VNIVERSIDAD NACIONAL AVTÓNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

ANÁLISIS DE ARQUITECTURAS DE SOFTWARE EN AMBIENTES DE DESARROLLO ÁGIL

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRA EN INGENIERÍA (COMPUTACIÓN)

P R E S E N T A:

MARÍA EVELIA CASALES CABRERA

DIRECTORA DE TESIS:
M. EN C. MA. GUADALUPE ELENA IBARGÜENGOITIA GONZÁLEZ

México, D.F. **2011.**

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Agradecimientos

A la M. en C. Ma. Guadalupe Elena Ibargüengoitia por sus enseñanzas, paciencia, confianza y apoyo.

Al jurado integrado por la Dra. Hanna Oktaba, M. en C. Gustavo Arturo Márquez Flores, M. en C. Gustavo Adolfo Arellano Sandoval y M. en C. Alfonso Martínez Martínez por los comentarios y sugerencias que hicieron de este un mejor trabajo.

A CONACYT y a la Coordinación de Estudios de Posgrado por el apoyo económico.

A Esmeralda Hernández por su apoyo, ayuda incondicional en los momentos de mayor tensión y su amistad.

A Rubén Murguía y Omar Soto por su amistad y compañerismo.

A mis padres y hermana por su apoyo, comprensión y por estar presentes en mi vida.

Índice

Introducción

En la actualidad existe dentro del área de Ingeniería de Software un gran interés dentro de las metodologías ágiles debido al éxito que han tenido dentro de proyectos con requerimientos muy cambiantes y cuando se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. Para adaptarse a este tipo de ambientes de desarrollo, estas metodologías hacen hincapié en la adopción de los valores y principios ágiles que promueven.

La mayoría de las decisiones que se toman dentro de las metodologías ágiles son basadas en la experiencia del equipo de desarrollo, es por ello que existe cierto escepticismo en su implantación.

Recientemente la mayoría de las metodologías ágiles han incrementado su interés por formalizar las arquitecturas de software debido a la importancia que tienen como factor de éxito en el desarrollo de software independientemente de la metodología que se utilice. Ejemplos de esto se encuentra la propuesta de integración de los métodos centrados en la arquitectura del *Software Engineering Institute* (SEI) en las metodologías ágiles como lo explica Robert Nord en [1], entre otras muchas iniciativas.

La mayoría de las propuestas ágiles que proponen actividades relacionadas con la arquitectura se enfocan en la organización del equipo de desarrollo y en el ofrecimiento de las experiencias que los autores han tenido al desarrollar un proyecto en ambientes ágiles integrando actividades relacionadas con la arquitectura. Es hasta últimas fechas que se ha tratado de formalizar todas estas experiencias de desarrolladores como en el trabajo que propone James Coplien y Gertud Bjørnvig en [2].

Partiendo de lo anterior se plantea como mejora incremental una guía que apoye en el proceso de evolución de arquitecturas en desarrollos ágiles que integre el razonamiento consiente de las actividades relacionadas con el desarrollo de la arquitectura de software para desarrollar proyectos de software exitosos.

Para lograr lo anterior se realizó una investigación bibliográfica profunda para entender los conceptos relacionados con la arquitectura de software y proponer una guía que integrara ese conocimiento fundamentándose en los valores y principios ágiles, y en la experiencia que se tuvo en el desarrollo del caso de estudio.

El producto del trabajo e investigación realizados se expone en los capítulos 1 a 5 de este documento.

El Capítulo 1 explica brevemente el contexto de los requerimientos de software, enfocándose en los requerimientos no funcionales. En el Capítulo 2 se explican los conceptos relacionados con las arquitecturas de software que se consideran dentro de la guía como su definición, la importancia de los atributos de calidad dentro del diseño arquitectónico, su influencia en el diseño de software, la diferencia entre tareas arquitectónicas, diseño e implementación, el ciclo de desarrollo de una arquitectura de software y su relación con los patrones y los estilos

arquitectónicos, entre otros conceptos. En el Capítulo 3 se expone brevemente el contexto en el que surgen las metodologías ágiles, así como los valores y principios que promueven, las metodologías actuales y su comparación. En el Capítulo 4 se encuentra la guía propuesta la cual hace referencia a los capítulos anteriores. Partiendo de la guía se expone un caso de estudio en el Capítulo 5. Se considera que los capítulos 4 y 5 son los resultados visibles de este trabajo. Finalmente se muestran las conclusiones de este trabajo de tesis y el trabajo futuro que puede desprenderse de éste.

Problemática

Importancia de las Arquitecturas de Software

Las arquitecturas de software son importantes para el desarrollo de software independientemente de la metodología o proceso que se siga para su desarrollo. Se considera a las arquitecturas de software como un indicador clave de calidad y éxito en los proyectos.

Los sistemas con arquitecturas pobres o no bien definidas tienden a ser difíciles de construir y mantener, usan estrategias inconsistentes de integración, y es difícil tener todos los diferentes componentes trabajando conjuntamente. En general, se considera que estos sistemas pueden tener muchos problemas si los equipos de desarrollo siguen prácticas de construcción pobres de acuerdo a la experiencia de desarrolladores e investigadores como los trabajos de [3] y [4].

El diseño de las arquitecturas de software es uno de los pilares dentro del desarrollo de software, algunas de las razones de su importancia [5] se resumen en los siguientes puntos:

- Las arquitecturas de software son de especial importancia ya que la manera en que se estructura un sistema tiene impacto directo sobre la capacidad de éste para satisfacer los atributos de calidad del sistema.
- Las arquitecturas de software juegan un papel fundamental para guiar el desarrollo, ya que al conocer la estructura de un sistema se permite realizar la división en componentes que serán asignados a individuos o grupos de individuos, permitiendo de esta manera apoyar a las tareas de administración del proyecto, independientemente del proceso de desarrollo que se utilice.
- Otro valor importante que se obtiene de los diseños arquitectónicos que se crean en una organización, es que pueden ser reutilizados para crear distintos sistemas, permitiendo reducir costos y aumentar la calidad.

Arquitecturas de Software y Metodologías Ágiles

Como se ha mencionado anteriormente, el diseño de la arquitectura de software es uno de los pilares dentro del desarrollo de software. Aunque muchos precursores de las metodologías ágiles consideran que sus actividades generan mucha documentación y que se requiere de un diseño muy detallado desde el inicio, se recomienda realizar actividades orientadas al diseño arquitectónico sobre todo en aquellos sistemas que poseen una gran dificultad en su desarrollo, ya sea porque son muy grandes, con funcionalidades muy complicadas de lograr o existe la presencia de un equipo de desarrollo distribuido, entre otras causas. Es entonces necesario considerar agilizar las actividades de diseño de la arquitectura para evitar el colapso, debido a un enfoque arquitectónico pobre.

¿Cómo Integrar las Decisiones Relacionadas a la Arquitectura de Software durante las Iteraciones del Proceso de Desarrollo Ágil?

Al inicio del primer ciclo de desarrollo de un sistema en un ambiente ágil no se tienen suficientemente claros los requerimientos ni el alcance del sistema, se cuenta con muy pocos elementos para definir una arquitectura de software definitiva, por lo que generalmente se define una arquitectura basada en las experiencia del equipo de desarrollo y en la poca información que proporciona el cliente/usuario. Dentro de esta información preliminar están algunos de los requerimientos no funcionales o características que el software deberá cumplir.

Al avanzar en el proyecto es necesario evaluar el cumplimiento de los requerimientos no funcionales relacionados a la arquitectura para comprobar que sigue siendo la apropiada para el sistema de acuerdo a la información que va surgiendo durante los ciclos posteriores de desarrollo [6].

En caso de que no se cumpla con los requerimientos en ciclos posteriores, de acuerdo a las metodologías ágiles se "*refactoriza*" de tal manera que se garantice su cumplimiento, en la medida de lo posible.

Debido a la naturaleza de las metodologías ágiles no hay un método formal que integre la evolución de las arquitecturas de software en el ciclo de desarrollo de un proyecto, como se mencionó anteriormente generalmente se hace basándose en los conocimientos empíricos de los desarrolladores, teniendo como desventajas que el conocimiento adquirido en los proyectos se puede perder al desintegrarse el equipo debido a múltiples causas.

Partiendo de todo lo anterior existe la necesidad de guías que ayuden en los desarrollos ágiles a evolucionar las decisiones arquitectónicas importantes, además de que sirvan a los equipos de desarrollo como referencias independientemente de la experiencia que tengan.

Objetivo

Se propone realizar una guía que permita integrar la evolución de los requerimientos funcionales, atributos de calidad y restricciones dentro de las arquitecturas de software conforme se avanza en ciclos de desarrollo en un ambiente ágil, incluyendo actividades tales como la documentación y la evaluación de dichas arquitecturas, apegándose a los valores y principios que promueven las metodologías ágiles.

Las características a analizar a lo largo de la guía propuesta son los atributos de calidad, ya que es a partir de éstos que se desarrollan las arquitecturas de software de los sistemas bajo cualquier proceso de desarrollo.

Capítulo 1. Requerimientos de Software

1.1 Definición e Importancia de los Requerimientos de Software

De las muchas definiciones que existen para requerimiento, a continuación se presentan algunas:

a) La IEEE menciona que un "Requerimiento" es una condición o necesidad de un usuario para resolver un problema o alcanzar un objetivo.

b) Un "Requerimiento" es una condición o capacidad que debe estar presente en un sistema o componentes de sistema para satisfacer un contrato, estándar, especificación u otro documento formal.

c) Un "Requerimiento" es una representación documentada de una condición o capacidad como en (a) o (b).

Los Requerimientos son la pieza fundamental en un proyecto de desarrollo de software, partiendo de ellos es posible:

Planear el proyecto y los recursos que se usarán en él. Los líderes de proyecto usan los requerimientos como una base para la estimación del esfuerzo necesario en un proyecto. Especificar el tipo de verificaciones que se habrán de realizar al sistema. Por ejemplo: cuando se está tratando de alinearse a cierta norma oficial o estándar.

Planear la estrategia de prueba a la que habrá de ser sometido el sistema. Los requerimientos son la base sobre la cual se decide si un caso de prueba fue ejecutado exitosamente por el sistema o no.

Son el fundamento del ciclo de vida del proyecto. Los requerimientos documentados son la base para crear la documentación del sistema. De ahí su importancia y la importancia de que deban de ser definidos y manejados de la forma más adecuada posible.

1.2 Características de los Requerimientos

Las características de un requerimiento son sus propiedades principales. Un conjunto de requerimientos en estado de madurez, deben presentar una serie de características tanto individualmente como en grupo. A continuación se presentan las más importantes:

- **Necesario:** Un requerimiento es necesario si su omisión provoca una deficiencia en el sistema a construir, y además su capacidad, características físicas o factor de calidad no pueden ser reemplazados por otras capacidades del producto o del proceso.
- **Conciso:** Un requerimiento es conciso si es fácil de leer y entender. Su redacción debe ser simple y clara para aquellos que vayan a consultarlo en un futuro.
- **Completo:** Un requerimiento está completo si no necesita ampliar detalles en su redacción, es decir, si se proporciona la información suficiente para su comprensión.

- **Consistente:** Un requerimiento es consistente si no es contradictorio con otro requerimiento.
- **No ambiguo:** Un requerimiento no es ambiguo cuando tiene una sola interpretación.
- **Verificable:** Un requerimiento es verificable cuando puede ser cuantificado de manera que permita hacer uso de los siguientes métodos de verificación: inspección, análisis, demostración o pruebas.

1.3 Tipos de Requerimientos

Los Requerimientos puedes dividirse en Requerimientos Funcionales y Requerimientos No Funcionales [7].

Los **Requerimientos Funcionales** definen las funciones que el sistema será capaz de realizar. Describen las transformaciones que el sistema realiza sobre las entradas para producir salidas.

Los **Requerimientos No Funcionales** son aquellos que no se refieren directamente a las funciones específicas que entrega el sistema, sino a las propiedades emergentes de éste como la fiabilidad, la respuesta en el tiempo y la capacidad de almacenamiento. De forma alternativa, definen las restricciones del sistema, como la capacidad de los dispositivos de entrada/salida y la representación de datos que se utiliza en las interfaces del sistema. Sin embargo, estos requerimientos no siempre se refieren específicamente al sistema de software a desarrollar.

Muchos requerimientos no funcionales se refieren al sistema como un todo más que a rasgos particulares del mismo. Esto significa que a menudo son más críticos que los requerimientos funcionales particulares. Mientras que el incumplimiento de los funcionales degradará el sistema, una falla en un requerimiento no funcional del sistema lo inutiliza.

Los requerimientos no funcionales surgen de la necesidad del usuario, debido a las restricciones en el presupuesto, a las políticas de la organización, a la necesidad de interoperabilidad con otros sistemas de software o hardware o a factores externos como los reglamentos de seguridad, las políticas de privacidad, etcétera.

1.4 Clasificación de los Requerimientos No Funcionales

Los Requerimientos No Funcionales a su vez pueden clasificarse de acuerdo con sus implicaciones de la siguiente manera [7]:

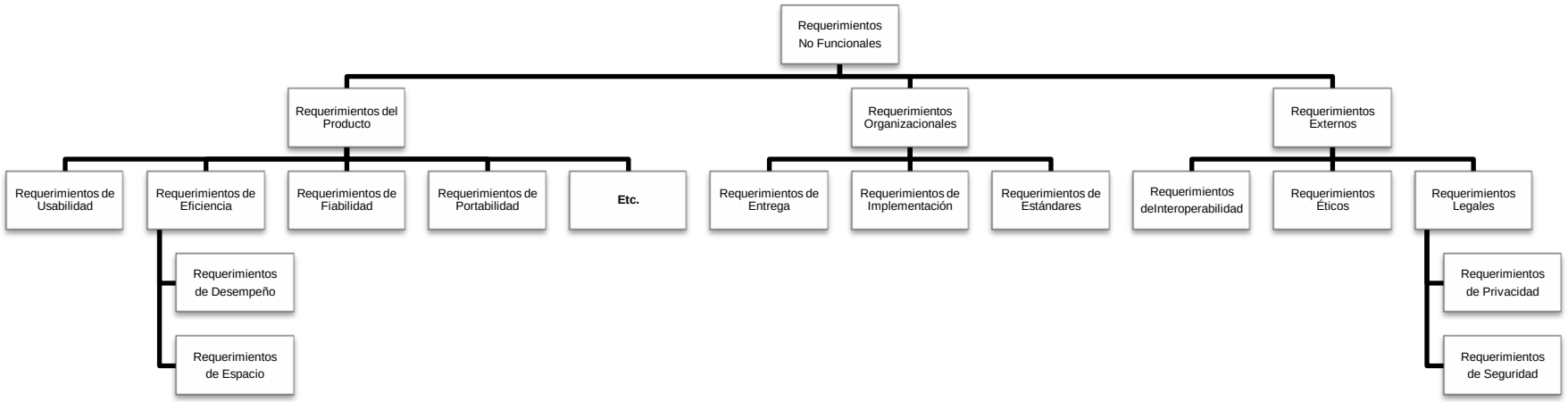

Figura 1. **Clasificación de los Requerimientos No Funcionales [7]**

Requerimientos del producto. Especifican el comportamiento del producto; como los requerimientos de desempeño en la rapidez de ejecución del sistema y cuánta memoria se requiere; los de fiabilidad que fijan la tasa de fallas para que el sistema sea aceptable; los de portabilidad y los de usabilidad.

Requerimientos organizacionales. Se derivan de las políticas y procedimientos existentes en la organización del cliente y en la del desarrollador: estándares en los procesos que deben utilizarse; requerimientos de implementación como los lenguajes de programación o el método de diseño a utilizar, y los requerimientos de entrega que especifican cuándo se entregará el producto y su documentación.

Requerimientos externos. Se derivan de los factores externos al sistema y de su proceso de desarrollo. Incluyen los requerimientos de interoperabilidad que definen la manera en que el sistema interactúa con los otros sistemas de la organización; los requerimientos legales que deben seguirse para asegurar que el sistema opere dentro de la ley, y los requerimientos éticos. Estos últimos son impuestos al sistema para asegurar que será aceptado por el usuario y por el público en general.

Otra clasificación de los Requerimientos No Funcionales propone los **Atributos o Características de Calidad** que son las propiedades del sistema que definen la manera en la que se comportarán los Requerimientos Funcionales y que hacen a un sistema bueno o malo desde una perspectiva técnica.

Por otra parte se encuentran las **Restricciones** y se diferencian de los Atributos de Calidad en que no están ligados directamente con las propiedades del producto y se derivan del dominio del sistema más que de las necesidades específicas de los usuarios, como el presupuesto, las políticas de la organización, etc.

Capítulo 2. Arquitecturas de Software

2.1 Introducción a las Arquitecturas de Software

2.1.1 Diseño de Software

El diseño de software es una actividad realizada por un desarrollador de software que resulta en la arquitectura de software de un sistema. Está relacionada con la especificación de los componentes de un sistema de software y las relaciones entre ellos dados ciertas propiedades funcionales y no funcionales.

Convencionalmente se usan los términos de "arquitectura de software", "diseño de arquitectura de software", o "diseño general" a la subdivisión de alto nivel estructural del sistema, y "diseño" o "diseño detallado" a una planeación más detallada. Se denota a toda la actividad de construcción del sistema de software como "diseño del software" y los artefactos resultantes "arquitectura de software".

Muchos desarrolladores actualmente prefieren el término "arquitectura de software" en vez de "diseño del software" para denotar a todos los artefactos que resultan de las actividades de diseño. Al hacerlo, se expresa el hecho de que no sólo se descompone la funcionalidad del sistema en un conjunto cooperativo de componentes, sino más bien se construye una arquitectura de software. Se trata de mostrar que debe centrarse explícitamente en la construcción apropiada de los componentes del sistema de software, en el que se incluyan sus responsabilidades, su funcionalidad e interfaces, sus estructuras internas, sus múltiples relaciones entre ellos y la forma en que colaboran, todo esto bajo consideración de las propiedades no funcionalidades como la cambiabilidad y la portabilidad. Varios desarrolladores no están de acuerdo en que las decisiones de alto nivel se puede hacer independientemente de las decisiones de bajo nivel [8].

2.1.2 ¿Qué es una Arquitectura de Software?

Actualmente no existe una definición estándar sobre Arquitectura de Software, sin embargo el *Software Engineering Institute* ofrece múltiples definiciones creadas por un gran grupo de autores e investigadores del área dentro de su sitio web[1], algunas de las definiciones más notables son las siguientes:

1. La arquitectura de software de un programa ó de un sistema de cómputo, es la estructura o estructuras de un sistema, que comprende a los elementos del software, las propiedades visibles de forma externa de estos elementos y las relaciones entre ellos [9].
   El término "elementos" dentro de esta definición es vago a propósito, pues puede referirse a distintas entidades relacionadas con el sistema. Los elementos pueden ser entidades que existen en tiempo de ejecución (objetos, hilos), entidades lógicas que existen en tiempo de desarrollo (clases, componentes) y entidades físicas (nodos, directorios).

[1] http://www.sei.cmu.edu/architecture/start/definitions.cfm

La "propiedades externamente visibles" se refieren a aquellas hipótesis que unos elementos hacen de otros elementos, como: proveer servicios, características de desempeño en la ejecución, manejo de fallas, uso de recursos compartidos, entre otros.
Esto significa que las relaciones entre elementos dependen de propiedades visibles (o públicas) de los elementos, quedando ocultos los detalles de implementación. Finalmente, cada conjunto de elementos relacionados de un tipo particular corresponde a una estructura distinta, de ahí que la arquitectura está compuesta por distintas estructuras.

2. En el estándar ANSI/IEEE 1471 - 2000 "Prácticas Recomendadas para la Descripción Arquitectónica de Sistemas de Software - Intensivo", define a la Arquitectura como la organización fundamental de un sistema, representada en sus componentes, las relaciones entre ellos y el ambiente, y los principios que gobiernan su diseño y evolución.
3. La Arquitectura de Software de un sistema o de un conjunto de sistemas consiste en aquellas decisiones importantes de diseño acerca de la estructura del software y de las interacciones entre las estructuras que comprenden a los sistemas. Estas decisiones de diseño apoyan a un conjunto de características de calidad que un sistema debe tener para ser exitosos. Las decisiones de diseño proveen de una base conceptual para el desarrollo, soporte y mantenimiento del sistema [10].

2.1.3 Diferencia entre Arquitectura, Diseño e Implementación

Para explicar la relación entre arquitectura, diseño e implementación el SEI propone su "Tesis de la Intensión/Localidad"[2], menciona que se debe distinguir entre dos interpretaciones:

1. **Intensional (vs. Extensional).** En lógica, filosofía del lenguaje y otras disciplinas que estudian los signos y el significado, la intensión de una expresión es su significado o connotación, en contraste con la extensión de la misma, que consiste en las entidades a las cuales la expresión se aplica. Una definición intensional de "soltero (a)" es "persona no casada", siendo una persona no casada una propiedad esencial para referirse a soltero. Es una condición necesaria, ya que no se puede ser soltero sin ser una persona no casada. Es una condición suficiente ya que cualquier persona no casada es soltera. Una definición extensional de "soltero(a)" sería una lista de todas aquellas personas en el mundo que no son casadas.[3]

   Las especificaciones son intensionales cuando su definición es suficiente para aplicarla ilimitadamente en más de una implementación. Una especificación es extensional al tener una serie de características particulares en cierta implementación.

[2] http://www.sei.cmu.edu/library/abstracts/news-at-sei/architect1q03.cfm

[3] http://en.wikipedia.org/wiki/Intensional_definition, http://en.wikipedia.org/wiki/Extensional_definition, http://es.wikipedia.org/wiki/Extensi%C3%B3n_%28sem%C3%A1ntica%29, http://es.wikipedia.org/wiki/Definici%C3%B3n_intensional

Las especificaciones de diseño intensionales son "abstractas" en el sentido de que pueden ser formalmente caracterizadas por el uso de variables lógicas en un dominio ilimitado. Por ejemplo: un patrón arquitectónico de capas no restringe al arquitecto un número de capas; se aplica igualmente bien para 2 o 12 capas. En su lado opuesto las especificaciones de diseño extensionales restringen el dominio de las variables lógicas, es decir son particulares a una implementación.

2. **No Local (vs. Local).** Las especificaciones no locales son "abstractas" en el sentido de que se pueden aplicar en todas las partes del sistema. En las locales son limitadas ya que sólo se pueden aplicar en ciertas partes del sistema.

Ambas interpretaciones contribuyen a distinguir entre arquitectura, diseño e implementación resumido en la "Tesis de la Intensión/Localidad" de la siguiente manera:

1. Las especificaciones arquitectónicas son intensionales y no locales.
2. Las especificaciones de diseño son intensionales pero locales.
3. Las especificaciones de implementación son extensionales y locales.

La siguiente tabla resume estas distinciones:

Tesis de la Intensión/Localidad		
Arquitectura	Intensional	No Local
Diseño	Intensional	Local
Implementación	Extensional	Local

Tabla 1. Tesis de la Intensión/Localidad

Consideremos el concepto estrictamente de una arquitectura de capas (es un estilo arquitectónico en el que a cada capa se le permite el uso sólo de la capa inmediata inferior). Para saber si es un estilo arquitectónico debemos preguntarnos si realmente es un estilo intensional y si es local ó no local:

- Primero, ¿Existen un número ilimitado de implementaciones que califiquen como capas?, claramente las hay, entonces es un estilo intensional.
- Segundo, ¿El estilo de capas es local ó no local?, para contestar esta pregunta necesitamos considerar alguna violación para este estilo, en el que una capa depende de la capa superior, o de varias capas por debajo. Dado que estas suposiciones serían una violación al estilo arquitectónico, se considera entonces que es un estilo no local.

Por lo tanto al ser intensional y no local este estilo es realmente arquitectónico.

Ahora consideremos un patrón de diseño, por ejemplo el patrón "*Factory*". Es intensional puesto que puede haber un número ilimitado de implementaciones del patrón de diseño "*Factory*" en un sistema. ¿Es local ó no local?, se puede utilizar el patrón de diseño en partes específicas del sistema, ó simplemente no usarlo, e inclusive se puede violar en una parte diferente de ese mismo sistema, por lo tanto es local. Se puede decir entonces que si es un patrón de diseño.

De igual manera, es fácil mostrar que el término "Implementación" se refiere únicamente a los artefactos, que claramente son extensionales y locales.

2.1.4 ¿Por qué es Importante la Arquitectura de Software?

Cervantes en [5] menciona que la arquitectura de software es de especial importancia debido a que la manera en que se estructura un sistema tiene un impacto directo sobre la capacidad de éste para satisfacer los atributos de calidad del sistema.

Los atributos de calidad son parte de los requerimientos no funcionales del sistema y son características que deben expresarse en forma cuantitativa. No tiene sentido, por ejemplo, decir que el sistema debe devolver una petición "de manera rápida", o presentar una página "ligera", ya que no es posible evaluar objetivamente si el sistema cubre o no esos requerimientos.

Ejemplos de atributos de calidad son el *desempeño*, que tiene que ver con el tiempo de respuesta del sistema a las peticiones que se le hacen, la *usabilidad*, que tiene que ver con qué tan sencillo les resulta a los usuarios realizar operaciones con el sistema, o bien la *modificabilidad*, que tiene que ver con qué tan simple resulta introducir cambios en el sistema. Estos son algunos de los atributos de calidad que se expondrán en secciones subsecuentes.

La manera en que se estructura un sistema permitirá o impedirá que se satisfagan los atributos de calidad. Por ejemplo, un sistema estructurado de tal manera que una petición deba transitar por muchos componentes antes de que se devuelva una respuesta podría tener un desempeño pobre. Por otro lado, un sistema estructurado de tal manera que los componentes estén altamente acoplados entre ellos limitará severamente la modificabilidad. Curiosamente la estructuración tiene un impacto mucho menor respecto a los requerimientos funcionales del sistema. Por ejemplo, un sistema difícil de modificar puede satisfacer plenamente los requerimientos funcionales que se le imponen.

Además de los atributos de calidad, la arquitectura de software juega un papel fundamental para guiar el desarrollo. Una de las múltiples estructuras que la componen se enfoca en partir el sistema en componentes que serán desarrollados por individuos o grupos de individuos. La identificación de esta estructura de asignación de trabajo es esencial para apoyar las tareas de planeación del proyecto.

Finalmente, los diseños arquitectónicos que se crean en una organización pueden ser reutilizados para crear sistemas distintos. Esto permite reducir costos y aumentar la calidad, sobre todo si dichos diseños han resultado previamente en sistemas exitosos [5].

2.1.5 Ciclo de Desarrollo de la Arquitectura

Dentro de un producto de desarrollo, e independientemente del proceso que se utilice, se puede hablar del "desarrollo de la arquitectura de software". Este desarrollo precede a la construcción del sistema, está dividido en las siguientes etapas: requerimientos, diseño, documentación y evaluación. Hay que señalar que las actividades relacionadas con el desarrollo de la arquitectura

de software generalmente forman parte de las actividades definidas dentro del proceso de desarrollo [5].

A continuación se describen dichas etapas:

- **Requerimientos.** La etapa de requerimientos se enfoca en la captura, documentación y priorización de requerimientos que influencian la arquitectura. Los atributos de calidad juegan un papel preponderante dentro de estos requerimientos, así que esta etapa hace énfasis en ellos. Otros requerimientos, sin embargo, son también relevantes para la arquitectura, estos son los requerimientos funcionales primarios y las restricciones.
- **Diseño.** La etapa de diseño es la etapa central en relación con la arquitectura y probablemente la más compleja. Durante esta etapa se definen las estructuras que componen la arquitectura. La creación de estas estructuras se hace en base a patrones de diseño, tácticas de diseño y elecciones tecnológicas. El diseño que se realiza debe buscar ante todo satisfacer los requerimientos que influencian a la arquitectura, y no simplemente incorporar diversas tecnologías por que están "de moda".
- **Documentación.** Una vez creado el diseño de la arquitectura, es necesario poder comunicarlo a otros involucrados dentro del desarrollo. La comunicación exitosa del diseño muchas veces depende de que dicho diseño sea documentado de forma apropiada. La documentación de una arquitectura involucra la representación de varias de sus estructuras que pueden ser representadas a través de distintas vistas. Una vista generalmente contiene un diagrama, además de información adicional, que apoya en la comprensión de dicho diagrama.
- **Evaluación.** Dado que la arquitectura de software juega un papel crítico en el desarrollo, es conveniente evaluar el diseño una vez que éste ha sido documentado con el fin de identificar posibles problemas y riesgos. La ventaja de evaluar el diseño es que es una actividad que se puede realizar de manera temprana (aún antes de codificar), y que el costo de corrección de los defectos identificados a través de la evaluación es mucho menor al costo que tendría el corregir estos defectos una vez que el sistema ha sido construido.

2.1.6 El Rol de Arquitecto

Las actividades descritas anteriormente requieren de habilidades particulares que son la responsabilidad del **arquitecto de software**. El arquitecto es un líder técnico que debe conocer los principios relacionados con la arquitectura de software, tener un amplio conocimiento respecto a la tecnología, y tener excelentes habilidades de comunicación oral y escrita.

Desafortunadamente, en la actualidad pocos arquitectos de software que laboran en la industria han recibido una formación teórica respecto al tema. Esto se debe a que no es sino hasta épocas recientes que se han establecido de manera más formal los conceptos relacionados con la arquitectura de software, y que actualmente pocas instituciones ofrecen cursos enfocados en el tema. El desconocimiento de los principios relativos a la arquitectura de software frecuentemente impacta de manera negativa a los proyectos de desarrollo [5].

2.2 Calidad Arquitectónica

Barbacci y colegas en [11] establecen que el desarrollo de formas sistemáticas para relacionar atributos de calidad de un sistema a su arquitectura provee una base para la toma de decisiones objetivas sobre acuerdos de diseño y permite a los ingenieros realizar predicciones razonablemente exactas sobre los atributos del sistema que son libres de prejuicios y asunciones no triviales. El objetivo de fondo es lograr la habilidad de evaluar cuantitativamente y llegar a acuerdos entre múltiples atributos de calidad para alcanzar un mejor sistema de forma global.

2.2.1 Clasificación de los Atributos de Calidad en las Arquitecturas de Software

La arquitectura de software de un sistema promueve, hace cumplir, y trata de predecir los atributos ó características de calidad que el sistema deberá soportar. Los atributos de calidad son las propiedades del sistema que van más allá de la funcionalidad del sistema y que hacen a un sistema bueno o malo desde una perspectiva técnica, por esto es que entran en la categoría de requerimientos no funcionales, y son identificados en el proceso de análisis de requerimientos.

Dado que la arquitectura de software de un sistema es un diseño parcial de un sistema antes de su construcción, es la responsabilidad del arquitecto de software identificar aquellos atributos de calidad que son más importantes e intentar diseñar una arquitectura que los refleje.

Existen varios autores que proponen clasificaciones de atributos de calidad como los descritos por Frank Buschmann en [8], Mary Shaw y David Garlan en [3] o Kai Qian y colegas en [12], partiendo de estos se considerará en este trabajo de tesis la siguiente clasificación dividida en tres grupos:

1. **Atributos de implementación (no observables/medibles en tiempo de ejecución)**

    - **Interoperabilidad:** Accesibilidad universal y la habilidad de intercambiar información entre componentes internos y con el mundo exterior.
    - **Integrabilidad:** La capacidad del sistema para integrarse con otros sistemas. La integrabilidad de un sistema depende de la medida en que el sistema utiliza estándares abiertos de integración y de qué tan bien está diseñada su API de tal manera que otros sistemas puedan usar los componentes del sistema que se construyó.
    - **Modificabilidad (Mantenibilidad y Extensibilidad):** Medida sobre la facilidad en la que se puede realizar un cambio para incorporar nuevos requerimientos. Los dos principales aspectos a considerar son costo y tiempo.
    - **Facilidad de prueba:** Grado en el que un sistema facilita el establecimiento de casos de prueba (esfuerzo humano, herramientas automatizadas de pruebas, inspecciones, entre otros). Usualmente requiere de la documentación acompañada por el diseño e implementación del sistema.
    - **Portabilidad:** El nivel de independencia en las plataformas de software y hardware. Los sistemas desarrollados usando lenguajes de programación de alto nivel usualmente tienen un buen grado de portabilidad (Por ejemplo, Java ya que

los programas desarrollados en este lenguaje son compilados una sola vez y pueden ejecutarse en todos lados).

- **Escalabilidad:** La habilidad del sistema a adaptarse a las peticiones incrementales del usuario. La escalabilidad disminuye la posibilidad de los cuellos de botella en el diseño del sistema.
- **Flexibilidad:** La facilidad de modificación del sistema para atender a diferentes ambientes o problemas para los cuales el sistema no fue diseñado originalmente.
- **Reusabilidad o Reutilización:** La habilidad para reusar o reutilizar porciones del sistema en otras aplicaciones.
- **Variabilidad:** Que tan bien una arquitectura puede manejar nuevos requerimientos. La variabilidad viene en muchas formas, los nuevos requerimientos pueden ser planeados o no planeados. En tiempo de implementación, el código fuente deberá ser fácil de extender para ejecutar nuevas funciones. Está relacionado con el atributo de modificabilidad.
- **Capacidad del sistema para soportar un subconjunto de características requeridas del sistema (*Subsetability*):** Es la propiedad del sistema que permite construir y ejecutar un pequeño subconjunto de características y añadir nuevas durante la construcción de todo el sistema. Para un desarrollo incremental, es importante que un sistema pueda ejecutar una parte de la funcionalidad prevista a lo largo de las iteraciones de desarrollo del producto.
- **Integridad conceptual:** Es la capacidad de una arquitectura de comunicar clara y concisamente la visión del sistema.
- **Edificabilidad:** Cuando una arquitectura puede ser/no ser construida usando el presupuesto, personal y tiempo disponibles para entregar el proyecto.

2. Atributos de tiempo de ejecución (observables/medibles en tiempo de ejecución)

- **Disponibilidad:** Cantidad de tiempo que un sistema está en operación. Las mediciones más frecuentes son: cálculo de la cantidad de tiempo que transcurre entre fallas y cálculo del tiempo para determinar que tan rápido es capaz un sistema de recuperarse tras suceder una falla. La disponibilidad puede ser alcanzada vía replicación y un diseño cuidadoso para hacer frente a las fallas de hardware, software ó de la red.

  Este atributo está relacionado muy cercanamente con el atributo de confiabilidad. Mientras más confiable sea un sistema, entonces se dice que tiene mayor disponibilidad.
- **Seguridad:** La capacidad del sistema para hacer frente a ataques maliciosos del exterior al interior del sistema. La seguridad puede ser mejorada instalando firewalls, estableciendo procesos de autenticación y autorización, y usando encriptación.
- **Desempeño:** Medida de la eficiencia del sistema de un requerimiento funcional. Incluye la eficiencia del sistema en lo que respecta al tiempo de respuesta,

rendimiento y utilización de recursos, atributos que usualmente tienen conflictos entre ellos.

- **Usabilidad:** Que tan fácil es para el usuario entender y operar el sistema así como el nivel de satisfacción humana al usarlo. La usabilidad incluye cuestiones sobre completitud, correctitud, compatibilidad, así como una interfaz de usuario amigable, documentación completa y apoyo técnico.
- **Confiabilidad:** La capacidad del sistema para operar en el tiempo. Incluye la frecuencia de fallas, la exactitud de los resultados de salida, el "Tiempo-Promedio-Entre-Fallas" (*Mean-Time-To-Failure*, MTTF), la habilidad para recuperarse de las fallas, y la predictibilidad de las fallas.
- **Funcionalidad:** La capacidad del sistema de realizar la tarea para la cual fue diseñado.
- **Mantenibilidad (Extensibilidad, adaptabilidad, utilidad, facilidad de prueba, compatibilidad y configurabilidad):** La facilidad del sistema de software al cambio.
- **Variabilidad:** En tiempo de ejecución el sistema deberá permitir componentes conectables (*pluggable*) que permitan modificar la conducta del sistema sobre la marcha.

3. **Atributos del negocio**

- **Tiempo de comercialización:** Es el tiempo que toma desde el análisis de requerimientos a la fecha en que el producto es liberado.
- **Costo:** Incluye el costo de construcción, mantenimiento y operación del sistema.
- **Tiempo de vida:** Periodo de tiempo en que el producto esta "vivo" antes del retiro.

Algunas veces se usa el término "ágil" para describir una arquitectura. La "agilidad" es un término que se considera como una composición de los atributos de calidad de: modificabilidad, portabilidad, reusabilidad (reutilización), integrabilidad y facilidad de prueba [10].

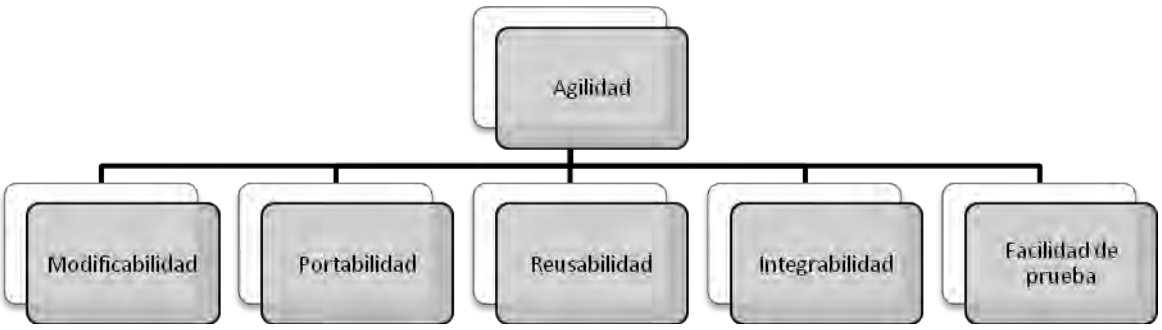

Figura 2. Composición de la Agilidad [10]

Es muy difícil que un sólo estilo arquitectónico pueda cumplir con todos los atributos de calidad simultáneamente. Los arquitectos de software frecuentemente necesitan hacer un balance de compensaciones entre dichos atributos. Las parejas de atributos de calidad típicas a compensar son las siguientes:

- **Compensación entre espacio y tiempo**: Por ejemplo, incrementar el tiempo de eficiencia de una tabla hash significa un decremento en la eficiencia en el espacio.
- **Compensación entre confiabilidad y desempeño**: Por ejemplo, los programas Java están bien protegidos contra desbordamiento de búferes debido a medidas de seguridad como verificación de límites en arreglos. Tales características de fiabilidad existen a costa de la eficiencia del tiempo, comparado con el simple y rápido lenguaje C que utiliza los peligrosos pero eficientes apuntadores.
- **Compensación entre escalabilidad y desempeño**: Por ejemplo, un enfoque típico para incrementar la escalabilidad de un servicio es la replicación de servidores. Para asegurar la consistencia en todos los servidores (asegurarse que cada servidor tiene la misma coherencia lógica de datos), el desempeño en tiempo de todo el servicio se ve comprometido.

Cuando un estilo arquitectónico no satisface todos los atributos de calidad deseados, los arquitectos de software deben trabajar con los analistas del sistema y los demás interesados para establecer la prioridad de los atributos de calidad. Los arquitectos de software pueden seleccionar un diseño óptimo identificando diseños arquitectónicos alternativos y evaluando los atributos de calidad que contienen cada uno.

2.2.2 Relación de los Atributos de Calidad dentro de las Arquitecturas de Software

Bass y colegas en [13] establecen que para alcanzar un atributo específico, es necesario tomar decisiones de diseño arquitectónico que requieren un pequeño conocimiento de la funcionalidad. Por ejemplo, el desempeño depende de los procesos del sistema y su ubicación en los procesadores, caminos de comunicación, etc. Por otro lado, establecen que al considerar una decisión de arquitectura de software, el arquitecto se pregunta cuál será el impacto de ésta sobre ciertos atributos; por ejemplo, modificabilidad, desempeño, seguridad, usabilidad, etc.

Por esta razón, en [13] se afirma que cada decisión incorporada en una arquitectura de software puede afectar potencialmente los atributos de calidad. Cada decisión tiene su origen en preguntas acerca del impacto sobre estos atributos, y el arquitecto puede argumentar cómo la decisión tomada permite alcanzar algún objetivo. Con frecuencia, el objetivo es un atributo de calidad en particular, por lo que al tomar decisiones de arquitectura de software que afecte a los atributos de calidad, se tienen dos consecuencias:

- Se formula un argumento que explica la razón por la que la decisión tomada permite alcanzar uno o varios atributos de calidad. Esto resulta de gran importancia porque además permite comprender las consecuencias de un cambio de decisión.
- Surgen preguntas sobre el impacto de una decisión sobre otros atributos, que a menudo se responden en el contexto de otras decisiones.

En su planteamiento, Bass en [13] presenta los problemas que existen en relación a la documentación de la relación entre arquitectura de software y atributos de calidad. De igual

forma, establecen que la relación entre la arquitectura de software y los atributos de calidad no se encuentra sistemática y completamente documentada, debido a diversas razones:

- Existen atributos de calidad que, luego de ser estudiados durante años, poseen definiciones generalmente aceptadas. Sin embargo, existen algunos que carecen de definición, lo que inhibe el proceso de exploración de la relación en estudio.
- Los atributos no están aislados ni son independientes entre sí. Muchos atributos conforman un subconjunto de otro, es decir, se definen en función de otros atributos que lo contienen. Por ejemplo, el atributo disponibilidad puede ser un atributo por sí mismo; sin embargo, puede ser subconjunto de usabilidad y seguridad.
- El análisis de atributos no se presta a estandarizaciones, puesto que existen diferentes patrones con distintos niveles de profundidad, y resulta complicado establecer cuáles patrones se pueden utilizar para analizar calidad.
- Las técnicas de análisis son específicas para un atributo en particular. Por esta razón es difícil comprender la interacción entre varios análisis de atributos específicos.

A pesar de estas dificultades, Bass en [13] establece que la relación entre arquitectura de software y atributos de calidad tiene diversos beneficios, que justifican su documentación:

- Realza en gran medida el proceso de análisis y diseño arquitectónico, puesto que el arquitecto puede reutilizar análisis existentes y determinar acuerdos explícitamente en lugar de hacerlo sobre la marcha. Los arquitectos experimentados hacen esto intuitivamente, basados en su experiencia en codificación. Por ejemplo, durante el análisis, un arquitecto puede reconocer el impacto de la codificación de una estructura en los atributos de calidad.
- Una vez que el arquitecto entiende el impacto de los componentes arquitectónicos sobre uno o varios atributos de calidad, estaría en capacidad de reemplazar un conjunto de componentes por otro cuando lo considere necesario.
- Una vez que se codifica la relación entre arquitectura y atributos de calidad, es posible construir un protocolo de pruebas que habilitará la certificación por parte de terceros.

Por todo lo expuesto, se reconoce la importancia de la arquitectura de un sistema de software como la base de diseño de un sistema [14], así como también un artefacto que determina atributos de calidad [15].

Ahora bien, es interesante considerar que tanto la arquitectura de un sistema de software como el sistema en sí mismo, se encuentran íntimamente relacionados con su entorno, por lo que es necesario tomarlo en cuenta para efectos del estudio de la calidad y la determinación de los atributos de calidad.

Las características adicionales del sistema o atributos de calidad, están estrechamente relacionadas con el uso que se le dará al sistema propuesto [15], pues es el contexto quien define el comportamiento del mismo, sin dejar de lado la funcionalidad [11]. Por ello es necesario

analizar el contexto del sistema, dado que de aquí se deriva mucha información relativa a su calidad.

Se considera que el sistema y su entorno son compañeros en un contrato en el cual cada uno tiene expectativas y obligaciones [11]. En muchos casos, el nivel de compromiso entre el sistema y su entorno no queda establecido de forma clara con el simple análisis de los requerimientos funcionales de un sistema. En este sentido, Barbacci y colegas en [11] propone la necesidad del análisis de toda la información disponible, tanto del sistema como del contexto.

Barbacci y colegas en [11] plantean un esquema general que permite establecer elementos que deben ser tomados en consideración y que vienen dados por distintos tipos de actividades. Por ejemplo, con base en las actividades del sistema y su importancia, para efectos del sistema y su entorno, es posible determinar que, propiedades como el desempeño o la disponibilidad del sistema, deben ser consideradas como atributos de calidad para el diseño de la arquitectura del mismo. De igual forma, Barbacci y colegas propone el uso de técnicas como los escenarios, listas de chequeo y cuestionarios, como formas cualitativas de determinar el nivel del contrato, elementos de hardware y la arquitectura de software, teniendo como objetivo principal velar por la calidad del sistema.

A lo largo del proceso de diseño y desarrollo, los atributos de calidad juegan un papel importante, pues en base a estos se generan las decisiones de diseño y argumentos que los justifican [13]. Dado que la arquitectura de software inhibe o facilita los atributos de calidad [9], resulta de particular interés analizar la influencia de ciertos elementos de diseño utilizados para la definición de la misma, determinando sus características. Estos elementos de diseño son los *patrones arquitectónicos*, los *estilos arquitectónicos*, y los *patrones de diseño*.

2.2.3 Modelos de Calidad

En la práctica, los modelos calidad resultan de utilidad para la predicción de confiabilidad y en la gerencia de calidad durante el proceso de desarrollo, así como para efectuar la medición del nivel de complejidad de un sistema de software [16]. Es interesante destacar que la organización y descomposición de los atributos de calidad ha permitido el establecimiento de modelos específicos para efectos de la evaluación de la calidad arquitectónica.

Pressman en [17] indica que los factores que afectan a la calidad del software no cambian, por lo que resulta útil el estudio de los modelos de calidad que han sido propuestos en este sentido desde los años 70. Dado que los factores de calidad presentados para ese entonces siguen siendo válidos, se estudiarán los modelos más importantes propuestos hasta ahora: McCall (1977), Dromey (1996), FURPS (1987), ISO/IEC 9126 (1991) e ISO/IEC 9126 adaptado para arquitecturas de software, propuesto por Losavio en [18].

2.2.3.1 Modelo de McCall

El modelo de McCall en [19] describe la calidad como un concepto elaborado mediante relaciones jerárquicas entre factores de calidad, en base a criterios y métricas de calidad. Este enfoque es sistemático, y permite cuantificar la calidad a través de las siguientes fases:

- Determinación de los factores que influyen sobre la calidad del software.
- Identificación de los criterios para juzgar cada factor.
- Definición de las métricas de los criterios y establecimiento de una función de normalización que define la relación entre las métricas de cada criterio y los factores correspondientes.
- Evaluación de las métricas.
- Correlación de las métricas a un conjunto de guías que cualquier equipo de desarrollo podría seguir.
- Desarrollo de las recomendaciones para la colección de métricas.

En el modelo de McCall, los factores de calidad se concentran en tres aspectos importantes de un producto de software: características operativas, capacidad de cambios y adaptabilidad a nuevos entornos.

En este modelo, el término *factor de calidad* define características claves que un producto debe exhibir. Los atributos del factor de calidad que define el producto son los nombrados *criterios de calidad*. Las *métricas de calidad* denotan una medida que puede ser utilizada para cuantificar los criterios. McCall identifica una serie de criterios, tales como rastreabilidad, simplicidad, capacidad de expansión, etc. Las métricas desarrolladas están relacionadas con los factores de calidad y la relación que se establece se mide en función del grado de cumplimiento de los criterios.

La Tabla 2 muestra, para el modelo de McCall, los factores de calidad y sus criterios asociados. En ella se observa que algunos de los criterios son compartidos por más de un factor.

Factor	Criterio
Correctitud	Rastreabilidad Completitud Consistencia
Confiabilidad	Consistencia Exactitud Tolerancia a fallas
Eficiencia	Eficiencia de ejecución Eficiencia de almacenamiento
Integridad	Control de acceso Auditoría de acceso
Usabilidad	Operabilidad Entrenamiento Comunicación
Mantenibilidad	Simplicidad Concreción
Capacidad de Prueba	Simplicidad Instrumentación Auto-descriptividad Modularidad
Flexibilidad	Auto-descriptividad Independencia del sistema Independencia de máquina
Reusabilidad	Auto-descriptividad Generalidad Modularidad

	Independencia del sistema
	Independencia de máquina
Interoperabilidad	Modularidad
	Similitud de comunicación
	Similitud de datos

Tabla 2. Criterios asociados a los factores de calidad - Modelo de McCall [19]

2.2.3.2 Modelo de Dromey

Dromey en [20] propuso un marco de referencia – o meta modelo - para la construcción de modelos de calidad, basado en cómo las propiedades medibles de un producto de software pueden afectar los atributos de calidad generales, como por ejemplo, confiabilidad y mantenibilidad. El problema que se plantea es cómo conectar tales propiedades del producto con los atributos de calidad de alto nivel. Para solventar esta situación, Dromey sugiere el uso de cuatro categorías que implican propiedades de calidad, que son: correctitud, internas, contextuales y descriptivas [20].

La Tabla 3 presenta la relación que establece Dromey entre las propiedades de calidad del producto y los atributos de calidad de alto nivel.

Propiedades del Producto	Atributos de Calidad
Correctitud	Funcionalidad
	Confiabilidad
Internas	Mantenibilidad
	Eficiencia
	Confiabilidad
Contextuales	Mantenibilidad
	Reusabilidad
	Portabilidad
	Confiabilidad
Descriptivas	Mantenibilidad
	Reusabilidad
	Portabilidad
	Confiabilidad

Tabla 3. Relación entre propiedades del producto y atributos de calidad – Modelo de Dromey [20]

El proceso de construcción de modelos de calidad propuesto por Dromey consta de 5 pasos, basados en las propiedades mencionadas. Los pasos del marco de referencia propuesto son:

- Especificación de los atributos de calidad de alto nivel (por ejemplo, confiabilidad, mantenibilidad).
- Determinación de los distintos componentes del producto a un apropiado nivel de detalle (por ejemplo, paquetes, subrutinas, declaraciones).
- Para cada componente, determinación y categorización de sus implicaciones más importantes de calidad.
- Proposición de enlaces que relacionan las propiedades implícitas a los atributos de calidad, o, alternativamente, uso de enlaces de las cuatro categorías de atributos propuestas.

- Iteración sobre los pasos anteriores, utilizando un proceso de evaluación y refinamiento.

Para ilustrar sus planteamientos, Dromey demuestra el uso de su procedimiento para la construcción de un modelo de calidad de implementación, un modelo de calidad de requerimientos, y un modelo de calidad de diseño.

2.2.3.3 Modelo FURPS

El modelo de McCall ha servido de base para modelos de calidad posteriores, y este es el caso del modelo FURPS, producto del desarrollo de Hewlett-Packard [21]. En este modelo se desarrollan un conjunto de factores de calidad de software, bajo el acrónimo de FURPS: funcionalidad *(Functionality)*, usabilidad *(Usability)*, confiabilidad *(Reliability)*, desempeño *(Performance)* y capacidad de soporte *(Supportability).* La Tabla 4 presenta la clasificación de los atributos de calidad que se incluyen en el modelo, junto con las características asociadas a cada uno [17].

Factor de Calidad	Atributos
Funcionalidad	Características y capacidades del programa Generalidad de las funciones Seguridad del sistema
Facilidad de uso	Factores humanos Factores estéticos Consistencia de la interfaz Documentación
Confiabilidad	Frecuencia y severidad de las fallas Exactitud de las salidas Tiempo medio de fallos Capacidad de recuperación ante fallas Capacidad de predicción
Rendimiento	Velocidad del procesamiento Tiempo de respuesta Consumo de recursos Rendimiento efectivo total Eficacia
Capacidad de soporte	Extensibilidad Adaptabilidad Capacidad de pruebas Capacidad de configuración Compatibilidad Requisitos de instalación

Tabla 4. Atributos de calidad – Modelo FURPS [17]

El modelo FURPS incluye, además de los factores de calidad y los atributos, restricciones de diseño y requerimientos de implementación, físicos y de interfaz [21]. Las restricciones de diseño especifican o restringen el diseño del sistema. Los requerimientos de implementación especifican o restringen la codificación o construcción de un sistema (por ejemplo, estándares requeridos, lenguajes, políticas). Por su parte, los requerimientos de interfaz especifican el comportamiento de los elementos externos con los que el sistema debe interactuar. Por último, los requerimientos físicos especifican ciertas propiedades que el sistema debe poseer, en términos de materiales, forma, peso, tamaño (por ejemplo, requisitos de hardware, configuración de red).

2.2.3.4 Modelo ISO/IEC 9126

El estándar ISO/IEC 9126 ha sido desarrollado en un intento de identificar los atributos clave de calidad para un producto de software [17]. Este estándar es una simplificación del Modelo de McCall [19], e identifica seis características básicas de calidad que pueden estar presentes en cualquier producto de software. El estándar provee una descomposición de las características en subcaracterísticas, que se muestran en la Tabla 5.

Característica	Subcaracterística
Funcionalidad	Adecuación
	Exactitud
	Interoperabilidad
	Seguridad
Confiabilidad	Madurez
	Tolerancia a fallas
	Recuperabilidad
Usabilidad	Facilidad de entendimiento
	Capacidad de aprendizaje
	Operabilidad
Eficiencia	Comportamiento en tiempo
	Comportamiento de recursos
Mantenibilidad	Analizabilidad
	Modificabilidad
	Estabilidad
	Capacidad de pruebas
Portabilidad	Adaptabilidad
	Instalabilidad
	Reemplazabilidad

Tabla 5. Características y subcaracterísticas de calidad – Modelo ISO/IEC 9126 [17]

Es interesante destacar que los factores de calidad que contempla el estándar ISO/IEC 9126 no son necesariamente usados para mediciones directas [17], pero proveen una valiosa base para medidas indirectas, y una excelente lista para determinar la calidad de un sistema.

2.2.3.5 ISO/IEC 9126 Adaptado para Arquitecturas de Software

Losavio en [18] propone una adaptación del modelo ISO/IEC 9126 de calidad de software para efectos de la evaluación de arquitecturas de software. El modelo se basa en los atributos de calidad que se relacionan directamente con la arquitectura: funcionalidad, confiabilidad, eficiencia, mantenibilidad y portabilidad.

Los autores plantean que la característica de usabilidad propuesta por el modelo ISO/IEC 9126 puede ser refinada para obtener atributos que se relacionan con los componentes de la interfaz con el usuario. Dado que estos componentes son independientes de la arquitectura, no son considerados en la adaptación del modelo. La Tabla 6 presenta los atributos de calidad planteados por [18] que poseen subcaracterísticas asociadas con elementos de tipo arquitectónico.

Característica	Subcaracterística	Elementos de tipo arquitectónico
Funcionalidad	Adecuación	Refinamiento de los diagramas de secuencia
	Exactitud	Identificación de los componentes con las funciones responsables de los cálculos
	Interoperabilidad	Identificación de conectores de comunicación con sistemas externos
	Seguridad	Mecanismos o dispositivos que realizan explícitamente la tarea
Eficiencia	Desempeño	Componentes involucrados en un flujo de ejecución para una funcionalidad
	Utilización de recursos	Relación de los componentes en términos de espacio y tiempo
Mantenibilidad	Acoplamiento	Interacciones entre componentes
	Modularidad	Número de componentes que dependen de un componente
Portabilidad	Adaptabilidad	Presencia de mecanismos de adaptación
	Instalabilidad	Presencia de mecanismos de instalación
	Coexistencia	Presencia de mecanismos que faciliten la coexistencia
	Reemplazabilidad	Lista de componentes reemplazables para cada componente

Tabla 6. Atributos de calidad planteados por Losavio en [18], que poseen subcaracterísticas asociadas con elementos de tipo arquitectónico

En la literatura es posible encontrar planteamientos en relación al establecimiento de los atributos de calidad y su relación con la arquitectura de software [13]. Clements y colegas en [15] establecen que la arquitectura se determina por los atributos de calidad. Bass en [13] explica la relación existente entre estos, conjuntamente con los problemas y beneficios de la documentación de esta relación.

2.3 Patrones Arquitectónicos

2.3.1 ¿Qué son los Patrones Arquitectónicos?

Un patrón para una arquitectura de software describe un problema particular recurrente que surge en contextos específicos de diseño, y presenta un esquema genérico de eficiencia probada para su solución. El esquema de solución es especificado describiendo los componentes de los que se constituye, sus responsabilidades y relaciones, y la forma en la que colaboran entre ellos [8].

Nota: A lo largo de este trabajo se usará "Patrones" como "Patrones Arquitectónicos".

2.3.1.1 ¿Para qué sirven los Patrones?

Algunas de las funcionalidades de los patrones son los siguientes:

- Los patrones documentan experiencias de diseño existentes y bien probadas.
- Identifican y especifican abstracciones que están sobre el nivel de las clases e instancias individuales, o de los componentes. Típicamente un patrón describe varios componentes, clases o objetos, y detalla sus responsabilidades y relaciones, así como su cooperación. Todos los componentes unidos resuelven el problema en el que el patrón se centra, y usualmente más efectivamente que un componente individual.
- Los patrones proveen de un vocabulario común y la comprensión de los principios de diseño. Los nombres de los patrones forman parte de un lenguaje extendido de diseño, que facilita una discusión efectiva de los problemas de diseño y su solución.
- Los patrones son un medio para documentar las arquitecturas de software. Describen la visión que se tenía en mente al momento de diseñar el sistema, lo que permite a otras

personas evitar violar la visión inicial al momento de extender o modificar la arquitectura original, o cuando se modifica el código del sistema.

- Los patrones apoyan la construcción del software con ciertas propiedades definidas. Proveen de un esquema de conducta funcional que permite ayudar a implementar la funcionalidad de la aplicación. Adicionalmente, los patrones conllevan explícitamente requerimientos no funcionales para los sistemas de software, como facilidad de cambio, confiabilidad, facilidad para desarrollar pruebas o reusabilidad.
- Los patrones ayudan a construir arquitecturas de software complejas y heterogéneas. Cada patrón provee un conjunto predefinido de componentes, roles y relaciones entre ellos, que pueden ser usados para especificar aspectos particulares de estructuras concretas de software. El usar artefactos de diseño predefinidos apoya a realizar un diseño más rápido y con calidad.
- Aunque los patrones determinan la estructura básica de una solución para un problema particular de diseño, no especifica una solución totalmente detallada. Un patrón provee el esquema de una solución genérica para una familia de problemas, por lo que se debe implementar dicho esquema de acuerdo a las necesidades del problema de diseño que se esté manejando. Los patrones ayudan a resolver problemas, pero no proveen soluciones completas.
- Los patrones ayudan a administrar la complejidad del software. Cada patrón describe una forma probada para manejar un problema específico: los tipos de componentes que se necesitan, sus roles, los detalles que deben estar ocultos, las abstracciones que deben ser visibles y como trabajan en conjunto. Cuando se encuentra en una situación concreta de diseño cubierta por un patrón, no hay necesidad de gastar tiempo inventando un solución nueva para el problema que se esté manejando.
- Los patrones son construidos con los principios de desarrollo de software para y con reusabilidad, y diseño para el cambio.

2.3.1.2 Sistema de Patrones

Un patrón individual no permite la construcción detallada y completa de una arquitectura de software, sólo ayuda a diseñar un aspecto de una aplicación.

Para usar los patrones efectivamente, entonces se necesita organizarlos en *sistemas de patrones*. Un sistema de patrones describe uniformemente patrones, los clasifica, y más importante aún, muestra como están interrelacionados entre ellos. Los sistemas de patrones también ayudan a encontrar el patrón correcto para resolver un problema o identificar alternativas de solución. Esto en contraste con un catálogo de patrones, en donde cada patrón es descrito más ó menos en aislamiento de otros patrones. Un sistema de patrones nos ayuda a usar todo el poder que los patrones proveen.

2.3.1.3 Categorías de Patrones

De acuerdo a [8], los patrones se pueden agrupar en tres categorías:

- Un *patrón arquitectónico* expresa un esquema de organización estructural fundamental para un sistema de software. Provee un conjunto predefinido de subsistemas, especifica sus responsabilidades, e incluye reglas y directrices para organizar las relaciones entre ellos. Los patrones arquitectónicos son plantillas para arquitecturas de software concretas. Especifican las propiedades estructurales de todo el sistema de una aplicación, y tienen impacto en los subsistemas de la arquitectura. La selección de un patrón arquitectónico es por consiguiente, una decisión fundamental de diseño en el desarrollo de un sistema de software.
- Los subsistemas de una arquitectura de software, así como las relaciones entre estos, usualmente consiste de varias unidades pequeñas arquitectónicas, son descritas usando *patrones de diseño*. Un patrón de diseño provee un esquema para refinar los subsistemas o componentes de un sistema de software, o las relaciones entre ellos. En él se describe una estructura comúnmente recurrente de los componentes de comunicación, en el que resuelve un problema de diseño general en un contexto particular. La aplicación de un patrón de diseño no tiene efecto en la estructura fundamental del sistema de software, pero puede tener una fuerte influencia en la arquitectura de un subsistema.
- Los *modismos* tratan con la implementación de cuestiones específicas de diseño. Un modismo es un patrón de bajo nivel específico para un lenguaje de programación, maneja aspectos de diseño e implementación. Describe como implementar aspectos particulares de componentes o las relaciones entre ellos usando las características de un lenguaje de programación determinado. Capturan la experiencia existente de programación, con frecuencia un modismo es diferente para diferentes lenguajes, algunas veces un modismo que es útil para un lenguaje de programación no tiene sentido en otro.

De esta manera, los patrones arquitectónicos pueden ser usados al principio del diseño general del sistema, los patrones de diseño durante toda la fase de diseño, y los modismos durante toda la fase de implementación.

2.3.1.4 Implementando Patrones

Otro aspecto que considerar para la integración de patrones en una arquitectura de software es el paradigma para implementarlo. Muchos patrones de software actuales tienen un tono diferente al orientado a objetos lo que significa que este paradigma no es esencial para implementarlos, y además aplicando cierta abstracción en los patrones que usan orientación a objetos, pueden ser modificados para adaptarlos al paradigma que se quiera utilizar.

Se dice que se puede implementar los patrones en casi todos los paradigmas de programación y en casi todos los lenguajes de programación. Adicionalmente, cada lenguaje de programación tiene sus patrones específicos, es decir sus modismos, en los que se captura la experiencia existente de programación en el lenguaje y estilo de programación definido en él.

En conclusión, se puede decir que no hay un patrón individual o lenguaje para implementar los patrones. Los patrones pueden ser integrados en cualquier paradigma usado para construir arquitecturas de software.

2.3.2 Arquitectura de Software en el Contexto de Patrones

Una arquitectura de software dentro del área de patrones es una descripción de *los subsistemas y componentes* de un sistema de software, y de las *relaciones* entre ellos. Los subsistemas y los componentes son típicamente especificados en diferentes *vistas* para mostrar las propiedades funcionales y no funcionales relevantes de un sistema de software. La arquitectura de un sistema de software es un artefacto (producto) y es el resultado de la actividad de diseño del software.

A continuación se describen los elementos que intervienen en la descripción anterior:

Un *componente* es una parte encapsulada de un sistema de software. Un componente tiene una interfaz. Los componentes sirven como bloques de construcción para la estructura de un sistema. Dentro del nivel de lenguajes de programación los componentes pueden ser representados como módulos, clases, objetos o un conjunto de funciones relacionadas. El término "componente" es independiente de su eventual manifestación en código fuente.

Una *relación* denota una conexión entre componentes. Una relación puede ser estática o dinámica. Las relaciones estáticas se muestran directamente en el código fuente, tratan con la colocación de los componentes dentro de una arquitectura de software. Las relaciones dinámicas tratan con conexiones temporales e interacciones dinámicas entre componentes, pueden no ser fácilmente visibles desde la estructura estática del código fuente.

Las relaciones entre componentes tienen un gran impacto en la calidad general de una arquitectura de software. Por ejemplo, la posibilidad de cambiar (changeability - cambiabilidad) es mejor soportada en las arquitecturas de software en cuyas relaciones permiten la variación de componentes, en contraste con aquellas arquitecturas en que cualquier cambio a un componente afecta la implementación de los clientes y los colaboradores.

Una *vista* representa un aspecto parcial de una arquitectura de software en la que se muestran propiedades específicas de un sistema de software. Existen varias propuestas para describir las arquitecturas de software, una de ellas es la que propone [22] en cuatro diferentes vistas:

- **Vista Lógica:** Describe el diseño del modelo de objetos, o el modelo correspondiente como el diagrama entidad - relación.
- **Vista del Proceso:** Muestra aspectos de concurrencia y sincronización.
- **Vista Física:** Se describe el mapeo del software sobre el hardware y sus aspectos de distribución.
- **Vista de Desarrollo:** Describe la organización estática del software en el ambiente de desarrollo.

2.3.3 Estilos Arquitectónicos

En 1992 Dwayne E. Perry y Alexander L. Wolf introdujeron la noción de estilo arquitectónico:

Un estilo arquitectónico define una familia de sistemas de software en términos de su organización estructural. Un estilo arquitectónico expresa los componentes y las relaciones entre ellos, con las limitaciones para su aplicación, y las reglas de composición y diseño para su construcción [23].

Es decir, un estilo arquitectónico expresa un tipo particular de estructura fundamental de un sistema de software junto con un método asociado que especifica la manera de construirlo. Un estilo arquitectónico también comprende información acerca de cuándo usar el estilo que se describe, sus invariantes y especializaciones, así como las consecuencias de su aplicación.

Los patrones arquitectónicos propuestos por [8] son muy similares a los estilos arquitectónicos descritos por [24] y [25]. De tal manera que dichos estilos arquitectónicos pueden ser descritos como un patrón arquitectónico.

Aunque por otro lado, los estilos arquitectónicos difieren de los patrones en los siguientes aspectos:

- Los estilos arquitectónicos sólo describen el marco de trabajo estructural para aplicaciones. Los patrones para las arquitecturas de software existen en varios rangos, empezando con los patrones para definir la estructura básica de una aplicación (patrones arquitectónicos) y terminando con los patrones que describen como implementar un problema de diseño particular en un lenguaje de programación (modismos).
- Los estilos arquitectónicos son independientes entre ellos. Pero los patrones pueden depender de patrones pequeños que tienen contenidos, de los patrones con los que interactúa, y de los patrones más grandes en los que están contenidos.
- Los patrones son más orientados al problema que los estilos arquitectónicos. Los estilos arquitectónicos expresan las técnicas de diseño desde un punto de vista que es independiente de la situación actual de diseño. Un patrón expresa un muy específico problema recurrente de diseño y presenta una solución a este, todo desde el punto de vista del contexto en el que el problema aparece.

2.3.3.1 Marcos de Trabajo (*Frameworks*)

Un marco de trabajo es un sistema (subsistema) informático parcialmente completo que pretende ser una instancia. Define la arquitectura para una familia de sistemas (subsistemas) y provee los bloques de construcción básicos para crearlas. También define los lugares donde se pueden realizar adaptaciones de funcionalidad. En un ambiente orientado a objetos un marco de trabajo contiene clases abstractas y concretas.

La instanciación a un marco de trabajo implica la composición y creación de subclases para las clases existentes. Un marco de trabajo para aplicaciones en un dominio específico es llamado un "marco de aplicaciones".

Un marco de aplicaciones consiste de "puntos congelados" y "puntos calientes". Los puntos congelados definen totalmente la arquitectura de un sistema de software, sus componentes básicos y las relaciones entre ellos. Estos se mantienen sin cambios en cualquier instanciación del marco de trabajo. Los puntos calientes representan aquellas partes del marco de aplicaciones que son específicas a sistemas de software individuales. Los puntos calientes son diseñados para ser genéricos, pueden ser adaptados a las necesidades de la aplicación en desarrollo.

Cuando se crea un sistema de software en concreto con un marco de aplicaciones, sus puntos calientes son especializados de acuerdo a las necesidades específicas y los requerimientos del sistema. Para lograr adaptabilidad y cambiabilidad en un marco de aplicaciones, no se está restringido a las técnicas orientadas a objetos tales como herencia y polimorfismo, se puede usar patrones.

2.3.4 Principios de Diseño y Patrones de Diseño para las Arquitecturas de Software

Estos principios facilitadores son independientes de un método específico de desarrollo de software, y muchos de ellos han sido conocidos y reconocidos por años, su importancia para el éxito de desarrollo de software se ha incrementado con los años, y están fuertemente ligados a la disciplina de arquitecturas de software. Los patrones para arquitecturas de software son explícitamente construidos bajo estos principios, muchos de ellos ponen una especial atención a un principio en particular. A continuación se resumen algunos de estos principios facilitadores para las arquitecturas de software:

- **Abstracción.** Es uno de los principios fundamentales que se utiliza para hacer frente a la complejidad. Grady Booch define abstracción como "Las características esenciales de un objeto que lo distinguen de cualquier otro tipo de objeto y que definen nítidamente los límites conceptuales en relación con la perspectiva del espectador". La palabra objeto puede ser remplazada por componente para una definición más general de abstracción. Existen muchas formas de abstracción, como la abstracción de entidad, acción, máquina virtual o coincidente. Este principio es manejado por patrones como el de "*Layers*" (Capas) y "*Abstract Factory*", entre otros.
- **Encapsulación.** Se refiere a la agrupación de los elementos de una abstracción que constituyen su estructura y comportamiento. La encapsulación provee barreras explícitas entre diferentes abstracciones. El patrón "*Forwarder-Receiver*", encapsula los detalles de implementación de los mecanismos de comunicación entre procesos. La encapsulación fomenta las propiedades no funcionales de cambiabilidad y reusabilidad.
- **Ocultamiento de la Información.** Está relacionado con ocultar los detalles de la implementación de los componentes de los clientes, para manejar mejor la complejidad del sistema y minimizar la unión entre componentes. Cualquier detalle de un componente que el cliente no necesite conocer con el fin de utilizar la propiedad debe ser ocultado por el componente. El patrón "*Whole-Part*" maneja este principio explícitamente. El principio de encapsulación es usado frecuentemente como una forma de lograr la ocultación de

información. El ocultamiento de la información puede ser logrado también al usar el principio de separación de la interfaz y la implementación.

Lo que será ocultado dentro de un componente depende algunas veces de la aplicación. Aspectos que los clientes no necesiten conocer en una aplicación no deben ser visibles en otra.

El concepto de reflejo relaja el principio de ocultamiento de la información. El patrón "*Reflection*" abre la implementación de un sistema de software o de un componente de una forma definida para proveer mayor flexibilidad para la adaptación y el cambio. Este es uno de los conceptos fundamentales y más importantes en la ingeniería de software.

- **Modularización.** Está relacionada con la descomposición significativa de un sistema de software y con su agrupación en subsistemas y componentes. La mayor tarea es decidir como empacar físicamente las entidades que forman la estructura lógica de la aplicación. El principal objetivo de la modularización es manejar la complejidad del sistema con la introducción de límites bien definidos y documentados dentro de un programa. Los módulos sirven físicamente como contenedores para funcionalidades y responsabilidades de una aplicación. Está relacionada muy cercanamente con el principio de encapsulación, Algunos patrones que manejan este principio son "*Layers*", "*Pipes and Filtres*" y "*Whole-Part*".
- **Separación de Tareas (*Concerns*).** Responsabilidades diferentes o no relacionadas deben ser separadas de otras dentro de un sistema de software uniéndolos a distintos componentes. Los componentes que contribuyen a la solución de tareas especificas deben ser separados de los componentes envueltos en el cálculo de otras tareas. Si un componente juega diferentes roles en diferentes contextos, estos roles deben ser independientes y separados entre ellos dentro del componente. Casi todos los patrones de sistemas manejan este principio fundamental de alguna manera. En el patrón "*Model-View-Controller*" se separan las tareas de los modelos internos, la presentación al usuario y el procesamiento de las entradas.
- **Acoplamiento y Cohesión (*Coupling and Cohesion*).** Son principios originalmente introducidos como parte del enfoque de diseño estructurado. El acoplamiento se centra en los aspectos entre módulos, mientras que la cohesión hace énfasis en las características dentro de los módulos.

  El acoplamiento es la medida de la fuerza de asociación establecida de la conexión de un módulo con otro. Un acoplamiento fuerte complica un sistema, ya que dificulta entender, cambiar o corregir un módulo si está altamente relacionado con otros módulos. La complejidad puede ser reducida diseñando sistemas con acoplamientos débiles entre módulos.

  La cohesión mide el grado de conectividad entre las funciones y elementos de un módulo individualmente. Hay muchas formas de cohesión, la más deseable es la funcional, en la que los elementos de un módulo o componente "trabajan juntos para proveer una conducta bien limitada". La peor forma es la coincidente, en la que las abstracciones son totalmente ajenas en el mismo módulo. Otros tipos de cohesión son: lógica, temporal, de procedimiento, comunicacional, secuencial e informal.

Este principio es manejado por los patrones de diseño propuestos por [8] para organizar las comunicaciones entre componentes, como en "*Client-Dispatcher-Server*" y "*Publisher-Suscriber*".

- **Suficiencia, Completitud y Carácter Primitivo.** Suficiente quiere decir que el componente debe capturar las características de una abstracción que sean necesarias para permitir una interacción significativa y eficaz. Completitud significa que un componente debe capturar todas las características relevantes de su abstracción. Carácter primitivo significa que todas las tareas deben ser ejecutadas de una forma fácil. La mayor meta de un patrón es ser suficiente y completo con respecto a la solución dada de un problema. Muchos patrones son relativamente primitivos y fáciles de implementar, como el patrón "*Strategy*".
- **Separación de Políticas e Implementación.** Un componente de un sistema de software debe tratar con las políticas o la implementación, pero no ambas.

  Un componente de políticas trata con las decisiones inestables o sensibles, con el conocimiento acerca de la semántica y la interpretación de la información, con el ensamblaje de múltiple cálculos disjuntos en un resultado o con la selección de los valores de los parámetros.

  Los componentes de implementación tratan con la ejecución de un algoritmo totalmente especificado en el que no se hacen decisiones inestables o sensibles. El contexto y la interpretación son externos, y son normalmente suministrados por argumentos al componente.

  Debido a su independencia de ciertos contextos, los componentes de implementación únicamente entre ellos son más fáciles de reutilizar y mantener, mientras que los componentes de políticas son frecuentemente específicos de la aplicación y sujetos al cambio.

  Si no es posible separar las políticas y la implementación en diferentes componentes dentro de una arquitectura de software, deben tener al menos clara la separación entre las funcionalidades de política e implementación dentro del componente. El patrón "*Strategy*" se centra en este principio.
- **Separación de Interfaces e Implementación.** Cualquier componente debe consistir en dos partes:
  - La parte de la interfaz que define la funcionalidad provista por el componente y especifica cómo usarlo. Esta interfaz es accesible por los clientes del componente. Una interfaz de exportación de este tipo por lo general consta de firmas de función.
  - La parte de implementación que incluye el código actual para la funcionalidad provista por el componente. Esta parte puede incluir también funciones adicionales y estructuras de datos que son únicamente usadas internamente dentro del componente. No es accesible por los componentes del cliente.

  El principal objetivo de este principio es proteger los componentes del cliente de los detalles de su implementación, y sólo proveer clientes con la especificación de interfaz del componente y las directrices para su uso. Adicionalmente, este principio permite

implementar la funcionalidad de un componente independientemente de su uso por otros componentes. La separación de la interfaz y la implementación es, como en la encapsulación, una técnica para lograr el ocultamiento de la información, principio que hace que "el cliente sólo conozca lo que necesita conocer".

Este principio también apoya a la cambiabilidad, un componente es mucho más fácil de cambiar si su interfaz está separada de su implementación. Esta separación previene que los clientes sean directamente afectados por un cambio. Facilita especialmente la tarea de cambiar la conducta de los componentes o su representación. La separación de la interfaz y la implementación se observa en el patrón "*Bridge*".

- **Único Punto de Referencia.** Cualquier detalle dentro de un sistema de software debe ser declarado y definido sólo una vez. El principal objetivo de este principio es evitar problemas de inconsistencia.
- **Dividir y Conquistar.** Este principio es muy bien conocido, es usado muy fuertemente en las arquitecturas de software. En el diseño "*top-down*", divide una tarea o componente en partes más pequeñas que puedan ser diseñadas independientemente. El patrón "*Whole-Part*" se enfoque en esta técnica. Otros patrones que también aprovechan la subdivisión de manera más genérica, por ejemplo en el patrón "*Microkernel*". Este principio frecuentemente provee una forma de realizar el principio de separación de tareas.

Es importante notar que no todos estos principios son complementarios, algunos son contradictorios y otros principios están muy relacionados entre sí.

2.3.5 Atributos de Calidad de las Arquitecturas de Software Reflejados Dentro de los Patrones Arquitectónicos y de Diseño

Los atributos de calidad de un sistema de software tienen un gran impacto es su desarrollo y mantenimiento, en su operación en general y en el uso de los recursos computacionales. Tienen el mismo impacto que las propiedades funcionales del sistema dentro de la calidad de la aplicación y en su arquitectura. Mientras más grande y complejo sea un sistema de software, y además tenga un tiempo de vida grande, se hacen más importantes los atributos de calidad. Los patrones para arquitecturas de software consideran explícitamente estos aspectos no funcionales o de calidad.

Algunos de los atributos de calidad más importantes para las arquitecturas de software dentro de los patrones arquitectónicos son:

***Cambiabilidad*.** Los sistemas de software industriales y comerciales a gran escala usualmente tienen una vida útil larga, algunas veces de veinte años ó más. Muchas de estas aplicaciones no permanecen estáticas después de su fase original de desarrollo, tienden a evolucionar constantemente durante su tiempo de vida. Los requerimientos existentes cambian y algunos otros son agregados. Para reducir el costo de mantenimiento y la carga de trabajo que genera el cambiar una aplicación, es importante preparar a las arquitecturas para su modificación y evolución.

Parnas [26] menciona respecto al envejecimiento de software lo siguiente: "Los programas, como las personas, envejecen. No podemos prevenir el envejecimiento, pero podemos entender las causas, tomar pasos para limitar sus efectos, revertir temporalmente un poco del daño causado, y prepararnos para el día en que el software no sea más viable".

Parnas también menciona varias razones del envejecimiento del software:

- Por falta de movimiento, el software envejece si no es actualizado frecuentemente.
- Por "cirugía ignorante", los cambios hechos por personas que no entienden el diseño original destruyen gradualmente la arquitectura.
- Si el software es inflexible desde un comienzo.
- Si la documentación es inadecuada, permitiendo que la comprensión del sistema sea erosionada con el tiempo.

El costo del envejecimiento del software se refleja en una creciente inhabilidad para mantenerse al día en el mercado mediante la introducción de nuevas características, un desempeño reducido y la confianza en descenso. Esto puede ser prevenido mediante una documentación apropiada, preservando la estructura al introducir cambios, revisiones intensivas, y diseñando para el cambio a priori [26].

La cambiabilidad tiene cuatro aspectos:

- **Mantenibilidad.** Trata principalmente con la corrección del problema, con la "reparación" de un sistema de software después de que ocurre un error. Una arquitectura de software que está bien preparada para ser mantenida tiende a localizar los cambios y minimizar los efectos secundarios en otros componentes.
- **Extensibilidad.** Se centra en la extensión de un sistema de software con nuevas características, así como con el reemplazo de componentes en versiones mejoradas y la eliminación de características o componentes no deseados o innecesarios. Para lograr la extensibilidad de un sistema de software se requiere de componentes débilmente acoplados. El objetivo es una estructura que permite intercambiar componentes sin afectar a sus clientes. Es necesario tener soporte para integrar nuevos componentes en una arquitectura existente.
- **Reestructuración.** Trata con la reorganización de los componentes de un sistema de software y de las relaciones entre ellos, por ejemplo al cambiar la colocación de un componente moviéndolo a un subsistema diferente. El soporte para reestructurar un sistema de software necesita un diseño cuidadoso de las relaciones entre componentes. Idealmente se debe permitir flexibilidad al configurar los componentes sin afectar grandes partes de su implementación.
- **Portabilidad.** Trata con la adaptación del sistema de software en una variedad de plataformas de usuario, interfaces de usuario, sistemas operativos, lenguajes de programación o compiladores. Para ser portable un sistema de software necesita estar organizado de tal manera que las dependencias del hardware, otros sistemas de software

y ambientes están tomados en cuenta por fuera, en componentes especiales como en un sistema y las bibliotecas de las interfaces de usuario.

Un sistema de software diseñado para cambios también debe soportar la construcción de variantes para clientes diferentes. Muchos patrones permiten la cambiabilidad, por ejemplo el patrón "*Reflection*" y el patrón "*Bridge*".

Finalmente, unas palabras de precaución cuando se diseña para el cambio: Con el creciente uso de los patrones hay personas que han exagerado su uso, causando que las clases no sean simples. Cada "pedazo" de código es altamente "flexible" y se puede adaptar a contextos diferentes, tal flexibilidad tiene un precio. Un software flexible consume generalmente muchos recursos al usar más niveles de indirección o consumo incremental de almacenamiento. También mayor tiempo de análisis y trabajo en codificación. Los buenos diseñadores por consiguiente tratan de decidir por adelantado qué partes del software deben ser altamente flexibles para hacer frente a cambios previsibles, y cuáles partes serán probablemente bastante estáticas. Si prueban mal, hay formas para introducir adicionalmente flexibilidad a través de una reestructuración cuidadosa de las partes del sistema, o usando un patrón que soporte diseño para el cambio. Esta es una opción más económica que la construcción de la cambiabilidad total desde el principio.

***Interoperabilidad*.** El software que forma parte de un sistema no existe independientemente. Frecuentemente interactúa con otros sistemas o ambientes. Para soportar la interoperabilidad una arquitectura de software debe ser diseñada para ofrecer un acceso bien definido a funcionalidades externamente visibles y estructuras de datos. La interacción de un programa con un sistema de software escrito en otro lenguaje de programación es un aspecto de la interoperabilidad que afecta también a la arquitectura de software de una aplicación. Un patrón que maneja interoperabilidad es la arquitectura de "*Broker*".

Eficiencia. Trata con el uso de los recursos disponibles para la ejecución del software, y cómo estos impactan en los tiempos de respuesta, rendimiento y consumo de almacenamiento. No está relacionada únicamente con el uso de algoritmos sofisticados. La distribución apropiada de las responsabilidades a los componentes así como su unión, son importantes actividades arquitectónicas para lograr la eficiencia en una aplicación.

La eficiencia también juega un rol importante en los sistemas de software distribuidos. Los mecanismos IPC (*inter-process communication*) subyacentes en las aplicaciones distribuidas deben ser lo suficientemente rápidas para transferir mensajes e información con suficiente velocidad. Los patrones como "*Forwarder-Receiver*" manejan temas relacionados con la eficiencia. Muchos patrones, de cualquier manera, introducen un nivel adicional de indirección para resolver este problema, que puede hacer disminuir en lugar de aumentar la eficiencia.

Confiabilidad. Trata con la habilidad general del sistema de software para mantener su funcionalidad, tanto frente a la aplicación, como frente a los errores de la aplicación del sistema y en situaciones inesperadas por uso incorrecto. Se distinguen dos aspectos de confiabilidad:

- **Tolerancia a fallas.** Trata de garantizar una comportamiento correcto en caso de errores, y su "reparación interna", como cuando al perder la conexión de un componente remoto en un sistema de software distribuido, subsecuentemente se reconecta a él. Después de reparar el error, el sistema de software debe reanudar o repetir la ejecución de la operación que estaba en progreso cuando ocurrió el error.
- **Robustez.** Trata de proteger la aplicación contra el uso incorrecto y de las entradas corruptas o deterioradas, manteniendo la aplicación en un estado definido en caso de errores inesperados. En contraste con la tolerancia a fallas, en la robustez no significa que el software es capaz de continuar con las operaciones que ser realizaban al momento de surgir el error, no garantiza que el software termine de una forma definida.

La arquitectura de software tiene un impacto mayor en la confiabilidad del sistema de software. Ejemplos de cómo una arquitectura de software soporta la confiabilidad puede ser al incluir redundancia en una aplicación, o por medio de la integración de monitoreo de componentes y manejo de excepciones. Un patrón que muestra como soportar aspectos específicos de confiabilidad es el patrón "*Master-Slave*".

Facilidad de prueba. Con el incremento del tamaño y la complejidad de los sistemas de software, especialmente en los industriales, el hacer pruebas es más caro y difícil. Un sistema de software necesita soporte de su arquitectura de software para facilitar la evaluación de su corrección, demostrar la corrección está fuera del alcance en la mayoría de los casos. Las estructuras de software que soportan la facilidad de pruebas permiten una mejor detección de fallas y su corrección, también permiten una integración temporal de código y componentes de depuración.

Aunque los patrones descritos por [8] no manejan explícitamente la facilidad de pruebas, muchos de ellos tienen un impacto mayor en esta tarea, como es el caso del patrón "*Command Processor*" que facilita las pruebas a nivel de la interacción del usuario permitiendo el registro y reproducción de objetos de comando de usuario. El patrón "*Broker*" facilita las pruebas de clientes individuales y componentes del servidor en un sistema distribuido. Esta arquitectura libera a los componentes de dependencias de sus asociados de comunicación y de los mecanismos de comunicación que usa.

El patrón "*Broker*", complica probar la colaboración entre clientes y servidores, porque introduce componentes adicionales para soportar su independencia. A diferencia de las implementaciones en que los clientes y servidores están fuertemente unidos, el depurar un error en la entrega de un mensaje de un cliente a un servidor es una tarea más difícil, esto debido a que muchos componentes distintos están envueltos en el despliegue y no despliegue de información, y con el envío de mensajes a través de los límites del proceso.

Reusabilidad o Reutilización. Es actualmente uno de los temas más discutidos en la ingeniería de software. Promete una reducción de costo y tiempo de desarrollo de los sistemas de software, así como una mejor calidad del software [27]. Adele Goldberg [28] define a la reutilización como "el acto de lograr lo que es deseado con la ayuda de lo que ya existe". La reutilización tiene dos

aspectos: el desarrollo de software con reutilización y el desarrollo de software para la reutilización.

- El desarrollo de software con reutilización, significa reutilizar componentes existentes y resulta de proyectos previos o de bibliotecas comerciales, del análisis de diseño, de las especificaciones de diseño o del código de los componentes. Estos artefactos reutilizables son integrados a la aplicación bajo desarrollo, como son o con modificaciones. Practicar el desarrollo de software con reutilización requiere de la construcción de arquitecturas de software que permitan "conectar" las estructuras prefabricadas y del código de componentes. También permite soportar la composición de software, lo que significa adaptar una aplicación fuera de los componentes existentes adaptándolos a las necesidades del desarrollo e implementando los componentes que "pegan" para conectarlos.
- El desarrollo de software para la reutilización , se centra en producir componentes que son reutilizables potencialmente en futuros proyectos como parte del desarrollo de software actual. Requiere de arquitecturas de software que permitan partes autónomas que deberán tomarse de la aplicación en desarrollo y reusadas en otros sistemas sin modificaciones importantes.

Aunque los patrones propuestos por [8] no manejan la reusabilidad explícitamente, casi todos los patrones que soportan la cambiabilidad también soportan la reusabilidad. Por ejemplo el patrón "*Model-View-Controller*" soporta el intercambio de vistas y controladores, y la reusabilidad del modelo.

Algunos atributos de calidad requieren de técnicas arquitectónicas similares para alcanzarlas, por ejemplo el diseño de la reusabilidad y la cambiabilidad. Otros sirven con un propósito global similar, por ejemplo el diseño de la portabilidad e interoperabilidad tratan con la integración de los sistemas de software en un ambiente, mientras que la confiabilidad y la eficiencia tratan con la usabilidad en general.

Los atributos de calidad pueden contradecirse entre ellos así como complementarse. Por ejemplo, cuando se describe la funcionalidad de una aplicación para lograr la tolerancia a fallas, la estructura resultante es usualmente poco eficiente y más cara que una estructura sin tal redundancia. Cuando se especifican los atributos de calidad para una arquitectura de software, se necesita explícitamente considerar las interdependencias y compensaciones que existirán entre estos. También se necesita especificar el orden de prioridad entre diferentes atributos de calidad, para definir la preferencia de un requerimiento contra otro en caso de conflicto.

Aunque los atributos de calidad son muy importantes en las arquitecturas de software, su logro es difícil de medir. Los criterios detallados que una arquitectura de software debe satisfacer son únicamente especificados por pocas propiedades, por ejemplo la reusabilidad y la cambiabilidad. Por esta razón, estimar el grado en que una arquitectura logra una determinada propiedad no funcional sigue estando basada en la experiencia de los ingenieros de software.

2.4 Clasificación de Estilos Arquitectónicos

Un estilo arquitectónico de acuerdo con [3], define una familia de sistemas en términos de un patrón de organización estructural. Dentro de un estilo arquitectónico se determina el vocabulario de componentes y conectores que se pueden usar, juntos forman un conjunto de constantes que se pueden combinar de una forma en particular.

Una clasificación es la siguiente:

- Abstracción de Datos y Organización Orientada a Objetos
- Flujo de Datos
  - Secuencial en lotes
  - Red de flujo de datos (Tuberías y filtros)
  - Control de procesos
- Orientadas en Datos
  - Repositorios
  - Pizarra
- Jerárquicas
  - Principal - Subrutina (Main-Subrutine)
  - Maestro - Esclavo (Master-Slave)
  - Capas (Layers)
  - Máquina virtual (Virtual Machine)
- Comunicación Implícita Asíncrona
  - Invocación Implícita Basada en Eventos sin Búfer
  - Invocación Implícita Basada en Mensajes con Búfer
- Orientadas a Interacción
  - Model-View-Controller (MVC)
  - Presentation-Abstraction-Control (PAC)
- Distribuidas
  - Cliente - Servidor
  - Multi-niveles
  - Broker
  - Orientadas a servicios (SOA)
- Basadas en Componentes
- Heterogéneas

2.4.1 Abstracción de Datos y Organización Orientada a Objetos

En este estilo las representaciones y sus operaciones primitivas asociadas son encapsuladas en un tipo de datos abstracto o objeto. Los componentes para este estilo son los objetos o en su caso instancias de los tipos de datos abstractos.

A los objetos también se les denomina “administradores” porque son responsables de preservar la integridad de los recursos a través de una representación (que la esconde de otros objetos). Los objetos interactúan a través de funciones o por la invocación de procedimientos.

El uso de tipos de datos abstractos, y el incremento en el uso de sistemas orientados a objetos es ampliamente extendido. Por ejemplo, algunos sistemas permiten "objetos" en tareas concurrentes; otros permiten a los objetos tener múltiples interfaces.

Dentro de sus beneficios:

- Debido a que el objeto esconde su representación de los clientes, es posible cambiar su implementación sin afectarlos.
- La agrupación de un conjunto de rutinas de acceso a los datos permite a los diseñadores descomponer los problemas en una colección de agentes interactivos.

Su principal limitación:

- Los objetos deben conocer la identidad de los objetos con los que van a interactuar, es decir que si se cambia un objeto en algún punto, se debe modificar a todos aquellos objetos que lo invocan.

2.4.2 Flujo de Datos

El estilo arquitectónico de flujo de datos trata a un sistema de software como una serie de transformaciones en conjuntos sucesivos de datos, donde los datos y las operaciones en él son independientes entre sí. El sistema de software es descompuesto en elementos de procesamiento de datos en donde los datos dirigen y controlan el orden del procesamiento. Cada componente en esta arquitectura transforma sus datos de entrada en sus correspondientes datos de salida. La conexión entre los componentes de los subsistemas pueden ser implementados como flujos de entrada/salida, archivos de entrada/salida, buffers, entre otros tipos de conexiones. Los datos pueden fluir en una gráfica topológica con ciclos o en una estructura lineal sin ciclos, e inclusive en una estructura de árbol. Independientemente del tipo de topología, los datos se mueven desde un subsistema a otro. En general, no hay interacción entre módulos excepto por las salidas y las conexiones de datos de entrada. En otras palabras, los subsistemas son independientes entre ellos de tal manera que un subsistema puede ser sustituido por otro sin afectar el resto del sistema, siempre que el nuevo subsistema sea compatible con el formato de los datos de entrada y salida correspondientes. Dado que cada subsistema no necesita conocer la identidad de otro subsistema, la modificabilidad y reusabilidad son atributos importantes de las arquitecturas de flujos de datos.

Hay diferentes formas de conectar los datos de salida de un módulo a la entrada de otros módulos, lo que resulta en una gama de arquitecturas de flujo de datos. Hay dos categorías de secuencias de ejecución entre módulos: Secuencial en lotes (*Batch Sequential*) y de Red de flujo de datos ó de Tuberías y Filtros (*Nonsequential Pipeline Model*).

La arquitectura de flujo de datos es aplicable en los dominios de ciertos problemas, como en aplicaciones en la que estén involucradas una serie de transformaciones o cálculos independientes de datos, a través de entradas y salidas bien definidas y ordenadas. Los ejemplos típicos son los compiladores y el procesamiento de datos de negocio secuenciales. Ninguno de estos requieren de la interacción del usuario.

Los detalles de las subcategorías mencionadas anteriormente se presentan a continuación.

2.4.2.1 Secuencial en Lotes

El estilo arquitectónico secuencial en lotes representa un modelo de procesamiento de datos tradicional ampliamente usado entre 1950 y 1970. Dos lenguajes típicos de programación que trabajan con este modelo son RPG y COBOL .

En la arquitectura secuencial en lotes, cada subsistema o módulo de transformación de datos empieza su proceso hasta que su subsistema previo completa sus cálculos. El flujo de datos lleva una serie de datos en conjunto de un subsistema a otro. La Figura 3 muestra un ejemplo típico de este estilo.

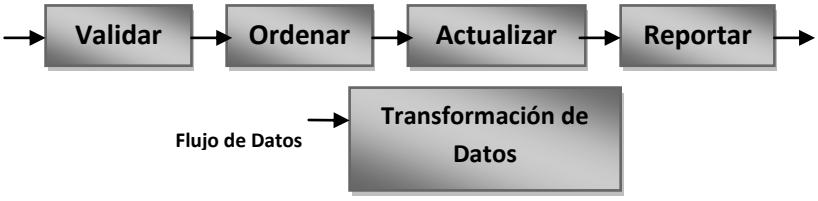

Figura 3. Arquitectura Secuencial en Lotes [12]

En el ejemplo de la Figura 3, el primer subsistema valida las solicitudes de transacción (insertar, borrar y actualizar) en su totalidad. Luego el segundo subsistema ordena todos los registros de las transacciones en orden ascendente de las llaves primarias de los registros de datos para acelerar la actualización en el archivo maestro. Posteriormente, la transacción del módulo actualiza el archivo maestro con las solicitudes ordenadas de la transacción, y el módulo de reporte genera una nueva lista. Esta arquitectura es de un flujo de orden lineal.

Todas las comunicaciones (las flechas funcionan como enlace de conexión) entre los módulos de los subsistemas son conducidas a través de archivos intermedios transitorios que pueden ser removidos por subsistemas sucesivos. Generalmente se usa un script para realizar la secuencia de lotes de subsistemas en un sistema. Aplicaciones típicas de esta arquitectura son el procesamiento de datos del negocio como en los bancos y en la facturación de utilidades, así como en modelos matemáticos, físicos u astronómicos, por mencionar algunos ejemplos.

Dominios aplicables para la arquitectura secuencial en lotes son:

- En aplicaciones donde los datos están lotes.
- Los archivos intermedios son archivos de acceso secuencial.
- Cada subsistema lee sus archivos relacionados de entrada y escribe archivos de salida.

Algunos de los beneficios que ofrece:

- Divisiones simples en subsistemas.
- Cada subsistema puede ser un programa autónomo que trabaje con datos de entrada para producir datos de salida.

Algunas limitaciones son:

- La implementación requiere de control externo.
- No provee de una interfaz interactiva.
- No soporta concurrencia, por lo tanto este tipo de arquitectura tiene un rendimiento bajo.
- Debido a la forma en que trabaja tiene una alta inactividad.

2.4.2.2 Red de Flujo de Datos (*Pipe and Filter*)

Este es otro tipo de arquitectura donde el flujo es conducido por los datos. Esta arquitectura descompone todo el sistema en componentes de datos de origen, filtros, tuberías y consumidores de datos (*data sink*). Las conexiones entre componentes son los flujos de datos. Una propiedad particular para este tipo de arquitectura es su ejecución concurrente e incremental.

Un flujo de datos es un búfer del tipo primeras entradas/primeras salidas (*first-in/first-out*) que puede ser un flujo de bytes, caracteres e inclusive archivos XML o de otro tipo. La mayoría de los sistemas operativos y lenguajes de programación proveen un mecanismo de flujo de datos; esto es una herramienta importante para serializar y deserializar (*marshaling y unmarshaling*) en cualquier sistema distribuido.

Cada filtro es un transformador de flujos de datos independiente, lee los datos de su flujo de datos de entrada, transforma y procesa los datos, y luego escribe los datos transformados del flujo de datos sobre una tubería para su procesamiento en el siguiente filtro. Un filtro no necesita esperar por datos en lotes como un todo. Tan pronto como los datos lleguen a través de la tubería conectada al filtro, este empieza a trabajar de inmediato. Un filtro no conoce la identidad del flujo superior e inferior. Un filtro trabaja en un modo local e incremental (las salidas empiezan antes de que se consuman las entradas).

Una tubería transporta un flujo de entrada de un filtro a otro, puede transportar flujos binarios o caracteres. Una tubería es colocada entre dos filtros, estos pueden ejecutar en hilos diferentes el mismo proceso como en los flujos de entrada y salida de Java.

Existen tres formas para hacer el flujo de datos:

- *Push only (Write only)*
  - La fuente de datos empuja los datos al flujo inferior.
  - El filtro debe empujar los datos al flujo inferior.
- *Pull only (Read only)*
  - Un consumidor de datos jala los datos del flujo superior.
  - Un filtro debe jalar los datos del flujo superior.
- *Pull/Push (Read/Write)*
  - Un filtro puede jalar datos del flujo superior y empujar los datos transformados al flujo inferior.

Nota: push - envío, pull - extraer

Hay dos tipos de filtros:

- Los filtros activos jalan los datos y los datos transformados los empujan fuera (*pull/push*). Trabajan con los tuberías pasivas que proveen mecanismos de lectura/escritura para jalar y empujar. El mecanismo de tuberías y filtros de Unix adopta este modo.
- Los filtros pasivos permite conectar tuberías de envío y extracción de datos. Trabaja con tuberías activas que extraen los datos de los filtros y meten los datos al siguiente filtro. El filtro debe proveer en este caso mecanismos de lectura/escritura. Es parecido a la arquitectura de flujo de datos.

Los ejemplos mejor conocidos de este estilo arquitectónico son los programas escritos en el *shell* de Unix. Unix soporta este estilo proveyendo una notación para conectar los componentes (en este caso los procesos) y mecanismos que permiten implementar los filtros en tiempo de ejecución.

Otro ejemplo es el de los compiladores, tradicionalmente son vistos como sistemas "*pipeline*" aunque las fases no son incrementales. Los pasos en el "*pipeline*" incluyen el análisis léxico, análisis (*parsing*), análisis semántico, generación de código.

Otros ejemplos se encuentran en dominios de procesamiento de señales, en programación funcional y sistemas distribuidos.

Dentro de sus beneficios se encuentran:

- Permitir al diseñador comprender el comportamiento total de entrada/salida del sistema como una composición del comportamiento individual de los filtros.
- Concurrencia, proporciona un alto rendimiento global ante el procesamiento excesivo de datos.
- Reusabilidad, la encapsulación de los filtros hace fácil su sustitución, sus componentes funcionan como elementos "*plug and play*".
- Modificabilidad, la estructura de esta arquitectura cuenta con bajo acoplamiento entre filtros, al añadir nuevos filtros el impacto en su estructura es mínimo y permite realizar modificaciones en la implementación de cualquier filtro existente siempre que y cuando las interfaces de entrada/salida no se hayan modificado. Su estructura también facilita el mantenimiento y el mejoramiento.
- Simplicidad, ofrece una división clara entre dos filtros cualquiera conectados por una tubería
- Flexibilidad, soporta una ejecución tanto secuencial como paralela.
- Permite el análisis especializado sobre rendimiento y puntos muertos.

Sus desventajas/limitaciones son:

- A menudo conducen al procesamiento organizado por lotes.
- Esta arquitectura no es adecuada cuando se tienen interacciones dinámicas.

- El diseñador debe proveer de una transformación completa de entradas y salidas para cada filtro.
- Se requiere de un bajo denominador común para la transmisión de datos en los formatos ASCII ya que los filtros necesitan manipular los flujos de datos en diferentes formatos, por ejemplo en registros, en XML en lugar de caracteres.
- Hay sobrecarga en la transformación de los datos entre dos filtros consecutivos.
- Puede ser difícil configurar dinámicamente las tuberías y los filtros en un sistema.

2.4.2.3 Arquitectura para Procesos de Control

Este tipo de arquitectura es adecuada cuando se presenta un diseño de sistemas de software embebido, donde el sistema es manipulado por un proceso de control de datos variables. La arquitectura de control de procesos descompone todo el sistema en subsistemas (módulos) y conexiones entre subsistemas. Hay dos tipos de subsistemas: Una unidad de procesamiento ejecutora para cambiar las variables de control de procesos y una unidad controladora para calcular la cantidad de cambios. La Figura 4 muestra el flujo de datos de la retroalimentación en un ciclo cerrado de un sistema de control de procesos. Las conexiones entre los subsistemas son el flujo de datos.

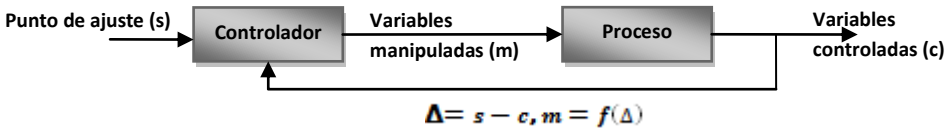

Figura 4. Flujo de Datos en la Arquitectura para Procesos de Control [12]

Un sistema de procesos de control debe tener los siguientes datos de control de procesos:

- Variable controlada, consiste en una variable que se busca controlar como en el caso de la velocidad en un sistema de control de cruceros. Tiene un punto de meta a alcanzar. Los datos de la variable controlada pueden ser medidos por sensores con una referencia retroalimentada para volver a calcular las variables manipuladas.
- Variable de entrada, datos de entrada medidos como la temperatura de retorno de aire en un sistema de control de temperatura.
- Variable manipulada, esta puede ser ajustada por el controlador.

Las variables de entrada y las manipuladas son aplicadas en la ejecución del procesador lo que resulta en una variable controlada. El punto de ajuste y las variables controladas son los datos de entrada al controlador, la diferencia entre el valor de la variable controlada y el valor del punto de ajuste es usada para alcanzar un nuevo valor manipulado. Ejemplos de aplicaciones para este tipo de arquitectura de software de control de procesos son el control de cruceros de automóviles y los sistemas de control de temperatura de edificios.

Dominios aplicables para la arquitectura de procesos de control son:

- Sistemas de software embebido que involucren acciones continuas.

- Sistemas que necesiten mantener datos de salida en un nivel estable.
- Algún sistema en el que se necesite tener un punto de ajuste o meta que deba alcanzar a nivel operacional.

Los beneficios que la arquitectura de procesos de control con retroalimentación en ciclo cerrado sobre una arquitectura abierta son:

- Ofrece una mejor solución para el control de aquellos sistemas donde no hay una fórmula precisa proporcionada al controlador para decidir sobre la variable manipulada.
- El software puede ser completamente embebido en los dispositivos.

2.4.3 Orientadas en Datos

Las arquitecturas de software centradas en datos se caracterizan por un almacén centralizado de datos que es compartido por todos los componentes de software que lo rodean. El sistema de software es descompuesto en dos particiones principales: el almacén de datos y los componentes de software independientes (agentes). Las conexiones entre el módulo de datos y los componentes de software son implementados tanto por el método de invocación explícita como por el método de invocación implícita. En arquitecturas de software puramente centradas en datos, los componentes de software no se comunican entre ellos directamente, todas las comunicaciones son conducidas a través del almacén de datos. El módulo de datos compartido provee de todos los mecanismos para que los componentes de software puedan acceder a él; Los mecanismos son del tipo: inserción, borrado, actualización y recuperación.

Existen dos categorías de arquitecturas centradas en datos: *repositorio* y *pizarrón*. Se diferencian por la estrategia de control de flujo.

El almacén de datos en la arquitectura del repositorio es pasiva; esto es, los clientes (componentes de software o agentes) controlan el flujo lógico. Los cliente pueden acceder al repositorio interactivamente o por una solicitud de transacción por lotes. El estilo de repositorio es ampliamente usado en los sistemas de administración de bases de datos, sistemas de información de bibliotecas, compiladores, en ambientes CASE (*Computer Aided Software Engineering*), en todos los ambientes de desarrollo interactivos (IDE) y en kits similares de desarrollo de software son buenos ejemplos del dominio para la arquitectura de repositorio. En general son ampliamente usados en sistemas de administración de información complejos donde uno de los temas más importantes a considerar es una administración de datos fiable.

El almacén de datos en la arquitectura de pizarrón es activa y sus clientes son pasivos; esto indica que el flujo de la lógica es determinado por el estado de los datos actuales en el almacén de datos. Los clientes del pizarrón se llaman fuentes de conocimiento, oyentes o suscriptores. Un nuevo cambio en los datos desencadena eventos en los que las fuentes de conocimiento toman acción para responder ante ellos. Estas acciones pueden resultar en nuevos datos, los cuales pueden cambiar el flujo lógico, esto puede suceder continuamente hasta que la meta es alcanzada. Muchas aplicaciones diseñadas con la arquitectura de pizarrón incluyen los sistemas de

inteligencia artificial basados en conocimiento , sistemas de reconocimiento de voz e imágenes, sistemas de seguridad, sistemas de administración de recursos del negocio, etc.

La Figura 5 muestra un diagrama de bloques general de las arquitecturas centradas en datos. Las líneas sólidas del diagrama describen enlaces de datos bidireccionales (obtener y guardar datos), mientras que las líneas punteadas describen enlaces de control de flujo bidireccionales (control sobre los datos o control sobre los agentes).

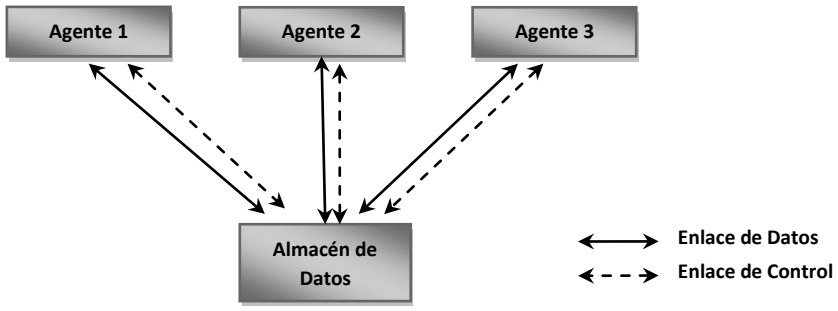

Figura 5. Diagrama de Bloques de una Arquitectura Centrada en Datos Típica [12]

2.4.3.1 Estilo Arquitectónico de Repositorio

El estilo arquitectónico de repositorio es una arquitectura centrada en datos que soporta interacción de los usuarios para el procesamiento de datos (en comparación con el procesamiento por lotes de transacciones secuenciales). Los agentes del componente de software del almacén de datos controlan el cálculo y el flujo lógico del sistema. La Figura 6 muestra una imagen general de la arquitectura de repositorio. Las líneas punteadas que apuntan hacia el repositorio indican que los clientes del repositorio tienen control total sobre el flujo lógico. Los clientes pueden obtener y colocar datos en el almacén de datos. Clientes diferentes pueden tener interfaces diferentes y diferentes privilegios de acceso a los datos.

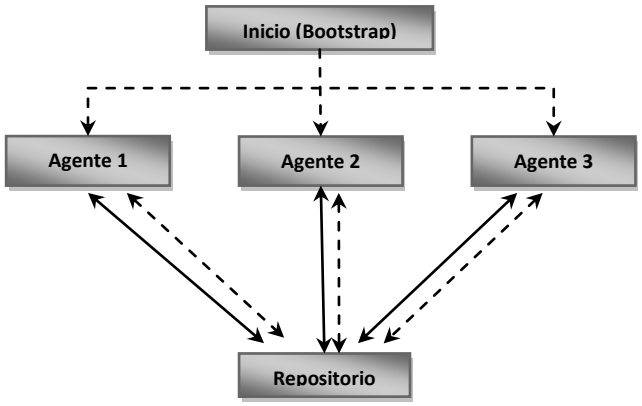

Figura 6. Arquitectura de Repositorio [12]

Como se mencionó anteriormente algunos de los dominios típicos de diseño para la arquitectura de repositorio son los sistemas de administración de bases de datos relacionales, sistemas CASE, construcción de compiladores principalmente.

Existen variantes de la arquitectura de repositorio como el repositorio virtual y el repositorio descentralizado (distribuido). El repositorio virtual es construido en la parte superior de múltiples repositorios físicos. Muchos sistemas de bases de datos permiten a los usuarios o desarrolladores crear vistas que son repositorios virtuales dado que no existen físicamente. Este enfoque simplifica la complejidad de toda la estructura de la base de datos, puede proveer también una administración de seguridad para privilegios de autoridad en términos del alcance de los datos y de los tipos de manipulación para diferentes usuarios y grupos.

En un sistema de repositorio distribuido, también conocido como sistema de base de datos distribuidos o sistema de información de la empresa, todos los datos son distribuidos sobre todos los sitios enlazados por la red. Los datos son replicados para mejorar la fiabilidad y la accesibilidad local.

Dominios aplicables para la arquitectura de repositorio:

- Es adecuada para sistemas de información largos y complejos donde muchos clientes componentes del software necesitan acceder al repositorio en maneras diferentes.
- Cuando se requiera transacciones de datos para conducir el flujo de control o los cálculos.

Algunos de sus beneficios:

- Integridad de los datos, facilidad para respaldar y restaurar datos.
- Escalabilidad de los sistemas y reusabilidad de los agentes, facilidad para agregar nuevos componentes de software porque no tienen comunicación directa entre ellos.
- Reduce la sobrecarga de datos transitorios entre componentes de software.

Algunas de sus limitaciones:

- La confiabilidad del almacén de datos y la disponibilidad son temas a considerar más profundamente al elegir este tipo de arquitectura. Los repositorios centralizados son más vulnerables comparados con los repositorios distribuidos con replicación de datos.
- Tiene una alta dependencia entre la estructura de los datos, el almacén de datos y sus agentes. Cambios en la estructura de los datos tienen un impacto significativo en sus agentes. La evolución de los datos es más difícil y cara.
- Hay un alto costo al mover los datos por la red si los datos son distribuidos.

Arquitecturas relacionadas: Capas, Multi-niveles y MVC.

2.4.3.2 Estilo Arquitectónico de Pizarrón

La arquitectura de pizarrón fue desarrollada para aplicaciones de reconocimiento de diálogos en los 70´s. Otras aplicaciones para este tipo de arquitectura son los sistemas de reconocimiento de patrones en imágenes y sistemas de emisión del clima. Ejemplos típicos de esta arquitectura son el sistema experto de reconocimiento de diálogos Hearsay-II y el sistema de análisis de estructura molecular CRYSTALIS.

La palabra pizarrón surge dada la similitud que tiene este tipo de arquitectura con un salón de clases donde se enseña y se aprende, donde los maestros y los estudiantes pueden compartir datos para resolver problemas en clase a través del pizarrón. Los estudiantes y los maestros juegan el rol de agentes que contribuyen para resolver un problema. Pueden trabajar en paralelo o independientemente para encontrar la mejor solución.

La idea de la arquitectura del pizarrón es similar a la del pizarrón usado para resolver problemas sin resultados deterministas. Todo el sistema es descompuesto en principalmente en dos particiones, una de estas particiones es llamada el "pizarrón", que es usado para almacenar datos (hipótesis y hechos), y la otra partición es denominada las "fuentes de conocimiento" que almacenan el conocimiento específico del dominio. Puede haber una tercera partición llamada "controlador" que es usado para iniciar el pizarrón y las fuentes de conocimiento, es decir toma el rol de un iniciador (*bootstrap*) y de control de supervisión global.

Las conexiones entre el subsistema de pizarrón y las fuentes del conocimiento son básicamente invocaciones implícitas desde el pizarrón hacia una fuente de conocimiento específica, que están registradas en el pizarrón de antemano. Al cambiar los datos en el pizarrón se activan una o más fuentes de conocimiento para un procesamiento continuo. Los cambios en los datos pueden ser causados por nueva información deducida o por resultados de hipótesis de las fuentes de conocimiento. Esta conexión puede ser implementada en el modo publicar/suscribir.

La Figura 7 ilustra el diagrama de bloques para la arquitectura de pizarrón. Las líneas sólidas indican enlaces de datos, mientras que las líneas punteadas representan el control del flujo lógico, el cual es controlado por cualquier cambio en los datos en el almacén de datos, esto es, los datos en el almacén del pizarrón dirigen el flujo de los cálculos.

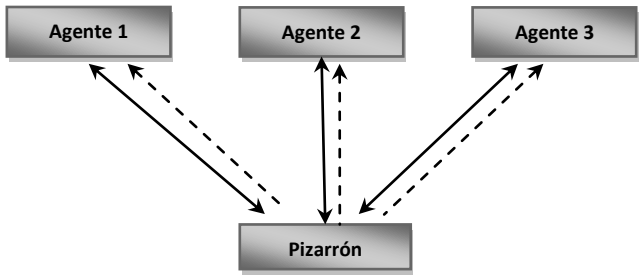

Figura 7. Arquitectura de Pizarrón [12]

Dominios aplicables para la arquitectura de pizarrón son:

- Es adecuada para resolver problemas sin límites fijos (abiertos) y complejos como en los problemas de inteligencia artificial donde no hay soluciones preestablecidas.
- Donde los problemas abarcan múltiples disciplinas, y cada problema envuelve completamente diferentes tipos de experticia en el conocimiento. También donde se requiera de cooperación en los paradigmas de resolución de problemas.
- Donde una solución parcial o aproximada es aceptable.

- Donde una búsqueda exhaustiva es imposible e impráctica debido a que el conocimiento disponible o las hipótesis no estén completas o no son exactamente veraces.

Algunos de sus beneficios:

- Escalabilidad, ofrece facilidad para añadir o actualizar las fuentes de conocimiento.
- Concurrencia, todas las fuentes de conocimiento pueden trabajar en paralelo gracias a que son independientes entre ellas.
- Soporta experimentación para hipótesis.
- Reusabilidad de los agentes de fuentes de conocimiento.

Algunas de sus limitaciones:

- Debido a la dependencia cerrada entre el pizarrón y las fuentes de conocimientos, el cambio en la estructura del pizarrón puede tener un impacto significante en todos sus agentes.
- Dado que se esperan únicamente soluciones parciales o aproximaciones, puede ser difícil decidir en qué momento debe terminar el razonamiento.
- Una dificultad es la sincronización de múltiples agentes. Debido a que múltiples agentes trabajan y comparten datos en el pizarrón simultáneamente, debe ser coordinada la preferencia o prioridad de ejecución de múltiples agentes.
- Es un desafío la depuración y la realización de pruebas.

Arquitecturas relacionadas: Arquitecturas de invocación implícita como las basadas en eventos y la arquitectura del MVC.

2.4.4 Jerárquicas

Las arquitecturas de software jerárquicas se caracterizan por ver a un sistema como una estructura jerárquica. Los sistemas de software son descompuestos en módulos lógicos (subsistemas) a niveles diferentes en la jerarquía. Los módulos a niveles diferentes son conectados por métodos de invocación implícitos y explícitos. En otras palabras, un módulo de bajo nivel provee servicios a los módulos adyacentes de nivel superior, los cuales invocan los métodos o procedimientos de los niveles bajos. En los lenguaje de procedimientos, las funciones y los procedimientos deben ser organizados en archivos de encabezado o en una biblioteca. Para hacer uso de los servicios, un módulo de nivel superior debe llamar a las funciones o procedimientos de esos archivos o a las bibliotecas. En las implementaciones orientadas a objetos de este estilo arquitectónico, los servicios son organizados en un paquete de clases, este paquete es importado por los módulos de nivel superior para obtener los servicios necesarios haciendo llamadas a las correspondientes operaciones de las clases.

Los sistemas de software son típicamente diseñados usando los estilos arquitectónicos jerárquicos, ejemplos de estos incluyen .NET de Microsoft, el sistema operativo UNIX, TCP/IP. etc. Algo que tienen en común es que los servicios a niveles bajos proveen funcionalidades más específicas hacia los servicios fundamentales de utilidad, como los servicios E/S (I/O),

transacciones, planeación, servicios de seguridad, etc. Las capas de intermedias de una aplicación proveen de funciones más dependientes del dominio como la lógica del negocio o los servicios básicos de procesamiento. Las capas superiores proveen una funcionalidad más abstracta en forma de interfaces de usuario como en los intérpretes de líneas de comando, GUIs, servicios de programación en *shell*, etc. Cada capa provee servicios a su capa inmediata superior. Cualquier cambio a una capa específica afecta únicamente a su capa adyacente superior sólo que su interfaz sufra cambios, de otra manera no hay efectos.

Esta categoría de arquitectura se caracteriza por la estructura jerárquica y los estilos de las conexiones por métodos de invocación explícita (*call-and-return*). Es también usada en la organización de las bibliotecas de clases como en la API de Java en paquetes jerárquicos.

Una variedad de tipos de diseño (orientada a procedimientos, orientada a objetos, orientada a dominios específicos) pueden implementar las arquitecturas de software jerárquicas.

Este estilo arquitectónico puede trabajar junto con otros estilos, de hecho es difícil encontrar diseños de software que únicamente usen este estilo. La estructura jerárquica es uno de los estilos más populares que se combinan con otros estilos.

Hay cuatro estilos particulares que son jerárquicos: Principal-Subrutina, Maestro-Esclavo, Capas y Máquina Virtual.

La Figura 8 muestra el diagrama de bloques de una típica arquitectura de software jerárquica.

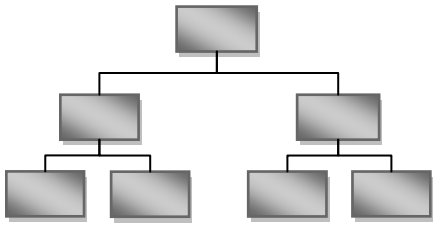

Figura 8. Arquitectura Jerárquica [12]

2.4.4.1 Principal - Subrutina *(Main - Subrutine)*

El diseño de la arquitectura de principal-subrutina ha dominado las metodologías de diseño de software por mucho tiempo. El propósito de este estilo arquitectónico es reutilizar subrutinas y tener subrutinas individuales desarrolladas independientemente. En el paradigma de procedimientos clásico, los datos son típicamente compartidos con subrutinas relacionadas al mismo nivel. En la orientación a objetos los datos son encapsulados en cada objeto individual así que la información está protegida. Las personas se refieren al estilo principal-subrutina como un estilo más tradicional que el orientado a objetos.

Usando este estilo, los sistemas de software son descompuestos en subrutinas jerárquicas refinadas de acuerdo la funcionalidad deseada del sistema. Los refinamientos se llevan a cabo verticalmente hasta que la subrutina descompuesta es lo suficientemente simple para tener una

sola responsabilidad independiente y esta funcionalidad puede ser reusada y compartida por múltiples llamados de las capas superiores.

Los datos se pasan como parámetros a subrutinas de las llamadas. Hay dos formas de pasar parámetros:

- Paso por referencia, donde la subrutina puede cambiar el valor del dato referenciado por el parámetro, y
- Paso por valor, donde la subrutina sólo usa el dato pasado pero no lo puede cambiar.

Otra forma menos frecuente es el paso por nombre, dependiendo de la tecnología de implementación usada. En ésta se puede pasar en una referencia a un procedimiento o una función para implementar lo que se conoce como retrollamado o *callback*.

Típicamente un programa principal maneja el control sobre la secuencia de los llamados a subrutina al menos una vez en el ciclo de invocaciones en el mismo orden. La Figura 9 muestra un ejemplo de la participación de las subrutinas del estilo jerárquico principal-subrutina.

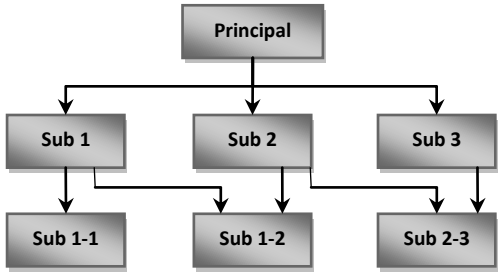

Figura 9. Arquitectura Principal-Subrutina [12]

Este estilo arquitectónico hace un mapeo entre la especificación de requerimientos a su estilo de diseño a través de los diagramas de flujo de datos (DFD), el procedimiento se explica brevemente en [12] y una explicación más profunda en [17].

Su dominio es aplicable a todos aquellos sistemas que elijan tener este estilo arquitectónico considerando sus beneficios y limitaciones.

Algunos de sus beneficios:

- Es fácil de descomponer el sistema basado en la definición de tareas en un refinamiento *top-down*.
- Esta arquitectura puede ser usada en un subsistema de diseño orientado a objetos.

Sus limitaciones:

- En el estilo clásico principal-subrutina al compartir datos globalmente se tienen mayor cantidad de vulnerabilidades.

- Acoplamientos estrechos ocasionará mayor número de efectos dominó al realizar cambios comparado con un diseño orientado a objetos.

2.4.4.2 Maestro-Esclavo *(Master-Slave)*

La arquitectura maestro-esclavo es una variante del estilo arquitectónico principal-subrutina que soporta tolerancia a fallas y confiabilidad en el sistema. En este tipo de arquitectura, los esclavos proveen servicios de replicación al maestro, y el maestro selecciona un resultado particular entre esclavos de acuerdo con ciertas estrategias de selección. Los esclavos ejecutan la misma tarea funcional por diferentes algoritmos y métodos o una funcionalidad totalmente diferente.

La Figura 10 muestra la arquitectura maestro-esclavo donde todos los esclavos implementan el mismo servicio. El maestro configura las invocaciones de los servicios replicados y recibe los resultados de regreso de todos los esclavos, luego determina cuales de los resultados regresados serán seleccionados.

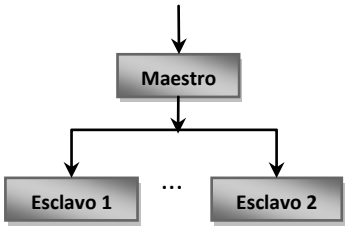

Figura 10. Diagrama de Bloques para la Arquitectura Maestro-Esclavo [12]

Otras características de esta arquitectura incluyen el cómputo paralelo y la exactitud de los cálculos. Todos los esclavos pueden ser ejecutados en paralelo. Dado que la misma tarea puede ser delegada a varias implementaciones diferentes, los resultados pueden ser descartados con una estrategia de mayoría de votos u otros algoritmos.

Dentro del dominio de aplicación se considera esta arquitectura en aquellos sistemas de software donde la confiabilidad es crítica, esto se logra debido a la replicación (redundancia) de servidores.

2.4.4.3 Capas (*Layers*)

Como su nombre sugiere, un sistema arquitectónico de capas se descompone en un número de capas altas y bajas en una jerarquía, cada capa consiste de un grupo de clases relacionadas que son encapsuladas en un paquete, en un componente de despliegue, o como un grupo de subrutinas en el formato de bibliotecas de métodos o de archivos de cabecera. También, cada capa tiene una sola responsabilidad en el sistema.

Una solicitud a la $capa_{i+1}$ invoca los servicios proveídos por la $capa_i$ a través de la interfaz de la capa. La respuesta regresa a la $capa_{i+1}$ si la tarea es completada, si no es así la $capa_i$ invoca continuamente los servicios de la $capa_{i-1}$ de abajo. La interfaz de cada capa encapsula todas las implementaciones de los servicios en detalle de la capa actual y las interfaces de las capas de abajo. Una solicitud de una capa superior a una capa de abajo se hace a través de un método de invocación y la respuesta regresa a través de un método de retorno.

Cada capa tiene dos interfaces: la interfaz superior provee los servicios para la capa de arriba y la interfaz inferior requiere de los servicios de la capa de abajo.

En una jerarquía puramente de capas, cada capa sólo provee servicios a la capa adyacente superior y sólo solicita servicios de la capa adyacente directamente inferior. Algunos casos especiales utilizan un conexión de tipo puente, donde una capa superior obtiene servicios de una capa de niveles más bajos a la capa inmediata inferior, y una conexión de brecha donde una capa inferior obtiene servicios de su capa superior.

La capa superior provee servicios más genéricos u orientados a la aplicación, por lo que es más abstracta; la capa inferior provee servicios específicos del tipo de la utilidad requerida, por lo que es menos abstracta y maneja servicios que muchos de los componentes de las capas superiores necesitan.

La Figura 11 muestra una arquitectura típica de capas para software aplicado al negocio con interacción del usuario. Las líneas sólidas indican la dirección del camino de la solicitud de servicio. Mientras más alta sea la capa, los servicios son más abstractos (en términos de la distancia de la capa física del sistema operativo). En el nivel más alto, los usuarios sólo ven las interfaces de usuario como las GUI, pero no los detalles de implementación.

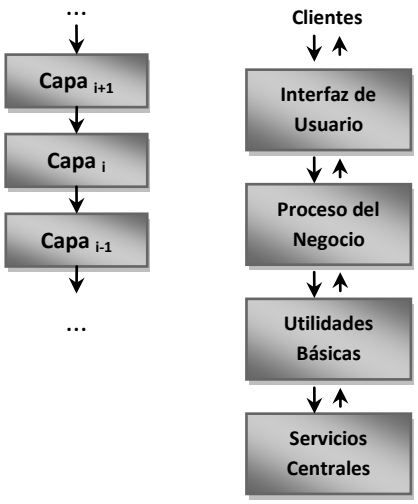

Figura 11. Arquitectura Parcial de Capas [12]

Un sistema de software simple consiste de dos capas: la capa de interacción y la capa de procesamiento:

- La capa de interacción provee las interfaces de usuario a los clientes, toma las solicitudes, las valida y las envía a la capa de procesamiento para su procesamiento y responde a los clientes.
- La capa de procesamiento recibe las solicitudes enviadas y ejecuta los procesos de lógica del negocio, accede a la base de datos, regresa los resultados a su capa superior y deja que

la capa superior responda a los clientes, dado que la capa superior tiene las responsabilidades de las interfaces GUI.

Hay muchos modelos diseñados ampliamente conocidos que usan la arquitectura de capas, como el modelo OSI de 7 capas que cuenta con las capas de aplicación, presentación, sesión, transporte, red, enlace de datos y física. Otro ejemplo de modelos con capas son los servicios web SOAP, XML, HTML, TCP e IP; otros ejemplos son el diseño del sistema operativo UNIX con las capas de shell, la capa de servicios básicos y la capa de drivers de dispositivos.

Dominios aplicables a la arquitectura de capas:

- Cualquier sistema que pueda ser dividido en porciones específicas de la aplicación y en porciones específicas de la plataforma que son las que proveen servicios genéricos a la aplicación del sistema.
- Aplicaciones que tienen divisiones limpias entre servicios básicos, críticos, de interfaces de usuario, etc.
- Aplicaciones que tienen un número de clases que están relacionadas entre ellas de tal manera que puedan ser agrupadas en un paquete para proveer los servicios a otras.

Entre sus beneficios:

- Desarrollo de software incremental basado en niveles incrementales de abstracción.
- Tiene una elevada independencia entre las capas superior e inferior, debido a que no hay impacto al haber cambios en los servicios de la capa inferior siempre que sus interfaces correspondientes se mantengan sin cambios.
- Tiene una elevada flexibilidad: la intercambiabilidad y la reusabilidad son elevadas debido a la separación de la interfaz estándar y su implementación.
- La tecnología basada en componentes es una tecnología adecuada para implementar una arquitectura en capas, ya que hace mucho más fácil al sistema permitir nuevos componentes "*plug-and-play*".
- Promociona la portabilidad, cada capa puede ser una máquina abstracta desplegada independientemente.

Sus limitaciones:

- Bajo desempeño en tiempo de ejecución dado que las solicitudes del cliente o una respuesta al cliente puede pasar potencialmente a través de varias capas. Surgen temas a considerar en este tipo de diseño como la sobrecarga en la distribución y acumulamiento de datos por cada capa.
- Muchas aplicaciones no se ajustan a este diseño arquitectónico.
- El incumplimiento de las comunicaciones entre capas puede causar puntos muertos, y el "puenteo" puede causar un estrecho acoplamiento.
- Otras cuestiones a considerar cuando se elige este estilo son el manejo de excepciones y errores, ya que las fallas en una capa se propagan hacia todas las capas de arriba.

Arquitecturas relacionadas: Máquina virtual, repositorio y cliente-servidor.

2.4.4.4 Máquina Virtual *(Virtual Machine)*

Una máquina virtual es construida sobre un sistema existente y provee una abstracción virtual, un conjunto de atributos y operaciones. En la mayoría de los casos una máquina virtual separa el lenguaje de programación o el ambiente de aplicación de la plataforma de ejecución. Una máquina virtual puede parecer similar a un software de emulación.

El diagrama de la Figura 12 describe al sistema operativo Unix como una máquina virtual, que provee múltiples *shells* como *C shell*, *Korn shell* y *Born shell* arriba del *kernel* de Unix. El *kernel* o núcleo provee todas las capacidades centrales y las bibliotecas de utilidades, lo que hace a todos los *shells* del sistema independientes del los drivers de los dispositivos subyacentes y de los dispositivos físicos actuales.

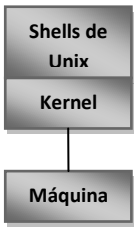

Figura 12. Máquina Virtual de Unix [12]

El CLR (*Common Language Runtime*) de la plataforma .NET de Microsoft también juega el rol de máquina virtual ya que usa un solo lenguaje intermedio para unificar múltiples módulos VB.NET, VC.NET, y C# (Figura 13), de esta manera un cliente VB.NET puede usar un componente en C# o en C++.

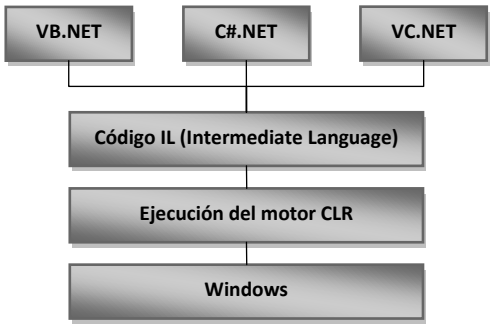

Figura 13. Máquina Virtual CLR en la Plataforma .NET [12]

Otro ejemplo bien conocido de máquina virtual es la Máquina Virtual de Java (JVM). Este es un ambiente en tiempo de ejecución que hace el lenguaje de programación Java una plataforma independiente. En otras palabras, el *bytecode* de Java y otro código interno de Java generado por compilación en el sistema puede correr en cualquier sistema operativo que soporte la JVM. La JVM hace los programas de Java portables, lo que es una de las ventajas más importantes sobre cualquier otro lenguaje de programación ejecutable como C++.

La Figura 14 describe el rol de la JVM y como la JVM separa el *bytecode* del código de máquina del sistema operativo.

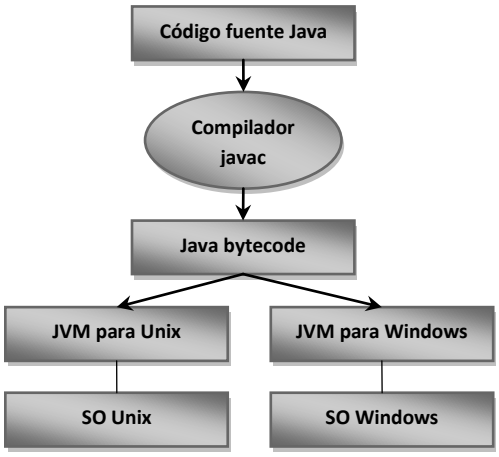

Figura 14. Rol de la Máquina Virtual de Java (JVM) [12]

Dominios aplicables a la arquitectura de máquina virtual:

- Es adecuada para resolver problemas por simulación o traducción si no hay una solución directa.
- Ejemplos de aplicaciones incluyen intérpretes de microprogramación, procesamiento de XML, ejecución de lenguajes de comandos, ejecución de sistemas basados en reglas, interpretes de lenguajes de programación del tipo de Smalltalk y Java.

Algunos beneficios:

- Portabilidad e independencia de la plataforma de las máquinas.
- Simplicidad en el desarrollo de software.

Sus limitaciones:

- Bajo desempeño del intérprete debido a la naturaleza de éste.
- Sobrecarga adicional debido a la nueva capa.

Arquitecturas relacionadas: Intérprete, repositorio y de capas.

2.4.5 Comunicación Implícita Asíncrona

Una comunicación por invocación implícita asíncrona se puede especificar en dos modos diferentes: con búfer y sin búfer. Este estilo está muy relacionado con los patrones publicador-suscriptor (*publisher-suscriber*) o el productor-consumidor (*producer-consumer*) donde los suscriptores/consumidores están interesados en los eventos o mensajes que surgen del publicador/productor. Los suscriptores se registran a sí mismos con las fuentes del evento. El suscriptor es de hecho un receptor que después de su registro es notificado de las ocurrencias. Una vez que un evento es disparado por una fuente de eventos, todos los suscriptores son

notificados y cada uno realiza sus acciones correspondientes. Corresponde a los suscriptores decidir las acciones a ejecutar. El patrón Observador (*Observer*) es otro nombre usado para este tipo de arquitectura.

La cola de mensajes y el tópico de mensajes son típicas arquitecturas asíncronas con búfer que los suscriptores/consumidores también necesitan para registrar sus intereses, el evento/mensaje es disparado cuando está disponible en el búfer de la cola de mensajes o en el tópico de mensajes. Una cola de mensajes es una arquitectura uno a uno o punto a punto entre los expedidores de mensajes y los receptores de los mensajes; mientras que un tópico de mensajes es una arquitectura uno a muchos entre los publicadores y los suscriptores.

A pesar del tipo de arquitectura asíncrona, el propósito principal de este tipo de arquitectura de comunicación es proveer el desacoplamiento entre el evento/mensaje, el publicador/productor y el suscriptor/cliente. Estas son arquitecturas muy comunes en las aplicaciones distribuidas. Ejemplos específicos de arquitecturas con invocación implícita incluyen: los componentes JavaBean, ActiveX, .NET, el mecanismo de retrollamado (*callback*) de CORBA (*Common Object Request Broker Architecture*), los métodos de retrollamado de los componentes EJB (Enterprise JavaBean), los apuntadores de paso de función como parámetros de funciones en la invocación de métodos en C++, Java SAX parser, y los mecanismo remotos de .NET de MS.

2.4.5.1 Invocación Implícita Basada en Eventos sin Búfer

La arquitectura por invocación implícita basada en eventos sin búfer divide el sistema de software en dos particiones: en fuentes de eventos y receptores de eventos. El proceso de registro de eventos conecta estas dos particiones, no hay un búfer disponible entre estas dos particiones.

La invocación implícita basada en eventos es parte de lenguajes como SmallTalk donde cada objeto mantiene su propia lista de dependencias. Cualquier cambio del estado del objeto impacta a sus dependientes. En la Figura 15 se muestra la arquitectura de software por invocación implícita basada en eventos de SmallTalk. Los componentes gráficos "Vista" se registran a sí mismos con la fuente de eventos que les interesa. Cuando los datos cambian en el "Modelo" (la fuente o origen de eventos) el objetivo es notificado a través del "Espacio de Eventos" y el objetivo maneja el evento en consecuencia. Estas son las bases del concepto de invocación implícita basada en eventos mediante el cual el invocador debe esperar la respuesta del módulo llamado; en este caso el invocador no procede hasta que las partes llamadas respondan. El MVC adopta este tipo de conexión entre el modelo y la vista.

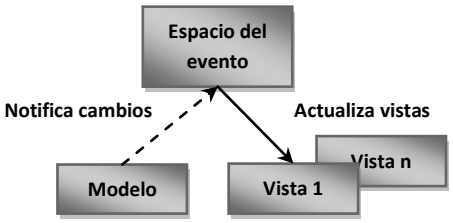

Figura 15. Eventos de Invocación Implícita en SmallTalk [12]

El diagrama de la Figura 16 muestra la arquitectura de las conexiones explícita síncrona y la implícita asíncrona en un diagrama de secuencia de UML. En la parte superior del diagrama se muestra la invocación síncrona, en la parte inferior se muestra la asíncrona. A partir de la flecha de en medio de la figura se indica una secuencia asíncrona, en esta el solicitante de servicios no espera la respuesta, en su lugar se genera un subproceso independiente para recibir la respuesta del proveedor de servicios y reanudar su propia ejecución. En muchos casos el proveedor de servicios gasta una cantidad significante de tiempo de cálculos antes de que pueda responder a los solicitantes, para evitar pérdida de tiempo se utiliza entonces una comunicación asíncrona. En otros casos como en las aplicaciones con interfaces GUI (*Graphical User Interface*), no hay necesidad de ciclos de verificación en las acciones de los usuarios debido a las características no deterministas de las aplicaciones, en estos casos las arquitecturas asíncronas basadas en eventos ayudan a este tipo de aplicaciones, donde los eventos desencadenan y conducen los cálculos.

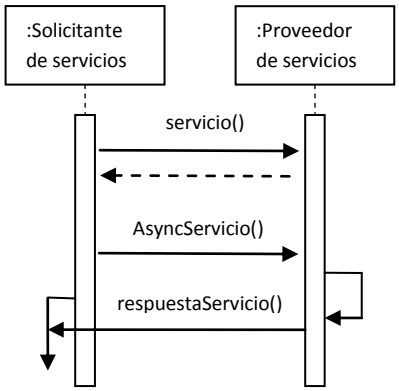

Figura 16. Invocaciones Síncronas vs Asíncronas [12]

Las arquitecturas con invocación implícita basadas en eventos son una buena solución cuando se tiene en la aplicación interacción de los usuarios con las interfaces de usuario, por ello es que se puede encontrar en muchos diseños de aplicaciones de software. Un ejemplo se encuentra en la función "*trigger*" de la aplicación de bases de datos de Oracle Developer. Muchas IDEs para herramientas CASE aplican también las conexiones basadas en eventos, para el manejo de funciones de edición, como cuando sucede al finalizar la edición de un archivo de código fuente se desencadena su compilación, o su recompilación; esto a su vez desencadena el proceso de vinculación, etc. Otro ejemplo relacionado es el manejo del proceso de depuración: al parar en un punto de ruptura desencadena que el editor se haga cargo para que el "*tester*" pueda checar el estado actual del programa.

Un ejemplo dentro de las aplicaciones de negocios es un sistema de administración de almacén mediante el cual un usuario puede establecer los límites superiores e inferiores de los niveles de un elemento específico. Cuando el valor de los niveles es muy alto o muy bajo, el usuario será notificado automáticamente por el sistema y se realizará la acción que el usuario configuró por adelantado. En un sistema de e-comercio, cuando el nivel del inventario de un producto disminuye bajo un valor mínimo establecido, el sistema notifica a los proveedores para ordenar más productos con el fin de mantener ese producto en la reserva del almacén.

Dominios aplicables para la arquitectura basada en eventos sin búfer son:

- Herramientas con componentes interactivos GUI y en ambientes de desarrollo integrados (IDE).
- Aplicaciones que requieran bajo acoplamiento entre componentes que necesiten notificar o desencadenar otros componentes para tomar acción sobre notificaciones asíncronas.
- En la implementación de máquinas de estado.
- Cuando el manejo de eventos en las aplicaciones no sea predecible.

Algunos de sus beneficios:

- Disponibilidad del marco de trabajo, muchos vendedores de APIs como el AWT de Java y los componentes Swing están disponibles.
- Reusabilidad de componentes, es fácil de conectar los componentes en nuevos manejadores de eventos sin afectar al resto del sistema.
- Mantenimiento y evolución del sistema, tanto las fuentes de los eventos como los objetivos son fáciles de actualizar.
- Independencia y flexibilidad en la conectividad, se puede realizar el registro y dar de baja componentes dinámicamente en tiempo de ejecución.
- Es posible la ejecución en paralelo de manejo de eventos.

Dentro de sus limitaciones:

- Es difícil realizar pruebas y depuración en los sistemas debido a que es difícil predecir y verificar respuestas y el orden de las respuestas de los receptores. El desencadenamiento de eventos no puede determinar cuándo una respuesta ha terminado o la secuencia de todas las respuestas.
- Hay un estrecho acoplamiento entre las fuentes de eventos y sus receptores que en la invocación implícita basados en cola de mensajes y en tópicos de mensajes. El compartir y pasar datos en los objetos de eventos desde las fuentes de datos a los receptores de eventos también hacen un acoplamiento estrecho y difícil la depuración y las pruebas.
- Dentro de los temas a considerar al elegir esta arquitectura son la confiabilidad y la sobrecarga de las invocaciones implícitas.

Arquitecturas relacionadas: PAC, basadas en mensajes, multi-niveles y arquitecturas de máquinas de estados.

2.4.5.2 Invocación Implícita Basada en Mensajes con Búfer

La arquitectura de software con invocación implícita basada en mensajes con búfer divide al sistema de software en tres particiones: productores de mensajes, consumidores de mensajes y proveedores de mensajes de servicio. Están conectados asíncronamente por una cola de mensajes o por un tópico de mensajes. Esta arquitectura es considerada también como centrada en datos. En un sistema basado en mensajes, también conocido como un sistema de disparo y olvido, un remitente envía un mensaje que requiere únicamente de un mensaje de respuesta de entrega

garantizada. Es típicamente implementado como un middleware orientado a mensajes (MOM) que provee de un servicio de mensajes confiables en un sistema distribuido.

Las arquitecturas basadas en mensajes han sido usadas por mucho tiempo. Los sistemas de mensajes son usados para construir aplicaciones distribuidas confiables, escalables y flexibles que soporten comunicación asíncrona. La arquitectura de un sistema de mensajes es esencialmente una arquitectura cliente- servidor *peer-to-peer*. El alto grado de independencia de los componentes dentro de los sistemas de mensajes es una de sus principales características. Su alta escalabilidad, su interoperabilidad en redes heterogéneas y su confiabilidad hacen a los sistemas de mensajes muy populares.

Las arquitecturas basadas en mensajes son ampliamente usadas en la administración de infraestructura de redes, servicios de telecomunicaciones, e-comercio, atención al cliente, pronóstico del tiempo, administración de la cadena de servicio, sistemas bancarios, entre otros sistemas. También son usadas como puentes para la fusión de los sistemas en la integración de las empresas.

Muchas plataformas proveen sus propios mecanismos de colas de mensajes, incluyendo el MQ de Unix y Microsoft, mientras que otros como MQseries de IBM, Progress SonicMQ, JBossMQ y FioranoMQ implementan directamente a Java Message Server (JMS). JMS es una API típica en la plataforma J2EE que soporta mensajería asíncrona.

Un mensaje está formado por datos estructurados con un identificador, una cabecera, propiedades y cuerpo, un ejemplo típico de mensaje es un documento XML.

La mensajería es un mecanismo o tecnologías que maneja la entrega de mensajes asincrónicamente y sincrónicamente de manera efectiva y confiable. Un cliente de mensajería puede producir y enviar mensajes a otros clientes y puede consumir mensajes de otros clientes. Cada cliente se debe registrar con un destinatario a través de una sesión de conexión provista por un proveedor de mensajes de servicio para crear, enviar, recibir, leer, validar y procesar los mensajes.

La mensajería soporta un acoplamiento pobre entre componentes de software en comunicaciones distribuidas esto es similar a las comunicaciones implícitas basadas en eventos. De cualquier manera, un receptor de menajes no necesita estar disponible al mismo tiempo que el remitente para tener comunicación. De hecho, el receptor y el remitente no necesitan conocer entre ellos sus identidades. El receptor, sin embargo, necesita conocer el formato del mensaje y el destino del mensaje donde el mensaje está disponible.

Muchos sistemas de mensajes soportan comunicación asíncrona para entregar mensajes a los clientes conforme van llegando, así un consumidor no necesita solicitar mensajes en orden para recibirlos.

Este tipo de sistemas también soportan una transmisión fiable de mensajes que garantiza que los mensajes sean entregados exactamente una vez.

Este tipo de sistemas de mensajería son similares al correo electrónico excepto que los productores y los receptores de un correo electrónico son humanos en lugar de componentes de software.

Mensajería Punto a Punto *(Point-to-Point Messaging, P2P)*. La arquitectura de la cola de mensajes es una estructura punto a punto entre productores y consumidores. Una arquitectura de mensajes P2P está compuesta por la cola de mensajes, remitentes y destinatarios. Cada mensaje es enviado a un destino (una cola específica) la cual es mantenida por el consumidor, los clientes consumidores extraen mensajes de estas colas. La cola conserva todos los mensajes que recibe hasta que el mensaje es consumido o el mensaje expira. Cada mensaje tiene un sólo consumidor, esto quiere decir que un mensaje "se va" una vez que se entrega. Este enfoque permite múltiples receptores de mensajes pero sólo uno de ellos recibe un mensaje de acuerdo a como lo determine el proveedor de mensajes de servicio. Los remitentes y destinatarios de mensajes no tienen dependencias de tiempo, el destinatario puede recibir un mensaje aún cuando no esté disponible al momento que el remitente lo envié. La mensajería P2P requiere que cada mensaje enviado a la cola de mensajes sea consumido exitosamente. Es mucho más confiable que los sistemas basados en eventos.

Un ejemplo es el Message Driven Bean (MDB) de EJB (ver Figura 17) un consumidor de la cola de mensajes soportado por JMS.

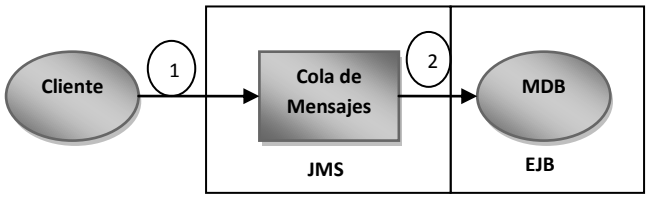

Figura 17. Cola de Mensajes de JMS [12]

Mensajería Publicar-Suscribir *(Publish-Suscribe Messaging P&S)*. La arquitectura de la mensajería P&S es parecida a un *hub* donde los clientes publicadores envían mensajes a un tópico de mensajes que actúa como un tablero de boletines. Los publicadores y suscriptores del tópico de mensajes no son conscientes uno del otro. Una diferencia entre P&S y P2P es que cada tópico de mensajes puede tener múltiples consumidores, el sistema entrega los mensajes a sus múltiples suscriptores en vez de un solo destinatario, como en la cola de mensajes del sistema.

Los publicadores y suscriptores tienen una dependencia de tiempo, un consumidor del tópico de mensajes se debe suscribir al tópico antes de que se publique, a menos que sea una "suscripción durable" que es capaz de recibir cualquier tópico de mensajes enviado mientras los suscriptores no estén activos o listos.

De manera similar a las arquitecturas basadas en eventos, las arquitecturas de las conexiones basadas en mensajes son ampliamente implementadas por comunicaciones asíncronas (Pueden ser síncronas). Un componente de software puede registrar un receptor de mensajes con un consumidor (Un receptor de mensajes es similar a un receptor de eventos).

La Figura 18 describe el tópico de mensajes de JMS, muestra un tópico publicador suscrito por múltiples suscriptores.

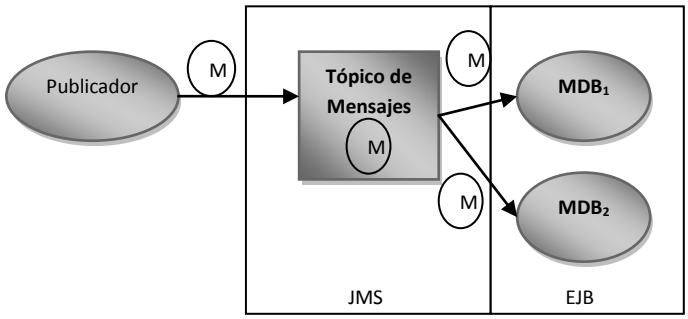

Figura 18. Tópico de Mensajes de JMS [12]

Una aplicación web empresarial en línea *bussiness-to-bussiness* (B2B) puede usar una arquitectura basada en mensajes en situaciones como las siguientes:

- El componente inventario manda un mensaje al componente proveedor cuando el nivel del inventario de un producto cae debajo de un cierto nivel para ser repuesto.
- El componente inventario también manda un mensaje a la oficina del negocio para preparar una cantidad de dinero para adquirir más productos para el inventario y actualiza los presupuestos.
- El componente proveedor manda un mensaje de notificación al componente catálogo para actualizar la información después de que el componente proveedor obtiene más productos.

Algunas aplicaciones de mensajería pueden ser en las áreas de servicios financieros, de salud, académicos, entre otros.

Dominios aplicables para la arquitectura basada en eventos sin búfer son:

- Adecuada para sistemas de software donde las comunicaciones entre productor y receptor requiere de invocación implícita asíncrona basada en mensajes para propósitos de desempeño y distribución.
- Cuando el proveedor requiere de componentes que funcionen independientemente de la información acerca de otras interfaces de componentes, así que los componentes son más fáciles de reemplazar.
- El proveedor requiere de una aplicación que corra cuando todos o no todos los componentes corran simultáneamente.
- Cuando el modelo de la aplicación del negocio permite a un componente mandar información y continuar operando por sí mismo sin esperar una respuesta inmediata.

Sus beneficios:

- Anonimato, provee de un alto grado de anonimato entre los productores y consumidores de mensajes, ya que el consumidor de mensajes no conoce: quien produce el mensaje (independencia de usuario), donde vive el productor en la red (independencia en locación), o cuando fue producido el mensaje (independencia de tiempo).
- Soporta concurrencia tanto entre consumidores como entre productores y consumidores.
- Escalabilidad y confiabilidad en la entrega de mensajes, dentro de los mecanismos de confiabilidad incluye ajustes de control de nivel de mensajes de reconocimiento, ajustes en la persistencia de los mensajes sin pérdida, ajustes del nivel de prioridad de mensajes y ajustes en la expiración de mensajes.
- Soporta procesamiento en lotes.
- Soporta un bajo acoplamiento ente productores y consumidores de mensajes, y entre sistemas heredados y modernos para el desarrollo de la integración.

Dentro de sus limitaciones:

- Límite de la capacidad de la cola de mensajes, no es una limitación inherente pero si es un tema a considerar al elegir esta arquitectura que puede ser minimizado si la cola es implementada por una estructura de datos dinámica (listas ligadas por ejemplo). De cualquier manera hay un límite absoluto basado en la disponibilidad de la memoria.
- Tiene una completa separación de la presentación y la abstracción debida al control de cada componente lo que genera complejidad en el desarrollo, dado que la comunicación entre componentes sólo toma lugar entre el control de estos.
- Aumento de la complejidad del diseño e implementación del sistema.

Arquitecturas relacionadas: Sistemas basados en eventos, capas y MVC.

2.4.6 Orientadas a Interacción

Las arquitecturas de software orientadas a interacción descomponen a un sistema en tres grandes particiones: módulo de datos, módulo de control y el módulo de presentación de vista, cada módulo tiene sus propias responsabilidades. El módulo de datos provee la abstracción de los datos y toda la lógica principal del negocio en el procesamiento de los datos. El módulo de presentación de vista es responsable de la presentación visual o de audio de los datos de salida y también puede proveer las interfaces de usuario de entrada cuando sea necesario. El módulo de control determina el flujo del control envolviendo la selección de las vistas, la comunicación entre módulos, la repartición de tareas, la inicialización de de ciertos datos y algunas acciones de configuración del sistema. El punto principal de esta arquitectura es la separación de las interacciones del usuario de la abstracción de los datos y de el procesamiento de los datos del negocio. Puede haber muchas presentaciones de vistas en diferentes formatos, múltiples vistas pueden ser soportadas por el mismo conjunto de datos. Aún para presentaciones de vistas específicas, las interfaces o vistas pueden necesitar cambiar frecuentemente, así que un bajo acoplamiento entre las abstracciones de los datos y su presentación es útil, y además es soportada

por este estilo. Un bajo acoplamiento entre conexiones se puede implementar en una variedad de formas, puede ser a través del método de invocación implícita o por el método de invocación implícita de registro/notificación. El módulo de control juega un rol central que media el módulo de datos y el módulo de presentación de vistas, todos estos módulos deben estar totalmente conectados.

A continuación se presentan el estilo de las dos principales categorías de arquitecturas orientadas a interacción: *Model-View-Controller* (MVC) y *Presentation-Abstraction-Control* (PAC).

Estos dos modelos son muy similares en el sentido que proponen una descomposición en tres componentes. El módulo de presentación del PAC es similar al módulo de Vista del MVC; el módulo de abstracción del PAC se parece al módulo del modelo o datos del MVC; el módulo de control del PAC es parecido al módulo controlador del MVC. Tanto MVC como PAC son usados en aplicaciones interactivas como en aplicaciones web online y aplicaciones distribuidas con múltiples tareas e interfaces de usuario. Difieren en el flujo del control y organización, el PAC es una arquitectura jerárquica basada en agentes, mientras que el MVC no tiene una estructura jerárquica clara y los tres módulos son conectados entre ellos.

Una diferencia importante entre el MVC y el PAC es que los datos en el MVC son activos mientras que los datos en el PAC son pasivos. Los datos activos del MVC notifican a los otros dos módulos de cualquier cambio en ellos. Los cambios en los datos pasivos del PAC están bajo control total del componente de control.

Aunque ambas arquitecturas son populares en aplicaciones interactivas con GUIs, PAC es una mejor opción para aquellos sistemas donde los subsistemas que requieran de sus propias interfaces interactivas personalizadas. La arquitectura del MVC es ampliamente usada en el diseño de aplicaciones web, mientras que la arquitectura PAC es ampliamente usada en sistemas basados en agentes donde cada agente tiene su tarea específica e interfaz. PAC es una buena opción para aplicaciones de comunicaciones móviles inalámbricas donde cada dispositivo necesita tener sus propios datos e interfaces interactivas, además de comunicación con otros dispositivos.

2.4.6.1 Modelo-Vista-Controlador (MVC)

Muchos desarrolladores web están familiarizados con la arquitectura MVC (Modelo-Vista-Controlador) porque es ampliamente adoptada para el diseño de aplicaciones interactivas web como las tiendas en línea, encuestas, registro de estudiantes, entre otros sistemas de servicios interactivos. La arquitectura MVC es especialmente usada en aplicaciones donde las interfaces son propensas a cambios de datos, también soporta características "*look and feel*" en sistemas GUI. Los componentes *Java Swing* y los controladores de distribución Java Swing son diseñados usando una arquitectura MVC.

Esta arquitectura fue introducida por primera vez en Smalltalk-80. De acuerdo a Glenn Krasner y Stephen Pope [29]:

La programación del Modelo-Vista-Controlador consiste en la aplicación de estas tres formas de fabricación mediante el cual los objetos de diferentes clases toman el control de las operaciones relacionadas al dominio de la aplicación (el Modelo), la visualización del estado de la aplicación (la Vista) y de la interacción del usuario con el modelo y la vista (el Controlador).

Modelos: El modelo de una aplicación es la simulación o implementación de software específico para el dominio de la estructura central de la aplicación.

Vistas: En este caso las vistas tratan con todo lo relacionado gráficamente y de las solicitudes de datos de su modelo y de su visualización.

Controladores: Contienen la interfaz entre sus modelos asociados, vistas y dispositivos de entrada (teclados, dispositivos apuntadores, etc.).

En resumen, el Controlador administra las solicitudes de entrada del usuario, controla las secuencias de las interacciones del usuario, selecciona las vistas deseadas para las pantallas de salida, y administra la inicialización, instanciación y registro de los otros módulos en el sistema MVC. El módulo del Modelo provee todos los servicios funcionales básicos y encapsula todos los detalles de los datos. El módulo del Modelo no depende de otros módulos, y no conoce cuales vistas están registradas o ligadas a él. El módulo de Vista es responsable de mostrar los datos proveídos por el módulo del modelo y actualizar las interfaces cuando se notifique que los datos han cambiado.

MVC-I

El MVC-I es una versión simple de la arquitectura MVC donde el sistema es simplemente descompuesto en dos subsistemas: El Controlador-Vista y el Modelo. Básicamente, el Controlador-Vista se encarga del procesamiento de las entradas y las salidas y sus interfaces; el módulo del Modelo hace frente a las funcionalidades básicas y los datos. El módulo del Controlador-Vista se registra con el módulo de los datos. El módulo del Modelo notifica al módulo del Controlador-Vista de cualquier cambio en los datos así que cualquier representación gráfica de datos se cambiará de acuerdo a los nuevos datos, el controlador también toma las acciones apropiadas en los cambios.

La conexión entre el Controlador-Vista y el Modelo pueden ser diseñados en un patrón suscribir-notificar (*subscribe-notify*) mediante el cual el Controlador-Vista se suscribe al Modelo, y el Modelo notifica al Controlador-Vista de cualquier cambio. En otras palabras, el Controlador-Vista es un observador de los datos en el módulo Modelo. En la Figura 19 el Controlador y la Vista son combinados juntos para actuar como una interfaz de usuario de entrada/salida y el Modelo provee todos los datos y servicios del dominio.

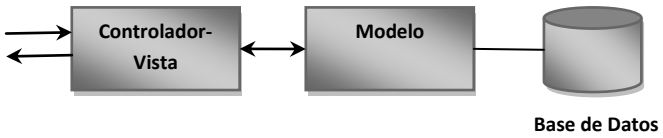

Figura 19. Arquitectura de MVC-I [12]

MVC-II

La arquitectura MVC-II es un desarrollo mejorado de la arquitectura MVC-I. El módulo de Modelo provee todas las funcionalidades básicas y el soporte para datos para una base de datos. El módulo de Vista muestra los datos mientras que el módulo del Controlador toma las solicitudes de entrada, valida los datos de entrada, inicia el Modelo y la Vista y su conexión, y reparte tareas. El Controlador y la Vista se registran con el módulo del modelo. Cuando los datos en el módulo del Modelo sufren cambios, el módulo de Vista y el módulo del Controlador son notificados. En otras palabras, el módulo del Modelo juega un rol activo en la arquitectura MVC-II comparado con la arquitectura del MVC-I. En la arquitectura del MVC-II, el módulo de la Vista y el Controlador están separados. Esto permite la división del trabajo, por ejemplo, los programadores expertos pueden trabajar en el Controlador, mientras que los expertos en el diseño de interfaces gráficas pueden trabajar en el desarrollo de la Vista. También, debido a que las tecnologías de interfaces gráficas son actualizadas rápidamente y los requerimientos del negocio son cambiados muy frecuentemente, es mucho mejor mantener la Vista separada del Controlador. El MVC en la Figura 20 muestra el Controlador y la Vista separados.

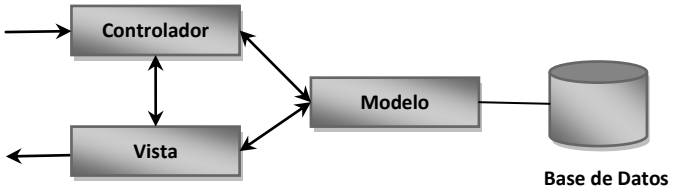

Figura 20. Arquitectura de MVC-II [12]

La Figura 21 muestra un diagrama de secuencias para una típica arquitectura MVC. Después de que los clientes comienzan la aplicación MVC, el Controlador inicializa el Modelo y la Vista, y se adhiere a sí mismo y la Vista al Modelo (esto se denomina un registro al Modelo). Posteriormente, el Controlador intercepta una solicitud del usuario ya sea directamente por línea de comando o por la interfaz de la Vista, y envía la solicitud al Modelo para actualizar los datos en el Modelo. Los cambios en el Modelo provocan que el Modelo notifique a todos los oyentes adheridos a él sobre los cambios, y las interfaces en la Vista son actualizadas inmediatamente.

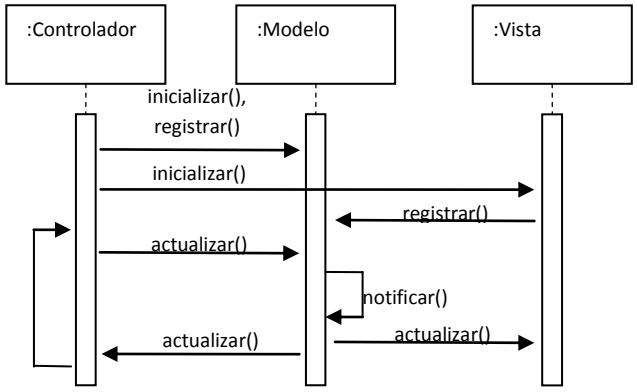

Figura 21. Diagrama de Secuencias para la Arquitectura MVC [12]

Dominios aplicables a la arquitectura MVC:

- Aplicaciones interactivas donde múltiples vistas necesitan de un sólo modelo de datos y las interfaces son propensas a cambios de datos frecuentes.
- Aplicaciones con divisiones claras entre los módulos de controlador, vista y datos, así que pueden ser asignados profesionales diferentes para trabajar en diferentes aspectos de la aplicación.
- Un *framework* (marco de trabajo) basado en el patrón arquitectónico MVC bien conocido es "Apache Struts" el cual es ampliamente usado en el diseño de aplicaciones web.

Algunos de sus beneficios:

- Están disponibles muchos *frameworks* que contienen herramientas para el MVC.
- Permite tener múltiples vistas sincronizadas con el mismo modelo de datos.
- Es fácil de cambiar nuevas vistas de interfaces, permitiendo actualizar las vistas de las interfaces con nuevas tecnologías sin revisar el resto del sistema.
- Muy efectivo para desarrolladores si los profesionales de desarrollo de gráficos, programadores y de base de datos están trabajando en equipo en un proyecto diseñado.

Dentro de sus limitaciones:

- No es adecuada para aplicaciones orientadas a agentes como las aplicaciones interactivas para móviles y las robóticas.
- Al tener múltiples pares de controladores y vistas basados en un mismo modelo de datos hacen que cualquier modelo de datos sea muy caro.
- En algunos casos la división entre la Vista y el Controlador no es clara.

Arquitecturas relacionadas: PAC, invocación implícita tal como la basada en eventos y la arquitectura multiniveles.

2.4.7 Distribuidas

Un sistema distribuido es un conjunto de dispositivos computacionales y de almacenamiento conectados a través de una comunicación de red. En este tipo de sistemas los datos, el software y los usuarios están distribuidos. Los subsistemas o componentes dentro de un sistema distribuido se comunican con otros usando un número de métodos que incluye el paso de mensajes, llamadas a procedimientos remotos, invocación de métodos remotos, entre otros. Dos de los elementos más importantes al diseñar sistemas distribuidos son: la topología de la red, es decir la forma en la que las entidades están organizadas para formar una red conectada; y el modo de comunicaciones, el método por el que los componentes se comunican con otros.

Muchos sistemas en el mundo real son naturalmente distribuidos, estos sistemas son ampliamente usados en ambientes empresariales grandes como los sistemas de bases de datos que habilitan a los datos para ser remotamente accedidos.

Un sistema distribuido puede ser modelado por una arquitectura cliente servidor, siendo éste la base para las arquitecturas multiniveles. Algunas alternativas son las arquitecturas "*broker*" como CORBA y las arquitecturas orientadas a servicios (SOA) como los servicios web y los servicios de red. Las características principales de las arquitecturas distribuidas son su transparencia en la ubicación de los servicios, fiabilidad y disponibilidad. Adicionalmente, existen muchos marcos de trabajo tecnológicos que soportan las arquitecturas distribuidas como: .NET, J2EE, CORBA, servicios web .NET, servicios web AXIS Java, y los servicios de red GloBus.

2.4.7.1 Cliente-Servidor

El modelo cliente-servidor es la arquitectura de sistemas distribuidos más común. Se basa en dos procesos de comunicación, usualmente corren en procesadores diferentes, y por lo tanto se descompone en dos subsistemas principales: el cliente y el servidor. El primer proceso, el cliente, emite una solicitud al segundo proceso, el servidor. El proceso servidor recibe la solicitud (servir datos de una base de datos, impresión de un documento, etc.), lo lleva a cabo, y manda una respuesta al cliente.

La Figura 22 muestra un ejemplo para la arquitectura cliente-servidor de dos niveles. El nivel de la interfaz o *front-end* se preocupa por la interacción del usuario, y el nivel *back-end* se preocupa en la lógica del negocio y la administración de la base de datos.

La separación del cliente del servidor de datos libera a los clientes de realizar la administración de los datos, como en el caso de desarrollos de SQL, de manera que apoye desarrollos paralelos de diferentes niveles respectivamente.

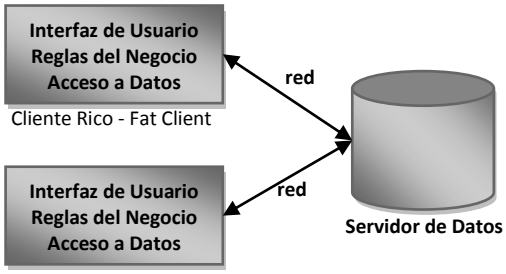

Figura 22. Arquitectura Cliente-Servidor de Dos Niveles [12]

Sus ventajas:

- Separación de responsabilidades, como la presentación de la interfaz de usuario y el procesamiento de la lógica del negocio.
- Reusabilidad de los componentes de servidor.

Sus desventajas:

- Falta de infraestructura heterogénea para tratar con los cambios en los requerimientos.
- Complicaciones de seguridad.
- Baja disponibilidad y confiabilidad de los servidores.
- Baja facilidad de pruebas y escalabilidad.
- Los clientes ricos (*fat-clients*) tienen juntos la presentación y la lógica del negocio.

2.4.7.2 MultiNiveles

En una arquitectura multiniveles (supongamos una arquitectura de tres niveles) el nivel de *front-end* se encarga de la presentación de la interfaz de usuario. El nivel intermedio administra la lógica del negocio y ejecución. El nivel de *back-end* generalmente maneja la administración de la base de datos. Las arquitecturas multiniveles han estado ganando popularidad en la actualidad en aplicaciones empresariales. La Figura 23 es un ejemplo de arquitectura de tres niveles.

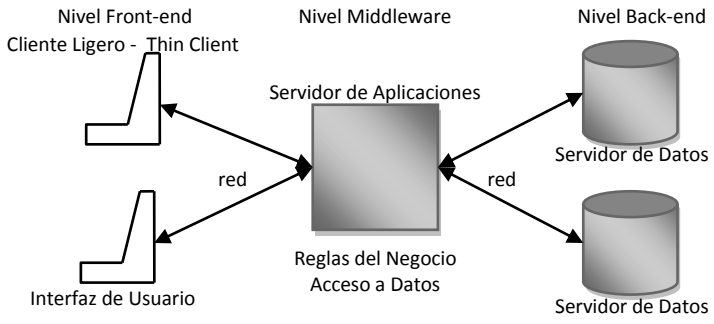

Figura 23. Arquitectura de Tres Niveles [12]

La ventaja de una arquitectura de tres niveles sobre una de dos es el aumento de la reusabilidad y la escalabilidad al agregar un nivel intermedio. Por ejemplo, el nivel intermedio en una arquitectura de tres niveles puede tener un diseño e implementación portable y no propietaria,

también puede proveer soporte multihilos para escalabilidad. Las arquitecturas multiniveles reducen el tráfico en la red.

Su principal desventaja es que no es fácil realizar pruebas debido a la falta de herramientas adecuadas para este tipo de arquitectura. Al agregar múltiples servidores en un sistema hacen que la confiabilidad y la disponibilidad sean más críticos.

2.4.7.3 Broker

La arquitectura *Broker* es una arquitectura middleware usada en cómputo distribuido para coordinar y facilitar la comunicación entre servicios y clientes registrados.

Buschmann desarrolla el patrón arquitectónico *Broker* para diseñar un sistema distribuido en componentes que interactúan por invocaciones a servicios remotos:

> *El patrón arquitectónico Broker puede ser usado para estructurar sistemas de software distribuidos con componentes desacoplados que interactúan por invocaciones a servicios remotos. Un componente Broker es responsable de coordinar las comunicaciones, como del reenvío de solicitudes, así como la transmisión de resultados y excepciones* [8].

Un *broker* puede ser tanto un servicio orientado a invocación donde los clientes mandan peticiones de invocación, como un *broker* orientado a mensajes o documentos (como un documento XML) donde los clientes mandan un mensaje o documento.

Uno de los atributos más importantes de este tipo de arquitectura es que tiene un mejor desacoplamiento entre clientes y servidores gracias al uso del *broker*. En otras palabras, la comunicación nunca se hace directamente entre el cliente y el servidor. Un sistema *broker* también es llamado un sistema basado en *proxy*.

Los servidores hacen disponibles sus servicios a los clientes registrando y publicando sus interfaces con el *broker*. Los clientes pueden solicitar servicios a los servidores a través del *broker* estática o dinámicamente. Un componente *broker* es responsable de mantener la conexión entre clientes y servidores y de coordinar las comunicaciones intermediando las solicitudes/peticiones de servicios, localizando el servidor apropiado, reenviando y despachando solicitudes, y enviando respuestas o excepciones de regreso a los clientes. Un *broker* actúa como un policía en una intersección transitada que interactúa con los componentes clientes y servidores.

Un ejemplo de implementación de la arquitectura *Broker* es CORBA (*Common Object Request Broker Architecture*).

Con el patrón *Broker*, un cliente distribuido puede acceder a servicios distribuidos simplemente llamando al método remoto del objeto remoto de la misma manera que si fuera una llamada a un método local. Este concepto es similar al de *Remote Procedure Call* (RPC) usado en la programación distribuida estructurada de Unix y al *Remote Method Invocation* (RMI) en programación orientada a objetos distribuida de Java. Los clientes distribuidos sólo necesitan obtener una referencia al objeto apropiado, en lugar de escribir código detallado para una

comunicación orientada a protocolos. Además, los clientes pueden invocar dinámicamente los métodos remotos aun si las interfaces de objetos remotos no están disponibles en tiempo de compilación.

El cliente tiene una conexión directa a su "*proxy* de cliente" y el servidor tiene una conexión directa a su "*proxy* de servidor". El *proxy* habla con el *broker* mediador. El *proxy* es un patrón bien conocido para esconder detalles de bajo nivel del procesamiento de comunicaciones como la serialización/deserialización de datos, procesamiento de puertos de entrada/salida, y soportar transparencia en la localización. Un *proxy* de cliente reside en la dirección del espacio del cliente e implementa la interfaz de servicio en dicho espacio. El *proxy* del cliente juega un rol de mediador o interceptor, que intercepta las solicitudes del cliente, obtiene todos los argumentos, los empaqueta, serializa y da formato al paquete de acuerdo al formato del protocolo de comunicación, y luego lo envía al *broker*. Por la misma razón, existe un *proxy* del lado del servidor para liberar al servidor de conocer la ubicación del cliente y de los detalles del protocolo de comunicación.

A continuación se explican los subcomponentes de la arquitectura de *broker*:

- ***Broker*:** Coordina las comunicaciones pasando las solicitudes y regresando las respuestas. El *broker* almacena toda la información de registros del servidor, incluyendo la funcionalidad y servicios, así como información de la ubicación. El *broker* provee: de APIs a los clientes para solicitudes y a los servidores para responder, registro/desregistro de componentes del servidor, transferencia de mensajes y ubicación de los servidores.
- ***Stub* (*proxy* del lado del cliente):** Media entre el cliente y el *broker*, y provee de transparencia adicional entre ellos. Para el cliente un objeto remoto parece igual que uno local. El proxy oculta los interprocesos de comunicación a nivel de protocolo, serializa los valores de los parámetros, y deserializa los resultados del servidor. El *stub* es generado en tiempo estático de compilación y desplegado en el lado del cliente para ser usado como *proxy* para el cliente.
- ***Skeleton* (*proxy* del lado del servidor):** También es generado estáticamente por la compilación de la interfaz de servicio y luego desplegado en el lado del servidor. Encapsula las funciones de red de bajo nivel específicas del sistema de manera parecida a la *proxy* del cliente. Recibe y desempaca las solicitudes, deserializa los argumentos del método, y llama al servicio apropiado. Cuando recibe el resultado de regreso del servidor también serializa el resultado antes de enviarlo de regreso al cliente.
- ***Bridges* (Puentes):** Son componentes adicionales usados para ocultar detalles de implementación cuando interactúan dos *brokers*. Bridges encapsula los detalles de implementación subyacentes a la red y media *brokers* diferentes como en los *brokers* de Java CORBA y en .NET Remote. Pueden traducir solicitudes y parámetros de un formato a otro. Un puente puede conectar dos diferentes redes basadas en diferentes protocolos de comunicación.

- **Red:** Conecta los componentes usando los protocolos de comunicación designados como TCP/IP, OIIP y SOAP. La solicitud lleva datos en un documento de mensajes o formato del método de invocación.

El diagrama de la Figura 24 muestra los objetos envueltos en un sistema *broker*. Un *broker* obtiene las solicitudes de los clientes y administra esas solicitudes reenviándolas a los proveedores de servicios directamente o despachando las solicitudes a otro *broker* conectado. Una vez que se regresan los resultados al *broker*, este los manda de regreso a los clientes. Muchos *brokers* pueden trabajar juntos en un sistema complejo como el de la Figura 25.

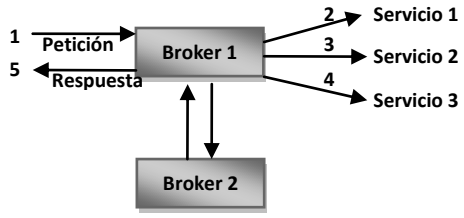

Figura 24. Modelo de Broker [12]

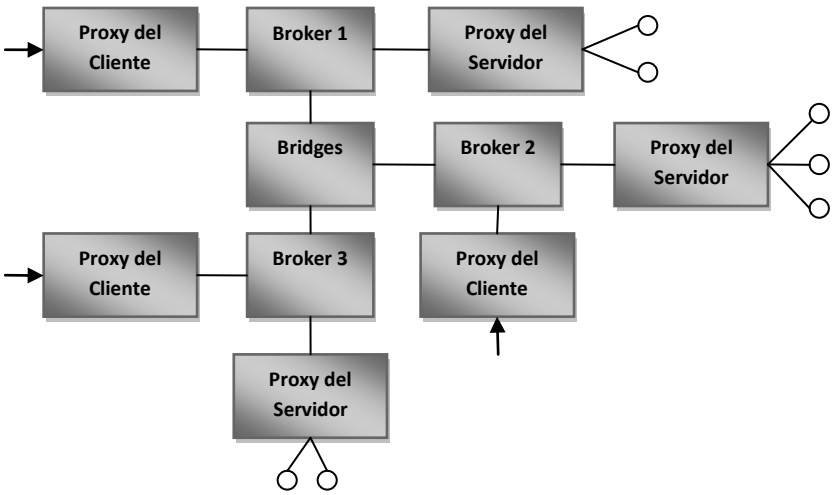

Figura 25. Brokers conectados con el proxy de los clientes y servidores [12]

En general el patrón *proxy* juega un rol importante en la arquitectura *Broker*. Un *proxy* es como un sustituto para el componente de un servidor que provee una interfaz local al objeto remoto real.

Ventajas de esta arquitectura:

- Transparencia en la implementación del componente del servidor y en su ubicación.
- Cambiabilidad y extensibilidad.
- Simplicidad para los clientes para acceder al servidor y la portabilidad del servicio.
- Interoperabilidad a través de los puentes del *broker*.
- Reusabilidad.

- Confiabilidad de cambios en tiempo de ejecución de los componentes del servidor (agregar o remover componentes del servidor al vuelo).

Desventajas:

- Ineficiencia debido a sobrecargas en las *proxys*.
- Baja tolerancia a fallas.
- Dificultad para hacer pruebas debido a la cantidad de *proxys*.

2.4.7.4 Orientadas a servicios (SOA)

Una arquitectura orientada a servicios (SOA) se inicia con un proceso de negocios, en este contexto, un servicio es una funcionalidad del negocio que está bien definida, es autosuficiente e independiente de otros servicios, y publicada y disponible para ser usada a través de una interfaz estándar de programación. El software administra los procesos del negocio a través de una arquitectura orientada a servicios con interfaces estándares bien definidas que pueden construir, mejorar y expandir la infraestructura existente de una manera flexible. Los servicios SOA pueden ser usados extensivamente dentro de un dominio o línea de producto, e inclusive entre sistemas heredados. Un bajo acoplamiento en la orientación de servicios provee una gran flexibilidad para que las empresas hagan uso de todos los recursos de servicios disponibles sin importar la plataforma y las restricciones tecnológicas.

La conexión entre servicios es conducida por protocolos orientados a mensajes comunes y universales, como el protocolo SOAP (*Simple Object Access Protocol*) de servicios web, que puede entregar peticiones y respuestas entre los servicios libremente. Las conexiones pueden ser establecidas estática y dinámicamente.

La Figura 26 ilustra cómo trabaja SOA, un cliente puede encontrar un servicio a través de un directorio de servicios y luego accederlos en un modo de servicio de petición-respuesta.

Figura 26. Cliente con Servicios y Directorio de Servicios [12]

Una típica aplicación orientada a servicios hace uso de muchos servicios disponibles usando un lenguaje de control de flujo (por ejemplo, BPEL para servicios web). Estos lenguajes de orquestación permiten especificar la secuencia y orden lógico de las ejecuciones del negocio basadas en la lógica del negocio. La Figura 27 presenta una aplicación de servicios web para adquisiciones del negocio, servicios como: orden de adquisiciones, procesamiento de crédito, control de inventario, transportación y manejo, y su propio procesamiento lógico del negocio. En términos de programación se puede hacer este tipo de aplicaciones mediante la incorporación de servicios web para reducir el costo y el tiempo de desarrollo significativamente. Las

implementaciones internas de estos servicios web atómicos (por ejemplo, el procesamiento de crédito y el control de inventario) pueden ser cambiadas con el tiempo siempre que se mantengan apegados a sus interfaces públicas. Otra característica positiva es la reusabilidad de todos los servicios web atómicos. Algunos servicios pueden ser reusados por otras aplicaciones para los cuales no estaban diseñados originalmente. Por ejemplo, el servicio de procesamiento de crédito puede ser vendido a otras empresas si desean adquirir el soporte para pagos en línea.

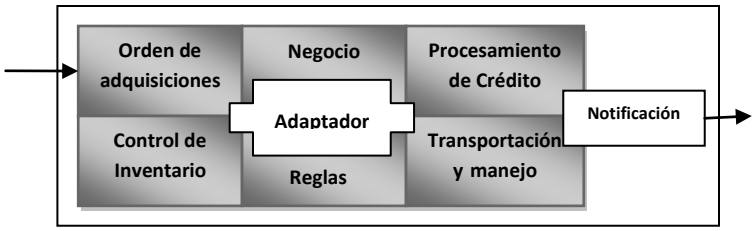

Figura 27. Ejemplo de Composición de Servicios [12]

Una aplicación compleja orientada a servicios puede estar constituida por muchos servicios de tal manera que unos sean responsables de recibir las peticiones, otro se encarguen de responder, y otros pueden no estar necesariamente conectados con los usuarios externos (Ver en la Figura 28 los servicios 2 y 3).

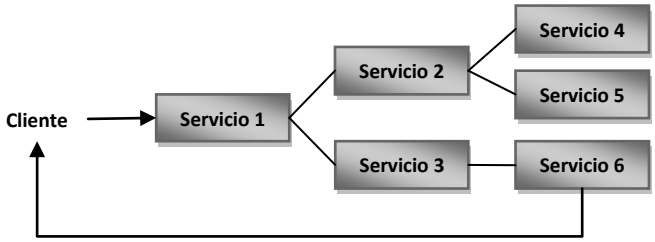

Figura 28. Reúso de Servicios [12]

Los servicios generales consisten de múltiples servicios que trabajan juntos de una manera coordinada. A través de la agregación y composición, los servicios pueden ser construidos recursivamente para satisfacer necesidades del negocio más complejas. Un servicio nuevo puede ser construido de servicios existentes usando los conceptos de estructuración orientada a objetos de agregación y composición (Ver Figura 29). Una agregación solo extiende un servicio para hacer una nueva interfaz para el nuevo servicio. En la contención se tiene una interfaz que envuelve todos los servicios usados. De esta forma SOA es una forma de organizar y compartir ampliamente funciones del negocio.

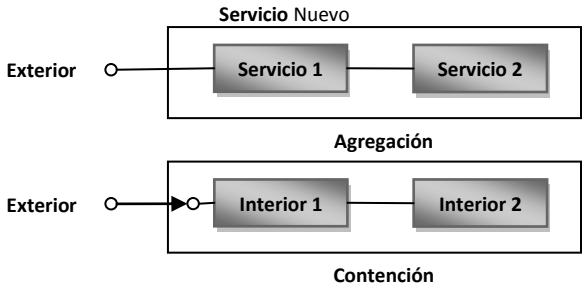

Figura 29. Modelo de Composición de Servicios [12]

Sus ventajas:

- Conexiones con bajo acoplamiento: Esta característica es clave en las arquitecturas orientadas a servicios. Cada componente del servicio es independiente debido a la característica de servicios sin estados. La implementación de un servicio no afecta a la aplicación siempre que la interfaz expuesta no cambie. Esto hace a este tipo de arquitecturas mucho más fácil de evolucionar y actualizar.
- Interoperabilidad: Técnicamente, cualquier cliente o servicio puede acceder otros servicios independientemente de la plataforma, tecnología, vendedores, o implementaciones de lenguajes.
- Reusabilidad: Cualquier servicio puede ser reusado por cualquier otro servicio. Dado que los clientes de un servicio únicamente necesitan conocer sus interfaces públicas, la composición e integración se hace mucho más fácil. Esto hace al desarrollo de las aplicaciones de negocio basadas en SOA mucho más eficientes en términos de tiempo y costo.
- Escalabilidad: En general los servicios con bajo acoplamiento son fáciles de escalar. Las características de generalidad, orientación a documentación y asincronía de los servicios mejoran el atributo de escalabilidad.

2.4.8 Basadas en Componentes

La arquitecturas de software basadas en componentes dividen un problema en sub-problemas cada uno asociado con particiones de componentes. Las interfaces de los componentes juegan un rol importante en el diseño basado en componentes. La principal motivación detrás del diseño basado en componentes es la reusabilidad o reutilización: un componente encapsula la funcionalidad y conductas de un elemento de software en una unidad binaria reutilizable y auto desplegable. Los diseños pueden hacer uso de *componentes reutilizables comerciales off-the-shell* (COTS) existentes o desarrollarlos totalmente para producir componentes reutilizables para un uso futuro. Esto incrementa la confiabilidad del sistema dado que la confiabilidad de cada componente individual refuerza la confiabilidad de todo el sistema a través de la reutilización.

La arquitectura basada en componentes hace énfasis a la descomposición del software en componentes funcionales. Esta descomposición permite convertir componentes pre-existentes en piezas más grandes de software.

Este proceso de construcción de una pieza de software con componentes ya existentes, da origen al principio de reutilización del software, mediante el cual se promueve que los componentes sean implementados de una forma que permita su utilización funcional sobre diferentes sistemas en el futuro.

Hay muchos *frameworks* de componentes estándar como COM/DCOM, JavaBean, EJB, CORBA, .NET, servicios web y servicios de red. Estas tecnologías de componentes son ampliamente usadas en diseños de aplicaciones de escritorio locales GUI como en los componentes gráficos de JavaBean, los componentes de MS ActiveX y los componentes COM, los cuales pueden ser reusados a través de "*drag and drop*". Muchos componentes son invisibles, especialmente aquellos distribuidos en aplicaciones empresariales y en aplicaciones web como los componentes en EJB (*Enterprise JavaBean*), .NET y CORBA. La combinación de las tecnologías con orientación a servicios y componentes están siendo muy usadas actualmente.

Un diseño orientado a componentes representa una abstracción de alto nivel y es equivalente al diseño orientado a objetos, el primero define componentes y conexiones entre ellos en lugar de clases y conexiones de clases entre ellos. Un componente es un concepto de alto nivel, que usualmente incorpora más de una clase. Así que en el diseño orientado a componentes primero se identifican los componentes y sus interfaces en lugar de identificar clases y sus relaciones.

Las arquitecturas de software orientadas a componentes tienen muchas ventajas sobre su contraparte orientada a objetos, incluyen: un tiempo reducido para el mercado, bajos costos en el desarrollo debido a la reutilización de componentes existentes, y un incremento en la confiabilidad.

Un componente de software se define típicamente como algo que puede ser utilizado como una caja negra, en donde se tiene de manera externa una especificación general, la cual es independiente de la especificación interna.[4]

Un componente es modular (cohesivo), desplegable (portable), reemplazable (*plug-and-play*) y reutilizable, que contiene un conjunto de funcionalidades bien definidas que encapsulan su implementación y son exportadas como una interfaz de alto nivel. Un componente es un paquete de software desplegable que provee servicios a sus clientes, que también puede requerir servicios de otros componentes. Un componente es auto suficiente y sustituible siempre que su interfaz se mantenga intacta.

Se presentan tres conceptos ligados con la definición de un componente:

- Interior del componente: es una pieza de software que cumple con un conjunto de propiedades y que se encuentra conformada como un artefacto del cual se espera que sea reutilizable.

[4] http://www.users.globalnet.co.uk/~rxv/CBDmain/cbdfaq.htm#Component-Based%20Development

- Exterior del componente: es una interfaz que cumple con un conjunto de propiedades y provee un servicio a los agentes humanos u otros artefactos de software.
- Relación interior-exterior: es la que define el proceso de relación entre el interior y exterior el componente, a través de conceptos como especificación, implementación y encapsulación.

De forma evidente se puede determinar que, el principal elemento de software dentro de un arquitectura basada en componentes son precisamente los componentes de software.

Existen algunos principios definidos por Clemens Szyperski [30], que definen a un componente de software como elemento de la arquitectura:

- Múltiple uso: se refiere al hecho de que un componente es escrito dentro de un contexto que permita que su funcionalidad sea útil en la creación de distintas piezas de software.
- Contexto no específico: en relación con la orientación conceptual de la especificación de un componente, debe estar planteada de una forma general que permita su adaptación en distintos sistemas, sin que el contexto tenga prioridad.
- Encapsulación: se refiere a la especificación interna oculta o no investigable a través de la interfaz. Así se protege que el resto de componentes y piezas de software dentro de un sistema, no se vean afectados por cambios en el diseño de uno de los componentes.
- Una unidad independiente de desarrollo con su propio control de versiones: este principio muy relacionado con la encapsulación, permite que un componente pueda ser desarrollado de manera independiente, cambiando el diseño o agregando nuevas funcionalidades, sin afectar significativamente el resto del sistema.

La estructura de la arquitectura basada en componentes contempla 3 partes:

1. El nombre de los componentes: El nombre de un componente debe ser la identificación de la funcionalidad y uso que tiene como software. Generalmente, los desarrolladores usan algún tipo de convención que facilite la identificación de componentes, especialmente, cuando se trabaja en proyectos de gran envergadura.
2. La interfaz de los componentes: Es el área de intercambio (input-output) entre el interior y el exterior de un componente de software. La interfaz es quien permite acceder a los datos y funcionalidades que estén especificadas en el interior del componente (acceder funcionalmente, no a su especificación). Adicional a la interfaz se encuentra la documentación que muestra la información sobre cómo utilizar un componente.
3. Cuerpo y código de implementación: Es la parte del componente que provee la forma (implementación) sobre la cual un fragmento del componente realiza sus servicios y funcionalidades. Este es el área que debe cumplir con el principio de encapsulación.

Dominios aplicables para una arquitectura basada en componentes:

- En aplicaciones donde los contratos entre los subsistemas son claros.

- En aplicaciones que requieran un bajo acoplamiento entre componentes y donde muchos componentes reutilizables estén disponibles.
- Adecuada para sistemas con una organización basada en bibliotecas de clases. La biblioteca de clases de .NET y la API de Java son construidas en una arquitectura de componentes.

Sus beneficios:

- Reusabilidad de componentes.
- En el mantenimiento y evolución de los sistemas ya que es fácil de cambiar y actualizar la implementación sin afectar el resto del sistema.
- Ofrece independencia y flexibilidad en la conectividad de los componentes.
- Permite el desarrollo independiente de componentes por grupos diferentes en paralelo.
- Tiene una buena productividad en el desarrollo de software actual y futuro.
- Muchas herramientas de diseño orientadas a objetos pueden ser usadas en el desarrollo de software basado en componentes.

Entre sus limitaciones:

- Puede ser difícil encontrar componentes disponibles adecuados para reutilizar.
- Un tema a considerar en el diseño es la adaptación de los componentes.
- Pocas herramientas de diseño orientadas a componentes están disponibles.

Arquitecturas relacionadas: Orientadas a objetos, distribuidas y SOA.

2.4.9 Heterogéneas

En la práctica, múltiples estilos arquitectónicos son usados en el mismo proyecto, en otras palabras los sistemas reales son una combinación de los denominados estilos puros.

Existen diferentes formas en la que los estilos arquitectónicos pueden ser combinados [3]:

- Una primera opción puede ser una combinación jerárquica, donde un componente del sistema usa un tipo de estilo en su estructura interna, diferente a la definida en el sistema.
- Una segunda opción sucede cuando un componente usa una combinación de conectores arquitectónicos.
- Una tercera opción es describir algún nivel de abstracción con otro estilo diferente. Es decir, usar un cierto estilo en uno o más componentes.

Las arquitecturas heterogéneas buscan combinar los beneficios de múltiples estilos y asegurar la calidad y pertinencia de todo un sistema.

2.5 Comparación de los Atributos de Calidad por Estilo Arquitectónico

Kai Qian y colegas en [12] hacen un análisis de los estilos arquitectónicos descritos en las secciones 2.4.1 a 2.4.9, como resultado proponen la siguiente tabla:

	Economía en Tiempo	Economía en Espacio	Completitud	Seguridad	Interoperabilidad	Independencia de Hardware	Independencia de Software	Facilidad de Instalación	Reusabilidad	Tolerancia a Errores	Disponibilidad	Facilidad de Entendimiento	Interfaces de Usuario	Facilidad de Aprendizaje
OO	+	+	+				+		+				+	-
Secuencial en lotes					-				-					+
Tuberías y filtros		-			-				+					+
Control de procesos	+					+								-
Repositorios	+	+							+			-		-
Pizarra	-	+							+					-
Principal-Subrutina	+	+		-	-				-			-		-
Maestro-Esclavo	+								-					-
Capas	-			+		+			+	+	+	+		-
Máquina Virtual	--	-		+		++	++	+	+	+	+			-
Basada en eventos sin búfer					+							+		+
Basada en mensajes con búfer		-			+							+		+
MVC					+									+
PAC					+									+
Cliente-Servidor														+
Multi-niveles	-	-		+	+									+
Broker	-													-
SOA	-	-			++	++	++	+	++		+	+	+	++
Basada en Componentes					++		+		++		+			++

Tabla 7. Tabla Informativa de Estilos Arquitectónicos [12]

Cada renglón denota si un estilo arquitectónico responde al atributo de calidad en cada columna, de la siguiente manera: "+" significa bueno, "++" significa muy bueno, "-" significa malo, "--" significa muy malo. Una celda vacía denota que no hay un juicio explícito para ese par estilo-atributo. Al examinar cada estilo arquitectónico contra los atributos de calidad y considerando el dominio de la aplicación para cada estilo arquitectónico, se puede obtener una idea de la aplicabilidad de un estilo arquitectónico en ese proyecto o de las arquitecturas de software que se pueden usar en conjunto para al final tener una Arquitectura.

2.6 Documentación de Arquitecturas de Software

La arquitectura es una herramienta de comunicación que permite explicar a los interesados (*stakeholders*) las decisiones de diseño y las consecuencias de estas decisiones. En particular, la arquitectura es la guía para la implementación y tiene que ser comunicada efectivamente a los desarrolladores.

La documentación de la arquitectura es creada para facilitar la comunicación entre interesados y suele planear las iteraciones, se vuelve especialmente importante cuando hay muchas personas envueltas en el desarrollo del sistema de software, cuando la persona que implementa el sistema no es la que creó la arquitectura, o cuando los equipos de desarrollo están distribuidos geográficamente.

Otro motivo importante para realizar la documentación es para realizar el mantenimiento del producto, generalmente al realizarse se involucra gente nueva, los desarrolladores originales se encuentran en otros proyectos o dejaron la empresa de desarrollo, etc. Al realizar la documentación se facilita enormemente el realizar el mantenimiento.

Para realizar la documentación existen varias propuestas como son las Vistas Arquitectónicas y los Lenguajes de Descripción Arquitectónica que pueden ser utilizadas individualmente o en conjunto dependiendo del nivel de detalle que se requiera en el sistema en desarrollo. Algunas de las herramientas que se pueden utilizar se mencionan a continuación.

Un enfoque común es producir la documentación usando herramientas CASE de edición para texto, de modelado y dibujo de diagramas. Desafortunadamente, esta documentación típicamente termina en los estantes juntando polvo después de ser usada en un periodo inicial para proporcionar algunas ideas acerca de la arquitectura. Algunas de las razones por lo que esto sucede son:

1. Es muy difícil mantener la documentación de la arquitectura actualizada en un mundo donde los requerimientos son muy cambiantes, especialmente en los sistemas grandes. El(los) arquitecto(s) pueden no tener toda la información acerca de los cambios y nuevos requerimientos. Si los *stakeholders* (por ejemplo, los desarrolladores) encuentran un sólo dato desactualizado en la documentación, entonces esta se considerará poco fiable.
2. Las herramientas disponibles deben ayudar a los arquitectos a diseñar y evolucionar una arquitectura, pero estas herramientas no son ampliamente aceptadas o usadas por aquellos *stakeholders* que no sean los desarrolladores. Aún para los desarrolladores, estas herramientas podrían resultar complicadas de usar porque no están integradas en el ambiente de implementación.
3. El proceso de cambiar la arquitectura (como en los tableros de control de cambios) pueden ser muy lentas para los desarrolladores. Una vez que la implementación está en su lugar, es muy fácil cambiar el código, cambiar la arquitectura se considera una pérdida de tiempo. Muy frecuentemente el código cambia debido a la corrección de errores o mejoras, estas refactorizaciones no son reflejadas en la documentación de la arquitectura.
4. La documentación no es efectiva porque:
    - Los escritores hacen muchos supuestos acerca de lo que los lectores saben.
    - Los diagramas no son entendidos debido a una notación ambigua.
    - El texto es demasiado detallado y repetitivo.
    - Los documentos están mal estructurados y es difícil de navegar a través de ellos.

Una solución ante el panorama descrito anteriormente es propuesta por Bachmann y Merson, proponen el uso de herramientas de documentación basadas en la Web, particularmente de ***Wikis*** para realizar la documentación de la arquitectura [31].

En [32] se propone la utilización de herramientas multimedia para realizar la documentación, parte del hecho de que generalmente los desarrolladores en vez de leer muchas hojas de

documentación prefieren preguntarle al experto en proyectos, ya que es una forma visual y rápida de comunicación. Para esto propone que una manera ágil para realizar la documentación es a través de herramientas multimedia (voz y video), en los que el equipo expone las decisiones que va tomando durante el desarrollo del software. Menciona que pueden usarse herramientas de bajo costo como presentaciones electrónicas o herramientas de alto costo como cámaras de video y grabadoras de voz, y sean distribuidas a través de *videocast o podcast*. Recomienda que el contenido dure entre 10 a 15 minutos y se centre en la información que realmente se quiere compartir (como en el caso de las prácticas de Scrum). Sin embargo esta forma de documentación no es la adecuada para proyectos con contenidos muy complejos, extremadamente frecuentes o en los casos que se requiera documentación muy formal.

Otra forma de documentación es a través del uso de pizarrones blancos, cada que se termine de modelar se saca una fotografía de este y se distribuye al equipo, ya sea por Wikis, email, etc.

En caso de realizar la documentación a través de editores de texto se recomienda seguir un estándar de organización, por ejemplo usando una plantilla, esto facilita al lector la navegación a través de una estructura familiar. Una plantilla también permite al escritor guardar información tan rápido como vaya siendo conocida y medir el trabajo que queda por hacer [33].

2.6.1 Vistas Arquitectónicas

De acuerdo al nivel de responsabilidad dentro del desarrollo de un sistema y la relación que se establezca con el mismo, son muchas las partes involucradas e interesadas en la arquitectura de software [14], estas son:

- El *analista del sistema*, quien la utiliza para organizar y expresar claramente los requerimientos y entender las restricciones de tecnología y los riesgos.
- *Usuarios finales y clientes*, que necesitan conocer el sistema que están adquiriendo.
- El *administrador de proyecto*, que la utiliza para organizar el equipo y planificar el desarrollo.
- Los *diseñadores*, que lo utilizan para entender los principios subyacentes y localizar los límites de su propio diseño.
- Otras *organizaciones desarrolladoras* (si el sistema es abierto), que la utilizan para entender cómo interactuar con el sistema.
- Las *compañías subcontratadas*, que la utilizan para entender los límites de su sección de desarrollo.
- Los *arquitectos*, quienes velan por la evolución del sistema y la reutilización de componentes.

Todas estas personas deben comunicarse de manera efectiva para discutir y razonar acerca de la arquitectura, y así alcanzar las metas del desarrollo. En virtud de esto, Kruchten plantea que debe tenerse una representación del sistema que todos puedan comprender [14].

Una única representación de la arquitectura del sistema resultaría demasiado compleja y poco útil para todos los involucrados, pues contendría mucha información irrelevante para la mayoría de

estos. Es por ello que se plantea la necesidad de representaciones que contengan únicamente elementos que resultan de importancia para cierto grupo de involucrados.

Buschmann [8] establece que una *vista arquitectónica* representa un aspecto parcial de una arquitectura de software, que muestra propiedades específicas del sistema. Bass [9], haciendo uso indistinto de los términos estructura y vista, propone que las *estructuras arquitectónicas* pueden definirse agrupando los componentes y conectores de acuerdo a la funcionalidad del sistema, sincronización y comunicación de procesos, distribución física, propiedades estáticas, propiedades dinámicas y propiedades de ejecución, entre otras.

Por su parte, Kruchten define una *vista arquitectónica* como una descripción simplificada o abstracción de un sistema desde una perspectiva específica, que cubre intereses particulares y omite entidades no relevantes a esta perspectiva [14]. Para la definición de una vista, Kruchten propone la identificación de ciertos elementos, que se mencionan a continuación:

- Punto de vista: involucrados e intereses de los mismos.
- Elementos que serán capturados y representados en la vista y las relaciones entre estos.
- Principios para organizar la vista.
- Forma en que se relacionan los elementos de una vista con otras vistas.
- Proceso a ser utilizado para la creación de la vista.

Kruchten [14], Bass [9], Clements, Kazman y Klein [15] y Hofmeister [34], proponen, en función de las características del sistema o del proceso de desarrollo del mismo, distintas vistas arquitectónicas. Es importante resaltar que las vistas propuestas no son independientes entre sí, puesto que son perspectivas distintas de un mismo sistema [14]. Por tal motivo, las vistas arquitectónicas deben estar coordinadas, de tal manera que al realizar cambios, estos se vean correctamente reflejados en las vistas afectadas, garantizando consistencia entre las mismas.

Ante la diversidad de planteamientos sobre las distintas perspectivas de un mismo sistema, resulta interesante establecer comparaciones entre los mismos, puesto que, en algunos casos, hacen referencia a un mismo tipo de perspectiva bajo nombres de vistas distintos, o por el contrario, bajo el mismo nombre expresan perspectivas diferentes. De igual forma, hay vistas que contemplan varias perspectivas, así como también varias vistas pueden crear una única perspectiva [15].

2.6.1.1 Comparación de Vistas Arquitectónicas

De acuerdo al análisis realizado en [35], la Tabla 8 presenta las similitudes observadas entre las vistas propuestas por Kruchten [14], Bass [9], Clements [15] y Hofmeister [34], en función de las perspectivas que éstas ofrecen. Para cada perspectiva se presenta el nombre de la vista planteado por cada uno de los autores analizados. Adicionalmente, se hace referencia a los interesados en cada una de las vistas y las propiedades no funcionales de la arquitectura que maneja cada perspectiva.

Perspectiva	Kruchten [14]	Bass [9]	Clements [15]	Hofmeister [34]	Interesados (Stakeholders)	Atributo de Calidad
Abstracción de requerimientos funcionales del sistema	Vista Lógica	Vista Conceptual o Lógica	Vista Funcional	Vista Conceptual	Cliente Usuario Final Analista	Modificabilidad Reusabilidad Dependencia Seguridad Externa
Creación de procesos e hilos de ejecución, comunicación entre ellos y recursos compartidos	Vista de Proceso	Vista de Procesos o Coordinación + Vista de Llamadas	Vista de Concurrencia	Vista de Ejecución	Arquitectos Desarrolladores Equipo de Pruebas Mantenimiento	Desempeño Disponibilidad
Organización de los elementos arquitectónicos implementados	Vista de Implantación	Vista Física + Vista de Módulos	Vista de Desarrollo	Vista de Código	Programadores Mantenimiento Gerentes de Configuración Gerentes de Desarrollo	Mantenibilidad Modificabilidad Capacidad de Prueba
Distribución de procesos en la plataforma	Vista de Desarrollo	Vista de Flujo de Control	Vista Física + Vista de Concurrencia	Vista de Módulos y Vista de Ejecución	Arquitectos Desarrolladores Equipo de Pruebas Mantenimiento Ing. Hardware	Desempeño Escalabilidad Disponibilidad Seguridad Interna
Escenarios y Casos de Uso	Vista de Casos de Uso	Vista de Usos	-	Vista Conceptual	Cliente Usuario Final Analista	Reusabilidad Disponibilidad Modificabilidad
Especificación abstracta de clases. objetos, funciones y procedimientos	-	Vista de Clases + Vista de Flujo de Datos	Vista de Código	-	Diseñadores Desarrolladores	Modificabilidad Portabilidad Mantenibilidad

Tabla 8. Comparación de vistas arquitectónicas en función de las perspectivas del sistema. Nota: El signo "+" indica combinación de vistas

2.6.2 Lenguajes de Descripción Arquitectónica

El problema inherente en la mayoría de los desarrollos de software es la naturaleza abstracta de un programa de computación. A diferencia de otros productos de distintas áreas de la ingeniería (autos, casas, aviones, etc.), el software no es tangible, no posee una forma natural de visualización y no hay solución perfecta al problema planteado. Actualmente, la forma más exacta de descripción del sistema es el código fuente o el código compilado. De aquí que el problema de la descripción de una arquitectura de software es encontrar una técnica que cumpla con los propósitos del desarrollo de software; en otras palabras, la comunicación entre las partes interesadas, la evaluación y la implementación [36].

Hasta la fecha, las arquitecturas de software han sido representadas por esquemas simples de cajas y líneas [9], [36] en los que la naturaleza de los componentes, sus propiedades, la semántica de las conexiones y el comportamiento del sistema como un todo, se define de manera muy pobre. En este sentido, Bass [9] proponen que aunque este tipo de representación ofrece una imagen intuitiva de la construcción del sistema, existen muchas preguntas que no pueden responderse con este tipo de representación. Entre ellas se encuentran, por ejemplo, la esencia de los componentes, cuál es su tipo, qué hacen, cómo se comportan, de qué otros componentes dependen y de qué manera, así como qué significan las conexiones y qué mecanismos de comunicación están involucrados, entre otras. Los *Lenguajes de Descripción Arquitectónica (Architecture Description Languages, ADL)* surgen como posible solución ante inquietudes de este estilo [9], [37].

2.6.2.1 Conceptos y características de los lenguajes de descripción arquitectónica

Bass [9] propone que el establecimiento de una notación estándar para la representación de arquitecturas, a través de un lenguaje de descripción arquitectónica, permite mejorar la comunicación entre el autor y el lector, logrando tener un medio de entendimiento común, ahorrando tiempo en indagar el significado de los gráficos que representan la arquitectura y sus componentes. El lenguaje de descripción arquitectónica sirve de soporte para el análisis y las decisiones tempranas de diseño, y sería factible la construcción de herramientas que asistan en el proceso de desarrollo. Además, este tipo de lenguaje provee un mecanismo para la construcción de la arquitectura como artefacto, transferible a otros sistemas, de manera tal que pueda ser tomada como marco de referencia o como punto de partida para el resto de las tareas del proceso de desarrollo.

Los *ADL* resultan de un enfoque lingüístico para el problema de la representación formal de una arquitectura [9], y por ende, son usados para describir una arquitectura de software [37]. Este tipo de lenguaje puede ser descriptivo formal o semi-formal, un lenguaje gráfico, o incluir ambos [36] y sus características vienen dadas por los requerimientos que implica.

En la literatura [9], [38] y [39] es posible encontrar múltiples conjuntos de requerimientos que intentan definir qué debe hacer un lenguaje de este tipo.

Shaw [38] propone que un lenguaje de descripción arquitectónica debe poseer las siguientes propiedades:

- Capacidad para representar componentes (primitivos o compuestos), así como sus propiedades, interfaces e implementaciones.
- Capacidad de representar conectores, así como protocolos, propiedades e implementaciones.
- Abstracción y encapsulamiento.
- Tipos y chequeo de tipos.
- Capacidad de integración de herramientas de análisis.

Por su parte, Luckham propone en [39] los siguientes requerimientos para un lenguaje de descripción arquitectónica:

- Integridad comunicacional, es decir, limitación de la comunicación a los componentes conectados entre sí a nivel arquitectónico
- Capacidad de modelar arquitecturas dinámicas
- Capacidad para razonar acerca de causalidad y tiempo
- Soporte para refinamiento jerárquico
- Capacidad de correspondencia de ciertos comportamientos a arquitecturas posiblemente diferentes

En este mismo sentido, Bass agrega que un lenguaje de descripción arquitectónica debe proveer:

- Soporte a las tareas de creación, refinamiento y validación de una arquitectura.
- Debe englobar reglas acerca de lo que constituye una arquitectura completa o consistente.
- Debe proveer la capacidad de representar la mayoría de los *patrones arquitectónicos* conocidos directa o indirectamente [38].
- Debe tener la capacidad de proveer vistas del sistema que expresen información arquitectónica, pero al mismo tiempo suprimir la implementación de información no arquitectónica.
- Si el lenguaje puede expresar información que se encuentra a nivel de implementación, debe entonces poseer capacidad para realizar correspondencias con más de una implementación a nivel arquitectónico. En otras palabras, debe soportar la especificación de familias de implementaciones que satisfacen una arquitectura común.
- Debe soportar capacidad analítica basada en información de nivel arquitectónico, o bien la capacidad para la rápida generación de prototipos de implementación.

2.6.2.2 Ventajas del uso de lenguajes de descripción arquitectónica

Bass [9] presentan una serie de ventajas que proporciona el uso de los lenguajes de descripción arquitectónica en el desarrollo de un sistema de software. En principio, proponen que la descripción inicial del sistema puede ser llevada a cabo de forma textual o gráfica, basada en estilos arquitecturales y tipos de componentes, así como también hacer la descripción de un sistema o subsistema en función de la información que recibe o produce. De igual forma, es posible hacer la descripción del comportamiento y sus elementos asociados, tales como el tipo de eventos que producen, o a los que responden, incluyendo descripciones o documentación de alto nivel.

Otra ventaja que presentan los ADL es la facilidad con la que puede introducirse y mantenerse la información referente al sistema. En este sentido, no sólo es posible efectuar análisis a distintos niveles de detalle, sino que también es posible establecer cambios de tipos sobre los componentes. Así mismo, es posible realizar análisis de desempeño, disponibilidad o seguridad, en tanto el lenguaje de descripción arquitectónica provea la facilidad para ello.

Por último, Bass indica que los componentes pueden ser refinados en la medida que sea necesario, para distintos tipos de análisis. En cualquier momento un componente puede ser visto conjuntamente con cualquier información que se conozca de él. De igual manera, a partir de las descripciones asociadas a los componentes, se establece la posibilidad de que los mismos puedan ser llevados a nivel de código, o plantillas de código.

Ahora bien, aunque es posible establecer la definición de un lenguaje de descripción arquitectónica y cuáles son las ventajas de su uso, no hay formas claras de diferenciarlos de otros lenguajes. Sin embargo, Bass propone que puede hacerse una distinción en función de cuánta información de tipo arquitectónico es posible representar mediante el lenguaje de descripción arquitectónica. Es necesario aclarar la diferencia que existen entre los ADL y los lenguajes de programación, lenguajes de requerimientos y lenguajes de modelado.

2.7 Evaluación de Arquitecturas de Software

El primer paso para la evaluación de una arquitectura es conocer qué es lo que se quiere evaluar [15]. De esta forma, es posible establecer la base para la evaluación, puesto que la intención es saber qué se puede evaluar y qué no. Resulta interesante el estudio de la evaluación de una arquitectura: si las decisiones que se toman sobre la misma determinan los atributos de calidad del sistema, es entonces posible evaluar las decisiones de tipo arquitectónico con respecto al impacto sobre estos atributos.

La arquitectura de software posee un gran impacto sobre la calidad de un sistema, por lo que es muy importante estar en capacidad de tomar decisiones acertadas sobre ella, en diversos tipos de situaciones, entre las cuales destacan [36]:

- Comparación de alternativas similares.
- Comparación de la arquitectura original y la modificada.

- Comparación de la arquitectura de software con respecto a los requerimientos del sistema.
- Comparación de una arquitectura de software con una propuesta teórica.
- Valoración de una arquitectura en base a escalas específicas.

En este sentido, Clements, Kazman y Klein proponen que, mediante la arquitectura de software, es posible también determinar la estructura del proyecto de desarrollo del sistema, sobre elementos como el control de configuración, calendarios, control de recursos, metas de desempeño, estructura del equipo de desarrollo y otras actividades que se realizan con la arquitectura del sistema como apoyo principal. En este sentido, la garantía de una arquitectura correcta cumple un papel fundamental en el éxito general del proceso de desarrollo, además del cumplimiento de los atributos de calidad del sistema [15].

De esta manera, el interés se centra en determinar el momento propicio para efectuar la evaluación de una arquitectura. En este sentido, en [15] se amplía el panorama clásico, indicando las ocasiones en que se hace posible hacer la evaluación de una arquitectura. Su planteamiento establece que es posible realizarla en cualquier momento, pero propone dos variantes que agrupan dos etapas distintas: la *temprana y* la *tardía*.

Para la primera variante, Clements y sus colegas en [15] establecen que no es necesario que la arquitectura esté completamente especificada para efectuar la evaluación, y esto abarca desde las fases tempranas de diseño y a lo largo del desarrollo. En este punto, resulta interesante resaltar el planteamiento de Bosch en [40], quien propone que es posible efectuar decisiones sobre la arquitectura a cualquier nivel, puesto que se pueden imponer distintos tipos de cambios arquitectónicos, producto de una evaluación en función de los atributos de calidad esperados. Ahora bien, es necesario destacar que tanto Clements y sus colegas en [15] como Bosch en [40] establecen que mientras mayor es el nivel de especificación, mejores son los resultados que produce la evaluación.

La segunda variante propuesta por Clements y sus colegas en [15] consiste en realizar la evaluación de la arquitectura cuando ésta se encuentra establecida y la implementación se ha completado. Este es el caso general que se presenta al momento de la adquisición de un sistema ya desarrollado. Los autores consideran muy útil la evaluación del sistema en este punto, puesto que puede observarse el cumplimiento de los atributos de calidad asociados al sistema, y cómo será su comportamiento general.

Por su parte, Bosch en [40] afirma que la evaluación de una arquitectura de software es una tarea no trivial, puesto que se pretende medir propiedades del sistema en base a especificaciones abstractas, como por ejemplo los diseños arquitectónicos. Por ello, la intención es más bien la evaluación del potencial de la arquitectura diseñada para alcanzar los atributos de calidad requeridos. Las mediciones que se realizan sobre una arquitectura de software pueden tener distintos objetivos, dependiendo de la situación en la que se encuentre el arquitecto y la

aplicabilidad de las técnicas que emplea. Los objetivos que menciona Bosch son tres: *cualitativos*, *cuantitativos* y *máximos y mínimos teóricos*.

La medición *cualitativa* se aplica para la comparación entre arquitecturas candidatas y tiene relación con la intención de saber la opción que se adapta mejor a cierto atributo de calidad. Este tipo de medición brinda respuestas afirmativas o negativas, sin mayor nivel de detalle. La medición *cuantitativa* busca la obtención de valores que permitan tomar decisiones en cuanto a los atributos de calidad de una arquitectura de software. El esquema general es la comparación con márgenes establecidos, como lo es el caso de los requerimientos de desempeño, para establecer el grado de cumplimiento de una arquitectura candidata, o tomar decisiones sobre ella. Este enfoque permite establecer comparaciones, pero se ve limitado en tanto no se conozcan los valores teóricos máximos y mínimos de las mediciones con las que se realiza la comparación. Por último, la *medición de máximo y mínimo teórico* contempla los valores teóricos para efectos de la comparación de la medición con los atributos de calidad especificados. El conocimiento de los valores máximos o mínimos permite el establecimiento claro del grado de cumplimiento de los atributos de calidad.

En líneas generales, el planteamiento anterior de Bosch presenta los objetivos para efectos de la medición de los atributos de calidad [40]. Sin embargo, en ocasiones, la evaluación de una arquitectura de software no produce valores numéricos que permiten la toma de decisiones de manera directa [15]. Ante la posibilidad de efectuar evaluaciones a cualquier nivel del proceso de diseño, con distintos niveles de especificación, Clements y sus colegas en [15] proponen un esquema general en relación a la evaluación de una arquitectura con respecto a sus atributos de calidad. En este sentido, afirman que de la evaluación de una arquitectura no se obtienen respuestas del tipo “si - no”, “bueno – malo” o “6.75 de 10”, sino que explica cuáles son los puntos de riesgo del diseño evaluado.

Una de las diferencias principales entre los planteamientos de [15] y [40] es el enfoque que utilizan para efectos de la evaluación. El método de diseño de arquitecturas planteado por Bosch en [40] tiene como principal característica la evaluación explícita de los atributos de calidad de la arquitectura durante el proceso de diseño de la misma. El autor afirma que el enfoque tradicional en la industria de software es el de implementar el sistema y luego establecer valores para los atributos de calidad del mismo. Este enfoque, según su experiencia, tiene la desventaja de que se destina gran cantidad de recursos y esfuerzo en el desarrollo de un sistema que no satisface los requerimientos de calidad. En este sentido, Bosch plantea las técnicas de evaluación: basada en escenarios, basada en simulación, basada en modelos matemáticos y basada en experiencia.

Por su parte, Clements y sus colegas proponen que resulta de poco interés la caracterización o medición de atributos de calidad en las fases tempranas del proceso de diseño, dado que estos parámetros son, por lo general, dependientes de la implementación. Su enfoque se orienta hacia la mitigación de riesgos, ubicando en qué lugar un atributo de calidad de interés se ve afectado por decisiones arquitectónicas. En su estudio presentan tres métodos de evaluación de

arquitecturas de software, que son *Architecture Trade-off Análisis Method (ATAM), Software Architecture Analysis Method (SAAM) y Active Intermediate Designs Review (ARID)*.

Tanto Bosch como Clements y colegas indican la importancia de la especificación exhaustiva de los atributos de calidad como base para efectos de la evaluación de una arquitectura de software. Descripciones tales como *"el sistema debe ser robusto"* o *"el sistema debe exhibir un desempeño aceptable"* resultan ambiguas, puesto que lo que se entiende de ellos puede ser diferente para los distintos involucrados con el sistema [15]. El punto es entonces definir los atributos de calidad en función de sus metas y su contexto, y no como cantidades absolutas, según Kazman y sus colegas.

De los planteamientos de evaluación establecidos por [15] y [40] se tiene que la evaluación de las arquitecturas de software puede ser realizada mediante el uso de diversas técnicas y métodos. En este sentido, resulta interesante estudiar las distintas opciones que existen en la actualidad para llevar a cabo esta tarea.

De acuerdo a lo mencionado en los párrafos anteriores se puede decir que la propuesta de Bosch es la que más se apega al desarrollo de software bajo una metodología ágil, aunque no por esto se deben descartar totalmente la propuesta de Clements y colegas como se explica en [1], en donde resalta como método de evaluación ATAM.

2.7.1 Técnicas de Evaluación de Arquitecturas de Software

Según Bosch las técnicas utilizadas para la evaluación de atributos de calidad requieren grandes esfuerzos por parte del Ingeniero de Software para crear especificaciones y predicciones. Estas técnicas requieren información del sistema a desarrollar que no está disponible durante el diseño arquitectónico, sino al principio del diseño detallado del sistema [40].

En vista de que el interés es tomar decisiones de tipo arquitectónico en las fases tempranas del desarrollo, son necesarias técnicas que requieran poca información detallada y puedan conducir a resultados relativamente precisos [40]. Las técnicas existentes en la actualidad para evaluar arquitecturas permiten hacer una evaluación cuantitativa sobre los atributos de calidad a nivel arquitectónico, pero se tienen pocos medios para predecir el máximo (o mínimo) teórico para las arquitecturas de software. Sin embargo, debido al costo de realizar este tipo de evaluación, en muchos casos los Arquitectos de Software evalúan cualitativamente, para decidir entre las alternativas de diseño [40].

2.7.1.1 Evaluación Basada en Escenarios

Un *escenario* es una breve descripción de la interacción de alguno de los involucrados en el desarrollo del sistema con éste. Por ejemplo, un usuario hará la descripción en términos de la ejecución de una tarea; un encargado de mantenimiento hará referencia a cambios que deban realizarse sobre el sistema; un desarrollador se enfocará en el uso de la arquitectura para efectos de su construcción o predicción de su desempeño [15].

Un escenario consta de tres partes: el *estímulo*, el *contexto* y la *respuesta*. El estímulo es la parte del escenario que explica o describe lo que el involucrado en el desarrollo hace para iniciar la

interacción con el sistema. Puede incluir ejecución de tareas, cambios en el sistema, ejecución de pruebas, configuración, etc. El contexto describe qué sucede en el sistema al momento del estímulo. La respuesta describe, a través de la arquitectura, cómo debería responder el sistema ante el estímulo. Este último elemento es el que permite establecer cuál es el atributo de calidad asociado [15].

Los escenarios proveen un vehículo que permite concretar y entender atributos de calidad. En [15] y [41] coinciden en la importancia del uso de los escenarios, no sólo para efectos de la evaluación de arquitecturas de software. Entre las ventajas de su uso están:

- Son simples de crear y entender
- Son poco costosos y no requieren mucho entrenamiento
- Son efectivos

Actualmente las técnicas basadas en escenarios cuentan con dos instrumentos de evaluación relevantes, estos son: el *Utility Tree* propuesto en [15] y los *Perfiles* (*Profiles)*, propuestos en [40].

Utility Tree [15]

Un *Utility Tree* es un esquema en forma de árbol que presenta los atributos de calidad de un sistema de software, refinados hasta el establecimiento de escenarios que especifican con suficiente detalle el nivel de prioridad de cada uno.

La intención del uso del *Utility Tree* es la identificación de los atributos de calidad más importantes para un proyecto particular. No existe un conjunto preestablecido de atributos, sino que son definidos por los involucrados en el desarrollo del sistema al momento de la construcción del árbol.

El Utility Tree contiene como nodo raíz la *utilidad general* del sistema. Los atributos de calidad asociados al mismo conforman el segundo nivel del árbol, los cuales se refinan hasta la obtención de un escenario lo suficientemente concreto para ser analizado y otorgarle prioridad a cada atributo considerado.

Cada atributo de calidad perteneciente al árbol contiene una serie de escenarios relacionados, y una escala de importancia y dificultad asociada, que será útil para efectos de la evaluación de la arquitectura.

Perfiles (Profiles) [40]

Un perfil (*profile*) es un conjunto de escenarios generalmente con alguna importancia relativa asociada a cada uno de ellos. El uso de perfiles permite hacer especificaciones más precisas del requerimiento para un atributo de calidad. Los perfiles tienen asociados dos formas de especificación: perfiles completos y perfiles seleccionados.

Los *perfiles completos* definen todos los escenarios relevantes como parte del perfil. Esto permite al Ingeniero de Software realizar un análisis de la arquitectura para el atributo de calidad estudiado de una manera completa, puesto que incluye todos los posibles casos. Su uso se reduce a sistemas relativamente pequeños y sólo es posible predecir conjuntos de escenarios completos para algunos atributos de calidad.

Los *perfiles seleccionados* se asemejan a la selección de muestras sobre una población en los experimentos estadísticos. Se toma un conjunto de escenarios de forma aleatoria, de acuerdo a algunos requerimientos. La aleatoriedad no es totalmente cierta por limitaciones prácticas, por lo que se fuerza la realización de una selección estructurada de elementos para el conjunto de muestra. Si bien es informal, permite hacer proposiciones científicamente válidas.

Dado que los escenarios se construyen cuidadosamente, Bosch plantea que puede asumirse que el perfil representa una imagen exacta de la población de escenarios. Para la definición de un perfil, es necesario seguir tres pasos: definición de las categorías del escenario, selección y definición de los escenarios para la categoría y asignación del "peso" a los escenarios.

Bosch establece que la definición de categorías de escenarios divide la población de escenarios en poblaciones más pequeñas, que cubren aspectos particulares del sistema. La selección y definición de escenarios para cada categoría selecciona un conjunto de escenarios representativo para la subpoblación. Luego, en la asignación del peso a los escenarios, dependiendo del perfil, el peso de un escenario tiene diferentes significados. Se definen escalas que de alguna forma sean cuantificables y puedan ser convertidas a pesos relativos.

Cada atributo de calidad tiene un perfil asociado. Algunos perfiles pueden ser usados para evaluar más de un atributo de calidad. Han sido seleccionados cinco atributos de calidad que son considerados de mayor relevancia para una perspectiva general de ingeniería de software [40]. Tales atributos son: desempeño, mantenibilidad, confiabilidad, seguridad externa *(safety)* y seguridad interna *(security)*. La Tabla 9 presenta para cada atributo de calidad, el perfil asociado, la forma en que se definen las categorías, el significado de los "pesos" y posibles métricas de evaluación, de acuerdo al planteamiento de Bosch.

Según Bosch, la técnica de evaluación basada en escenarios es dependiente de manera directa del perfil definido para el atributo de calidad que va a ser evaluado. La efectividad de la técnica es altamente dependiente de la representatividad de los escenarios. El autor propone que existe la evaluación de funcionalidad basada en escenarios, y es utilizada en el diseño orientado a objetos para especificar comportamiento del sistema. La diferencia entre este tipo de evaluación y la evaluación arquitectónica basada en escenarios radica en que la última utiliza los escenarios para evaluar atributos de calidad, en lugar de verificar o describir funcionalidad, además de que se usan otros escenarios para definir otros atributos de calidad.

Es importante destacar que la definición de los casos de uso debe ser independiente del *perfil de uso*, debido a que los casos de uso permiten optimizar el diseño arquitectónico, y pueden resultar

una muestra no representativa de la población de escenarios para la evaluación de cierto atributo de calidad [40].

De acuerdo con Bosch, la evaluación basada en escenarios puede ser empleada para comparar dos arquitecturas y para la evaluación absoluta de una arquitectura. La diferencia está en que la evaluación absoluta requiere mayor cantidad de datos estimados y cuantitativos necesarios para la evaluación.

Bosch explica que la técnica consiste en dos etapas: *análisis de impacto* y *predicción de atributos de calidad*. El *análisis de impacto* toma como entrada el perfil y la arquitectura de software. Para cada escenario del perfil, se evalúa el impacto de la arquitectura y se obtienen los resultados que serán usados en la etapa de *predicción de atributos de calidad*, donde se pronostica el valor del atributo de calidad estudiado de acuerdo a las métricas existentes.

Atributo	Perfil	Categorías	Pesos	Métricas
Mantenibilidad	Perfil de mantenimiento *(Maintainance profile)*	Se organizan alrededor de las interfaces del sistema (sistema operativo, interfaces con otros sistemas). Los escenarios de cambio describen modificaciones en los requerimientos	Indican la probabilidad de ocurrencia del cambio de escenario en un período de tiempo	• Impacto en términos de líneas de código que tienen que cambiarse • Se requiere un estimado de líneas de código de los componentes arquitectónicos
Desempeño	Perfil de uso *(Usage profile)*	Descompone los escenarios de uso basado en los tipos de usuario y/o interfaces del sistema	Representan la frecuencia relativa del escenario	• Funcionalidad de componentes • Comportamiento del sistema en respuesta a los escenarios de uso en el perfil • Promedio y peor caso de latencia por sincronización y sobrecarga en el sistema
Confiabilidad	Perfil de uso *(Usage profile)*	Confiabilidad de los componentes, genera la confiabilidad de los escenarios de uso	Indica la robustez del sistema	• Datos estimados de confiabilidad del componente • Datos históricos de confiabilidad del componente
Seguridad Interna	Perfil de seguridad *(Security profile)* Perfil de uso *(Usage profile)*	Basada en todas las interfaces del sistema	Indica la probabilidad de fallas	Depende del aspecto de seguridad a ser evaluado. Por ejemplo. la disponibilidad puede evaluarse en términos del número de veces que se ejecutan operaciones de seguridad.
Seguridad Externa	Perfil de peligro *(Hazard profile)*	Se organizan de acuerdo a documentos de certificación (sistemas médicos, puntos de interacción del sistema con el mundo real, o componentes críticos del	Indican la probabilidad de falla u ocurrencia de consecuencias desastrosas	*Bosch (2000) no establece ejemplos*

sistema)

Tabla 9. Perfiles, categorías, pesos y métricas asociados a atributos de calidad según Bosch en [40]

2.7.1.2 Evaluación Basada en Simulación

Bosch establece que la evaluación basada en simulación utiliza una implementación de alto nivel de la arquitectura de software. El enfoque básico consiste en la implementación de componentes de la arquitectura y la implementación (a cierto nivel de abstracción) del contexto del sistema donde se supone va a ejecutarse. La finalidad es evaluar el comportamiento de la arquitectura bajo diversas circunstancias. Una vez disponibles estas implementaciones, pueden usarse los perfiles respectivos para evaluar los atributos de calidad [40].

El proceso de evaluación basada en simulación sigue los siguientes pasos [40]:

- **Definición e implementación del contexto.** Consiste en identificar las interfaces de la arquitectura de software con su contexto, y decidir cómo será simulado el comportamiento del contexto en tales interfaces.
- **Implementación de los componentes arquitectónicos.** La descripción del diseño arquitectónico debe definir, por lo menos, las interfaces y las conexiones de los componentes, por lo que estas partes pueden ser tomadas directamente de la descripción de diseño. El comportamiento de los componentes en respuesta a eventos sobre sus interfaces puede no ser especificado claramente, aunque generalmente existe un conocimiento común y es necesario que el arquitecto lo interprete, por lo que éste decide el nivel de detalle de la implementación.
- **Implementación del perfil.** Dependiendo del atributo de calidad que el arquitecto de software intenta evaluar usando simulación, el perfil asociado necesitará ser implementado en el sistema. El arquitecto de software debe ser capaz de activar escenarios individuales, así como también ejecutar un perfil completo usando selección aleatoria, basado en los pesos normalizados de los mismos.
- **Simulación del sistema e inicio del perfil.** El arquitecto de software ejecutará la simulación y activará escenarios de forma manual o automática, y obtendrá resultados de acuerdo al atributo de calidad que está siendo evaluado.
- **Predicción de atributos de calidad.** Dependiendo del tipo de simulación y del atributo de calidad evaluado, se puede disponer de cantidades excesivas de datos, que requieren ser condensados. Esto permite hacer conclusiones acerca del comportamiento del sistema.

En el ámbito de las simulaciones, se encuentra la técnica de implementación de prototipos (*prototyping*). Esta técnica implementa una parte de la arquitectura de software y la ejecuta en el contexto del sistema. Es utilizada para evaluar requerimientos de calidad operacional, como desempeño y confiabilidad. Para su uso se necesita mayor información sobre el desarrollo y disponibilidad del hardware, y otras partes que constituyen el contexto del sistema de software. Se obtiene un resultado de evaluación con mayor exactitud [40].

La exactitud de los resultados de la evaluación depende, a su vez, de la exactitud del perfil utilizado para evaluar el atributo de calidad y de la precisión con la que el contexto del sistema simula las condiciones del mundo real.

En términos de los instrumentos asociados a las técnicas de evaluación basadas en simulación, se encuentran los lenguajes de descripción arquitectónica y los modelos de colas.

2.7.1.3 Evaluación Basada en Modelos Matemáticos

Bosch establece que la evaluación basada en modelos matemáticos se utiliza para evaluar atributos de calidad operacionales. Permite una evaluación estática de los modelos de diseño arquitectónico, y se presentan como alternativa a la simulación, dado que evalúan el mismo tipo de atributos. Ambos enfoques pueden ser combinados, utilizando los resultados de uno como entrada para el otro.

El proceso de evaluación basada en modelos matemáticos sigue los siguientes pasos [40]:

- **Selección y adaptación del modelo matemático.** La mayoría de los centros de investigación orientados a atributos de calidad han desarrollado modelos matemáticos para medir sus atributos de calidad, los cuales tienden a ser muy elaborados y detallados, así como también requieren de cierto tipo de datos y análisis. Parte de estos datos requeridos no están disponibles a nivel de arquitectura, y la técnica requiere mucho esfuerzo para la evaluación arquitectónica, por lo que el arquitecto de software se ve obligado a adaptar el modelo.
- **Representación de la arquitectura en términos del modelo.** El modelo matemático seleccionado y adaptado no asume necesariamente que el sistema que intenta modelar consiste de componentes y conexiones. Por lo tanto, la arquitectura necesita ser representada en términos del modelo.
- **Estimación de los datos de entrada requeridos.** El modelo matemático aún cuando ha sido adaptado, requiere datos de entrada que no están incluidos en la definición básica de la arquitectura. Es necesario estimar y deducir estos datos de la especificación de requerimientos y de la arquitectura diseñada.
- **Predicción de atributos de calidad.** Una vez que la arquitectura es expresada en términos del modelo y se encuentran disponibles todos los datos de entrada requeridos, el arquitecto está en capacidad de calcular la predicción resultante del atributo de calidad evaluado.

Entre las desventajas que presenta esta técnica se encuentra la inexistencia de modelos matemáticos apropiados para los atributos de calidad relevantes [40], y el hecho de que el desarrollo de un modelo de simulación completo puede requerir esfuerzos sustanciales.

Entre los instrumentos que se cuentan para las técnicas de evaluación de arquitecturas de software basada en modelos matemáticos, se encuentran las *Cadenas de Markov* y los *Reliability Block Diagrams*.

2.7.1.4 Evaluación Basada en Experiencia

Bosch establece que en muchas ocasiones los arquitectos e ingenieros de software otorgan valiosas ideas que resultan de utilidad para la evasión de decisiones erradas de diseño. Aunque todas estas experiencias se basan en evidencia anecdótica; es decir, basada en factores subjetivos como la intuición y la experiencia. Sin embargo, la mayoría de ellas puede ser justificada por una línea lógica de razonamiento, y pueden ser la base de otros enfoques de evaluación [40].

Existen dos tipos de evaluación basada en experiencia: la *evaluación informal*, que es realizada por los arquitectos de software durante el proceso de diseño, y la realizada por equipos externos de evaluación de arquitecturas.

La Tabla 10 presenta de forma resumida los instrumentos utilizados por las diferentes técnicas de evaluación.

Técnica de Evaluación	Instrumento de Evaluación
Basada en Escenarios	• Perfiles (Profiles) • Utility Tree
Basada en Simulación	• Lenguaje de Descripción Arquitectónica (ADL) • Modelos de Colas
Basada en Modelos Matemáticos	• Cadenas de Markov • Reliability Block Diagrams
Basada en Experiencia	• Intuición y Experiencia • Tradición • Proyectos Similares

Tabla 10. Instrumentos Asociados a las Distintas Técnicas de Evaluación de Arquitecturas de Software

De las técnicas anteriores resaltan las Basadas en Escenarios, Simulación y Experiencia como técnicas que pueden ser integradas en desarrollos bajo metodologías ágiles.

Clements y colegas en [15] proponen que la existencia de un método de análisis de arquitecturas de software hace que el proceso sea repetible, y ayuda a garantizar que las respuestas correctas con relación a la arquitectura pueden hacerse temprano, durante las fases tempranas de diseño. Es en este punto donde los problemas encontrados pueden ser solucionados de una forma relativamente poco costosa. De manera similar, un método de evaluación sirve de guía a los involucrados en el desarrollo del sistema, en la búsqueda de conflictos que puede presentar una arquitectura, y sus soluciones. Por esta razón, resulta conveniente estudiar los métodos de evaluación de arquitecturas de software propuestos hasta el momento.

2.7.2 Métodos de Evaluación de Arquitecturas de Software

De acuerdo con [15], hasta hace poco no existían métodos de utilidad general para evaluar arquitecturas de software. Si alguno existía, sus enfoques eran incompletos, ad hoc, y no repetibles, lo que no brindaba mucha confianza. En virtud de esto, múltiples métodos de evaluación han sido propuestos. A continuación se explican algunos de los más importantes.

2.7.2.1 Software Architecture Analysis Method (SAAM)

El Método de Análisis de Arquitecturas de Software (*Software Architecture Analysis Method, SAAM*) [15] es el primero que fue ampliamente promulgado y documentado. El método fue originalmente creado para el análisis de la modificabilidad de una arquitectura, pero en la práctica ha demostrado ser muy útil para evaluar de forma rápida distintos atributos de calidad, tales como modificabilidad, portabilidad, escalabilidad e integrabilidad.

El método de evaluación SAAM se enfoca en la enumeración de un conjunto de escenarios que representan los cambios probables a los que estará sometido el sistema en el futuro. Como entrada principal, es necesaria alguna forma de descripción de la arquitectura a ser evaluada. Las salidas de la evaluación del método SAAM son las siguientes:

- Una proyección sobre la arquitectura de los escenarios que representan los cambios posibles ante los que puede estar expuesto el sistema.
- Entendimiento de la funcionalidad del sistema, e incluso una comparación de múltiples arquitecturas con respecto al nivel de funcionalidad que cada una soporta sin modificación.

Con la aplicación de este método, si el objetivo de la evaluación es una sola arquitectura, se obtienen los lugares en los que la misma puede fallar, en términos de los requerimientos de modificabilidad. Para el caso en el que se cuenta con varias arquitecturas candidatas, el método produce una escala relativa que permite observar qué opción satisface mejor los requerimientos de calidad con la menor cantidad de modificaciones.

La Tabla 11 presenta los pasos que contempla el método de evaluación SAAM, con una breve descripción [15].

1. Desarrollo de Escenarios	Un escenario es una breve descripción de usos anticipados o deseados del sistema. De igual forma, estos pueden incluir cambios a los que puede estar expuesto el sistema en el futuro.
2. Descripción de la arquitectura	La arquitectura (o las candidatas) debe ser descrita haciendo uso de alguna notación arquitectónica que sea común a todas las partes involucradas en el análisis. Deben incluirse los componentes de datos y conexiones relevantes, así como la descripción del comportamiento general del sistema. El desarrollo de escenarios y la descripción de la arquitectura son usualmente llevados a cabo de forma intercalada, o a través de varias iteraciones.
3. Clasificación y asignación de prioridad de los escenarios	La clasificación de los escenarios puede hacerse en dos clases: directos o indirectos. Un escenario *directo* es el que puede satisfacerse sin la necesidad de modificaciones en la arquitectura. Un escenario *indirecto* es aquel que requiere modificaciones en la arquitectura para poder satisfacerse. Los escenarios indirectos son de especial interés para SAAM, pues son los que permiten medir el grado en el que una arquitectura puede ajustarse a los cambios de evolución que son importantes para los involucrados en el desarrollo.
4. Evaluación individual de los escenarios indirectos	Para cada escenario indirecto, se listan los cambios necesarios sobre la arquitectura, y se calcula su costo. Una modificación sobre la arquitectura significa que debe introducirse un nuevo componente o conector, o que alguno de los existentes requiere cambios en su especificación.

5. Evaluación de la interacción entre escenarios	Cuando dos o más escenarios indirectos proponen cambios sobre un mismo componente, se dice que *interactúan* sobre ese componente. Es necesario evaluar este hecho, puesto que la interacción de componentes semánticamente no relacionados revela que los componentes de la arquitectura efectúan funciones semánticamente distintas. De forma similar puede verificarse si la arquitectura se encuentra documentada a un nivel correcto de descomposición estructural.
6. Creación de la evaluación global	Debe asignársele un peso a cada escenario, en términos de su importancia relativa al éxito del sistema. Esta asignación de peso suele hacerse con base en las metas del negocio que cada escenario soporta. En el caso de la evaluación de múltiples arquitecturas, la asignación de pesos puede ser utilizada para la determinación de una escala general.

Tabla 11. Pasos Contemplados por el Método de Evaluación SAAM

2.7.2.2 Architecture Trade-off Analysis Method (ATAM)

El Método de Análisis de Acuerdos de Arquitectura (*Architecture Trade-off Analysis Method, ATAM*) [15] está inspirado en tres áreas distintas: los estilos arquitectónicos, el análisis de atributos de calidad y el método de evaluación SAAM, explicado anteriormente. El nombre del método ATAM surge del hecho de que revela la forma en que una arquitectura específica satisface ciertos atributos de calidad, y provee una visión de cómo los atributos de calidad interactúan con otros; esto es, los tipos de acuerdos que se establecen entre ellos.

El método se concentra en la identificación de los estilos arquitectónicos o enfoques arquitectónicos utilizados. Clements y colegas proponen el término enfoque arquitectónico dado que no todos los arquitectos están familiarizados con el lenguaje de estilos arquitectónicos, aún haciendo uso indirecto de estos. De cualquier forma, estos elementos representan los medios empleados por la arquitectura para alcanzar los atributos de calidad, así como también permiten describir la forma en la que el sistema puede crecer, responder a cambios, e integrarse con otros sistemas, entre otros [15].

El método de evaluación ATAM comprende nueve pasos, agrupados en cuatro fases. La Tabla 12 presenta las fases y sus pasos enumerados, junto con su descripción.

Fase 1. Presentación	
1. Presentación del ATAM	El líder de evaluación describe el método a los participantes, trata de establecer las expectativas y responde las preguntas propuestas.
2. Presentación de las metas del negocio	Se realiza la descripción de las metas del negocio que motivan el esfuerzo, y aclara que se persiguen objetivos del tipo arquitectónico.
3. Presentación de la arquitectura	El arquitecto describe la arquitectura, enfocándose en cómo ésta cumple con los objetivos del negocio.
Fase 2. Investigación y análisis	
4. Identificación de los enfoques arquitectónicos	Estos elementos son detectados, pero no analizados.
5. Generación del *Utility Tree*	Se especifica en forma de escenarios cada atributo de calidad que engloba la "utilidad" del sistema (desempeño, disponibilidad, seguridad, modificabilidad, usabilidad, etc.), especificados en forma de escenarios. Se anotan los estímulos y respuestas, así como se establece la prioridad entre ellos.
6. Análisis de los enfoques arquitectónicos	Con base en los resultados del establecimiento de prioridades del paso anterior, se analizan los elementos del paso 4. En este paso se identifican

	riesgos arquitectónicos, puntos de sensibilidad y puntos de balance.
Fase 3. Pruebas	
7. Lluvia de ideas y establecimiento de prioridad de escenarios	Con la colaboración de todos los involucrados, se completa el conjunto de escenarios.
8. Análisis de los enfoques arquitectónicos	Este paso repite las actividades del paso 6, haciendo uso de los resultados del paso 7. Los escenarios son considerados como casos de prueba para confirmar el análisis realizado hasta el momento.
Fase 4. Reporte	
9. Presentación de los resultados	Basado en la información recolectada a lo largo de la evaluación del ATAM, se presentan los hallazgos a los participantes.

Tabla 12. Pasos del Método de Evaluación ATAM

2.7.2.3 Active Reviews for Intermediate Designs (ARID)

El método ARID [15] es conveniente para realizar la evaluación de diseños parciales en las etapas tempranas del desarrollo. En ocasiones, es necesario saber si un diseño propuesto es conveniente, desde el punto de vista de otras partes de la arquitectura. Según los autores, ARID es un híbrido entre *Active Design Review (ADR)* y *Architecture Trade-Off Method (ATAM)*, descrito anteriormente.

ADR es utilizado para la evaluación de diseños detallados de unidades del software como los componentes o módulos. Las preguntas giran en torno a la calidad y completitud de la documentación y la suficiencia, el ajuste y la conveniencia de los servicios que provee el diseño propuesto.

Clements y colegas proponen que tanto ADR como ATAM proveen características útiles para el problema de la evaluación de diseños preliminares, dado que ninguno por sí solo es conveniente. En el caso de ADR, los involucrados reciben documentación detallada y completan cuestionarios, cada uno por separado. En el caso de ATAM, está orientado a la evaluación de toda una arquitectura.

Ante esta situación, y la necesidad de evaluación en las fases tempranas del diseño, en [15] proponen la utilización de las características que proveen tanto ADR como ATAM por separado. De ADR, resulta conveniente la fidelidad de las respuestas que se obtiene de los involucrados en el desarrollo. Así mismo, la idea del uso de escenarios generados por los involucrados con el sistema es tomada del ATAM. De la combinación de ambas filosofías surge ARID, para efecto de la evaluación temprana de los diseños de una arquitectura de software.

La Tabla 13 presenta los pasos que involucra el método de evaluación ARID, con una breve descripción de cada uno.

Fase 1. Actividades Previas	
1. Identificación de los encargados de la revisión	Los encargados de la revisión son los ingenieros de software que se espera que usen el diseño, y todos los involucrados en el diseño. En este punto, converge el concepto de encargado de *revisión* de ADR e *involucrado* del ATAM
2. Preparar el informe de diseño	El diseñador prepara un informe que explica el diseño. Se incluyen ejemplos del uso del mismo para la resolución de problemas reales. Esto permite al facilitador anticipar el tipo de preguntas posibles, así como identificar áreas en las que la presentación puede ser mejorada.
3. Preparar los escenarios base	El diseñador y el facilitador preparan un conjunto de escenarios base. De forma similar a los escenarios del ATAM y el SAAM, se diseñan para ilustrar el concepto de escenario, que pueden o no ser utilizados para efectos de la evaluación.
4. Preparar los materiales	Se reproducen los materiales preparados para ser presentados en la segunda fase. Se establece la reunión, y los involucrados son invitados.
Fase 2. Revisión	
5. Presentación del ARID	Se explica los pasos del ARID a los participantes.
6. Presentación del diseño	El líder del equipo de diseño realiza una presentación, con ejemplos incluidos. Se propone evitar preguntas que conciernen a la implementación o argumentación, así como alternativas de diseño. El objetivo es verificar que el diseño es conveniente.
7. Lluvia de ideas y establecimiento de prioridad de escenarios	Se establece una sesión para la lluvia de ideas sobre los escenarios y el establecimiento de prioridad de escenarios. Los involucrados proponen escenarios a ser usados en el diseño para resolver problemas que esperan encontrar. Luego, los escenarios son sometidos a votación, y se utilizan los que resultan ganadores para hacer pruebas sobre el diseño.
8. Aplicación de los escenarios	Comenzando con el escenario que contó con más votos, el facilitador solicita pseudo-código que utiliza el diseño para proveer el servicio, y el diseñador no debe ayudar en esta tarea. Este paso continúa hasta que ocurra alguno de los siguientes eventos: • Se agota el tiempo destinado a la revisión. • Se han estudiado los escenarios de más alta prioridad. • El grupo se siente satisfecho con la conclusión alcanzada. Puede suceder que el diseño presentado sea conveniente, con la exitosa aplicación de los escenarios, o por el contrario, no conveniente, cuando el grupo encuentra problemas o deficiencias.
9. Resumen	Al final, el facilitador recuenta la lista de puntos tratados, pide opiniones de los participantes sobre la eficiencia del ejercicio de revisión, y agradece por su participación.

Tabla 13. Pasos del Método de Evaluación ARID

En el contexto de los métodos de evaluación, In en [42] proponen un modelo de referencia general para efectos de la toma de decisiones, basado en *Cost Benefit Analisys Method (CBAM)* y el método de negociación *WinWin*. Su propuesta pretende ayudar en la selección sistemática entre arquitecturas candidatas, con requerimientos que se negocian entre los involucrados del desarrollo. En este sentido, se presentan los conceptos del modelo de negociación WinWin, CBAM y el marco de referencia propuesto por In en [42].

2.7.2.4 Modelo de Negociación *WinWin*

De acuerdo con In en [42], el modelo *WinWin* provee un marco de referencia general para identificar y resolver conflictos de requerimientos, mediante la obtención y negociación de artefactos en función de las condiciones de ganancia. El modelo utiliza la teoría "W", que pretende que todo involucrado salga ganador. De esta forma, asiste a los involucrados en el desarrollo a identificar y negociar distintos aspectos, reconciliando conflictos entre las opciones de ganancias para todos.

Aunque el modelo propone la resolución de posibles conflictos, no siempre es posible llegar a un acuerdo [42]. En este sentido, el método CBAM provee medios para establecer los balances necesarios, y un marco de referencia para la discusión que puede llevar a una posible solución del problema.

2.7.2.5 Cost-Benefit Analysis Method (CBAM)

De acuerdo con In en [42], el método de análisis de costos y beneficios es un marco de referencia que no toma decisiones por los involucrados en el desarrollo del sistema. Por el contrario, ayuda en la obtención y documentación de los costos beneficios e incertidumbre, y provee un proceso de toma de decisiones racional. Uno de los elementos que introduce el método son las llamadas estrategias arquitectónicas, que consisten en posibles opciones para la resolución de conflictos entre atributos de calidad presentes en una arquitectura.

El método CBAM abarca los siguientes aspectos [42]:

- Selección de escenarios
- Evaluación de los beneficios de los atributos de calidad
- Cuantificación de los beneficios de las estrategias arquitectónicas
- Cuantificación de los costos de las estrategias arquitectónicas y las implicaciones de calendario
- Cálculo del nivel de deseabilidad
- Toma de decisiones

El modelo de referencia para evaluación de arquitecturas de software propuesto por In en [42] comienza con el método *WinWin*, para efectos de la obtención de las necesidades de los involucrados y la exploración de las opciones de resolución de conflictos. El método CBAM se propone como solución al esquema sistemático planteado por *WinWin*, dado que propone la evaluación de costos y beneficios. La Tabla 14 presenta los pasos pertenecientes al marco de referencia para evaluación de arquitecturas de software propuesto por In en [42].

1. Selección de escenarios	Cada involucrado en el desarrollo identifica sus condiciones de ganancia. Este paso provee las bases para la identificación de las características ideales del sistema esperadas por los involucrados.
2. Identificación de los conflictos entre atributos de calidad	La lista de condiciones de ganancia es revisada con la intención de identificar conflictos entre atributos de calidad. Los conflictos son categorizados en directos o potenciales.
3. Exploración de las	Con base en los conflictos detectados en el paso anterior, los involucrados

opciones en busca de la resolución de conflictos	pueden generar opciones de resolución de conflictos, o estrategias arquitectónicas.
4. Medición de los beneficios de los atributos de calidad	Para ayudar en el proceso de toma de decisiones, los involucrados en el desarrollo deben calcular tanto los costos como los beneficios de las estrategias arquitectónicas elaboradas en el paso anterior. Para esto, se calcula una escala de atributos de calidad, donde se asigna un puntaje a cada atributo.
5. Cuantificación de los beneficios	Las escalas establecidas son utilizadas para la evaluación de las estrategias arquitectónicas planteadas en el paso 3. El resultado de la evaluación permite observar el beneficio de cada uno de los cambios arquitectónicos propuestos.
6. Cuantificación de costos e implicaciones de calendario	En este paso se calculan los costos e implicaciones de calendario que aplican para cada una de las estrategias arquitectónicas propuestas en el paso 3. No se propone un modelo específico para la realización de la estimación de costos y tiempo.
7. Cálculo del nivel de deseabilidad	Este paso contempla una métrica especial, que se refleja como el cociente entre los beneficios y los costos obtenidos. Las estrategias arquitectónicas que resulten con valores mayores se proponen como las más recomendables.
8. Alcanzar un acuerdo	La recomendación general es la documentación del proceso. La acumulación de evidencia permite el establecimiento de un posible consenso, a la hora de la toma de decisiones sobre la arquitectura de un sistema de software.

Tabla 14. Pasos Contemplados en el Marco de Referencia para la Evaluación de Arquitecturas de Software Propuesto por In en [42]

2.7.2.6 Método de Diseño y Uso de Arquitecturas de Software propuesto por Bosch

Bosch plantea, en su método de diseño de arquitecturas de software [40], que el proceso de evaluación debe ser visto como una actividad iterativa, que forma parte del proceso de diseño, también iterativo. Una vez que la arquitectura es evaluada, pasa a una fase de transformación, asumiendo que no satisface todos los requerimientos. Luego, la arquitectura transformada es evaluada de nuevo.

El proceso de evaluación propuesto por Bosch se divide en dos etapas, que son presentadas en la Tabla 15.

Etapa I	
1. Selección de atributos de calidad	Deben seleccionarse aquellos atributos que se consideran cruciales para el éxito del sistema, y cuya satisfacción resulte poco clara a nivel de arquitectura. Resulta necesario porque es poco factible y poco útil evaluar todos los atributos de calidad, dado que requiere una gran cantidad de tiempo.
2. Definición de los perfiles	Para cada atributo de calidad seleccionado, se definen los perfiles respectivos para efectos de la evaluación.
3. Selección de una técnica de evaluación	Para la evaluación de los atributos de calidad dependientes del diseño de la arquitectura se recomienda utilizar la evaluación basada en escenarios, así como también los modelos basados en métricas o modelo matemáticos. Los atributos de calidad operacionales (observables vía ejecución) pueden evaluarse con técnicas de simulación o modelos matemáticos. La selección de la técnica, y la implementación concreta de ésta depende del objetivo y exactitud de la evaluación.
Etapa II	
4. Ejecución de la evaluación	Para cada atributo de calidad, las técnicas arrojan valores cuantitativos.
5. Obtención de	Los resultados se resumen en una tabla que contiene el nivel requerido, el nivel

resultados	predicho, y un indicador, que puede tener diversos significados: si el atributo se satisface o no, si necesita ser negociado con el cliente, o existencia de alguna relación negativa con otro atributo de calidad. El arquitecto puede decidir acerca de la realización de transformaciones sobre la arquitectura actual, y efectuar una nueva evaluación. Una vez que concluye el proceso de evaluación, con los resultados obtenidos es posible decidir entre la continuación, renegociación o cancelación del proyecto.

Tabla 15. Etapas Contempladas por el Método de Evaluación de Arquitecturas Propuesto por Bosch [40]

2.7.2.7 Método de comparación de arquitecturas basada en el modelo ISO/IEC 9126 adaptado para arquitecturas de software

Losavio en [18] proponen un método para evaluar y comparar arquitecturas de software candidatas, que hace uso del modelo de especificación de atributos e calidad adaptado del modelo ISO/IEC 9126. Los autores plantean que la especificación de los atributos de calidad haciendo uso de un modelo basado en estándares internacionales ofrece una vista amplia y global de los atributos de calidad, tanto a usuarios como arquitectos del sistema, para efectos de la evaluación. El método contempla siete actividades, que son descritas en la Tabla 16.

Actividades
1. Analizar los requerimientos funcionales y no funcionales principales del sistema, para establecer las metas de calidad
2. Utilizar el modelo de calidad ISO/IEC 9126 adaptado para arquitecturas de software. Algunas métricas pueden definirse con mayor nivel de detalle.
3. Presentar las arquitecturas candidatas iniciales.
4. Construir la tabla comparativa para las arquitecturas candidatas.
5. Establecer prioridades para las subcaracterísticas de calidad tomando en cuenta los requerimientos de calidad del sistema.
6. Analizar los resultados obtenidos y resumidos en la tabla, de acuerdo con las prioridades establecidas.

Tabla 16. Actividades Contempladas en el Método de Comparación de Arquitecturas Basado en el Modelo ISO/IEC 9126 Adaptado para Arquitecturas de Software. Fuente: Losavio [18]

2.7.2.8 Comparación entre métodos de evaluación

La Tabla 17 presenta una comparación entre los métodos de evaluación Software Architecture Analysis Method (SAAM), Architecture Trade-off Analysis Method (ATAM), Active Reviews for Intermediate Designs (ARID), Modelo de Negociación WinWin, Cost- Benefit Analysis Method (CBAM), el método de evaluación de arquitecturas propuesto por Bosch en [40] y el método de comparación de arquitecturas basada en el modelo ISO/IEC 9126 adaptado para arquitecturas de software, planteado por Losavio en [18].

	ATAM	SAAM	ARID	WinWin	CBAM	Bosch (2000)	Losavio (2003)
Atributos de calidad contemplados	• Modificabilidad • Seguridad • Confiabilidad • Desempeño	• Modificabilidad • Funcionalidad	• Conveniencia del diseño evaluado	• Funcionalidad	• Funcionalidad • Modificabilidad	• Seleccionados por el arquitecto, de acuerdo a la importancia sobre el sistema	• Funcionaliidad • Confiabilidad • Usabilidad • Eficiencia • Mantenibilidad • Portabilidad
Objetos analizados	• Estilos arquitectónicos • Documentación • Flujo de datos • Vistas arquitectónicas	• Documentación • Vistas Arquitectónicas	• Especificación de los componentes	• Conflictos entre requerimientos	• Estilos arquitectónicos • Documentación	• Vistas arquitectónicas • Estilos arquitectónicos • Patrones Arquitectónicos • Patrones de Diseño • Patrones de Idioma (Idioms)	• Especificación de atributos de calidad
Etapas del proyecto en las que se aplica	Luego de que el diseño de la arquitectura ha sido establecido	Luego de que la arquitectura cuenta con funcionalidad ubicada en módulos	A lo largo de la arquitectura	Luego de que el diseño de la arquitectura ha sido establecido	Luego de que el diseño de la arquitectura ha sido establecido	Luego de que el diseño de la arquitectura ha sido establecido	Luego de que el diseño de la arquitectura ha sido establecido
Enfoques utilizados	• Utility Tree y lluvia de ideas para articular los requerimientos de calidad • Análisis arquitectónico que detecta puntos sensibles, puntos de balance y riesgos	• Lluvia de ideas para escenarios y articular los requerimientos de calidad • Análisis de los escenarios para verificar funcionalidad o estimar el costo de los cambios	• Revisiones de diseños, lluvia de ideas para obtener escenarios	• Teoría "W"	• Teoría "W" • Análisis de escenarios	• Análisis de perfiles (*profiles*)	• Análisis de comparación de los resultados obtenidos para las arquitecturas candidatas

Tabla 17. Comparación entre Métodos de Evaluación. Fuente: Clements et al. [15]

2.7.3 Auditorías Arquitectónicas

El propósito de la evaluación de una arquitectura de software es evaluar el estado del proyecto visto desde su arquitectura, su organización y su proceso. Cualquier proyecto se puede beneficiar de la realización periódica de evaluaciones, estas sirven por lo menos para tres metas:

- Hacen visibles los riesgos para alcanzar el éxito de un proyecto, haciendo posible diseñar un plan para mitigar dichos riesgos.
- Dan al proyecto la oportunidad de hacer correcciones a mitad de camino.
- Dan al equipo una sensación de cierre a través del reconocimiento de lo que se ha logrado.

Pueden ser realizadas interna o externamente. El tener un evaluador externo es particularmente útil para exponer las áreas de riesgo que están en negación o se ignoran dentro del proyecto.

El simple hecho de realizar la evaluación de la arquitectura de software de un proyecto tiene valor porque obliga al equipo a comunicarse entre los miembros y a racionalizar la arquitectura del proyecto. Además permite evaluar el ambiente de desarrollo y el equipo involucrado en el proyecto, Booch en [43] menciona que si durante la preparación de una evaluación cesan las actividades normales del proyecto y se tarda el equipo en estar listo para realizar dicha evaluación es un signo de advertencia de que algo está fundamentalmente mal. Los proyectos en un estado saludable requieren de una preparación mínima, esto debido a que internamente han reconocido la importancia de la arquitectura y han organizado al equipo y al proceso como corresponde.

Philippe Kruchten describe el proceso de la evaluación arquitectónica de la siguiente forma:

Definir la(s) pregunta(s) exactas o el enfoque: ¿Qué evaluar? y ¿Porqué?
Definir el equipo, la agenda y la información necesaria
Obtener la información:
    Reunir resúmenes técnicos
    "Minar" la arquitectura/Definir escenarios/Definir mediciones
ciclo
    Entrevistas/Examinar/Leer/Comparar
    Identificar problemas potenciales
    Confirmar que realmente son problemas
    Identificar mejoras o cambios
fin del ciclo
Preparar borrador del reporte y recomendaciones
Revisar el borrador del reporte, eliminar tendencias, refinar acciones
Preparar reporte final
Presentar conclusiones
Explotar los resultados/ Realizar las acciones pertinentes

La esencia del proceso anterior es establecer los puntos a ser examinados, explorar la arquitectura, la organización y los procesos, para después actuar para resolver los problemas que se encontraron.

Dependiendo del tamaño del proyecto, una evaluación típica toma en promedio desde un día hasta un par de semanas. Para proyectos grandes Booch recomienda hacer estas evaluaciones periódicamente para ver las tendencias a largo plazo, y permite tener tiempo dentro del proyecto para realizar correcciones tempranamente.

Debido a la diversidad de proyectos no hay una receta para realizar una evaluación, sin embargo dentro de las áreas más comunes en las que surgen los problemas son:

Arquitectura

En lo que respecta a la arquitectura, la primera pregunta es "¿Está definida una arquitectura?", si la respuesta es NO entonces este proyecto este proyecto necesita ayuda seriamente. Al presentarse esta situación hay dos opciones: 1) Crear una arquitectura y desarrollar un plan para imponerla incrementalmente en el sistema, 2) Abandonar el proyecto. A veces el proyecto está más allá de necesitar ayuda y usualmente lo más racional en estos casos es mejor cancelarlo.

Si está definida una arquitectura, las siguientes preguntas a realizar son sobre la naturaleza de ésta:

Primero, "¿Todo el equipo ve la arquitectura de la misma manera?", para responder esta pregunta se deben entrevistar a todos los miembros del proyecto individualmente separándolos del administrador del proyecto, esto permitirá descubrir si las políticas del proyecto están en práctica y conocer si el equipo tiene la misma visión arquitectónica.

Al preguntar cuestiones básicas de la arquitectura como "¿Cuál es el significado de una clase cliente?" a diferentes desarrolladores a niveles diferentes del sistema permite conocer la presencia arquitectónica, al haber consistencia en las respuestas se encuentra un indicador de una fuerte presencia arquitectónica, si hay inconsistencia es un indicador de una deficiencia arquitectónica o.

También se debe analizar la forma en la que se muestra la arquitectura de software, ya sea los diagramas de clases o categorías, etc., la forma en la que se divulga y está disponible esta información, esta es una forma de ver también la consistencia de la información arquitectónica con la que cuentan los desarrolladores y permite mostrar si hay una visión en común.

Segundo, "¿En el proyecto se resuelven problemas comunes en formas comunes?", por ejemplo: "¿Cuál es el mecanismo de manejo de excepciones?", "¿Hay un estándar *look-and-feel* para las interfaces?", "¿Existe un estándar para acceder a datos persistentes?". Básicamente, lo que se trata de encontrar con estas preguntas son dos cosas: Buscar el uso de patrones y Buscar el enfoque del proyecto frente a problemas que surgen comúnmente en los sistemas.

Durante el análisis de la arquitectura también se debe considerar la existencia de métricas, por ejemplo:

- El número de clases en relación con el número de líneas de código
- La distribución de las responsabilidades entre clases.
- El número de categorías en el sistema y el número de clases/mecanismos por categoría.

Booch [43] menciona que la experiencia muestra que debe haber un balance entre estas categorías. Por ejemplo, un sistema con sólo 10 clases y 100,000 líneas de código es probablemente fuera de balance. De manera similar, un sistema con 1000 clases y sólo un par de categorías está probablemente fuera de balance.

Organización

Al evaluar el equipo del proyecto, algunas de las preguntas a realizar son:

- ¿Hay un arquitecto identificable?
- ¿Cuál es la relación entre la administración del proyecto y los desarrolladores?
- ¿Existen usuarios finales como parte del proceso de desarrollo?
- Para proyectos grandes: ¿Existe una capa intermedia de administración técnica identificable?

Básicamente, estas preguntas permiten saber si el equipo está organizado para fomentar la comunicación entre múltiples partes, o si es un grupo osificado que se basa en ceremonias para llevar a cabo su trabajo.

Proceso

Para evaluar el proceso la principal pregunta es "¿El equipo tiene un ritmo regular de entregas?", en otras palabras "¿Se tiene instituido un proceso incremental e iterativo?". Los proyectos en crisis tienden a correr a un ritmo frenético, generando nuevas entregas oportunamente. Los proyectos estancados tienden a tener entregas poco regulares, y lo tratan de solucionar dejando todo para un solo evento de integración. Se deben buscar un programa de entregas regulares (por ejemplo semanales para entregas internas, cada pocos meses para entregas externas). Estas preguntas buscan revelar si los proyectos tienen ritmo, si no es así las acción apropiada es ejercer control sobre el proyecto haciendo que tenga un ritmo regular.

Otras preguntas respecto al proceso son:

- ¿Cómo se llevan a cabo las revisiones?
- ¿Qué métricas se reúnen?
- ¿Cuáles son los artefactos tangibles que se generan, independientemente del software?

La presencia de revisiones, métricas y artefactos permiten conocer mucha información respecto al proceso. Los proyectos en crisis tienden a abandonar todas estas cuestiones y no vuelven a

retomarlas. Los proyectos saludables tienden a realizar revisiones arquitectónicas tanto como revisiones entre pares. De manera similar, los proyectos en crisis ignoran toda clase de métricas formales, mientras que los proyectos saludables siguen al menos métricas básicas como densidad de defectos, descubrimiento de defectos, el ritmo de cambio de las partes del sistema. Finalmente, la ausencia de cualquier documentación es signo de un proyecto al borde del colapso, la presencia de la documentación adecuada (por ejemplo, la descripción arquitectónica) es un buen signo de un proyecto saludable.

Una evaluación efectiva de una arquitectura de software debe tener tres elementos básicos:

- Proveer una evaluación franca del estado del proyecto. De lo contrario se estará negando la realidad.
- Identifica las áreas de riesgo. En otras palabras, cuáles son los problemas que se deben evitar o resolver para alcanzar el éxito.
- Establece un plan de acción para mitigar los riesgos encontrados.

La entrega de una evaluación requiere de mucho tacto y diplomacia. Los proyectos necesitan escuchar las buenas noticias y las malas, algunas veces esto significa decir cosas que perturbarán a la estructura política de la organización. De cualquier manera, si el equipo está centrado en construir software de calidad y no imperios, entonces no importa lo desagradable que sea el mensaje, una evaluación franca sirve para hacer el proyecto más saludable o cancelarlo en última instancia.

Capítulo 3. Metodologías Ágiles

3.1 Breve Historia de las Metodologías Ágiles

Actualmente existen numerosas propuestas de metodologías para desarrollar software. Tradicionalmente estas metodologías se centran en el control del proceso, estableciendo rigurosamente las actividades, herramientas y notaciones al respecto, bajo estas reglas las metodologías se caracterizan por ser rígidas y dirigidas por la documentación que se genera en cada una de las actividades desarrolladas.

Este enfoque no resulta ser el más adecuado para muchos de los proyectos actuales donde el entorno del sistema es muy cambiante, y en donde se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad.

En este escenario, las metodologías ágiles emergen como una posible respuesta para llenar ese vacío metodológico. En febrero de 2001, tras una reunión celebrada en Utah-EEUU, nace el término ágil aplicado al desarrollo de software. En esta reunión participan un grupo de 17 expertos de la industria del software incluyendo algunos de los creadores o impulsores de metodologías de software. Su objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar software respondiendo a los cambios que puedan surgir a lo largo del proyecto.

Tras esta reunión se creó *The Agile Alliance*[5] una organización, sin ánimo de lucro, dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a las organizaciones para que adopten dichos conceptos. El punto de partida fue el Manifiesto Ágil, un documento que resume la filosofía ágil.

3.2 Manifiesto Ágil[6]

Según el manifiesto se valora:

- **A los individuos e interacciones del equipo** sobre los *procesos y herramientas.* La gente es el principal factor de éxito de un proyecto software. Es más importante construir un buen equipo que construir el entorno.
- **El desarrollo de software que funciona** sobre una *documentación exhaustiva.* La regla a seguir es no producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante. Estos documentos deben ser cortos y centrarse en lo fundamental.
- **La colaboración con el cliente** sobre la *negociación de contratos.* Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. La colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.
- **Responder ante los cambios** sobre el *seguimiento estricto de un plan*. La habilidad de responder a los cambios que puedan surgir a lo largo del proyecto (cambios en los

[5] http://www.agilealliance.com/
[6] http://agilemanifesto.org/, http://www.agile-spain.com/manifiesto_agil

requerimientos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

El manifiesto considera que aunque los elementos a la derecha tienen valor, se valora por encima de ellos los que están a la izquierda.

3.3 Principios Ágiles[7]

Los valores del manifiesto anteriores inspiraron doce principios. En estos principios se fundamentan las características que diferencian un proceso ágil de uno tradicional. Los dos primeros principios son generales y resumen gran parte del espíritu ágil. El resto tienen que ver con el proceso a seguir y con el equipo de desarrollo, en cuanto metas a seguir y organización del mismo.

Los principios son:

1. La mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor.
2. Aceptar requerimientos cambiantes, incluso en etapas avanzadas. Los procesos ágiles aprovechan el cambio para proporcionar una ventaja competitiva al cliente.
3. Entregar software frecuentemente, con una periodicidad desde un par de semanas a un par de meses, con preferencia por los periodos más cortos posibles.
4. Los responsables de negocio y los desarrolladores deben trabajar juntos diariamente a lo largo del proyecto.
5. Construir proyectos con profesionales motivados. Dándoles el entorno y soporte que necesitan, y confiando en ellos para que realicen el trabajo.
6. El método más eficiente y efectivo de comunicar la información a un equipo de desarrollo y entre los miembros del mismo es la conversación cara a cara.
7. La principal medida de progreso es el software que funciona.
8. Los procesos ágiles promueven el desarrollo sostenible. Promotores, desarrolladores y usuarios deben ser capaces de mantener un ritmo constante de forma indefinida.
9. La atención continua a la calidad técnica y los buenos diseños mejoran la agilidad.
10. La simplicidad es esencial.
11. Las mejores arquitecturas, requerimientos y diseños surgen de equipos que se auto organizan.
12. A intervalos regulares el equipo debe reflexionar sobre cómo ser más efectivo, entonces mejora y ajusta su comportamiento de acuerdo a sus conclusiones.

[7] http://www.agile-spain.com/principios_agiles

3.4 *Empowerment* y *Feedback*

La distinción entre las tareas relevantes y las que no agregan valor se consigue a través de la creación de contextos con alto nivel de *empowerment y feedback*.

El *empowerment* consiste en otorgar autonomía para tomar decisiones al equipo de desarrollo, y genera un clima de sinergia grupal que permite al grupo avanzar a pesar de las complicaciones y dificultades que ocurren habitualmente en los proyectos.

El *feedback* o retroalimentación constante y presente en varios niveles permite el desarrollo incremental y el crecimiento adaptativo de la programación, así también como una mejora constante en la forma de trabajo de los equipos, lo que permite detectar problemas y resolverlos antes de que afecten la calidad o el tiempo y costo del desarrollo. La retroalimentación ocurren a nivel producto, procesos y código.

En las metodologías ágiles el cliente evalúa periódicamente en cortos lapsos de tiempo el estado real del software que se está creando, lo que asegura que lo entregado al final del proyecto coincidirá con lo esperado. Esto se consigue a través de un desarrollo incremental: el producto puede probarse desde las primeras semanas del proyecto al menos en cuanto a su funcionalidad más básica, que luego va creciendo y mejorando.

El software es desarrollado en una unidad de tiempo denominada iteración o ciclo, la cual debe durar de una a cuatro semanas. Cada iteración del ciclo de vida incluye: planificación, análisis de requerimientos, diseño, codificación, revisión y documentación. Una iteración no debe agregar demasiada funcionalidad para justificar el lanzamiento del producto al mercado, pero la meta es tener un prototipo funcional al final de cada iteración. Al final de cada iteración el equipo vuelve a evaluar las prioridades del proyecto.

A nivel procesos se realizan frecuentes reuniones retrospectivas donde los integrantes de los equipos comentan y discuten en profundidad tanto sus aciertos (para poder repetirlos y convertirlos en hábitos), así también como el trabajo que no se realizó correctamente o no llevó al equipo a obtener los resultados esperados.

Adicionalmente los programadores suelen trabajar mucho en equipo y también por parejas, revisando juntos el código y resolviendo problemas en lugar de tratar de cubrirlos, lo que repercute en un producto de mejor calidad, mejor documentado, y simple de mantener.

3.5 Metodologías Tradicionales vs Metodologías Ágiles

La siguiente tabla resume las diferencias entre las metodologías tradicionales y las ágiles, no se refiere únicamente a las diferencias en el proceso, también muestra las diferencias en el contexto de los equipos y la organización.

Metodologías Ágiles	Metodologías Tradicionales
Basadas en heurísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparados para cambios durante el proyecto	Cierta resistencia a los cambios
Prácticas impuestas internamente (por el equipo de desarrollo)	Prácticas impuestas externamente
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas políticas/normas
No existe contrato tradicional o al menos es bastante flexible	Existe un contrato prefijado
El cliente es parte del equipo de desarrollo	Grupos grandes y posiblemente distribuidos
Pocos artefactos	Muchos artefactos
Pocos roles	Muchos roles
Menos énfasis en la arquitectura del software	La arquitectura del software es esencial y se expresa mediante modelos

Tabla 18. Diferencias entre metodologías ágiles y tradicionales

3.6 Metodologías Ágiles Actuales

Aunque los creadores e impulsores de las metodologías ágiles más populares han suscrito el manifiesto ágil y coinciden con los principios enunciados anteriormente, cada metodología tiene características propias y hace hincapié en algunos aspectos más específicos. A continuación se resumen algunas de las metodologías ágiles más conocidas y empleadas actualmente.

Programación Extrema (Extreme Programming, XP)[8]

El padre de XP es Kent Beck, se trata de una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa en la retroalimentación continua entre el cliente y el equipo de desarrollo, una comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP es especialmente adecuada para proyectos con requerimientos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

Los principios y prácticas son de sentido común pero llevadas al extremo, de ahí su nombre.

Scrum[9]

Desarrollada por Ken Schwaber, Jeff Sutherland y Mike Beedle. Define un marco para la administración de proyectos, que se ha utilizado con éxito durante los últimos 10 años. Está

[8] http://www.extremeprogramming.org/, http://xprogramming.com/index.php
[9] http://www.controlchaos.com/

especialmente indicada para proyectos con un rápido cambio de requerimientos. Sus principales características se pueden resumir en dos: El desarrollo de software se realiza mediante iteraciones, denominadas *sprints*, con una duración de 30 días. El resultado de cada sprint es un incremento ejecutable que se muestra al cliente. La segunda característica importante son las reuniones a lo largo proyecto, entre ellas destaca la reunión diaria de 15 minutos del equipo de desarrollo para coordinación e integración.

***Crystal Methodologies*[10]**

Se trata de un conjunto de metodologías para el desarrollo de software caracterizadas por estar centradas en las personas que componen el equipo y la reducción al máximo del número de artefactos producidos. Han sido desarrolladas por Alistair Cockburn. El desarrollo de software se considera un juego cooperativo de invención y comunicación, limitado por los recursos a utilizar. El equipo de desarrollo es un factor clave, por lo que se deben invertir esfuerzos en mejorar sus habilidades y destrezas, así como tener políticas de trabajo en equipo definidas. Estas políticas dependerán del tamaño del equipo, estableciéndose una clasificación por colores, por ejemplo Crystal Clear (3 a 8 miembros) y Crystal Orange (25 a 50 miembros).

***Dynamic Systems Development Method (DSDM)* [11]**

Define el marco para desarrollar un proceso de producción de software. Nace en 1994 con el objetivo de crear una metodología RAD (*Rapid Application Development*) unificada. Sus principales características son: es un proceso iterativo e incremental y el equipo de desarrollo y el usuario trabajan juntos. Propone cinco fases: estudio viabilidad, estudio del negocio, modelado funcional, diseño y construcción, y finalmente implementación. Las tres últimas son iterativas, además de existir realimentación a todas las fases.

***Adaptive Software Development (ASD)* [12]**

Su impulsor es Jim Highsmith. Sus principales características son: iterativo, orientado a los componentes de software más que a las tareas y tolerancia a los cambios. El ciclo de vida que propone tiene tres fases esenciales: especulación, colaboración y aprendizaje. En la primera de ellas se inicia el proyecto y se planifican las características del software; en la segunda desarrollan las características y finalmente en la tercera se revisa su calidad, y se entrega al cliente. La revisión de los componentes sirve para aprender de los errores y volver a iniciar el ciclo de desarrollo.

***Lean Development (LD) y Lean Software Development (LSD)* [13]**

La filosofía de LD es definida en un principio por Bob Charette a partir de su experiencia en proyectos con la industria japonesa del automóvil en los años 80 y utilizada en numerosos proyectos de telecomunicaciones en Europa. En LD, los cambios se consideran riesgos, pero si se

[10] http://alistair.cockburn.us/Crystal+light+methods
[11] http://www.dsdm.org/
[12] http://www.adaptivesd.com/
[13] http://www.poppendieck.com/

manejan adecuadamente se pueden convertir en oportunidades que mejoren la productividad del cliente. Su principal característica es la introducción de un mecanismo especial para implementar los cambios. LD ha evolucionado como Lean Software Development (LSD), su figura de referencia es Mary Poppendieck.

Agile Unified Process (AUP)[14]

El Proceso Unificado Ágil (AUP) es un enfoque al desarrollo de software que aparece por primera vez en septiembre del 2005 desarrollada por Scott W. Ambler. Esta es una propuesta basada en las prácticas comunes de XP y algunos procesos "ágiles" conservando la formalidad de RUP *(Rational Unified Proces)* de IBM. El ciclo de vida de AUP es "serial en lo grande" e "iterativo en lo pequeño", liberando entregables incrementables en el tiempo.

3.7 Comparación Entre Metodologías

En [44] se hace una comparación de las distintas aproximaciones ágiles en base a tres parámetros: vista del sistema como algo cambiante, colaboración entre los miembros del equipo y características más específicas de la propia metodología como son simplicidad, excelencia técnica, resultados, adaptabilidad, etc. También incorpora como referencia de modelo no ágil al *Capability Madurity Model (CMM)*.

	CMM	*ASD*	*Crystal*	*DSDM*	*FDD*	*LD*	*Scrum*	*XP*
Sistema como algo cambiante	1	5	4	3	3	4	5	5
Colaboración	2	5	5	4	4	4	5	5
Características en la Metodología (CM)								
-Resultados	2	5	5	4	4	4	5	5
-Simplicidad	1	4	4	3	5	3	5	5
-Adaptabilidad	2	5	5	3	3	4	4	3
-Excelencia técnica	4	3	3	4	4	4	3	4
-Prácticas de colaboración	2	5	5	4	3	3	4	5
Media CM	2.2	4.4	4.4	3.6	3.8	3.6	4.2	4.4
Media Total	1.7	4.8	4.5	3.6	3.6	3.9	4.7	4.8

Tabla 19. Ranking de "agilidad" (Los valores más altos representan una mayor agilidad) [44]

Como se observa en la Tabla 19, todas las metodologías ágiles tienen una significativa diferencia del índice de agilidad respecto a CMM y entre ellas destacan ASD, Scrum y XP como las más ágiles.

[14] http://www.ambysoft.com/unifiedprocess/agileUP.html

Capítulo 4. Guía para la Evolución de las Arquitecturas de Software en Ambientes Ágiles

4.1 Consideraciones Previas al Inicio del Proyecto

4.1.1 Consideraciones del Equipo y del Inicio del Desarrollo Arquitectónico

Como se ha mencionado hasta el momento una arquitectura de software es importante para el desarrollo de software sin importar la metodología o proceso que se utilice para crearlo. La mayoría de los sistemas exitosos tienen una arquitectura bien diseñada y detallada, que les provee de un plan de trabajo que contiene qué y cómo se deberá construir el sistema para el equipo de desarrollo. Los sistemas con arquitecturas pobres o no existentes tienden a ser difíciles de construir y mantener, usan estrategias inconsistentes de integración y es difícil hacer que los diferentes componentes trabajen juntos óptimamente. En general los sistemas con arquitecturas pobres tienden a tener un número mucho mayor de fallos que aquellos que tienen un buen diseño de alto nivel.

El primer paso es definir cómo, cuándo y quién(es) desarrollara(n) el diseño arquitectónico del sistema. De acuerdo con [4] existen tres enfoques básicos para desarrollar la arquitectura de un sistema en Scrum, aunque puede extrapolarse para cualquier metodología ágil:

4.1.1.1 Por Adelantado *(Up Front)*

Una arquitectura por adelantado sucede cuando la arquitectura para un sistema está total o significativamente desarrollada antes de que el equipo empiece a trabajar. El "arquitecto" trabaja con el "dueño del producto" y el "scrum master" para identificar y escribir los requerimientos funcionales que deberá soportar la arquitectura, además deberá tener el enfoque arquitectónico definido y documentado en suficiente detalle para transferirlo al equipo para empezar el diseño y construcción. Algunas cuestiones arquitectónicas como las clases y el diseño de la base de datos pueden ser diferidos al equipo, pero todos los demás aspectos deben ser conocidos y entendidos antes de que el equipo empiece a trabajar.

Esta opción es la más adecuada cuando existen múltiples equipos trabajando para el mismo producto final, y/o cuando hay limitaciones de recursos que impiden que un arquitecto tenga una presencia permanente en un equipo. También cuando los requerimientos funcionales y no funcionales están suficientemente claros y definidos. Teniendo una arquitectura definida en la que todos los equipos inmediatamente trabajen se reduce el cambio de soluciones múltiples e incompatibles para el mismo problema.

4.1.1.2 Arquitecto Dentro del Equipo *(On-Team Architect)*

Este enfoque empieza con el "dueño del producto" y el "*scrum master*" trabajando con el arquitecto para describir los requerimientos funcionales, pero el trabajo arquitectónico se hace dentro de una iteración junto con el resto del equipo. La arquitectura deberá estar lo suficientemente estructurada para que el primer ciclo no sea una pérdida de tiempo para el resto

del equipo. El arquitecto trabaja con un enfoque "justo a tiempo" para conocer las necesidades de los próximos requerimientos funcionales.

Este enfoque es muy usado cuando se tiene un arquitecto disponible y la solución para la aplicación es bastante compleja y existe un gran número de "desconocidos" relativos a las habilidades del equipo y de las tecnologías que serán usadas. El arquitecto es un "jugador especializado" en el equipo interpuesto para navegar sobre los "desconocidos" y las complejidades. El arquitecto frecuentemente trabajará con el equipo para realizar y experimentar soluciones potenciales.

4.1.1.3 Arquitectura del Equipo *(Team Architecture)*

Es similar al enfoque anterior, excepto que las decisiones y las actividades para crear el diseño de alto nivel son desarrolladas y pertenecen al equipo entero. Puede ser necesario tener un arquitecto fuera del equipo para facilitar las discusiones en el equipo. Por delante del equipo de desarrollo, el "dueño del producto" y el "*scrum master*" trabajan con el arquitecto para desarrollar las historias de usuario técnicas, pero las soluciones arquitectónicas son tomadas por el equipo.

Este es el modelo de elección para la mayoría de los equipos *Scrum* donde las habilidades son consideradas suficientes en el equipo, en general esta opción es la que más se apega a la filosofía ágil. El arquitecto debe ser un facilitador en las discusiones con el equipo en cómo enfocar la solución al rompecabezas arquitectónico, pero es el equipo entero que idea y construye la arquitectura. Este enfoque trabaja mejor en equipos maduros que han demostrado un buen historial en innovación y en creatividad técnica.

Esta técnica funciona también en situaciones multi-equipo, donde el "*Scrum de Scrums*" es un concepto empleado para llevar a todos los equipos una comprensión de cómo construir la aplicación. Frecuentemente, algunos equipos pueden no tener las habilidades o la experiencia para manejar las responsabilidades de la arquitectura completa, un sólo equipo puede tomar la iniciativa, juntando a los equipos restantes con ellos.

4.1.1.4 Arquitecto Ágil

Cuando se planea un proyecto se debe tener especial consideración a los requerimientos técnicos del sistema. Entendiendo las complejidades potenciales, las habilidades del equipo y las limitaciones de recursos, los equipos pueden combinar estos tres enfoques para diseñar y construir una aplicación de calidad que aporte un valor inmediato a los usuarios.

De acuerdo con [45], se aconseja que al tener equipos ágiles pequeños de 5 personas o menos, cada uno de los integrantes del equipo sea responsable de la arquitectura, de manera similar a la "arquitectura del equipo" descrita anteriormente. Esto incrementa el entendimiento y aceptación de la arquitectura por parte de todo el equipo porque trabajan juntos en desarrollarla. También incrementa la posibilidad de que los desarrolladores estén dispuestos a cambiar aspectos de la arquitectura cuando la arquitectura resulta insuficiente de una manera más libre. Respecto a esta práctica existen dos problemas básicos:

1. Algunas veces las personas no se ponen de acuerdo. Esta estrategia falla dramáticamente cuando el equipo no llega a un acuerdo, es entonces que se necesita a alguien en el "rol del dueño de la arquitectura", en caso de que no haya un arquitecto.

2. No se escala. Cuando el equipo es grande o geográficamente distribuido, es conveniente organizar al equipo en subequipos de acuerdo con ASM *(Agile Scaling Model)*. La arquitectura en escala requiere de un equipo de trabajo coordinado.

Al tener un sistema razonablemente complejo es necesario invertir tiempo en la arquitectura, se recomienda hacer la concepción de la arquitectura por adelantado no por completo pero si lo suficiente para empezar en la dirección correcta, de manera que la arquitectura pueda evolucionar partiendo de ésta. En estas circunstancias es necesario también el "rol del arquitecto", generalmente es la persona con mayor experiencia técnica en el equipo, esta persona será la responsable de facilitar el modelado arquitectónico y su evolución.

El "arquitecto ágil" es diferente "arquitecto tradicional". El "arquitecto tradicional" generalmente es el creador primario de la arquitectura y es una de las pocas personas que trabajan en ella. Su trabajo es desarrollar la arquitectura y "presentarla" al equipo de desarrollo. El "arquitecto ágil" trabaja colaborativamente con el equipo para desarrollar y evolucionar la arquitectura. A pesar de ser la persona con la autoridad final de tomar las decisiones cuando se trata de la arquitectura, estas decisiones deben ser realizadas de una manera colaborativa. Los "arquitectos ágiles" efectivos son aquellos desarrolladores que tienen experiencia en las tecnologías con las que trabaja la organización y que tienen la habilidad de trabajar en "*spikes*" arquitectónicos (Un *spike* consiste en una prueba rápida acerca de un concepto con duración máxima de una semana) para explorar nuevas estrategias. Deben tener un buen conocimiento del dominio del negocio y las habilidades necesarias para comunicar la arquitectura a los desarrolladores y a otros interesados.

4.1.2 Premisas y Restricciones de la Guía Propuesta

Los hilos conductores de esta guía son los valores y principios ágiles (Ver sección 3.3 y 3.4) de los que se destacan la "Aceptación de requerimientos cambiantes" y "Entrega frecuente de software que funciona", por lo que puede utilizarse en proyectos en los que no se tienen bien definidos los requerimientos, permitiendo descubrirlos durante el transcurso del desarrollo de una aplicación.

Lo anterior es posible ya que se propone que se realicen entregas continuas de versiones del producto de manera que el cliente/usuario pueda decidir si se van cumpliendo las necesidades propuestas en un inicio e inclusive descubriendo necesidades que no se tenían previstas y que pueden ser nuevas áreas de oportunidad.

Se debe resaltar que para la aplicación de esta guía se debe contar con el apoyo continuo del cliente/usuario y el compromiso por parte del equipo de desarrollo puesto que es gracias a estos que se va a permitir la evolución de la aplicación de una manera más ágil y satisfactoria.

Dentro de las restricciones a considerar es que se elaboró esta guía teniendo en mente un equipo auto-organizado y centralizado para facilitar la ejecución de las tareas propuestas. Se diseño

teniendo en cuenta un proyecto no tan complejo en lo que respecta a las reglas del negocio y la organización del equipo de desarrollo.

También se debe considerar que se propone la aplicación de la guía en proyectos nuevos, de manera que se permita el desarrollo de la arquitectura desde el inicio del proyecto. Aunque si se considera la reutilización de componentes para facilitar el diseño e implementación de la arquitectura.

Dentro de la guía no se consideró algún mecanismo de control de cambios y solución de compensaciones (*tradeoffs*) formal, estas actividades recaen en las decisiones tomadas por el equipo de desarrollo de acuerdo a su experiencia.

Tampoco se tienen definidas las técnicas y herramientas para llevar a cabo la documentación de los productos propuestos, para apoyar en esta tarea se puede hacer uso de la sección 2.6.

4.2 Guía de Actividades para el Ciclo de Desarrollo de la Arquitecturas de Software

En general la actividad más compleja durante el desarrollo de una aplicación es la transformación de los requerimientos de la especificación en la arquitectura del sistema. Las actividades posteriores aunque también son retadoras, por ejemplo la implementación, son actividades que son mejor entendidas que cuentan con un soporte metodológico y tecnológico disponible para los ingenieros de software.

Para el año 2000 el proceso del diseño arquitectónico era considerablemente menos formalizado y con poco soporte metodológico disponible, se le consideraba como una actividad artesanal [40]. Dentro de la industria de construcción de software bajo ambientes de desarrollo ágil se considera al diseño de la arquitectura de software como una actividad basada en la experiencia más que una actividad de ingeniería bien definida.

Aunque desde el inicio del desarrollo de software los sistemas han contado con arquitecturas de software, recientemente se ha reconocido la importancia de especificarlas, analizarlas y diseñarlas, debido al hecho de que los atributos de calidad tienen una amplia influencia en la arquitectura del sistema. Ejemplos de esto se encuentran en las técnicas propuestas por Bosch (Ver sección 2.7.1) y los métodos propuestos por el SEI (Ver sección 2.7.2).

El diseño arquitectónico es una actividad de diseño con múltiples objetivos donde el ingeniero de software tiene que balancear los requerimientos durante el proceso de diseño.

El diseño de una arquitectura de software no es una actividad independiente, es más bien un paso en el proceso de desarrollo y evolución de productos de software. Como se mencionó en el capítulo 2 en la sección 2.1.5 las actividades relacionadas con el desarrollo de la arquitectura de software generalmente forman parte de las actividades definidas dentro del proceso de desarrollo. Partiendo de lo anterior se propone adaptar en esta guía el ciclo de vida de un producto en cuatro etapas iterativas y sus respectivas actividades. Todo lo anterior sin olvidar que

el desarrollo arquitectónico está dividido en las etapas de: requerimientos, diseño, documentación y evaluación, y que precede a la etapa de construcción/implementación del sistema (Ver sección 2.1.5).

La Figura 30 y Figura 31 tienen el mismo contenido distribuido de diferente forma para facilitar el entendimiento de la guía propuesta, ambas muestran un diagrama que contiene de manera general las etapas propuestas y su explicación es la siguiente:

1. La etapa de "Versiones del Producto" que se muestra en el rectángulo exterior de la Figura 30 se ocupa de la evolución de los requerimientos del producto durante su maduración. En las metodologías ágiles al momento de realizar las entregas iterativas de versiones del producto a los clientes se pueden encontrar nuevos requerimientos o modificarlos según vayan surgiendo. Esta etapa muestra la evolución del producto a través de versiones del producto liberado al cliente/usuario hasta que ya no haya más requerimientos.
2. La siguiente etapa denominada "Desarrollo Iterativo del Producto" representada en el siguiente rectángulo puede suceder cuando se construye un sistema donde los requerimientos son descubiertos o retomados durante el proceso iterativo de desarrollo. Dentro de las técnicas utilizadas en las metodologías ágiles, la organización desarrolla varios prototipos o versiones del producto en desarrollo como parte de los requerimientos antes de finalizar el desarrollo del producto y entrar en la actividad de despliegue o puesta en operación.

   Esta etapa también entra en acción cuando los involucrados (clientes, usuarios, ingenieros, etc.) definen o cambian nuevos requerimientos del software, por ejemplo, cuando los ingenieros identifican nuevos avances tecnológicos que desean incluir en el proyecto.
3. La etapa de "Selección de requerimientos" que se muestra en el siguiente rectángulo, se realiza de acuerdo con la prioridad que se ha definido con el cliente/usuario, se debe iniciar por seleccionar un subconjunto de aquellos requerimientos que tengan mayor prioridad para entrar a la siguiente etapa. Basados en este subconjunto se va a desarrollar una versión inicial de la arquitectura y posteriormente al repetirse esta etapa se utilizará un subconjunto extendido de los requerimientos y restricciones hasta que todos (o los factibles) han sido incluidos en el diseño arquitectónico.
4. La etapa de "Diseño de la arquitectura" es donde se diseña, evalúa y transforma la arquitectura de software de acuerdo a los atributos de calidad especificados (Ver Capítulo 2, Sección 2.2.1). Esta etapa es la que recibe el conjunto (o subconjunto) de requerimientos con los que se basará para desarrollar la arquitectura. Se trata de una etapa de diseño objetivo y racional, que hace balance y optimiza los atributos de calidad, sin olvidar los requerimientos funcionales y restricciones. Esta etapa evalúa iterativamente el grado en que la arquitectura soporta cada atributo de calidad y mejora la arquitectura a través de la transformación hasta que se cumplen la mayoría de los atributos de calidad. También cuenta con una actividad de diseño arquitectónico basado en la funcionalidad, es aquí donde se diseñan los componentes que se encargarán de los requerimientos funcionales, restricciones, reglas del negocio, etc. Lo que trata de hacerse es de

complementar los métodos tradicionales de diseño integrando los atributos de calidad y no centrándose únicamente en las funcionalidades.

Para mostrar las etapas del ciclo de vida de una arquitectura de software (Ver sección 2.1.5) acopladas al ciclo de desarrollo se utiliza la siguiente notación:

- En color naranja se encuentran las actividades y productos relacionados con la etapa de requerimientos.
- En color verde las actividades y productos relacionados con la etapa de diseño y evaluación.
- En general los productos que se muestran están relacionados con la etapa de documentación, la cual se realiza de manera paralela a las actividades que generan dichos productos.
- Las actividades (6 a 9) y productos en azul son aquellas que utilizan la arquitectura definida y forman parte del ciclo de desarrollo del producto.
- Se considera un ciclo cuando se entrega una "Versión del Producto Liberado".
- Se considera una iteración cuando se genera una "Versión del Producto en Desarrollo".

En las siguientes secciones se explicarán las actividades de cada etapa.

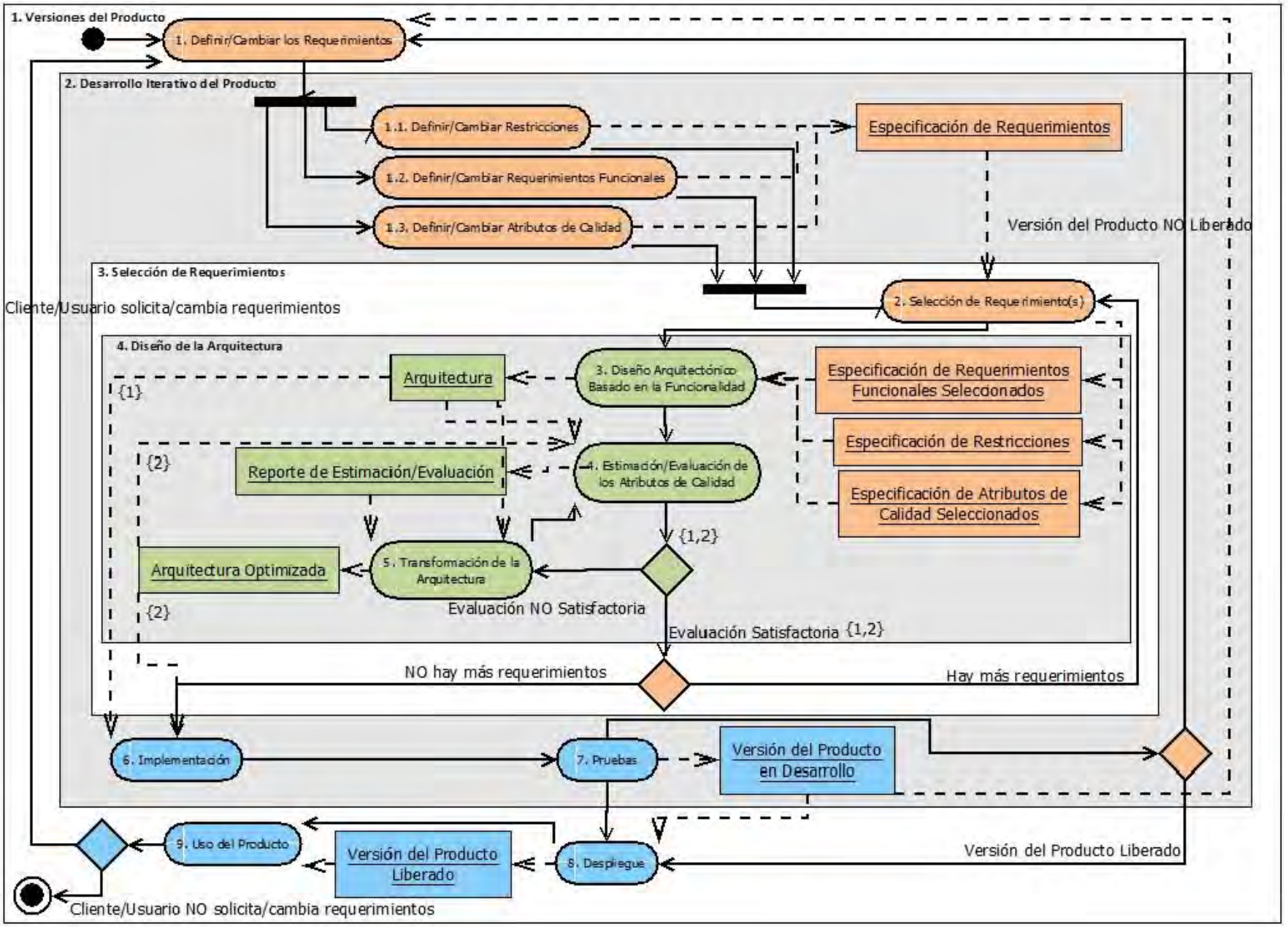

Figura 30. Diagrama de Actividades Propuestas por Etapas del Ciclo de Vida de una Arquitectura de Software para su Integración en el Proceso de Desarrollo Ágil (Versión 1)

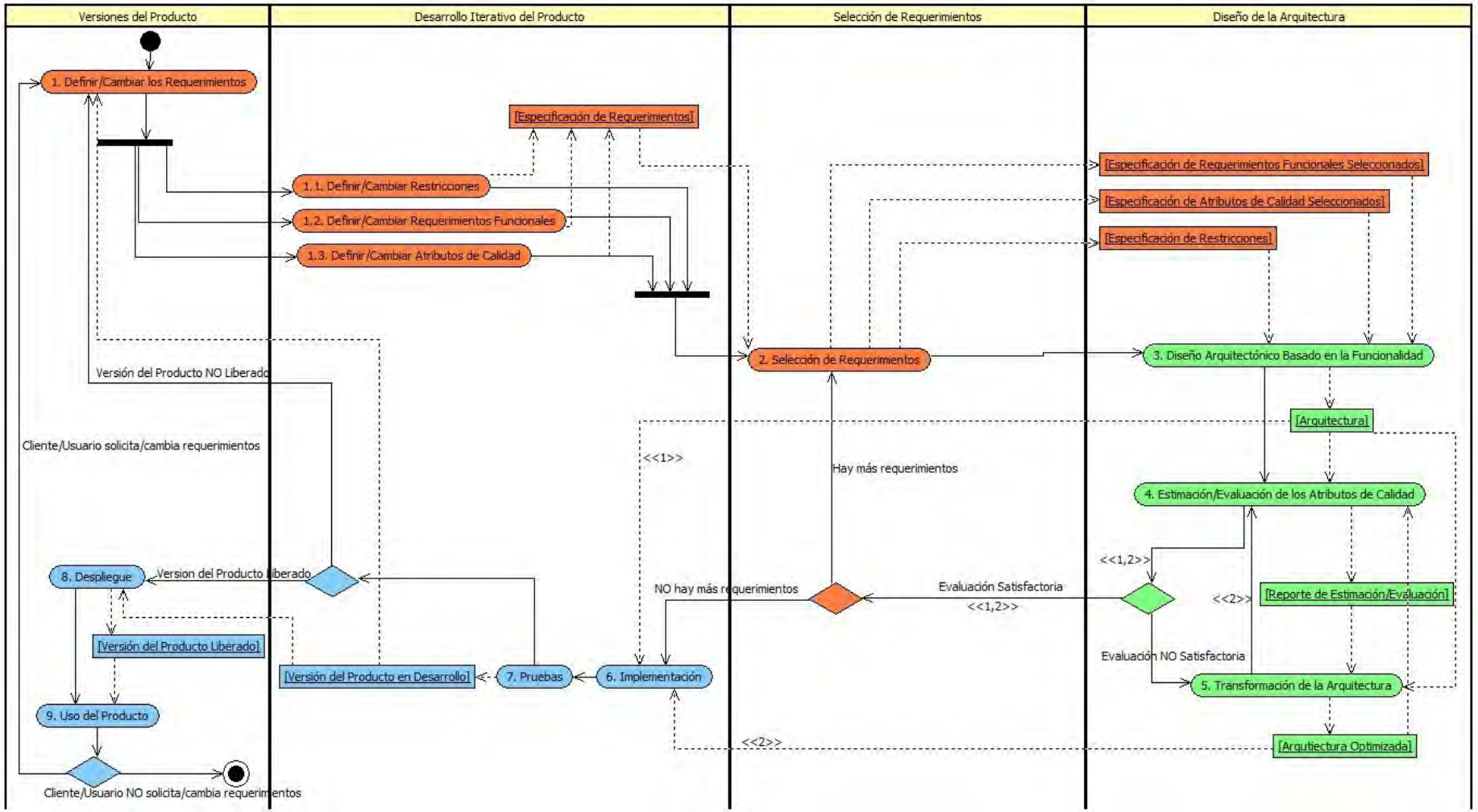

Figura 31. Diagrama de Actividades Propuestas por Etapas del Ciclo de Vida de una Arquitectura de Software para su Integración en el Proceso de Desarrollo Ágil (Versión 2)

4.2.1 Etapa de Requerimientos

La etapa de requerimientos se enfoca en la captura, documentación y priorización de requerimientos que influencian la arquitectura. Los atributos de calidad juegan un papel preponderante dentro de estos requerimientos (Ver Capítulo 2, Sección 2.2.1), así que esta etapa hace énfasis en ellos. Otros requerimientos, sin embargo, son también relevantes para la arquitectura, estos son los requerimientos funcionales primarios y las restricciones.

Las actividades de esta etapa son las siguientes:

1. **Definir/Cambiar los Requerimientos.** Esta actividad depende directamente de los interesados puesto que estos van a ser los que van a decidir los requerimientos que requieren o son necesarios para la aplicación. De los requerimientos definidos/cambiados en esta actividad se deben realizar las siguientes tres actividades:

   1.1. Definir/Cambiar Restricciones.
   1.2. Definir/Cambiar Requerimientos Funcionales.
   1.3. Definir/Cambiar Atributos de Calidad.

   De estas actividades se obtiene la "Especificación de Requerimientos".

2. **Selección de Requerimientos.** Partiendo de la "Especificación de Requerimientos" se seleccionan aquellos requerimientos que tienen mayor prioridad de acuerdo a las necesidades del negocio o aquellos que debido a experiencias previas los desarrolladores consideren que tienen un mayor impacto arquitectónico. De esta actividad se obtienen la "Especificación de Requerimientos Funcionales Seleccionados" y la "Especificación de Atributos de Calidad Seleccionados" de acuerdo con las decisiones realizadas.

Esta etapa está relacionada muy estrechamente con la "Ingeniería de Requerimientos". En [46] se proponen algunas técnicas de obtención de requerimientos que se adecúan a las necesidades de los ambientes de desarrollo ágiles.

4.2.2 Etapa de Diseño y Evaluación

La etapa de diseño es la etapa central en relación con la arquitectura y probablemente la más compleja. Durante esta etapa se definen las estructuras que componen la arquitectura. La creación de estas estructuras se hace en base a patrones de diseño, tácticas de diseño y elecciones tecnológicas (Ver Capítulo 2, Secciones 2.3 y 2.4). El diseño que se realiza debe buscar ante todo satisfacer los requerimientos que influencian a la arquitectura, y no simplemente incorporar diversas tecnologías por que están "de moda".

Dado que la arquitectura de software juega un papel crítico en el desarrollo, es conveniente evaluar el diseño una vez que éste ha sido documentado con el fin de identificar posibles problemas y riesgos. La ventaja de evaluar el diseño es que es una actividad que se puede realizar de manera temprana (aún antes de codificar), y que el costo de corrección de los

defectos identificados a través de la evaluación es mucho menor al costo que tendría el corregir estos defectos una vez que el sistema ha sido construido.

Antes de iniciar con las actividades de diseño para optimizar los recursos se recomienda aplicar una estrategia de reusabilidad, cuyos beneficios se explicaron en la sección 2.3.5. Para lograrlo antes de realizar el diseño se deben identificar todas aquellas partes que de acuerdo a la experiencia del equipo son reutilizables en sus dos formas: el desarrollo de software con reutilización y el desarrollo de software para la reutilización. La idea es que al desarrollar un producto, se especifiquen y diseñen las partes reutilizables, colocarlas dentro de una "Biblioteca de Reutilización", y usarlas durante el desarrollo del producto actual y subsecuentes.

Partiendo de la "Especificación de Requerimientos Funcionales Seleccionados" y la "Especificación de Atributos de Calidad Seleccionados" obtenidos en la actividad **(2)** de la etapa de requerimientos se realiza el diseño arquitectónico de acuerdo a las siguientes actividades:

3. **Diseño Arquitectónico Basado en la Funcionalidad.** Partiendo de la "Especificación de Requerimientos Funcionales Seleccionados" se realiza el diseño de alto nivel del sistema. Dentro de un ambiente de desarrollo ágil se comienza esta actividad realizando un diseño sin tantos detalles como en las metodologías tradicionales, ya que conforme se vaya avanzando se irán identificando a más detalle los requerimientos funcionales teniendo presentes la "Especificación de Atributos de Calidad" y "Especificación de Requerimientos" que se vayan generando paralelamente.

   Generalmente se comienza identificando las entidades arquitectónicas (objetos, etc.) junto con sus propiedades, para lograrlo se apoya generalmente de los métodos de análisis del dominio (diagramas de actividades, casos de uso, diagramas entidad-relación, diagramas de transición de estados, diagramas de clases, entre otros), sin olvidar que las arquitecturas generalmente cubren múltiples dominios. Una vez que se tienen definidas las entidades arquitectónicas se definen las interacciones entre ellos a mayor detalle, sin olvidarse de los atributos de calidad y restricciones. Se puede usar la sección 2.4 del capítulo 2 de este trabajo como referencia en esta actividad, en donde se exponen los estilos arquitectónicos junto con su descripción de forma resumida, los dominios aplicables, beneficios y limitaciones para cada estilo arquitectónico.

   La arquitectura basada en requerimientos funcionales tiende a ser muy general, por lo que puede ser reutilizada en sistemas con el mismo dominio pero con diferentes atributos de calidad.

4. **Estimación/Evaluación de los Atributos de Calidad**. En esta actividad tiene dos escenarios:

   El primer escenario sucede cuando se está realizando la primera iteración, por lo que se debe proponer un estilo arquitectónico de acuerdo a los atributos de calidad seleccionados, en este trabajo de tesis se propone utilizar la Tabla 7 de la sección 2.5 para proponer la "Arquitectura".

El segundo escenario sucede en las siguientes iteraciones, en estos casos se debe hacer un balance de los atributos de calidad que se habían seleccionado anteriormente y los nuevos, para lograrlo se recomienda el uso de la Tabla 7 (Ver sección 2.5) o alguna de las técnicas expuestas en la sección 2.7.1: Técnica de Evaluación Basada es Simulación, Técnica de Evaluación Basada en Escenarios y la Técnica de Evaluación Basada en Experiencia. Se recomienda el uso de estas técnicas puesto que se considera que son las más apegadas a los principios ágiles. Es en este escenario se produce el "Reporte de Estimación/Evaluación" en el que se explican los resultados obtenidos de ésta actividad y permitir la transformación de la arquitectura en la actividad **(5)**.

Las actividades **(3)** y **(4)** están estrechamente relacionadas ya que las decisiones tomadas en cualquiera de éstas afectarán a la otra. Se debe buscar un balance entre los requerimientos funcionales y requerimientos no funcionales, enfocándose en los atributos de calidad.

En caso de que los requerimientos funcionales y atributos de calidad no se acoplen se debe negociar con los interesados para hacer las compensaciones necesarias (Ver sección 2.2.1) y en caso extremo de que la arquitectura definida ya no sea la adecuada se aplica la técnica de refactorización propuesta por XP (Modificación de la estructura interna del código sin alterar su comportamiento externo).

5. **Transformación de la Arquitectura**. Una vez que los atributos de calidad de la arquitectura han sido evaluados, los resultados obtenidos en el reporte de estimación/evaluación son comparados con la especificación de requerimientos. Si uno o más de los atributos de calidad no son satisfechos, la arquitectura tiene que ser modificada para cubrir dichos requerimientos. En este punto el ingeniero de software analiza la arquitectura y decide porque la propiedad de la arquitectura es inhibida. Generalmente, la evaluación por si sola muestra indicios de que partes o decisiones tomadas son las causantes del problema. Una vez que se encuentran los problemas, se procede a transformar la arquitectura ya sea negociando las compensaciones correspondientes con los clientes/usuarios o refactorizando la arquitectura utilizando algún patrón arquitectónico o de diseño (Ver secciones 2.3 y 2.4), produciendo una "Arquitectura Optimizada".

La ejecución de estas actividades van diseñando, detallando y refinando la "Arquitectura".

Una práctica útil al finalizar las iteraciones es el uso de las "Auditorías Arquitectónicas" explicadas en la sección 2.7.3, en caso de que la arquitectura haya sufrido modificaciones sustanciales o se quiera evaluar la arquitectura, la organización y el proceso que se esté utilizando.

4.2.3 Etapa de Documentación

Una vez creado el diseño de la arquitectura, es necesario poder comunicarlo a otros involucrados dentro del desarrollo. La comunicación exitosa del diseño muchas veces depende de que dicho diseño sea documentado de forma apropiada. La documentación de una arquitectura involucra la representación de varias de sus estructuras que son representadas a través de distintas vistas.

Una vista generalmente contiene un diagrama, además de información adicional, que apoya en la comprensión de dicho diagrama.

En general para los productos relacionados con la arquitectura, generados durante el ciclo de desarrollo, es necesario realizar su documentación y actualizarla conforme se vayan tomando decisiones que afecten a la arquitectura. En otras palabras, se propone que la documentación de los productos se haga de manera paralela a las actividades que los generan. Los productos a generar son los siguientes:

1. La **Especificación de Requerimientos** que estará integrada por los Requerimientos Funcionales, Restricciones del Sistema y Atributos de Calidad, todos estos serán la "Lista de Deseos" de los involucrados.
2. **Especificación de Requerimientos Funcionales Seleccionados**. Este producto puede estar integrado dentro de la Especificación de Requerimientos inicial, la diferencia es que se deben especificar en un apartado los requerimientos seleccionados viables de acuerdo a la actividad de "Selección de Requerimientos".
3. **Especificación de Atributos de Calidad Seleccionados**. De la misma manera que el producto anterior, dentro de la Especificación de Requerimientos inicial se deben especificar los atributos de calidad que se van seleccionando y que son viables para realizar el diseño arquitectónico.
4. **Especificación de Restricciones.** También es un apartado de la Especificación de Requerimientos donde se encuentran aquellos requerimientos no funcionales que no son atributos de calidad y las restricciones solicitadas por los interesados.
5. **Arquitectura/Arquitectura optimizada.** En este producto se deben explicar cuáles son los componentes arquitectónicos y sus interrelaciones, de manera breve se debe explicar las razones por las que se definieron de esa manera y en caso de ser necesario explicar aquellas decisiones tomadas que afecten a la arquitectura durante el proceso de desarrollo. Las Figuras 30 y 31 muestran la evolución de la "Arquitectura" en dos escenarios. El primer escenario sucede cuando la evaluación es satisfactoria es entonces que se pasa directamente a la actividad de implementación. El segundo escenario sucede cuando la "Arquitectura" tiene una evaluación no satisfactoria y pasa a la actividad de transformación produciendo una "Arquitectura optimizada".
6. **Reporte de Estimación/ Evaluación.** Es un documento donde se explican los resultados de la Estimación/Evaluación de los Atributos de Calidad y nos permite documentar la evolución de los atributos de calidad que afectan al diseño arquitectónico. Este reporte permite conocer la capacidad de la arquitectura de soportar los atributos de calidad deseados.

Para elaborar estos productos se pueden utilizar las "Plantillas de Desarrollo de Software" de "Especificación de Requerimientos" (Ver Apéndice A. Plantilla de Análisis) y "Diseño" (Ver Apéndice B. Plantilla de Diseño) propuestas en la tesis [47], su adaptación dependerá de las necesidades de comunicación del proyecto.

Para documentar la arquitectura se recomienda utilizar alguna de las vistas propuestas en la sección 2.6.1, dependiendo de la perspectiva que se quiera utilizar de acuerdo a las necesidades y complejidad del proyecto que se esté desarrollando.

Como notación para realizar la de documentación arquitectónica se propone utilizar los diagramas de: paquetes, componentes o distribución de UML sobre los lenguajes de descripción arquitectónica puesto que UML se considera como un estándar común de fácil adopción para todos los integrantes del equipo de desarrollo. Dependiendo de las necesidades del proyecto se decide cuales son los diagramas que serán implementados.

La idea de documentar las decisiones que se toman durante las actividades de diseño actualmente está cobrando cada vez más importancia. Esta documentación es importante pues permite que se evalúe el diseño además de que permite comprender la toma de decisiones al momento de realizar el mantenimiento del sistema.

4.2.4 Etapa de Construcción/Implementación del Sistema

Una vez que se tiene una "Arquitectura"/"Arquitectura Optimizada" y ya no hay más requerimientos que considerar en el diseño de la arquitectura para la versión del producto en curso se procede a la actividad **(6)** de implementación del sistema basándose en la arquitectura diseñada y posteriormente a la actividad **(7)** de pruebas para generar una "Versión del Producto en Desarrollo" que es precisamente el producto/prototipo que se maneja de manera interna dentro del equipo de desarrollo. Forman parte de la etapa de Desarrollo Iterativo del Producto porque son las actividades que generan al producto conforme se vaya construyendo la aplicación o sistema, permitiendo al equipo de desarrollo evaluar si se han cumplido o no con los requerimientos y es posible liberarlo para el uso del cliente/usuario. También es en este punto que se permite evaluar si es necesario definir/cambiar requerimientos que puedan mejorar o afectar a la aplicación desarrollada antes de hacer entrega de la versión del producto.

Una vez que se libera el producto internamente se pasa a la actividad **(8)** de despliegue o puesta en marcha en el ambiente de producción o ambiente del cliente para producir una "Versión del Producto Liberado" y pasar a la actividad **(9)** de uso del producto. Transcurrido un lapso de tiempo definido entre los involucrados se evalúa si hay nuevos requerimientos o es necesario cambiarlos. De haber solicitudes/cambios en los requerimientos se pasa de nuevo a la actividad **(1)** para iniciar un nuevo ciclo, de lo contrario se da por terminado el proyecto.

4.2.5 Resumen de las Actividades de la Guía

Para concluir se resumen las actividades propuestas en la siguiente tabla:

Actividades/ Subactividades	Etapa	Productos Requeridos	Productos Generados	Actividades/ Subactividades subsecuentes
1. Definir/Cambiar los Requerimientos	Versiones del Producto	Los interesados definen los requerimientos nuevos o en su caso los cambian		Se divide en las subactividades **1.1, 1.2 y 1.3**
1.1 Definir/Cambiar Restricciones **1.2 Definir/Cambiar Requerimientos Funcionales** **1.3 Definir/Cambiar Atributos de Calidad**	Desarrollo Iterativo del Producto		Especificación de Requerimientos	**2**
2. Selección de Requerimientos	Selección de Requerimientos	Especificación de Requerimientos	• Especificación de Requerimientos Funcionales Seleccionados • Especificación de Atributos de Calidad Seleccionados • Especificación de Restricciones	**3**
3. Diseño Arquitectónico Basado en la Funcionalidad	Diseño de la Arquitectura	• Especificación de Requerimientos Funcionales Seleccionados • Especificación de Atributos de Calidad Seleccionados • Especificación de Restricciones	Arquitectura	**4**
4. Estimación/Evaluación de los Atributos de Calidad	Diseño de la Arquitectura	Arquitectura	• Reporte de Estimación/Evaluación • Arquitectura (Evaluada satisfactoriamente)	• Si la evaluación NO fue satisfactoria a **5** • Si la evaluación fue satisfactoria y hay más requerimientos a **2**

Actividades/ Subactividades	Etapa	Productos Requeridos	Productos Generados	Actividades/ Subactividades subsecuentes
				➢ Si la evaluación fue satisfactoria y NO hay más requerimientos a **6**
5. Transformación de la Arquitectura	Diseño de la Arquitectura	➢ Arquitectura ➢ Reporte de Estimación/Evaluación	Arquitectura Optimizada	Una vez que la arquitectura es satisfactoria: ➢ Si hay más requerimientos a **2** ➢ Si NO hay más requerimientos a **6** Si la arquitectura sigue sin ser satisfactoria pasa a **4**
6. Implementación	Desarrollo Iterativo del Producto	➢ Arquitectura (Evaluada satisfactoriamente) ➢ Arquitectura Optimizada	Implementación del sistema basándose en la arquitectura diseñada	**7**
7. Pruebas	Desarrollo Iterativo del Producto	Implementación de la Arquitectura en el Producto	Versión del Producto en Desarrollo	➢ Si el producto es liberado pasa a **8** ➢ Si el producto no es liberado pasa a **1**
8. Despliegue	Versiones del Producto	Versión del Producto en Desarrollo	Versión del Producto Liberado	**9**
9. Uso del Producto	Versiones del Producto	Versión del Producto Liberado	Los interesados solicitan o cambian requerimientos o el producto se da por terminado al no haber más solicitudes o cambios de requerimientos	Si hay nuevos requerimientos o cambios a **2,** de lo contrario se da por terminado el producto

Tabla 20. Actividades Propuestas para el Establecimiento de una Arquitectura de Software para su Integración en Ambientes de Desarrollo Ágil

4.3 Acoplamiento de los Valores y Principios Ágiles en la Guía

De acuerdo con la filosofía ágil expuesta en el capítulo 3 en la sección 3.2 y 3.3, la guía propuesta enfatiza los valores y principios ágiles de la siguiente manera:

A los individuos e interacciones del equipo sobre los *procesos y herramientas.* Puesto que la gente es el principal factor de éxito de un proyecto software es importante involucrar al equipo durante el proceso de desarrollo arquitectónico con el objetivo de fortalecer la comunicación y crear un sentido de pertenencia arquitectónica para todo el equipo ya que son estos quienes la diseñan, desarrollan y evolucionan. Dentro de la guía se propone desde las consideraciones iniciales el trabajo en equipo principalmente dentro de las modalidades de "Arquitecto Dentro del Equipo" y "Arquitectura del Equipo".

La colaboración con el cliente sobre la *negociación de contratos* y **Responder ante los cambios** sobre el *seguimiento estricto de un plan*. Puesto que la interacción constante entre el cliente y el equipo de desarrollo es uno de los puntos clave dentro de las metodologías ágiles, se propone la colaboración entre ambos constantemente y la habilidad de responder a los cambios dentro de la guía en los tres procesos iterativos exteriores que se muestran en la Figura 30, estos son: "Versiones del Producto", "Desarrollo Iterativo del Producto" y "Selección de Requerimientos".

El desarrollo de software que funciona sobre una *documentación exhaustiva.* Dentro de la guía se proponen la creación únicamente de los documentos necesarios para tomar decisiones importantes respecto a la arquitectura de forma inmediata, estos son las Especificación de Requerimientos que contiene la lista original de requerimientos deseados junto con las Especificaciones de Requerimientos Funcionales y de Atributos de Calidad (Requerimientos No Funcionales) Seleccionados y el documento de Diseño que contendrá la Arquitectura de la Aplicación que se van a ir actualizando y modificando conforme se realicen los procesos iterativos propuestos en la guía.

De acuerdo con [48] los artefactos generados durante el proceso no muestran por si mismos la agilidad del proceso, la agilidad radica en toda la experiencia al realizarse todo el proceso ya que es hasta entonces que se puede medir la agilidad por el impacto que tiene el proceso en el negocio. También en [48] se propone una manera de medir la agilidad de un proceso basado en metas y no en métricas, propone las metas porque son libres de detalles particulares del proceso y permiten conocer cómo queremos estar al utilizar el proceso más que cómo estamos al realizar el proceso, permitiendo mejorar el proceso y hacer un análisis a futuro del negocio.

Los principios ágiles se aplican en la guía de la siguiente manera:

	Principios ágiles	Aplicación dentro de la guía
1	*La mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor.*	Para lograrlo se requiere que el cliente esté involucrado durante el proceso de desarrollo junto con el compromiso que el equipo de desarrollo adquiere para lograrlo.
2	*Aceptar requerimientos cambiantes, incluso en etapas avanzadas. Los procesos ágiles aprovechan el cambio para proporcionar una ventaja competitiva al cliente.*	Los requerimientos cambiantes son considerados dentro del carácter iterativo de los procesos propuestos, ya que son estos los que van a ir evolucionando tanto la arquitectura como el producto en general.
3	*Entregar software frecuentemente, con una periodicidad desde un par de semanas a un par de meses, con preferencia por los periodos más cortos posibles.*	Se aplica principalmente en el proceso "Versiones del Producto", el equipo de desarrollo se debe comprometer a entregar dichas versiones del software con valor para el cliente en iteraciones cortas continuamente.
4	*Los responsables de negocio y los desarrolladores deben trabajar juntos diariamente a lo largo del proyecto.*	Se aplica al trabajar dentro de las iteraciones de los procesos propuestos, ya que es gracias a esta interacción que el producto evolucionará. La guía promueve este principio al trabajar bajo las modalidades de "Arquitecto Dentro del Equipo" o "Arquitectura del Equipo"
5	*Construir proyectos con profesionales motivados. Dándoles el entorno y soporte que necesitan, y confiando en ellos para que realicen el trabajo.*	Se fomenta principalmente dentro de la organización del equipo de desarrollo, ya que se le da el poder decisión al equipo sobre el producto a generar.
6	*El método más eficiente y efectivo de comunicar la información a un equipo de desarrollo y entre los miembros del mismo es la conversación cara a cara.*	De la misma manera que en el principio 5, para tomar las decisiones necesarias debe existir una comunicación directa y continua entre todo el equipo de desarrollo para evitar conflictos y fomentar la participación activa del mismo.
7	*La principal medida de progreso es el software que funciona.*	El equipo se debe comprometer a entregar software funcional (con valor) al finalizar el proceso de "Desarrollo Iterativo del Producto".
8	*Los procesos ágiles promueven el desarrollo sostenible. Promotores, desarrolladores y usuarios deben ser capaces de mantener un ritmo constante de forma indefinida.*	De la misma manera que en el principio 4, todos los involucrados deben trabajar juntos manteniendo un ritmo constante a lo largo del proyecto. Se considera que es un compromiso que adquieren todas las partes involucradas.
9	*La atención continua a la calidad técnica y los buenos diseños mejoran la agilidad.*	Se adquiere como compromiso durante todo el proceso de desarrollo al enfocarse en los atributos de calidad en conjunción con los requerimientos funcionales, puesto que al final lo que se quiere obtener es un producto con calidad sin sobrecarga de trabajo e información. Se aplica principalmente dentro de las actividades de diseño e implementación del producto.
10	*La simplicidad es esencial.*	De la misma manera que en el principio 9.
11	*Las mejores arquitecturas, requerimientos y diseños surgen de equipos que se auto organizan.*	Principalmente al trabajar con las modalidades de "Arquitecto Dentro del Equipo" y "Arquitectura del Equipo".

12	*A intervalos regulares el equipo debe reflexionar sobre cómo ser más efectivo, entonces mejora y ajusta su comportamiento de acuerdo a sus conclusiones.*	Es una práctica continua que se aplica principalmente al finalizar iteraciones y antes de iniciar otras. Ya que sirve para hacer retrospectiva del trabajo realizado, buscar soluciones en caso de haber problemas y fomentar las buenas prácticas.

Tabla 21. Aplicación de los principios ágiles dentro de la guía de actividades propuesta

Todos los valores y principios están relacionados entre sí, por lo que en conjunto van a intervenir en el grado de "agilidad" que tenga la aplicación de la guía propuesta.

Capítulo 5. Caso de Estudio

5.1 Descripción de Hemosist

Como parte de la asignatura “Ingeniería de Software Orientada a Objetos” impartida dentro de la Maestría en Ciencia e Ingeniería de la Computación en el área de Ingeniería de Software y Bases de Datos, se desarrolló una aplicación web de manejo de información de pacientes del Instituto Nacional de Cardiología (INC) “Ignacio Chávez” para el área de Hemodinámica (Hemosist) a través de un convenio entre esta institución y el Posgrado en Ciencia e Ingeniería de la Computación en la UNAM.

En el primer ciclo el objetivo fue permitir la administración de datos de los pacientes, diagnósticos y procedimientos practicados en ellos, en el área de Hemodinámica. También se realizó un módulo de administración básico que permite gestionar a los usuarios del sistema así como el catálogo del personal médico. Para esta primera versión del producto se tenía una fecha de entrega límite el 31 de diciembre del 2009.

Dentro del mismo convenio se estableció un segundo ciclo de garantía de un mes sobre la primera versión del sistema que se llevó a cabo al concluir el primer ciclo, durante este periodo se realizaron mejoras y se agregaron nuevas funcionalidades.

De la misma manera se solicitó un tercer ciclo en el que se ampliaron las funcionalidades del sistema para explotar la base de datos creada en el primer ciclo a través de la extracción de la información almacenada de la base de datos en un formato sencillo que facilitara el análisis estadístico de dicha información y con ello apoyar la investigación médica que se realiza en el Instituto Nacional de Cardiología.

Estos tres ciclos permitieron utilizarlos como casos de estudio sobre la toma de decisiones arquitectónicas utilizando el Proceso Unificado Ágil como metodología de desarrollo y las prácticas que promueve XP.

En la tesis [49] se explica a mayor profundidad los aspectos de administración de este proyecto ágil.

5.2 Organización del Equipo de Desarrollo

El Equipo de Desarrollo (ED) utilizó los roles definidos en el Proceso de Equipo de Software (TSP, por sus siglas en inglés) [50] para la organización del equipo ya que se adaptaban al número de integrantes y a la personalidad de cada uno. Todos los integrantes desempeñaban el rol de Ingeniero de Software (IS) más uno de los siguientes roles: Administrador de Planeación (AP), Administrador de Desarrollo (AD), Líder del Equipo (LE), Administrador de Proceso y Calidad (APC) y Administrador de Apoyo (AA). Además de los roles del equipo, se incluyó el rol de Gerente de proyecto (GP) y fue desempeñado por la titular de la clase. Además se contaba con el Equipo del Proyecto (EP) formado por el cliente y el usuario final el cual tomaba las decisiones y en caso de

que implicará un impacto mayor éste se comunicaba con el cliente. Una explicación a mayor detalle de los roles definidos se encuentra en la tesis [49].

Se trabajó bajo el esquema de "Arquitectura del Equipo" y el Administrador de Desarrollo fungió como el "Arquitecto Ágil" durante todos los ciclos de desarrollo de Hemosist.

Todo el equipo junto con el usuario final estaban motivados a comprometerse desde el inicio del proyecto a tener una relación continua y directa, lo cual fue un factor que facilitó el desarrollo del proyecto con éxito ya que gracias a esta interacción se facilitó el entendimiento del dominio.

5.3 Aplicación de la Guía en Hemosist

5.3.1 Primer Ciclo

El Equipo de Desarrollo decidió que se realizarían "Versiones del Producto en Desarrollo" en iteraciones que incluirían las actividades de análisis y diseño englobadas en el proceso propuesto de "Diseño de la Arquitectura" e implementación, pruebas y despliegue como parte de las actividades propuestas en "Versiones del Producto" y "Desarrollo Iterativo del Producto" por cada caso de uso. Una vez terminados los casos de uso se entregó en el ambiente de despliegue la primera "Versión del Producto Liberado".

Para mitigar por completo el riesgo de desconocimiento de la tecnología, el equipo decidió comenzar con el desarrollo del módulo de administración, con una pareja y un trió de programadores, juntando a un desarrollador con experiencia con otro(s) de menor experiencia.

Una vez concluido el módulo de administración en una iteración, se comenzó a trabajar con el módulo de captura de información de pacientes siguiendo el proceso descrito a continuación:

1. El usuario analizaba con el AD uno o más casos de uso según lo considerará el Administrador de Planeación.
2. Si el caso de uso era muy grande el AD lo dividía con ayuda del usuario para conservar la lógica.
3. La interfaz de usuario de la mayoría de los casos de uso era programada por un IS.
4. Cada miembro del ED elegía el caso de uso que desarrollaría. Si el IS encargado de realizar las interfaces tenía sobrecarga de trabajo, esta tarea se volvía responsabilidad del IS que había elegido el caso de uso.
5. El ED calculaba el tiempo en que podría mostrar avances al usuario o concluir el caso de uso y lo convocaba a una junta para aclaración de dudas y validación del avance logrado o entrega del caso de uso terminado.
6. El IS integraba su caso de uso validado y aceptado por el usuario.

A lo largo de todo el ciclo de desarrollo se tuvo una relación continua y constante entre el ED y el usuario para definir el contenido de los módulos de información y la forma en que se reflejaría dicha información en las interfaces.

De acuerdo a la guía propuesta las actividades por etapa que diseñaron y definieron a la arquitectura de "Hemosist" se realizaron de la siguiente manera:

1. **Versiones del Producto:** Para este primer ciclo se realizaron las iteraciones correspondientes a cada módulo de información entregadas en la primera versión del producto liberado que se entregó el 21 de diciembre del 2009. La actividad **(1)** de definición de los requerimientos se realizó en una reunión con el usuario final en la que se explicó el dominio en el que se trabajaría y se definieron los primeros requerimientos para la primera versión.
2. **Desarrollo Iterativo del Producto y**
3. **Selección de Requerimientos:**
   Se definieron los siguientes requerimientos y restricciones de la actividad **(2)**:

   - **Restricciones:** La principal restricción fue trabajar con software libre para evitar problemas de licenciamiento y elevación del costo del sistema.

   - **Atributos de Calidad:** Se definieron dos:

     - ✓ **Usabilidad.** Debido a la gran cantidad de información manejada por paciente, se definió la importancia de que la interfaz fuera intuitiva y lo más sencilla posible.

     - ✓ **Escalabilidad.** Puesto que se planeó agregar nuevas funcionalidades en ciclos subsecuentes.

   - **Requerimientos Funcionales:** Debido a la complejidad de los datos solicitados el cliente dividió a la información en módulos de información del paciente de acuerdo a su conveniencia. Cada módulo de información correspondía a un caso de uso a desarrollar cuyos datos se refinarían conforme el avance iterativo del proyecto.

4. **Diseño de la Arquitectura**: Las actividades **(3)** y **(4)** propuestas en la guía de actividades se realizaron únicamente al inicio del proyecto.

   La "Arquitectura" propuesta al realizar la actividad **(3)** por el AD de acuerdo a los requerimientos funcionales y atributos de calidad solicitados utiliza los siguientes estilos arquitectónicos:

   1. Por ser una aplicación web, se usó el estilo de arquitectura distribuida *Cliente-Servidor* y jerárquica de *Capas* [3], [8], [12], [51]:

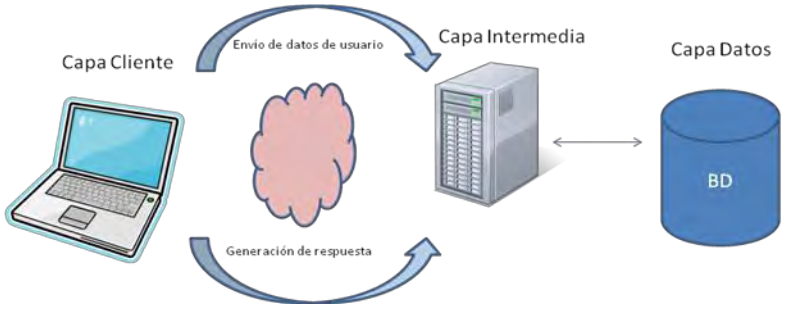

Figura 32. Estilo de Capas y Cliente-Servidor en una Aplicación Web

2. De acuerdo con [3] y [12] se utilizó el estilo de "Abstracción de datos y Organización orientada a objetos" debido a que se utilizó Java como lenguaje de programación.

3. Se utilizó el estilo "Orientado a interacción" ya que se utilizó la arquitectura Modelo Vista Controlador (MVC) [3] ,[8], [12] que es de las mejores opciones para las aplicaciones web (Ver Capítulo 2, Sección 2.4.6.1). A continuación se introduce cada una de sus partes de acuerdo a como se manejó en el proyecto:

   Struts es un marco de trabajo que implementa el estilo arquitectónico MVC. Este marco de trabajo pide la creación de clases de tipo *Action*, que son las encargadas de manejar los distintos tipos de peticiones que llegan desde el cliente. Por ejemplo, la clase *AutenticarAction* se encarga de mandar llamar la parte del modelo que autenticará al usuario.

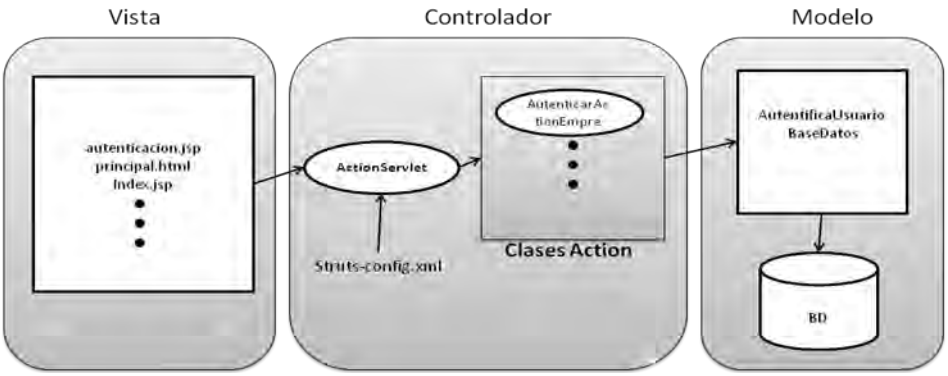

Figura 33. MVC - STRUTS

Dentro de las ventajas de usar *Struts* se encuentran:

- Transporte automático de los datos introducidos en el cliente (JSP) hasta el controlador *(Action)* mediante formularios *(ActionForm)*.
- Transporte automático de los datos enviados por el controlador *(Action)* a la parte de presentación (JSP) mediante formularios *(ActionForm)*.

- Implementa la parte común a todas las aplicaciones en la parte de Controlador *(ActionServlet)*; la parte particular de cada aplicación es fácilmente configurable *(struts-config.xml)*.
- Facilita la validación de formularios.
- La separación de los componentes en capas (MVC) simplifica notablemente el desarrollo y su mantenimiento.

Algunas de sus desventajas:

- El hecho de no abarcar todas las capas de la aplicación web (deja fuera la capa de negocio y la capa de persistencia) hace que el interfaz entre Struts y estas capas no esté tan automatizado, convirtiendo los accesos a los datos (DAO) en monótonos de desarrollar.
- Mayor curva de aprendizaje.
- Menor transparencia debido a la complejidad del marco de trabajo.

Del lado del modelo se tienen las clases que se encargarán de la conexión a la base de datos y de la lógica del negocio, para lo cual se utilizaron los patrones DAO (*Data Access Object)* y VO *(Value Object)*, donde las clases de tipo VO son representaciones de las tablas de la base de datos y las clases DAO son las que permiten realizar operaciones con la base de datos, haciendo transparente el acceso a la misma.

Se eligió esta arquitectura debido a la experiencia que el AD tuvo en aplicaciones pasadas sobre STRUTS como un marco de trabajo que ofrece los beneficios del estilo arquitectónico Modelo-Vista-Controlador descritos previamente en la sección 2.4.6.1 de estilos arquitectónicos los cuales coincidían con los de este proyecto junto con los beneficios del estilo de Capas y de la Orientación a Objetos en Java, resultados obtenidos de la actividad **(4)** usando la "Técnica de Evaluación Basada en Experiencia" . El "Reporte de Estimación/Evaluación" se resume en la Tabla 22.

	Economía en Tiempo	**Economía en Espacio**	***Completitud***	**Seguridad**	**Interoperabilidad**	**Independencia de Hardware**	**Independencia de Software**	**Facilidad de Instalación**	***Reusabilidad***	**Tolerancia a Errores**	**Disponibilidad**	***Facilidad de Entendimiento***	***Interfaces de Usuario***	***Facilidad de Aprendizaje***
OO	+	+	+				+		+				+	-
Capas	-			+		+			+	+	+	+		-
MVC					+									+
Cliente-Servidor														+

Tabla 22. Resultados de la Evaluación Usando la Tabla Informativa de Estilos Arquitectónicos

De los atributos de calidad que resaltan de la arquitectura heterogénea obtenida son la "Economía en Tiempo", "Completitud de la Información" e "Interfaces de Usuario" al usar el estilo arquitectónico Orientado a Objetos, la "Reusabilidad" y "Facilidad de

Entendimiento" proporcionada por el estilo jerárquico de Capas y la "Facilidad de Aprendizaje" de los cuatro estilos. Los cuales se reflejan durante la experiencia de desarrollo de Hemosist.

La "Arquitectura" final se muestra en los siguientes diagramas de paquetes generados al utilizar *Struts*:

Figura 34. Diagrama de Paquetes del Controlador y Modelo Utilizados en Struts para Hemosist

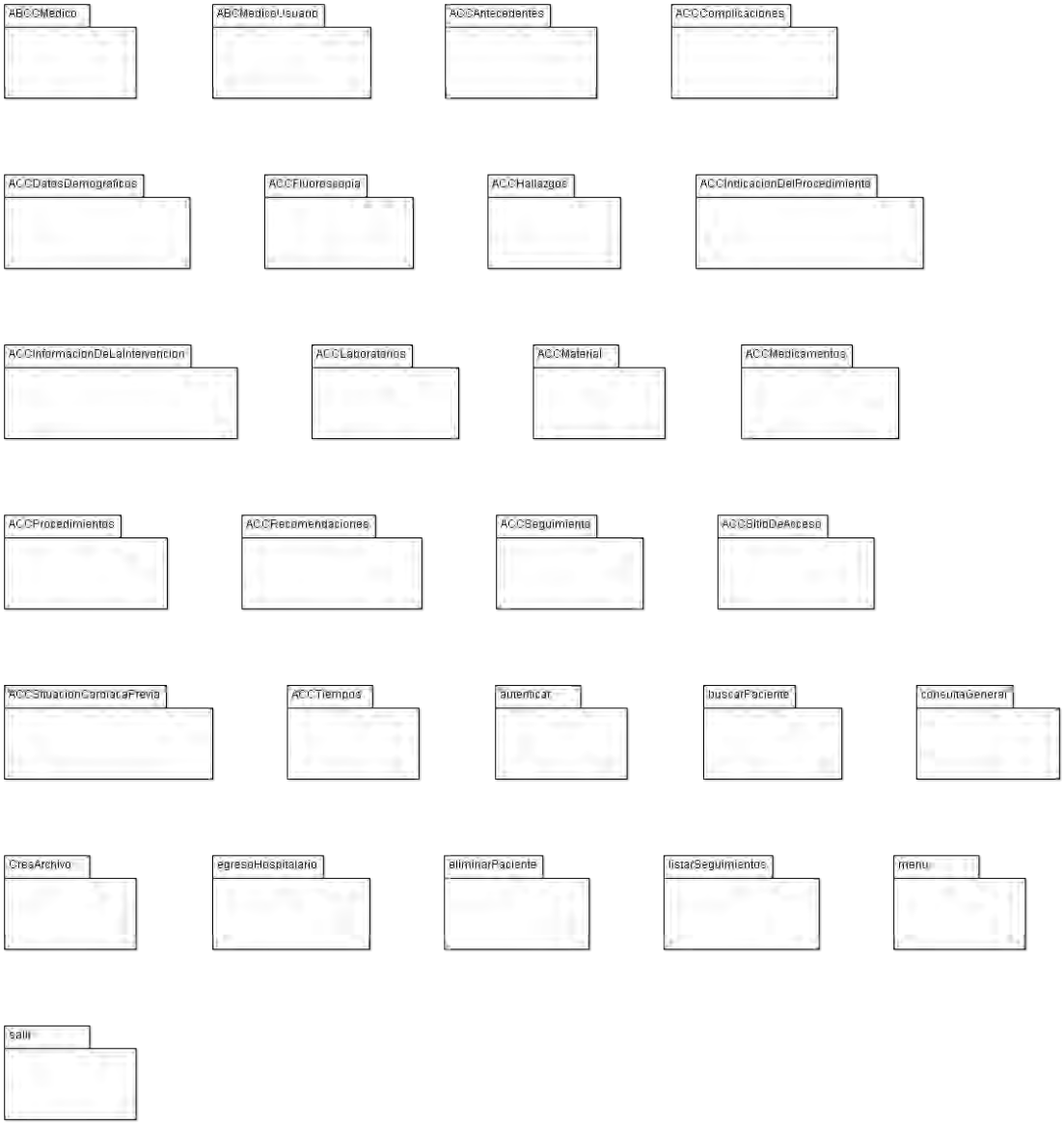

Figura 35. Diagrama de Paquetes de "Control" (Controlador)

El paquete "formBeans" contiene los VOs y "Modelo" los DAO juntos forman el Modelo del MVC en *Struts*.

Figura 36. Diagrama de Paquetes de "formBeans"

Figura 37. Diagrama de Paquetes del "Modelo"

El paquete "util" contiene algunas clases y utilidades que se utilizan para la presentación de datos calculados en la interfaces y algunas funcionalidades del menú.

El paquete que forma la Vista del MVC es parecido al paquete de "formBeans" puesto que las clases contenidas en este último permiten la alta, baja, cambios y consultas dentro de los formularios de cada una de las interfaces.

Para este ciclo no hubo necesidad de realizar la actividad **(5)**, por lo que partiendo de la arquitectura definida anteriormente se prosiguió con las siguientes actividades normalmente.

En general en este ciclo se realizó el mayor trabajo arquitectónico puesto que fue en éste que se diseño y desarrollaron las bases de la arquitectura. El mayor reto arquitectónico al que se enfrentó el equipo fue al entendimiento del marco de trabajo *Struts* (tecnología que implementa el estilo arquitectónico MVC) debido a la inexperiencia que se tenía en ese momento respecto a su uso, pero gracias a la programación en parejas propuesta dentro de las prácticas de XP, el equipo aprendió el uso de esta tecnología tempranamente y aceptó la arquitectura de manera oportuna y se tuvo una mayor velocidad de desarrollo sin mayores contratiempos dentro del diseño arquitectónico.

5.3.2 Segundo Ciclo

El objetivo del segundo ciclo consistía en mejorar la aplicación refinando la información y la forma como se presentaba en las interfaces de acuerdo a la primera versión entregada en el primer ciclo y agregar nuevas funcionalidades.

Para el segundo ciclo no se realizaron modificaciones arquitectónicas importantes puesto que únicamente se realizaron correcciones y modificaciones a los datos de los módulos de información del paciente, así como se agregaron nuevos módulos de información de los que se tenían en el primer ciclo, en la parte de administración: Eliminar paciente, Eliminar procedimiento, ABC de personal médico en la sección de administración de catálogos. En la parte del módulo de capturista: ACC Datos de Ingreso, Listar pacientes para realizar seguimiento extra hospitalario, Listar pacientes con egreso hospitalario retrasado.

5.3.3 Tercer Ciclo

El objetivo del tercer ciclo consistió en ampliar las funcionalidades originales aumentando las capacidades del módulo de administración para permitir realizar consultas básicas a los módulos de información existentes y pasar esa información a un archivo cuyo formato fuera fácil de manipular para permitir realizar el análisis estadístico de dicha información y con ello apoyar la investigación médica.

El reto que se enfrentó fue el de encontrar un formato que permitiera la exportación de las consultas básicas de manera que se pudiera manipular fácilmente por el usuario y la implementación de este para cada módulo de información. Al final se eligió el formato de los archivos **CSV** [15] (del inglés *comma-separated values*) los cuales son un tipo de documento en formato abierto sencillo para representar datos en forma de tabla, en las que las columnas se separan por comas (o punto y coma en donde la coma es el separador decimal) y las filas por saltos de línea. La ventaja que ofrece este tipo de archivo es su compatibilidad para usarse en cualquier programa de hojas de cálculo (Excel dentro de Microsoft Office, Calc en Open Office, entre otros).

Debido a que se tenía muy bien establecida la arquitectura del sistema, no se realizaron modificaciones importantes en la arquitectura.

[15] http://tools.ietf.org/html/rfc4180

5.4 Conclusiones Respecto al Diseño y Evolución de la Arquitectura de Hemosist

Se puede concluir que los estilos arquitectónicos elegidos para el sistema Hemosist favorecieron los atributos de calidad de *usabilidad* y *escalabilidad* solicitados por el Instituto Nacional de Cardiología en los tres ciclos descritos.

No hubo que realizar modificaciones arquitectónicas importantes para el segundo ciclo y principalmente en el tercer ciclo en el que se agregó la funcionalidad de exportación de la información en un formato que facilitara el análisis estadístico. Debido a lo anterior no se tuvieron mayores problemas respecto a la *escalabilidad*.

Por parte de la *usabilidad* los usuarios del INC manifestaron en general su satisfacción respecto a la interfaz que se les presentó, ya que la forma en la que se presenta la información de cada paciente corresponde a sus necesidades de captura y consulta.

La complejidad del proyecto radicó principalmente en la forma de organizar la enorme cantidad de datos en módulos de información del paciente de manera que fueran intuitivos para los usuarios del INC. Para lograrlo se requirió de la interacción constante entre el ED y el usuario durante todo el desarrollo del proyecto.

Conclusiones

Las arquitecturas de software tienen un gran impacto dentro del desarrollo de software independientemente de la metodología de desarrollo que se utilice.

Debido a que las metodologías ágiles se fundamentan en experiencias de desarrolladores expertos no existe mucha formalidad en la mayoría de estas metodologías respecto a las arquitecturas de software. Sin embargo, recientemente se ha incrementado el interés en las arquitecturas de software y principalmente en la integración de actividades de diseño arquitectónico en las metodologías ágiles [2].

El objetivo principal de este trabajo fue desarrollar una guía que se apegara a los valores y principios de las metodologías ágiles para apoyar en el proceso de evolución de la arquitectura de software de un proyecto en un entorno ágil, de manera que al aplicarla permitiera al desarrollador definir una arquitectura y evaluarla al paso de las iteraciones.

Se pretende que la guía propuesta sea una herramienta de apoyo arquitectónico en el desarrollo de software bajo ambientes ágiles, puesto que integra actividades de diseño arquitectónico en el proceso de desarrollo que facilitan la integración de la filosofía ágil. Debe recordarse que la agilidad se alcanza a través de un conjunto de buenas prácticas que tienen como origen la filosofía ágil y es un conjunto que ofrece un impacto significativo para el negocio. Por lo que la guía por sí sola no es una herramienta ágil, la utilización de la guía propuesta, junto con la integración de las buenas prácticas propuestas por la *Agile Alliance,* permitirán agilizar el proceso de desarrollo de software.

Este trabajo comprende la recopilación y resumen de los conceptos clave de las arquitecturas de software, los estilos y patrones arquitectónicos, conceptos y modelos dentro de la calidad arquitectónica, documentación y evaluación de las arquitecturas de software, y por último las metodologías ágiles. Toda esta información está integrada dentro de la "Guía para la Evolución de las Arquitecturas de Software en Ambientes Ágiles", la cual fue aplicada dentro del caso de estudio de Hemosist.

La experiencia en el análisis de la evolución de la arquitectura de Hemosist además de permitir ser un caso de estudio de aplicación, permitió refinar las actividades propuestas en la guía de acuerdo a la experiencia que se tuvo como desarrollador durante el proceso de desarrollo ágil del sistema. Lo anterior hace que sea una guía que se fundamenta en la teoría integrando la experiencia en un proyecto real.

Como trabajo futuro y temas de investigación futura se proponen lo siguiente:

- Someter la aplicación de la guía a proyectos de desarrollo de software más complejos para refinar las actividades que aquí se proponen.
- Adecuar la guía para su aplicación bajo proyectos con equipos distribuidos.

- Implementar dentro de la guía algún mecanismo de control de cambios y solución de compensaciones (*trafeoffs*).
- Proponer dentro de la guía técnicas y/o herramientas para llevar a cabo la documentación de los productos propuestos de manera que se adecúen a los valores y principios ágiles.
- Desarrollar una nueva guía para el desarrollo de líneas de productos de software tomando como referencia este trabajo de tesis.

Apéndice A. Plantilla de Análisis

Análisis

1. Introducción

[La introducción del Documento del Análisis de Software (DAS) ofrece una visión general de todo el documento. Incluye el propósito, alcance, definiciones, acrónimos, abreviaturas, referencias, y una descripción general de la fase de Análisis.]

1.1 Propósito

[Especifica el propósito del DAS. Se describe brevemente la estructura del documento, primordialmente se deberá indicar a quién va dirigido este documento así como la manera en que se espera sea interpretado el contenido de éste.]

1.2 Alcance

[Breve descripción del alcance del DAS, indicando qué es influenciado o afectado con el contenido de este documento.]

1.3 Definiciones, Acrónimos y Abreviaturas

[Proveer de las definiciones, términos y acrónimos requeridos, para la correcta interpretación del documento. Se puede incluir una referencia al glosario de términos en caso de que éste exista.]

Término	Definición
[Término, acrónimo o abreviatura]	*[Definición del término]*

1.4 Referencias

[Esta sección provee un listado de todos los documentos a los que se haga referencia dentro del contenido del DAS. Éste debe ser lo suficientemente específico para poder localizarse, se puede incluir el identificador o nombre del documento referido. Especificar la fuente de donde se ha obtenido la referencia.]

2. Descripción general

[Esta sección debe describir lo que el resto del DAS contiene y explica la forma en que el documento está organizado.]

3. Representación de la Arquitectura

[Esta sección describe los requisitos de software y los objetivos que tienen algún impacto significativo en la arquitectura, por ejemplo, la seguridad, privacidad, portabilidad, distribución y reutilización.]

3.1 Metas de la arquitectura

[Esta sección contiene los objetivos esperados del sistema que tienen que ver con la arquitectura del sistema, este punto debe contener los suficientes puntos de apoyo para sostener una discusión, y una posterior elección, de la arquitectura a utilizar durante la fase de Diseño.]

3.2 Restricciones de la arquitectura

[Esta sección contiene todas las restricciones a las que se enfrenta el desarrollo del sistema y que impactan en la arquitectura.]

4. Modelo del Análisis

[Esta sección contiene al conjunto de paquetes de más alto nivel identificados a partir de los requerimientos presentados en la ERS. Se representarán las abstracciones en subsistemas o capas del sistema.]

[Nota: si la decisión de la arquitectura ha sido tomada previamente por el cliente o existe una alternativa óptima cuya elección es irrevocable, este punto puede omitirse y especificarse hasta el DDS.]

5. Clases del Análisis

5.1 Identificación de las clases

[Esta sección contendrá los diagramas que describen el comportamiento estático del sistema y los tipos de relaciones existentes entre las clases. Se generará un diagrama de clases por cada paquete que contenga el sistema.

Para generar el documento, es conveniente como primer paso identificar qué clases será necesario crear para que el sistema cumpla con los requerimientos establecidos, esto se realizará revisando cada caso de uso identificando qué necesidades cubre cada uno de éstos y con ello decidir qué clases cumplirán con ese trabajo y situarlas en el paquete adecuado.]

5.1.1 Clases de Interfaz o del paquete 1

5.1.2 Clases de entidad o del paquete 2

5.1.3 Clases de control o del paquete 3

6. Realización de Caso de Uso - Análisis

[Cada caso de uso deberá ser representado con un diagrama de secuencia, éstos muestran cómo se comportan los objetos entre ellos a través del tiempo. Representan el comportamiento dinámico de los casos de uso. Deberá tomarse en cuenta que por cada caso de uso deben generarse al menos 2 diagramas de secuencia ejemplificando los flujos normales y aquellos que manejan excepciones. Además se deberá identificar cada una de las clases participantes en el caso de uso para que se representen en el diagrama de secuencia. Estas clases deberán aparecer en alguno de los diagramas de clases.

Para mostrar la navegación en el sistema se usarán diagramas de estado. Se construye un diagrama para indicar cómo navega cada tipo de usuario a través de las interfaces del sistema. Cada diagrama tendrá un

estado inicial que se identificará con una circunferencia negra que rodea un punto negro, los estados serán representados por un rectángulo redondeado con el nombre de la interfaz en la que se encuentra el usuario al realizar alguna acción, las acciones estarán determinadas en el documento por flechas con un estado origen, el evento que origina el cambio a un estado destino. Es necesario indicar aquellos eventos que generarían un error y manejarlos de acuerdo a sus necesidades, por ejemplo enviando al usuario a un estado donde pueda recuperarse para continuar sin necesidad de reiniciar todo el proceso.]

6.1 Diagramas de interacción

[Diagramas de secuencia.]

6.2 Diagramas de navegación

[Diagramas de estado.]

7. Justificación de las decisiones del Análisis

[Esta sección debe contener las justificaciones de las decisiones tomadas que resultaron del Análisis de los requerimientos del sistema definidos anteriormente.

Este planteamiento es especialmente importante en casos como:

Más de una alternativa puede satisfacer algunos de los requisitos asignados al sistema.

Dos o más de los asignados a los requisitos del sistema se oponen.

Existen restricciones de diseño.

La arquitectura del sistema o parte de ella se define por el cliente.

El sistema debe interactuar con sistemas ya existentes.]

8. Información de apoyo

[La información de apoyo hace que el DAS sea más fácil de usar. Incluye:

Tabla de contenidos

Índice

Apéndices]

Apéndice B. Plantilla de Diseño

Diseño

1. Introducción

[La introducción del Documento de Diseño del Software (DDS) ofrece una visión general de todo el documento DDS. Incluye el propósito, alcance, definiciones, acrónimos, abreviaturas, referencias, y una descripción general de la DDS.]

1.1 Propósito

[Especifica el propósito de la DDS. Este documento ofrece un panorama de la arquitectura del sistema, utilizando un número de diferentes puntos de vista arquitectónico para representar diferentes aspectos del sistema. Se tiene la intención de captar y transmitir las decisiones importantes de arquitectura que se han hecho en el sistema.]

1.2 Alcance

[Una breve descripción de la Arquitectura de Software, las decisiones del ambiente de implementación y lo que se ve afectado o influenciado por este documento.]

1.3 Definiciones, Acrónimos y Abreviaturas

[Proveer de las definiciones, términos y acrónimos requeridos, Se puede incluir una referencia al glosario de términos en caso de que éste exista.]

Término	Definición
[Término, acrónimo o abreviatura]	*[Definición del término]*

1.4 Referencias

[Esta sección provee un listado de todos los documentos a los que se haga referencia dentro del contenido de la DDS. Éste debe ser lo suficientemente específico para poder localizarse, se puede incluir el identificador o nombre del documento referido. Especificar la fuente de donde se ha obtenido la referencia.]

2. Descripción general

[Esta sección debe describir lo que el resto del DDS contiene y explica la forma en que el documento está organizado.]

3. Representación de la Arquitectura

[Esta sección describe la arquitectura del software para el sistema actual, y la forma en que estará representada. Enumera las vistas que son necesarias, y explica los tipos de modelos de elementos que contiene cada una de ellas.]

3.1 Descripción de la arquitectura

[Recoge las condiciones específicas que pueden aplicarse: el diseño y la estrategia de aplicación, el ambiente de desarrollo y su versión, las herramientas de diagramación para UML, si se requiere un manejador de bases de datos, cuál se usará y en qué versión, si la aplicación es para web, se define el servidor de aplicaciones, herramientas para automatizar las pruebas unitarias, etc.]

4. Vista Lógica

4.1 Descripción

[Esta sección describe de manera general la descomposición del modelo mediante jerarquía de paquetes.]

4.2 Paquetes de la arquitectura

[Por cada paquete se debe incluir un nombre, una breve descripción y un diagrama con las clases más importantes que contiene.]

4.2.1 <Nombre del Paquete 1>

4.2.1.1 Descripción breve del paquete

[La descripción debe expresar brevemente el propósito del paquete.]

4.3 Clases del Diseño

[Para cada clase del paquete se debe incluir un detallado de las clases identificadas, una descripción de las responsabilidades de la clase como sus atributos y operaciones.]

4.3.1 <Diagrama de clases detallado>

5. Vista del Proceso

5.1 Descripción

[Esta sección describe la descomposición del sistema en procesos y grupos de procesos. Los organiza de acuerdo a las relaciones de interacción que existen entre ellos. Describe los principales medios de comunicación utilizados en dichas interacciones.]

6. Vista de Despliegue

6.1 Descripción

[En esta sección se debe describir la configuración de la red física del software para su ejecución. Se deberán indicar los nodos (computadoras, routers, etc.) que son necesarios para el funcionamiento correcto del software, además de las interconexiones entre los nodos (bus, LAN, etc.). Idealmente se debe incluir el mecanismo o protocolo de comunicación entre los nodos.]

6.2 Modelo de despliegue

[Distribución de la arquitectura.]

7. Vista de Datos

7.1 Descripción

[Debe incluirse una descripción del mecanismo de almacenamiento de datos del sistema. Esta sección es opcional, si hay poca o ninguna persistente de datos, o la traducción entre el modelo de diseño y el modelo de datos es trivial.]

7.1.1 <Diagrama de la Base de Datos>

8. Plan de Pruebas de Integración

8.1 Descripción

[Contiene una breve descripción de cómo se planea realizar la integración del sistema. Se establece el orden de los componentes que se irán integrando. La recomendación es tomar en cuenta la arquitectura y la dependencia estática, es decir, identificar qué paquetes dependen de otros para funcionar correctamente.]

8.2 Subsistemas

[Listado de los subsistemas que se integrarán en la iteración, este listado debe estar establecido de acuerdo al orden de integración.]

8.3 Requerimientos para la integración

[Aquí se debe incluir el listado de los requerimientos necesarios para obtener un ambiente que sea adecuado para realizar la integración.]

8.4 Estrategia de Integración

[La integración será dividida en una serie de incrementos, cada incremento resulta en un producto funcional, a dicho producto debe de estar relacionado a un conjunto de casos de prueba de integración. Para cada paso de la integración se debe especificar cómo se construye y los criterios para su evaluación.]

8.4.1 Objetivo

[Detectar fallas de interacción entre las distintas unidades que componen al sistema, asegurando una funcionalidad conjunta.]

8.4.2 Técnica

[Establecer claramente el orden de integración de cada uno de los componentes, esto debe basarse en la dependencia de paquetes definida en la Arquitectura. Además se deberán incluir aspectos de:

- *Construcción*

  Construir scripts e instrucciones de cómo debe integrarse el sistema

  Registros de referencia que definan las versiones de los elementos de configuración utilizados para la integración

- *Evaluación y prueba*

  Los criterios de evaluación, son una descripción de las cualidades que debe cumplir el producto integrado al final de cada incremento.

  Instrucciones de instalación y configuración para ejecutar y probar la integración.]

8.4.3 Criterio de terminación

[Todas las pruebas previstas se han ejecutado y todos los defectos identificados han sido registrados. Cuando los pasos de prueba, los procedimientos de prueba, scripts de prueba hayan arrojado los resultados esperados.]

8.4.4 Consideraciones especiales

[Identificar o describir los temas o asuntos (internos o externos) que afectan a la aplicación y ejecución de las pruebas.]

8.5 Herramientas

[Listado de las herramientas empleadas para la integración.]

8.6 Recursos

[Esta sección presenta los recursos necesarios para llevar a cabo la integración y las pruebas, es un listado de las principales responsabilidades de cada recurso y el conocimiento y habilidades necesarias para completar las pruebas por parte de cada elemento.]

8.7 Casos de prueba

8.7.1 <Iteración 1>

9. Justificación de los requerimientos de software

[Esta sección debe contener las justificaciones del por qué se tomaron las decisiones importantes del diseño del sistema definidas anteriormente.

Este planteamiento es especialmente importante en casos como:

Más de una alternativa puede satisfacer algunos de las necesidades del sistema.

Existen restricciones de diseño.

La arquitectura del sistema o parte de ella la define el cliente.

El sistema debe interactuar con sistemas ya existentes.]

10. Información de apoyo

[La información de apoyo hace que la DDS sea más fácil de usar. Incluye:

Tabla de contenidos

Índice

Apéndices]

Referencias

1. **Nord, Robert L., Tomayko, James E. y Wojcik, Rob.** *Integrating Software-Architecture-Centric Methods into Extreme Programming (XP).* Software Engineering Institute, Carnegie Mellon University. Pittsburgh, MA : CMU-SEI, 2004. pág. 45.

2. **Coplien, James y Bjørnvig, Gertud.** *Lean Architecture: For Agile Software Development.* Chichester, UK : John Wiley & Sons, 2010.

3. **Shaw, Mary y Garlan, David.** *An Introduction to Software Architecture.* New Jersey : World Scientific, 1993, Vol. I.

4. **Sullivan, Ray.** Scrum and Architecture. *The Quality Zone.* [En línea] 16 de Junio de 2009. [Citado el: 4 de Agosto de 2010.] http://www.raysullivan.com/2009/06/scrum-and-architecture/.

5. **Cervantes, Humberto.** *Arquitectura de Software.* No. 27, México, D.F. : Brainworx S.A de C.V, 2010, SG Software Guru, Vol. I. 1870-0888.

6. **Nord, Robert L. y Tomayko, James E.** *Software Architecture- Centric Methods and Agile Development.* s.l. : IEEE Software, 2006.

7. **Somerville, Ian.** *Ingeniería del Software.* Séptima. Madrid : Pearson Educación, 2008.

8. **Buschmann, Frank.** *A System of Patterns. Patterns-oriented Software Architecture.* Chichester, UK : John Wiley & Sons, 1996.

9. **Bass, Len, Clements, Paul y Kazman, Rick.** *Software Architecture in Practice.* Boston : Addison-Wesley, 2003.

10. **McGovern, James y Ambler, Scott W.** *A practical guide to enterprise architecture.* s.l. : Prentice Hall, 2003.

11. **Barbacci, Mario, y otros.** *Quality Atributtes.* Pittsburgh, MA : CMU/SEI, 1995. http://www.sei.cmu.edu/reports/95tr021.pdf.

12. **Qian, Kai.** *Software Architecture and Design Illuminated.* Sudbury : Jones and Bartlett Illuminated Series, 2010.

13. **Bass, Len, Klein, Mark y Bachmann, Felix.** *Quality Attribute Design Primitives.* Pittsburgh, MA : CMU/SEI, 2000.

14. **Kruchten, Philippe.** *The Rational Unified Process An Introduction.* Boston, MA : Addison-Wesley, 2004.

15. **Clements, Paul, Kazman, Paul y Klein, Mark.** *Evaluating Software Architectures: Methods and Case Studies.* Boston : SEI Series in Software Engineering, Addison Wesley, 2002.

16. **Kan, S., Basili, V. y Shapiro, L.** *Software Quality: An overview from the perspective of total quality management.* 1, s.l. : IBM, 1994, IBS Systems Journal, Vol. 33, págs. 4-19.

17. **Pressman, Roger.** *Software Engineering: A Practitioner´s Approach.* New York : McGraw-Hill, 2001.

18. **Losavio, Francisca, y otros.** *Quality Characteristics for Software Architecture.* 2, s.l. : ETH Zurich, Chair of Software Engineering ©JOT, 2003, Journal of Object Technology, Vol. 2, págs. 133-150. http://www.jot.fm/issues/issue_2003_03/article2.pdf.

19. **McCall, Jim A., Richards, Paul K. y Walters, Gene F.** *Factors in Software Quality.* New York : Rome Air Development Center, 1977. RADC-TR-77-369.

20. **Dromey, R. Geoff.** *Cornering the Chimera.* 1, IEEE Software : IEEE Computer Society Press, 1996, Vol. 13, págs. 33-43. http://www98.griffith.edu.au/dspace/bitstream/10072/19821/1/4889_1.pdf.

21. **Grady, Robert B. y Caswell, Deborah L.** *Software metrics: establishing a company-wide program.* Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1987.

22. **Kruchten, P. B.** *The 4+1 View Model of Architecture.* s.l. : IEEE Software, 1995.

23. **Perry, Dewayne E. y Wolf, Alexander L.** *Foundations for the Study of Software Architecture.* 4, s.l. : ACM SIGSOFT Notes, 1992, Vol. 17.

24. **Shaw, Mary y Garlan, David.** *Software Architecture - Perspectives on an Emerging Discipline.* s.l. : Prentice Hall, 1996.

25. **Soni, D., C., Nord y C., Hofmeister.** *Software Architecture in Industrial Applications.* Seattle, Washington : ACM Press, 1995.

26. **Parnas, D. L.** *Software Aging.* s.l. : IEEE Procedings of the 16th International Conference on Software Engineering, 1994.

27. **Karlsson, E. A.** *Software Reuse- A Holistic Approach.* s.l. : John Wiley & Sons, 1995.

28. **Goldberg, A.** *Object-Oriented Project Management.* Paris : Tutorials TOOLS Europe, 1991.

29. **Krasner, Glenn y Pope, Stephen.** *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80.* 53, s.l. : SIGS Publications, 1988, Journal of Object Oriented Programming, Vol. I, págs. 26-49.

30. **Szypersky, Clemens.** *Component software: beyond object-oriented programming.* USA : Addison-Wesley, 2002.

31. **Bachmann, Felix y Merson, Paulo.** *Experience Using the Web-Based Tool Wiki for Architecture Documentaction.* Pittsburgh, MA : CMU/SEI, September 2005.

32. **Vashishtha, ShriKant.** Agile way of documentation!!! *Xebia.* [En línea] Xebia Labs, 5 de Mayo de 2008. [Citado el: 7 de Septiembre de 2010.] http://blog.xebia.com/2008/05/05/agile-way-of-documentation/#more-547.

33. **Clements, Paul.** *Documentig Software Architectures: Views and Beyond.* Boston : SEI Series in Software Engineering, Addison Wesley, 2003.

34. **Hofmeister, Christine, Nord, Robert y Soni, Dilip.** *Applied Software Architecture.* Reading, MA : Addison-Wesley, 2000.

35. **Camacho, Erika, Cardeso, Fabio y Nuñez, Gabriel.** *Arquitecturas de Software - Guía de Estudio.* Departamento de Procesos y Sistemas, Universidad Simón Bolivar. Caracas, Venezuela : http://prof.usb.ve/lmendoza/Documentos/PS-6116/Guia%20Arquitectura%20v.2.pdf, 2005. Revisores: Prof. Ma. A. Pérez De Ovalles, Prof. Anna Grimán y Prof. Luis E. Mendoza.

36. **Bengtsson, PerOlof.** *Design and Evaluation of Software Architecture.* Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby. Sweden : s.n., 1999.

37. **Clements, Paul.** *A Survey of Architecture Description Languages.* Software Engineering Institute, Carnegie Mellon University. Pittsburgh, PA : SEI / CMU, 1996. www.sei.cmu.edu/library/assets/Survey_of_ADLs.pdf.

38. **Shaw, Mary y DeLine, Robert.** *Abstractions for Software Architecture and Tools to Support Them.* 4, Piscataway, NJ : IEEE Press, April de 1995, IEEE Trans. Softw. Eng., Vol. 21, págs. 314-335.

39. **Luckham, David, y otros.** *Specification and Analysis of System Architecture Using Rapide.* s.l. : Stanford University Technical Report, 1993.

40. **Bosch, Jan.** *Design & Use of Software Architectures-Adopting and Evolving a Product-line Approach.* Reading, Massachusetts : Addison-Wesley, 2000.

41. **Kazman, Rick, Carrière, S. Jeromy y Woods, Steven G.** *Toward a discipline of scenario-based architectural engineering.* 1-4, Red Bank, NJ, USA : J. C. Baltzer AG, Science Publishers, 2000, Annals of Software Engineering, Vol. 9, págs. 5-33.

42. **In, Hoh, Kazman, Rick y Olson, David.** *From Requirements Negotiation to Software Architectural Decisions.* Software Engineering Institute, Carnegie Mellon University. 2001. http://www.cin.ufpe.br/~straw01/epapers/paper13.pdf.

43. **Booch, Grady.** *Best of Booch, Designing Strategies for Object Technology.* Cambridge, UK : Cambridge University Press, 1998.

44. **Highsmith, Jim y Highsmith, James A.** *Agile Software Development Ecosystems.* Boston : Addison-Wesley, 2002.

45. **Ambler, Scott W.** Agile Architecture: Strategies for Scaling Agile Development. *Agile Modeling, Effective Practices for Modeling and Documentation.* [En línea] Ambysoft. [Citado el: 8 de Agosto de 2010.] http://www.agilemodeling.com/essays/agileArchitecture.htm.

46. **Larman, Craig.** *Agile & Iterative Development, A Manager´s Guide.* Boston, MA : Pearson Education, 2004.

47. **Morales Trujillo, Miguel Ehécatl.** *El proceso de desarrollo y mantenimiento de software propuesto por COMPETISOFT de acuerdo al proceso unificado.* Posgrado en Ciencia e Ingeniería en Computación, UNAM. México : Tesis de Maestría en Ingeniería en Computación, 2010.

48. **Lappo, Peter y Andrew, Henry C. T.** Assessing Agility. [ed.] Jutta and Baumeister, Hubert Eckstein. *Extreme Programming and Agile Processes in Software Engineering.* s.l. : Springer Berlin / Heidelberg, 2004. Lecture Notes in Computer Science.

49. **Hernández Jiménez, Diana Esmeralda.** *Administración de proyectos ágil.* Posgrado en Ciencia e Ingeniería en Computación, UNAM. México : Tesis de Maestría en Ingeniería en Computación, 2010.

50. **Humphrey, Watts S.** *Introduction to Team Software Process.* Massachusetts : Addison Wesley, 2000.

51. **Martín Sierra, Antonio J.** *Struts.* México, D.F. : Alfaomega Grupo Editor S.A. de C.V., 2008.