pero están: Pepe, Yorge, Robe, Chan, Alberto, Armando (Letal), Augusto (El Vietnam)…
También está dedicado a todas las personas que han confiado en mí, que constantemente han preguntado por mi trabajo, que estaban interesados en mis ideas…

Raymond

Quisiera dedicarle esta tesis a mi mamá Virginia Pérez Brisuela por ser mi apoyo, mi guía y mi motivo de inspiración en todos estos años.
También se la dedico a mi abuela Aracelis, mi padrastro Tomas, mi tía Yaquelín y su esposo Ernesto, que siempre dieron lo mejor para que estos años en la universidad fueran más fáciles para mí.

Lisandra

AGRADECIMIENTOS

A mis "padres" y hermanos por el apoyo durante mis años en la UCI y en la vida. Yanitza por compartir conmigo…y ayudarme a salir adelante. También su familia: Mireydis, Marcia e Israel.

Una persona que dentro de los límites de la UCI, fue mi padre, hermano, amigo y compañero, sus consejos, ejemplos y ayuda no faltaron: Yoel Catalá.

Mi tutor Noel por guiarme en el trabajo y apoyarme.

Lisandra por acompañarme en este trabajo y representar tenacidad.

A Yanny que empezó conmigo el trabajo, pero lamentablemente no pudo seguir por otras razones.

Todos los que han compartido en algún momento algo conmigo: Apto, Aula, Laboratorio.

Finalmente a aquellas personas que de alguna forma me hicieron pasar por momentos malos, porque permitieron hacerme más fuerte.

Raymond

Antes de todo quiero agradecerle 70 veces 7 a Jesús, por su apoyo y amor incondicional, por demostrarme cada día que no todo esta perdido y siempre hay una razón por seguir adelante.

A mi mamá:

Por estar para mí en los momentos más difíciles y ser motivo para no detenerme aun cuando todo parece imposible. No puedo dejar de agradecerle, enseñarme a ser como soy.

A mi familia:

Por brindar siempre lo mejor y formar parte de todos mis éxitos y fracasos.

A mis amigos:

Coti, Yeny, Eddy, Yaima, Eylen, Betty, Yeilyn, Yen, Labrada, Iyugnis, Carlos, y todos aquellos que aunque no los he mencionado, demostraron ser amigos en todo momento y permitieron que sintiera que la UCI era mi casa. A mi tutor Noel por su tiempo dedicado a este trabajo y a Raymond mi compañero de tesis por su paciencia y apoyo.

A Fidel Castro Ruz y la Revolución que gracias a ellos la UCI fue el proyecto que hizo mi sueño realidad.

Lisandra

DECLARACIÓN DE AUTORÍA

Declaramos que somos los únicos autores de este trabajo y autorizamos a la Facultad 1 de la Universidad de la Ciencias Informáticas a hacer uso del mismo en su beneficio.

Para que así conste firmamos la presente a los ____días del mes de _______________del año________.

Raymond Abreu Maceo	Lisandra Arevalo Pérez	Noel Rondán Domínguez
______________________	______________________	______________________
Autor	**Autor**	**Tutor**

DATOS DE CONTACTO

Ing. Noel Rondán Domínguez

Profesor de la Universidad de las Ciencias Informáticas.

Años de experiencia: 2

Correo electrónico: nrondan@uci.cu

RESUMEN

El presente trabajo se encamina a la realización de un estudio acerca de los Lenguajes de Descripción de Arquitectura (en adelante, ADLs), que existen actualmente en la industria, recorren una cifra aproximada de 20, sin embargo algunos no responden a un correcto funcionamiento y otros no constituyen un ADL en sentido estricto, aunque la literatura de referencia acostumbra a tratarlos como tal, estos aspectos también serán tratados.

Este compromiso surge de la necesidad de alcanzar una mejora en la arquitectura de los proyectos de la Universidad de las Ciencias Informáticas (UCI) y de interrelacionar unidades funcionales diferentes de una aplicación para que los servicios montados en una diversidad de sistemas interactúen entre sí y de carácter universal. Para esto se necesita tener un lenguaje que sea capaz de describir una Arquitectura Orientada a Servicios (en adelante, SOA) y que además se ajuste a las características propias del entorno productivo (Facultad 1), que especifique de qué manera los sistemas se combinan para formar configuraciones y definir familias de arquitecturas o estilos.

El objetivo principal es obtener el ADL, que corresponda con las condiciones que existen en los proyectos de la Facultad 1, luego de comprobar que este ADL cumple con las condiciones requeridas y proponerlo a nivel de universidad.

Los aspectos esenciales a medir:

- Disponibilidad de plataforma.
- Herramientas gráficas afines.
- Capacidades para generar códigos ejecutables.

PALABRAS CLAVES: ADL, Modelado, Arquitectura de Software.

TABLAS Y FIGURAS

Anexo 1. Ambiente de edición de AcmeStudio con diagrama de tuberías y filtros. 64
Anexo 2. Ambiente gráfico de Aesop con diagrama de tuberías y filtros. 65
Anexo 3. Diagrama de tuberías en Darwin. 65
Anexo 4. Tipos de componentes y conectores. 66
Anexo 5. Estudio de caso en Jacal. 66
Anexo 6. Diagrama correspondiente al código en Wright. 67
Anexo 7. Comparación entre ADLs. 67

ÍNDICE

DEDICATORIA I
AGRADECIMIENTOS II
DECLARACIÓN DE AUTORÍA III
DATOS DE CONTACTO IV
RESUMEN V
TABLAS Y FIGURAS VI

Introducción 1
CAPITULO 1 6
FUNDAMENTACIÓN TEÓRICA 6
1.1 Introducción 6
1.2 Conceptos asociados al dominio del problema. 6
1.2.1 ¿Qué es arquitectura de software? 6
1.2.2 ¿Qué es SOA? 7
1.3 Objeto de estudio. 8
1.3.1 ¿Qué es un ADL? 8
1.3.2 Descripción general. 9
1.3.3 Descripción actual del dominio del problema (Facultad 1) 17
1.4 Los ADLs y la Arquitectura de Software. 22
Conclusiones 23

CAPÍTULO 2 24
ADLS MÁS CONOCIDOS 24
2.1 Introducción 24
2.2 Lenguajes de Descripción de Arquitectura. 24
2.2.1 Acme-Armani 24
2.2.2 Aesop 28
2.2.3 Darwin 29
2.2.4 Jacal 32
2.2.5 Rapide 35
2.2.6 Wright 37

2.2.7 UML 40
2.2.8 Comparación de los ADLs. 41
Conclusiones.. 41

CAPÍTULO 3 42
SOLUCIÓN PROPUESTA 42
3.1 Introducción.. 42
3.2 UML 2.0 42
3.2.1 Elementos deseables en un ADL.. 43
3.2.2 ¿Qué no puede UML? 54
Conclusiones.. 55

CONCLUSIONES 56
RECOMENDACIONES 57
REFERENCIAS BIBLIOGRÁFICAS.. 58
BIBLIOGRAFÍAS 61
ANEXOS 63
GLOSARIO 68

Introducción

La informatización es un proceso progresivo e imparable en todos los sectores de la sociedad, no es más que la utilización ordenada y masiva de las Tecnologías de la Información y las Comunicaciones (TICs) en la vida cotidiana, para satisfacer las necesidades de todas las esferas de la sociedad, en su esfuerzo por lograr cada vez más eficacia y eficiencia en todos los procesos y por consiguiente mayor generación de riqueza y aumento en la calidad de vida de los ciudadanos.

Cuba ha identificado desde muy temprano la conveniencia y necesidad de dominar e introducir en la práctica social estas tecnologías y lograr una cultura digital como una de las características imprescindibles del hombre nuevo, lo que facilitaría a nuestra sociedad acercarse más hacia el objetivo de un desarrollo sostenible.

La UCI juega un papel importante en el desarrollo de la informatización de la sociedad cubana y en el desarrollo de la Industria Cubana del Software, se centran los programas de informatización de sectores fundamentales del país: Salud, Educación, Deporte, Cultura, Turismo, Prensa y Software Libre. Además en la UCI se coordinan y desarrollan los proyectos estratégicos de exportación asociados a la Industria del Software y se trabaja en las necesidades propias de la ciudad universitaria. La estructura de la UCI define 10 facultades, cada una atiende distintos sectores.

En estos momentos en la UCI, debido a los innumerables procesos de negocios que se manejan y que van en aumento, además por el incremento del alcance en el diseño y niveles de complejidad de los sistemas de información, se necesita realizar cambios en la arquitectura establecida, por una que sea capaz de satisfacer requerimientos que la actual estructura no cumple. Para esto es necesario orientar la arquitectura hacia los procesos que generalmente se realizan, es decir, aplicar una SOA, la cual es capaz de adaptarse a los continuos cambios que se realizan en la universidad. En este sentido en la UCI han surgido diferentes iniciativas encaminadas a poner en práctica las ventajas que proporciona esta arquitectura. Una de ellas ha sido la idea de establecer SOA en algunos proyectos de la Facultad 1.

Para implantar una arquitectura de tal dimensión e importancia, además de que se considera un suceso nuevo en nuestro país, se necesita contar con un elemento esencial, que juega un papel decisivo en materia de precisión para un arquitecto de software: los ADLs.

Actualmente en los proyectos productivos de la facultad 1 no se manejan los ADLs para ningún propósito, peor, no se tiene conocimiento de lo que es un ADL. Por tanto, no se conoce la medida en que éste pudiera influir sobre la arquitectura. Lo que se utiliza son documentos descriptivos que contienen diferentes vistas de la arquitectura.
Todo esto conduce, a la necesidad de que todos los integrantes de los proyectos logren conocer que son los ADLs, que función realizan, importancia de contar con uno de ellos, tener claro en que momento darle paso a sus funciones, entre otras cosas de vital importancia. Claro está, se recomienda tener cuidado cuando se habla de ADL o cuando se define cuál usar, un sistema en dependencia de la complejidad que tenga puede implicar diferentes mecanismos de modelado.

Este trabajo está encaminado al estudio de los ADLs más usados en el mundo, de los que más bibliografía abunda y los que más análisis de referencia y referencia de implementación poseen. El presente estudio pretende ganar en conocimiento sobre cada uno de los elementos antes mencionados que se necesitan introducir en la arquitectura, que son más que importantes en el desarrollo de una eficiente arquitectura.

El hecho de conocer que un ADL permite eficiencia para modelar una arquitectura, no constituye una solución, la siguiente **situación problémica** lo explica**:**

- El campo de los ADLs es complejo, y de la decisión que se tome puede depender el éxito o el fracaso de un proyecto.
- El uso de los ADLs implica el estudio de una sintaxis especializada.
- No son adaptables a soluciones de cualquier empresa, ni a cualquier estilo arquitectónico.

Partiendo de lo anterior se ha definido el siguiente **Problema científico:** ¿Cómo lograr una eficiente estandarización de una SOA en los proyectos productivos de la Facultad 1 a partir de la propuesta de un ADL?

¿Cuál será el ADL que permitirá estandarizar una SOA en los proyectos productivos de la Facultad 1?

Por tanto el **Objeto de estudio** son los ADLs.

Para resolver el problema se tiene como **Objetivo general:** Determinar el ADL más óptimo para estandarizar una SOA en los proyectos productivos de la Facultad 1.

El **Campo de acción** se enmarca en los siguientes ADLs: Acme-Amani, Aesop, Darwin, Jacal, Rapide, Wright y UML.

Idea a defender: El ADL más óptimo logrará una eficiente estandarización de una SOA en los proyectos productivos de la Facultad 1.

Para alcanzar los objetivos trazados se propusieron las siguientes **Tareas de investigación:**

- Estudiar los ADLs más utilizados en la actualidad.
- Estudiar el entorno de producción de la Facultad 1.
- Seleccionar los ADLs que cumplen con las características del entorno de producción de la Facultad 1.
- Proponer el ADL más adecuado para su aplicación en los proyectos productivos de la Facultad 1.

Se hace uso de algunos **métodos de investigación**:

Teóricos:

- **Analítico-sintético:** Permitió la división mental de las partes que forman los ADLs y luego estableció la relación entre ellas.
- **Histórico-lógico:** Hizo posible analizar la trayectoria de los ADLs y hallar la lógica interna de ellos.

Empírico:

- **Observación:** Permitió el conocimiento planificado dirigido a los ADLs.

Particular:

- **Entrevista:** Constituyó un medio para el conocimiento de los ADLs a través del entrevistado.

Para lograr una correcta organización del documento y dar cumplimiento a los objetivos trazados en la investigación, se propone el desarrollo de los siguientes capítulos:

- **Capítulo 1.Fundamentación teórica**

En este capítulo se hace referencia a todo lo que respecta al surgimiento de los ADLs, primeramente explicando todos lo conceptos que permitirán un mejor entendimiento del fenómeno, después fundamentando su estado actual a nivel mundial, porque en el ámbito nacional y a nivel de centro (UCI), se carece de información y conocimiento. Luego se expresa la situación que rodea al objeto de estudio, desde el punto de vista de la arquitectura, así como características propias de los proyectos de la Facultad 1.

- **Capítulo 2. ADLs más conocidos.**

El presente capítulo ofrece uno por uno los ADLs más conocidos a nivel internacional, dotándolos de las más difíciles características encontradas en cualquier sitio de referencia. Es un análisis, lo más explicito que se pudo, que facilita sugerir a cualquier lector, que conozca o no sobre el tema, que ADL es el más apropiado para los proyectos que usen arquitectura de software.

- **Capítulo 3. Solución propuesta.**

Esta sección aborda, como lo dice el nombre, la solución propuesta. Por qué se determinó UML como solución, que ventajas posee que lo hacen superior al resto de los elementos estudiados, que características domina que posibilita hacer uso de su diagramas y no perder información, entre otras cosas que no deben ser olvidadas.

CAPITULO 1

FUNDAMENTACIÓN TEÓRICA

1.1 Introducción

El objetivo del presente capítulo es dar cuenta del estado del arte en el desarrollo de los ADLs desde el período 2004. También se examinará detalladamente la disponibilidad de diversos ADLs y herramientas correspondientes, la forma en que engranan en materia de arquitectura y su relación con el campo de los patrones arquitectónicos.
Se opta por describir los ADLs en función de variables, que a veces son las mismas en todos los casos y en otras oportunidades son específicas de la estructura de un ADL en particular.

1.2 Conceptos asociados al dominio del problema.

Los ADLs son el resultado de la tendencia de SOA, que se está poniendo en práctica en la UCI, este tema es nuevo y para poder entenderlo es necesario revisar algunos conceptos que permiten una mejor visión de la información que se transmite.

1.2.1 ¿Qué es arquitectura de software?

Existen muchas definiciones de Arquitectura de Software y no parece que ninguna de ellas haya sido totalmente aceptada.

Según Wikipedia:

- En los inicios de la informática, la programación se consideraba un arte, debido a la dificultad que entrañaba para la mayoría de los programadores, pero con el tiempo se han ido desarrollando metodologías para conseguir esos propósitos. Y a todas estas técnicas se les llama *Arquitectura de Software.*

- Una *Arquitectura de Software*, también denominada *Arquitectura lógica*, no es más que un conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia necesario para guiar la construcción del software para un sistema de información, establece los fundamentos para que analistas, diseñadores, programadores, etc. trabajen en una línea común que permita alcanzar los objetivos del sistema de información, cubriendo todas las necesidades.
- Es el diseño de más alto nivel de la estructura de un sistema, programa o aplicación y tiene la responsabilidad de:
  - ✓ Definir los módulos principales
  - ✓ Definir las responsabilidades que tendrá cada uno de estos módulos
  - ✓ Definir la interacción que existirá entre dichos módulos:
    - Control y flujo de datos
    - Secuenciación de la información
    - Protocolos de interacción y comunicación
    - Ubicación en el hardware

De los conceptos antes expuestos, se puede deducir que la Arquitectura de Software es, a grandes rasgos, una vista del sistema, que contiene los principales componentes del mismo, la conducta de esos componentes según se percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión propuesta.

1.2.2 ¿Qué es SOA?

Según IBM:

- Es un modelo de componente que interrelaciona unidades funcionales diferentes de una aplicación, denominado servicios, a través de interfaces y contratos bien definidos entre estos servicios. La interfaz se define de una manera neutral que debe ser independiente de la plataforma de hardware, del sistema operativo y del lenguaje de programación en los que se implemente el

servicio. Esto permite que los servicios, construidos en una variedad de tales sistemas, interactúen entre sí de una manera uniforme y universal.

Según Wikipedia:

- Es un concepto de arquitectura de software que define la utilización de servicios para dar soporte a los requerimientos de software del usuario. SOA permite la creación y cambios de los procesos de negocio desde la perspectiva de Tecnología Informática de forma ágil, a través de la composición de nuevos procesos, utilizando las funcionalidades de negocio que están contenidas en la infraestructura de aplicaciones actuales o futuras.
- SOA establece un marco de diseño para la integración de aplicaciones independientes de manera que, desde la red pueda accederse a sus funcionalidades, las cuales se ofrecen como servicios. La forma más habitual de implementarla es mediante Servicios Web, una tecnología basada en estándares e independiente de la plataforma, con la que SOA puede descomponer aplicaciones monolíticas en un conjunto de servicios e implementar esta funcionalidad en forma modular.

De los conceptos expuestos, se concluye que SOA fomenta la creación de aplicaciones que se adaptan fácilmente a los inevitables y frecuentes cambios del negocio. Es tanto un marco de trabajo como un marco de implementación para el desarrollo de los procesos de negocio de una organización, es utilizada con fines organizativos. Propone desglosar las funcionalidades principales de las aplicaciones en servicios independientes y luego organizarlos para formar nuevas aplicaciones de negocio. Estos servicios se encuentran disponibles en la red y presentan interfaces estándares bien definidas, que permiten el flujo de mensajes entre proveedores y consumidores.

1.3 Objeto de estudio.

1.3.1 ¿Qué es un ADL?

Lenguaje descriptivo de modelado que se centra en la estructura de alto nivel de la aplicación antes que en los detalles de implementación de sus módulos concretos.

Una vez que el arquitecto de software, tras conocer el requerimiento, se decide a delinear su estrategia y articular los patrones que se le ofrecen, se supone que debería expresar las características de su sistema, en otras palabras, modelarlo aplicando una convención gráfica o algún lenguaje avanzado de alto nivel de abstracción.

1.3.2 Descripción general.

A esta altura del desarrollo de la arquitectura de software, podría pensarse que hay abundancia de herramientas de modelado que facilitan la especificación de desarrollos basados en principios arquitectónicos, que dichas herramientas han sido aceptadas y estandarizadas hace tiempo y que son de propósito general, adaptables a soluciones de cualquier mercado vertical y a cualquier estilo arquitectónico. La creencia generalizada sostendría que modelar arquitectónicamente un sistema se asemeja al trabajo de articular un modelo en ambientes ricos en prestaciones gráficas, como es el caso del modelado de tipo CASE (Computer Aided Software Engineering, Ingeniería de Software Asistida por Ordenador) o Lenguaje Unificado de Modelado, (en adelante, UML), y que el arquitecto puede analizar visualmente el sistema sin sufrir el aprendizaje de una sintaxis especializada. También podría pensarse que los instrumentos incluyen la posibilidad de diseñar modelos correspondientes a proyectos basados en tecnología de internet, Web Services o soluciones de integración de plataformas heterogéneas, y que, una vez trazado el modelo, el siguiente paso en el ciclo de vida de la solución se produzca con naturalidad y esté servido por técnicas bien definidas. Este documento, expresa por qué la situación es otra, no la que se piensa, la misma dista de ser clara y es más compleja.

En primer lugar, el escenario de los Web Services ha forzado la definición de un estilo de arquitectura que no estaba contemplado a la escala debida en el inventario canónico de tuberías y filtros, repositorio, eventos, capas, llamada y retorno, y máquinas virtuales. El contexto de situación, como lo reveló la impactante tesis de Roy

Fielding sobre REST[1], es hoy en día bastante distinto al de los años de surgimiento de los estilos y los ADLs.

Los ADLs se utilizan además para satisfacer requerimientos descriptivos de alto nivel de abstracción. Entre las comunidades consagradas al modelado Orientado a Objetos y la que patrocina o frecuenta los ADLs (así como entre las que se inclinan por el concepto de estilos arquitectónicos y las que se trabajan en función de patrones) existen relaciones complejas de contradicción.

Según señala Mary Shaw[2] y Nenan Medvidovic[3] en sus revisiones de los ADLs, el uso que se da en la práctica y en la referencia a conceptos de arquitectura de software o estilos, a veces es informal. En la práctica industrial, las configuraciones arquitectónicas se describen por medio de diagramas de cajas y líneas, con algunos añadidos; los entornos para esos diagramas son de espléndida calidad gráfica, comunican con relativa efectividad la estructura del sistema y siempre brindan algún control de consistencia respecto a qué clase de elemento se puede conectar con otro cualquiera, pero a la larga proporcionan escasa información sobre la computación efectiva representada por las cajas y las interfaces expuestas por los componentes o la naturaleza de sus computaciones.

En la década de 1990 y en lo que va del siglo XXI se han materializado diversas propuestas para describir y razonar en términos de arquitectura de software; muchas de ellas han asumido la forma de ADLs. Estos suministran construcciones para especificar abstracciones arquitectónicas y mecanismos para descomponer un sistema en componentes y conectores, especificando la manera en que estos elementos se combinan para formar configuraciones y definiendo familias de arquitecturas o estilos. Contando con un ADL, un arquitecto puede razonar sobre las propiedades del sistema con precisión, pero a un nivel de abstracción convenientemente genérico. Algunos ejemplos de esas propiedades podrían ser, protocolos de interacción, anchos de banda y latencia, localización del almacenamiento, conformidad con estándares arquitectónicos y previsiones de evolución futura del sistema.

Hasta la publicación de las sistematizaciones de Shaw y Garlan, Kogut y Clements o Nenan Medvidovic existía relativamente poco conocimiento respecto a qué es un ADL,

cuáles lenguajes de modelado califican como tales y cuáles no, qué aspectos de la arquitectura se deben modelar con un ADL y cuáles de éstos son más adecuados para modelar un tipo de arquitectura o estilo. También existía, y aún subsiste, cierta ambigüedad a propósito de la diferencia entre ADLs, especificaciones formales (como CHAM y Z), lenguajes de interconexión de módulos (MIL, como MIL75 o Intercol), lenguajes de modelado de diseño (UML), herramientas CASE y hasta determinados lenguajes con reconocidas capacidades para modelar, como es el caso de CODE, un lenguaje de programación paralela que en opinión de Paul Clements podría tipificar como un ADL satisfactorio.

La definición de ADL que habrá de aplicarse en lo sucesivo, es la de un lenguaje descriptivo de modelado, que se centra en la estructura de alto nivel de la aplicación antes que en los detalles de implementación de sus módulos concretos[4]. Los ADLs que existen actualmente en la industria rondan la cifra de 20. Otros, que suman 40 ó 50, han experimentado dificultades en su desarrollo o no se han impuesto en el mercado de las herramientas de arquitectura. Todos ellos oscilan entre constituirse como ambientes mecánicos con eventuales interfaces gráficas o presentarse como sistemas rigurosamente formales con abundancia de notación simbólica, en la que cada entidad responde a algún teorema. Alrededor de algunos ADLs se ha establecido una constelación de herramientas de análisis, verificadores de modelos, aplicaciones de generación de código, parsers, soportes de rutinas, etcétera.

No existe hasta hoy una definición aprobada y uniforme de ADL, pero comúnmente se acepta que un ADL debe proporcionar un modelo explícito de componentes, conectores y sus respectivas configuraciones. Se estima deseable, además, que un ADL suministre soporte de herramientas para el desarrollo de soluciones basadas en arquitectura y su posterior evolución.

La delimitación categórica de los ADLs es problemática. Ocasionalmente, otras notaciones y formalismos se utilizan como si fueran ADLs para la descripción de arquitecturas. Esos ejemplos serían CHAM, UML y Z. Aunque se hará alguna referencia a ellos, cabe puntualizar que no son ADLs en sentido estricto. Se hará una excepción con UML, sin embargo. A pesar de no calificar en absoluto como ADL, se ha probado que UML puede utilizarse no tanto como un ADL por derecho propio, sino

como metalenguaje para simular otros ADLs, y en particular C2 y Wright[5]. Otra excepción concierne a CHAM, por razones similares.

Éstos se elaboraron mayormente en organismos de estandarización, casi todos los ADLs se originan en ámbitos universitarios.

1.3.2.1 Criterios de definición de un ADL.

Los ADLs se remontan a los lenguajes de interconexión de módulos (MIL) de la década de 1970, pero se comenzaron a desarrollar con su denominación actual a partir de 1992 o 1993, poco después de fundada la propia arquitectura de software como especialidad profesional. Las siguientes definiciones son los rasgos que permiten conformar a un ADL, según varios científicos.

La definición más simple es la de Tracz[6] que define un ADL como una entidad consistente en cuatro "C":

- Componentes
- Conectores
- Configuraciones
- Restricciones (Constraints).

Una de las definiciones más tempranas es la de Vestal[7] quien sostiene que un ADL debe modelar o soportar los siguientes conceptos:

- Componentes
- Conexiones
- Composición jerárquica.
- Paradigmas de computación.
- Paradigmas de comunicación.
- Modelos formales subyacentes.
- Soporte de herramientas para modelado, análisis, evaluación y verificación.
- Composición automática de código aplicativo.

Basándose en su experiencia sobre Rapide, Luckham y Vera[8] establecen como requerimientos:

- Abstracción de componentes
- Abstracción de comunicación
- Integridad de comunicación
- Capacidad de modelar arquitecturas dinámicas
- Composición jerárquica
- Relatividad

Tomando como parámetro de referencia a UniCon (Universal Connector Support, un ADL) Shaw y otros[9] alegan que un ADL debe exhibir:

- Capacidad para modelar componentes con garantías de propiedades, interfaces e implementaciones.
- Capacidad de modelar conectores con protocolos, aserción de propiedades e implementaciones
- Abstracción y encapsulamiento
- Tipos y verificación de tipos
- Capacidad para integrar herramientas de análisis

Otros autores, como Shaw y Garlan[10] estipulan que en los ADLs los conectores sean tratados explícitamente como entidades de primera clase y han afirmado que un ADL genuino tiene que proporcionar:

- Propiedades
- Composición
- Abstracción
- Reusabilidad
- Configuración
- Heterogeneidad
- Análisis

La especificación más completa y sutil es la de Medvidovic[11]:

- Componentes
  - Interfaces
  - Tipos
  - Semántica
  - Restricciones
  - Evolución
  - Propiedades no funcionales
- Conectores
  - Interfaces
  - Tipos
  - Semántica
  - Restricciones
  - Evolución
  - Propiedades no funcionales

- Configuraciones
  - Comprensibilidad
  - Composición jerárquica
  - Heterogeneidad
  - Restricciones
  - Refinamiento y trazabilidad
  - Escalabilidad
  - Evolución
  - Dinamismo
  - Propiedades no funcionales

1.3.2.2 Elementos constitutivos primarios.

Después de sugeridos varios criterios, los requerimientos presentes en todos ellos son los siguientes:

Componentes: son las entidades computacionales activas de un sistema. Ellas realizan tareas mediante cómputo interno y comunicación externa con el resto del sistema. La relación entre un componente y su entorno se define explícitamente como una colección de puntos de interacción o puertos.
Un componente tiene sus datos y espacio de ejecución independientes, aunque podría compartirlos con otros componentes.

Conectores: definen la interacción entre los componentes. Cada conector provee una forma para que una colección de puertos esté en contacto y define lógicamente el protocolo a través del cual un conjunto de componentes puede interactuar. Los puertos definen los puntos de interacción de los conectores. Al igual que los componentes, un conector tiene una interfaz, la cual consiste en un conjunto de roles.

Configuración o topología: es una colección de instancias de componentes que interactúan mediante instancias de conectores. En otras palabras, es un grafo de componentes y conectores que describen la estructura de la arquitectura. Las características de nivel de configuración se agrupan en tres categorías generales: calidad de la descripción de la configuración, calidad de la descripción del sistema, propiedades de la descripción del sistema.

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, la latencia probable, o cuestiones de seguridad, escalabilidad, dependencia de bibliotecas o servicios específicos, configuraciones mínimas de hardware y tolerancia a fallas.

Restricciones: están presentes en cada una de las tres características anteriormente expuestas (componentes, conectores y configuración). Las restricciones son propiedades o afirmaciones sobre un sistema o alguna de sus partes.

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.

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.

Propiedades no funcionales: la especificación de estas, es necesaria para simular y analizar la conducta de los componentes, imponer restricciones, mapear implementaciones sobre procesadores determinados, etcétera.

1.3.2.3 Caso particular UML.

Cualquier rama de ingeniería o arquitectura ha encontrado útil desde hace mucho tiempo la representación de los diseños de forma gráfica. Desde los inicios de la informática se han estado utilizando distintas formas de representar los diseños de una forma más bien personal o con algún modelo gráfico. La falta de estandarización en la manera de representar gráficamente un modelo impedía que los diseños gráficos realizados se pudieran compartir fácilmente entre distintos diseñadores.
Se necesitaba por tanto un lenguaje no sólo para comunicar las ideas a otros desarrolladores sino también para servir de apoyo en los procesos de análisis de un problema. Con este objetivo se creo el UML, que se ha convertido en un estándar muy ansiado para representar y modelar la información con la que se trabaja en las fases de análisis y, especialmente, de diseño.
Tal como indica su nombre, UML es un lenguaje de modelado. Un modelo es una simplificación de la realidad. El objetivo del modelado de un sistema, es capturar las partes esenciales del sistema. Para facilitar este modelado, se realiza una abstracción y se plasma en una notación gráfica. Esto se conoce como modelado visual.

El modelado visual permite manejar la complejidad de los sistemas a analizar o diseñar. De la misma forma que para construir una choza nos hace falta un modelo, cuando se intenta construir un sistema complejo como un rascacielos, es necesario abstraer la complejidad en modelos que el ser humano pueda entender.
Otro objetivo de este modelado visual es que sea independiente del lenguaje de implementación, de tal forma que los diseños realizados usando UML se puedan implementar en cualquier lenguaje que soporte sus posibilidades.

1.3.3 Descripción actual del dominio del problema (Facultad 1).

Uno de los múltiples puntos que hay tener en cuenta para seleccionar un ADL, es conocer el entorno donde se va a exponer, no es otra cosa que una breve descripción de los elementos que componen la arquitectura, por ejemplo, en la Facultad 1 existen varios proyectos y de ellos se conoce que tienen los siguientes rasgos:

1.3.3.1 Tipos de aplicaciones que se desarrollan.

Aplicación: es un tipo de programa informático diseñado para facilitar al usuario la realización de un determinado tipo de trabajo. Esto lo diferencia principalmente de otros tipos de programas, como los sistemas operativos, que hacen funcionar al ordenador; las utilidades, que realizan tareas de mantenimiento o de uso general, y los lenguajes de programación, con el cual se crean los programas informáticos, que realizan tareas más avanzadas y no pertinentes al usuario común.

Resultar una solución informática para la automatización de ciertas tareas complicadas como pueden ser la contabilidad, la redacción de documentos o la gestión de un almacén. Algunos ejemplos de aplicaciones son los procesadores de textos, hojas de cálculo, base de datos, etc.

Los proyectos de la facultad 1 trabajan con dos tipos de aplicaciones fundamentalmente, incluyendo que algunos trabajan con las dos al mismo tiempo: de Gestión y Web.

Aplicación web: es un sistema informático que los usuarios utilizan accediendo a un servidor web, a través de Internet o de una intranet. Dichas aplicaciones son populares debido a la practicidad del navegador web como cliente ligero. La facilidad para actualizar y mantener aplicaciones web sin distribuir e instalar software en miles de potenciales clientes es otra razón de su popularidad. Aplicaciones como los webmails, wikis, weblogs, tiendas en línea y la Wikipedia misma son ejemplos bien conocidos de aplicaciones web.

Aunque muchas variaciones son posibles, una aplicación web está comúnmente estructurada como una aplicación de tres-capas. En su forma más común, el navegador web es la primera capa (Interfaz), un motor usando alguna tecnología web dinámica (ejemplo: PHP, Java o ASP) es la capa de en medio (Negocio), y una base de datos como última capa (Acceso a datos). El navegador web manda peticiones a la capa negocio, que la entrega valiéndose de consultas y actualizaciones a la base de datos generando una interfaz de usuario.

Se entiende como **aplicación de gestión** aquella que se diseña para sustituir uno o varios procedimientos, tanto comerciales como administrativos, que habitualmente realiza una persona en una empresa o institución de forma presencial, a través de un ordenador, que permitan realizar al cliente los mismos procedimientos de forma no presencial.

1.3.3.2 Metodologías

Todos los procesos de desarrollos de software deben ser guiados por una metodología, encargada de elaborar estrategias de desarrollo de software.

Existen varias metodologías, cada una con características fundamentales, por las que se debe regir un desarrollador: XP, MSF, RUP, entre otras.

La metodología usada en los proyectos de la Facultad 1 es RUP (Proceso Unificado de Rational), constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas.

No es un sistema con pasos firmemente establecidos, sino un conjunto de metodologías adaptables al contexto y necesidades de cada organización.

Originalmente se diseñó un proceso genérico y de dominio público, el Proceso Unificado, y una especificación más detallada, el *RUP*, que se vendiera como producto independiente.

1.3.3.3 Estilos arquitectónicos.

Los estilos arquitectónicos son clasificaciones de los sistemas de software en grandes familias cuyos integrantes comparten un patrón estructural común.

Estilos arquitectónicos más comunes.

- Estilo de Flujo de Datos.
- Estilo Centrado en Datos.
- Estilo de Llamada y Retorno.
- Estilo de Código Móvil.
- Estilo Peer To Peer (Punto a Punto).

El estilo usado en los proyectos de la Facultad 1 es el de Llamada y Retorno, que encierra la Arquitectura en Capas, Arquitectura Orientada a Objetos y la Arquitectura Basada en Componentes.

- ***Arquitectura en Capas:*** En este estilo arquitectónico cada capa proporciona servicios a la capa superior y se sirve de las prestaciones que le brinda la inferior, al dividir un sistema en capas, cada capa puede tratarse de forma independiente, sin tener que conocer los detalles de las demás. La división de un sistema en capas facilita el diseño modular, en la que cada capa encapsula un aspecto concreto del sistema y permite además la construcción de sistemas débilmente acoplados, lo que significa que si se minimiza las dependencias

entre capas, resulta más fácil sustituir la implementación de una capa sin afectar al resto del sistema.

- ***Arquitectura Orientada a Objetos:*** Los componentes de este estilo son los objetos o instancias de los tipos de dato abstractos. En la caracterización clásica de David Garlan y Mary Shaw, los objetos representan una clase de componentes que ellos llaman managers, debido a que son responsables de preservar la integridad de su propia representación.

- ***Arquitectura basada en Componentes:*** Los sistemas de software basados en componentes se basan en principios definidos por una ingeniería de software específica. Los componentes son las unidades de modelado, diseño e implementación. Las interfaces están separadas de las implementaciones, y conjuntamente con sus interacciones son el centro de responsabilidades en el diseño arquitectónico. Los componentes soportan algún sistema de abstracción, de modo que su funcionalidad y propiedades puedan ser descubiertas y utilizadas en tiempo de ejecución.

1.3.3.4 Lenguaje de programación.

Un **lenguaje de programación** es un lenguaje que puede ser utilizado para controlar el comportamiento de una máquina, particularmente una computadora. Consiste en un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones.

Aunque muchas veces se usa lenguaje de programación y lenguaje informático como si fuesen sinónimos, no tiene por qué ser así, ya que los lenguajes informáticos engloban a los lenguajes de programación y a otros más.

El lenguaje usado en la Facultad 1 es Java.

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems. El lenguaje en sí mismo toma mucha de su sintaxis de C y C++, pero

tiene un modelo de objetos más simple y elimina herramientas de bajo nivel, que suelen inducir a muchos errores, como la manipulación directa de punteros o memoria.

Las aplicaciones Java están típicamente compiladas en un *bytecode*, aunque la compilación en código máquina nativo también es posible. En tiempo de ejecución, el *bytecode* es normalmente interpretado a código nativo, aunque la ejecución directa por hardware del *bytecode* por un procesador Java también es posible.

1.3.3.5 Plataformas

Una plataforma es precisamente el principio, en el cual se constituye un hardware, sobre el cual un software puede ejecutarse o desarrollarse, es un sistema operativo o sistema complejo, que sirve para crear programas. No debe confundirse esto con arquitecturas.

Las plataformas que lideran en este momento el desarrollo de aplicaciones son: Linux y Windows. En la Facultad 1, en vista a la migración a Linux, se está desarrollando este sistema operativo en algunos proyectos, pero por completo no se ha dejado de usar Windows.

Windows es una familia de sistemas operativos desarrollados y comercializados por Microsoft. Existen versiones para hogares, empresas, servidores y dispositivos móviles, como computadores de bolsillo y teléfonos inteligentes. Hay variantes para procesadores de 16, 32 y 64 bits.

La unión de Windows NT/2000 y la familia de Windows 9.x se alcanzó con Windows XP en su versión Home y Professional. Windows XP usa el núcleo de Windows NT. Incorpora una nueva interfaz y hace alarde de mayores capacidades multimedia. Además dispone de otras novedades como la multitarea mejorada, soporte para redes inalámbricas y asistencia remota. Se puede agregar inmediatamente de haber lanzado el último Service Pack (SP2) Microsoft diseñó un sistema orientado a Empresas y Corporaciones llamado Microsoft Windows XP Corporate Edition, algo similar al

Windows XP Profesional, solo que diseñado especialmente a Empresas. En el apartado multimedia, XP da un avance con la versión Media Center (2002-2005). Esta versión ofrece una interfaz de acceso fácil a todo lo relacionado con multimedia (TV, fotos, reproductor DVD, Internet...).

Linux es un sistema operativo tipo Unix (también conocido como GNU/Linux) que se distribuye bajo la Licencia Pública General de GNU (GNU GPL), es decir que es software libre. Su nombre proviene del Núcleo de Linux. Es usado ampliamente en servidores y súper-computadoras y cuenta con el respaldo de corporaciones como Dell, Hewlett-Packard, IBM, Novell, Oracle, Red Hat y Sun Microsystems.
Puede ser instalado en gran variedad de hardware, incluyendo computadoras de escritorio y portátiles (PCs x86 y x86-64 así como Macintosh y PowerPC), computadoras de bolsillo, teléfonos celulares, dispositivos empotrados, videoconsolas (Xbox, PlayStation 3, PlayStation Portable, Dreamcast, GP2X) y otros (como máquinas de juegos, enrutadores y algunos modelos de dispositivos iPod).

1.4 Los ADLs y la Arquitectura de Software.

Es cierto que los ADLs no son adaptables a cualquier estilo arquitectónico, pero tampoco son de una arquitectura específica, por tanto la propuesta de un ADL para SOA no significa que sean propios de ésta. Cualquier ADL puede representar una solución para cualquier arquitectura, siempre que éste cumpla con las condiciones que se quieren.
Anteriormente se explicó que en los proyectos de la Facultad 1, no existe conocimiento acerca de lo que es capaz de resolver un ADL, por tanto no se cuenta con ninguno que sea capaz de describir la arquitectura que se emplea.
Pero SOA es una arquitectura ambiciosa, que requiere de un conjunto de necesidades y objetivos básicos que son estratégicos, por tanto, para asegurar que la implantación tenga un margen de error mínimo, se ha decidido tomar todas las medidas pertinentes para una eficiente estandarización de SOA.

Conclusiones

Este capítulo permitió un acercamiento a la definición de los ADLs que constituyen una posible solución al problema que se planteó, a través de un resumen de las características del entorno de producción de la Facultad 1, que no es más que el reflejo de un pequeño subsistema de la UCI, destacándose metodologías usadas en los proyectos, estilos arquitectónicos, los tipos de aplicaciones que se desarrollan, entre otros.

Es importante destacar que existen lenguajes que no pertenecen al grupo de los ADLs, pero que realizan las misma funciones que estos, incluso en ocasiones son más efectivos, por este motivo juegan un papel significativo en la realización de una comparación entre ADLs, tal es el caso de UML.

CAPÍTULO 2

ADLS MÁS CONOCIDOS

2.1 Introducción

Este capítulo realiza una breve recopilación de los ADLs más utilizados en la actualidad.

Dentro de la gama de los ADLs que son muchos, los más difundidos son tratados en este documento, que además, en cierta manera constituyen una respuesta parcial o total al problema planteado. Algunos, no cumplen con la definición exacta de ADL[1], pero de igual manera van a ser objeto de análisis para poder arribar a conclusiones que servirán de apoyo a planteamientos posteriores.

2.2 Lenguajes de Descripción de Arquitectura.

Al ser los más reconocidos por las instituciones que se encargan de esta parte tan desconocida de la arquitectura, ostentan la mayor bibliografía existente sobre el tema y el resto de ellos se aleja de lo más deseado en un ADL. Por esta razón se han tomado únicamente los siguientes lenguajes como caso de estudio, cada uno va a ser objeto de explicación de cómo se comportan las características mínimas necesarias para ser un ADL.

2.2.1 Acme-Armani

Acme: se define como una herramienta capaz de soportar el mapeo de especificaciones arquitectónicas entre diferentes ADLs, en otras palabras, un lenguaje de intercambio de arquitectura. No es entonces un ADL en sentido estricto, aunque la literatura de referencia acostumbra a tratarlo como tal. De hecho, posee numerosas

[1] Hay lenguajes que por sus características entran en el ámbito de los ADLs, aunque exactamente no cumplen con la definición propia de ellos, sin embargo serán tratados, al igual que la literatura de referencia lo hace, permitido por sus autores.

prestaciones que también son propias de los ADLs. En su sitio oficial se reconoce que como ADL no es necesariamente apto para cualquier clase de sistemas, al mismo tiempo que se destaca su capacidad de describir con facilidad sistemas “relativamente simples”.

Sitio oficial de referencia: El proyecto Acme comenzó a principios de 1995 en la Escuela de Ciencias de la Computación de la Universidad Carnegie Mellon. Hoy en día este proyecto se organiza en dos grandes grupos, que son el lenguaje Acme propiamente dicho y el Acme Tool Developer’s Library (AcmeLib).

Objetivo principal: La motivación fundamental de Acme es el intercambio entre arquitecturas e integración de ADLs. Garlan considera que Acme es un lenguaje de descripción arquitectónica de segunda generación; podría decirse que es de segundo orden: un metalenguaje, una lengua para el entendimiento de dos o más ADLs, incluido Acme mismo. Con el tiempo, la dimensión metalingüística de Acme fue perdiendo prioridad y los desarrollos actuales profundizan su capacidad exclusiva como ADL puro. (Ver Anexo 1).

Acme soporta la definición de varios tipos de arquitectura:

- Estructura: organización de un sistema en sus partes constituyentes.
    - ✓ Componentes.
    - ✓ Conectores.
    - ✓ Sistemas.
    - ✓ Puertos.
    - ✓ Roles.
    - ✓ Representaciones.
    - ✓ Rep-mapas: mapas de representación.
- Propiedades de interés: información que permite razonar sobre el comportamiento local o global, tanto funcional como no funcional.
- Restricciones: lineamientos sobre la posibilidad del cambio en el tiempo.

- Tipos y estilos.

Componentes: Representan elementos computacionales y almacenamientos de un sistema. Un componente se define siempre dentro de una familia de componentes.

Interfaces: Todos los ADLs conocidos soportan la especificación de interfaces para sus componentes. En Acme cada componente puede tener múltiples interfaces. Igual que en Aesop y Wright los puntos de interfaz se llaman puertos. Los puertos pueden definir interfaces tanto simples como complejas, desde una señal de procedimiento hasta una colección de rutinas a ser invocadas en cierto orden, o un evento de multicast.

Semántica: Muchos lenguajes de tipo ADL no modelan la evolución de los componentes más allá de sus interfaces. En este sentido, Acme sólo soporta cierta clase de información semántica en listas de propiedades. Estas propiedades no se interpretan, y sólo existen a efectos de documentación.

Estilos: Acme posee manejo intensivo de estilos. Esta capacidad está construida naturalmente como una jerarquía de propiedades correspondientes a tipos. Acme considera, en efecto, tres clases de tipos: propiedades, estructurales y estilos. Así como los tipos estructurales representan conjuntos de elementos estructurales, un estilo representa un conjunto de sistemas. Una familia Acme se define especificando tres elementos de juicio: un conjunto de tipos de propiedades y tipos estructurales, un conjunto de restricciones y una estructura por defecto, que prescribe el conjunto mínimo de instancias que debe aparecer en cualquier sistema de la familia. El uso del término “familia” con preferencia a “estilo” recupera una idea de uno de los precursores tempranos de la arquitectura de software, David Parnas[12].
Interfaz gráfica: La versión actual de Acme soporta una variedad de front-ends de carácter gráfico. El ambiente primario, llamado AcmeStudio, es un entorno gráfico basado en Windows, configurado para soportar visualizaciones específicas de estilos e invocación de herramientas auxiliares. Un segundo entorno, llamado Armani, utiliza Microsoft Visio como front-end gráfico y un back-end Java, que con alguna

transformación puede ser Microsoft Visual J++ o incluso Visual J# de .NET. Armani no es un entorno redundante sino, por detrás de la fachada de Visio, un lenguaje de restricción que extiende Acme basándose en lógica de predicados de primer orden, y que es por tanto útil para definir invariantes y heurísticas de estilos. Un tercer ambiente, más experimental, utiliza sorprendentemente el editor de PowerPoint para manipulación gráfica.

Generación de código: En los últimos años se ha estimado cada vez más que un ADL pueda generar un sistema ejecutable, aunque más no sea de carácter prototípico. De tener que hacerlo manualmente, se podrían originar problemas de consistencia y trazabilidad entre una arquitectura y su implementación. Acme, al igual que Wright, se concibe como una notación de modelado y no proporciona soporte directo de generación de código.

Disponibilidad de plataforma*:* Un front-end gráfico programado en Visual C++ y Java que corre en plataforma Windows y que proporciona un ambiente completo para diseñar modelos de arquitectura. La sección de Java requiere JRE, pero también se puede trabajar en términos de COM y Win32 ejecutando AcmeStudio.exe. Los otros dos ambientes gráficos (Armani y el entorno de ISI) son nativos de Windows e implementan intensivamente tecnología COM.

Armani se constituyó en un lenguaje de tipo ADL, especializado en la descripción de la estructura de un sistema y su evolución en el tiempo. Es un lenguaje puramente declarativo que describe la estructura del sistema y las restricciones a respetar, pero no hace referencia alguna a la generación del sistema o a la verificación de sus propiedades no funcionales o de consistencia.

Armani: se basa en siete entidades para describir las instancias del diseño:

- Componentes
- Conectores
- Puertos
- Roles
- Sistemas

- Representaciones
- Propiedades.

Para capturar las experiencias Armani implementa además otras seis entidades que son:

- Tipos de elementos de diseño
- Tipos de propiedades
- Invariantes de diseño
- Heurísticas
- Análisis
- Estilos arquitectónicos.

2.2.2 Aesop

El nombre oficial es Aesop (Software Architecture Design Environment Generator). Se ha desarrollado como parte del proyecto ABLE de la Universidad Carnegie Mellon. Es un ADL de ambiente Orientado a Objeto.

Objetivo principal: 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. Aesop sólo soporta nativamente desarrollos realizados en C++. (Ver Anexo 2).

Sitio de referencia: se encuentra siguiendo el rastro de la Universidad Carnegie Mellon, la escuela de computación científica y el proyecto ABLE.

Estilos: En Aesop, conforme a su naturaleza orientada a objetos, el vocabulario relativo a estilos arquitectónicos se describe mediante la definición de sub-tipos de los

tipos arquitectónicos básicos: Componente, Conector, Puerto, Rol, Configuración y Binding.

Interfaces: En Aesop, igual que en ACME y Wright, los puntos de interfaz se llaman puertos.

Modelo semántico: Aesop presupone que la semántica de una arquitectura puede ser arbitrariamente distinta para cada estilo. Por lo tanto, no incluye ningún soporte nativo para la descripción de la semántica de un estilo o configuración, sino que apenas presenta unos cuadros vacantes para colocar esa información como comentario.

Soporte de lenguajes: Aesop, igual que Darwin, sólo soporta nativamente desarrollos realizados en C++.

Generación de código: Aesop genera código C++, opera primariamente desde una interfaz visual y posee una estructura de orden más bien declarativo.

Disponibilidad de plataforma: Aesop no está disponible para plataforma Windows, aunque puede utilizarse para modelar sistemas implementados en cualquier plataforma.

2.2.3 Darwin

Es un lenguaje de descripción arquitectónica desarrollado por Jeff Magee y Jeff Kramer[13,14]. 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 (que se espera ocurran en el entorno). Las configuraciones se desarrollan instanciando las declaraciones de componentes y estableciendo vínculos entre ambas clases de servicios. Contiene un framework para la integración de componentes.

Este soporta la descripción de arquitecturas que se reconfiguran dinámicamente a través de dos construcciones: instanciación tardía y construcciones dinámicas explícitas. Utilizando instanciación tardía, se describe una configuración y se

instancian componentes sólo en la medida en que los servicios que ellos provean sean utilizados por otros componentes. En cambio, la estructura dinámica explícita, se realiza mediante construcciones de configuración imperativos. De este modo, la declaración de configuración sucede de un programa en tiempo de ejecución, antes que una declaración estática de la estructura.

Darwin no proporciona una base adecuada para el análisis de la conducta de una arquitectura, debido a que el modelo no dispone de ningún medio para describir las propiedades de un componente o de sus servicios más que como comentario. Los componentes de implementación se interpretan como cajas negras, mientras que la colección de tipos de servicio es una colección dependiente de plataforma cuya semántica tampoco se encuentra interpretada en el framework de Darwin[15]. (Ver Anexo 3)

Objetivo principal: Darwin está orientado más que nada al diseño de arquitecturas dinámicas y cambiantes.

Estilos: El soporte de Darwin para estilos arquitectónicos se limita a la descripción de configuraciones bajo parámetros. Esta descripción, en particular, indica que una tubería es una secuencia lineal de filtros, en la que la salida de cada filtro se vincula a la entrada del filtro siguiente en la línea. Un estilo será entonces expresable en Darwin en la medida en que pueda ser constructivamente caracterizado; en otras palabras, para delinear un estilo hay que construir un algoritmo capaz de representar los miembros de un estilo. Dada su especial naturaleza, es razonable suponer que Darwin se presta mejor a la descripción de sistemas que poseen características dinámicas.

Interfaces: En Darwin las interfaces de los componentes consisten en una colección de servicios que pueden ser provistos o requeridos.

Conectores: Al pertenecer a la clase en la que Medvidovic[11] agrupa a los lenguajes de configuración in-line, en Darwin, al igual que en Rapide, no es posible ponerle nombre, teclear o reutilizar un conector. Tampoco se pueden describir patrones de interacción independientemente de los componentes que interactúan.

Semántica: Darwin proporciona una semántica para sus procesos estructurales mediante cálculo. Cada servicio se modela como un nombre de canal, y cada declaración de enlace se entiende como un proceso que trasmite el nombre de ese canal a un componente que requiere el servicio. Este modelo se ha utilizado para demostrar la corrección lógica de las configuraciones de Darwin. Dado que el cálculo ha sido designado específicamente para procesos móviles, su uso como modelo semántico confiere a las configuraciones de Darwin un carácter potencialmente dinámico. En un escenario como el Web, en el que las entidades que interactúan no están ligadas por conexiones fijas ni caracterizadas por propiedades definidas de localización, esta clase de cálculo se presenta como un formalismo extremadamente útil. Ha sido, una de las herramientas formales que estuvo en la base de los primeros modelos del XLANG de Microsoft, que ahora se encuentra derivando hacia BPEL4WS (Business Process Execution Language for Web Services).

Análisis y verificación: A pesar del uso de un modelo de cálculo para las descripciones estructurales, Darwin no proporciona una base adecuada para el análisis del comportamiento de una arquitectura. Esto es debido a que el modelo no posee herramientas para describir las propiedades de un componente o de los servicios que presta.

Interfaz gráfica: Darwin proporciona notación gráfica. Existe también una herramienta gráfica (Software Architect's Assistant) que permite trabajar visualmente con lenguaje Darwin. El desarrollo de SAA parecería estar discontinuado y ser fruto de una iniciativa poco formal, lo que sucede con alguna frecuencia en el terreno de los ADLs.

Soporte de lenguajes: Darwin, igual que Aesop, soporta desarrollos escritos en C++, aunque no admite que los componentes de un sistema real estén programados en algún lenguaje en particular.

Observaciones: Darwin, lo mismo que UniCon, carece de la capacidad de definir nuevos tipos, soportando sólo una amplia variedad de tipos de servicio predefinidos. Darwin presupone que la colección de tipos de servicio es suministrada por la plataforma para la cual se desarrolla una implementación, y confía en la existencia de

nombres de tipos de servicio que se utilizan sin interpretación, sólo verificando su compatibilidad.

Disponibilidad de plataforma: Aunque el ADL fue originalmente planeado para ambientes tan poco vinculados al modelado corporativo como hoy en día lo es Macintosh, en Windows se puede modelar en lenguaje Darwin utilizando SAA. Esta aplicación requiere JRE.

2.2.4 Jacal

Es un lenguaje de descripción de arquitecturas de software de propósito general creado en la Universidad de Buenos Aires, por un grupo de investigación del Departamento de Computación de la Facultad de Ciencias Exactas y Naturales.

Objetivo principal: El objetivo principal de Jacal es lo que actualmente se denomina "animación" de arquitecturas. Esto es, poder visualizar una simulación de cómo se comportaría en la práctica un sistema basado en la arquitectura que se ha representado.

Más allá de este objetivo principal, 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 de conectores, cada uno con una representación distinta.

Estilos: Jacal no cuenta con una notación particular para expresar estilos, aunque por tratarse de un lenguaje de propósito general, puede ser utilizado para expresar arquitecturas de distintos estilos. No ofrece una forma de restringir una configuración a un estilo específico, ni de validar la conformidad.

Interfaces: Cada componente cuenta con puertos que constituyen su interfaz y a los que pueden respaldarse conectores.

Semántica: Jacal tiene una semántica formal que está dada en función de redes de Petri. Se trata de una semántica de notación que asocia a cada arquitectura una red correspondiente. La semántica operacional estándar de las redes de Petri es la que justifica la animación de las arquitecturas.

Además del nivel de interfaz, que corresponde a la configuración de una arquitectura ya que allí se determina la conectividad entre los distintos componentes, Jacal ofrece un nivel de descripción adicional, llamado nivel de comportamiento. En este nivel, se describe la relación entre las comunicaciones recibidas y enviadas por un componente, usando diagramas de transición de estados con etiquetas en los ejes que corresponden a nombres de puertos por los que se espera o se envía un mensaje.

Análisis y verificación: Las animaciones de arquitecturas funcionan como casos de prueba. La herramienta de edición y animación disponible en el sitio del proyecto, permite dibujar arquitecturas mediante un editor orientado a la sintaxis, para luego animarlas y almacenar el resultado de las ejecuciones en archivos de texto. Esta actividad se trata exclusivamente de una tarea de prueba, debiendo probarse cada uno de los casos que se consideren críticos, para luego extraer conclusiones del comportamiento observado o de las trazas generadas. Si bien no se ofrecen actualmente herramientas para realizar procesos de verificación automática como chequeo de modelos, la traducción a redes de Petri ofrece la posibilidad de aplicar al resultado otras herramientas disponibles en el mercado.

Generación de código: En su versión actual, Jacal no genera código de ningún lenguaje de programación, ya que no fuerza ninguna implementación única para los conectores. Por ejemplo, un conector de tipo message (mensaje) podría implementarse mediante una cola de alguna plataforma de middleware, como MSMQ, o directamente como código en algún lenguaje. No obstante, la herramienta de edición de Jacal permite exportar a un archivo de texto la estructura estática de una arquitectura, que luego puede ser convertida a código fuente para usar como base para la programación.

Disponibilidad de plataforma: La herramienta que actualmente está disponible para editar y animar arquitecturas en Jacal es una aplicación Win32, que no requiere instalación, basta con copiar el archivo ejecutable para usarla.

Interfaz gráfica: La notación principal de Jacal es gráfica y hay una herramienta disponible en línea para editar y animar visualmente las arquitecturas.

En el nivel de interfaz, existen símbolos predeterminados para representar cada tipo de componente y cada tipo de conector, como se muestra a continuación. (Ver Anexo 4)

Call: transfiere el control y espera una respuesta.
Message: coloca un mensaje en una cola y sigue ejecutando.
Interrupt: interrumpe cualquier flujo de control activo en el receptor y espera una respuesta.
Fork: agrega un flujo de control al receptor, el emisor continúa ejecutando.
Kill: detiene todos los flujos de control en el receptor, el emisor continúa ejecutando.
Datagram: si el receptor estaba esperando una comunicación por este puerto, el mensaje es recibido; de lo contrario, el mensaje se pierde; en cualquier caso el emisor sigue ejecutando.
Read: la ejecución en el emisor continúa, de acuerdo con el estado del receptor.
Write: cambia el estado del receptor, el emisor continúa su ejecución.

El ambiente consiste en una interfaz gráfica de usuario, donde pueden dibujarse representaciones Jacal de sistemas, incluyendo tanto el nivel de interfaz como el de comportamiento. Se pueden editar múltiples sistemas simultáneamente y, abriendo distintas vistas, visualizar simultáneamente los dos niveles de un mismo sistema, para uno o más componentes. (Ver Anexo 5)

El editor es orientado a la sintaxis, en el sentido de que no permite dibujar configuraciones no válidas. Por ejemplo, valida la compatibilidad entre el tipo de un componente y los conectores asociados. Para aumentar la flexibilidad, especialmente en el orden en que se dibuja una arquitectura, se dejan otras validaciones para el momento de la animación.

Cuando se anima una arquitectura, los flujos de control se generan en comunicaciones iniciadas por el usuario, representa una interacción con el mundo exterior, haciendo clic en el extremo de origen de un conector que no esté ligado a ningún puerto. Los flujos de control se muestran como círculos de colores aleatorios que se mueven a lo largo de los conectores y las transiciones. Durante la animación, el usuario puede abrir vistas adicionales para observar el comportamiento interno de uno o más componentes, sin dejar de tener en pantalla la vista global del nivel de interfaz.

2.2.5 Rapide

Se puede caracterizar como un lenguaje de descripción de sistemas de propósito general que permite modelar interfaces de componentes y su conducta. Sería tanto un ADL como un lenguaje de simulación. La estructura de Rapide es sumamente compleja, y en realidad articula cinco lenguajes: de tipos, describe las interfaces de los componentes; de arquitectura, describe el flujo de eventos entre componentes; de especificación, describe restricciones abstractas para la conducta de los componentes; ejecutable, describe módulos ejecutables; y de patrones, describe patrones de los eventos. Los diversos sub-lenguajes comparten la misma visibilidad y reglas de denominación, así como un único modelo de ejecución.

Sitio de referencia: Universidad de Stanford.

Objetivo principal: Simulación y determinación de la conformidad de una arquitectura.

Interfaces: En Rapide los puntos de interfaz de los componentes se llaman

constituyentes.

Conectores: Siendo lo que Medvidovic llama un lenguaje de configuración in-line, en Rapide, al igual que en Darwin, no es posible poner nombre, teclear o reutilizar un conector.

Semántica: Mientras muchos lenguajes de tipo ADL no soportan ninguna especificación semántica de sus componentes más allá de la descripción de sus interfaces, Wright y Rapide permiten modelar la conducta de sus componentes. Rapide define tipos de componentes en términos de una colección de eventos de comunicación que pueden ser observados o iniciados. Las interfaces de Rapide definen el comportamiento computacional de un componente vinculando la observación de acciones externas, con la iniciación de acciones públicas. Cada especificación posee una conducta asociada, que se define a través de conjuntos de eventos parcialmente ordenados; Para describir comportamientos Rapide también implementa un lenguaje cuyo modelo de interfaz se basa en Standard ML, extendido con eventos y patrones de eventos.

Análisis y verificación automática: En Rapide, el monitoreo de eventos y las herramientas nativas de filtrado facilitan el análisis de arquitectura. También es posible implementar verificación de consistencia y análisis mediante simulación. En esencia, en Rapide toda la arquitectura es simulada, generando un conjunto de eventos que, se supone, es compatible con las especificaciones de interfaz, conducta y restricciones. La simulación es entonces útil para detectar alternativas de ejecución. Rapide también proporciona una caja de herramientas específica para simular la arquitectura junto con la ejecución de la implementación. Sin embargo un proceso de ejecución solamente provee una idea del comportamiento con un juego particular de variables, antes que una confirmación de la conducta frente a todos los valores y escenarios posibles. Esto implica que una corrida del proceso de simulación simplemente testea la arquitectura, y no proporciona un análisis exhaustivo del escenario. Nada garantiza que no pueda surgir una inconsistencia en una ejecución diferente. En general, la arquitectura de software mantiene una actitud de reserva crítica frente a la simulación. Paul Clements escribió: “La simulación es inherentemente una herramienta de validación débil en la

medida que sólo presenta una sola ejecución del sistema cada vez; igual que la prueba, sólo puede mostrar la presencia antes que la ausencia de fallas. Más poderosos son los verificadores o probadores de teoremas que son capaces de comparar una aserción de seguridad contra todas las posibles ejecuciones de un programa simultáneamente" [16].

Interfaz gráfica: Rapide soporta notación gráfica.

Soporte de lenguajes: Rapide soporta construcción de sistemas ejecutables especificados en VHDL, C. C++, Ada y Rapide mismo. El siguiente es un ejemplo próximo a nuestro caso de referencia de tubería y filtros, sólo que en este caso es bidireccional, ya que se ha definido una respuesta de notificación. Rapide no contempla estilos de la misma manera que la mayor parte de los otros ADLs.

Observaciones: En materia de evolución y soporte de subtipos, Rapide soporta herencia, análoga a la de los lenguajes OOP.

Implementación de referencia: Aunque se ha señalado su falta de características de escalabilidad, Rapide se ha utilizado en diversos proyectos de gran escala. Un ejemplo representativo es el estándar de industria X/Open Distributed Transaction Processing.

Disponibilidad de plataforma: Rapide ha desarrollado un conjunto de herramientas que sólo se encontraba disponible para Solaris 2.5, SunOS 4.1.3 y Linux.

2.2.6 Wright

Puede ser caracterizado 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.

Objetivo principal: Wright es probablemente la herramienta más acorde con criterios académicos de métodos formales. Su objetivo aparente 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.

Sitio de referencia: Escuela de Ciencias Informáticas de la Universidad Carnegie Mellon.

Estilos: En Wright se introduce un vocabulario común declarando un conjunto de tipos de componentes y conectores y un conjunto de restricciones. Cada una de las restricciones declaradas por un estilo representa un predicado que debe ser satisfecho por cualquier configuración de la que se declare que es miembro del estilo. La notación para las restricciones se basa en el cálculo de predicados de primer orden. Las restricciones se pueden referir a los conjuntos de componentes, conectores, a las puertos y a las computaciones de un componente específico y a los roles y ligamentos de un conector particular. Es asimismo posible definir sub-estilos que heredan todas las restricciones de los estilos de los que derivan. No existe, sin embargo, una manera de verificar la conformidad de una configuración a un estilo canónico estándar.

Interfaces: En Wright (igual que en Acme y Aesop) los puntos de interfaz se llaman puertos (ports).

Semántica: Mientras muchos lenguajes de tipo ADL no soportan ninguna especificación semántica de sus componentes más allá de la descripción de sus interfaces, Wright y Rapide permiten modelar la conducta de sus componentes. En Wright existe una sección especial de la especificación llamada Computation que describe la funcionalidad de los componentes.

Análisis y verificación automática: Wright define verificaciones de consistencia que se aplican estáticamente, y no mediante simulación. Esto lo hace más confiable que, Rapide. También se puede especificar una forma de compatibilidad entre un puerto y un rol, de modo que se pueda cambiar el proceso de un puerto por el proceso de un

rol sin que la interacción del conector detecte que algo ha sido modificado. Esto permite implementar relaciones de refinamiento entre procesos. En Wright existen además recaudos que garantizan que un conector esté libre en todos los escenarios posibles. Esta garantía no se obtiene directamente sobre Wright, sino implementando un verificador de modelos comercial llamado FDR. Herramientas complementarias generan datos de entrada para FDR a partir de una especificación Wright.

Interfaz gráfica: Wright no proporciona notación gráfica en su implementación nativa.

Generación de código: Wright se considera como una notación de modelado autocontenida y no genera (al menos en forma directa) ninguna clase de código ejecutable.

La imagen ilustra la representación gráfica correspondiente al estilo TUBERIAS. (Ver Anexo 6)

Implementación de referencia: Aunque se ha señalado su falta de características explícitas de escalabilidad, Wright se utilizó para modelar y analizar la estructura de rutina del Departamento de Defensa de Estados Unidos. Se asegura que permitió condensar la especificación original y detectar varias inconsistencias en ella.

Disponibilidad de plataforma: Wright es en principio independiente de plataforma y es un lenguaje formal, antes que una herramienta de paquete. Debido a que se basa en CSP (Communicating Sequential Process), cualquier solución que pueda tratar código CSP en plataforma Windows es apta para obtener ya sea una visualización del modelo o la consistencia del modelo. El código idóneo de ser manejado por herramientas de CSP académicas o comerciales requiere tratamiento previo por un módulo de Wright, que por el momento existe sólo para Linux; pero una vez que se ha generado el CSP se puede tratar en Windows con cualquier solución ligada a ese lenguaje.

2.2.7 UML

UML forma parte del repertorio conocido como lenguajes semi-formales de modelado. En términos de número, la masa crítica de los conocedores de UML no admite comparación con la todavía modesta población de especialistas en ADLs. La literatura crítica de UML es ya un tema clásico de la reciente arquitectura de software.

Objetivo general: UML es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; constituye un estándar oficial, y está respaldado por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. UML ofrece un estándar para describir un "plano" del sistema, incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

Se utiliza para definir un sistema de software, para detallar los artefactos en el sistema y para documentar y construir. En otras palabras, es el lenguaje en el que está descrito el modelo.

Interfaz gráfica: Utiliza una gran cantidad de software para su modelado, los mismos se pueden encontrar libres o privativos. En su totalidad suman la cifra aproximada de 20.

Soporte de lenguaje: Soporta la realización de trabajos especificados en varios lenguajes de programación.

Implementación de referencia: Es un lenguaje conocido por la mayoría de los especialistas en la materia y por empresas de renombre internacional, por tanto es normal su uso en la mayoría de los centros y proyectos que requieren arquitectura.

Generación de código: Sus diagramas de representación pueden ser llevados a cualquier lenguaje de programación y viceversa.

Disponibilidad de plataforma: Las herramientas de software que son capaces de utilizar UML, pueden utilizarse en plataforma Windows, Linux, Mac OS, entre otros.

UML no es un ADL en el sentido usual de la expresión. Sin embargo, debido al hecho de que constituye una herramienta de uso habitual en modelado, importantes autoridades en ADLs, siguiendo a Nenad Medvidovic, han investigado la posibilidad de utilizarlo como metalenguaje para implementar la semántica de al menos tres ADLs, que son C2, SADL y Wright. También se ha afirmado que el meta-modelo de UML puede constituir un equivalente a la filosofía arquitectónica de Acme.

2.2.8 Comparación de los ADLs.

La siguiente tabla expresa de manera breve, datos importantes sobre los ADLs abordados en este capítulo, los que realmente se necesitan. (Ver Anexo 7)

Conclusiones

Se logró hacer una comparación entre los ADLs escogidos, que en teoría son capaces de describir una SOA en los proyectos de la Facultad 1 y se evidenció la necesidad de tomar en cuenta de forma priorizada UML, que en este capítulo ha tomado la delantera en cuanto a ventajas que posee a la hora de modelar.

CAPÍTULO 3

SOLUCIÓN PROPUESTA

3.1 Introducción

En este capítulo se demuestra una forma de representación de la arquitectura de software basada en UML, aprovechando las ventajas de este lenguaje de modelado e incluyendo varias estructuras que facilitan la representación de amplia variedad de sistemas.

3.2 UML 2.0

Uno de los desarrollos más importantes dentro de la construcción del software ha sido el desarrollo de la arquitectura de software, que permite representar la estructura de un sistema a un nivel mayor que el dado por la programación o incluso el diseño [16], [17].

Para representar adecuadamente la arquitectura de un sistema es necesario contar con varios diagramas o vistas [18]. Dada la cantidad de características y de elementos que tiene un sistema de software no es posible incluirlos todos en un solo diagrama y que sirva, además, para todas las personas que participan en el desarrollo. Cada una de estas vistas es una estructura de la arquitectura del sistema, que muestran una parte del sistema como un conjunto de componentes, conectores y restricciones sobre sus tipos y relaciones. Además, cada estructura puede relacionarse con las demás para complementar la visión integral del sistema.

La arquitectura, conformada por diferentes visiones del sistema, constituye un modelo de cómo está estructurado dicho sistema, sirviendo de comunicación entre las personas involucradas en el desarrollo y ayudando a realizar diversos análisis que orienten el proceso de toma de decisiones.

Para que la arquitectura se convierta en una herramienta útil dentro del desarrollo y mantenimiento de los sistemas de software es necesario que se cuente con una manera precisa de representarla.

Las herramientas que se han elaborado para representar una arquitectura de software son los ADLs. Sin embargo, los lenguajes desarrollados hasta el momento presentan diferentes problemas para su utilización en una empresa.

Las desventajas que se presentan en estos lenguajes pueden ser superadas si se utiliza un lenguaje de modelado que sea conocido en la industria y que además esté apoyado por herramientas y metodologías de desarrollo, este lenguaje de modelado es UML, que se está convirtiendo en una notación estándar en las empresas.

UML permite que se represente de manera semi-formal la estructura general del sistema, con la ventaja de que este mismo lenguaje puede ser usado en todas las etapas de desarrollo del sistema y su representación gráfica puede ser usada para comunicarse con los usuarios. Por este motivo se toma como solución propuesta.

3.2.1 Elementos deseables en un ADL.

Para la representación de una arquitectura se han creado diferentes ADLs con el fin de que posean ciertos elementos, que unidos a las características del entorno productivo de la Facultad 1 se quiere que tengan:

- Expresividad gráfica para el entendimiento de personas ajenas a la construcción de software.
- Capacidad para generar códigos ejecutables.
- Soporte de desarrollos realizados en varios lenguajes.
- Desarrollos en varias plataformas (Windows y Linux principalmente).
- Herramientas y metodologías de apoyo.
- Capacidad para descomponer un sistema en componentes y conectores.

Si se logra demostrar que UML es capaz de resolver esta serie de puntos, indiscutiblemente se ha encontrado la forma de lograr una eficiente estandarización de SOA, ya que se descartarían los ADLs por presentar la siguiente situación:

- Requieren una extensa capacitación.

- No son amigables para presentar la arquitectura a personas ajenas a la construcción del software.
- No tienen herramientas ni metodologías de apoyo.
- Algunos se encuentran especializados solo en un tipo particular de sistemas.
- Sólo tienen en cuenta una sola estructura del sistema.

3.2.1.1 Expresión gráfica de UML.

UML es un lenguaje gráfico de modelado que usa conceptos de orientación por objetos. En UML se utilizan para el modelado de un sistema, diferentes elementos y relaciones. Estos elementos se agrupan en diagramas preestablecidos que corresponden a diferentes proyecciones del sistema.

Los elementos básicos de UML, aquellos que representan principalmente las partes estáticas del sistema, son:

- Clases
- Casos de uso
- Componentes
- Nodos
- Paquetes

Las relaciones que se utilizan para establecer conexiones entre los elementos son:

- Dependencia
- Asociación
- Generalización
- Realización

Cada uno de estos elementos y relaciones tiene una representación gráfica y puede complementarse su información utilizando lo que se conoce como especificación. La especificación de un elemento o relación, generalmente no es visible en la

representación gráfica, o sólo lo es parcialmente, y corresponde a los datos o propiedades adicionales que completan o detallan la semántica del elemento o relación, y por lo tanto del sistema en general.

Los elementos y relaciones se agrupan en diagramas que representan diferentes aspectos del sistema. **Los diagramas de UML son:**

- ***Diagrama de clases:*** Presenta las clases, junto con sus atributos, operaciones, interfaces y relaciones. También presenta el agrupamiento de clases en paquetes y las relaciones entre ellos.
- ***Diagrama de objetos:*** Muestra instancias de clases (objetos) con valores en sus atributos y relaciones.
- ***Diagrama de casos de uso:*** Los escenarios de uso del sistema, incluyendo los roles de los usuarios.
- ***Diagramas de interacción:*** Comprende los diagramas de secuencia y de colaboración. Presenta objetos y relaciones entre ellos desde el punto de vista dinámico.
- ***Diagrama de estado:*** Representa los posibles estados, eventos y transiciones entre las clases u objetos.
- ***Diagrama de componentes:*** Organización y dependencia entre componentes físicos.
- ***Diagrama de despliegue:*** La distribución y comunicación de los componentes en los dispositivos de hardware.
- ***Diagrama de actividades:*** Muestran el flujo de trabajo desde el punto de inicio hasta el punto final, detallando muchas de las rutas de decisiones que existen en el progreso de eventos contenidos en la actividad.

La diversidad de diagramas con que cuenta UML, lo hacen superior a los ADLs por su gran expresividad gráfica, que permite definir con una sintaxis y semántica bien definida todas las etapas de desarrollo [19], [20] y todos los aspectos de la arquitectura. Elemento fundamental para lograr una comunicación entre los demás integrantes de un proyecto.

3.2.1.2 Generación de código.

Anteriormente se hizo referencia a la necesidad de que un ADL pueda generar un sistema ejecutable, para así evitar entre otras cosas, problemas de consistencia y trazabilidad entre una arquitectura y su implementación. UML es una herramienta de modelado, por tanto no proporciona soporte directo de generación de código, pero su notación gráfica a través de sus diagramas ofrece expresiones de lenguajes de programación y cuenta con herramientas que permiten que los diagramas UML puedan ser convertidos a lenguajes ejecutables. Aesop es el único ADL que lo permite y sencillamente en C++.

3.2.1.3 Soporte de lenguaje.

Aesop y Darwin lo hacen en C++ únicamente, algo que es imposible adaptar a los proyectos de la Facultad 1, ya que no es el lenguaje de programación utilizado y Rapide lo hace en VHDL, C, C++, Ada y Rapide, a pesar de ser abundantes son poco estilados. Estos sin contar que de los 3 ADLs mencionados hay uno solo que tiene interfaz gráfica e igualmente uno solo genera código en C++. Sin embargo UML soporta trabajos realizados en una serie de lenguajes, incluyendo los principales usados en la Facultad 1.

3.2.1.4 Interfaz gráfica y plataforma.

En la UCI al mismo tiempo que la mayoría de los proyectos están inmersos en una profunda investigación sobre el tema de la nueva arquitectura (SOA), se lleva a cabo el proceso lentamente de una posible migración a Linux, por lo que algunos proyectos usan Software Libre y otros privado, esto provoca la necesidad de contar con un lenguaje que sea capaz de modelar en cualquier plataforma.

Un ADL idóneo en este aspecto y el único, es Wright, pero a la vez no tiene capacidad para generar código ejecutable, tampoco interfaz gráfica, ni soporta desarrollos realizados en cualquier lenguaje. Sin embargo UML posee un sinnúmero de herramientas de desarrollo para varias plataformas.

3.2.1.4.1 Software para modelado en UML.

Estos programas están bajo licencias libres, siendo posible su libre uso, estudio y modificación.

- ArgoUML, Herramienta de modelado UML escrito en java
- BOUML, Ligera herramienta de modelado UML y generación de código C++, Java e IDL. Disponible para Windows, Unix/Linux y Mac OS X.
- Fujaba, No solo sirve para modelar sino que puede generar código Java automáticamente. También es capaz de hacer ingeniería inversa y crear los diagramas a partir del código Java.
- Dia, Puede ser usado para modelar varios tipos de diagramas UML.
- gModeler, Herramienta para modelado de UML basada en Flash, que permite generar código Action Script 2.0 Compatible.
- MonoUML, Herramienta CASE para la plataforma mono.
- Papyrus, Herramienta gráfica basada en Eclipse para el modelado con UML 2, es de código abierto y se ofrece bajo licencia EPL.
- StarUML, Herramienta de modelado para Windows desarrollada en Delphi. Bastante estable y usable.
- Umbrello, Herramienta para modelado UML para el entorno KDE.
- UMLet, Herramienta para modelado rápido de UML también escrita en Java Netbeans modulo UML.

3.2.1.4.2 Freeware para modelado en UML.

Aunque gratuitos, estos programas se encuentran bajo licencias que no permiten el estudio y modificación de los mismos.

- JUDE, Community Herramienta de modelado UML
- Omondo plugin para Eclipse. Herramienta de modelado UML para Java

- Oracle JDeveloper, Un IDE para Java con soporte de diagramas UML
- Visual Paradigm for UML, Herramienta de modelado UML y herramienta CASE que cuenta con una versión gratuita denominada Community Edition.

3.2.1.4.3 Otro software.

Software privativo para modelado.

- Borland Together
- Corel iGrafx
- Microsoft Visio
- PowerDesigner de Sybase
- Rational Rose y Rational ClearCASE de IBM
- Poseidon for UML de GentleWare
- Enterprise Architect
- MagicDraw UML

3.2.1.5 Mecanismos de UML para hacer más precisa la representación del sistema.

UML tiene principalmente tres mecanismos de extensión que permiten construir nuevos elementos o modificar la semántica de los ya existentes, para hacer más precisa la representación del sistema. Estos mecanismos son:

1. **Valores Adicionados**: Mediante estos valores es posible adicionarle nuevas propiedades o atributos a los elementos del modelo de UML.
2. **Restricciones**: Las restricciones permiten adicionar nueva semántica o modificar la existente.

3. **Estereotipos**: Los estereotipos permiten crear nuevos elementos en el modelo basados en otros ya existentes. Cada nuevo estereotipo puede reunir propiedades y restricciones particulares.

A través de estas extensiones es posible enriquecer el modelo de UML para representar adecuadamente los diferentes aspectos del sistema. Lo que da lugar a una mayor comprensión con el modelado de UML.

3.2.1.6 Proceso de desarrollo (RUP).

UML no es un método de desarrollo. No dice cómo pasar del análisis al diseño y de este al código. RUP sí es una metodología de desarrollo y aunque UML es bastante independiente del proceso de desarrollo que se siga, los mismos creadores de UML han propuesto su propia metodología de desarrollo, llamada RUP.

Los resultados de la entrevista realizada a los arquitectos de los proyectos de la Facultad 1, expresaron que RUP es la metodología más usada, por tanto esta es otra ventaja de UML sobre los ADLs.

RUP es un proceso para el desarrollo de un proyecto de un software que define claramente quien, cómo, cuándo y qué debe hacerse en el proyecto. Es una guía de cómo usar UML de la forma más efectiva.

3.2.1.7 Estructuras de arquitectura en UML.

La primera fase del proyecto es la identificación de los componentes que participan en la descripción de la arquitectura de un sistema, y luego sus relaciones en las diferentes estructuras. Para cada componente y conector se determinan los elementos de UML que los representan, con su sintaxis y semántica. Algunos componentes o estructuras no tendrán una representación directa y en este caso se utilizarán los mecanismos de extensión que provee UML, como estereotipos o restricciones.

Como parte integral de cada estructura se deben incluir restricciones adicionales que determinan las relaciones y los tipos de componentes y conectores que pueden aparecer en dicha estructura.
Por último en el proyecto se presentan las relaciones existentes entre las diferentes estructuras y la manera de verificar dichas relaciones, lo que ayudará a la persona que modela la arquitectura del sistema a validar la consistencia de esta última.

1. Casos de uso (componentes conceptuales)
Un caso de uso representa un requerimiento funcional del sistema o un proceso de negocio que se implementa en el sistema de software.

2. Actores
Un actor es una persona, sistema o dispositivo que interactúa con el sistema, iniciando, recibiendo los resultados o participando en alguna de las acciones de un caso de uso. Por lo general representa un rol, por ejemplo: jefe de contabilidad, profesor, etc.

3. Módulos
Un módulo es una división conceptual del sistema que puede ser visto como una agrupación de funciones que tengan alguna relación entre ellas y, por lo tanto, puede presentar un servicio completo al exterior una vez se ha desarrollado.

4. Clases
Una clase es la representación abstracta de un conjunto de objetos o artefactos que debe modelar el sistema. Cada clase incluye las características y el comportamiento de los objetos que representa. Una clase puede ser de tipo interfaz (interactúa con el exterior), control (realiza operaciones y controla otras clases) o entidad (hace persistentes los datos).

5. Unidades de Software

Conjunto de funciones (en programas o procedimientos) que realizan las acciones del sistema y que se implementan en archivos físicos. Las unidades de software tienen asociado un tipo (valor adicionado), que puede ser: filtro, procedimental, objetos, repositorio de datos activo u otro.

6. Sistemas externos

Un sistema externo representa un sistema de la organización que interactúa con el sistema que se está desarrollando. Por ejemplo, el sistema de contabilidad (si se está desarrollando el de recursos humanos).

7. Herramientas de software

Sistemas o programas que contribuyen al adecuado funcionamiento del sistema. Por ejemplo, el sistema operativo, un programa navegador de Internet, la máquina virtual de java, etc.

8. Procesador

Este componente representa un computador (procesador y memoria) donde se localizan programas o datos y donde, por lo general, se corren dichos programas. Este procesador puede tener roles como servidor, cliente, terminal, etc. Además, puede establecerse su ubicación física (ciudad o área de la organización) para complementar la información de este componente.

9. Dispositivo

El dispositivo es un componente o elemento de hardware que presenta una interacción con el sistema. Por ejemplo, un medidor de presión, un terminal de computadores o un módem.

Estructuras

Para el proyecto de investigación se ha desarrollado la forma de representación de ocho estructuras. Para cada una se identifican los componentes, los conectores y las restricciones que deben cumplirse. A continuación, a manera de ejemplo, se presentan dos estructuras, la estructura funcional y la estructura de llamados.

1. Estructura funcional

Esta estructura representa las funciones que ofrece el sistema a los usuarios finales, a otros sistemas o dispositivos, es decir, lo que representa el sistema para los que interactúan con él. Puede verse esta estructura como la visión conceptual del sistema, permitiendo determinar los aspectos del negocio que se desean implementar en el sistema.

Esta es una de las estructuras más importantes en un sistema, ya que ayuda a la captura de los requerimientos y se convierte en un medio de comunicación útil con los usuarios, permitiendo ver gráficamente las relaciones de los usuarios con el sistema y los procesos del negocio identificados por ellos mismos.

La estructura funcional corresponde al diagrama de casos de uso de UML, pero pueden adicionarse nuevos tipos de conectores o restricciones, que muestren relaciones de alto nivel entre los casos de uso y/o actores.

A. *Componentes:*

- Casos de uso
- Actores

B. *Conectores:*

Generalización (herencia): Indica que un componente hereda el comportamiento y atributos del otro.

- Participación: Permite relacionar a los actores con los casos de uso, indicando así que se presenta algún tipo de interacción entre el actor y el sistema a través del caso de uso.
- Inclusión: Representa que las acciones del caso de uso que recibe la relación se adicionan a las acciones del caso de uso que la inicia.
- Inclusión condicionada: Se adicionan las acciones del caso de uso que recibe la relación, pero sólo cuando se cumple una condición dada.

C. *Restricciones:*

- Entre dos casos de uso no puede presentarse simultáneamente la relación inclusión e inclusión condicionada, sólo una de ellas.
- Entre dos actores sólo puede utilizarse el conector generalización.
- El conector de participación sólo puede relacionar un actor y un caso de uso.
- Un componente (caso de uso o actor) no puede ser simultáneamente hijo y padre de otro componente usando el conector generalización
- El conector generalización no puede establecerse entre un caso de uso y un actor.

2. Estructura de llamados

En esta estructura se presentan los servicios que ofrece y los eventos que genera cada unidad de software, señalando además las relaciones de dependencia que se presentan entre estas unidades por llamar un servicio o escuchar un evento de otra unidad.

A. *Componentes:*

- Unidades de software
- Sistemas externos

B. *Conectores:*

- Servicio: Este conector representa los servicios que ofrece una unidad de software o sistema externo, y que pueden ser utilizados por otros componentes o por el usuario final mediante un llamado explícito.
- Evento: Representa los eventos que dispara una unidad de software o sistema externo, y que pueden ser escuchados por otros componentes.
- Si la generación del evento se realiza de manera condicional puede incluirse la condición en la representación gráfica.
- Uso: Este conector permite que se relacionen los componentes con los conectores servicio y evento, que presentan los otros componentes, indicando así la interacción o dependencia entre ellos. El tipo del conector

indica la forma por la cual se realiza la interacción. Este tipo puede ser pipe, llamado remoto, llamado directo, escucha trigger, escucha evento u otro.

C. *Restricciones:*

- Todo conector uso debe estar relacionado con uno y solo un conector servicio o evento.
- Todo sistema externo debe tener por lo menos un conector, ya sea servicio, evento o uso.
- Si un sistema externo tiene un conector servicio o evento, éste no puede encontrarse sin relación con un conector uso de otro componente.
- Los conectores servicio o evento pueden relacionarse con múltiples conectores uso.
- Un conector uso, que se relacione con un conector servicio sólo puede ser de tipo pipe, llamado remoto, llamado directo u otro.
- Un conector que se relaciona con un conector evento sólo puede ser de tipo escucha trigger, escucha evento u otro.

3.2.1.8 Otros elementos que convierten a UML como lenguaje candidato.

- Es un estándar.
- UML sirve para el modelado completo de sistemas complejos.
- Es independiente del lenguaje de implementación.

3.2.2 ¿Qué no puede UML?

Todas las incapacidades que se encuentren a la hora de modelar con UML pueden ser resueltas con la creación de nuestros propios componentes, la idea más lógica es llevarlas a una descripción textual, se le llama documento descriptivo, el cual contiene diferentes vistas de la arquitectura.

Conclusiones

Es fácil predecir que UML será el lenguaje de modelado de software de uso universal. Las principales razones para ello son:

- En el desarrollo han participado investigadores de reconocido prestigio.
- Ha sido apoyado por prácticamente todas las empresas importantes de informática.
- Se ha aceptado como un estándar por la OMG.
- Prácticamente todas las herramientas CASE y de desarrollo la han adaptado como lenguaje de modelado.

En resumen, UML resuelve de forma bastante satisfactoria un viejo problema del desarrollo de software como es su modelado gráfico. Además, se ha llegado a una solución unificada basada en lo mejor que hay hasta el momento, lo cual lo hace todavía más excepcional.

CONCLUSIONES

- El uso de los ADL en los proyectos de la facultad 1 es una vía eficiente para el modelado de una SOA por permitir satisfacer requerimientos descriptivos de alto nivel de abstracción.
- Se logró el cumplimiento del objetivo planteado en la investigación a través de la idea a defender, que culminó con la propuesta de un ADL para estandarizar una Arquitectura Orientada a Servicios en la facultad 1.
- UML como propuesta de solución se comporta como un ADL aunque no lo sea estrictamente. Y posee un sinnúmero de herramientas de desarrollo para varias plataformas.

RECOMENDACIONES

- Investigar sobre cada software para modelado en UML; conocer las ventajas y desventajas para poder lograr un mejor uso de UML como ADL para estandarizar SOA.

- Tratar de poner en práctica a UML cuanto antes. Probar sus alcances para el modelado de una SOA en la Facultad 1 para luego ser usado en los demás proyectos de la UCI.

- Mantener una investigación sobre los nuevos ADLs que surgen en la actualidad para poder hacer útiles modelados de sistema, modernos y fáciles a la vista de los desarrolladores.

REFERENCIAS BIBLIOGRÁFICAS

[1] **Roy Fielding.** “Architectural styles and the design of network-based software architectures”. Tesis de Doctorado. University of California at Irvine, 2000.

[2] **Mary Shaw.** “Patterns for software architecture”. First Annual Conference on the Pattern Languages of Programming, 1994.

[3] **Nenan Medvidovic**. “A classification and comparison framework for software Architecture Description Languages”. Technical Report UCI-ICS-97-02, 1996.

[4] **Steve Vestal.** “A cursory overview and comparison of four Architecture Description Languages”. Technical Report, Honeywell Technology Center, Febrero de 1993.

[5] **Jason E. Robbins**, Nenad Medvidovic, David F. Redmiles y David S. Rosenblum. “Integrating Architecture Description Languages with a Standard Design Method.” En Proceedings of the 20th International Conference on Software Engineering (ICSE’98), pp. 209-218, Kyoto, Japón, 19 al 25 de Abril de 1998.

[6] **Alexander Wolf**. “Succeedings of the Second International Software Architecture Workshop” (ISAW-2). ACM SIGSOFT Software Engineering Notes, pp. 42-56, enero de 1997.

[7] **Steve Vestal**. “A cursory overview and comparison of four Architecture Description Languages”. Technical Report, Honeywell Technology Center, Febrero de 1993.

[8] **David Luckham y James Vera**. “An Event-Based Architecture Definition Language”. IEEE Transactions on Software Engineering, pp. 717-734, Setiembre de 1995.

[9] **Mary Shaw, Robert DeLine**, Daniel Klein, Theodore Ross, David Young y Gregory Zelesnik. “Abstractions for Software Architecture and Tools to Support Them”. IEEE Transactions on Software Engineering, pp. 314-335, Abril de 1995.

[10] **Mary Shaw y David Garlan**. “Characteristics of Higher-Level Languages for Software Architecture”. Technical Report CMU-CS-94-210, Carnegie Mellon University, Diciembre de 1994.

[11] **Nenan Medvidovic.** “A classification and comparison framework for software Architecture Description Languages”. Technical Report UCI-ICS-97-02, 1996.

[12] **David Parnas.** “On the Design and Development of Program Families.” IEEE Transactions on Software Engineering SE-2, pp. 1-9, Marzo de 1976.

[13] **Jeff Magee y Jeff Kramer.** “Dynamic structure in software architectures”. En Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 3–14, San Fransisco, Octubre de 1996.

[14] **Jeff Magee, Naranker Dulay**, Susan Eisenbach y Jeff Kramer. “Specifying distributed software architectures”. En Proceedings of the Fifth European Software Engineering Conference, ESEC’95, Setiembre de 1995.

[15] **Robert Allen.** “A formal approach to Software Architecture”. Technical Report, CMU-CS-97-144, 1997.

[16] **Maarten Boasson.** The Artistry of Software Architecture. IEEE Software, Vol. 12, No. 6, November 1995 (pp. 13-16).

[17] **Mary Shaw y David Garlan.** Software Architecture Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[18] **Len Bass, Paul Clements y Rick Kazman.** Software Architecture in Practice. Sei Series In Software Architectures. Addison Websley, 1998.

[19] **Jason E. Robbins, Nenad Medvidovic, David F. Redmiles y David S. Rosenblum**. Integrating Architec ture Description Languages with a Standard Design Method. 20th International Conference on software Engineering, 1998.

[20] **Grady Booch, James Rumbaugh e Ivar Jacobson.** The Unified Modeling Language User Guide. Rational Software Corporation. Addison-Wesley, 1999.

BIBLIOGRAFÍAS

BILLY REYNOSO. Introducción a la Arquitectura de Software. UNIVERSIDAD DE BUENOS AIRES.2005

Disponible en: download.microsoft.com/download/4/F/F/4FF88340-43CC-4C5B-8E50-09002969D0DD/20051122-ARC-BA.ppt

D. C. LUCKHAM and J. VERA. Architecture Description Languages (ADLs).2005

Disponible en: http://sunset.usc.edu/classes/cs578_2005/February8.pdf

D. GARLAN and A. J. KOMPANEK. UML and architecture. 2005.

Disponible en: http://sunset.usc.edu/classes/cs578_2005/March1.pdf

HERNÁNDEZ LEÓN ROLANDO ALFREDO-COELLO GONZÁLEZ SAYDA. El paradigma científico de la investigación científica. Universidad de Ciencias Informáticas.

JOSÉ A. TROYANO, MANUEL MEJÍAS, JESUS TORRES Y MIGUEL TORO. Extensiones al Sistema de Clasificaciones de UML. Departamento de Lenguajes y sistemas Informáticos. Universidad de Sevilla. España. 2003

Disponible en: http://www.cic.ipn.mx/revistas/pages/vol03-03/CYS03305.pdf

LIDIA FUENTES y ANTONIO VALLECILLO. Una Introducción a los Perfiles UML. Depto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga Campus de Teatinos. E29071- Málaga (SPAIN). **Disponible en:** http://www.lcc.uma.es/~av/Publicaciones/04/UMLProfiles-Novatica04.pdf

JUAN BERNARDO. Seminario de Arquitectura de Software. Universidad de Antioquia. 2007

Disponible en:

http://www.aprendeenlinea.udea.edu.co/lms/moodle/file.php/120/Programa_200701.pdf

LIDIA FUENTES, NADIA GÁMEZ y MÓNICA PINTO. DAOPxADL: UNA EXTENSIÓN DEL LENGUAJE DE DESCRIPCIÓN DE ARQUITECTURAS xADL CON ASPECTOS *. Departamento de Lenguajes y Ciencias de la Computación

E.T.S.I. Informática Universidad de Málaga. 2006

Disponible en:

http://cmapspublic3.ihmc.us/servlet/SBReadResourceServlet?rid=1204146008425_784578660_3101

CARLOS E. CUESTA. Universidad Rey Juan Carlos. 2005

Disponible en:

http://www.kybele.escet.urjc.es/documentos/ISI/Arquitecturas%20de%20SW.pdf

ALBERTO MOLPECERES. Procesos de desarrollo: XP, RUP y FDD. 2003

Disponible en: http://www.javahispano.org/licencias/

ANEXOS

Entrevista realizada a arquitectos de los proyectos productivos de la Facultad 1 vinculada con el uso de los ADLs.

Tema: ADLs

Nombre del proyecto: ______________________________

Nombre del arquitecto: ____________________________

1-Tipo de aplicación que desarrollan:

a) ____Gestión

b) ____Web

c) ____Procesamiento de imágenes

d) ____Otra ¿Cuál? ____________________

2-Metodología que utiliza:

a) ____XP

b) ____MSF

c) ____RUP

d) ____Otra ¿Cuál?

3-Tipo de estilo arquitectónico:

a) ____Estilo de Flujo de Datos.
b) ____Estilo Centrado en Datos.
c) ____Estilo de Llamada y Retorno.
d) ____Estilo de Código Móvil.
e) ____Estilo Peer To Peer (Punto a Punto).

4- ¿Su proyecto usa Arquitectura Orientada a Servicios (SOA)?

a) ____Si

b) ____No

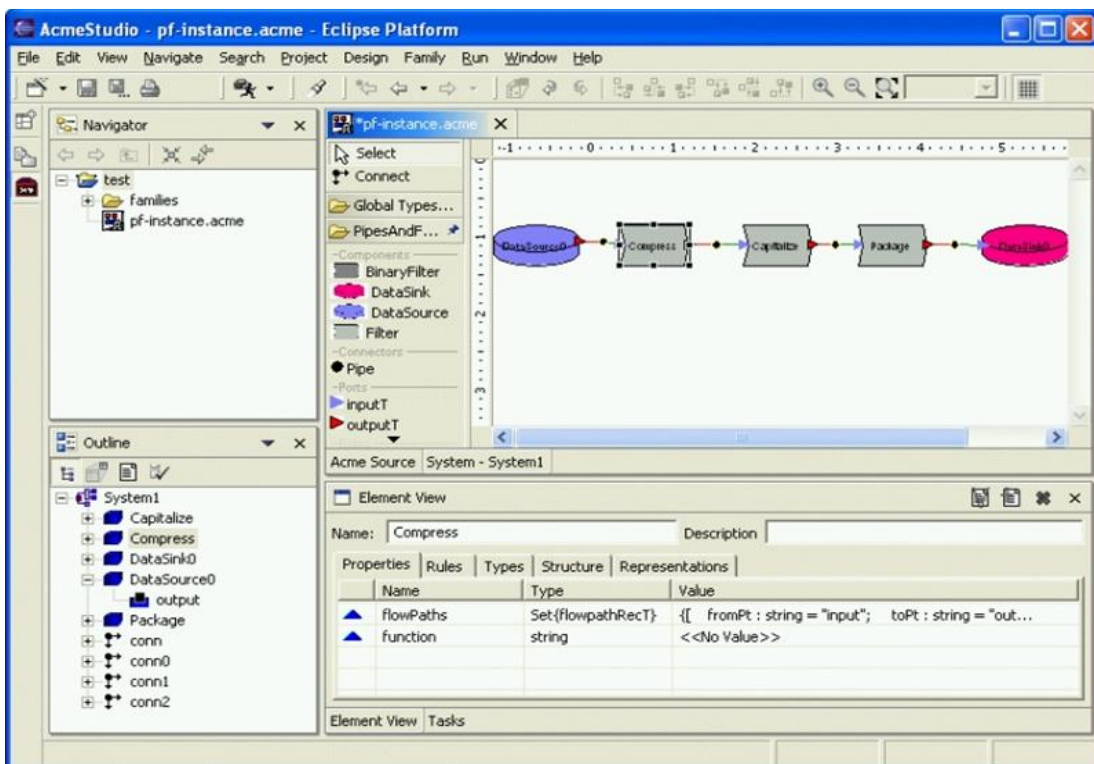
c) ____Está en planes.

5- ¿Utilizan algún Lenguaje de Descripción de Arquitectura (ADL)?

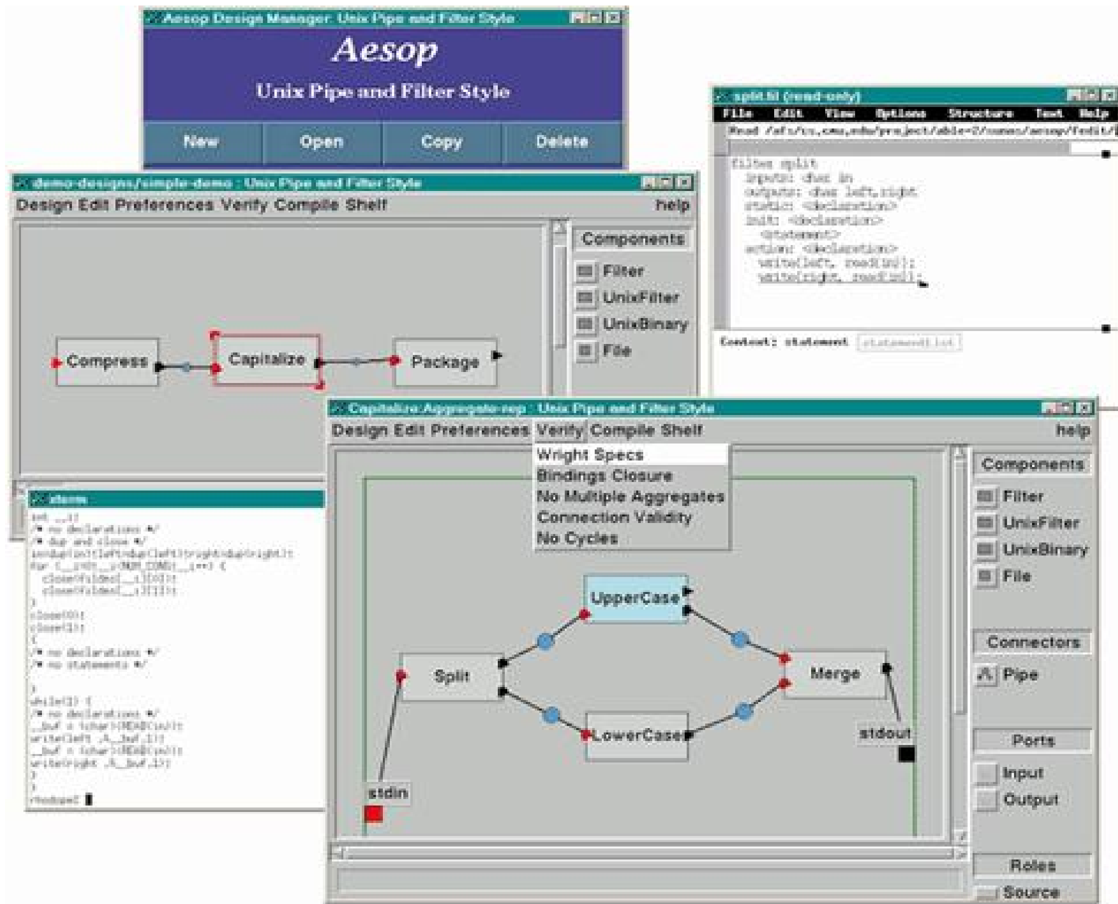
a) ____Si

b) ____No

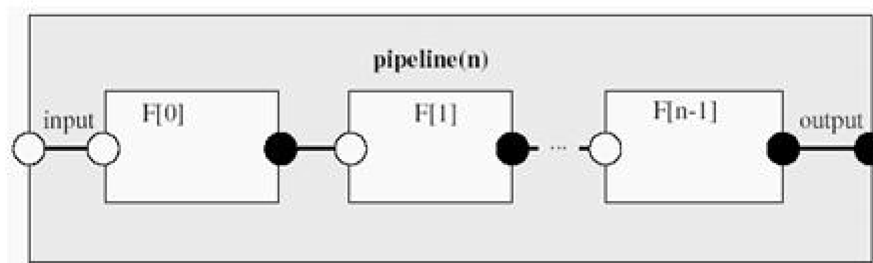
c) ____ ¿Cuál?



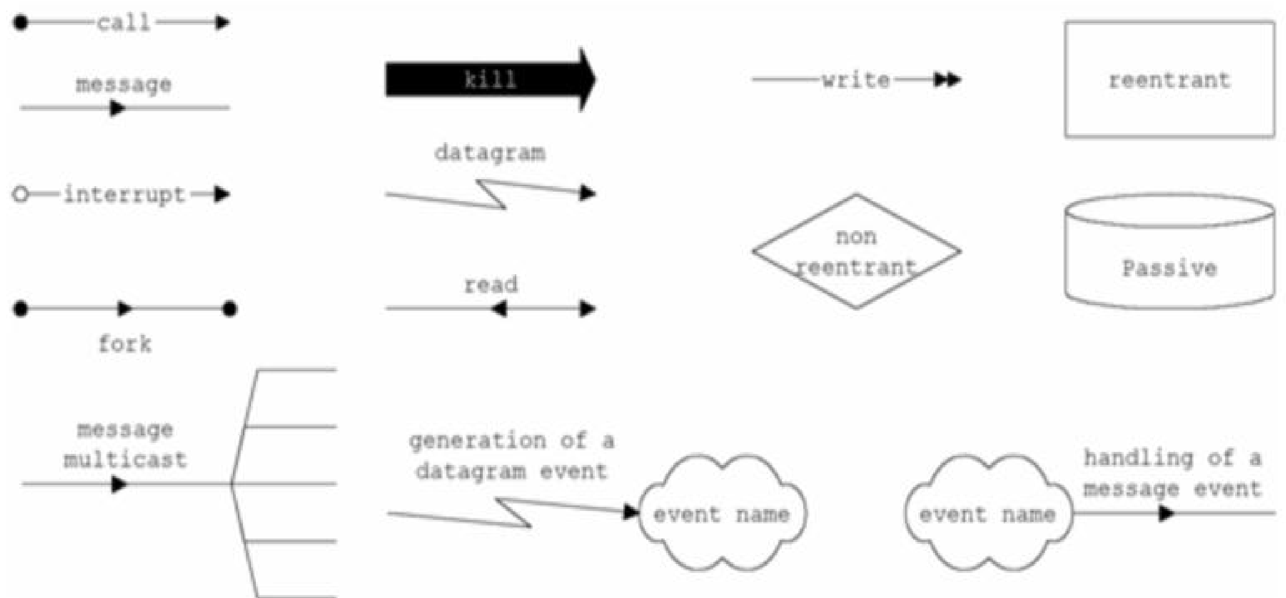
Anexo 1. Ambiente de edición de AcmeStudio con diagrama de tuberías y filtros.



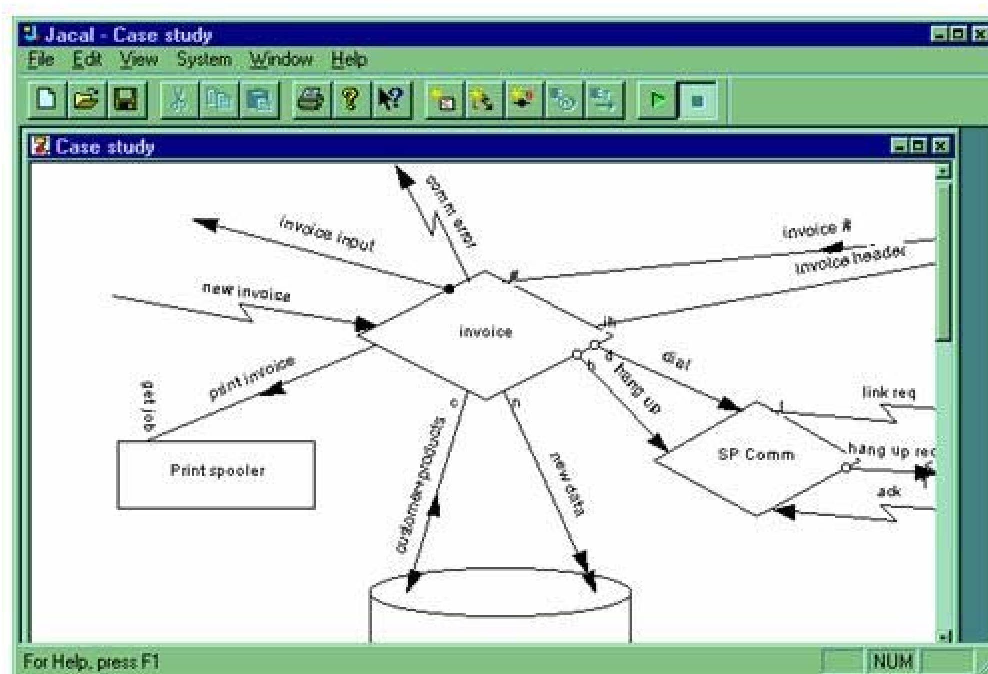
Anexo 2. Ambiente gráfico de Aesop con diagrama de tuberías y filtros.



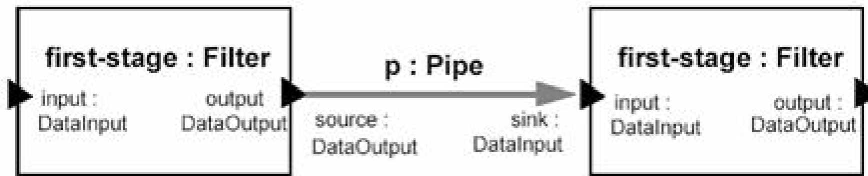
Anexo 3. Diagrama de tuberías en Darwin.



Anexo 4. Tipos de componentes y conectores.



Anexo 5. Estudio de caso en Jacal.



Anexo 6. Diagrama correspondiente al código en Wright.

	Generación de código	**Plataforma**	**Soporte de lenguajes**	**ADL?**	**Interfaz gráfica**
Acme-Armani	-	Windows	-	No	AcmeStudio
Aesop	C++	-	C++	Si	Desconocida
Darwin	-	Windows	C++	Si	SAA
Jacal	-	Win32	-	Si	Jacal
Rapide	-	Solaris 2.5, SunOS 4.1.3 y Linux	VHDL, C, C++, Ada y Rapide	Si	-
Wright	-	Windows y Linux	-	Si	-
UML*	Herramientas que lo permiten.	Sus herramientas corren en varias plataformas.	Varios	No	Abundantes herramientas.

Anexo 7. Comparación entre ADLs.

GLOSARIO

Binding: es una "ligadura" o referencia a otro símbolo más largo y complicado, y que se usa frecuentemente. Este otro símbolo puede ser un valor de cualquier tipo, numérico, de cadena, etc. o el nombre de una variable que contiene un valor o un conjunto de valores.
En el campo de la programación, un binding es una adaptación de una biblioteca para ser usada en un lenguaje de programación distinto de aquél en el que ha sido escrita.

Estandarización o normalización: es la redacción y aprobación de normas que se establecen para garantizar el acoplamiento de elementos construidos independientemente, así como garantizar el repuesto en caso de ser necesario, garantizar la calidad de los elementos fabricados y la seguridad de funcionamiento.

Framework: es una estructura de soporte definida en la cual otro proyecto de software puede ser organizado y desarrollado. Típicamente, un framework puede incluir soporte de programas, bibliotecas y un lenguaje interpretado entre otros software para ayudar a desarrollar y unir los diferentes componentes de un proyecto.
Un framework representa una arquitectura de software que modela las relaciones generales de las entidades del dominio. Provee una estructura y una metodología de trabajo la cual extiende o utiliza las aplicaciones del dominio.

Front-end: es la parte del software que interactúa con el o los usuarios y el **back-end** es la parte que procesa la entrada desde el front-end.

Interfaz: es el conjunto de comandos y/o métodos que permiten la intercomunicación del programa con cualquier otro programa, entre partes del propio programa o elemento interno o externo. De hecho, los periféricos son controlados por interfaces.
Ejemplo. El teclado de un teléfono sería una interfaz de usuario, mientras que la clavija sería la interfaz que permite al teléfono comunicarse con la central telefónica.

Multicast (multidifusión): es el envío de la información en una red a múltiples destinos simultáneamente, usando la estrategia más eficiente para el envío de los

mensajes sobre cada enlace de la red sólo una vez y creando copias cuando los enlaces en los destinos se dividen.

Plataforma: se refiere al sistema operativo o a sistemas complejos que a su vez sirven para crear programas, como las plataformas de desarrollo.

Procesos de negocio: es un conjunto de tareas relacionadas lógicamente, llevadas a cabo para lograr un resultado de negocio definido. Cada proceso de negocio tiene sus entradas, funciones y salidas. Las entradas son requisitos que deben tenerse antes de que una función pueda ser aplicada. Cuando una función es aplicada a las entradas de un método, tendremos ciertas salidas resultantes.

Es una colección de actividades estructurales relacionadas que producen un valor para la organización, sus inversores o sus clientes. Es, por ejemplo, el proceso a través del que una organización ofrece sus servicios a sus clientes.

Red de Petri: es una representación matemática de un sistema distribuido discreto. Las redes de Petri fueron definidas en los años 1960 por Carl Adam Petri. Son una generalización de la teoría de autómatas que permite expresar eventos concurrentes.
Una red de Petri está formada por lugares, transiciones y arcos dirigidos, así como por fichas que ocupan posiciones. Los arcos conectan un lugar a una transición o una transición a un lugar. No puede haber arcos entre lugares ni entre transiciones. Los lugares contienen un número cualquiera de fichas. Las transiciones se disparan, es decir consumen fichas de una posición de inicio y producen fichas en una posición de llegada. Una transición está habilitada si tiene fichas en todas sus posiciones de entrada.

Servicios Web: es un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferente y ejecutada sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes de ordenadores como Internet.