VNIVERSIDAD NACIONAL
AVTONOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

POSGRADO EN CIENCIA E INGENIERA DE LA COMPUTACIÓN

ARQUITECTURA DE SOFTWARE EN MÉTODOS ÁGILES

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN INGENIERÍA (COMPUTACIÓN)

P R E S E N T A:

DANIEL CORTES PICHARDO

DIRECTORA DE TESIS:
DRA. HANNA JADWIGA OKTABA

México, D.F. **2011**

DEDICATORIA

A Dios:

A Dios, por concederme la gala espiritual de la conclusión de una etapa más de mi vida en este valle terrenal.

A mi madre:

La Dra. Bertha Pichardo Osorio, quien gracias a su ejemplo y meritos, que pocas personas han podido lograr, me ha inspirado en la culminación de una etapa más de mi vida espiritual y profesional en este mundo.

A mis hermanos:

Nancy y Hugo, a quienes amo y respeto profundamente.

A mi novia:

Angeles, el amor de mi vida y de mi corazón.

AGRADECIMIENTOS

A la Dra. Hanna Oktaba por dirigir esta tesis.

A los miembros del jurado: M. en C Guadalupe Ibargüengoitia, M. en C Gustavo Marquez y la Dra. María del Pilar Ángeles, por la revisión de este trabajo.

Al M. en C. Gustavo Adolfo Arellano Sandoval y al M.I. Alejandro Alberto Ramírez Ramos, por su amistad, y sus valiosos comentarios para este trabajo en el área de arquitectura de software.

Índice

1 Introducción

En la actualidad, los métodos ágiles han estado ganando gran popularidad como una alternativa a los métodos tradicionales de desarrollo de software. Los métodos ágiles ofrecen un mecanismo para reducir los costos e incrementar la habilidad de manejar los cambios en condiciones dinámicas del mercado. Sin embargo, hay una preocupación significante sobre el rol e importancia de los asuntos relacionados con la arquitectura de software de un sistema que está siendo desarrollado bajo un enfoque ágil [Babar2009], [Falessi]. Para esto, una posible dicotomía, un oxímoron o el intento de mezclar agua y aceite, son las percepciones que se tienen entre el enfoque ágil y los centrados en la arquitectura; pero, ¿realmente no existe una brecha entre ellos? o ¿claramente la arquitectura de software y sus prácticas no tiene cavidad en los ambientes ágiles? En el presente trabajo se exploran los caminos que han tomado los métodos ágiles con respecto a la arquitectura, así como también, la importancia de unir las prácticas de análisis, diseño y evaluación de la arquitectura de software en ambientes ágiles.

1.1 Planteamiento del problema

A pesar de la gran popularidad de los métodos ágiles, los defensores de los métodos tradicionales dudan de la escalabilidad de un sistema que sea construido mediante algún enfoque de desarrollo que no ponga la suficiente atención en la arquitectura, además, las compañías, donde las prácticas de arquitectura de software están bien cimentadas, suelen ver a las prácticas ágiles como prácticas de aficionados, no probadas y limitadas a muy pequeños sistemas sociotécnicos basados en web [Abrahamsson], encontrando las siguientes desventajas desde el punto de vista de la arquitectura [Babar2009]:

- Es muy arriesgado utilizar un enfoque de desarrollo ágil para los nuevos proyectos, en especial, cuando las posibles soluciones no se comprenden muy bien.
- No hay consideración sobre mejores alternativas para el diseño de la arquitectura de un sistema.
- No hay atención en los atributos de calidad del sistema, excepto en algunos casos de desempeño.

Por lo anterior, se dice que los métodos ágiles se mueven lejos del formalismo y rigor, sin ningún tipo de proceso sistemático para el crecimiento y cambios de la arquitectura, por lo que el crecimiento gradual de la arquitectura de software tiene muchas incertidumbres desde el punto de vista de la corrección; lo que a menudo resulta en arquitecturas deterioradas [Babar2009].

Es claro que las prácticas del diseño arquitectónico no son bien vistas por los proponentes de los métodos ágiles, creando de esta manera, una discrepancia entre el término ágil y arquitectura,

donde esta discrepancia puede radicar en el hecho de que un método ágil es adaptativo, es decir, decidir en el último momento o cuando ocurra el cambio, mientras que la arquitectura de software es previsora, es decir, la planeación con demasiada anticipación.

Actualmente resulta difícil encontrar documentación sobre cómo definir una arquitectura en un proyecto de desarrollo de software guiado por un método ágil. La documentación disponible en muchos casos es sobre algunos lineamientos mínimos, como el caso de la noción de *System Metaphor* de *Extreme Programming* [BeckKent]. Originalmente, la arquitectura no era tenida en cuenta en los métodos ágiles, y si se consideraba, se hacía de manera superficial; además de que las prácticas arquitectónicas no son muy conocidas por los practicantes de los métodos ágiles [Nord]. Con el tiempo, el rol de la arquitectura ha venido surgiendo en varios trabajos [Farhan], [Madison 00], [Ethan Hadar] , [Abrahamsson], [Faber], [Nord], [Babar2009], [Kanwal], [IshamM2008], [Falessi], [RobertL], los cuales hablan sobre métodos y marcos híbridos que permiten manejar las prácticas arquitectónicas dentro de un enfoque de desarrollo ágil. Estas propuestas tratan de hacer un balance entre: el enfoque tradicional de desarrollo de software y los métodos ágiles.

1.2 Objetivo

Integrar las prácticas del diseño arquitectónico y el rol del arquitecto de software dentro de los métodos ágiles.

1.3 Objetivos específicos

Las prácticas y actividades que se emplean para el análisis, diseño y evaluación de una arquitectura de software son tan importantes dentro de los métodos tradicionales como en los métodos ágiles, por lo tanto, los objetivos específicos de esta tesis son:

- Obtener las recopilaciones bibliográficas de las prácticas relacionadas con la arquitectura de software, utilizadas dentro de los métodos ágiles.
- Proponer una definición de las competencias que deberá cubrir un arquitecto ágil.
- Proponer un marco híbrido para la integración de las prácticas arquitectónicas dentro de los métodos ágiles.
- Lograr la validación de la propuesta por parte de dos expertos en métodos ágiles.

1.4 Contribución y relevancia

La contribución y relevancia que aporta este trabajo se desglosa en los siguientes puntos:

- Es un tema novedoso del cual se hace una síntesis de algunos de los artículos más importantes hasta la fecha.
- Se muestra cómo implementar prácticas tradicionales del proceso de Arquitectura de Software con un enfoque ágil.
- Se aportarán lineamientos para la implementación de las prácticas del proceso de arquitectura de manera ágil.
- Se discutirá cómo deben aplicarse estos lineamientos, y en que parte del proceso deben intervenir.
- Se proponen las competencias que debe cumplir un arquitecto ágil.
- Se discutirá como cuidar los atributos de calidad en arquitectura.
- Se sentarán las bases para desarrollar un proceso de arquitectura ágil.

1.5 Panorama de la tesis

La tesis está conformada por seis capítulos.

Capítulo I

Se expone el planteamiento del problema, los objetivos de la tesis y la organización de este documento.

Capítulo II

En este apartado se introduce al lector sobre el tema de la arquitectura de software, se hace una síntesis de la historia, conceptos básicos y el proceso de diseño de la arquitectura de un sistema.

Capítulo III

En este capítulo se introduce al lector sobre el tema de los métodos ágiles, analizando principalmente dos de sus máximos representantes; *Scrum* y *Extreme Programming.*

Capítulo IV

En este capítulo se presenta el resultado de la investigación sistemática sobre el rol, prácticas, actividades y productos de trabajo de la arquitectura que son utilizados dentro de los métodos ágiles.

Capítulo V

Con los resultados de la investigación sistemática, expuestos en el capítulo IV, en este capítulo se presenta una propuesta de un marco híbrido basado en *Scrum, Extreme Programming* y prácticas arquitectónicas.

Capítulo VI

Conclusiones y trabajo a futuro.

2 Arquitectura de software

En esta sección se revisan los conceptos importantes que envuelven a la Arquitectura de Software y que son de interés para darle seguimiento al presente trabajo. Se comienza con una breve introducción histórica, donde se mencionan algunos hechos importantes que dieron lugar a la Arquitecturas de software como temas de estudio, así como también, se citan algunos precursores de la Arquitectura de software. También, se revisarán algunas definiciones sobre la Arquitectura de software, con el fin de encontrar la que más se adapte al presente estudio. Por último, se revisara el proceso tradicional del diseño de la arquitectura de software.

2.1 Introducción histórica

El estudio de la arquitectura de software se enfoca en gran medida en el estudio de la estructura de software, cuyo origen se remonta a Edsger Dijkstra en el año 1968, donde habla acerca de la importancia de preocuparse sobre el establecimiento de una división y estructuración correcta de los sistemas de software antes de comenzar a programar, con el fin de producir un resultado correcto [PaulCLindaNorth96]. Después es David Parnas quien retoma la apreciación de Dijkstra y contribuye con sus ideas sobre módulos que oculten información, estructuras de software y familias de programas, entendiendo por familia de programas al conjunto de programas para los cuales es rentable o útil considerarlos como un grupo. Más tarde en la conferencia de la OTAN en 1969, P.I.Sharp, formula un conjunto de apreciaciones importantes, comentando sobre las ideas de Dijkstra y estableciendo la diferencia que existe entre la ingeniería de software y la arquitectura de software. Hasta la década de los 90´s, el término "arquitectura de software" se vio de distintas maneras, sobretodo se ve muy ligado al diseño. En esta década se hablaba de un nivel de abstracción, sin embargo aún no estaba en su lugar los elementos que permitieran reclamar la necesidad de una disciplina y una profesión particular.

En el *Sofware Engineering Institute* (SEI), se reconoce a "*Studying Software Architecture Through Design Spaces and Rules*", de Thomas G. Lane, como el primer libro publicado por el SEI, sobre el tema de la Arquitecturas de Software escrito en 1990. En este libro, Lane establece una definición de Arquitectura de Software, basada en el concepto expuesto por Mary Shaw, en el año de 1989, en su libro "Larger Scale Systems Require Higher-Level Abstractions" publicado por la IEEE Computer Society. En su libro Lane define la Arquitectura de Software como:

"*Arquitectura de software es el estudio de la estructura a gran escala y el rendimiento de los sistemas de software. Aspectos importantes de la arquitectura de un sistema incluye la división de*

funciones entre los módulos del sistema, los medios de comunicación entre módulos, y la representación de la información compartida [Lane1990]."

Con estos aportes, se ve que hay una búsqueda hacia un modelo con el que se pueda estructurar el software, dando lugar al diseño de un conjunto de modelos de dominios, basados en diseños genéricos, como son DSSA (Domain-Specific Software Architectures) elaborado por Mettala y Graham, en el año 1992.

Después, con el lanzamiento del libro "An Introduction to Software Architecture", de Mary Shaw y David Garlan [GarlanShaw1994], en el año 1994, donde se plantea, que debido al aumento del tamaño y complejidad de los productos de software, el problema principal ya no radicaba en la complejidad de los algoritmos y las estructuras de datos que se empleaban en los sistemas, sino que se dirige a la organización de los componentes que conforman el sistema, introduciendo de esta manera, la necesidad de la creación de la arquitectura de software, como disciplina científica.

El objetivo de estudio de la arquitectura de software es la determinación de un conjunto de paradigmas que establezcan una organización del sistema a alto nivel, la interrelación entre los distintos componentes que lo conforman y los principios que orientan su diseño y evolución.

En el libro "An Introduction to Software Architecture" se introduce por primera vez el término "*architectural styles*", y se definen algunos de ellos, entre los que se encuentran: Tubería y filtros, Abstracción de datos y organización orientada a objetos, Repositorio, Arquitectura en Capas, Arquitectura basada en eventos, *Table Driven Interpreters*, entre otros.

Hasta ese momento el tema del rol de arquitecto de software dentro del proceso y desarrollo del software, no se expone a la luz, sin embargo, en el libro, titulado: "Pasado, presente y futuro", se expone un grupo de áreas de interés para el estudio posterior de la disciplina, dentro de las que se destaca, "lograr un mejor entendimiento del rol del arquitecto en el ciclo de vida del proceso."

En este mismo año, 1994, se realiza otro acontecimiento arquitectónico relevante. Mary Shaw y David Garlan, lanzan el concepto de Lenguajes de Descripción Arquitectónica (ADLs), en su libro "Characteristics of Higher Level Languages for Software Architecture". En el estudio se señala las alternativas que hasta el momento se poseen para la definición de la arquitectura de software de un sistema. En primer lugar, a través de la modularización de las herramientas existentes de programación y de los módulos de conexión entre ellas y, en segundo lugar, describe sus diseños usando diagramas informales y frases idiomáticas [GarlanShaw011994]. A partir de estas reflexiones, definen un conjunto de regularidades y propiedades específicas, que constituyeron las bases de los ADLs.

Después de los ADLs, la disciplina ha ido creciendo grandemente, destacándose un conjunto de acontecimientos como el libro "*Coming Attractions in Software Architecture*", de Paul Clements, en el año 1996, donde define cinco temas fundamentales en torno de los cuales se agrupa la disciplina; diseño o selección de la arquitectura, representación de la arquitectura, evaluación y análisis, etc.

Otro suceso relevante fue el lanzamiento de la tesis de Roy Fielding, en el año 2000, en la cual presenta el modelo REST, el cual establece definitivamente el tema de las tecnologías de Internet y los modelos orientados a servicios y recursos en el centro de las preocupaciones de la disciplina.

En el mismo año se publica la versión definitiva de la recomendación IEEE Std 1471, que procura homogeneizar y ordenar la nomenclatura de descripción arquitectónica y homologar los estilos como un modelo fundamental de representación conceptual.

2.2 Definiciones de arquitectura de software

En esta sección se explorarán algunas definiciones de Arquitectura de Software con el fin de encontrar sus similitudes y diferencias. Es de suma importancia mencionar que actualmente existe una gran cantidad de definiciones de arquitectura de software, llegando incluso a ser grandes compendios de definiciones alternativas o contrapuestas, como ejemplo está la colección de definiciones que posee SEI, en su página [SEI2010].

A continuación se exponen algunas definiciones sobre arquitectura de software.

1. IEEE Std 1471 define a la arquitectura de software como "la organización fundamental de un sistema encargada de sus componentes, las relaciones entre ellos y el ambiente o entorno y los principios que orientan su diseño y evolución" [IEEE 1471 2000].
2. Garlan y Shaw definen la arquitectura de software como "la descripción de los elementos que comprenden un sistema, la interacción y patrones de estos elementos, los principios que guían su composición, y las restricciones de esos elementos" [GarlanShaw1994].
3. Paul Clements 1996: "La arquitectura de software es a grandes rasgos, una vista del sistema que incluye los componentes principales del mismo, la conducta de estos componentes según se la percibe desde el resto del sistema y las formas en que los componentes interactúan y se coordinan para alcanzar la misión del sistema. La vista arquitectónica es una vista abstracta, aportando el más alto nivel de comprensión y la supresión o diferimiento del detalle inherente a la mayor parte de las abstracciones".
4. La arquitectura de software de un programa o 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 [SEI2010].

De las definiciones anteriores se menciona el trabajo dinámico de estipulación de la arquitectura de software dentro del proceso de ingeniería o el diseño mencionando su lugar en el ciclo de vida del software, la configuración estática de los sistemas de software contemplada desde un elevado nivel de abstracción y la caracterización de la disciplina que se ocupa de uno de esos dos asuntos, o de ambos.

El presente trabajo se basa en el estándar IEEE 1471 [IEEE 1471 2000]. A continuación se explicarán algunos conceptos importantes que articulan a la definición de arquitectura de software dada por este estándar.

Sistema: es una colección de componentes organizados para llevar a cabo una función específica o un conjunto de funciones. El termino sistema abarca aplicaciones individuales, sistemas en el sentido tradicional, subsistemas, sistemas de sistemas, líneas de productos, familias de productos, empresas y otras agrupaciones de interés. Un sistema existe para cumplir con una o más misiones en su entorno [IEEE 1471 2000].

Entorno (ambiente): también conocido como contexto, determina la configuración y circunstancias de desarrollo, operacional, político y otras que puedan influir en el sistema [IEEE 1471 2000].

Misión: es la utilización o explotación a que se destina un sistema, mediante el cual, uno o más *stakeholders* conocen un conjunto de objetivos [IEEE 1471 2000].

***Stakeholders* del sistema:** es un individuo, equipo u organización con intereses en un área o preocupación relativa al sistema [IEEE 1471 2000].

El termino componente no está definido en este estándar, pero una definición puede encontrarse en la sección 2.3.1.

2.3 Conceptos fundamentales de Arquitectura de Software

En esta sección se revisan algunos de los conceptos y principios esenciales que articulan a la arquitectura de software.

2.3.1 Componente de software

Podemos ver a un componente de software como un elemento que encapsula el procesamiento y los datos en la arquitectura de un sistema, que además es independiente de todo lo externo a éste. Como definición de componente de software se tomará la de [RichardN2010].

Definición: "Un componente de software es una entidad arquitectónica que encapsula un conjunto de funcionalidades de un sistema o dato, restringe el acceso a través de una interfaz definida explícitamente, y tiene definido explícitamente dependencias que necesite en su contexto de ejecución."

Las propiedades características de un componente son las siguientes [ClementsSzyperski]:

- Es una unidad de despliegue independiente
- Es una unidad compuesta por terceros
- No tiene estado observable externamente

De lo anterior concluimos que un componente presupone un contexto de la arquitectura definido por sus interfaces y que éste también puede ser reemplazable. En la mayoría de los casos se

entiende por componente de software, objetos pre compilados con interfaces bien definidas listos para ser utilizados en diferentes ambientes.

2.3.2 Conector

El conector en una arquitectura de software es redefinido durante el proceso de diseño y es influenciado por el entorno de despliegue del proyecto. En la forma más abstracta, un conector indica la necesidad que tiene un elemento de enviar un mensaje a otro elemento durante la ejecución de un sistema y que éste último le regrese un mensaje. Los conectores son clasificados de acuerdo con varios atributos, incluyendo modo de sincronización, iniciador, tipo de implementación, intervalo de tiempo activo, información que transmite, entre otros.

2.3.3 Configuración Arquitectónica

Los componentes y los conectores están compuestos en una manera específica en la arquitectura de un sistema dado a fin de alcanzar el objetivo del sistema. Esta composición representa la configuración del sistema, también conocida como topología. Se tomará la definición de configuración de [RichardN2010]

Una configuración arquitectónica es un conjunto de asociaciones específicas entre los componentes y conectores de software de la arquitectura del sistema.

Una configuración puede ser representada como una gráfica en donde los nodos representan componentes y conectores, cuyas aristas representan sus asociaciones (topología o interconectividad).

2.3.4 Estilos arquitectónicos

Los estilos arquitectónicos, que en ocasiones también son llamados “patrones de arquitectura”, abstraen las propiedades comunes de una familia de diseños similares. De esta manera, un estilo arquitectónico es un conjunto de reglas, restricciones y patrones que muestran como estructurar un sistema dentro de un conjunto de elementos y conectores. Se tomará como definición de estilo arquitectónico la dada en [RichardN2010]:

Un estilo arquitectónico es una colección de decisiones de diseños arquitectónicos que son aplicables a un contexto de desarrollo dado, limitan las decisiones de diseño arquitectónico que son específicos de un determinado sistema dentro de un contexto para obtener cualidades beneficiosas en cada sistema resultante.

A continuación se exponen los componentes principales de un estilo arquitectónico [KaiQian]:

- Elementos que realizan funciones requeridas por un sistema.
- Conectores que establecen una comunicación, coordinación y cooperación entre elementos.
- Restricciones que definen como los elementos pueden ser integrados para formar el sistema.
- Atributos que describen las ventajas y desventajas de la estructura elegida.

La descripción de un estilo arquitectónico se puede definir utilizando el lenguaje natural o empleando diagramas, pero lo mejor es hacerlo en un lenguaje de descripción arquitectónica (ADL) o en algún lenguaje formal de especificación como UML.

Es importante mencionar que a diferencia de los patrones de diseño, que en la actualidad son muchos, los estilos se ordenan en unos pocos. Sobre las arquitecturas complejas o compuestas, éstas resultan del agregado o la composición de los estilos básicos y son conocidas como arquitecturas heterogéneas. La Figura 2-1 muestra las familias de los estilos arquitectónicos básicos [GarlanShaw1994].

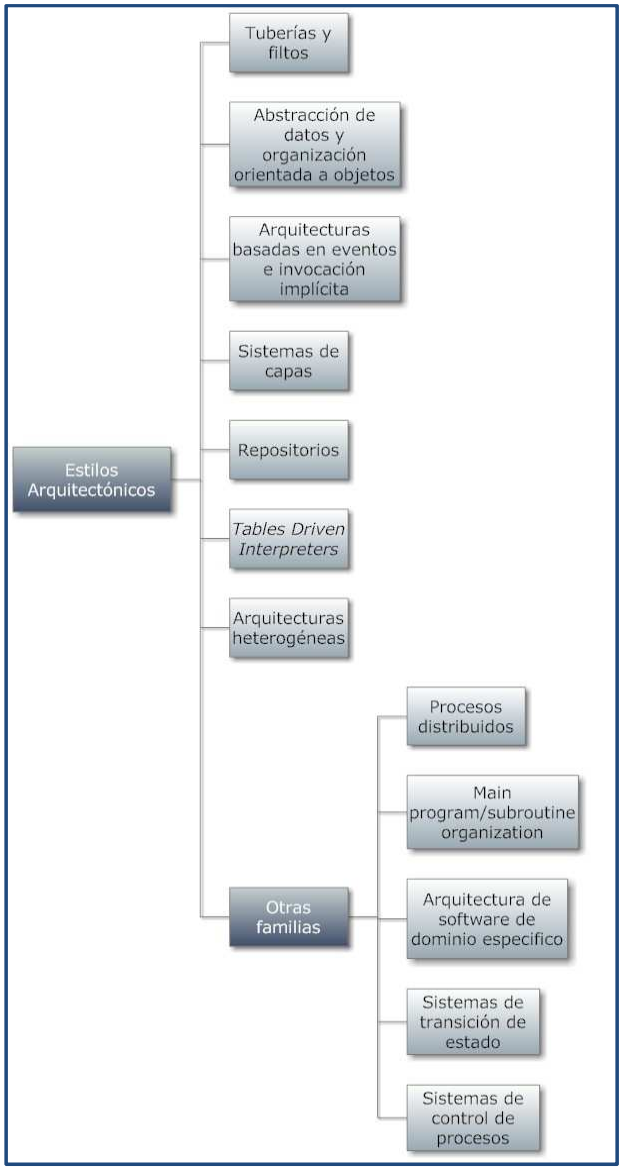

Figura 2-1 Familias de estilos arquitectónicos

En lo que resta de esta sección, se analizará cada uno de los estilos arquitectónicos expuestos anteriormente y más tarde en la sección 2.5.3 se explicará la importancia de conocer las propiedades que cada uno de estos estilos engloban, desde el punto de vista de los atributos de calidad que cada estilo arquitectónico cuida.

2.3.4.1 Tuberías y filtros

En un estilo arquitectónico de tuberías y filtros, cada componente tiene un conjunto de entradas y un conjunto de salidas. Un componente lee los flujos de datos en sus entradas y produce un flujo de datos en sus salidas, entregando una instancia completa del resultado en una solicitud estándar. Esto se logra generalmente mediante la aplicación de una transformación local a los

flujos de entrada y realizando una computación incremental, de tal manera que la salida comienza antes de haberse consumido todo el flujo de entrada. Por lo tanto los componentes se llaman "filtros". Los conectores de este estilo sirven como conductos para los flujos que transmitan los resultados de un filtro a las entradas de otro. Por lo tanto, los conectores se denominan tubos. La Figura 2-2 ilustra este estilo arquitectónico.

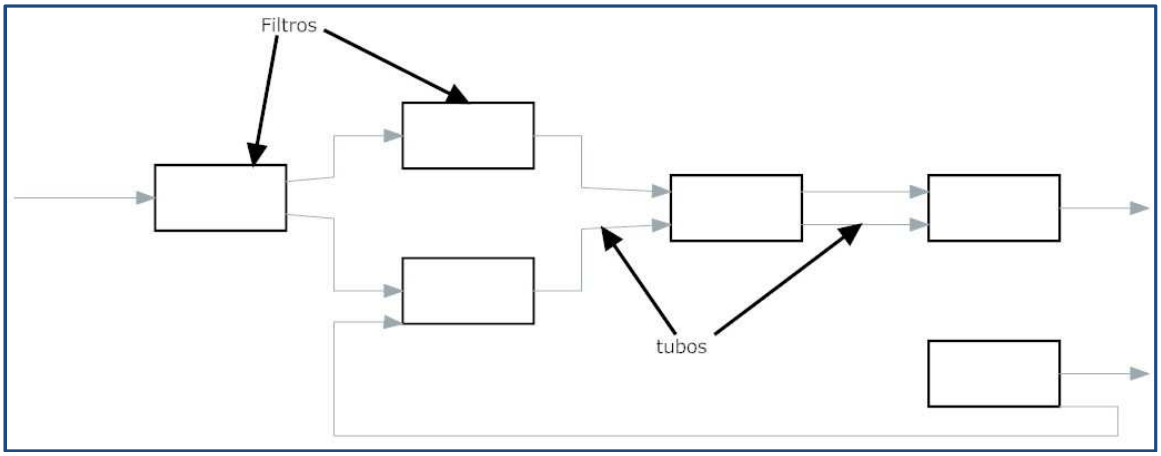

Figura 2-2 Ejemplo de tuberías y filtros

2.3.4.2 Abstracción de datos y organización orientada a objetos

En este estilo arquitectónico, la representación de datos y sus operaciones primitivas asociadas, es encapsulada en un tipo de dato abstracto u objeto. Los componentes de este estilo son los objetos o instancias de un tipo de dato abstracto. Los objetos son ejemplos de un tipo de componente llamado gestor, porque es el responsable de preservar la integridad de un recurso, entendiéndose como integridad de un recurso a su implementación. Los objetos interactúan a través de la invocación de funciones o procedimientos. Los dos aspectos importantes de este estilo son:

a) Que un objeto es responsable de preservar la integridad de su representación (su implementación).
b) La representación se oculta para otros objetos

La Figura 2-3 ilustra este estilo arquitectónico.

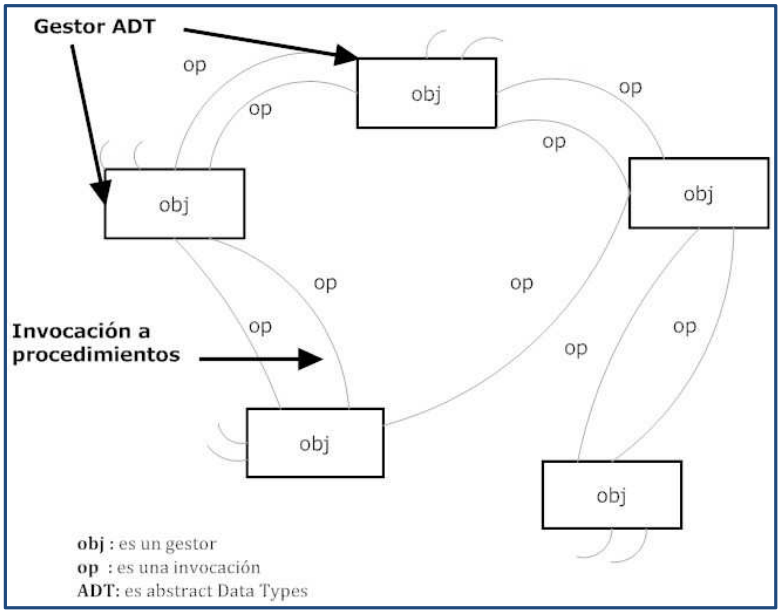

Figura 2-3 Ejemplo de abstracción de datos y organización orientada a objetos

2.3.4.3 Arquitecturas basadas en eventos, invocación implícita

La idea detrás de la invocación implícita es que en lugar de invocar un procedimiento directamente, un componente puede anunciar uno o más eventos. Otros componentes del sistema pueden registrar un interés en un evento mediante el hecho de asociar un procedimiento con un evento. Cuanto el evento es anunciado el sistema por si mismo invoca todos los procedimientos que han sido registrados para el evento. De esta manera un evento anunciado implícitamente causa la invocación del procedimiento en otros módulos. Como ejemplo en el campo de los sistemas, herramientas como los editores que permiten escribir programas en algún lenguaje de alto nivel tienen la posibilidad de permitirle al programador depurar (*debbuger)* su código. Cuando un *debbuger* se detiene en el *breakpoint,* anuncia un evento que permite al sistema automáticamente invocar los métodos en estas herramientas registrados. Estos métodos pueden hacer que el editor se desplace a la línea de código fuente apropiada. En este esquema, el *debugger* simplemente anuncia un evento, pero no sabe que otras herramientas están atentas con este evento, o que harán cuando el evento sea anunciado.

Desde el punto de vista de la arquitectura, los componentes en un estilo de invocación implícita son módulos, cuyas interfaces proveen tanto una colección de procedimientos como un conjunto de eventos. Un componente puede registrar algunos de sus procedimientos con eventos del sistema, causando que estos procedimientos sean invocados cuando estos eventos sean anunciados en tiempo de ejecución. Los conectores en un sistema de invocación implícita incluyen el llamado a procedimientos tradicionales y vínculos entre los eventos anunciados y las llamadas a los procedimientos.

2.3.4.4 Sistemas de capas

Un sistema de capas se organiza jerárquicamente, cada capa provee servicio a una capa de arriba de ella y sirve como un cliente a la capa de debajo de ella. En algunos sistemas de capas, las capas internas se encuentran ocultas de todas excepto a la capa externa adyacente y ésta sólo puede acceder a ciertas funciones cuidadosamente elegidas para exportar. De esta manera en estos sistemas los componentes implementan una maquina virtual hacia alguna capa en la jerarquía. Los conectores están definidos mediante protocolos que determinan la manera de cómo las capas interactuarán. Las restricciones topológicas incluyen interacciones limitadas hacia capas adyacentes. La Figura 2-4 ilustra este estilo arquitectónico.

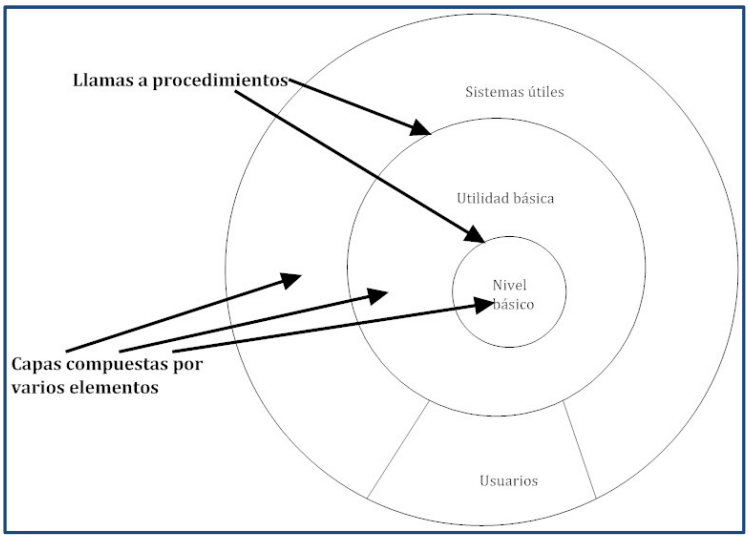

Figura 2-4 Ejemplo de sistemas de capas

2.3.4.5 Repositorios

En este estilo hay dos tipos de componentes: una estructura de datos central que representa el estado actual y una colección de componentes independientes que operan en un depósito de datos central. Las interacciones entre el repositorio y sus componentes externos pueden variar significativamente entre el sistema.

Como parte central en esta arquitectura aparece un almacén de datos el cual es accedido frecuentemente por otros componentes que actualizan, añaden y borran dichos componentes. El software cliente accede a un repositorio vacío. Existen dos tipos de repositorios los cuales son:

- **Repositorio pasivo:** el cliente software accede a los datos independientemente de cualquier cambio a los datos o a las acciones de otros clientes software.

- **Repositorio activo o *blackboard*:** el repositorio envía información a los clientes cuando los datos de interés cambian, siendo por lo tanto un ente activo.

La Figura 2-5 ilustra una simple vista de una arquitectura *blackboard.*

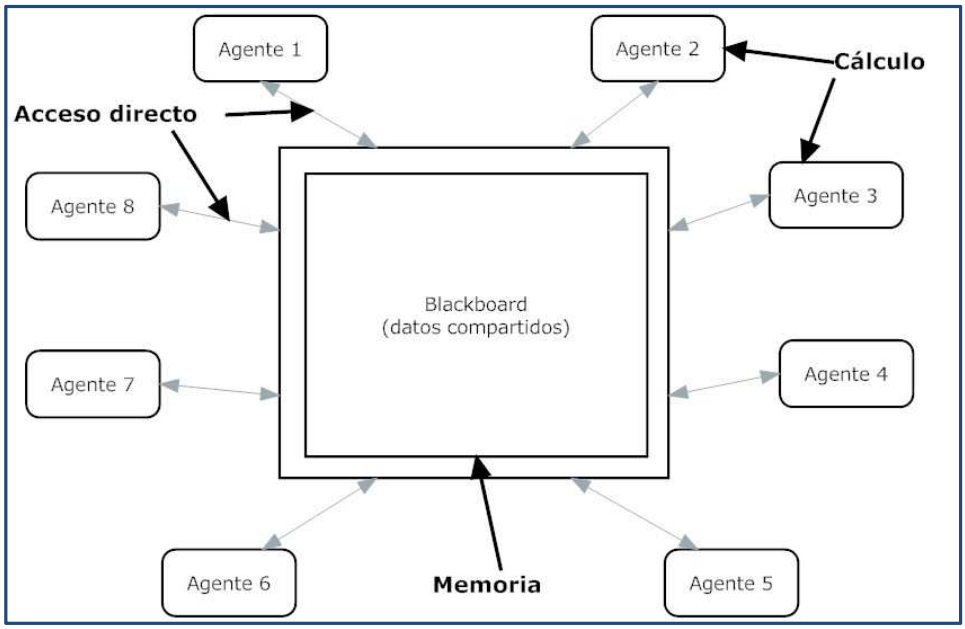

Figura 2-5 Ejemplo de una arquitectura Blackboard

La palabra *blackboard* o pizarra viene de la idea de un salón de clases. El profesor y el alumno pueden compartir datos cuando resuelven un problema mediante el uso de un pizarrón. Los estudiantes y los profesores juegan el rol de agentes para contribuir a la resolución del problema. Todos ellos pueden trabajar en paralelo e independientemente, tratando de encontrar la mejor solución. La idea de la arquitectura de pizarra es similar a la del pizarrón del salón de clases usado para resolver problemas. El sistema completo está compuesto de dos divisiones principales. Una división, llamada *blackboard* es usada para almacenar datos, mientras que la otra división, llamada *Knowledge sources,* almacena datos de un dominio especifico del sistema. También puede haber una tercera división, llamada control, el cual se usa para iniciar el *blackboard* y el *Knowledge sources,* además de tomar el rol de supervisar todo el control.

2.3.4.6 Tables Driven Interpreters

Un intérprete incluye un pseudo programa siendo interpretado y el motor de interpretación en sí mismo. El pseudo programa incluye el programa y la analogía de la interpretación de su estado de ejecución también conocido como registro de activación. El motor de interpretación incluye tanto

la definición del el interprete como el estado actual de su ejecución. De esta manera un intérprete generalmente tiene cuatro componentes: un motor de interpretación para hacer algo, una memoria que contiene el pseudocódigo que será interpretado, una representación del estado de control del motor de interpretación y una representación del estado actual del programa que está siendo simulado. La Figura 2-6 ilustra este estilo arquitectónico.

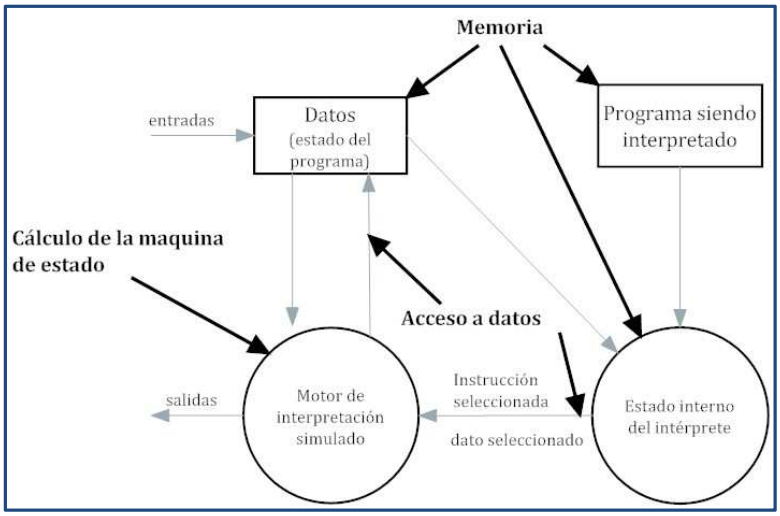

Figura 2-6 Ejemplo de *tables driven interpreters*

2.3.4.7 Otras familias de arquitecturas:

Como se ha mencionado anteriormente existen numerosos estilos de arquitecturas y patrones, donde unos son muy generales y otros son específicos en un dominio particular. En el presenta trabajo solo se mencionarán las categorías o familias en los que estos estilos pueden aparecer en [GarlanShaw1994].

2.3.4.7.1 Procesos distribuidos

Un sistema distribuido es una colección de dispositivos computacionales y de almacenamiento conectados a través de una red de comunicación. Para este tipo de sistemas los datos, software y usuario se encuentran distribuidos. En un sistema distribuido los componentes o subsistemas se comunican entre ellos usando un numero de métodos que incluyen el envió de mensajes, llamadas a procedimientos remotos e invocación de métodos remotos.

Los sistemas distribuidos se pueden caracterizar principalmente por sus características topológicas, entre las que encontramos la configuración estrella y anillo. Algunas otras se caracterizan en términos del tipo de protocolo que es usado para comunicarse.

Un ejemplo de una forma comun de la arquitectura de un sistema distribuido es la arquitectura cliente servidor [KaiQian]. En la arquitectura cliente servidor, un servidor representa un proceso

que provee un servicio a otros procesos conocidos como clientes, el servidor no conoce por adelantado la identidad o número de clientes que accederán en tiempo de ejecución, por lo que deberá estar preparado para ello. Por su parte, los clientes, conocen la identidad del servidor o la averiguan a través de otros servidores y acceden al servidor utilizando llamadas a procedimientos remotos.

2.3.4.7.2 Main program/subroutine organization

La principal manera de organizar un sistema es mediante el lenguaje de programación en el cual está escrito. Cuando el lenguaje no soporta modularización entonces se suele organizar en un sistema que consta de un programa principal (*main program*) y un conjunto de subrutinas. En este caso el programa principal es el encargado de controlar las subrutinas que suelen proveer un bucle de control para iterar a través de las subrutinas en algún orden específico.

2.3.4.7.3 Arquitectura de software de dominio especifico

Estas arquitecturas proveen una estructura organizacional que se ajustan a una familia de aplicaciones, tales como aviónica, comando y control, o incluso para sistemas de gestión de vehículos. Una ventaja de especializar la arquitectura en un dominio es que hace posible incrementar la descripción de una estructura y que no quede tan general como en los estilos comunes, aunque, hay que notar que esto repercutirá en la flexibilidad del sistema. De hecho, en muchos casos la arquitectura está lo suficientemente restringida para que un sistema ejecutable pueda ser generado automáticamente a partir la descripción arquitectónica.

2.3.4.7.4 Sistemas de transición de estado

La arquitectura de sistemas de transición de estado es empleada por los sistemas reactivos, los cuales son definidos en términos de un conjunto de estados y un conjunto de transiciones que mueven a un sistema de un estado a otro.

2.3.4.7.5 Sistemas de control de procesos

Los sistemas de control intentan proveer un control dinámico de un ambiente físico, suelen ser organizados como sistemas de control de procesos. Estos sistemas son caracterizados como sistemas retroalimentados, en los cuales las entradas provienen de un sensor que es usado por el sistema de control del proceso para determinar el conjunto de salidas que producirá un nuevo estado en el ambiente.

Arquitecturas heterogéneas

Los estilos que se han visto hasta ahora son considerados como estilos arquitectónicos puros. La mayoría de los sistemas típicamente están compuestos por una combinación de varios estilos y el resultado es a lo que se le conoce como arquitectura heterogénea.

Existen diferentes maneras en las cuales un estilo arquitectónico puede ser combinado. Una manera es mediante la jerarquía, esto es, un componente de un sistema diseñado con un estilo arquitectónico puede tener una estructura interna que fue desarrollada con un estilo arquitectónico diferente. La segunda manera es mediante el permitir un solo componente para usar una mezcla de conectores arquitectónicos, por ejemplo, un componente puede acceder a un repositorio a través de su interface, pero interactúan a través de tuberías con otros componentes en un sistema y aceptan el control de información a través de otra de sus interfaces. Una tercera manera es elaborar un completo nivel de descripción de arquitectura con un estilo arquitectónico completamente diferente.

Para una descripción más detallada de estos estilos arquitectónicos y otros, se recomienda revisar las referencias [GarlanShaw1994] y [KaiQian].

2.3.5 Patrón arquitectónico

Un estilo arquitectónico proporciona decisiones de diseño generales, mientras que un patrón arquitectónico proporciona un conjunto de decisiones de diseño específicas que han sido identificadas como efectivas para organizar ciertas clases de un sistema de software o, más típicamente, subsistemas específicos. Como definición de patrón de arquitectura se tomará el dado en [RichardN2010]:

Un patrón arquitectónico es una colección de decisiones de diseños arquitectónicos que son aplicables a un problema de diseño recurrente, que se pueden parametrizar para tener en cuenta en diferentes contextos de desarrollo de software en los cuales el problema aparece.

Se puede ver a simple vista que esta definición es similar a la dada en estilo arquitectónico, de hecho, las dos nociones son similares y no siempre es posible identificar los límites entre ellos. Sin embargo, en general los estilos y los patrones difieren en al menos tres maneras importantes [RichardN2010]:

1. **Alcance**: Un estilo arquitectónico aplica a un contexto de desarrollo, mientras que un patrón arquitectónico aplica a un problema de diseño específico. Un problema es significantemente más concreto que un contexto. En otras palabras, los estilos arquitectónicos son estratégicos, mientras que los patrones son herramientas tácticas de diseño.

2. **Abstracción**: Un estilo ayuda a limitar las decisiones del diseño arquitectónico hechos sobre un sistema. Sin embargo, un estilo requiere una interpretación humana de acuerdo con un diseño relacionado que capture las características generales del contexto de desarrollo para el problema de diseño relacionado con un sistema específico.

3. **Relación**: los patrones son parámetros para tener en cuenta los diferentes contextos en los cuales un problema dado aparece, esto significa que un solo patrón podría ser aplicado a sistema diseñado de acuerdo con las directrices de múltiples estilos. Por otro lado, un sistema diseñado de acuerdo con las reglas de un solo estilo puede envolver el uso de muchos patrones.

2.3.6 Modelo Arquitectónico

La arquitectura de software de un sistema se captura en un modelo arquitectónico, usando una notación de modelado particular.

Un modelo arquitectónico es un artefacto que captura algunas o todas las decisiones de diseño que comprenden una arquitectura de un sistema. El modelado de una arquitectura es la consideración y documentación de esas decisiones de diseño [RichardN2010].

Un modelo es el resultado de la actividad de modelado, el cual constituye una porción significante de la responsabilidad del arquitecto de software. Un sistema puede tener muchos modelos asociados a él. Los modelos pueden variar en cuanto a detalle. Por lo anterior, es importante definir una notación de modelado de arquitectura.

La notación de modelado arquitectónico es un lenguaje que captura decisiones de diseño [RichardN2010].

La notación para modelar arquitecturas de software es conocida como "*architecture description languages" (ADL).* El ADL puede ser textual o gráfico, informal, semi informal, o formal, de dominio especifico o de propósito general, propietario o estandarizado (en la siguiente sección se detalla mas sobre *ADL's)*.

El modelo arquitectónico es usado como la base para la mayoría de las demás actividades en los procesos de desarrollo de software basados en la arquitectura, tales como el análisis, implementación del sistema, despliegue entre otros.

2.3.7 Lenguaje de descripción arquitectónica

El lenguaje de descripción de arquitectura, proporciona un medio para describir formalmente un sistema de software a un alto nivel de abstracción, captura la estructura de alto nivel y de

comportamiento del sistema. Los ADLs se remontan a los lenguajes de interconexión de módulos (MIL) de la década de 1970, pero se han comenzado a desarrollar con su denominación actual a partir de 1992 o 1993. La definición más simple es la de [AlexanderWolf] dada en 1997 que define un ADL como una entidad que consiste de cuatro partes: componentes, conectores, configuraciones y restricciones.

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

Dentro de los ADLs fundamentales de la arquitectura de software están: Acme-Armani, ADML, Aesop, ArTek, CHAM, Darwin, Jacal, Weaves, Wright, UniCon entre otros.

2.3.8 Abstracción

El concepto de abstracción ha sufrido también diversas acepciones, con un núcleo de significados común. Las diferencias en el uso del concepto de abstracción ayudan también a identificar las diversas corrientes en el seno de la arquitectura de software. La definición que proporciona Grady Booch, por ejemplo, revela que el autor identifica la abstracción arquitectónica con el encapsulamiento propio de la tecnología de objetos: “Una abstracción escribe Booch, denota las características esenciales de un objeto que lo distinguen de otras clases de objeto y provee de este modo delimitaciones conceptuales bien definidas, relativas a la perspectiva del observador”.

Por último, la abstracción siempre conlleva una heurística positiva al lado de una negación. Tanto para la IEEE como para Rumbaugh, Shaw y otros autores, la abstracción consiste en extraer las propiedades esenciales, o identificar los aspectos importantes, o examinar selectivamente ciertos aspectos de un problema, posponiendo o ignorando los detalles menos sustanciales, que distraen o son irrelevantes.

2.3.9 Escenarios

Los escenarios son una técnica desarrollada por el SEI para tratar los asuntos relativos a la arquitectura mediante un manual de evaluación y pruebas. Los escenarios se relacionan con los atributos de calidad.

El método *Architecture Tradeoff Analysis Method* (ATAM), describe detalladamente a los escenarios y su construcción.

Los escenarios de atributos de calidad son requerimientos específicos de los atributos de calidad de un sistema. Éstos consisten de seis partes (ver Figura 2-7) [PaulClements]:

- **Origen del estímulo:** Éste puede ser cualquier entidad que genere un estímulo en el sistema (humano, computadora, o alguno otro actor).
- **Estimulo:** El estimulo es una condición que necesita ser considerada cuando ésta ocurra en el sistema.
- **Ambiente:** El estimulo ocurre bajo ciertas condiciones. El sistema puede estar operando bajo una condición de sobre carga o puede estar operando normalmente cuando el estimulo ocurre, o alguna otra condición puede ser verdad.
- **Artefacto:** Algún artefacto es estimulado. Pude ser todo el sistema o alguna pieza de él.
- **Respuesta:** La respuesta es la actividad tomada una vez que el estimulo ocurre.
- **Medida de la respuesta:** Cuando la respuesta ocurre, esta debería de ser apreciable de alguna manera, de modo que el requerimiento se pueda probar.

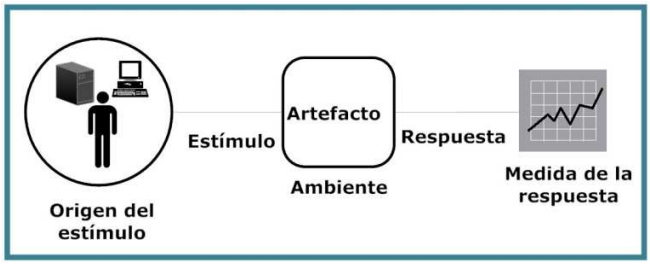

Figura 2-7 Partes de un escenario de un atributo de calidad

La Figura 2-8 muestra un ejemplo de escenario de modificabilidad [PaulClements]. En este ejemplo, un desarrollador (origen del estímulo) desea hacer un cambio en la interfaz de usuario (estímulo), en particular, se desea cambiar el color del fondo de la interfaz. Este cambio será realizado en el código (artefacto) durante la etapa de diseño (ambiente). Además, se considera que este cambio durará alrededor de tres horas (medida de la respuesta) y que la modificación no tendrá efectos secundarios (respuesta).

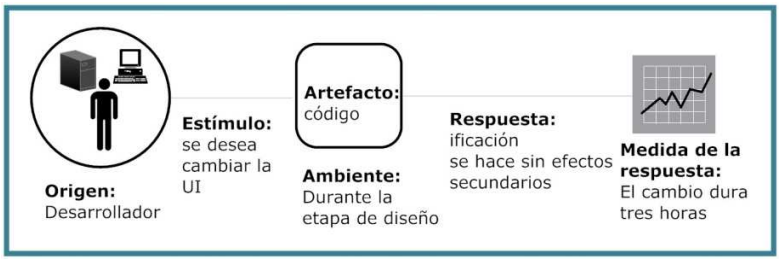

Figura 2-8 Ejemplo de escenario de atributo de calidad; modificabilidad

2.4 El proceso de diseño de la arquitectura de software

Podemos pensar que la arquitectura de un sistema es la visión común en la que todos los *stakeholders* deben estar de acuerdo y deben aceptar. La arquitectura nos da una clara perspectiva del sistema completo, necesaria para controlar el desarrollo, por lo que se necesita

una arquitectura que describa los elementos del modelo que son más importante para nosotros. Estos elementos, arquitectónicamente hablando, incluyen algunos de los subsistemas, dependencias, interfaces, colaboraciones, nodos y clases. Estos elementos deben de describir los cimientos del sistema, que son importantes para poder comprenderlo, desarrollarlo y producirlo [Ivar Jacobson].

En esta sección se describe brevemente los elementos principales que conforman el proceso de la arquitectura de software. Para hacer la descripción del proceso, nos enfocaremos en la descripción de las tareas que se realizan durante una sola iteración.

Primero comencemos identificando los elementos importantes que articulan el proceso de diseño de la arquitectura de software. Estos elementos se muestran en la Figura 2-9.

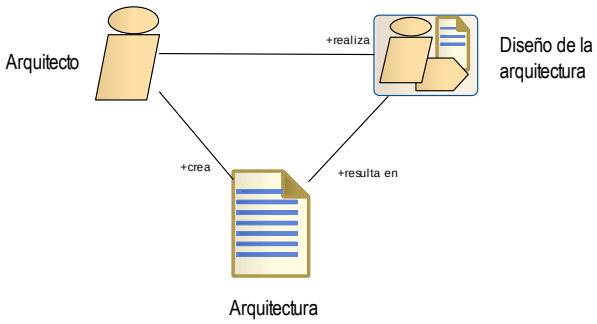

Figura 2-9 Elementos que participan en el proceso de diseño de la arquitectura de software

De la Figura 2-9 se identifican tres elementos que componen al proceso de diseño de la arquitectura de software: el arquitecto, el diseño de arquitectura y la arquitectura. La definición de arquitectura fue discutida en la sección 2.2, en esta sección nos enfocaremos en las características que deben cumplir un arquitecto de software y el contenido del documento de la arquitectura.

La arquitectura es el resultado de la actividad del "diseño de la arquitectura" y es realizada por "el arquitecto". El arquitecto es la persona, equipo u organización, responsable de la arquitectura de un sistema [IEEE 1471 2000]. Algunas de las características que deberá cumplir el arquitecto se muestran a continuación [Peter Eels]:

- Es un líder técnico
- El rol de arquitecto puede estar compuesto por un equipo.
- El arquitecto entiende el proceso de desarrollo de software.
- El arquitecto tiene conocimiento del dominio del negocio.
- El arquitecto tiene un conocimiento tecnológico.

- El arquitecto tiene habilidades de diseño.
- El arquitecto tiene habilidades para programar.
- El arquitecto es un buen comunicador.
- El arquitecto toma decisiones.
- El arquitecto es consciente de las políticas de una organización.
- El arquitecto es un negociador.

Hasta este momento se ha revisado el concepto de arquitectura y el rol del arquitecto, ahora es el momento de hablar sobre el diseño de la arquitectura. El diseño de la arquitectura lo define [IEEE 1471 2000] como:

"Las actividades de definir, documentar, mantener, mejorar y certificar la correcta aplicación de una arquitectura" [IEEE 1471 2000].

Con el fin de ilustrar los elementos que conforman el diseño de la arquitectura de software, se muestra en la Figura 2-10 un metamodelo de términos relacionados con el diseño de la arquitectura del software.

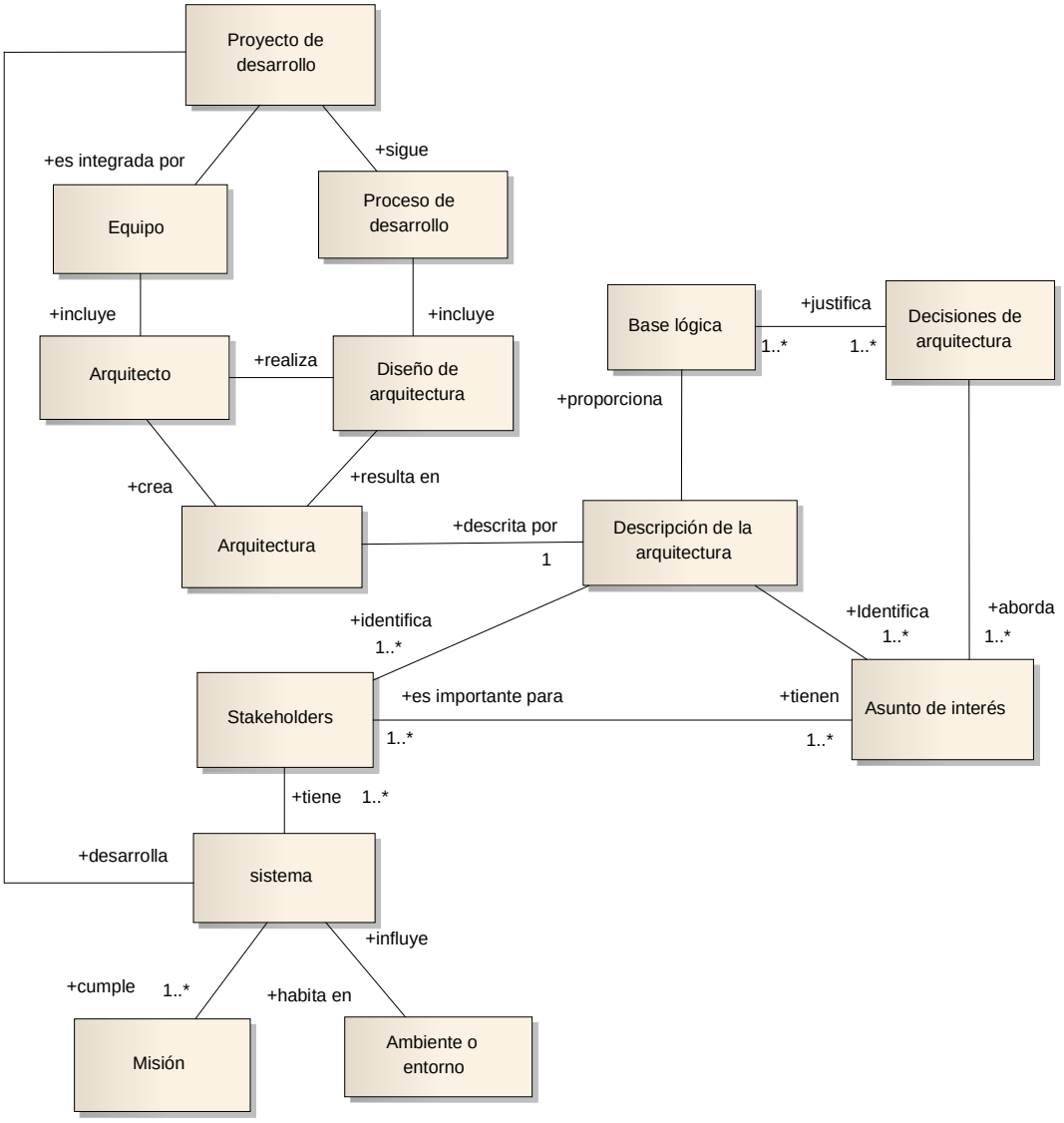

Figura 2-10 Metamodelo de términos involucrados en el proceso de diseño de la arquitectura de software [Peter Eels]

De la Figura 2-10 podemos decir lo siguiente:

- Un sistema tiene una arquitectura
- Un sistema cumple con una o más misiones.
- Un sistema tiene uno o más *stakeholders*.
- Un sistema habita en un ambiente (entorno).
- Un ambiente influye en un sistema.
- Una arquitectura es descrita por una descripción de arquitectura.
- Una descripción de arquitectura identifica uno o más stakeholders.

- Una descripción de arquitectura identifica uno o más asuntos de intereses.
- Una descripción de arquitectura provee una base lógica.
- Un *stakeholder* tiene uno o más asuntos de interés.
- Un asunto de interés es importante para uno o más stakeholders.
- Un proyecto de desarrollo es integrado por un equipo.
- Un proyecto de desarrollo sigue un proceso de desarrollo.
- Un proyecto de desarrollo desarrolla un sistema.
- Un proceso de desarrollo incluye un diseño de arquitectura.
- Un equipo incluye a un arquitecto.
- El arquitecto realiza el diseño de arquitectura.
- El arquitecto crea una arquitectura.
- El arquitecto es un tipo de *stakeholder.*
- El diseño de arquitectura resulta en una arquitectura.
- La base lógica justifica una o más decisiones de arquitectura.
- Una decisión de arquitectura aborda uno o más asuntos de interés.

El metamodelo presentado anteriormente, nos permite ver un panorama de los elementos que son importante únicamente para el diseño de la arquitectura de software, se omiten detalles sobre otros roles y actividades que no sean realizados por el arquitecto.

Para que el proceso de diseño de la arquitectura pueda tener éxito, una de las actividades importante es la documentación de la arquitectura. Como vimos anteriormente, la Figura 2-10 se enfoca principalmente en la "descripción de la arquitectura", el cual podemos ver como el contenedor de los elementos claves que conforman el diseño arquitectónico. En lo que sigue, nos enfocaremos en hablar sobre los elementos importantes para la documentación de la arquitectura.

2.4.1 Documentación de la arquitectura

2.4.1.1 Conceptos clave

De acuerdo con el estándar IEEE 1471-2000, existen tres conceptos importantes para la descripción de una arquitectura: el *viewpoint*, el *view* y el modelo. La Figura 2-11 muestra el metamodelo de los elementos relacionados para la descripción de una arquitectura [IEEE 1471 2000]:

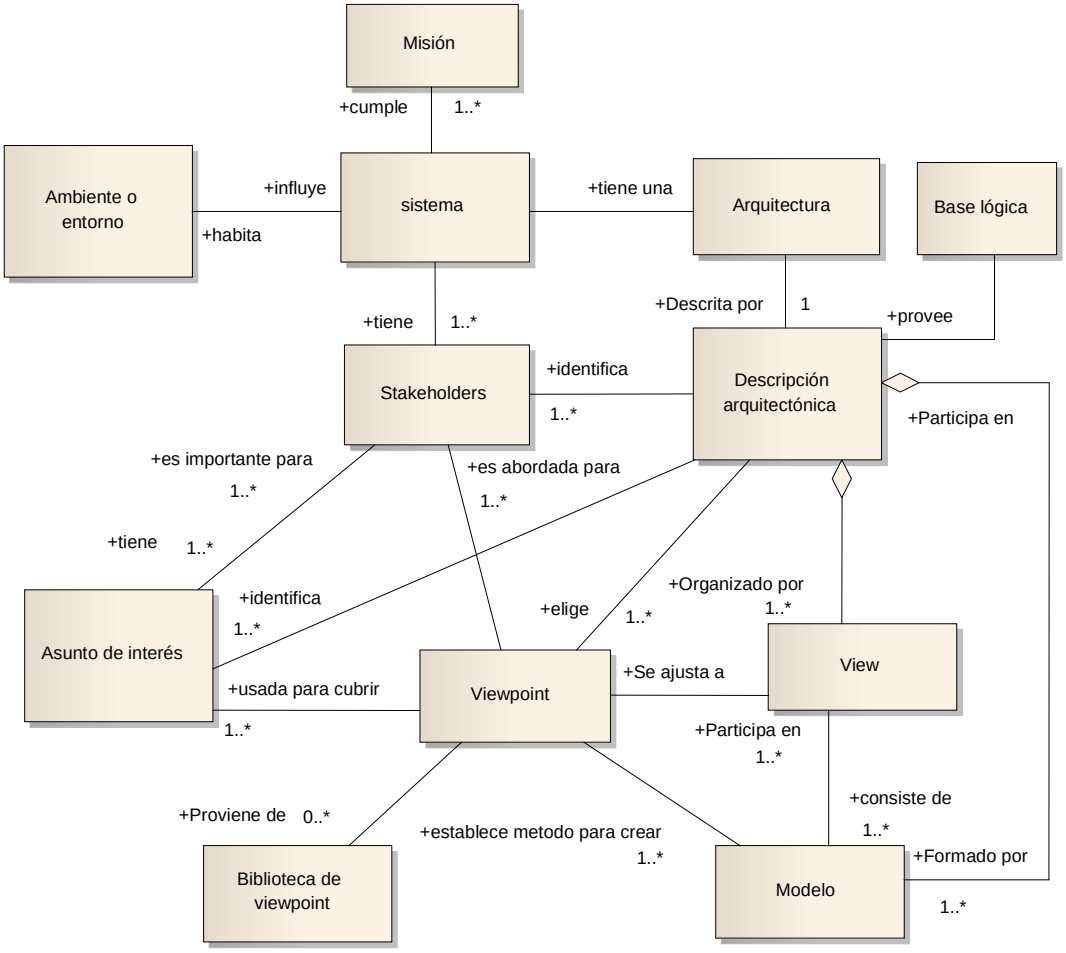

Figura 2-11 Metamodelo para la descripción de una arquitectura [IEEE 1471 2000]

De la Figura 2-11 se puede obtener lo siguiente:

- Una descripción arquitectónica es organizada por una o más *views* (vistas).
- Una *view* consiste de uno o más modelos.
- Un modelo participa en una o más *views*
- Un modelo participa en una o más descripciones arquitectónicas*.*
- Una *view* se ajusta a un *viewpoint.*
- Una descripción arquitectónica elige uno o más *viewpoints.*
- Un *viewpoint* establece el método para crear uno o más modelos.
- Un *viewpoint* es usado para cubrir uno o más asuntos de interés.

Ahora centraremos nuestra atención en los tres conceptos clave: *viewpoint, view* y modelo.

Un *Viewpoint* es: *una especificación de las convenciones para construir y usar una vista. Un patrón o plantilla para la cual desarrollamos "views" individuales mediante el hecho de establecer el propósito y audiencia para una "view" además de las técnicas de su creación y análisis* [IEEE 1471 2000].

Una *view* es: *una representación de todo el sistema desde una perspectiva de un conjunto relacionado de asuntos de interés* [IEEE 1471 2000].

Tomando las dos definiciones anteriores y observando el metamodelo de la Figura 2-11, podemos decir que el objeto es a una clase como el view es a un viewpoint. En cierto sentido esto es verdad, ya que el viewpoint explica como formar una view.

Un *viewpoint* tiene las siguientes características:

- Un *viewpoint* tiene una audiencia.
- Una *viewpoint* sirve para uno o más propósitos.
- Un *viewpoint* es un patrón o plantilla para una *view*.
- Un *viewpoint* define técnicas.

La Figura 2-12 muestra los viewpoints básicos y la Tabla 1 muestra una descripción de éstos.

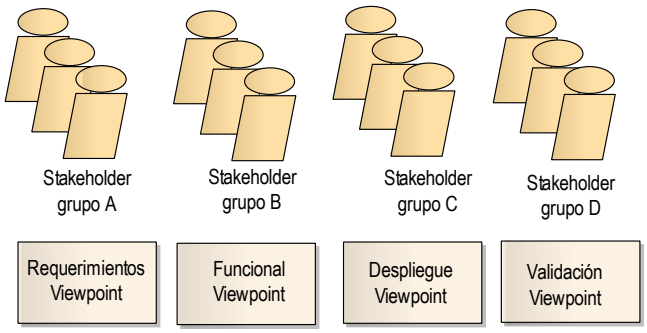

Figura 2-12 *Viewpoints* básico [Peter Eels].

Viewpoint	Descripción
Requerimientos	Se enfoca en proveer un indicador para los requerimientos del sistema que forman la arquitectura, se incluyen; requerimientos funcionales, requerimientos no funcionales y restricciones.
Funcional	Se enfoca a los elementos que soportan la funcionalidad del sistema (componentes, sus relaciones y comportamiento)
Despliegue	Se enfoca en los elementos que muestran la distribución del sistema (nodos, dispositivos y su conexión entre ellos)
Validación	Se enfoca en proveer un indicador para saber si el sistema proveerá la funcionalidad requerida, exhibe la calidad requerida y se ajusta a las restricciones definidas.

Tabla 1 Descripción de los viewpoints básicos

Existen otro tipo de *viewpoints,* llamados perspectivas o viewpoints transversales (*cross-cutting viewpoints*).

Una perspectiva arquitectónica es: *una colección de actividades, tácticas y guías que se aseguran de que el sistema exhiba un conjunto particular de atributos de calidad que requieran ser considerados a través de un número de views* (Rozanski 2005).

Podemos decir que una perspectiva es un tipo especial de *viewpoint,* la cual se enfoca en la calidad que el sistema debe exhibir. La Figura 2-13 tiene como objetivo ilustrar la perspectiva de rendimiento y la de seguridad a través de los cuatro *viewpoints* básicos.

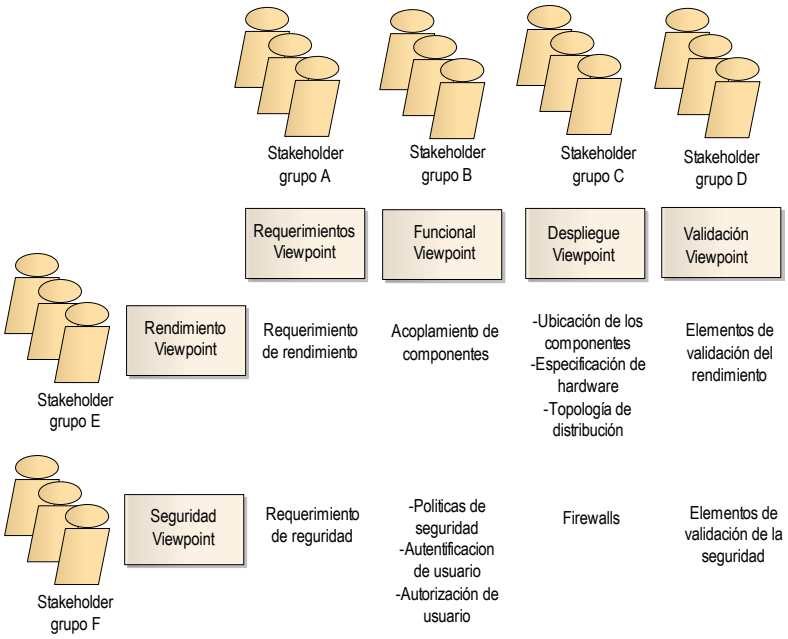

Figura 2-13 Viewpoints y perspectivas [Peter Eels]

Como podemos ver, el uso de *viewpoints* nos permite manejar la complejidad del sistema entre otras cosas. La razón de especificar y utilizar *viewpoints* se puede resumir como sigue:

1. Ayudan a manejar la complejidad del sistema.
2. Se enfocan en un aspecto específico del sistema.
3. Sirven para comunicarse con los stakeholders.

El último termino que resta por estudiar es el de modelo. El modelo es un término utilizado en el estándar IEEE 1471-2000 para referirse a un producto de trabajo que participa en una vista.

El modelo provee la descripción específica o contenido de una arquitectura. Por ejemplo, un *view* estructural podría consistir de un conjunto de modelos de la estructura de un sistema. El elemento de tales modelos podría incluir componentes identificables de un sistema junto con sus interfaces e interacciones entre estos elementos [IEEE 1471 2000].

Entre los modelos básicos podemos mencionar el modelo funcional, el modelo de despliegue y el modelo de datos.

2.4.1.2 Marco de trabajo para la descripción arquitectónica

Un marco de trabajo para la descripción arquitectónica representa un conjunto de viewpoints que pueden ser usados para describir una arquitectura [Peter Eels].

Podemos decir que un marco de trabajo para la descripción arquitectónica nos provee un conjunto de *viewpoints* mediante los cual, en lugar de comenzar desde cero, utilizamos un catálogo con los viewpoints que mejor se ajusten a nuestro proyecto.

Algunos de los marcos de trabajo para la descripción arquitectónica con los que nos podemos encontrar son: 4+1 views (Kruchten 1995), IEEE 1471-2000 [IEEE 1471 2000], views and beyond, zachman framework (zachman 1987), cross-cutting concerns (Rozanski 2005).

Un ejemplo simplificado de un marco de trabajo para la descripción arquitectónica lo podemos encontrar en [Peter Eels].

2.4.1.3 Documento de arquitectura de software

El documento de arquitectura de software provee un resumen general sobre la arquitectura del sistema, usando varias vistas arquitectónicas para representar diferentes aspectos del sistema. (RUP 2008).

El documento de arquitectura de software es un entregable que se ajusta al marco de trabajo de descripción arquitectónica elegido. El formato que puede tener un documento de arquitectura de software es el siguiente:

- Preliminares (titulo de la página, tabla de contenido, lista de figuras, referencias, entre otras)
- Objetivos del documento de la arquitectura de software.
- Resumen de la arquitectura.
- Decisiones arquitectónicas.
- Los *views* que se consideraron para el proyecto, por ejemplo:
    - Requerimientos.
    - Funcional.
    - Despliegue.
    - Validación.
    - Desempeño
    - Seguridad
- Apéndice.

2.4.2 Actividades dentro del proceso de arquitectura

Las principales actividades que componen el proceso de diseño de la arquitectura se muestra en la Figura 2-14:

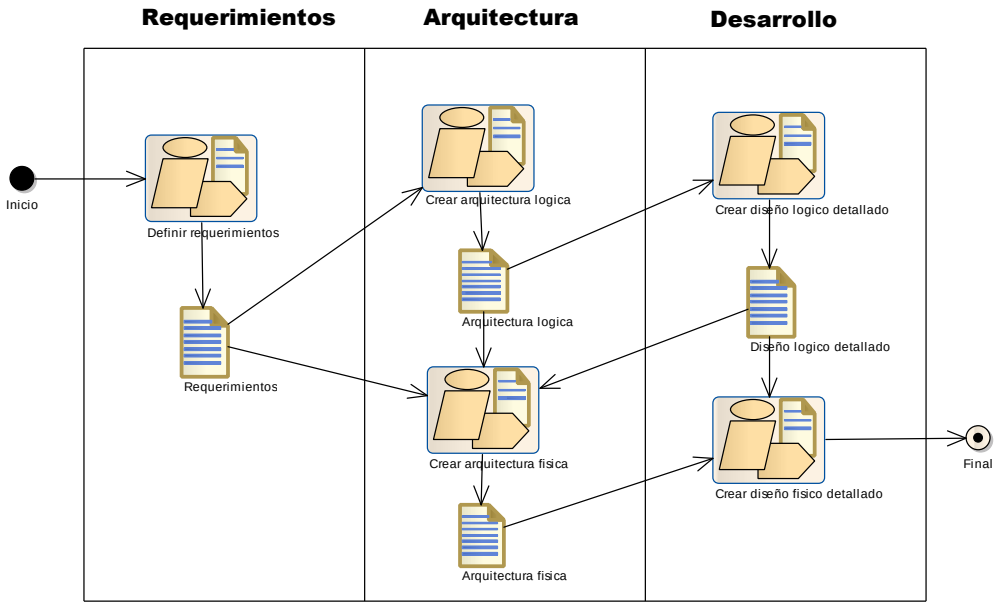

Figura 2-14 Resumen de actividades [Peter Eels].

La Figura 2-14 muestra las actividades en las cuales el arquitecto de software está involucrado o es responsable. Como podemos ver, los esfuerzos de construcción para la arquitectura de un sistema se encuentran entre la etapa de requerimientos y el desarrollo. La iteración comienza en la etapa de requerimientos, ejecutando la actividad denominada "**Definir requerimientos".** En esencia esta actividad no es propia del arquitecto de software, sino más bien le pertenece al analista del negocio, más sin embargo, el arquitecto participa en algunas de las tareas definidas dentro esta actividad. Después el arquitecto es el encargado de crear una primera arquitectura del sistema, esto ocurre dentro de la actividad "**Crear arquitectura lógica**". El resultado de esta actividad es una arquitectura independiente de cualquier tecnología y es referida como *arquitectura lógica*. Podemos considerar que la creación de una arquitectura lógica es el paso que permite llevar los requerimientos del sistema a una arquitectura física, donde se utilizará una tecnología especifica y se formará la base para la implementación. La arquitectura lógica es la entrada para cualquier intento de un diseño detallado dentro de la actividad "**Crear diseño lógico detallado"** ubicado en la etapa de desarrollo. Es importante hacer notar que esta actividad da a lugar a la adición de cualquier detalle restante que pueda ser necesario a nivel lógico, pero no será considerado como arquitectura. Lo anterior se debe a la diferencia que existe entre arquitectura y diseño:

Toda la arquitectura es diseño, pero no todo el diseño es arquitectura. La arquitectura representa las decisiones de diseño significantes que forman un sistema, donde las decisiones significantes se miden por el costo del cambio [Booch 2009].

En otras palabras, la arquitectura se puede considerar como una estrategia de diseño, mientras que el diseño detallado se considera como la táctica del diseño.

Una vez vistas las tareas de definir requerimientos, crear arquitectura lógica y crear diseño lógico detallado, el arquitecto vuelve a definir la arquitectura, pero esta vez en la actividad de "**Crear arquitectura física"**, en la cual se considera la tecnología que se usará, teniendo como resultado la arquitectura física. Es la arquitectura física la que se considera dentro de cualquier intento de diseño detallado realizado en la actividad de "**Crear diseño físico detallado"**, mismo que representa la base para la implementación**.** Es notable mencionar que el arquitecto no es el responsable de las actividades de diseño detallado ni implementación, más bien, sólo guía al equipo de trabajo cuando sea necesario.

En lo que resta de esta sección se describirán las actividades antes mencionadas, en las cuales se podrá apreciar las tareas de las cuales el arquitecto de software está a cargo.

Las tareas que se comprenden dentro de la actividad "**Definición de requerimientos**" se muestran en la Figura 2-15.

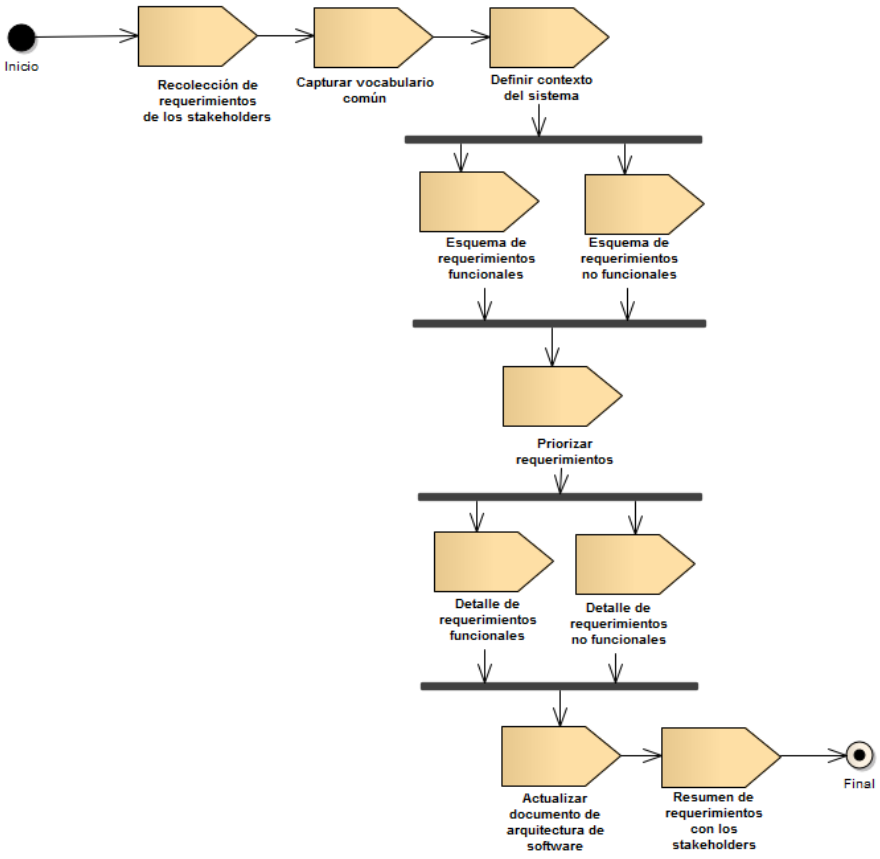

Figura 2-15 Tareas en la definición de requerimientos [Peter Eels].

Recolección de requerimientos de los stakeholders: La iteración comienza en la tarea llamada "Recolección de requerimientos de los stakeholders", que como su nombre sugiere se enfoca en entender las necesidades de varios de los *stakeholders.* Los requerimientos proveen al arquitecto un indicador inicial sobre el alcance del sistema que será construido.

Capturar vocabulario común: Después un vocabulario de términos es desarrollado en la tarea "Capturar vocabulario común".

Definir contexto del sistema: esta tarea es de particular interés para el arquitecto, ya que en ella se definen los elementos externos que deben interactuar con el sistema, tales como los usuarios finales y otros sistemas. Basándose en el contexto definido anteriormente, las tareas: "**Esquema de requerimientos funcionales**" y "**Esquema de requerimientos no funcionales**", identifican los requerimientos funcionales y no funcionales del sistema. Es importante mencionar que el arquitecto no sólo se enfoca en los requerimientos funcionales claves, sino también en la calidad del sistema y restricciones de la solución que comprenden los requerimientos no funcionales, los cuales son a menudo más complicados de resolver que los requerimientos funcionales.

El arquitecto de software está involucrado hasta cierto punto en la actividad de "**Definición de requerimientos**" para asegurarse que los requerimientos son especificados de manera que puedan alcanzarse de acuerdo con la tecnología disponible, dentro del tiempo requerido y con el presupuesto dato.

Priorizar requerimientos: una vez definidos los requerimientos, el arquitecto participa en la tarea de "*Priorizar requerimientos*", en la cual el arquitecto se asegura que las prioridades sean influenciadas por aquellos requerimientos que permitan que la arquitectura se estabilice los más pronto posible.

Los requerimientos con mayor prioridad dirigirán el resto de la iteración; en el sentido de que sólo los requerimientos de mayor prioridad serán considerados. Los requerimientos de menor prioridad serán considerados en iteraciones subsecuentes. Los requerimientos de mayor prioridad son detallados en las tareas; "**Detalle de requerimientos funcionales**" y "**Detalle de requerimientos no funcionales**". A continuación el arquitecto documenta formalmente la arquitectura en la tarea "**Actualizar documento de arquitectura de software**". Las tareas relacionadas con los requerimientos terminan con un resumen de los requerimientos en la tarea "**Resumen de requerimientos con los stakeholders**".

Una vez terminada la actividad "Definición de requerimientos", se comienza a ejecutar la actividad "**Crear arquitectura lógica**". La Figura 2-16 muestra las tareas que comprenden esta actividad.

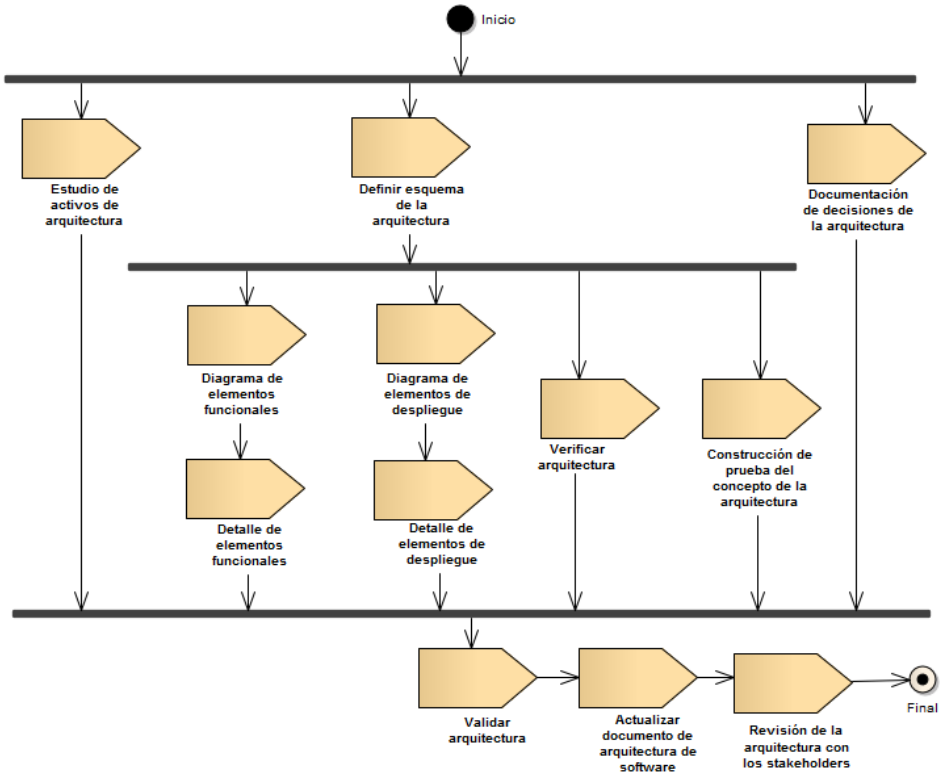

Figura 2-16 Tareas en la actividad "crear la arquitectura lógica" [Peter Eels].

A lo largo de esta actividad el arquitecto considera los activos existentes en la tarea "**Estudio de activos de arquitectura**". Incluso al principio del proyecto el arquitecto puede seleccionar un activo tal como una "arquitectura de referencia" que tenga influencia significativa en su arquitectura. Al paso que se va ejecutando cada una de estas tareas, el arquitecto captura estas decisiones en la tarea "**Documentación de decisiones de la arquitectura**". El estudio de activos de arquitectura es muy importante para cualquier tipo de proyecto de desarrollo de software (tanto para métodos tradicional como métodos ágiles). El número de activos de arquitectura de software con los que cuente una empresa es un indicador del nivel de madurez que se tiene para la construcción de software [PaulClements]. Debido a la importancia de la compresión de los activos de arquitectura, en la sección 2.4.3 se habla de ellos.

Basándose en los requerimientos de mayor prioridad, el arquitecto realiza un esquema de la solución candidata en la tarea "**Definir esquema de la arquitectura**". Después se realizan al mismo tiempo las tareas: "**Diagrama de elementos funcionales**" y "**Diagrama de elementos de despliegue**", subsecuentemente se refinan estos esquemas en términos de componentes, con sus interacciones y relaciones además de sus despliegues en nodos. La tarea de "**Verificar arquitectura**" se asegura que los elementos funcionales y de despliegue que forman la arquitectura, sean consistentes uno con el otro, en particular, asegura que cualquier problema que atraviesen estos elementos (por ejemplo, un atributo de calidad como el rendimiento que influye tanto en los elementos funcionales y de implementación) sean abordados adecuadamente.

El arquitecto también se enfoca en proveer ciertos aspectos de la arquitectura en la tarea "**Construcción del concepto de prueba de la arquitectura**". El objetivo es sintetizar al menos una solución que satisfaga los requerimientos que son significantes para la arquitectura, para determinar si tal solución, prevista por el arquitecto, existe. En particular, la prueba del concepto provee un vehículo para mitigar los problemas de riesgo referentes a la arquitectura y frecuentemente toma la forma de software ejecutable que permite evaluar tanto la funcionalidad del sistema como su calidad.

Después, los elementos arquitectónicos son detallados en las tareas: "**Detalle de elementos funcionales**" y "**Detalle de elementos de despliegue**". La tarea "**Validar arquitectura**", se asegura que los elementos arquitectónicos que satisfagan los requisitos establecidos (funcionales y no funcionales), así como las consideraciones del proyecto, como son: recursos, el presupuesto y las limitaciones de horario. Después el arquitecto hace un resumen de la arquitectura independiente de cualquier plataforma en la tarea "**Actualizar documento de arquitectura de software**". La descripción de la arquitectura resultante se utiliza para comunicar la arquitectura con los stakeholders, incluyendo diseñadores, programadores, *testers*, administrador de proyecto, mantenimiento y personal de apoyo. Finalmente la tarea "**Revisión de la arquitectura con los stakeholders**", permite un acuerdo sobre la arquitectura que se establece.

De la Figura 2-14 podemos ver que los productos de trabajo relacionados con la arquitectura, son los documentos de entrada requeridos en la actividad "**Crear diseño lógico detallado**", el cual añade el detalle a los elementos arquitectónicos identificados antes de pasar a la actividad "**Crear arquitectura física**".

La actividad "**Crear arquitectura física**" contiene exactamente las mismas tareas que las que se plantean en la actividad "Crear arquitectura lógica", pero en esta ocasión se toman en cuenta las pertinentes consideraciones tecnológicas. El conjunto de productos de trabajo relacionados a la arquitectura que provienen de la actividad "**Crear arquitectura física**", son la entrada para la actividad "**Crear diseño físico detallado**", la cual le agrega un nivel relevante de detalle para identificar los elementos arquitectónico y el cual puede ser usado como la base para la implementación. La implementación resulta en una entrega de software ejecutable que después es probado para asegurarse que cumple con los requerimientos asociados con la actual iteración.

Como conclusión podemos ver el proceso de diseño de la arquitectura en la Figura 2-17, donde los puntos importantes son:

- Requerimientos de la arquitectura
- Especificación de la arquitectura
    - Arquitectura conceptual
    - Arquitectura lógica
- Validación de la arquitectura
- Despliegue de la arquitectura

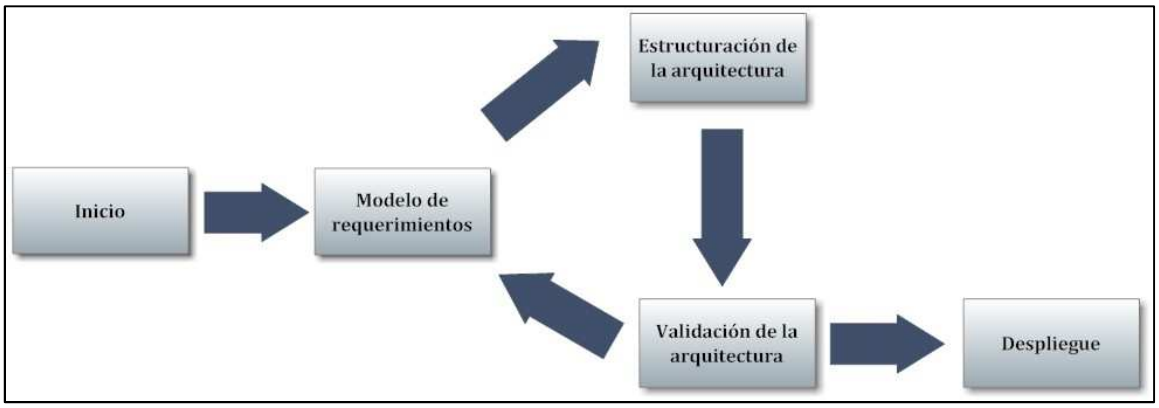

Figura 2-17 Proceso de diseño de la arquitectura de software

2.4.3 Activos arquitectónicos

De acuerdo con [Kruchten 01] existen tres maneras de crear una arquitectura: robando un diseño, utilizando un método para crearla y por intuición.

En términos del robo de un diseño, la mayoría de los elementos de una arquitectura de software son derivados de un sistemas previamente diseñado, cuyas características son muy similares, o una arquitectura encontrada en algún libro técnico.

Por otro lado, utilizando un método para crear una arquitectura se debe seguir un proceso sistemático que permita crear una arquitectura que se adecua a las necesidades del proyecto.

Por último, en términos de la intuición, se refleja la experiencia del arquitecto de software, quien reconoce algunos patrones, o encuentra otra inspiración para un elemento arquitectónico.

Cualquiera que sea el caso para especificar una arquitectura, el estudio de las fuentes que proveen el conocimiento para la creación e implantación de un modelo arquitectónico es fundamental para la definición de ésta. Estas fuentes son conocidas como activos arquitectónicos o activos reutilizables para la arquitectura [PeterEeles].

Los activos reutilizables pueden ser aplicados a lo largo de todo el proceso de desarrollo del sistema. Por ejemplo, un activo reutilizable puede representar a un requerimiento reutilizable (un requerimiento que ocurre una y otra vez en diferentes proyectos), un elemento que representa una solución reutilizable (tal como un patrón arquitectónico o código reutilizable), un caso de prueba u otros. En la presente sección se describen los activos reutilizables que son relevantes para el arquitecto de software. La Tabla 2 resume a los activos arquitectónicos más comunes de acuerdo con [PeterEeles].

Activo arquitectónico	Descripción
***Frameworks* de aplicación**	Son conocidos como *Frameworks,* y representan una implementación parcial de un área específica de una aplicación. Algunos de los *Frameworks* más conocidos es Java Server Faces y ASP.NET. En ocaciones las arquitecturas de referencia también son caracterizadas como *Frameworks;* por ejemplo J2EE, o .NET.
Arquitectura de referencia	Exhibe las mismas características que otras arquitecturas, con algunas diferencias importantes. La primera es que no tiene una manifestación física en la experiencia del autor y es sólo documentación (en la forma de modelos, por ejemplo). La segunda es que a menudo se encuentra incompleta. Las arquitecturas de referencia pueden clasificarse en dos: independientes de la tecnología y específicas a una tecnología. Ejemplos (La arquitectura de J2EE y la de .NET).
Mecanismos arquitectónicos	Son elementos que permiten solucionar un problema específico de diseño. Por ejemplo, los mecanismos que se emplean para la persistencia de datos. La diferencia entre un mecanismo y un patrón es que un mecanismo es más concreto y puede contener una implementación parcial. Además, un mecanismo puede adoptar varios patrones.
Estilos arquitectónicos	Definen una familia de sistemas en términos de un patrón de organización estructural. Más específicamente, un estilo arquitectónico define un vocabulario de componentes y tipos de conectores.
Componentes y bibliotecas de componentes	Estas son los elementos reutilizables más comunes, entre ellos se encuentran las clases, bibliotecas de clases, bibliotecas de procedimientos, entre otros.
Patrones de diseño	Un patrón de diseño es una solución a un problema de diseño. Para que ésta solución sea considerada un patrón de diseño, debe de haberse comprobado su efectividad resolviendo problemas similares en proyectos anteriores y, además, debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintos casos.
Aplicaciones heredadas	El principal activo arquitectónico, que todo arquitecto de software debería considerar, son los sistemas existentes. Normalmente el uso de tales sistemas requiere de una integración con el nuevo sistema a desarrollar, lo cual es conocido como integración de aplicaciones empresariales. (EAI, por sus siglas en ingles). Al igual que en las aplicaciones empaquetadas, el nivel de reutilización suele ser muy elevado, y los esfuerzos se enfocan más en la integración que en el desarrollo personalizado.
Aplicaciones empaquetadas	Son conocidas como Commercial-Off.The-Shelf (COTS), las cuales proveen un índice de reutilización importante; por ejemplo, *Enterprise Resource Planning* (ERP)

Tabla 2 Activos arquitectónicos

Como se puedo haber visto en esta sección, los activos arquitectónicos revelan un grado de madurez para el desarrollo de software, el cual se basa en la reutilización de compones que previamente fueron diseñados y documentado para la creación de nuevos proyectos. Por lo tanto, el estudio de activos arquitectónicos es una tarea fundamental para desarrollar aplicaciones cada vez más rápido.

2.4.4 Prácticas de análisis, síntesis y evaluación de la arquitectura de software

En esta sección se presentan algunos de los principales métodos propuestos por el SEI [SEI2010] para el análisis, síntesis y evaluación de la arquitectura. El objetivo de esta sección no hacer un estudio exhaustivo sobre estos métodos, más bien, se pretende dar una primera comprensión sobre ellos. Un estudio más profundo sobre estos métodos se puede consultar en las referencias que aquí se recomiendan.

2.4.4.1 Quality Attribute Workshop (QAW)

El taller de atributos de calidad (*Quality attribute workshop*, QAW) es un taller enfocado a la captura de los requerimientos significantes para el diseño de la arquitectura. Estos requerimientos son los atributos de calidad que son descritos a través de escenarios de atributos de calidad (ver sección 2.3.9 de este trabajo)

El QAW relaciona los *stakeholders* de un sistema de manera temprana en el ciclo de vida para descubrir los principales atributos de calidad en un sistema [QAW].

La Tabla 3 muestra los pasos llevados a cabo dentro del QAW.

Pasos	Descripción [QAW]
1. Presentación e introducción.	Describir el propósito del QAW: • se describe la importantica del Taller de atributos de Calidad y se explica cada paso del método. • se presentan a los *stakeholders* (posición en la organización y rol en el sistema)
2. Presentación de la misión del negocio.	Se realiza una presentación por parte del cliente, en la cual: • se dan a conocer los objetivos y directrices del negocio. Durante la presentación se capturan requerimientos funcionales, restricciones y atributos de calidad.
3. Presentación del plan arquitectónico.	Se dan a conocer los elementos arquitectónicos iniciales y el plan de desarrollo de la arquitectura como: • planes y estrategias para abordar los principales requerimientos del negocio. • principales requerimientos y restricciones técnicas. • diagrama de contexto, diagramas de alto nivel del sistema, entre otros.
4. Identificación de las directrices arquitectónicas.	Durante los pasos 2 y 3 se capturan las directrices arquitectónicas claves para abordar los atributos de calidad del sistema. Con frecuencia las directrices arquitectónicas incluyen: • Requerimientos en un alto nivel de abstracción. • Restricciones técnicas y varios atributos de calidad.
5. Lluvia de escenarios de atributos de calidad.	Mediante una lluvia de ideas se generan los escenarios de atributos de calidad (ver sección 2.3.9). La generación de escenarios de atributos de calidad es el paso principal del QAW.
6. Consolidación de los escenarios de atributos de calidad.	Se agrupan los escenarios que hacen referencia al mismo atributo de calidad.
7. Establecer la prioridad de los escenarios de atributos de calidad.	Se establece la prioridad a los escenarios de atributos de calidad basado en los votos que los participantes asignan a dichos escenarios.
8. Refinamiento de los escenarios de atributos de calidad.	Se detallan los 4 o 5 escenarios de atributos de calidad más votados. El detalle de un escenario incluye (ver sección 2.3.9): Estímulo, respuesta, fuente del estímulo, ambiente, artefacto estimulado y medición de la respuesta.

Tabla 3 Pasos del método QAW

Para un estudio más detallado sobre este método se recomienda consultar a [QAW], [Rick Kazman] y [SEI2010].

2.4.4.2 Attribute-Driven Design (ADD)

Una vez que se han capturado y priorizado los principales escenarios de atributos de calidad mediante el método QAW, se procede a ejecutar el método de diseño dirigido por atributos de calidad (Attribute-Driven Design, ADD). El método ADD [Rob Wojcik] fue desarrollado por el SEI [SEI2010] para diseñar la arquitectura de un sistema que satisfaga tanto los requerimientos funcionales como los de calidad.

El diseño de la arquitectura, que genera el método ADD, es un diseño que sigue un enfoque iterativo de descomposición del sistema en componentes cada vez más pequeños. Durante el ADD, se van realizando elecciones de diseño que permitan satisfacer los requerimientos descritos por los escenarios de atritos de calidad. Dentro de las elecciones de diseño encontramos el uso de patrones y tácticas (ver sección 2.5.2).

Los pasos del método ADD se presentan en la Figura 2-18 y se describen en la Tabla 4.

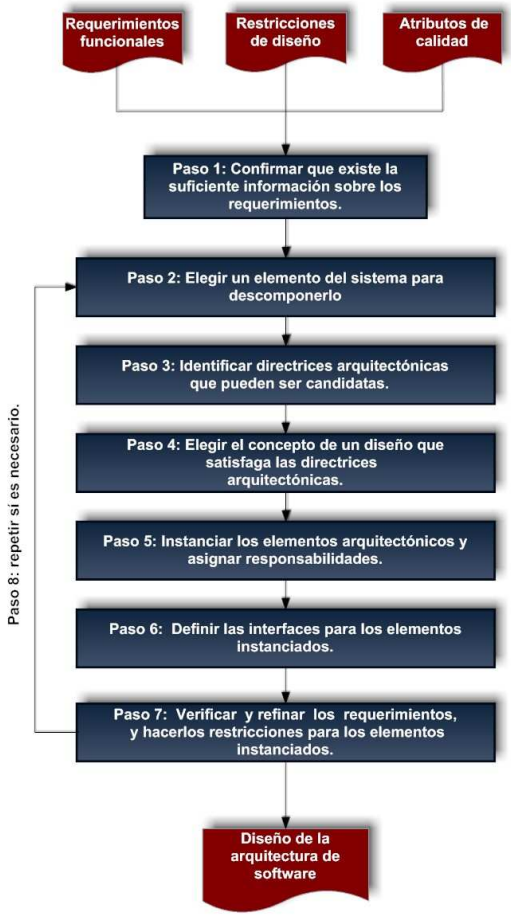

Figura 2-18 Pasos del Método ADD [Rob Wojcik]

Paso	Descripción [PaulClements]
1. Confirmar que existe la suficiente información sobre los requerimientos.	Confirmar que se cuenta con la información necesaria (requerimientos funcionales y no funcionales) antes de comenzar con el método ADD.
2. Elegir un elemento del sistema para descomponerlo	El elemento o módulo para comenzar generalmente es todo el sistema entero. Todas las entradas requeridas para este módulo deben estar disponibles (restricciones, requerimientos funcionales, requerimientos de calidad)
3. Identificar directrices arquitectónicas que pueden ser candidatas.	Este paso determina que es importante para la descomposición
4. Elegir el concepto de un diseño que satisfaga las directrices arquitectónicas.	Se crean (o seleccionan) los patrones basados en tácticas que pueden ser usadas para abordar las directrices arquitectónicas. Se identifican los módulos hijos requeridos para implementar las tácticas.
5. Instanciar los elementos arquitectónicos y asignar responsabilidades.	Se instancian los módulos y se les asigna la funcionalidad de los casos de uso. Por último se representar usando múltiples vistas.
6. Definir las interfaces para los elementos instanciados.	Se definen los servicios y las propiedades provistas y requeridas por los elementos de software. A estos servicios y propiedades se les llama la interfaz del elemento.
7. Verificar y refinar los requerimientos, y hacerlos restricciones para los elementos instanciados.	Este paso verifica que nada importante sea olvidado y prepara a los módulos hijos para una descomposición o implementación.
8. Repetir desde el paso 2 si es necesario.	Si más elementos o módulos necesitan una descomposición, entonces se vuelve al paso 2.

Tabla 4 Descripción de los pasos del método ADD [Rob Wojcik]

Para un estudio más detallado sobre este método se recomienda consultar a [PaulClements], [Rob Wojcik], [Rick Kazman] y [SEI2010].

2.4.4.3 Architecture Trade-off Analysis Method (ATAM)

El método ATAM se emplea para conocer qué tan bien la arquitectura propuesta logra satisfacer los atributos de calidad y revela además que riesgos, puntos sensibles y compromisos están involucrados en la arquitectura.

Las principales metas del método de evaluación ATAM son [Lawrence G]:

- Obtener y perfeccionar las directrices arquitectónicas que aborda a los atributos de calidad.

- Obtener y perfeccionar las decisiones de diseño arquitectónico.
- Evaluar las decisiones de diseño arquitectónico para determinar si abordar correctamente los atributos de calidad.

Para que el método de evaluación ATAM alcance sus metas, envuelve a todos los *stakeholders* del sistema incluyendo administradores, desarrolladores, equipo de mantenimiento, equipo de pruebas, usuarios finales y al cliente. El método ATAM es un medio para detectar áreas potenciales de riesgo dentro de la arquitectura de un sistema [PaulClements], [Lawrence G].

Entre los temas que se tratan dentro del método de evaluación ATAM se encuentra [Lawrence G]:

- Riesgo: alternativas que pueden crear futuros problemas en algún atributo de calidad.
- Puntos sensibles: alternativas para las cuales un pequeño cambio puedan afectar significativamente a un atributo de calidad.
- *Tradeoffs:* decisiones que pueden afectar a más de un atributo de calidad.

El método de evaluación ATAM está conformado por nueve pasos agrupados en cuatro fases. La Tabla 5 describe cada una de estas fases de acuerdo con [Lawrence G], [John K].

Fase 0 Asociación y Preparación	
Durante la fase 0, el líder del equipo de evaluación y el líder del proyecto se reúnen de manera informal para trabajar sobre el detalle de los ejercicios que se verán en el método de evaluación ATAM.	
Fase 1 Evaluación inicial	
1. Presentar el ATAM	El equipo de evaluación presenta una breve descripción de los pasos del método de evaluación ATAM, las técnicas usadas y las salidas del proceso.
2. Presentar las directivas del negocio	El administrador del sistema presenta brevemente las directivas del negocio y el contexto para la arquitectura.
3. Presentar la arquitectura	El arquitecto presenta un resumen de la arquitectura, enfocándose en cómo ésta aborda las directrices del negocio.
4. Identificar los enfoques arquitectónicos	Se detallan las decisiones arquitectónicas encontradas en el paso anterior. Los enfoques arquitectónicos son identificados por el arquitecto pero no son analizados.
5. Generar el árbol de utilidad	Identificar, priorizar y perfeccionar los atributos de calidad (desempeño, seguridad, modificabilidad, etc.) más importantes en el árbol de utilidad.
6. Análisis de los enfoques arquitectónicos	Verificar que los enfoques arquitectónicos elegidos abordan los principales atributos de calidad con el fin de identificar los riesgos, puntos sensibles y *tradeoffs.*
Fase 2 Evaluación completa	
7. Lluvia de ideas y establecimiento de la prioridad de los escenarios de atributos de calidad.	Crear y analizar los escenarios de atributos de calidad que representan los diferentes intereses de los *stakeholders* para comprender los requerimientos de atributos de calidad y su importancia relativa.
8. Análisis de los enfoques arquitectónicos	Continuar con la identificación de riesgos, puntos sensibles y *tradeoffs* mientras se observa el impacto de cada escenario de atributo de calidad en los enfoques arquitectónicos.
Fase 3 Seguimiento	
9. Presentar los resultados.	Basados en la información recolectada durante el método de evaluación ATAM (estilos, escenarios, árbol de utilidad, riesgos, puntos sensibles, *tradeoffs*) recapitular los pasos del ATAM, los resultados y recomendaciones.

Tabla 5 Pasos del método de evaluación ATAM

Para un estudio más detallado sobre este método se recomienda consultar a [PaulClements], [Lawrence G], [John K][Rick Kazman] y [SEI2010].

2.4.4.4 Active Reviews for Intermediate Designs (ARID)

El método ARID es método efectivo para asegurar la calidad y el diseño detallado del software. Éste combina la técnica ADR (*Active Design Reviews*) [Paul C] con el método de evaluación ATAM, con el único fin de crear una técnica para la inspección de diseños parcialmente completos [Rick Kazman]. Al igual que el método de evaluación ATAM el método ARID reúne a todos los *stakeholders* para crear un conjunto de escenarios, los cuales se utilizan para determinar sí el diseño es adecuado para trabajar con él.

El método ARID ayuda a encontrar los temas y problemas que dificultan el uso exitoso del diseño [Rick Kazman].

Los pasos del método ARID, de acuerdo con [Paul C], [Rick Kazman], se presentan en la Tabla 6 y Tabla 7.

Fase 1 Reunión previa	
1. Identificar a los encargados de la revisión.	Se identifican a las personas que deben estar durante la revisión.
2. Preparar una presentación con el informe del diseño.	Se prepara una breve presentación que explique el diseño. Lo anterior, con el objetivo de presentar el diseño a gran detalle para que la audiencia pueda utilizarlo. Por ejemplo, se pueden incluir ejemplos del uso del mismo para la resolución de problemas reales.
3. Preparar los escenarios que se revisarán durante el método.	Se preparan los escenarios que serán empleados durante el método. Al igual que los escenarios en el método de evaluación ATAM, estos son diseñados para ilustrar el concepto de un escenario, que pueden o no ser utilizados para efectos de la evaluación; los evaluadores son libres de utilizarlos, o rechazarlos.
4. Prepararse para la reunión de revisión.	Copias de la presentación, escenarios a utilizar y la agenda de la revisión son creadas y distribuidas a los evaluadores durante la reunión de la revisión.

Tabla 6 Pasos del método ARID (1/2)

Fase 2 Reunión de revisión	
5. Presentar el método ARID	El encargado de la revisión explicará durante 30 minutos los pasos del método ARID a los participantes.
6. Presentar el diseño	Se presenta el diseño a través de ejemplos. La duración de la presentación puede ser de dos horas. La meta de esta presentación es ver sí el diseño es entendible y no el de averiguar los detalles que se encuentran detrás de la implementación.
7. Lluvia de ideas y establecimiento de la prioridad de los escenarios de atributos de calidad.	Al igual que en el método de evaluación ATAM, los participantes sugieren y priorizan los escenarios en los cuales se puede utilizar el diseño para resolver problemas que ellos esperan encarar. Después, los escenarios son sometidos a votación, y se utilizan los que resultan ganadores para hacer pruebas sobre el diseño.
8. Realizar la revisión	Comenzando con el escenario que recabo más votos, se solicita el seudo-código que utiliza el diseño para proveer el servicio. Este paso se continúa hasta que ocurra alguno de los siguientes eventos: • Se agota el tiempo de la revisión. • Se han estudiado los escenarios con mayor 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. Presentar las conclusiones de la revisión	Al final, se resumen todos los asuntos tratados y se les invita a los participantes a externar su opinión sobre la revisión.

Tabla 7 Pasos del método ARID (2/2)

Para un estudio más detallado sobre este método se recomienda consultar a [Paul C], [Rick Kazman] y [SEI2010].

2.4.4.5 Cost-Benefit Analysis Method (CBAM)

El método de análisis y costos de beneficios CBAM provee una evaluación de los asuntos técnicos y económicos con respecto a las decisiones arquitectónicas [PaulClements]. Éste método ayuda a los *stakeholders* del sistema para la elección de arquitecturas alternativas que permitan mejorar el diseño del sistema actual, o para la fase de mantenimiento del mismo [Rick Kazman].

Los pasos del método CBAM se presentan en la Figura 2-19 y se describen en la Tabla 8 y Tabla 9.

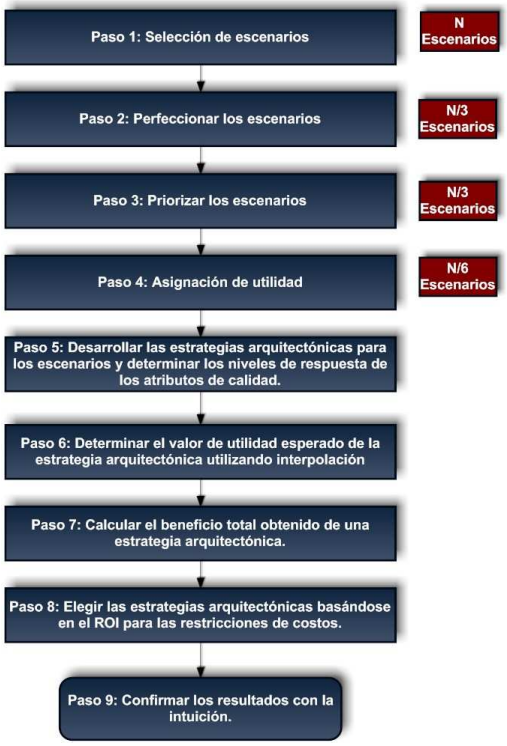

Figura 2-19 Pasos del método CBAM [PaulClements]

Pasos	Descripción
1. Selección de escenarios	Clasificar los escenarios obtenidos durante el método de evaluación ATAM y dar a los *stakeholders* la oportunidad de aportar otros nuevos. Priorizar los escenarios basándose en la satisfacción de las metas del negocio del sistema. Comenzando con el escenario de mayor prioridad, se elige un tercio de la lista de los escenarios priorizados.
2. Perfeccionar los escenarios	Perfeccionar los escenarios, enfocándose en la medición de su respuesta. Obtener el peor, actual, deseado y mejor caso del nivel de la respuesta del atributo de calidad para cada escenario.
3. Priorizar los escenarios	Reunir 100 votos, de cada uno de los *stakeholders,* y distribuirlos entre los escenarios, donde el voto de los *stakeholders* está basado en la consideración del valor de la respuesta deseada para cada escenario. Sumar los votos y elegir el 50% de los escenarios para los siguientes pasos.

Tabla 8 Pasos del método CBAM (1/2)

Pasos	Descripción
4. Asignación de utilidad	Determinar la utilidad para cada nivel de respuesta de los atributos de calidad en estudio (peor caso, actual, deseado y mejor caso).
5. Desarrollar las estrategias arquitectónicas para los escenarios y determinar los niveles de respuesta de los atributos de calidad.	Desarrollar las estrategias arquitectónicas que aborden los escenarios elegidos, y determinar el nivel de la respuesta de los atributos de calidad. Esta respuesta es el resultado de la implementación de las estrategias arquitectónicas. Dado que una estrategia arquitectónica puede afectar a otros escenarios, este cálculo debe ser realizado para cada escenario afectado.
6. Determinar el valor de utilidad esperado de la estrategia arquitectónica utilizando interpolación	Empleando los valores de utilidad, obtenidos en paso anterior, determinar la utilidad del nivel de la respuesta esperada de los atributos de calidad para cada estrategia arquitectónica. Determinar esta utilidad para cada atributo de calidad relevante en el paso anterior.
7. Calcular el beneficio total obtenido de una estrategia arquitectónica.	Restar el valor de utilidad, del valor actual de los niveles esperados, y normalizar el uso de los votos que se obtuvieron con anterioridad. Sumar el beneficio de una determinada estrategia arquitectónica en todos los escenarios y atributos de calidad relevantes.
8. Elegir las estrategias arquitectónicas basándose en el ROI para las restricciones de costos.	Determinar el costo y las implicaciones de cada estrategia arquitectónica. Calcular el valor del ROI para cada estrategia expresado en términos del beneficio y costo. Ordenar las estrategias arquitectónicas de acuerdo al valor del ROI y elegir la que mejor se ajuste al presupuesto y calendario.
9. Confirmar los resultados con la intuición.	De las estrategias arquitectónicas elegidas, considerar si éstas cumplen con los objetivos del negocio. Si no es así, tener en cuenta los problemas que puedan haberse originado al hacer este análisis. Si existen problemas importantes, repetir todos los pasos del método.

Tabla 9 Pasos del método CBAM (2/2)

Para un estudio más detallado sobre este método se recomienda consultar a [PaulClements], [Rick Kazman] y [SEI2010].

2.4.4.6 Ciclo de vida de los métodos de análisis y diseño propuestos por el SEI

La Tabla 10 muestra la etapa en el ciclo de vida donde se aplican los cinco métodos previamente vistos. Además, en ésta tabla se indican los productos de trabajo que son entradas, o salidas para el método.

Etapas del ciclo de vida	QAW	ADD	ATAM	CBAM	ARID
Necesidades y restricciones del negocio	Entrada	Entrada	Entrada	Entrada	
Requerimientos	Entrada Salida	Entrada	Entrada Salida	Entrada Salida	
Diseño de la arquitectura		Salida	Entrada Salida	Entrada Salida	Entrada
Diseño detallado					Entrada Salida
Implementación					
Pruebas					
Implantación					
Mantenimiento				Entrada Salida	

Tabla 10 Métodos y etapas del ciclo de vida

2.5 Atributos de calidad

Cada estilo arquitectónico tiene sus ventajas y desventajas además de sus riesgos, por lo que escoger el estilo correcto para satisfacer las funciones requeridas y los atributos de calidad específicos es muy importante [KaiQian].

La arquitectura de software de un sistema se encarga de promover, hacer cumplir y predecir los atributos de calidad que un sistema deberá soportar. Los atributos de calidad son propiedades del sistema que van más allá de la funcionalidad del sistema y que hacen que el sistema sea un sistema bueno o malo desde el punto de vista técnico. Como los atributos de calidad son vistos desde la parte técnica, entran en la clasificación de requerimientos no funcionales y son identificados en el proceso de análisis de requerimientos.

2.5.1 Clasificación de los atributos de calidad

Los atributos de calidad, de acuerdo con [SQuaRE2009], son los que se muestran en la Figura 2-20.

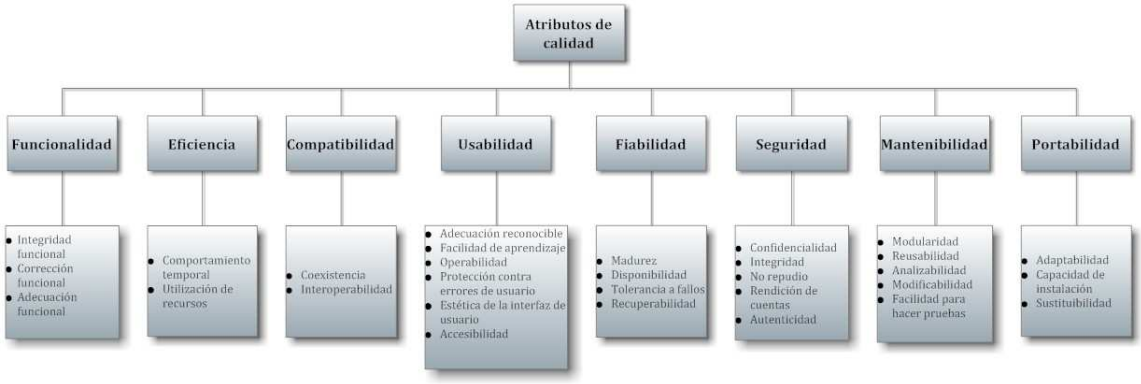

Figura 2-20 Atributos de calidad [SQuaRE2009]

A continuación se explica brevemente cada uno estos atributos de acuerdo con [SQuaRE2009].

2.5.1.1 Funcionalidad:

- **Integridad funcional:** es el grado para el cual los conjuntos de funciones cubren todas las tareas especificadas y los objetivos del usuario.
- **Corrección funcional:** es el grado para el cual el producto provee el correcto resultado con el grado necesitado de precisión.
- **Adecuación funcional:** es el grado para el cual las funciones son adecuadas para las tareas especificadas y los objetivos del usuario.

2.5.1.2 Eficiencia:

- **Comportamiento temporal:** es el tiempo de respuesta y de procesamiento y la tasa de rendimiento de un sistema cuando realiza sus funciones, bajo condiciones establecidas en relación a un punto de referencia establecido.
- **Utilización de recursos:** las cantidades y tipos de recursos utilizados cuando el producto cumple su función bajo condiciones establecidas en relación con un punto de referencia establecido.

2.5.1.3 Compatibilidad:

- **Coexistencia:** el grado en que el producto puede coexistir con otros productos independientes en un entorno común compartiendo recursos comunes sin ningún tipo de efectos perjudiciales.
- **Interoperabilidad:** el grado en que dos o más sistemas o componentes pueden intercambiar información y utilizar la información que se ha intercambiado.

2.5.1.4 Usabilidad:

- **Adecuación reconocible:** el grado en que los usuarios pueden reconocer si el producto es apropiado para sus necesidades.
- **Facilidad de aprendizaje:** el grado en que un producto puede ser utilizado por usuarios específicos para alcanzar las metas específicas de aprendizaje con eficacia, eficiencia, seguridad y satisfacción en un contexto de uso específico
- **Operabilidad:** el grado en que el producto tiene atributos que lo hacen fácil de operar y controlar.
- **Protección contra errores de usuario**: el grado en que el sistema protege a los usuarios contra los errores de decisiones
- **Estética de la interfaz de usuario:** el grado en que la interfaz de usuario permite una interacción agradable y satisfactoria para el usuario.
- ***Accesibilidad:*** el grado de eficacia, eficiencia, seguridad y satisfacción cuando la gente con la más amplia gama de capacidades de uso de un producto.

2.5.1.5 Fiabilidad:

- **Madurez:** el grado en que un sistema satisface las necesidades de fiabilidad en el funcionamiento normal.
- **Disponibilidad:** el grado en que un sistema o componentes está operativo y accesible cuando se requiere para el uso.
- **Tolerancia a fallos:** el grado en que un sistema o componente funciona según lo previsto a pesar de la presencia de fallas de hardware o software.
- **Recuperabilidad:** el grado en que el producto puede recuperar los datos directamente afectados y restablecer el estado deseado del sistema en caso de una interrupción o un fracaso

2.5.1.6 Seguridad:

- **Confidencialidad:** el grado en que la información y los datos están protegidos contra la divulgación no autorizada de información o datos, ya sea accidental o deliberada.
- **Integridad:** el grado en que un sistema o componente impide el acceso no autorizado o modificación de programas de un ordenador o de datos.
- **No repudio:** el grado en que las acciones o eventos se puede probar, de modo que los eventos o acciones no puedan ser repudiadas más tarde.
- **Rendición de cuentas:** el grado en que las acciones de una entidad puedan ser rastreadas únicamente a la entidad.
- **Autenticidad:** el grado en que la identidad de un sujeto o recurso puede ser probado para ser el reclamado.

2.5.1.7 Mantenibilidad:

- **Modularidad:** el grado en que un sistema o programa de computadora se compone de componentes discretos, de tal manera que un cambio en un componente tiene un impacto mínimo en otros componentes.

- **Reusabilidad:** el grado en que un activo puede ser utilizado en más de un sistema, o en la construcción de otros activos.
- **Analizabilidad:** a facilidad con que puede ser el impacto de un cambio intencionado en el resto del producto evaluado, o el producto puede ser diagnosticado por deficiencias o causas de los fracasos o las partes a modificar pueden ser identificadas.
- **Modificabilidad:** el grado en que un producto puede ser eficaz y eficiente modificada sin introducir defectos o disminuir el rendimiento.
- **Facilidad para hacer pruebas:** la facilidad con que los criterios de prueba se pueden establecer en un sistema o componente y las pruebas se pueden realizar para determinar si esos criterios se han cumplido.

2.5.1.8 Portabilidad:

- **Adaptabilidad:** el grado en que el producto eficaz y eficientemente adaptado para hardware diferente especificado, software u otros entornos operativos o de uso.
- **Capacidad de instalación:** la facilidad con que el producto puede ser instalado con éxito y desinstalado de en un entorno determinado.
- **Sustituibilidad:** el grado en que el producto puede ser utilizado en lugar de otro producto de software especificado para el mismo fin en el mismo entorno.

Es importante tener en cuenta que no es posible que algún sistema se optimice para todos los atributos de calidad vistos anteriormente; por ejemplo, mejorar la Mantenibilidad pude conducir a la pérdida de rendimiento. Por lo tanto, se deben de priorizar los atributos de calidad más importantes para el software que se va a desarrollar. De esta manera se sabrá que es más importante; considerar tácticas para el cumplimiento de la Mantenibilidad, o reducirlas a medida que se tenga el desempeño requerido.

Una suposición que se hace en la gestión de calidad de software, es que la calidad del software se relaciona directamente con la calidad del proceso de desarrollo del software [Sommerville 2011]. Como consecuencia tenemos que el proceso de desarrollo que se utilice tendrá una influencia muy importante sobre la calidad del software y, consecuentemente, los procesos que mantienen un seguimiento de los atributos de calidad tienen más probabilidad de conducir a software de buena calidad.

En la siguiente sección se exploran algunas de las tácticas que permiten alcanzar el éxito en el cumplimiento de los atributos de calidad, sin embargo, es importante mencionar que los atributos de calidad.

2.5.2 Tácticas para el cumplimiento de algunos atributos de calidad

El éxito en el cumplimiento de los atributos de calidad, se basa en la oportuna obtención de las decisiones fundamentales de diseño; también conocidas como tácticas [PaulClements]. De

acuerdo con [PaulClements], una táctica es una decisión de diseño que influye en el control de la respuesta de un atributo de calidad. Cada táctica es una opción de diseño para el arquitecto; por ejemplo, una de las tácticas es introducir redundancia para incrementar la disponibilidad de un sistema.

En la Tabla 11, Tabla 12, Tabla 13, Tabla 14, Tabla 15, Tabla 16, Tabla 17, Tabla 18, Tabla 19, Tabla 20, Tabla 21 y Tabla 22 se describen algunas de las principales tácticas para satisfacer los algunos de los atributos de calidad expuestos en [SQuaRE2009]. Un estudio más exhaustivo sobre las tácticas aquí expuestas, se puede encontrar en [PaulClements].

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Disponibilidad (1/2)	**Detección de fallas**	***Ping-Echo***	Esta táctica consiste en que un componente "A" emita una señal, denominada "Ping", hacia otro componente; componente "B". El componente receptor "B", al recibir la señal de "ping", deberá responder con otra señal; denominada "Echo". De esta manera el componente "A" estará probando la disponibilidad del componente "B".
		Hearbeat	Esta táctica consiste en que un componente emita una señal de manera periódica a otro componente que se encuentre escuchándole. Si el componente receptor deja de recibirse esta señal, se asume que el primer componente, que origina la señal, ha fallado.
		Exceptions	Un componente desencadena una excepción cuando una falla es encontrada
	Recuperación a fallas (1/2)	***Voting*:**	Son procesos que se ejecutan en procesadores redundantes, cada uno toma entradas equivalentes y calculan un valor de salida que se envía aun "votante". Si el "votante" detecta algún comportamiento desviado de algún procesador, entonces éste falla. El algoritmo, para la táctica de *voting,* puede ser el de "*majority rules*[1]" o algún otro algoritmo. Este método es utilizado para la corrección de fallas en la operación de un algoritmo o fallas de un procesador y es frecuentemente utilizado es sistemas de control.
		Active redundancy	Todos los componentes redundantes responden a eventos en paralelo. Consecuentemente, todos ellos en el mismo estado: La respuesta desde un solo componente es utilizada (usualmente el primero en responder), y el resto es descartado. Cuando se produce una falla, el tiempo de inactividad de los sistemas, que utilizan esta táctica, es de milisegundos el tiempo que toma conmutar de la copia de seguridad actual a la nueva. Esta táctica es comúnmente utilizada en la configuración cliente/servidor, tales como: administración de sistemas de bases de datos, donde la respuesta rápida es necesaria aun cuando ocurra una falla. Es necesario definir mecanismos de sincronización.

Tabla 11 Disponibilidad (1/2)

[1] *Majority rules,* es una regla para la toma de decisiones que selecciona las alternativas que tienen más de la mitad de votos.

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Disponibilidad (2/2)	**Recuperación a fallas (2/2)**	***Passive redundancy***	Uno de los componentes (el principal) responde a los eventos e informa a los otros componentes (los sustitutos) de las actualizaciones de estado que deben tomar. Cuando ocurre una falla en el sistema, el componente principal se debe asegurar de que el estado de las copias de seguridad (los componentes sustitutos) están lo suficientemente actualizados antes de reanudar los servicios. Se deben definir mecanismos de sincronización.
		Spare	Esta táctica consiste en la construcción de una plataforma con repuestos, la cual está configurada para sustituir muchos de los diferentes componentes que puedan fallar.
		Shadow operation	Un componente que falló, puede estar funcionando en modo sombra (*shadown mode*) por un corto periodo de tiempo para asegurarse de que imita el comportamiento del resto de los componentes antes de restaurarlo a servicio.
		State resynchronization	Cuando los componentes, en las tácticas de *active redundancy y passive redundancy,* se encuentran desactivados, deben tener actualizado su estado antes de regresar a funcionar. El enfoque que se utilice para actualizar los estados, dependerá en gran medida del tiempo de inactividad que pueden ser prolongados, el tamaño de la actualización y el número de mensajes requeridos para la actualización.
		Checkpoint/rollback	Ésta táctica consiste en registrar un estado consistente de un componente; ya sea periódicamente o en respuesta a eventos específicos. Cuando se produzca una falla en el sistema, se puede revertir a ese estado consistente.
	Prevención de fallas	***Removal from service***	Esta táctica consiste en remover un componente del sistema que se encuentra en operación, de tal manera que éste se someta a algunas actividades que le permitan anticipar fallas. Por ejemplo, reiniciar un componente para prevenir la fuga de memoria causada por una falla.
		Process monitor	Una vez que una falla en un proceso ha sido detectada, el proceso de monitoreo puede borrar el proceso que ha fallado y crear una nueva instancia de él, inicializándolo en un estado apropiado.

Tabla 12 Disponibilidad (2/2)

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Modificabilidad (1/5)	**Localización de modificaciones**	**Mantener la coherencia de la semántica**	La coherencia de la semántica, se refiere a la relación entre las responsabilidades de un módulo. Su meta es asegurar que todas las responsabilidades, del módulo, trabajan en conjunto sin una excesiva dependencia con otros módulos. Las métricas de acoplamiento y cohesión, son un intento para medir la coherencia semántica, y deben ser medidas contra un conjunto de cambios que se tengan por anticipado.
		Anticiparse a los cambios esperados	Teniendo en cuenta a un conjunto de cambios previstos, esta táctica consiste en que el arquitecto considere, durante la descomposición de los módulos, todos los cambios que puedan ocurrir. Para lo cual, el arquitecto se deberá preguntar: para cada cambio ¿La descomposición propuesta limita al conjunto de módulos que necesitan ser modificados para lograrlo? En otras palabras ¿Los cambios afectan a diferentes módulos? Asignar responsabilidades basándose en la coherencia semántica asume que los cambios esperados serán semánticamente correctos. La táctica para anticiparse a los cambios, no se preocupa en la coherencia de las responsabilidades de un modulo, en su lugar, se preocupa de minimizar los efectos de los cambios. En realidad esta táctica no se utiliza de manera aislada; ya que no es posible anticiparse a todos los cambios. Por esta razón, la táctica es empleada junto con la táctica de coherencia semántica.
		Generalizar un módulo	Esta táctica consiste en crear un modulo general, que pueda ser configurado para poder recibir distintos tipos de parámetros e inter operar de manera general con varios módulos.

Tabla 13 Modificabilidad (1/5)

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Modificabilidad (2/5)	**Prevención de la propagación de cambios (1/4)**	**Tácticas para la dependencia entre módulos**	Dentro de este conjunto de tácticas, encontramos ocho tipos: – Sintaxis de: – *Datos*. Para que el módulo B compile correctamente, el tipo de datos producido por el módulo A, y consumido por el modulo B, deben ser consistente con el tipo de datos asumidos por el módulo B. – *Servicios*. Para que el módulo B compile correctamente, la firma de los servicios provistos por el módulo A, e invocados por el módulo B, deben ser consistentes con lo asumido por el módulo B. – Semántica de: – Datos. Para que el módulo B compile correctamente, la semántica de los datos, producidos por A y consumidos por B, deben ser consistentes con lo asumido por el módulo B. – Servicios. Para que el módulo B compile correctamente, la semántica de los servicios producidos por el módulo A, y usados por el módulo B, deben ser consistentes con lo asumido por el módulo B. – Secuencia de: – Datos. Para que el módulo B se ejecute correctamente, éste debe recibir los datos producidos por el módulo A en un cierto orden. – Control. Para que el módulo B se ejecute correctamente, el módulo A debió haberse ejecutado previamente dentro de un tiempo preestablecido.

Tabla 14 Modificabilidad (2/5)

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Modificabilidad (3/5)	**Prevención de la propagación de cambios (2/4)**	**Tácticas para la dependencia entre módulos**	– Identidad de la interfaz de un módulo: El módulo A puede tener múltiples interfaces. Para que el módulo B compile correctamente, la identidad (nombre) de la interfaz debe ser consistente con la que el módulo B supone. – Localización de componentes: para que el módulo B se ejecute correctamente, la localización del módulo A debe ser consistente con lo que el módulo B supone. – Calidad del servicio de un módulo: para que el módulo B se ejecute correctamente, algunas de las propiedades de calidad del servicio del módulo A, deben ser consistente con lo que el módulo B supone. – Existencia de un módulo: Para que el módulo B se ejecute correctamente, el módulo A debe existir. – Comportamiento de los recursos de un módulo: para que el módulo B se ejecute correctamente, el comportamiento de los recursos del módulo A debe ser consistente con lo que el módulo B supone.

Tabla 15 Modificabilidad (3/5)

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Modificabilidad (4/5)	**Prevención de la propagación de cambios (3/4)**	**Ocultamiento de la información**	El ocultamiento de la información es la descomposición de las responsabilidades para una entidad (sistema o alguna descomposición de un sistema) en piezas más pequeñas. Durante ésta descomposición, se elige la información que será privada y la que será pública. El objetivo es aislar los cambios dentro de un módulo y prevenir la propagación de los cambios.
		Mantenimiento de las interfaces existentes	Si el módulo B depende del nombre y firma de una interfaz definida en el módulo A, mantener sin cambios su interfaz y su sintaxis, permite que no se modifique el módulo B.
		Restringir las vías de comunicación	Esta táctica se refiere a la restricción entre módulos, con los cuales, un módulo comparte información. Esto es, restringir el número de módulos que consumen los datos producidos por el módulo A y el número de módulos que producen datos consumidos por el mismo módulo A.
		Usar un intermediario	Si el módulo B tiene algún tipo de dependencia sobre el módulo A, diferente al de la semántica, es posible insertar un intermediario entre el módulo B y A que gestione las actividades asociadas con la dependencia. Algunos de los tipos de intermediarios se muestran a continuación: – Intermediarios de datos (sintaxis): Los repositorios, tanto el *blackboard* como el *passive (ver sección 2.3.4.5*), actúan como intermediarios entre el módulo productor y el módulo consumidor de datos. Los repositorios pueden convertir la sintaxis producida por el módulo A, a la sintaxis esperada por el módulo B.

Tabla 16 Modificabilidad (4/5)

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Modificabilidad (5/5)	**Prevención de la propagación de cambios (4/4)**	**Usar un intermediario**	– Intermediarios de servicios (sintaxis): Los patrones de diseño *facade*, *bridge*, *mediator*, *strategy*, proxy y *factory* fungen como intermediarios que convierten la sintaxis de un servicio en alguna esperada. – Intermediarios para la identidad de una interfaz de un módulo A: El patrón *bróker* pude ser utilizado para enmascarar los cambios en la identidad de una interfaz. Si el módulo B depende de la identidad de la interfaz del módulo A, y ésta identidad cambia, entonces al agregar dicha entidad al *bróker*, éste se encargará de realizar la conexión con la identidad del módulo A, obteniendo de esta manera que el módulo B permanezca sin cambios. – Intermediarios para la localización de un módulo (en tiempo de ejecución): Un servidor de nombres, permite la localización de un módulo A que será cambiado sin afectar al módulo B. El módulo A es el responsable de registrar su ubicación actual dentro del servidor de nombres, de tal manera que cuando el módulo B quiera conocer la ubicación del módulo A recuperará dicha ubicación en el servidor de nombres. – Intermediarios para el comportamiento de recursos de un módulo o recursos controlados por un módulo: esta táctica sugiere la implementación de un administrador de recursos, el cual el intermediario responsable de la asignación de recursos. – Intermediarios para la existencia de un componente A: El patrón de diseño *factory* tiene la habilidad de crear instancias, según sea necesario, y por lo tanto la dependencia del módulo B sobre la existencia del módulo A es satisfecha mediante la acción del patrón *factory*.

Tabla 17 Modificabilidad (5/5)

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Desempeño (1/2)	**Demanda de recursos**	**Incrementar la eficiencia computacional**	Al mejorar los algoritmos empleados en áreas críticas del sistema, se logra reducir la latencia (retardos producidos por el procesamiento de datos o eventos).
		Reducir la carga computacional	Si no existen peticiones para un recurso, las necesidades del procesamiento se ven reducidas. Por ejemplo, el uso de intermediarios (muy importante para mantener cierta modificabilidad) incrementa los recursos consumidos en el procesamiento de una secuencia de eventos. Por lo tanto, al reducir el número de intermediarios entre los módulos, se reduce la latencia.
		Manejar la tasa de eventos	Es posible reducir la frecuencia de muestreo, de las variables monitoreadas en un sistema, para reducir la demanda.
		Controlar la frecuencia de muestreo	Si no hay control sobre la llegada de eventos generados externamente, las peticiones, colocadas en una cola de espera, pueden ser muestreadas a una frecuencia menor. La desventaja de esta táctica, se encuentra en la posible pérdida de peticiones.
		Limitar tiempos de ejecución	Esta táctica consiste en definir el intervalo de tiempo para el cual un proceso puede responder a un evento.
		Limitar el tamaño de la cola de espera	Esta táctica controla el número máximo que acepta una cola de espera y como consecuencia se controla el numero de recursos empleados para procesar las peticiones.
	Administración de recursos (1/2)	**Introducir concurrencia**	Si las peticiones pueden ser atendidas en paralelo, el tiempo de bloqueo puede ser reducido. Se puede introducir concurrencia mediante el procesamiento de diferentes flujos de eventos en diferentes hilos, o mediante la adición de hilos; con el objetivo de procesar diferentes conjuntos de actividades.

Tabla 18 Desempeño (1/2)

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Desempeño (2/2)	**Administración de recursos (2/2)**	**Introducir concurrencia**	Si las peticiones pueden ser atendidas en paralelo, el tiempo de bloqueo puede ser reducido. Se puede introducir concurrencia mediante el procesamiento de diferentes flujos de eventos en diferentes hilos, o mediante la adición de hilos; con el objetivo de procesar diferentes conjuntos de actividades.
		Mantener múltiples copias de datos o procesos	El propósito de contar con múltiples réplicas, es el de reducir la competencia que pudiera ocurrir si todos los procesos son ejecutados en un servidor central.
		Incrementar la disponibilidad de los recursos	Elegir recursos como: los procesadores más rápidos, añadir más memoria, optar por otros discos duros y redes más rápidas tienen el potencial de reducir el tiempo de latencia. Sin embargo, el costo es usualmente una restricción para la consideración de estos recursos.
	Mediación de recursos	***First-in/First-out* (FIFO):**	Esta táctica consiste en utilizar colas de tipo FIFO para la administración de todas las peticiones realizadas hacia un recurso. Esta táctica es una buena solución siempre y cuando las peticiones tengan la misma prioridad.
		Planificación con la prioridad de recursos fija	Esta táctica consiste en: asignar a cada petición una prioridad de ejecución, y asignar los recursos de acuerdo con esa prioridad. Existen varias estrategias para la priorización de recursos, las cuales están fuera del alcance de este trabajo. Para mayor información, consulte a [PaulClements].

Tabla 19 Desempeño (2/2)

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
seguridad	**Resistencia a ataques**	**Autenticación de usuarios:**	La autentificación de usuario se asegura que un usuario, o una computadora remota, es quien dice ser. Por ejemplo, para la autentificación de usuario, se suele utilizar la validación de nombre de usuario y contraseñas, además, pueden agregarse algunos certificados digitales; por ejemplo, la firma electrónica.
		Autorización de usuarios	La autorización de usuarios se asegura que un usuario, previamente autentificado, tiene los derechos para acceder y modificar algunos datos o servicios.
		Mantener los datos confidenciales	Los datos deberían estar protegidos de accesos no autorizados. La confidencialidad se puede alcanzar empleando algún mecanismo de cifrado de datos y de comunicación (*virtual private network* y *secure sockets layer*).
		Mantener la integridad de los datos	Los datos deben ser entregados para lo que fueron creados. Pueden llevar información redundante en su codificación.
		Limitar la exposición	Los ataques típicamente dependen de la explotación de una sola debilidad para atacar todos los datos y servicios alojados en un servidor. El arquitecto pude diseñar la asignación de servicios a servidores, de tal manera que los servicios limitados estén disponibles en cada servidor
		Limitar el acceso	Esta táctica sugiere el uso de *firewalls* para restringir el acceso de los datos provenientes de clientes no deseados.
	Detección de ataques	**Detectores de intrusos**	La detección de un ataque se realiza mediante un sistema de detección de intrusos. Este sistema trabaja comparando patrones de tráfico de red, los cuales se encuentra registrados en una base de datos. En el caso de que se detecte un mal uso, se compara el patrón de peticiones con los almacenados en la base de datos (los históricos)
	Recuperación a ataques	**Restauración de estado**	Las tácticas empleadas para la restauración del estado de un sistema o de datos, son las que se emplean para abordar el problema de la disponibilidad de un sistema; ya que estas tácticas abordad la recuperación de un estado consistente, son adecuadas para la recuperación después de un ataque.
		Identificación de intrusos	La táctica consiste en mantener un registro para realizar una auditoría. Los registros deben contener una copia de cada transacción aplicada a los datos del sistema junto con la información de su identificación. La información de la auditoria puede utilizarse para rastrear las acciones de un atacante.

Tabla 20 Seguridad

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Facilidad para realizar pruebas	**Entrada y salida**	**Registro/Reproducción**	Se refiere a la captura de la información a través de una interfaz y utilizarla como entrada para las pruebas. La información a través de una interfaz, durante una operación normal, es guardad en algún repositorio y representa tanto la salida de un componente como la entrada de otro.
		Separar la interfaz de la implementación	Esta táctica consiste en separar la interfaz de la implementación; de tal manera, que se permita sustituir la implementación por alguna otra que tenga algún propósito de prueba.
		Especializar rutas o interfaces de acceso	Esta táctica consiste en tener interfaces especializadas para pruebas, que permitan la captura, o la especialización de los valores de las variables de un componente, a través de un instrumento de prueba. Además, su ejecución debe ser independiente.
	Monitoreo integrado	**Monitores incorporados**	El componente puede mantener el estado, la carga de rendimiento, capacidad, seguridad u otro tipo de información que sea accesible a través de una interfaz. Esta interfaz pude ser una interfaz permanente o se puede introducir temporalmente a través de una técnica de instrumentación, como la programación orientada a aspectos.

Tabla 21 Facilidad para realizar pruebas

Atributo de calidad	Categoría de la Táctica	Tácticas especifica	Descripción
Usabilidad	**Tácticas en tiempo de ejecución**	**Mantener un modelo de la tarea:**	Esta táctica se utiliza para determinar el contexto de lo que el usuario está intentando hacer y, en base a éste, proporcionar distintos tipos de asistencias. Por ejemplo, se sabe que las oraciones, por lo general, comienzan con letras mayúsculas, entonces la aplicación deberá brindar sugerencias sobre la manera de comenzar una oración.
		Mantener un modelo del usuario	Esta táctica usa un modelo que determina el conocimiento que un usuario tiene sobre el sistema, el comportamiento del usuario en términos del tiempo de respuesta esperado y otros aspectos específicos a un usuario o clases de usuarios. Por ejemplo, mantener un modelo del usuario permite al sistema navegar entre páginas, de un documento, a una velocidad adecuada para el usuario.
		Mantener un modelo del sistema	Esta táctica sugiere determina el comportamiento esperado por el sistema, de tal manera que el sistema puede dar la retroalimentación adecuada al usuario. El modelo del sistema se encargará de predecir cosas como: el tiempo necesario para completar la actividad actual.
	Tácticas en tiempo de diseño	**Separar la interfaz de usuario del resto de la aplicación**	Localizar los cambios esperados, es el objetivo principal de la coherencia semántica. Debido a que se espera que la interfaz de usuario cambie constantemente, durante el desarrollo y después de la implantación, se sugiere mantener el código separado para localizar los cambios de manera más rápida. Uno de los estilos arquitectónicos desarrollados para implementar esta táctica, y apoyar la modificación de la interfaz de usuario, es el modelo vista controlador (MVC).

Tabla 22 Usabilidad

2.5.3 Importancia de los estilos arquitectónicos para el cuidado de los atributos de calidad

En la sección 2.3.4 se hablo sobre los estilos arquitectónicos más comunes, ahora toca hablar de la importancia de los estilos arquitectónicos desde el punto de vista de los atributos de calidad.

Los estilos arquitectónicos son de suma importancia, porque cada estilo tiene un conjunto de atributos de calidad a los cuales este estilo en particular promueve, y al identificar el estilo que un diseño de arquitectura de software soporta, podemos verificar si la arquitectura es consistente con los requerimientos especificados e identificar que táctica podemos usar para tener una mejor implementación de la arquitectura.

Teóricamente un estilo arquitectónico es una abstracción de una estructura de software independiente del dominio. En la mayoría de los casos, un sistema de software tiene su propio dominio de aplicación, tales como el procesamiento de imágenes, el control de un motor, un portal web, sistemas expertos o un servidor de correo. Dicho modelo de referencia divide las funcionalidades de un sistema dentro de subsistemas o componentes de software. En muchos casos, un sistema puede adoptar arquitecturas heterogéneas, por ejemplo, más de un estilo arquitectónico puede coexistir en el mismo diseño. También es verdad que un estilo arquitectónico puede ser utilizado en muchos dominios de aplicación.

2.6 Importancia de la Arquitectura de software

El creador del método en espiral Barry Boehm [Boehm95] comenta lo siguiente acerca de la arquitectura de software:

Si un proyecto no ha logrado una arquitectura del sistema, incluyendo su justificación, el proyecto no debe empezar el desarrollo en gran escala. Si se especifica la arquitectura como un elemento a entregar, se la puede usar a lo largo de los procesos de desarrollo y mantenimiento.

Basándose en lo dicho por Barry Boehm, en [PaulCLindaNorth96] se dan tres razones que explican porque es importante la arquitectura de software.

1. **Comunicación mutua**. La arquitectura de software representa un alto nivel de abstracción común que la mayoría de los participantes (stakeholders), si no todos, pueden usar como base para crear entendimiento mutuo, formar consenso y comunicarse entre sí. En sus mejores expresiones, la descripción arquitectónica expone las restricciones de alto nivel sobre el diseño del sistema, así como la justificación de decisiones arquitectónicas fundamentales.
2. **Decisiones tempranas de diseño**. La arquitectura de software representa la encarnación de las decisiones de diseño más tempranas sobre un sistema, y esos vínculos tempranos

tienen un peso fuera de toda proporción en su gravedad individual con respecto al desarrollo restante del sistema, su servicio en el despliegue y su vida de mantenimiento.

3. **Reutilización, o abstracción transferible de un sistema**. La AS encarna un modelo relativamente pequeño, intelectualmente tratable, de la forma en que un sistema se estructura y sus componentes se entienden entre sí; este modelo es transferible a través de sistemas; en particular, se puede aplicar a otros sistemas que exhiben requerimientos parecidos y puede promover reutilización en gran escala. El diseño arquitectónico soporta reutilización de grandes componentes o incluso de frameworks en el que se pueden integrar componentes.

En [Gar00] se explican las siguientes:

1. **Comprensión del problema**. La arquitectura de software simplifica nuestra capacidad de comprender los grandes sistemas, mediante la presentación de ellos a un nivel de abstracción en la que se puede diseñar un sistema de alto nivel fácil de entender. Por otra parte, la descripción arquitectónica expone las limitaciones de alto nivel sobre el diseño del sistema, así como la justificación para la toma de decisiones específicas de arquitectura.
2. **Reutilización.** La descripción arquitectónica apoya la reutilización en múltiples niveles. El trabajo actual sobre la reutilización se centra generalmente en bibliotecas de componentes. El diseño arquitectónico apoya, además, tanto la reutilización de piezas de gran tamaño como también los marcos en los que los componentes se pueden integrar.
3. **Construcción.** Una descripción arquitectónica proporciona un plan parcial para el desarrollo mediante la indicación de los principales componentes y dependencias entre ellos. Por ejemplo, una vista de capas de una arquitectura típicamente documentos límites de la abstracción entre las partes de la aplicación de un sistema, identificando claramente las interfaces internas de gran magnitud del sistema, y limitar lo que las partes de un sistema puede depender de los servicios prestados por otras partes.
4. **Evolución**. La AS puede exponer las dimensiones a lo largo de las cuales puede esperarse que evolucione un sistema. Haciendo explícitas estas “paredes” perdurables, quienes mantienen un sistema pueden comprender mejor las ramificaciones de los cambios y estimar con mayor precisión los costos de las modificaciones. Esas delimitaciones ayudan también a establecer mecanismos de conexión que permiten manejar requerimientos cambiantes de interoperabilidad, prototipado y reutilización.
5. **Análisis**. Las descripciones arquitectónicas aportan nuevas oportunidades para el análisis, incluyendo verificaciones de consistencia del sistema, conformidad con las restricciones impuestas por un estilo, conformidad con atributos de calidad, análisis de dependencias y análisis específicos de dominio y negocios.

6. **Administración**. La experiencia demuestra que los proyectos exitosos consideran una arquitectura viable como un logro clave del proceso de desarrollo industrial. La evaluación crítica de una arquitectura conduce típicamente a una comprensión más clara de los requerimientos, las estrategias de implementación y los riegos potenciales.

2.6.1 Importancia de la arquitectura de software desde el punto de vista técnico

En [PaulClements] se expone la importancia de la arquitectura de software desde el punto de vista Técnico. A continuación se describen estos puntos de vista:

1. ***Comunicación entre los stakeholders****: l*a arquitectura de software representa una abstracción común de un sistema que la mayoría si no es que todo el sistema de los stakeholders pueden usar como una base para entenderse mutuamente, negociar, consensar y comunicar.

2. ***Decisiones de diseño tempranas****:* la arquitectura de software manifiesta las decisiones de diseño tempranas acerca del sistema.

3. ***Abstracción transferible de un sistema:*** la arquitectura de software constituye un modelo relativamente pequeño e intelectualmente aprehensible sobre como un sistema está estructurado y como sus elementos trabajan juntos, y este modelo es transferible a través del sistema. En particular, la arquitectura de un sistema puede ser aplicada a otro sistema que exhiba atributos de calidad y requerimientos funcionales similares, además que se promueve al reúso a gran escala.

3 Métodos ágiles

Las metodologías agiles son un conjunto de metodologías, en ocasiones llamas livianas o ligeras, que en general utilizan prácticas muy similar, basadas en los resultados, la gente y la interacción.

En febrero del 2001 un grupo interesado en métodos iterativos y agiles, se reunieron con el fin de encontrar puntos en común de las prácticas ya existentes. Como resultado surgió el manifiesto ágil, el cual tiene cuatro postulado y se citan a continuación [ManifiestoAgil]:

Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- *Individuos e interacciones sobre procesos y herramientas*
- *Software funcionando sobre documentación extensiva*
- *Colaboración con el cliente sobre negociación contractual*
- *Respuesta ante el cambio sobre seguir un plan*

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.

Los precedentes postulados pueden llevarse a la práctica aplicando los principios agiles. Los principios agiles ayudan a entender mejor la esencia de los postulados del manifiesto ágil. A continuación se muestran los principios del manifiesto ágil [ManifiestoAgil]:

Principios del Manifiesto Ágil

1. Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
2. Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
3. Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
4. Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
5. Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
7. El software funcionando es la medida principal de progreso.
8. Los procesos Ágiles promueven el desarrollo sostenible.
9. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
10. La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
11. La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
12. Las mejores arquitecturas, requisitos y diseños emergen de equipos auto organizados.
13. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

En la actualidad existen varias metodologías ágiles, pero muchas de ellas comparten prácticas muy similares, por lo que en este trabajo sólo se verán algunas de ellas. En particular, se hará una síntesis de dos de los métodos ágiles con mayor impacto en el desarrollo de software; "*Extreme Programming* y *Scrum.* Para un estudio más detallado sobre metodologías agiles se recomienda revisar la referencia [AlianzaAgil].

3.1 Comparación entre los métodos ágiles y tradicionales

La Tabla 23 y la Tabla 24 muestran la comparación entre los métodos ágiles y los métodos tradicionales (dirigidos por un plan).

Métodos tradicionales o dirigidos por un plan	Métodos ágiles
Un enfoque basado en un plan para la ingeniería de software identifica etapas separadas en el proceso de software con salidas asociadas a cada etapa. Las salidas de una etapa se usan como base para planear la siguiente actividad del proceso.	Los enfoques ágiles en el desarrollo de software consideran el diseño y la implementación como las actividades centrales en el proceso del software. Incorporan otras actividades en el diseño y la implementación, como la adquisición de requerimientos y pruebas
La iteración ocurre dentro de las actividades con documentos formales usados para comunicarse entre etapas del proceso. Por ejemplo, los requerimientos evolucionarán y, a final de cuentas, se producirá una especificación de aquéllos. Esto es entonces una entrada al proceso de diseño y la implementación.	La iteración ocurre dentro de las actividades. Por lo tanto, los requerimientos y el diseño se desarrollan en conjunto, no por separado.
Soporta el desarrollo y la entrega incrementales. Es perfectamente factible asignar requerimientos y planear tanto la fase de diseño y desarrollo como una serie de incrementos.	Un proceso ágil no está inevitablemente enfocado al código y puede producir cierta documentación de diseño.

Tabla 23 Comparación entre los métodos ágiles y tradicionales [Sommerville 2011]

Parámetro de comparación	Métodos ágiles	Métodos tradicionales o dirigidos por un plan
Desarrolladores	Ágiles, eficientes y colaborativos	Dirigidos por un plan, habilidades adecuadas, acceso a conocimientos externos
Cliente	Dedicados, con conocimientos del dominio, colaborativos, representantes del negocio y con poder para tomar decisiones sobre el proyecto.	Colaborativos, representantes del negocio y con poder para tomar decisiones sobre el proyecto.
Requerimientos	De cambios rápidos, en gran medida emergentes.	Se conocen de manera temprana y en gran parte estables.
Arquitectura	Diseñada para las necesidades actuales	Diseñada para las necesidades actuales y las previsibles
Refactorización	No costoso	Costoso
Tamaño del equipo	Para pequeños equipos y productos	Para grandes equipos y productos
Objetivo principal	Valor rápido	De alta seguridad

Tabla 24 Comparación entre los métodos ágiles y tradicionales [Boehm2002]

La Figura 3-1 muestra la comparación entre el ciclo de vida de un modelo de cascada, un representante de los procesos tradicionales, y el ciclo de vida de un modelo ágil.

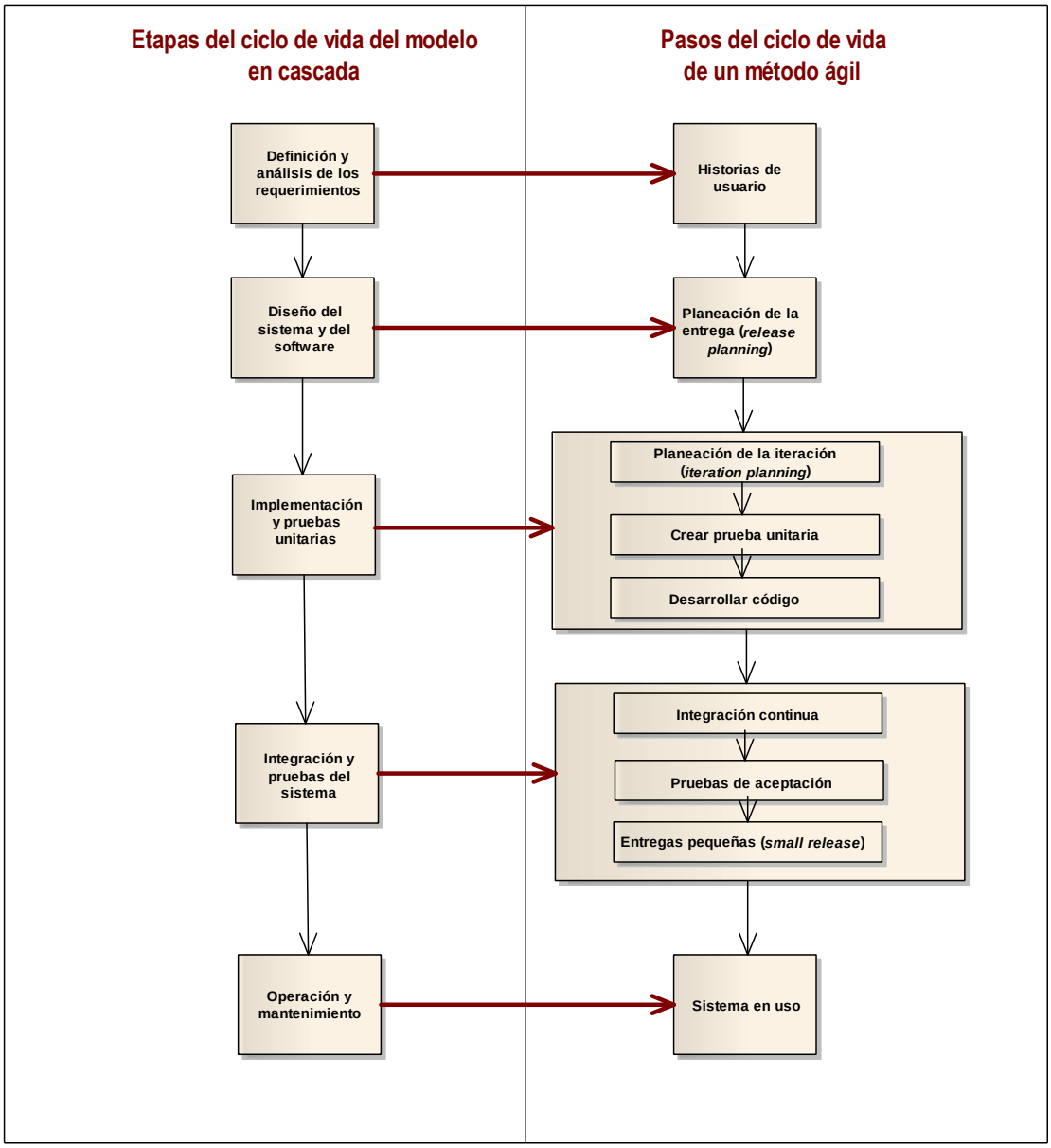

Figura 3-1 Comparación del modelo en cascada con los métodos ágiles [MingHuo]

La mayoría de los proyecto de software incluyen prácticas tanto de los enfoques ágil como de los tradicionales [Sommerville 2011]. Para decidir sobre el equilibrio entre un enfoque y otro, se deberá hacer un análisis del contexto del proyecto.

3.2 Ejemplos de metodologías ágiles

A continuación se da una breve descripción de algunas de las principales metodología ágiles que existen en la actualidad. Posteriormente, nos enfocaremos en una descripción más amplia sobre la metodología *Scrum y Extreme Programming*.

3.2.1 Extreme programming

Extreme *Programming* es quizá una de las metodologías agiles más conocidas y empleadas por muchos desarrolladores ágiles [DonWells]. Esto se debe principalmente al gran impulso que se le ha dado en la comunidad de desarrolladores y a los libros de su principal referente, Kent Beck [BeckKent]. Debido a su gran importancia para el presente trabajo, en la sección 3.4 se describe esta metodología.

3.2.2 Scrum

Scrum se ha utilizado para desarrollar productos complejos desde principios de los años noventa. *Scrum* no es un proceso o una técnica para desarrollar o crear productos, sino que es un marco en el que se pueden emplear diversos procesos y técnicas. El papel de *Scrum* es hacer aflorar la eficacia relativa de las prácticas de desarrollo empleadas por los desarrolladores, para que puedan mejorarlas, a la vez que proporciona un marco dentro del cual se pueden desarrollar productos complejos [ScrumOrg]. Debido a su gran importancia, en la sección 3.5 se detalla esta metodología.

3.2.3 Crystal Methodologies

Crystal Methodologies [Cockburn] son una familia de metodologías desarrolladas por Alistar Cockburn[2]. La elección de cada una de estas metodologías está en función de las necesidades del proyecto en que nos encontremos. *Crystal* incluye un conjunto de principios para adaptar las diferentes metodologías acorde a las circunstancias del proyecto. Cada una de las familias de Crystal tiene un color asignado de acuerdo a la complejidad de la metodología. Cuando más oscura se la tonalidad del color más compleja es la metodología, por ejemplo: Crystal Clear (3 a 8 miembros) y Crystal Orange (25 a 50 miembros).

3.2.4 Dynamic Systems Development Method

Dynamic System Development Method (DSDM), tiene sus inicios en el año de 1994 y desde entonces se ha convertido en marco de trabajo para el desarrollo rápido de aplicaciones.

[2] http://alistair.cockburn.us/Crystal+methodologies

Actualmente del DSDM es mantenido por el *Consortium*[3]. El proceso del DSDM está dividido en cinco fases, a saber: estudio de vialidad, estudio de negocio, modelo funcional de las iteraciones, interacciones de diseño y construcción e implementación. La idea fundamental de esta metodología es fijar primero el tiempo y el coste para después determinar las funcionalidades que se pueden implementar en el producto.

3.2.5 Feature Driven Development

Feature Driven Development (FDD) es un proceso diseñado que está probado para realizar entregas frecuentes y tangibles de trabajo de software [Stephen R]. FDD es un método enfocado en las construcciones del sistema con un constante énfasis en cuestiones de calidad y define claramente las formas de evaluación del progreso [Stephen R]. El método consiste de cinco pasos, durante los que se diseña y construye el sistema:

- Desarrollo del modelo general.
- Construcción de la lista de características o rasgos.
- Diseño por características o rasgos.
- Construcción de la característica o rasgo.

A diferencia de otros procesos ágiles, no cubre todo el ciclo de vida sino sólo las fases de diseño y construcción por lo que se suele utilizar como complemento de otras metodologías.

3.2.6 Adaptive Software Development

Adaptive Software Development (ASD) es una metodología desarrollada por Jim Highsmith[4], actual consultor de *Cutter Consortium*, que se basa en la constante adaptación a las circunstancias cambiantes de un proyecto en lugar de oponerse a ellas. No tiene un determinado ciclo para la planificación, diseño y construcción del software sino que cuenta con un ciclo dividido en tres etapas, a saber:

- Especular: se inicia el proyecto y se planifica las características del software.
- Colaborar: se desarrollan las características.
- Aprender: se revisa su calidad y se entrega al cliente.

Básicamente este método se enfoca en dos cosas, el equipo y la tolerancia a los cambios.

3.2.7 Lean Development

[3] http://www.dsdm.org/
[4] http://www.adaptivesd.com/

El Desarrollo de Software Lean[5] tiene sus inicios en el Sistema de Producción de Toyota (TPS) y ayuda a las organizaciones de software a optimizar sus procesos y sus métodos de producción de manera de poder entregar sus productos al mercado de manera más rápida y con mejor calidad. El movimiento Lean puede considerarse como un nuevo método de desarrollo que intenta identificar y erradicar todos los problemas y "desventajas" de metodologías antiguas, como Cascada.

Lean hace énfasis en las personas y la comunicación si se respeta a las personas que producen software y se pueden comunicar de forma eficiente, es más probable que logren entregar un buen producto y satisfacer al consumidor final.

3.3 Adopción de los métodos ágiles en la industria del software

La Tabla 25 muestra las empresas que utilizan métodos ágiles para el desarrollo de software de acuerdo con [AgilOrg].

Sector	Empresas que utilizan metodos ágiles como Scrum
Media y Telcos	BBC, BellSouth, British Telecom, DoubleYou, Motorola, Nokia, Palm, Qualcomm,Schibsted, Sony/Ericsson, Telefonica I+D, TeleAtlas, Verizon
Software, Hardware	Adobe, Autentia, Biko2, Central Desktop, Citrix, Gailén, IBM, Intel, Microfocus, Microsoft, Novell, OpenView Labs, Plain Concepts, Primavera, Proyectalis,Softhouse, Valtech, VersionOne.
Internet	Amazon, Google, mySpace, Yahoo
ERP	SAP
Banca e Inversión	Bank of America, Barclays Global Investors, Key Bank, Merrill Lynch
Sanidad y Salud	Patientkeeper, Philips Medical
Defensa y Aeroespacial	Boeing, General Dynamics, Lockheed Martin
Juegos	Blizzard, High Moon Studios, Crytek, Ubisoft, Electronic Arts
Otros	3M, Bose, GE, UOC, Ferrari

Tabla 25 Métodos ágiles en la industria [AgilOrg]

3.4 Extreme Programming

La presente sección no trata de ser una guía para el desarrollo de *Extreme Programming* (de aquí en adelante nos referiremos a *Extreme Programming* como XP) ni mucho menos ser un manual de la misma, más bien, se exploran las prácticas y principios que rodean XP, con el fin de dar una introducción y primera comprensión del tema. En la actualidad existe una vasta gama de artículos que pueden ser consultados para profundizar aún más en este tema; ejemplo de esto se puede encontrar en [BeckKent], cuyo autor es uno de los creadores de XP; [XPRonaldJeffries], el cual es

[5] http://www.poppendieck.com/

el sitio web de otro de los creadores de *Extreme Programming*; [DonWells], página dedicada a *Extreme Programming* basada en el libro "*Extreme programming explained*" de Kent Beck.

3.4.1 Definición de *Extreme Programming*

Extreme Programming fue creado por Kent Beck [BeckKent], Ward Cunningham y Ron Jeffries [XPRonaldJeffries]. Es un método de desarrollo iterativo e incremental que enfatiza en la satisfacción del cliente a través de la rápida creación de software de alto valor y técnicas de desarrollo de software sostenibles. A diferencia de las metodologías tradicionales, *Extreme Programming* está orientado al desarrollador, al usuario, al resultado real y al incremento de la funcionalidad.

Extreme Programming es quizá una de las metodologías agiles que más trascendencia tiene en la actualidad. Enfatiza en la colaboración, creación temprana y rápida de software y en prácticas hábiles de desarrollo [CraigLarman].

Extreme Programming se basa en cuatro valores: comunicación, simplicidad, retroalimentación y coraje [BeckKent]. Adicionalmente para obtener un desarrollo iterativo e incremental se recomiendan 12 prácticas [CraigLarman]:

1. Planificación del juego
2. Entregas pequeñas y frecuentes
3. Sistema de metáforas
4. Diseño simple
5. Prueba
6. Refactorización frecuente
7. Programación en pares
8. Equipo de propiedad de código
9. Integración continua
10. Ritmo sostenible
11. Todo el equipo junto
12. Normas de codificación

A continuación se explicará brevemente en qué consiste *Extreme Programming*, haciendo mención de sus prácticas y basándose en [XPRonaldJeffries].

Esta sección está dividida en:

- *Extreme Programming* (XP)

    1. Valores de XP
    2. Principales prácticas
    3. Ciclo de vida de un proyecto *Extreme programming*
    4. Roles de XP

3.4.2 Valores de XP

A continuación se describen los cuatro valores en los cuales se basa XP de acuerdo con [BeckKent] y [DonWells].

3.4.2.1 Comunicación

El primer valor de XP, es la comunicación. Los problemas con el proyecto pueden no ser transmitidos a otros miembros del equipo, ejemplo de esto es cuando ocurre un cambio crítico en el sistema y uno de los programadores responsables del cambio no comenta nada al respecto con el equipo de trabajo, pudiendo surgir serios problemas para el proyecto. *Extreme Programming* fomenta la buena comunicación entre los involucrados en el proyecto, fomentando la comunicación sin barreras (cara a cara) y constante para evitar las inaccesibles cadenas de solicitaciones.

3.4.2.2 Simplicidad

El segundo valor en XP es la simplicidad y es una de las bases principales en XP. Mediante la simplicidad, XP espera a que se afronten los problemas con la pregunta: ¿Cuál es la forma más simple de hacer el trabajo para que nos den los resultados esperados?

La simplicidad no es fácil[BeckKent], debido a que hoy podemos dar la solución a un problema, pero ¿esa solución podrá seguir funcionando en una semana más adelante?, o peor aún, ¿Seguirá funcionando en un mes más, cuando los requisitos hayan cambiado?, esto porque no debemos olvidar que XP, es una metodología ágil y como tal está basada en el manifiesto ágil, manifiesto que le obliga a cumplir con sus principios y uno de ellos es estar siempre abiertos al cambio, sin importar que éste modifique nuestra solución simple que se pudo haber pensado en algún momento. Sin embargo, pensar en la solución más simple en un ambiente en donde los requerimientos pueden estar cambiando constantemente no resulta tan incorrecto, ya que podemos pensar el caso en el cual se toma una solución más sofisticada, solución que demandará recursos de la empresa, tiempo y dinero. Tomando en cuenta este escenario, en el mejor de los casos los requerimientos no cambiarán y nuestra solución será con mucha seguridad satisfactoria, más sin embargo esto no ocurrirá así, los requerimientos cambiarán y en muchos de los casos la solución se tendrá que desechar, haciendo una perdida bastante costosa, no así si la solución fuese la más simple.

Para XP, la simpleza ayuda a evitar gastos innecesarios, incluyendo la forma de desarrollar, las herramientas empleadas, el tiempo y esfuerzo dedicado. Si hablamos desde el punto de vista técnico la simpleza se refiere a hacer el diseño lo más simple posible, así garantizando los tiempos de desarrollo y mantenimiento del sistema. Dentro de las prácticas que XP utiliza para mantener la simpleza encontramos a la refactorización y programación en pares.

3.4.2.3 Retroalimentación

El tercer valor en XP es la retroalimentación. La retroalimentación permite conocer el estado del proyecto rápidamente, como ejemplo tenemos el caso del cliente, que al estar integrado en el proyecto ofrece su opinión del sistema inmediatamente. Otro ejemplo es el código, el cual también puede ser una fuente de retroalimentación que mediante su comprobación en pruebas unitarias, permite encontrar errores que se pueden deber a cambios recientes en el código. Todos los clientes, desarrolladores y miembros del equipo deben informar sobre lo que están haciendo al equipo para permitirle al equipo ver en dónde están y para ajustar las prácticas a su situación única.

3.4.2.4 Coraje

El coraje y la valentía, se enfocan a atributos que deberán tener los directivos del proyecto. En realidad este valor que XP alienta, es debido a los tres primeros valores: comunicación, simpleza y retroalimentación, en los cuales se deberá creer, puesto que muchas de las prácticas que propone XP, se encuentran alejadas de las tradicionales y cuesta obtener indicadores reales sobre su rendimiento para poder creer en ellas, lo que dificulta su adopción.

3.4.3 Prácticas Principales

En esta sección se exponen las prácticas principales que articulan a XP de acuerdo con [XPRonaldJeffries].

3.4.3.1 Equipo completo

Todos los involucrados en un proyecto XP se sientan juntos y son miembros de un mismo equipo**.** El equipo tiene que incluir a un representante del negocio "el cliente" quien provee requerimientos, establece prioridades y guía al proyecto. Lo mejor es que el Cliente o uno de sus asistentes sea el usuario final que conoce al dominio y lo que necesita. Por supuesto, el equipo incluye a los programadores. El equipo puede incluir un *testers*, que ayude al cliente a definir las pruebas de aceptación del cliente. Los analistas pueden servir como asistentes del cliente, ayudándolo a definir los requerimientos. Suele haber un *coach*, que ayuda al equipo a mantener el rumbo y facilitar el proceso. Puede haber un *manager*, que brinda recursos, se encarga de la comunicación externa y coordina actividades. Ninguno de estos roles es propiedad exclusiva de una persona: todos en un equipo XP contribuyen de la manera que pueden. Los mejores equipos no tienen especialistas, sino contribuyentes generales con habilidades especiales.

3.4.3.2 Planificación

La planificación en XP responde dos preguntas clave del desarrollo de software: predecir qué se habrá terminado para la fecha de entrega y determinar qué hacer después. Se hace énfasis en guiar al proyecto, que es bastante sencillo, en vez de predecir exactamente lo que se necesitará y cuánto tiempo tomará hacerlo que es bastante difícil. Hay dos pasos claves en la planificación de XP, que responde estas dos preguntas:

Planificación de la Entrega, es una práctica en donde el Cliente presenta las características deseadas a los programadores, y los programadores estiman la dificultad. Teniendo las estimaciones de costo, y sabiendo la importancia de las características, el Cliente establece un plan para el proyecto. Los planes iniciales de entregas son necesariamente imprecisos: ni las prioridades ni las estimaciones son sólidas, y tampoco sabremos qué tan rápido trabaja el equipo hasta que empiece a trabajar. Sin embargo, incluso el primer plan de entrega es lo suficientemente preciso como para tomar decisiones, y el equipo XP revisa de forma regular el plan.

Planificación de la Iteración, es la práctica en donde el equipo establece el rumbo cada un par de semanas. Los equipos XP construyen software en iteraciones de dos semanas, y entregan software útil al finalizar cada iteración. Durante la Planificación de la Iteración, el Cliente presenta las características deseadas para las siguientes dos semanas. Los programadores las descomponen en tareas, y estiman su costo (a un nivel de detalle más fino que durante la Planificación de la Entrega). El equipo entonces se compromete a terminar ciertas características basándose en la cantidad de trabajo que pudieron terminar en la iteración anterior.

Estos pasos de planificación son muy simples, y le brindan al cliente muy buena información y excelente flexibilidad para guiar al proyecto. Cada dos semanas se hace completamente visible el progreso. No existe el "90% terminado" en XP: una historia está terminada, o no lo está. Este enfoque resulta en una paradoja: por un lado, con tanta visibilidad, el cliente está en la posición de cancelar el proyecto si el progreso no es suficiente. Por otro lado, como el progreso es tan visible, y hay completa libertad para decidir qué se hará después, los proyectos XP tienden a entregar más de lo necesario, con menos presión y estrés.

3.4.3.3 Pruebas automatizadas

Como parte de la presentación de cada característica deseada, el Cliente también define una o más pruebas de aceptación automatizadas para mostrar que la característica funciona. El equipo construye estas pruebas y las utiliza para demostrar al cliente y a ellos mismos que la característica se implementó de forma correcta. La automatización es importante porque, por la presión del tiempo, se suelen saltear las pruebas manuales. Y eso es como apagar las luces cuando se viene la noche**.**

Los mejores equipos XP tratan a las pruebas del cliente de la misma manera que a las pruebas de los programadores: una vez que la prueba se ejecuta bien, el equipo la mantiene corriendo bien de ahí en adelante. Esto significa que el sistema sólo mejora, siempre avanzando, nunca retrocediendo.

3.4.3.4 Entregas

Los equipos XP realizan entregas pequeñas de dos formas importantes:

Primero, el equipo entrega software probado y funcionando, que entrega el valor elegido por el Cliente, en cada iteración**.** El Cliente puede usar este software para cualquier propósito, sea como evaluación o incluso liberarlo para los usuarios finales (muy recomendado). El aspecto más importante es que el software sea visible y entregado al cliente, al final de cada iteración. Esto hace que todo se mantenga abierto y tangible.

Segundo, los equipos XP también entregan software de forma frecuente a los usuarios finales. Los proyectos web XP entregan a diario, los proyectos internos de forma mensual o más seguida. Incluso los productos enlatados se entregan cuatrimestralmente.

Puede parecer imposible crear buenas versiones tan seguidas, pero los equipos XP lo hacen todo el tiempo. Pueden ver la Integración Continua para más detalles, y nos daremos cuenta que estas entregas frecuentes son confiables por la obsesión de XP por las pruebas, como pueden ser las pruebas del cliente y el desarrollo guiado por pruebas (TDD).

3.4.3.5 Diseño simple

Los equipos de XP construyen software sobre un diseño simple. Empiezan simple, y a través de las pruebas de programación y las mejoras al diseño, lo mantienen así. Cualquier equipo XP mantiene al diseño para que sea el justo y necesario para cumplir la funcionalidad actual del sistema. No hay desperdicio, y el software siempre está listo para lo que sigue.

El diseño en XP no es algo de "única vez", o que se hace completo al principio, sino que es algo de todo momento**.** Hay diseño durante la planificación de la entrega y la planificación de la iteración; además, los equipos hacen sesiones rápidas de diseño y revisiones a través de la refactorización durante todo el proyecto. Resulta esencial tiene un buen diseño en los procesos iterativo incremental, como *Extreme Programimng*. Por esto se hace tanto énfasis en el diseño a través de toda la vida del proyecto.

3.4.3.6 Programación en pares

En XP todo el software productivo se escribe en pareja, dos programadores sentados lado a lado en una misma computadora**.** Esta práctica asegura que todo el código productivo fue revisado por al menos otro programador, y genera mejores diseños, mejores pruebas y mejor código.

Puede parecer ineficiente que dos programadores hagan el "trabajo de un programador", pero lo contrario es cierto. Los estudios sobre la programación de a pares muestran que las parejas producen mejor código en aproximadamente el mismo tiempo que un programador trabajando solo. Es cierto: ¡dos cabezas son mejor que una!

Muchos programadores se niegan a la programación de en pares antes de probarla. Se necesita tiempo y práctica para hacerlo bien, y es necesario aplicarlo por varias semanas para ver los resultados. El 90% de los programadores que aprenden a trabajar de a pares lo prefieren, por lo que la programación en pares es una práctica recomendada para todos los equipos**.**

Además de generar mejor código y pruebas, la programación de a pares también sirve para comunicar el conocimiento a través de los equipos**.** Como las parejas cambian, todos obtienen los beneficios del conocimiento especializado de las personas. Los programadores aprenden, mejoran sus habilidades, se vuelven más valiosos para el equipo y para la organización. La programación de a pares, incluso por fuera de XP, es una ventaja para todos.

3.4.3.7 Desarrollo guiado por pruebas (TDD)

Extreme Programming se obsesiona con la retroalimentación (*"feedback*") y en el desarrollo de software una buena retroalimentación necesita de buenas pruebas. Los mejores equipos de XP practican el Desarrollo Guiado por Pruebas (*test-driven development* o TDD), trabajando en ciclos muy cortos para agregar una prueba y después hacerla funcionar. Casi sin esfuerzo, los equipos producen código con casi un 100% de cobertura de pruebas, lo cual es un gran avance para la mayoría de los lugares.

No alcanza con escribir pruebas: debemos ejecutarlas. Aquí también XP es extremo. Estas "pruebas del programador" o "pruebas unitarias" se juntan, y cada vez que cualquier programador sube código al repositorio (y las parejas lo suelen hacer dos o más veces al día), cada una de las pruebas unitarias debe correr con éxito. Esto significa que los programadores tienen una retroalimentación inmediata sobre cómo están avanzando. Además, estas pruebas brindan un apoyo invaluable a medida que mejora el diseño del software.

3.4.3.8 Refactorización

Extreme Programming se enfoca en entregas por cada iteración. Para lograr esto a lo largo de todo el proyecto, el software debe estar bien diseñado. La alternativa es retrasarse hasta detenerse por completo. Es por esto que XP utiliza un proceso de mejora continua del diseño llamado *Refactoring*, sacado del libro de Fowler "*Refactoring: Improving the Design of Existing Code*".

El proceso de *refactoring* se enfoca en eliminar la duplicación (una clara señal de diseño pobre) y en incrementar la cohesión del código, disminuyendo el acoplamiento. Hace ya treinta años que se considera a la alta cohesión y al bajo acoplamiento como el máximo logro de un buen diseño. El resultado es que los equipos XP comienzan con un diseño simple y bueno y siempre tienen un

diseño simple y bueno para el software. Esto les permite mantener la velocidad y en general suele acelerar la velocidad a medida que el proyecto avanza.

El *refactoring* se apoya en pruebas completas que aseguran que, a medida que el diseño evoluciona, nada se rompa. Las pruebas del cliente y unitarias son críticas para permitir el *refactoring*. Las prácticas de XP se apoyan entre sí: son más poderosas que por separado**.**

3.4.3.9 Integración continua

Los equipos mantienen integrado al sistema todo el tiempo. Los equipos XP realizan construcciones muchas veces por día.

El beneficio de esta práctica lo vemos al pensar en esos proyectos que todos oímos (o incluso fuimos parte) en donde la construcción se realizaba semanalmente (o incluso con menos frecuencia), y que usualmente termina en un "infierno de integración", donde todo se rompe y nadie sabe porqué.

La integración infrecuente lleva a problemas serios en el proyecto de software. Primero, aunque la integración es crítica para entregar buen software, el equipo no la práctica y a menudo la delega a personas que no están familiarizadas con el sistema completo. Segundo, la integración infrecuente suele generar código con errores. Al momento de integrar aparecen problemas que no se detectaron en ninguna de las pruebas del software sin integrar. Tercero, los procesos de integración débiles generan largos períodos de "congelamiento" del código. El congelamiento de código, o "code freeze", significa que por largos períodos los programadores no pueden agregar nuevas características, aunque sea necesario. Esto debilita la posición en el mercado y con los usuarios finales.

3.4.3.10 Propiedad colectiva de código

En un proyecto XP, **cualquier pareja de programadores puede mejorar cualquier porción de código en cualquier momento**. Esto significa que el código se beneficia de la atención de las personas, lo cual resulta en una mayor calidad de código y en menos defectos. También hay otro beneficio importante: cuando los individuos son dueños del código, se suelen agregar características en lugares equivocados cuando un programador descubre que necesita agregar algo en un lugar que "no es suyo". El dueño está muy ocupado para hacerlo, por lo que le programador escribe la característica en su propio código, en donde no corresponde. Esto lleva a código difícil de mantener, lleno de duplicación y con baja (mala) cohesión.

La propiedad colectiva de código sería un problema si las personas trabajaran a ciegas en código que no entienden. XP evita estos problemas a través de dos técnicas: las pruebas del programador (unitarias) atrapan los errores, y la programación de a pares significa que la mejor forma de trabajar con código poco familiar es estar en pareja con un experto. Además de asegurar buenas

modificaciones cuando se las necesita, esta práctica ayuda a compartir el conocimiento en todo el equipo.

3.4.3.11 Estándares de código

Los equipos XP usan un estándar de código en común, de manera que el código del sistema se vea como si fuera escrito por una única persona muy competente. No importa mucho el estándar en sí mismo: lo importante es que el código se vea familiar, para permitir la propiedad colectiva.

3.4.3.12 Metáfora del sistema

Los equipos XP desarrollan una visión común sobre cómo funciona el programa, que llamamos la "metáfora". La metáfora es una descripción evocativa simple sobre cómo funciona el programa; por ejemplo "este programa funciona como una colmena de abejas, que salen a buscar polen y lo traen de vuelta a la colmena", podría ser una descripción para un sistema de recuperación de información a través de agentes.

A veces no aparece una metáfora poética. En ese caso, con o sin una imagen viva, los equipos XP usan un sistema común de nombres para asegurarse que todos entiendan cómo funciona el sistema, en dónde buscar la funcionalidad que queremos, o cómo encontrar el lugar adecuado para agregar algo nuevo.

3.4.4 Roles en XP

De acuerdo con [BeckKent] los roles en XP son los siguientes (ver Figura 3-2):

- Programador (*Programmer*)
- Cliente (*Customer*)
- Encargado de pruebas *(Tester)*
- Encargado de seguimiento *(Tracker)*
- Entrenador (*Coach)*
- Administrador *(Big Boss)*

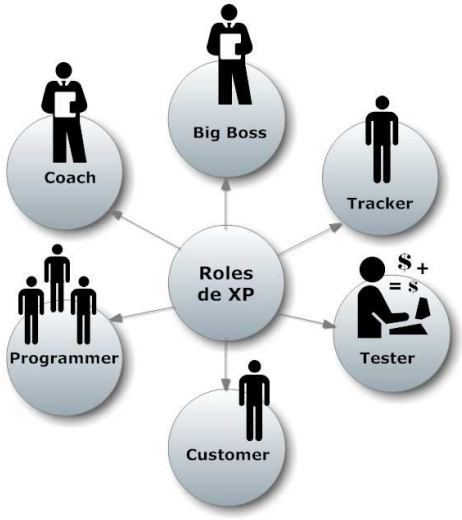

Figura 3-2 Roles en XP

3.4.4.1 Programador

El programador es el corazón de *XP.* Es el encargado de implementar el código que dará apoyo a las historias de usuario.

3.4.4.2 Cliente

Para XP, el cliente es otro elemento fundamental. El programador sabe como programar, pero el cliente sabe que programar. Para XP, el cliente es otro de los elementos fundamentales y como tal no tiene una tarea fácil. Hay habilidades que como cliente se tienen que aprender. A continuación se listan las actividades que tiene que hacer el cliente:

1. Escribir historias de usuario.
2. Tomar decisiones sobre la prioridad de las historias de usuario.
3. Escribir pruebas de funcionalidad.
4. Validar el producto.
5. Demostrar valor.

3.4.4.3 Encargado de pruebas

El rol del encardado de pruebas o *tester* se enfoca en el cliente, ayudándolo a elegir y escribir las pruebas de funcionalidad. El encardado de pruebas no es una persona que sólo se dedique a realizar pruebas del sistema. En general el encardado de pruebas tiene que cumplir con las siguientes tareas:

1. Ayudar al cliente a hacer las pruebas de funcionalidad.
2. Ejecutar las pruebas regularmente.

3.4.4.4 Encargado de seguimiento

El encargado de seguimiento o *tracker,* es la persona que deberá comprobar que el conjunto de tareas se estén realizando de manera adecuada y de acuerdo con las estimaciones hechas por el mismo. Las tareas que tiene que hacer son:

1. Hacer estimaciones y verificar que tanto se asemejan con la realidad.
2. Adoptar métricas.
3. Discutir y difundir métricas.
4. Refinar los métodos de estimación utilizados.
5. Realizar seguimiento de las iteraciones.
6. Reportar los progresos del equipo.
7. Conservar los valores históricos.

3.4.4.5 Entrenador

El entrenador o *coach,* es el responsable de todo el proceso, por lo que deberá conocerlo a fondo, de esta manera el es el encardado de explicarlo a todo el equipo.

3.4.4.6 Administrador

El administrador es el encargado de vincular a los clientes con los programadores.

3.4.5 Ciclo de vida de un proyecto XP

Dentro del ciclo de desarrollo de software con *Extreme programming* nos encontramos principalmente con lo siguiente [XPRonaldJeffries01]:

1. El cliente define el valor de negocio a implementar.
2. El programador estima el esfuerzo necesario para su implementación.
3. El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.
4. El programador construye ese valor de negocio.
5. Vuelve al paso 1.

Los pasos anteriores se repetirán hasta que se finalice el producto. En el ciclo de vida ideal de un proyecto XP se identifican seis fases [BeckKent], que son:

1. Fase de exploración
2. Fase de planeación
3. Fase de iteraciones
4. Fase de producción
5. Fase de mantenimiento

La Figura 3-3 muestra el ciclo de vida de un proyecto XP, en el cual se pueden apreciar los elementos que conforman a cada fase.

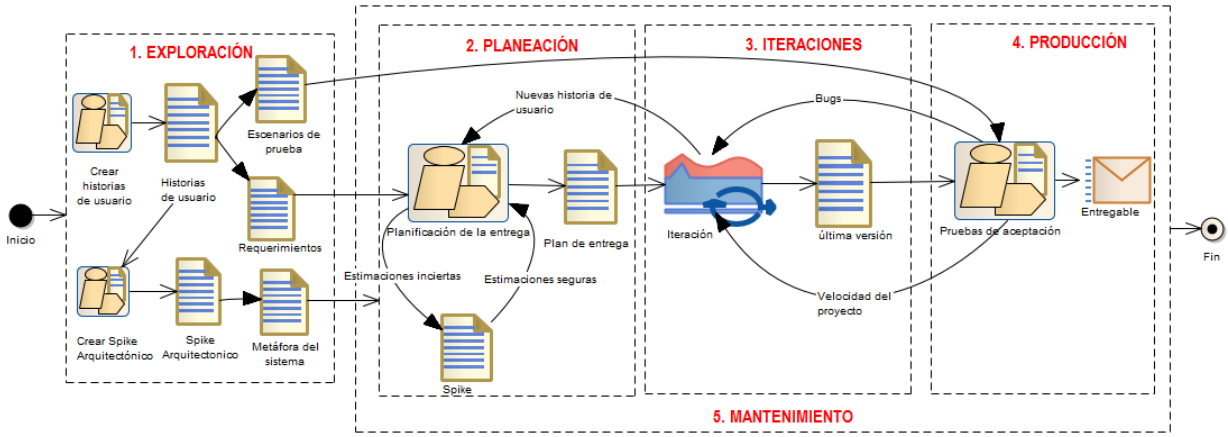

Figura 3-3 Ciclo de vida de un proyecto XP

En lo que resta se explicará cada una de estas fases.

3.4.5.1 Fase de Exploración

Esta es la primera fase, en donde los clientes plantean con alto nivel de abstracción y sin ocuparse de detalles, las historias de usuario que son de interés para la primera entrega del producto. Es en esta fase, donde el equipo de desarrollo comienza a familiariza con las herramientas que va a utilizar en el proyecto, así como también se exploran las posibilidades de la arquitectura del sistema construyendo un prototipo o varios llamados *spikes*.

Es importante hacer notar que la tarea llamada “estudio de activos de arquitectura” (ver sección 2.4 y sección 2.4.3), proveniente de los métodos tradicionales, no existe dentro de los métodos ágiles, a lo que es más, ni siquiera se comenta de la importancia qué esta tarea aporta en la entrega rápida y con valor para el cliente. Al ser muy importante el estudio de activos

arquitectónicos, en el capítulo 5 se propone que esta tarea se incluya durante la etapa de Exploración de un proyecto.

3.4.5.2 Fase de Planeación

Una vez que se han finalizado las historias de usuario, el cliente establece la prioridad de cada historia de usuario y los programadores realizan una estimación del esfuerzo necesario de para cada una. Se toman acuerdos sobre el contenido de la primera entrega y se determina un cronograma en conjunto con el cliente. Una entrega debería obtenerse en no más de tres meses. Esta fase dura unos pocos días. Las estimaciones de esfuerzo asociado a la implementación de las historias la establecen los programadores utilizando como medida el punto. Un punto, equivale a una semana ideal de programación. Las historias generalmente valen de 1 a 3 puntos. Por otra parte, el equipo de desarrollo mantiene un registro de la "velocidad" de desarrollo, establecida en puntos por iteración, basándose principalmente en la suma de puntos correspondientes a las historias de usuario que fueron terminadas en la última iteración. La planificación se puede realizar basándose en el tiempo o el alcance. La velocidad del proyecto es utilizada para establecer cuántas historias se pueden implementar antes de una fecha determinada o cuánto tiempo tomará implementar un conjunto de historias. Al planificar por tiempo, se multiplica el número de iteraciones por la velocidad del proyecto, determinándose cuántos puntos se pueden completar. Al planificar según el alcance del sistema, se divide la suma de puntos de las historias de usuario seleccionadas entre la velocidad del proyecto, obteniendo el número de iteraciones necesarias para su implementación.

3.4.5.3 Fase de Iteraciones

Esta fase incluye varias iteraciones sobre el sistema antes de ser entregado. El Plan de Entrega está compuesto por iteraciones de no más de tres semanas. En la primera iteración se puede intentar establecer una arquitectura del sistema que pueda ser utilizada durante el resto del proyecto. Esto se logra escogiendo las historias que fuercen la creación de esta arquitectura, sin embargo, esto no siempre es posible ya que es el cliente quien decide qué historias se implementarán en cada iteración (para maximizar el valor de negocio). Al final de la última iteración el sistema estará listo para entrar en la fase de producción. Los elementos que deben tomarse en cuenta durante la elaboración del Plan de la Iteración son: historias de usuario no abordadas, velocidad del proyecto, pruebas de aceptación no superadas en la iteración anterior y tareas no terminadas en la iteración anterior. Todo el trabajo de la iteración es expresado en tareas de programación, cada una de ellas es asignada a un programador como responsable, pero llevadas a cabo por parejas de programadores.

La Figura 3-4 muestra tanto los productos que son necesarios para llevar a cabo una iteración, como los productos que resultan al término de la misma.

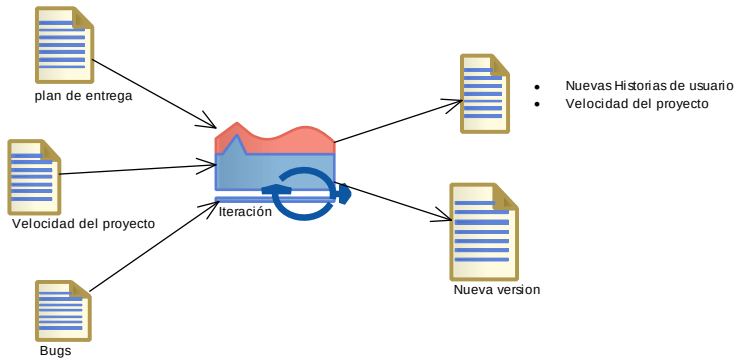

Figura 3-4 Productos de trabajo utilizados y generados durante una iteración

La Figura 3-5 trata de desglosar el contenido que hay en una iteración.

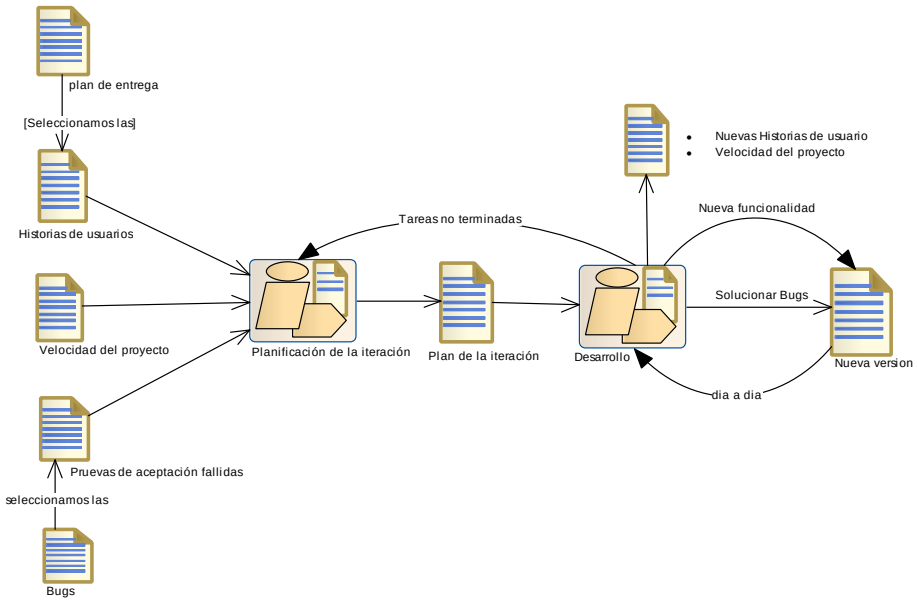

Figura 3-5 Contenido de una iteración

La Figura 3-6 muestra las tareas y actividades que ocurren dentro de la actividad de desarrollo.

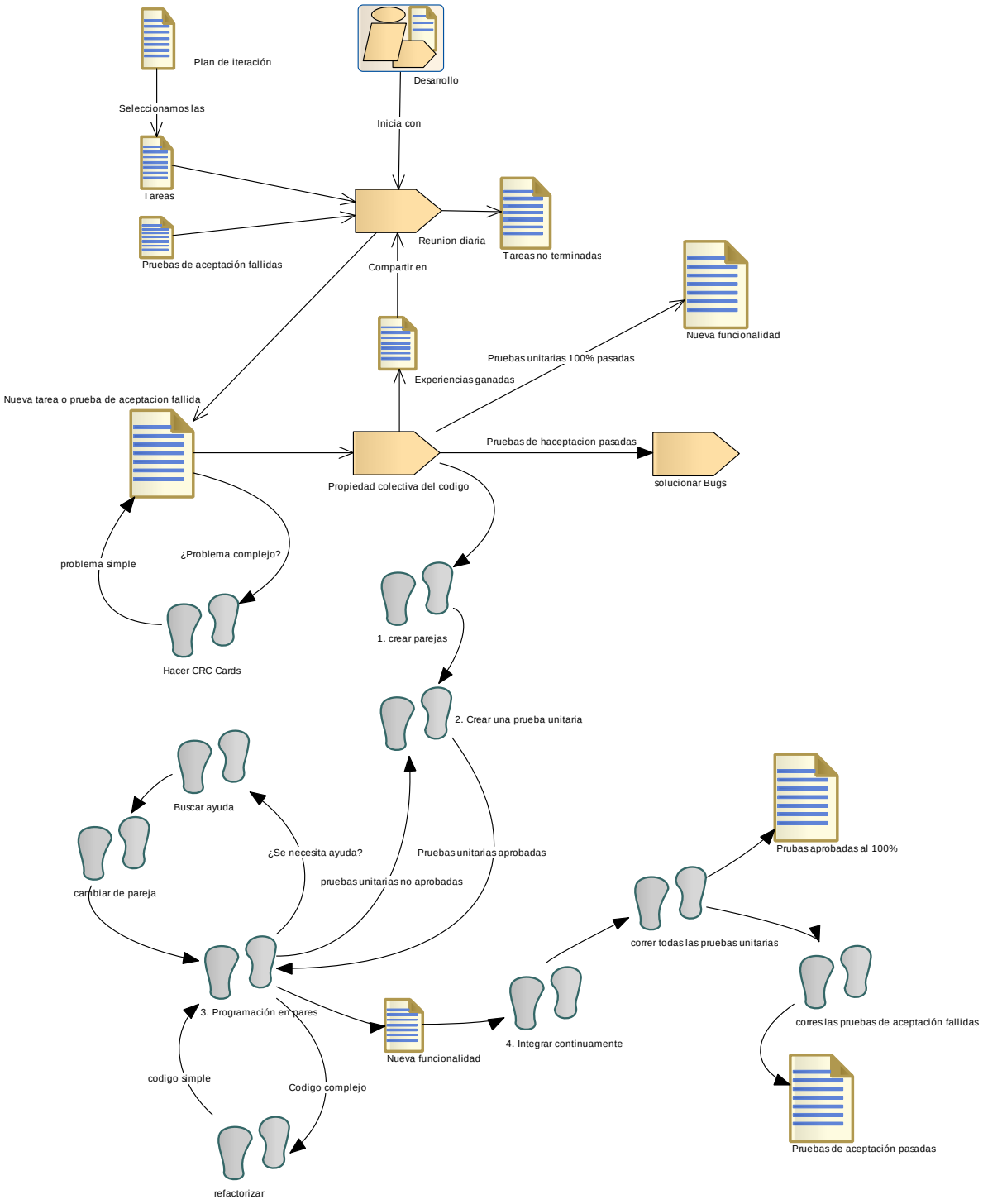

Figura 3-6 Tareas y actividades que ocurren dentro de la actividad de desarrollo

3.4.5.4 Fase de Producción

La fase de producción requiere de pruebas adicionales y revisiones de rendimiento antes de que el sistema sea trasladado al entorno del cliente. Al mismo tiempo, se deben tomar decisiones sobre la inclusión de nuevas características a la versión actual, debido a cambios durante esta fase. Es posible que se rebaje el tiempo que toma cada iteración, de tres a una semana. Las ideas que han sido propuestas y las sugerencias son documentadas para su posterior implementación (por ejemplo, durante la fase de mantenimiento).

3.4.5.5 Fase de Mantenimiento

Mientras la primera versión se encuentra en producción, el proyecto XP debe mantener el sistema en funcionamiento al mismo tiempo que desarrolla nuevas iteraciones. Para realizar esto se requiere de tareas de soporte para el cliente. De esta forma, la velocidad de desarrollo puede bajar después de la puesta del sistema en producción. La fase de mantenimiento puede requerir nuevo personal dentro del equipo y cambios en su estructura.

3.4.5.6 Fase de Muerte del proyecto

Es cuando el cliente no tiene más historias para ser incluidas en el sistema. Esto requiere que se satisfagan las necesidades del cliente en otros aspectos como rendimiento y confiabilidad del sistema. Se genera la documentación final del sistema y no se realizan más cambios en la arquitectura. La muerte del proyecto también ocurre cuando el sistema no genera los beneficios esperados por el cliente o cuando no hay presupuesto para mantenerlo.

3.4.6 Herramientas para *Extreme Programming*

En la Tabla 26 se listan algunas de las herramientas que puedes utilizarse para la gestión y seguimiento de proyectos *Extreme Programming.*

Herramienta	Descripción	Más información
XPlanner	Es una herramienta basada en web para la gestión y seguimiento de proyectos para los equipo de *Extreme Programing. Xplanner* fue implementado usando java, JSP, Struts, Hibernate y MySQL.	http://xplanner.org/
Spira Team	Es un sistema para la administración del ciclo de vida de un proyecto ágil. Está diseñado para el ciclo de vida de un proyecto *Extreme Programming.*	http://www.inflectra.com/HomePage.aspx
Scope Manger	Es una herramienta que le permite a los equipos XP automatizar y acelerar el desarrollo de software.	http://www.componentsource.com
JUnit	Son un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java.	http://www.junit.org/
DUnit	Es un marco de trabajo para hacer pruebas unitarias de aplicaciones Borland Delphi.	http://dunit.sourceforge.net/
NUnit	Es un marco de trabajo para pruebas unitarias creado por Microsoft.Net.	http://www.nunit.org/
PHPUnit	Es un marco de trabajo para hacer pruebas unitarias de aplicaciones PHP.	http://planet.phpunit.de/

Tabla 26 Herramientas para XP

3.5 Scrum

3.5.1 Introducción

En la actualidad existe una vasta gama de artículos que hablan acerca de Scrum, ejemplo de ello podemos encontrarlo en [scrumAlliance] y [AlianzaAgil]. El presente trabajo no trata de hacer un exhaustivo y profundo análisis sobre Scrum, más bien se pretende dar una idea general de Scrum, con el fin de poder identificar sus prácticas y funcionamiento. A continuación se describirán las bases que rigen a Scrum. La información aquí presentada puede ser consultada también en [scrumAlliance].

3.5.2 Breve Historia

Jeff Sutherland creador de Scrum en 1993, modifico el modelo para desarrollo de productos creado por Ikujiro Nonaka y Hirotaka Takeuchi, adaptándolo para la creación de software. En ocasiones hay metodologías que nos son consideradas como tal, más bien entran dentro de la categoría de marcos de trabajo, Scrum es una de ellas la cual no es una metodología, más bien es una forma de gestión de los equipos que implica la interacción de todos los programadores para decidir sobre los tiempos y formas de los trabajos. Scrum suele complementarse con otras metodologías agiles.

El nombre de Scrum proviene de la similitud que presentaron los autores en la forma de trabajo en equipo, ya que esta asemeja a la formación que se adopta en el rugby.

3.5.3 Definición

Como se mencionó anteriormente, a *Scrum* se le considera como un marco de trabajo ágil para desarrollar software. Todo trabajo en Scrum se organiza dentro de ciclos llamados *Sprints*, los cuales son iteraciones de trabajo que típicamente duran de dos a cuatro semanas. Durante cada Sprint, el equipo selecciona un conjunto de requerimientos del cliente de una lista priorizada, así que las características que son desarrolladas al principio son las de más alto valor para el cliente. Al final de cada Sprint se entrega un producto de software con las propiedades para ejecutarse en el ambiente requerido por el cliente

3.5.4 Valores de *Scrum*

Al igual que en otras metodologías agiles, en *Scrum* se intenta que todos los integrantes del equipo comprendan los siguientes valores:

- **Delegación**: habla respecto a que los equipos deben ser capaces de organizarse ellos mismos y promover los cambios que consideren necesario.
- **Respeto**: habla sobre el respeto que debe existir entre los integrantes profesionales de distintas disciplinas, con respecto a los diferentes puntos de vista que estos tengan.
- **Responsabilidad**: Los integrantes deben actuar con respecto a lo que se espera de el.
- **Priorizar el objetivo**: se debe priorizar la funcionalidad sobre la cual se está trabajando y asignarle en lo posible los recursos necesarios.
- **Visibilidad**: El equipo debe conocer en todo momento y cuando sea necesario la información correspondiente al proyecto, la cual deberá ser fácilmente requerida, localizada y consultada.

En lo que sigue se describen los elementos importantes en *Scrum.*

3.5.5 Proceso *Scrum*

El proceso *Scrum* se compone de actividades, roles y artefactos o productos. En esta sección se explica cada uno de los elementos de este proceso.

3.5.5.1 Roles en Scrum

Los roles en *Scrum* están divididos en dos categorías; la primera llamada roles cerdo y la otra roles gallina. La primera hace mención a los roles cuyo papel están totalmente implicados y comprometidos con el éxito del proyecto. En esta clasificación encontramos el rol de "Dueño del producto", "el equipo" y *Scrum Master.* La segunda clasificación hace mención sobre los participantes que no se encuentran implicados en el proceso *Scrum*.

Los principales roles en *Scrum* son (ver Figura 3-7) [CraigLarman]:

- Dueño del producto (*Product owner*)
- El equipo (*Scrum Team*)
- *Scrum Master*
- Gallinas (*chickens*)

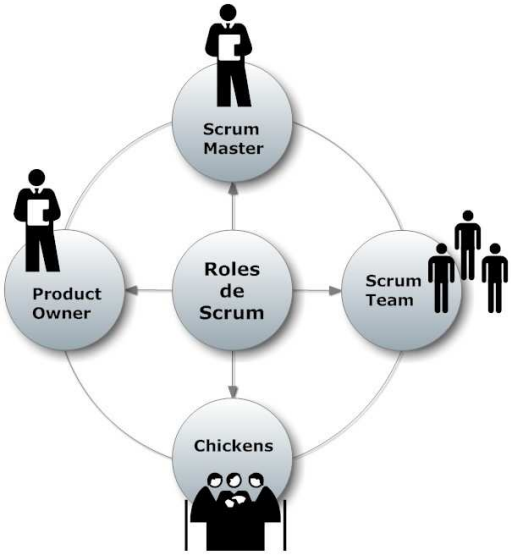

Figura 3-7 Roles en *Scrum*

A continuación se describe cada uno de estos roles.

3.5.5.1.1 Product Owner

Este rol tiene las siguientes responsabilidades [CraigLarman]:

1. Es la persona que es responsable de crear y priorizar el *Product backlog.*
2. Elegir las metas (desde el *product backlog*) para el siguiente *Sprint.*
3. Junto con los *stakeholders,* revisa el sistema al final de cada *Sprint.*

3.5.5.1.2 *Scrum* team

El elemento fundamental de *Scrum* es el equipo *Scrum* (*Scrum Team*), el cual es un grupo pequeño de personas que proveen productos o resultados útiles para los *Stakeholders* [scrumAlliance01].

Podría decirse que el rol más importante involucrado en *scrum* es el *Stakeholder*, ya que es él quien tienes los deseos, necesidades y es la razón por la cual el equipo está desarrollando software. Con frecuencia existe un *stakeholder* especial llamado dueño del producto (business owner), es quien controla el presupuesto del equipo y es quien llama al equipo a reunión [scrumAlliance01].

Mientras los stakeholders son la fuente más importante de validación para el proyecto, la persona más importante en el equipo *Scrum* es el dueño del producto. El dueño del producto es el encargado de trabajar con los *stakeholders,* representa sus intereses para el equipo y es la persona responsable del éxito del equipo. El dueño del producto debe encontrar un resultado que satisfaga

las necesidades y deseos de los *stakeholders.* El dueño del producto provee la dirección y metas para el equipo, además de priorizar que es lo que será hecho [scrumAlliance01].

Los miembros del equipo *Scrum*, incluyendo al dueño del producto, son las personas que hacen el trabajo para satisfacer las metas y prioridades que el dueño del producto tiene que establecer para ellos. El dueño del producto debe trabajar con el equipo para encontrar la dirección que el equipo pueda tomar para alcanzar los resultados deseados [scrumAlliance01].

En general el equipo tiene que trabajar en lo siguiente [CraigLarman]:

1. Trabaja en el *Sprint backlog.*
2. No tiene otro título más que el de "miembro del equipo".

3.5.5.1.3 Scrum Master

El *Scrum master* tiene que cumplir con lo siguiente [CraigLarman]:

1. Debe ser un 50% desarrollador y no sólo un administrador.
2. Conoce y refuerza el proyecto y la iteración.
3. Se encarga de verificar que se cumplan los valores y prácticas de *Scrum*.
4. Escucha al progreso y elimina los obstáculos.
5. Dirige las juntas diarias de *Scrum*.
6. Lleva a cabo la revisión del *Sprint.*

3.5.5.1.4 Roles Gallina (*Chickens*)

Los roles gallina son participantes indirectos, esto es, son participantes que se dejan de lado (en cuanto al proceso). Este rol significa que todos pueden observar pero no interferir o hablar durante una iteración.

3.5.5.2 Artefactos

3.5.5.2.1 Product *Backlog*

El trabajo de equipo en *Scrum* es dirigido por la lista de requerimientos (Product Backlog o Backlog), el cual es una lista de requerimientos funcionales o no funcionales del cliente ordenados de acuerdo con sus prioridades, esto es, el reflejo de los deseos del cliente sobre el producto. Todas las características que poseerá el producto y todos los trabajos que deban realizar los desarrolladores se manifiestan en este documento. Este documento puede compararse con su equivalente a la recolección de requisitos de una metodología tradicional, pero en comparación

con esta, el cliente participa como miembro del equipo de desarrollo y su visión de los requisitos puede ir cambiando a medida que el desarrollo del sistema crezca y con ello también una mejor comprensión del producto. Debido a que los requisitos son cambiantes, es importante mencionar que la lista de requerimientos *backlog,* siempre está creciendo. Cabe mencionar que las peticiones dadas por el cliente que están es la lista de requerimientos, pueden ser de diferente índole, como son: incluir funcionalidad de software, mercadotecnia, requerimientos no funcionales, peticiones técnicas y de infraestructura, soporte al negocio, mantenimiento de un sistema existente entre otros [scrumAlliance01].

Una vez creada la lista de requerimientos o el *product backlog,* el equipo estará trabajando activamente en los requerimientos de la lista durante el *Sprint,* esta parte es conocida como *sprint backlog,* el cual es frecuentemente pensado como una lista separada. El dueño del producto es el responsable de priorizar la lista de requerimientos, para que de esta manera se cree una distinción entre *sprint backlog y product backlog.* Desde el punto de vista del equipo, el *product backlog* es lo que se hará algún día, mientras que el *sprint backlog* es lo que se está comprometido a hacer [scrumAlliance01].

Cuando el proyecto *Scrum* comienza, el *product owner* debe iniciar el *backlog* mediante la comunicación con los *stakeholders* y otros miembros del equipo, además deberá de capturar lo que desea, sus necesidades y requerimientos. Mientras el proyecto progresa, el *product owner* y el equipo *Scrum* deberán trabajar continuamente con los *stakeholders* para priorizar nuevamente la lista de requerimientos y eliminar algunas incertidumbres que se tengan, además de refinar y generalmente limpiar algunos elementos de la lista para tenerlos listos para la planeación. Los esfuerzos del proyecto resultaran en un producto que frecuente mente clarificaran e identificaran requerimientos del *product backlog.* Este proceso es llamado *backlog grooming* y es un proceso continuo que se desarrollara a través de todo el proyecto [scrumAlliance01].

En general el *product backlog* es una lista que contiene los requerimientos del cliente, los cuales son ordenados de acuerdo con sus prioridades.

3.5.5.2.2 Sprint backlog

En el *sprint backlog* se seleccionan las funcionalidades que se esperan, de las cuales se compromete el equipo y están expresadas en el documento *product backlog,* para poder administrar las tareas necesarias para construirlas. Para crear este documento se debe considerar lo siguiente [scrumAlliance01]:

- Se deben definir todas las tareas que se van a realizar.
- El equipo puede acceder a él cuando lo requiera.
- Las tareas tiene que durar entre 4 y 16 horas

El contenido mínimo y recomendable con que debe contar son el sprint backlog son [scrumAlliance01]:

- Identificación del sprint
- Identificación de la tarea
- La persona responsable de cada actividad
- Los tiempos empleados y faltantes
- El estado de la actividad

3.5.5.2.3 Sprint backlog graph

El artefacto *Sprint backlog graph* es resumen visual de las horas de las tareas estimadas faltantes en cada sprint (*Sprint backlog*). Para *Scrum* este dato del proyecto es el más importante para el seguimiento [CraigLarman].

3.5.5.2.4 Incremento

El incremento es la parte resultante del sprint, este deberá ser en su totalidad funcional y entregable para el cliente. Se recomienda no generar sprints que no generen funcionalidad no entregable, por ejemplo crear un sprint con el objetivo de mejorar clases, hacer las pruebas de algún modulo entre otros [scrumAlliance01].

3.5.5.3 Actividades

Como en todas las metodologías en *Scrum* encontramos un conjunto de actividades que deberán llevarse a cabo de manera ordenada. Las actividades que encontramos en *Scrum* son:

- *Pre-game planning and staging*
- *Sprint planning.*
- *Sprint.*
- *Scrum meeting*
- *Sprint review*
- *Sprint retrospective.*

A continuación se describen brevemente cada una de estas actividades.

3.5.5.3.1 Pre-game planning and staging

Un proyecto de *Scrum* comienza con la visión del sistema que será desarrollado. La visión puede ser vaga al principio, posiblemente se puede establecer en términos del mercado en lugar de términos del sistema, pero tiene que irse aclarando conforme el proyecto avance [RedmondWashington].

Product Owner es el responsable de los fondos de financiamiento del proyecto para hacer que se obtenga lo que indica la visión del sistema, de tal manera que se maximice el retorno de inversión (ROI). Product Owner formula un plan para hacerlo, así que incorpora un Product Backlog [RedmondWashington].

Durante esta etapa, todos los *stakeholders* pueden contribuir para crear una lista de características, casos de uso, mejoras, defectos, entre otros que serán almacenadas en el *ProductBacklog* [CraigLarman].

Durante la creación de la visión del sistema se explorar algunos bosquejos sobre la arquitectura, sin embargo no se menciona como se pueden obtener dichos bosquejos. Por lo anterior se propone agregar la tarea llamada "estudio de activos de arquitectura" (ver sección 2.4 y sección2.4.3), proveniente de los métodos tradicionales. En el capítulo 5 se propone que esta tarea se incluya durante la etapa de Exploración y creación de la visión del sistema de un proyecto.

3.5.5.3.2 Sprint planning.

Antes de iniciar cada iteración o Sprint, se deberán llevar a cabo dos reuniones. En la primera reunión los *stakeholder* se reúnen para redefinir y volver a priorizar el *Product backlog,* además de elegir las metas para la siguiente iteración. En la segunda reunión el *Scrum Team* y el *Product Owner* se reúnen a discutir sobre el cómo alcanzar las peticiones plasmadas en el *Product backlog* y se crea el *Sprint backlog* [CraigLarman].

3.5.5.3.3 Sprint.

El trabajo es usualmente organizado en iteraciones de 30 días; cada iteración se llama Sprint [CraigLarman]. El resultado del sprint será siempre un elemento que pueda ser entregado al cliente cuando éste lo solicite. Es importante hacer notar que una vez iniciado el Sprint, no se pueden alterar los requisitos.

El objetivo de todo sprint es el incremento funcional del producto [scrumAlliance01].

Lo primero que se debe hacer en un sprint es la planeación de éste. Durante la planeación del sprint el *product owner* trabaja con el equipo para negociar que requerimientos del *product backlog* el equipo se comprometerá para hacer durante el sprint, de acuerdo con el soporte para las metas y estrategias actuales [scrumAlliance01].

3.5.5.3.4 Scrum meeting

Cada día de trabajo a la misma hora y en el mismo lugar, se deben mantener reuniones con los miembros del equipo formando un círculo, donde se responderán a preguntas generadas por los mismos miembros del equipo [CraigLarman], preguntas como:

- ¿Qué tareas realice?
- ¿Qué problemas tuve?
- ¿Cuáles son mis tareas pendientes?

3.5.5.3.5 Sprint review.

Al final de cada sprint hay una reunión con una duración máxima de cuatro horas que organiza el *Scrum master.* El *Scrum team, Product Owner,* y otros *stakeholder* asisten. Hay una demostración del producto. Las metas del *Sprint review* incluyen informar a los *Stakeholders* de las funcionalidades del sistema, diseño, fortalezas, debilidades, esfuerzos del equipo y futuros puntos problemáticos [CraigLarman].

Durante esta reunión se realiza una lluvia de ideas y retroalimentación entre los participantes, con el fin de tomar nuevas determinaciones en futuras iteraciones.

3.5.5.3.6 Sprint retrospective

Después del *sprint review* y antes del próximo *Sprint planning meeting,* el *ScrumMaster,* mantiene una reunión llamada *Sprint retrospective* con el *Scrum team,* cuya duración debe ser de tres horas. *Scrum Master* alienta al *Scrum* Team para revisar el proceso de desarrollo, dentro de la estructura del proceso de *Scrum* y sus prácticas, para hace el próximo Sprint más efectivo y agradable [RedmondWashington].

En conjunto, la reunión de planificación de Sprint, el *Scrum* Diario, la revisión de Sprint,y la retrospectiva de Sprint constituyen la inspección empírica y las prácticas de adaptación de *Scrum* [RedmondWashington].

La Figura 3-8 muestra un diagrama del ciclo de vida de un proyecto *Scrum*.

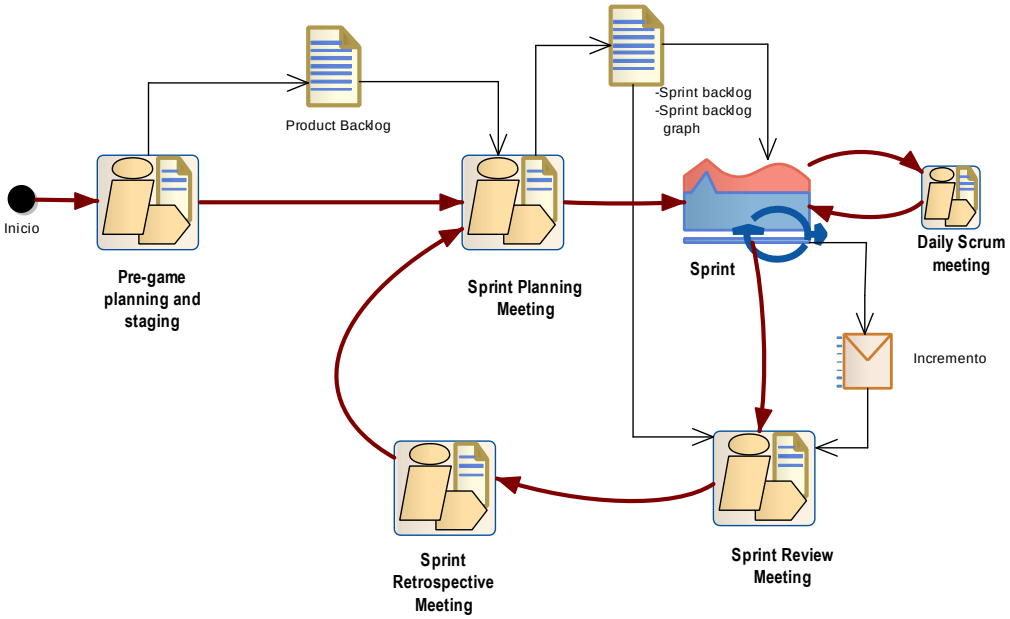

Figura 3-8 Ciclo de vida de un proyecto *Scrum*

3.5.6 Herramientas para *Scrum*

En la actualidad existen varias herramientas que permiten adoptar el proceso Scrum dentro un equipo de desarrollo ágil. Algunas son gratuitas y otras comerciales. En la Tabla 27 se resumen algunas de las herramientas que pueden utilizarse para la adopción de *Scrum*

Herramienta	Descripción	Más información
Agilo For Scrum	*Agilo for Scrum* aborda las necesidades de cada uno de los roles de *Scrum* mediante un flujo de trabajo intuitivo que facilita la documentación durante el proceso *Scrum.*	http://www.agiloforscrum.com/
Banana Scrum	Es una herramienta basada en Web para los equipos que utilizan Scrum. La intención de esta herramientas es la de reemplazar los documentos escritos que se puedan generar durante el proceso. Es perfecto para trabajar con equipos distribuidos geográficamente.	http://www.codesprinters.com/
Tackle	Es una herramienta basada en Web que provee un conjunto de características que ayudan a gestionar tanto grupos pequeños como grandes de *Scrum*. *Tackle* fue construido con ASP.Net y SQL Server.	http://tackle.codeplex.com/
XPlanner	Es una herramienta basada en web para la gestión y seguimiento de proyectos para los equipo de *Extreme Programing. XPlanner* fue implementado usando java, JSP, Struts, Hibernate y MySQL.	http://xplanner.org/
Scrumworks	Esta herramienta fue construida desde cero para ayudar a las empresas a adoptar de Scrum.	http://www.open.collab.net/
Ice Scrum	Es una plataforma libre para el desarrollo ágil de software. Ice Scrum es una aplicación Web adecuada para equipos agiles que se encuentran distribuidos geográficamente.	http://www.icescrum.org/en/
Mingle	Es una herramienta que ayuda, a los negocios, a la adopción de las mejores prácticas ágiles.	http://www.thoughtworks-studios.com/
Version one	Es un conjunto de herramientas para la gestión de proyectos ágiles. Es ideal para grupos pequeños que se encuentran distribuidos geográficamente.	http://www.versionone.com/

Tabla 27 Herramientas para Scrum

3.6 Comparación entre *Métodos ágiles*

Para concluir con este capítulo, se muestra en la Tabla 28 y Tabla 29 una comparación entre algunas metodologías ágiles [Kanwal].

Parámetro de comparación	XP	Scrum	Crystal	FDD
Tipo de proceso	Desarrollo	Administración	Desarrollo	Desarrollo
Tamaño del equipo	6-10	5-10	Clear: 3-6 Orange: 21-40	Máximo 800
Administración del equipo	Auto organizado	Poco organizado	Auto organizado	Controlado
Permite equipos distribuidos en distintas partes geográficas	No	Si	No	Si
Reunión de requerimientos	Historias de usuario	*Product backlog*	Casos de uso	Características
Maneja requerimientos altamente cambiantes	Si	Si	Si	Si
Modelado del dominio	No	No	Opcional	Si
Establecimiento de la prioridad de las tareas	Por el cliente	Por el *product owner*	Por el usuario	Por el experto en el dominio
Tiempo típico de una iteración	1-4 semanas	4 semanas	Clear: 2+/- meses Orange 3+/- meses	Máximo 2 semanas
Manejo del cambio	Cambios menores: A través de la comunicación oral. Cambios mayores: A través de la comunicación documentada.	A través del *product backlog* durante un *sprint.*	Cambios mayores: A través del cambio en el color de la familia Crystal.	Cambios menores: A través de los cambios en un conjunto de características Cambios mayores: A través del cambio en el modelo del dominio

Tabla 28 Comparación de algunas metodologías (1/2) [Kanwal]

Parámetro de comparación	XP	Scrum	Crystal	FDD
Interacción con el cliente	Durante las siguientes fases: • Fase de exploración • Fase de iteración • Fase de entrega	Durante las siguientes actividades: • *Product backlog* • *Sprint planning meeting* • *Sprint backlog*	Durante la fase de levantamiento de requerimientos	• Fase inicial • Fase plan por caracterís tica
Desarrollo	Enfoque orientado a objetos	No se especifica algún enfoque	Enfoque orientado a objetos	Enfoque orientado a objetos
Pruebas	A nivel de componentes	A nivel de sprint	A nivel de unidad	A nivel de unidad e iteración
Aseguramiento de la calidad	Inspección en pares, seguida de un equipo de inspección	Durante el *Sprint review meeting*	Durante el *Review and inspection conducted*	Inspección del diseño, después de una inspección del código.
Fortaleza	Simplicidad	Interacción entre el equipo	Tamaño del proyecto y criticidad.	Modelado del negocio

Tabla 29 Comparación de algunas metodologías (2/2) [Kanwal]

4 Arquitectura de software en métodos ágiles

Las metodologías ágiles para el desarrollo de software, han estado ganando gran popularidad como mecanismos para reducir los costos e incrementar la habilidad de manejar los cambios en condiciones dinámicas del mercado. Sin embargo, hay una preocupación significante sobre el rol e importancia de los asuntos relacionados con la arquitectura de software de un sistema, el cual está siendo desarrollado bajo un enfoque ágil [Babar2009] [Falessi]. Para esto, una posible dicotomía, un oxímoron o el intento de mezclar agua y aceite, son las percepciones que se tienen entre el enfoque ágil y los centrados en la arquitectura; pero, ¿realmente no existe una brecha entre ellos? O ¿claramente la arquitectura de software y sus prácticas no tiene cavidad en los ambientes ágiles? .En esta sección se exploran los caminos que han tomado los métodos ágiles con respecto a la arquitectura, así como también, la importancia de unir las prácticas de análisis, diseño y evaluación de la arquitectura de software en ambientes ágiles.

El objetivo de este capítulo es mostrar los resultados encontrados en la investigación sistemática realizada sobre los intentos de unificación de las prácticas arquitectónicas dentro de los métodos ágiles.

4.1 Arquitectura y agilidad

En esta sección se expone el debate que existente entre los seguidores de los métodos agiles y los métodos tradicionales que están centrados en la arquitectura. Principalmente, se tratan los siguientes puntos:

- La tensión que existe entre los proponentes de los métodos ágiles y los defensores de los métodos centrados en la arquitectura.
- La reconciliación entre la arquitectura y agilidad.
- La importancia de las prácticas arquitectónicas en enfoques ágiles.
- Los elementos que se deben considerar al intentar unificar estos dos enfoques.

4.1.1 La tensión que existe entre los proponentes de los métodos agiles y los defensores de los métodos tradicionales que están centrados en la arquitectura.

Los autores del artículo [Abrahamsson] comentan que a pesar de la gran popularidad de los métodos agiles, existe un incremento en la confusión sobre el rol de arquitecto de software y su

importancia en enfoques ágiles. Por un lado, se menciona que los defensores del rol de arquitecto de software, dudan de la escalabilidad de un sistema que sea construido mediante algún enfoque de desarrollo que no ponga la suficiente atención en la arquitectura, además, las compañías, en donde las prácticas de arquitectura de software están bien cimentadas, suelen ver a las prácticas ágiles como prácticas de aficionados, no probadas y limitadas a muy pequeños sistemas sociotécnicos basados en web. Babar [Babar2009] menciona que los métodos ágiles se mueven lejos del formalismo y rigor, sin ningún tipo de proceso sistemático para el crecimiento y los cambios de la arquitectura, por lo que el crecimiento gradual de la arquitectura de software tiene muchas incertidumbres desde el punto de vista de la corrección, lo que a menudo resulta en arquitecturas deterioradas.

Por otro lado, se menciona en [Abrahamsson] que los proponentes de enfoques agiles ven poco valor en el diseño por adelantado (*up-front design*) y en la evaluación de la arquitectura; además, ellos perciben a la arquitectura de software como algo del pasado, igualándola con el *Big Design Up-Front (BDUF),* algo que lleva a la masiva documentación e implementación del YAGNI (en ingles *you ain´t gonna need it*). También, creen que el diseño de la arquitectura tiene muy poco valor; que una metáfora debería ser suficiente en la mayoría de los casos y que la arquitectura debería de emerger gradualmente en cada sprint, como resultado de pequeñas refactorizaciones [Abrahamsson].

Por lo expuesto anteriormente, se cree que la tensión entre agilidad y arquitectura es una dicotomía; una división entre ágil y arquitectura [Abrahamsson].

4.1.2 Reconciliación entre arquitectura y agilidad

Los autores del artículo [Abrahamsson] creen que en realidad la tensión entre arquitectura y agilidad, está en eje de adaptación contra anticipación. Ellos observaron que los métodos agiles al ser adaptativos (decidir cuando el cambio ocurra), ven a la arquitectura como una entidad que se enfoca fuertemente en la anticipación (planear mucho por adelantado). Por lo que cualquier intento de unificar agilidad y arquitectura, deberá crear un balance entre estos dos enfoques.

Craig Larman asegura que la tensión entre agilidad y arquitectura, podría ser una falsa dicotomía. Además, otros representantes de métodos agiles parecen estar de acuerdo [Abrahamsson]. Philippe Kruchten, de la Universidad de British Columbia; Pekka Abrahamnsson, de la Universidad de Helsinki; y Muhammad Ali Babar de la Universidad de Copenhagen; creen fuertemente que la arquitectura se puede unir a cualquier método ágil [Abrahamsson].

A pesar de que los creadores de los métodos agiles no enfatizan en las prácticas de diseño de la arquitectura en sus metodologías, reconocen que esta es muy importante y debería de ser considerada. Ejemplo de esto, lo podemos encontrar en las palabras de Kent Beck, quien comenta que: “La arquitectura es tan importante en los proyectos XP como en cualquier otro proyecto de

software. Parte de la arquitectura es capturada mediante la metáfora del sistema; una de las practicas de XP", [BeckKent].

4.1.3 La importancia de las prácticas arquitectónicas en enfoques ágiles

A continuación se enuncian un conjunto de buenas razones por las cuales se debería de considerar la arquitectura dentro de los métodos ágiles.

Primeramente tenemos el artículo de [Faber], en donde Roland Faber menciona que a medida que la implementación de un sistema progresa, las estructuras definidas tempranamente y sus reglas, suelen dejar de ser las óptimas una vez que el entendimiento y comprensión del sistema crece, implicando cambios en la estructura del sistema, además, de que la naturaleza estática de la configuración arquitectónica llega a ser un obstáculo para su modificación. Roland Faber dice que para solucionar esto, un concepto más general de la arquitectura de software debería incluir a los aspectos estructurales de un sistema, como parte de un concepto más amplio de las cualidades de un sistema y que son los arquitectos los que proveen esas cualidades a un sistema como valor para sus clientes, comunicándolos e implementándolos en una cercana cooperación con los desarrolladores. Por lo que los arquitectos pueden y deberían jugar un rol importante en los proyectos de desarrollo ágil.

Una de las aportaciones, que el rol del arquitecto brinda a un proceso de desarrollo de software, está en proveer a sus clientes el buen manejo de los requerimientos que se encuentran detrás de la arquitectura.

Len Bass y sus colegas comentan que el valor que la arquitectura ofrece es el cumplimiento de los requerimientos de calidad no funcionales específicos de un sistema. Como consecuencia de lo anterior, se dice que el arquitecto es el *stakeholder* de la calidad de todo el sistema en cualquier tipo de proyecto.

Por su parte, en el artículo [Abrahamsson], se menciona que la arquitectura tiene valor en los métodos ágiles, en donde ésta se preocupa en liberar valor al negocio de manera temprana y constante.

Steven Frase [Steven Fraser], director del centro de investigación de Cisco, comenta que su premisa es que, cuando más grande sea el sistema, más necesaria es la arquitectura. Además comenta que para permitir economías de escala y de alcance[6], una apropiada arquitectura es absolutamente necesaria.

Ethan Hadar [Steven Fraser], quien es un distinguido ingeniero en la empresa CA In, comenta que la dedicación a la arquitectura y a las actividades de diseño, usualmente envuelven algunas

[6] Estamos frente a una economía de escala cuando volúmenes proporcionalmente mayores de producción pueden ser fabricados cada vez con costos proporcionalmente más bajos. La escala, entonces, se traduce en el tamaño de la empresa medido con relación a la cantidad de su producción.

discusiones sobre ¿Qué tan bajo vamos a llegar? si el diseño incluye componentes, integraciones; además, los participantes siempre estarán ansiosos por probar, intentar, construir y ver si esto funciona. Hadar, comenta que la arquitectura de un conjunto de componentes de despliegue y comprobables con interfaces bien definidas en un ambiente ambiguo, requieren de la planeación de la arquitectura y sus actividades relacionadas para poder ser ágiles.

Ethan Hadar [Steven Fraser], además comenta que su posición es que, la arquitectura es importante para mantener la separación entre las capas de la arquitectura (Integración externa, funcional, componentes comunes y del sistema), también como definir una arquitectura de referencia e implementación. Además, Hadar comenta que la esencia de un enfoque arquitectónico ágil, es proporcionar una ruta de conexión entre las arquitecturas que se combinan con una evaluación no funcional de la evolución de las necesidades del diseño.

Además, Hader comenta que existen algunas decisiones arquitectónicas que se oponen al cambio, como: marcos de trabajo (*frameworks*), bibliotecas, plataformas, entre otras. Mientras que otras pueden ajustarse frecuentemente; componentes de la estructura y sus responsabilidades.

Irit Hadar [Steven Fraser], miembro de la facultad en el departamento de sistemas de gestión de la información en la universidad de Haifa, comenta que los aspectos humanos influyen y son influenciados por la construcción de la arquitectura en diferentes enfoques, y que además, hay una gran diferencia entre las necesidades de la arquitectura y el impacto en pequeños equipos que están desarrollando pequeños programas en comparación con los sistemas de software a gran escala. Por lo que es importante distinguir sus diferencias e identificar los diferentes casos; como exactamente ellos difieren y si existen algunas similitudes entre ellos.

Por otro lado, Irita Hadar comenta que el beneficio de usar una metáfora en cualquier ciencia, es hacer los conceptos abstractos más concretos y familiares, de esta manera se facilita el esfuerzo cognitivo y soporte para la comunicación cuando se discute de él. Sin embargo, usar metáforas sólo permite llegar hasta ahí. En algún punto la metáfora no abarca toda la esencia del concepto que sustituye .En este punto, necesitamos en nuestro caso, un arquitecto experto que sea capaz de elevar la discusión al nivel de abstracción requerido, con el fin de alcanzar una solución de calidad."[Steven Fraser]

Dennis Mancl [Steven Fraser], distinguido miembro del personal técnico en Alcatel-lucent en Murray Hill, NJ comenta que en un mundo ágil, es tan importante invertir en la planeación de la arquitectura como antes. Además, menciona que en un mundo ágil puede haber muchos y pequeños sistemas de corta vida, en donde es razonable permitir que la arquitectura emerja durante el proceso de desarrollo. Sin embargo, para productos de software más reales, algún tipo de documentación de arquitectura ligera, y actividades de la evolución de la arquitectura, necesitan ser parte del ciclo de vida del desarrollo.

Por último, Granville Miller, un arquitecto que trabaja en proyectos de alto riesgo para Microsoft y coautor del libro "*Advanced use case modeling and a practical guide to Extreme Programming*", comenta que la arquitectura es un componente necesario para el desarrollo de un sistema de

software, y al ser tan obvia la importancia de la arquitectura de software en cualquier enfoque de desarrollo, no se debería de estar debatiendo su importancia dentro de los métodos ágiles. Además, Miller menciona que muchos de los profesionales del software confunden a la arquitectura con el BDUF (Big Design Up Front), de hecho comenta que la arquitectura no necesita estar desarrollada completamente antes de que el proyecto comience.

En lo que resta de este capítulo, se presenta una revisión sistemática sobre los intentos de integración de la arquitectura de software dentro de los métodos ágiles. En capitulo 5, se presenta un marco conceptual soportado por la revisión sistemática realizada en este capítulo.

4.1.4 Elementos que deben considerase al intentar unificar las prácticas arquitectónicas dentro de los métodos ágiles.

Para encontrar la unificación entre el enfoque ágil y los métodos centrados en la arquitectura, se tiene que clarificar varios conceptos, entre los cuales están: entender la definición de arquitectura, el contexto del proyecto, cuando considerar la arquitectura en el ciclo de vida del proyecto, el rol del arquitecto de software en ambientes ágiles, documentación necesaria, métodos arquitectónicos significantes en los métodos ágiles y, el valor y costo de la arquitectura [Abrahamsson]. A continuación se discute cada uno de estos puntos.

4.1.4.1 Entendiendo a la arquitectura de software

Para entender el valor y significado de la arquitectura, se tiene que saber lo que el proyecto u organización entiende por arquitectura. Primero, debido a que no existe una definición sobre arquitectura, en la cual todos estén de acuerdo (ver Capítulo 2), se propone que sería un buen comienzo empezar por definirla. Segundo, se debe tomar en cuenta que no todo el diseño es arquitectura; para comprende este planteamiento se presenta la Figura 4-1 extraída de [Abrahamsson].

En la Figura 4-1 se muestra el caso ideal (Figura 4-1 (a)) y caso real (Figura 4-1 (b)) sobre la clara diferencia que debe haber entre las decisiones de arquitectura y las decisiones de diseño. En la Figura 4-1 (a) y (b), todo el círculo amarillo representa todas las decisiones tomadas para un sistema de software; el círculo morado, representa todas las decisiones de diseño; el círculo rojo, las decisiones arquitectónicas; y el círculo verde, las restricciones y requerimientos.

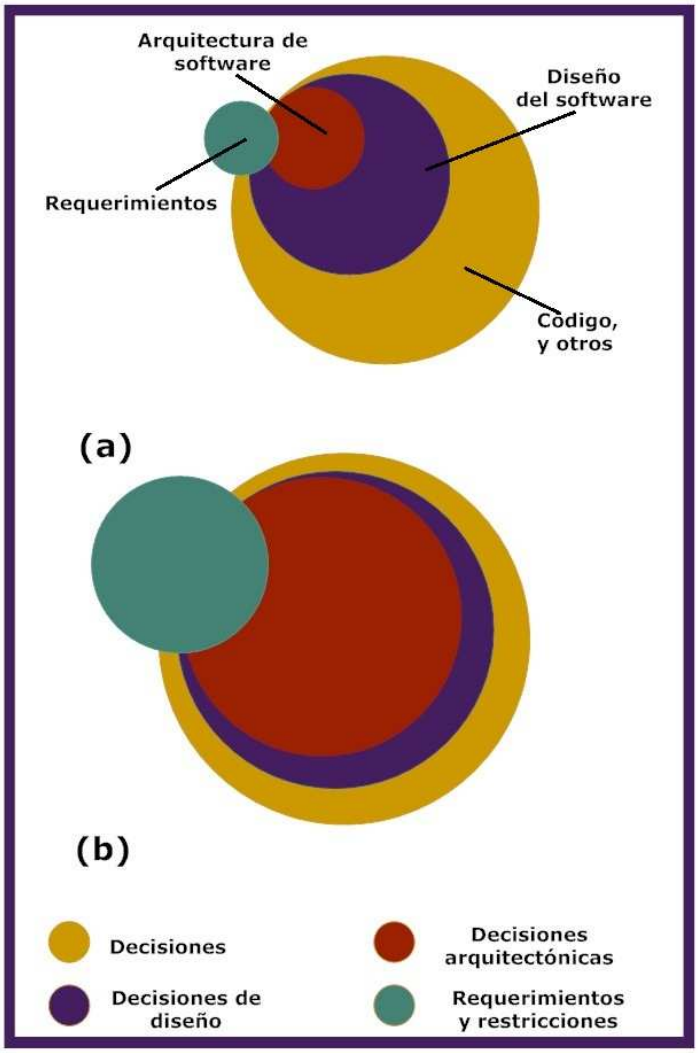

Figura 4-1 Arquitectura y diseño [Abrahamsson]

De la figura anterior, se observa que, el panorama de las decisiones de un sistema se parece más al de la figura b, en lugar de la figura a. En la figura b, no existe una clara diferencia entre las decisiones de diseño y arquitectura. Nuevamente se hace mención de que no todas las decisiones de diseño son arquitectura, sólo algunas lo son.

4.1.4.2 Clarificar el contexto del proyecto

Para conocer que actividades arquitectónicas, y en qué cantidad debe un proyecto adoptarlas, se debe conocer el contexto del proyecto [Abrahamsson]. El contexto del proyecto vendrá dado por algunos de los elementos que se muestran en la Figura 4-2, más sin embargo estos pueden cambiar de acuerdo con las necesidades del proyecto.

Figura 4-2 Contexto de un sistema [Abrahamsson]

Otros elementos que influyen en el contexto del sistema podrían ser: situación en el mercado, el poder y políticas de la compañía, duración de vida del producto, tipo de producto, cultura organizacional e historia.

Es importante mencionar que en un proyecto de desarrollo ágil, sólo deberían de necesitarse algunas pocas actividades de diseño de la arquitectura, sin embargo, en los proyectos ágiles que son extensos y complicados, se requiere de un esfuerzo significante para el diseño de la arquitectura [Abrahamsson]. Por lo anterior, es importante que los métodos ágiles consideren la necesidad de adoptar prácticas que sean inherentes al contexto del desarrollo del sistema; específicamente, se requiere la integración de prácticas arquitectónicas, cuando las variables que conforman el contexto del sistema sean complejas [Abrahamsson], [Falessi], [Sommerville 2011].

4.1.4.3 Cuando los métodos ágiles deben enfocarse en la arquitectura de software

Se tiene que especificar el momento en el que se deberán realizar los esfuerzos que aborden las prácticas relacionadas con la arquitectura de software dentro de un proyecto de desarrollo ágil. Así como también, las circunstancias bajo las cuales se considera relevantes incluir prácticas arquitectónicas. Esto último, vendrá dado por el contexto del proyecto.

En [Faber] se comenta que la comunicación entre arquitectos, proveedores de *frameworks* y los desarrolladores de aplicaciones, deben comenzar lo más pronto posible e intensificar su comunicación a lo largo del proyecto.

En [Abrahamsson], recomiendan enfocarse en la arquitectura de software lo suficientemente temprano, ya que aseguran que la arquitectura abarca el conjunto de decisiones significante acerca de la estructura y comportamiento del sistema.

4.1.4.4 Rol del arquitecto ágil

En esta sección se identifican y enunciar las responsabilidades y competencias que el arquitecto de software deberá cumplir en un ambiente de desarrollo de software ágil.

Primeramente, se encontró que las responsabilidades del rol del arquitecto de software dentro de los métodos ágiles han cambiando grandemente. Las responsabilidades del arquitecto de software, en los métodos ágiles, residen en el lado del cliente e interactúan con el cliente para obtener la priorización de las historias de usuario, además de crear el plan general de la arquitectura de software (Software Architectural Overall Plan SAOP) [Babar2009].

Los proyectos que utilizan enfoque ágiles, utilizan un rol llamado "arquitecto de soluciones", rol que puede también asumir el rol de *Scrum Master* dentro de *Scrum*. Un nuevo rol llamado "Arquitecto de aplicación" también puede ser introducido. La responsabilidad del arquitecto de aplicaciones es encontrar las historias de los usuarios y facilitar asesoramiento técnico a los desarrolladores. Por otro lado, el arquitecto de implementación es el responsable de decidir sobre el tiempo y cantidad de la refactorización que se realizará. Además, el arquitecto de aplicación se debe asegurar de que la refactorización no tenga efectos negativos en el sistema. [Babar2009]

En [Abrahamsson], comentan que el rol del arquitecto de software tiene que tener las siguientes características:

- Debe ser el creador y poseedor de las grandes decisiones, centrándose en la coordinación externa, y
- Mentor, modelador y solucionador de problemas, concentrándose más en el código y centrándose en la coordinación interna.

En [Faber] se describe que el arquitecto provee sus conocimientos como un servicio para el equipo de desarrollo, además, el arquitecto, dentro de un proceso de desarrollo ágil de software, deberá actuar como un proveedor de servicios, tanto para los desarrolladores de la aplicación como para los clientes.

Para una comunicación intensiva, los arquitectos deberían participar muy de cerca en el trabajo de desarrollo de la aplicación, incluyendo programar. Lo anterior dependerá del contexto del sistema, por ejemplo, en [Faber] comentan que el trabajar muy de cerca con los desarrolladores, suele ser más eficiente que escribir especificaciones.

Las razones para la comunicación directa expuestas en [Faber] son:

- La documentación escrita frecuentemente está fuera de fecha.
- Escribir y leer especificaciones llevan a múltiples riesgos, debido a malos entendidos.
- Los arquitectos validan mejor sus conceptos al experimentarlos por ellos mismos.
- Los arquitectos implementan directamente la calidad del sistema, por ejemplo, mediante el desarrollo de *frameworks*.
- Los arquitectos ganan experiencia muy útil en el dominio de la aplicación.

Oliver Creighton y Matthias Singer [Faber], explican que el arquitecto deber crear una cierta atmosfera de confianza y motivación para el éxito del proyecto. Además, en [Faber] se comenta que el arquitecto debe tener un conjunto de capacidades y conocimientos cuando escucha atentamente y cuando provee guía al equipo, estimulando el intercambio con sus compañeros.

También, se comenta que el arquitecto debe tomar diferentes roles para interactuar con los desarrolladores y otros *stakeholders*. Por un lado, deben comprender profundamente los requerimientos de calidad del sistema. Por otro lado, deben guiar el desarrollo al balance correcto de entre los atributos de calidad. Dependiendo la situación, el arquitecto escucha cuidadosamente o dirige firmemente las actividades del equipo. [Faber]

Como proveedores de servicios, los arquitectos se dan cuenta que la arquitectura no es final ni estática, sino un objeto de cambio como la comprensión del sistema que se incrementa durante su desarrollo. Para mantenerse al día, el arquitecto debe actualizar continuamente su conocimiento del dominio de aplicación. [Faber]

Como el stakeholder de la calidad del sistema, el arquitecto debe actuar como maestro. [Faber]

Craig larman y Bass Vodde describen a un arquitecto como un maestro programador, quien se asegura de la calidad de todo el código del sistema. [Faber]

La idea principal de que el arquitecto sea un proveedor de servicios, es que el arquitecto debería ayudar a romper sus propias reglas. Esto se debe a que el mantener las reglas no es la meta; la meta es la calidad de todo el sistema. También, el arquitecto no puede inhibir el rompimiento de una regla por completo, aun si el tratara. La presión de los requerimientos y el calendario del proyecto son muy altos. En su lugar, el arquitecto debe ayudar a romper las reglas correctamente. De no ser así, los desarrolladores con mayor estrés romperán las reglas sin avisar, probablemente ignorando el balance de la calidad, ocasionando inconsistencias o defectos en la arquitectura. Lo anterior, ilustra la necesidad de ganar la confianza de los desarrolladores. Si ellos no esperan ayuda del arquitecto, entones simplemente no avisarán cuando romperán una regla. [Faber]

Una cualidad, que el rol del arquitecto debe de tener, de acuerdo con Madison [Madison 00] es que un buen arquitecto está bien posicionado para dibujar requerimientos del negocio en la forma de un *storyboard*, explicar las restricciones técnicas para el negocio y reafirma las necesidades del negocio en términos técnicos para el equipo.

Madison comenta que hay cuatro habilidades principales que debe dominar un arquitecto, las cuales son:

- Saber descomponer el trabajo en formas *sprintable*s.
- Saber cómo apoyar al *Product Owner.*
- Saber crear el *Backlog* de la arquitectura.
- Conocimiento en el desarrollo arquitecturas empresariales (*Enterprise Architecture*).

4.1.4.5 ¿Qué documentar?

Mancl [Steven Fraser] explica cuatro razones por las cuales se debería de documentar la arquitectura:

- Una descripción temprana de la arquitectura es muy útil para la búsqueda de activos de arquitectura que se puedan reutilizar, una buena manera de reducir el esfuerzo en el desarrollo y el costo del producto final.
- Una arquitectura documentada, hace la prueba del producto más completa.
- Para un producto de software de larga vida, una descripción arquitectónica ayuda a los miembros nuevos de un equipo de desarrollo aprender acerca de la estructura del software, reduciendo errores.
- Puede haber módulos en el producto de software que puedan ser reutilizados en otro contexto y una descripción arquitectónica es necesaria para describir donde esos módulos trabajarán.

Por lo anterior, Mancl afirma que un proceso de desarrollo ágil debe incluir tareas y artefactos para soportar una cierta cantidad de planeación y evolución de la arquitectura [Steven Fraser]

¿Qué tanto documentar? O ¿qué documentar? En el articulo [Abrahamsson] los autores comentan que, la mayoría de las veces, será suficiente comenzar con un *walking skeleton* o esqueleto inicial a lo largo del proyecto, acompañado de un pequeño número de solidas metáforas para transmitir el mensaje al equipo. Pero, en algunas circunstancias, dicen que será necesario documentar la arquitectura de software más explícitamente, por ejemplo, para comunicar la arquitectura a una gran audiencia o para cumplir con regulaciones externas. Además, comentan que las especificaciones de arquitectura son útiles para algunos recién llegados o para mostrarles un esquema general a algunos *stakeholders* clave, para guiarlos a través del sistema o para proveer información que pueda ser incorporada inmediatamente en el código. Por lo tanto, en [Faber] afirman que la documentación debería de ser, tanta como se necesite y no más, además de que la comunicación directa debe acompañarla.

4.1.4.6 Métodos de análisis arquitectónicos.

Los autores del artículo [Abrahamsson], mencionan que aún cuando los métodos agiles no se oponen al concepto de arquitectura, todos ellos prefieren guardar silencio sobre como identificar elementos que competen a la arquitectura, por ejemplo, como identificar requerimientos significantes para la arquitectura, como realizar diseño arquitectónico incremental, como validar aspectos arquitectónicos, entre otros. Además, comentan que los métodos arquitectónicos existen, pero estos no son bien conocidos.

4.2 Arquitectura de software y métodos ágiles en la práctica.

En esta sección se presenta el resultado de la revisión sistemática de artículos y reportes que hablan sobre la integración de la arquitectura de software dentro de métodos ágiles. De la misma manera, estos artículos proporcionan consejos, técnicas, guías y métodos para integrar las prácticas arquitectónicas dentro de las actividades que realizan los equipos de desarrollo agiles.

El análisis de la bibliografía aquí presentada, permitirá definir un marco teórico de arquitectura ágil, el cual se presenta en el capítulo 5.

Principalmente, en esta sección se tratan los siguientes puntos:

1. Los uso de las prácticas arquitectónicas en los métodos ágiles.
2. Análisis, diseño y evaluación de la arquitectura de software dentro de los métodos ágiles.
3. Propuestas de integración de las prácticas arquitectónicas dentro de los métodos ágiles.

4.2.1 Usos del las prácticas arquitectónicas dentro de los métodos ágiles.

En esta sección se presentan las investigaciones y estudios realizados en la industria del software; cuyo fin, está en identificar las prácticas arquitectónicas más utilizadas por equipos de desarrollo ágil.

Primeramente, se presenta la encuesta llevada a cabo a 72 desarrolladores en IBM [Falessi], la cual sugiere una compatibilidad teórica entre los valores ágiles y las prácticas arquitectónicas. De la misma manera, la encuesta da buenos pronósticos para futuras integraciones en la práctica.

El objetivo de esta encuesta fue conocer si los desarrolladores agiles consideran relevante el uso de la arquitectura de software en sus trabajos, para lo cual la encuesta está dividida en tres partes; la primera, trata de identificar la relevancia de los usos de la arquitectura en contextos agiles; la segunda, trata de identificar el momento en el que es importante enfocarse en la arquitectura de software; y la tercera, tiene por objetivo encontrar la relación, o soporte, que los principios de los

métodos centrados en la arquitectura pueden brindar al soporte del cumplimiento de los cuatro valores del manifiesto ágil.

A fin de ilustrar la relevancia de esta encuesta, se detallará cada una de las partes en las que ésta se enfoca:

- **Relevancia de los uso de la arquitectura de software:** En esta parte de la encuesta, los autores de [Falessi] tratan de conocer si los desarrolladores agiles encuentran importante los usos de la arquitectura de software en su trabajo. Para lograr esto, toman como referencia la definición de los usos de la arquitectura descritos en ISO/IEC WD4 42010. La primer pregunta que se realizo al equipo de trabajo fue: "'*En el contexto de desarrollo ágil, ¿Qué tan relevante es cada uno de los siguientes usos de la arquitectura de software* (ver Tabla 30)?".

Posición	ISO/IEC 42010 usos de la arquitectura de software	Nivel de relevancia
1	Para la comunicación entre las organizaciones involucradas en el desarrollo, producción, operación y mantenimiento del sistema.	2.16
2	Como entrada para el diseño subsecuente del sistema y las actividades de desarrollo.	2.04
3	Para documentar supuestos hechos por el arquitecto acerca del sistema y la intensión de su uso y ambiente.	2.02
4	Para analizar y evaluar diferentes arquitecturas	1.98
5	Para comunicar las características y diseños de un sistema para los clientes potenciales.	1.98
6	Para dar soporte a la revisión, análisis y evaluación del sistema.	1.95
7	Para ayudar a la planificación de la transición de una arquitectura de software heredada a una nueva arquitectura de software.	1.80
8	Como la especificación para un grupo de sistemas que comparten un conjunto de características (ejemplo, líneas de productos).	1.75
9	Para apoyar la ampliación de las prácticas ágiles a los grandes proyectos.	1.74
10	Para documentar puntos de flexibilidad o limitaciones dentro del sistema para futuros requerimientos.	1.69
11	Como documento de desarrollo y mantenimiento	1.67
12	Para soporte operacional y de infraestructura; administración de la configuración y reparación; re diseño y mantenimiento de un sistema, subsistemas y sus componentes	1.66
13	Para establecer criterios para las implementaciones de certificación para la conformidad de la arquitectura de software.	1.62
14	Para la comunicación entre los clientes, los compradores y desarrolladores como parte de las negociaciones del contrato.	1.50
15	Para apoyar las actividades de planificación y presupuestación del sistema.	1.35
16	Para apoyar la preparación de los documentos de la adquisición.	1.28
17	Como entrada para la selección de la generación de sistemas y herramientas de análisis.	1.12
Promedio		1.72

Tabla 30 Nivel la relevancia de los usos de la arquitectura de software como la perciben los desarrolladores ágiles. [Falessi]

Para contestar la pregunta anterior, los participantes clasificaron el nivel de relevancia de cada uso de la arquitectura entre 0 (no relevante) y 3 (extremadamente relevante).

Los resultados fueron que sólo 3 de los 17 usos de la arquitectura de software son más irrelevantes que relevantes para la práctica ágil. Con lo que se concluye que los participantes consideran a la arquitectura de software relevante en el contexto del desarrollo ágil [Falessi].

- **Cuando enfocarse en la arquitectura de software:** Los autores de [Falessi] citan a (Grady booch, 2000) y dicen que "*No se necesita una arquitectura para construir una perrera, pero sería bueno considerar una si se va a construir un rascacielos*." Tomando en cuenta las palabras de Booch, la siguiente pregunta es "*en el contexto del desarrollo ágil, ¿Cuándo nos deberíamos de enfocar en la arquitectura de software?*; *siempre, nunca o cuando el proyecto sea complejo*".

  Además, para quien respondió "cuando el proyecto sea complejo" y debido a que el término complejo es muy amplio, se les pido que eligieran uno de los elementos que causa complejidad: distribución geográfica, número de requerimientos o líneas de código, numero de *stakeholders* y otros. La Figura 4-3muestra las respuestas de esta encuesta.

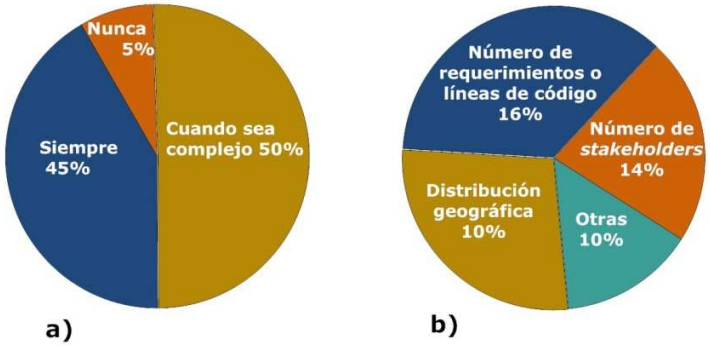

Figura 4-3 Resultados de la pregunta de cuándo el desarrollo ágil debe enfocarse en la arquitectura [Falessi]

La Figura 4-3 (a) muestra que la mitad de las respuestas indican que debemos enfocar en la arquitectura sólo cuando el proyecto sea complejo. La Figura 4-3 (b) reporta el porcentaje de los resultados que caracterizan a la complejidad. En ella se observa que el principal motivo de complejidad es cuando el número de requerimientos o líneas de código es complejo.

Por último, es importante notar que, esta encuesta respalda la importancia de conocer el contexto del proyecto [Abrahamsson] y de adoptar las prácticas arquitectónicas inherentes a él.

- **Valores ágiles y los principios centrados en la arquitectura**

  Para la tercera parte de la encuesta realizada en [Falessi], se trata de encontrar si existe una relación entre los valores ágiles y los principios centrados en la arquitectura. Para lo

cual, los participantes caracterizaron las relaciones que hay en todas las combinaciones posibles entre los cuatro valores del manifiesto ágil y los siguientes principios de los métodos centrados en la arquitectura:

- Dirigidos por los requerimientos no funcionales.
- Requerimiento de una inversión por adelantado en el diseño (*upfront investment*).
- Obligar el cumplimiento de la arquitectura

La Figura 4-4 muestra los resultados de la relación entre los valores ágiles en comparación con los principios centrados en la arquitectura; la primera fila, muestra la relación más favorable entre los cuatro valores ágiles y los centrados en la arquitectura; la segunda fila, muestra la relación más contrastante; y la última fila, muestra el promedio.

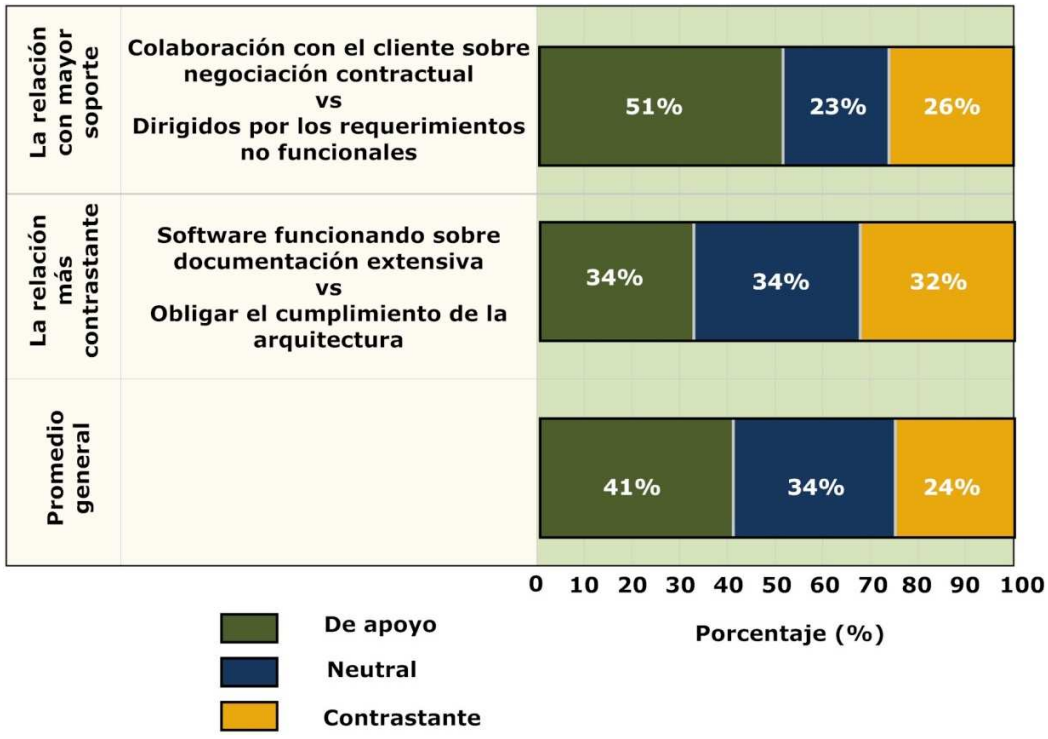

Figura 4-4 Resultados de la relación entre los valores ágiles en comparación con los principios centrados en la arquitectura [Falessi]

- **Resultado de la encuesta:** Como resultado de la encuesta se tiene que la mayoría de los desarrolladores agiles ven la necesidad de nuevos métodos y entrenamiento especial para integrar prácticas de la arquitectura (tales como el análisis de la arquitectura de software y revisión del diseño) dentro de enfoques ágiles.

Además, se afirma que el principal problema en la combinación de los métodos ágiles y los centrados en la arquitectura no reside en cuestiones teóricas, sino en los asuntos prácticos de la adopción [Falessi].

Por otro lado, el resultado de la encuesta muestra que los desarrolladores ágiles de software, perciben a la arquitectura de software tan importante y de apoyo para los valores agiles, en lugar de neutral o contrastivo [Falessi].

Otro caso de estudio lo encontramos en [Babar2009], cuyos autores realizaron un estudio enfocado en la identificación de las prácticas y retos arquitectónicos dentro de equipo de desarrollo ágil. La Tabla 31 muestra algunos de los hallazgos más importantes de este estudio.

Practicas más utilizadas	Prácticas que se emplearon sólo cuando fueron consideradas importantes	Prácticas que nunca se emplearon	Prácticas relacionadas con la arquitectura
Sprint, Sprint Planning, Spring Review, Daily Meetings, Refactoring, Simple Design, Coding Standards and Collective Code Ownership.	*Testing, Post Game Sessions y 40 Hours Week.*	*Pair Programming y Onsite Customers.*	*Spring Planning, Sprint Review, Re-factoring, Metaphor* y *Simple Design.*

Tabla 31 Resultado de prácticas ágiles utilizas en el estudio

El estudio realizado en [Babar2009], se enfoca principalmente en encontrar el uso de las prácticas relacionadas con la arquitectura y su utilización dentro de proyectos ágiles.

Dentro de las prácticas relacionadas con la arquitectura encontramos [Hofmeister]:

- Análisis arquitectónico
- Síntesis arquitectónica
- Evaluación arquitectónica

A continuación se exponen los resultados encontrados para cada una de las prácticas anteriormente mencionadas.

Análisis arquitectónico

El análisis que se encontró en [Babar2009], demuestra que las tareas relacionadas con la fase de análisis de la arquitectura (examinar contexto, definir problema) se habían colocado del lado del cliente, esto es, en un enfoque ágil el cliente es el responsable de dibujar el *roadmap*

arquitectónico a un alto nivel de abstracción, basándose solamente en sus requerimientos. Además, los clientes son los responsables de proveer al equipo del proyecto con las historias de usuario, para las cuales, las decisiones de diseño detalladas tenían que ser hechas e implementadas por el equipo de desarrollo. La mayoría de las decisiones hechas por el arquitecto de soluciones e implementador del equipo ágil, fueron basadas en las características que se requerían para entregar dentro un costo y tiempo fijo. Lo que significa, que no hubo atención para otras posibles soluciones. Otros resultados encontrados en [Babar2009], prueban que no existe ninguna atención en los atributos de calidad, lo anterior debido a que el cumplimiento de los atributos de calidad, no es considerado una medida de éxito para el proyecto.

Síntesis arquitectónica

La investigación en [Babar2009], reveló que los métodos ágiles aplican dos etapas para el diseño de la solución:

1. Los arquitectos de software trabajan con el cliente dibujando una arquitectura de alto nivel (*roadmap*).

2. Después, el arquitecto solucionador e implementador, realizan las decisiones de diseño, considerando las historias de usuario y sus prioridades, el presupuesto, tiempo, plataformas existentes y el SAOP (*Software Architectural Overall Plan*).

El equipo ágil, de la investigación en [Babar2009], considero un número limitado de soluciones potenciales para el diseño de una arquitectura, las cuales habían sido usadas en proyectos previos. Por otra parte, los equipos ágiles produjeron significativamente menor número de entregables desde esta fase, y la mayoría de ellos, estuvieron disponibles en una *Wiki*.

El principal entregable es el SAOP (*Software Architectural Overall Plan*)

Evolución arquitectónica

En equipos ágiles, la arquitectura es evaluada a un alto nivel de abstracción, sin ir a detalles. La mayoría de las veces, los desarrolladores de otros proyectos eran invitados para revisar la arquitectura y encontrar fallas que pudieran ser serias. La realización de la evaluación, hecha por los desarrolladores, fue generalmente una validación de diseño en lugar de una validación de arquitectura. Como contraste, en [PaulClements] se dice que la evaluación de la arquitectura, es usualmente realizada para los atributos de calidad. Sin embargo, los proponentes de los enfoques ágiles, dicen que la refactorización puede ayudar a alcanzar los atributos de calidad.

La Tabla 32 resumen las actividades y artefactos relacionados con la arquitectura que son utilizados por los equipo ágiles para aplicar e implementar soluciones de diseño, estas actividades y artefactos son extraídos de [ChristineHofmeister00].

Actividades [ChristineHofmeister00]	**Artefactos** [ChristineHofmeister00]	**Artefactos utilizados por equipos ágiles**
Análisis arquitectónico	Contexto.	Plataformas, establecimiento del costo y duración del proyecto. Historias de usuario enfocadas en las características que serán entregadas. Ninguna atención particular sobre los atributos de calidad.
	Requerimientos.	
	Requerimientos significantes para la arquitectura.	
Síntesis arquitectónica	Soluciones candidatas para la arquitectura	Limitar el número de soluciones por el equipo de arquitectura. Plan de la infraestructura de la arquitectura, historias de usuario.
	Diseño arquitectónico (vistas, perspectivas, prototipos)	
	Base lógica de la arquitectura	Base lógica de la arquitectura
Evaluación arquitectónica	Atributos de calidad	Atención en las características descritas por las historias de usuario. Ninguna atención en los atributos de calidad. Cooperación interna del equipo para el evaluar el diseño.
	Valoración arquitectónica	
Directrices generales del proceso	*Backlog*	*Product backlog* y el *Sprint backlog*

Tabla 32 Resumen de los artefactos relacionados con la arquitectura utilizados por los equipo ágiles [Babar2009]

Retos encontrados para los equipos ágiles de desarrollo de software

Dentro de los retos que se encontraron en [Babar2009], se mencionan los siguientes:

- **Incorrecta priorización de las historias de usuario:** El estudio realizado en [Babar2009], encontró que uno de los retos clave relacionados con la arquitectura, en los equipos ágiles, son las historias de usuario, las cuales usualmente son priorizadas sin tomar en cuenta consideraciones técnicas. Se puso de manifiesto que si una interdependencia fundamental entre historias de usuarios era encontrada, podría haber requerido de una importante refactorización con consecuencias para todo la estructura del software.

  En otras palabras, cuando las historias de usuario llegan a los desarrolladores ágiles, los clientes ya han decidido sobre la asignación de prioridades, por lo que se tienen

que encontrar la manera de implementar las historias de usuario de baja prioridad antes de implementar los de alta prioridad. Estos requisitos tienen por lo general efectos negativos sobre las decisiones de diseño. Lo anterior, debido a que la priorización de las historias de usuario, no se realiza teniendo en cuenta la interdependencia entre las decisiones de diseño, la cual se debe considerar para implementar todas las historias de usuario.

Una propuesta para solucionar esto es [Babar2009]: Los participantes proponen que el arquitecto y los desarrolladores se involucren en la priorización de las historias de usuario. Una estrategia es utilizar un "***Feature Analysis Workshops***" antes de comenzar con cada proyecto. En este *workshop,* todos los miembros principales del proyecto participan para entender y priorizar las historias de usuario.

- **Falta de tiempo y motivación para considerar las decisiones de diseño:** Se encontró que los equipos ágiles se vieron obligados a centrarse en un número limitado de soluciones para lograr las características requeridas en el tiempo y presupuesto establecidos. El riesgo de este enfoque es que los arquitectos pueden perder la posibilidad de elegir mejores elecciones de diseño al no hacer un diseño por adelantado completo (*upfront design*).
  Propuesta de solución: En [Babar2009], se propone una iteración para hacer el trabajo centrado en la arquitectura (o arquitectónico). Esta iteración es también conocida como la iteración cero entre los seguidores del enfoque ágil. Tambien, se propone que la iteración se puede combinar con el "Taller de análisis de características (FAW)" o Feature Analysis Workshop.

  Otra solución es, que haya tiempo asignado en cada iteración, para que los desarrolladores puedan pensar acerca de las diferentes opciones de diseño, con el fin de determinar la elección del diseño más adecuado.

- **Dominio desconocido y soluciones no probadas:** Los participantes fueron de la opinión de que los enfoques ágiles podían no ser adecuados cuando se trabaja en un dominio desconocido, con un nuevo cliente o con soluciones no probadas. Se comenta que trabajar con una nueva solución para una nueva unidad de negocio, es muy riesgoso si se usa un enfoque ágil. De aquí que, si no se tiene el conocimiento del dominio, las soluciones arquitectónicas bien probadas y los patrones para un dominio particular del cliente, sería muy difícil comenzar a entregar las características deseadas desde la primera iteración.

  Solución sugerida: en [Babar2009], sugieren que en estas circunstancias, se aplique un enfoque hibrido en lugar de uno puramente ágil. Lo anterior, debido a que los métodos ágiles deben ser utilizados en donde los clientes y las soluciones son bien conocidos.

- **Falta de enfoque en los atributos de calidad:** el estudio en [Babar2009], muestra que la falta de atención en los atributos de calidad, para hacer las decisiones de diseño, usualmente resultan en una estructura arquitectónica que difícilmente puede conocer los requerimientos de calidad en etapas más adelantes. Tales sistemas, necesitan un enorme presupuesto y tiempo para arreglar los atributos de calidad durante el mantenimiento del proyecto.

  Solución propuesta: La estrategia para tratar con esta situación, es hacer que la satisfacción de los atributos de calidad sea una medida de éxito, y ligarlos al presupuesto para el desarrollo y mantenimiento de los proyectos. Esto significa que, el trabajo realizado para los atributos de calidad, durante el mantenimiento del proyecto, deberían ser realizado durante el desarrollo del proyecto.

- Otros retos son [Babar2009],:
  1) La falta de de habilidad por parte del equipo ágil; por lo que los enfoques agiles son más adecuados para aquellos que cuenten con un equipo de desarrollo que sea realmente sea muy bueno desarrollando, que conozcan muy bien el sistema y que sean capaces de crear un diseño sofisticado, mientras implementan historias de usuario sin tener una actividad de diseño inicial (*upfront design)*.

  2) Documentación no planificada de las decisiones de diseño en una *Wiki*, suelen dar lugar a varias dificultades en encontrar la información necesaria sobre las decisiones de diseño clave. Esta búsqueda puede consumir mucho tiempo.

Por último, la Tabla 33 resume las ventajas y desventajas de usar métodos ágiles relacionados con los aspectos arquitectónicos de un sistema.

Ventajas	Desventajas
• Les brinda a los desarrolladores una comprensión temprana del proyecto para la toma de decisiones de diseño. • No se necesita invertir mucho tiempo en discutir y documentar las soluciones que puedan no ser implementadas. • Entregas claras y acordadas para la fecha establecida y el presupuesto desglosado en pequeñas iteraciones. • Se ahorra entre el 30 y 40 por ciento de las actividades arquitectónicas y las encargadas de documentar el diseño. • Las decisiones de diseño son compartidas con mayor facilidad y rapidez • Se gana conocimiento a través de las *Wikis* y reuniones de diseño.	• Se implementan las historias de usuario sin un buen conocimiento sobre las posteriores inter dependencias de las decisiones de diseño. • Desde el punto de vista de la arquitectura, es muy arriesgado para los nuevos proyectos cuando las posibles soluciones no se comprenden muy bien. • No se tiene mucho tiempo para poner atención en el diseño durante las iteraciones. • No hay consideración de arquitecturas alternativas y mejores soluciones de diseño no son consideradas. • No hay atención en los atributos de calidad del sistema, excepto en algunos casos de desempeño. • Buscar las decisiones de diseño en una *Wiki* puede ser complicado.

Tabla 33 Ventajas y desventajas de utilizar un enfoque ágil para construir la arquitectura de un sistema.

4.2.2 Análisis, diseño y evaluación de la arquitectura de software dentro de los métodos ágiles.

En esta sección se presenta los intentos de incluir las prácticas de análisis, diseño y evaluación de la arquitectura de software dentro de los métodos ágiles, en particular, los métodos del SEI (*Software Engineering Institute*) [SEI2010].

4.2.2.1 Arquitectura de software y XP

El diseño que emerge, dentro de un proceso ágil, es un producto proveniente de las relevantes historias de usuario que han sido identificadas. Pero la arquitectura depende, para su forma y calidad, en la experiencia del equipo de desarrollo [Nord].

Las actividades centradas en la arquitectura, pueden informar y regularizar el proceso de desarrollo, enfatizando en los atributos de calidad y enfocándose de manera temprana en las decisiones arquitectónica [Nord].

En situaciones, en donde los requerimientos cambian rápidamente y se ha garantizado un enfoque ágil, los conceptos arquitectónicos pueden mejorar el proceso de diseño de un sistema que conocerá sus requerimientos de manera temprana [Nord].

A continuación se presenta algunos de los métodos del SEI, que han intentado unificarse a los métodos ágiles, entre ellos encontramos los siguientes:

- **El método QAW (Quality attributes workshop) para la captura de los requerimientos**

  El QAW puede ayudar al equipo de desarrollo a entender el problema, mediante la obtención de los requerimientos de los atributos de calidad en la forma de escenarios. Basando los escenarios en las metas del negocio, se asegura de que los desarrolladores aborden correctamente el problema [Nord].

- **El método ADD para el diseño de la arquitectura**

  El método ADD (*Attribute-Driven Design*) define a la arquitectura de software mediante el hecho de basar el proceso de diseño en escenarios de atributos de calidad priorizados que el software debe cumplir. Lo anterior ayuda a identificar lo más importante que se debe de hacer para asegurar de que el diseño está en camino de conocer los principales atributos de calidad y entregar valor para el cliente [Nord].

- **ATAM Y CBAM**

  Los métodos ATAM (Architecture Trade-off Analysis Method) y CBAM (Cost-Benefit Analysis Method) proveen una guía detallada para el análisis del diseño, para que de esta manera, se obtenga una retroalimentación temprana sobre el riesgo. El equipo de desarrollo puede usar prácticas de diseño incremental para desarrollar un diseño e implementación detallados de la arquitectura [Nord].

Usar los métodos anteriores resulta en un enfoque centrado en la arquitectura: La arquitectura conecta las metas del negocio para la implementación, los atributos de calidad del diseño y las actividades centradas en la arquitectura para manejar el ciclo de vida del sistema de software. Con la ayuda de estos métodos, se hace el desarrollo de software más fácil y más consistente [Nord].

A continuación se detalla la integración de los métodos, expuestos anteriormente, dentro de XP.

Identificar requerimientos: QAW

El QAW, se realiza a principios del desarrollo del proceso, durante la producción de historias de usuario, ayudando a mostrar los requerimientos de atributos de calidad en la forma de escenarios. Esto es, el método QAW, sería apropiado para la primera iteración de un proyecto XP, ayudando a identificar los requerimientos de calidad claves para el sistema. En las iteraciones subsecuentes, los desarrolladores pueden colaborar con el cliente, ejerciendo la práctica "*on-site customer*" para obtener y afinar escenarios adicionales como se necesiten. [Nord]

A continuación se describe el valor que el método QAW agrega a XP:

- El QAW compromete a los *stakeholders* del sistema de manera temprana en el ciclo, para encontrar los requerimientos de calidad que manejarán la construcción del sistema. [Nord]
- El QAW está basado en el sistema, enfocándose en los *stakeholders* y se hace antes de la creación de la arquitectura del software. [Nord]
- Los stakeholders tienden a enfocarse en la funcionalidad y no en los atributos de calidad, por lo que el método QAW ayuda a crear el equilibrio entre estos. Los escenarios de los atributos de calidad son muy similares a las historias de usuario, con la diferencia de que los primeros permiten considerar a los atributos de calidad además de la funcionalidad. [Nord]
- También, este método mejora la planeación y el proceso de la generación de historias de usuarios, enfatizando en las metas del negocio, *stakeholders* y el papel de los atributos de calidad en el diseño de la arquitectura [Nord].
- Las metas del negocio son obtenidas y refinadas durante el QAW. [Nord]
- Los escenarios pueden ayudar a determinar lo que se encuentra fuera o dentro del alcance del sistema y puede dirigir a la creación o refinamiento del diagrama del contexto del sistema o su equivalente. Además, la generación de los escenarios ayuda a crear los casos de uso. [Nord]
- El QAW ayuda a los stakeholders a descubrir y priorizar los atributos de calidad. [Nord]
- Los escenarios pueden ayudar al cliente para preparar las pruebas de aceptación que crecerán con el producto. [Nord]
- Muchos de los clientes no saben cómo construir los casos de pruebas. El escenario de atributos de calidad puede dar información sobre que probar en caso de no haber recibido ayuda para construir estos casos de prueba. [Nord]

Diseño temprano: El método ADD

Los desarrolladores en XP aumentan el sistema incrementalmente. Cuando un sistema no soporta la nueva funcionalidad, entonces se hace una refactorización en el diseño. La primera iteración juega un rol crucial en la definición de toda la estructura de un sistema, ya sea implícitamente (la arquitectura emerge después de implementar la primera ronda de historias de usuario) o explícitamente [Nord].

En [Nord] afirman que el método ADD se enfoca en lo que algunas veces los desarrolladores de XP ignoran; la estructura general del sistema que los atributos de calidad forma. Además comentan que esta atención debería ocurrir en las primeras iteraciones y recurrir a ella en iteraciones más adelante, como un cambio sustancial en la arquitectura del software.

También, en [Nord] se comenta que el equipo de trabajo puede mantener la arquitectura, atributos de calidad y las restricciones en la pizarra dentro del salón de trabajo, donde todos

puedan verlos. De esta manera, se cuidan los atributos de calidad que forman a la estructura de la arquitectura y se aseguran que la funcionalidad sea asignada a esa estructura.

Además, en [Nord] comentan que la arquitectura ayuda a localizar los efectos de los cambios en el diseño que son causados por cambiar los requerimientos de la funcionalidad; por lo que, la arquitectura es influenciada por los requerimientos de los atributos de calidad y no es afectada por los cambios en los requerimientos de la funcionalidad. Por lo tanto, la arquitectura representa las decisiones de diseño más importantes y estas son hechas, en su mayoría, durante el método ADD.

Otro punto interesante, mencionado en [Nord], es que debido a que en la primera iteración se hace la primera articulación de la arquitectura, es necesariamente de grano grueso, y por lo tanto, se debe dedicar el tiempo suficiente para realizar un buen diseño.

De acuerdo con [Nord], las actividades de diseño en XP, comienzan donde el método ADD termina.

Además, se menciona en [Nord], que durante el método ADD, se hace un proceso de descomposición recursivo en donde, en cada etapa de la descomposición, el equipo de desarrollo elige las tácticas arquitectónicas y patrones para satisfacer un conjunto de escenarios de atributos de calidad. Entendiendo como táctica arquitectónica, a la manera de satisfacer una medida de la respuesta de un atributo de calidad, manipulando algunos aspectos del modelo del atributo de calidad a través de las decisiones de diseño arquitectónicas. De esta manera, se describe que la táctica provee una generación y prueba del modelo de diseño de la arquitectura, en donde la descomposición variará, dependiendo del contexto del negocio, conocimiento del dominio y cambio de la tecnología.

A continuación se puntualiza el valor que el método ADD agrega a XP [Nord]:

- El método ADD soporta el enfoque XP mediante la permisión de una descomposición inicial *breadth-first* para el primer nivel de descomposición, seguido de una descomposición *depth-first* para explorar el riesgo asociado con los cambios mediante un prototipo.
- El método ADD crea y documenta la arquitectura de software usando vistas (*views*). [Nord]

Análisis temprano: ATAM y CBAM

En [Nord] comentan que se puede utilizar el método ATAM y CBAM para hacer un seguimiento temprano de los riesgos técnicos y de negocio en el proceso y para ayudar a priorizar las historias de usuarios para la siguiente entrega. Además de que, el método ATAM y CBAM, ayuda a los practicantes de XP a entender como las decisiones de diseño, que se hacen mientras se crea la arquitectura de un sistema complejo, interactuarán.

El propósito del método ATAM de acuerdo con [Nord], es evaluar las consecuencias de las decisiones de la arquitectura a la luz de los requisitos de los atributos de calidad y los objetivos del negocio.

Por otro lado, se comenta en [Nord], que el método CBAM, ayuda a hacer el análisis de las decisiones arquitectónicas, las cuales son realizadas durante la ejecución de ATAM en una hoja de la ruta estratégica del diseño y evolución del software, mediante el hecho de asociar prioridades, costos y beneficios con cada decisión arquitectónica.

El valor que ATAM añada a XP, de acuerdo con [Nord], es que mediante el hecho de definir paso a paso un enfoque de evaluación de la arquitectura, se producen los temas de posibles riesgos y muestran su impacto en el alcance de las metas del negocio. [Nord]

Por último, la Tabla 34 resume los beneficios de los métodos QAW, ADD, ATAM y CBAM dentro de XP.

Actividades de XP	Valor agregado por las actividades arquitectónicas
Planeación e historias de usuario	Las metas del negocio determinan los atributos de calidad utilizando QAW – Las historias de usuario se complementan con los escenarios de atributos de calidad, que capturan los asuntos de los stakeholders referentes a los requerimientos de atributos de calidad. – Los escenarios ayudan a los *stakeholders* para comunicar los requerimientos de atributos de calidad a los desarrolladores y que estos influyan en el diseño. – El refinamiento y priorización de los escenarios, le dan al cliente y a los desarrolladores la información adicional para ayudarles a escoger sus historias para cada *Sprint* . – Durante el taller, los *stakeholders* mejoran la comunicación con el cliente.
Diseño	Los atributos de calidad dirigen el diseño, utilizando el método ADD – El enfoque *step-by-step* para definir la arquitectura de software, complementa el diseño incremental; el nivel de detalle es flexible. La arquitectura permite una mejor planeación para que los desarrolladores puedan hacer mejores estimaciones sobre el impacto que pueden causar los nuevos requerimientos. – Los desarrolladores crean la arquitectura suficiente para asegurar que el diseño va a producir un sistema que satisfaga los requerimientos de atributos de calidad y para mitigar los riesgos. – Las tácticas arquitectónicas ayudan a la refactorización, la cual es dirigida por las necesidades de los atributos de calidad (tales como hacerlo más rápido o más seguro)
Análisis y pruebas	El análisis del diseño provee una temprana retroalimentación al utilizar ATAM o CBAM – El equipo ágil puede utilizar los escenarios para evaluar el diseño y proveer una entrada para el análisis durante las pruebas. – La evaluación de la arquitectura tiene una noción semejante al triaje[7] para producir tanta información como sea necesaria y priorizar sobre la base de la importancia del negocio y la dificultad de los esfuerzos arquitectónicos. – La evaluación arquitectónica provee una temprana retroalimentación para comprender los *trade-offs* técnicos, el riesgo y el ROI (Retorno de inversión) de las decisiones arquitectónicas. El riesgo está relacionado con las decisiones técnicas y las metas del negocio, dada la justificación de los desarrolladores para invertir recursos para mitigarlos.

Tabla 34 *Extreme Programming* y las actividades centradas en la arquitectura

[7] Triaje (del francés *triage*) este término se emplea para la selección de pacientes en distintas situaciones y ámbitos. En situación normal se privilegia la atención del paciente más grave, el de mayor prioridad.

4.2.3 Propuestas de integración de las prácticas arquitectónicas dentro de los métodos ágiles.

A continuación se muestran algunos consejos, técnicas, guías y métodos para integrar las prácticas arquitectónicas dentro de las actividades que realizan los equipos de desarrollo ágil.

4.2.3.1 Método de Faber

El autor del artículo [Faber] ve al proceso de diseño de la arquitectura constituido por dos fases: preparación y soporte. La Figura 4-5 muestra un resumen de las tareas importantes en las dos fases de su método.

Figura 4-5 Tareas relevantes del método [Faber]

A continuación se describe brevemente cada una de las fases del método.

Preparación

Principalmente se describe que, durante la etapa de preparación, el arquitecto debe gana el suficiente conocimiento tecnológico para dar un eficiente soporte al desarrollo y el suficiente conocimiento sobre el sistema para el inicio del desarrollo, especialmente en términos de los requerimientos de calidad del sistema.

Especificación de la calidad del sistema: la preparación debería iniciar con una descripción específica de la calidad del sistema, que después será transformada en tareas para el equipo de desarrollo. El resultado es una colección completa de las descripciones priorizadas de las

cualidades del sistema. Para esta etapa, de especificación de la calidad del sistema, se recomienda utilizar el método de escenarios, que tiene como fin, describir los atributos de calidad de un sistema que soporten especificaciones de calidad que se puedan medir, especialmente para hacer pruebas.

Panorama general del sistema (*big picture*): el fin de esta tarea es crear toda la descripción arquitectónica, que más tarde, será utilizada por los desarrolladores y la cual deberá proveer la suficiente información para:

- Identificar las partes del sistema que serán cruciales para las cualidades del sistema.
- Implementar el esqueleto del sistema.
- Organizar el equipo desarrollo en sub equipos.

Además, en los detalles se deberá proveer las guías que los desarrolladores necesiten para comenzar la implementación.

Estructura y reglas: las convenciones que son aplicadas a todo el sistema, son costosas de modificar, por ejemplo: *frameworks*, guías de estilo para interfaces de usuario, guías de codificación, etc. Por lo que deben participar los desarrolladores de la aplicación lo más temprano posible para ayudar al arquitecto a evitar futuras modificaciones a estas convenciones.

En el artículo [Faber], Roland faber comenta que el proyecto en el que estaba trabajando, ellos separaron la interfaz de usuario de la lógica de negocio dentro de diferentes procesos del sistema, pero después de meses de trabajo encontraron que la inversión dentro de interfaces relacionadas no era viable. Por lo que tuvieron que modificar la regla, la cual no se habría requerido si ellos hubieran involucrado a los desarrolladores en una etapa más temprana.

Prototipos: el panorama general del sistema (*big picture*) muestra las partes cruciales de un sistema, el cual nos puede ayudar a abordar el tema de la calidad del sistema. Como ejemplo, se tiene el desempeño, en el cual el arquitecto debe asegurar la factibilidad de este atributo mediante el planteamiento de un prototipo apropiado que lo aborde correctamente.

Esqueleto del sistema: además de escribir las especificaciones, el resultado de la preparación debería ser un prototipo que muestre las nuevas conexiones del sistema y sus interfaces. Éste no deberá contener funcionalidad, pero sí debe mostrar el marco de trabajo (los huesos) y las interfaces (las uniones) con las que se trabajarán. Además, este esqueleto deberá soportar un prototipo del comportamiento del sistema y permitir a los desarrolladores integrar sus resultados tempranamente, algo similar a como lucirá el sistema final. Lo anterior. Ayuda a minimizar las refactorizaciones que se puedan hacer en el sistema, las cuales son a causa de las modificaciones del ambiente del sistema.

Durante la creación del esqueleto del sistema, el arquitecto ganará experiencia con la tecnología, probando su arquitectura al momento de codificar durante la etapa de preparación. Después de probar su arquitectura, el puede enseñarle al equipo de desarrollo cómo utilizarla.

Soporte: durante esta fase, el arquitecto deberá trabajar en las partes importantes del desarrollo: en la aplicación crucial, los atributos de calidad del sistema y el uso del marco de trabajo (*framework*).

Establecer la arquitectura de aplicación: de acuerdo con la calidad de todo el sistema, el arquitecto define la arquitectura de aplicación. Junto con el equipo de desarrollo, se crea un esqueleto de la aplicación que sea consistente con el esqueleto del sistema. Más tarde el equipo llenará el esqueleto con la funcionalidad (la carne del esqueleto del sistema), transformando el prototipo en código y el arquitecto puede moverse al siguiente punto importante.

Retroalimentación y mejorar el marco de trabajo: participar en la implementación de la aplicación provee una excelente oportunidad para obtener la retroalimentación sobre la arquitectura y la calidad del marco de trabajo.

Solucionar problemas: Los grandes conocimientos de los arquitectos los califican para resolver problemas complicados que se presenten a lo largo del sistema.

En el artículo [Faber], se hace un intento de adaptar los servicios provistos por un arquitecto dentro de *Scrum*. Ellos encontraron, que el rol del *product owner* está orientado fuertemente a la funcionalidad y que necesita una contraparte, quien sea el responsable de la calidad del sistema, esto es, "el representante de la arquitectura". Un rol similar fue también propuesto por J. Eckstein, en su libro "*Agile software 131ficial131ent in the large"* quien nombra a este rol como "*chief architect*". El representante de la arquitectura deberá cooperar muy de cerca con el *product owner,* quien es la persona encargada de elaborar el *backlog* mediante el balance de las prioridades de las tareas y que se hagan, de acuerdo con los requerimientos de calidad del sistema y los de funcionalidad. La Figura 4-6 es una extracción del artículo [Faber], el cual tiene por objetivo mostrar la continua transformación de los requerimientos en tareas de desarrollo para los cada *sprints* dentro de *Scrum*.

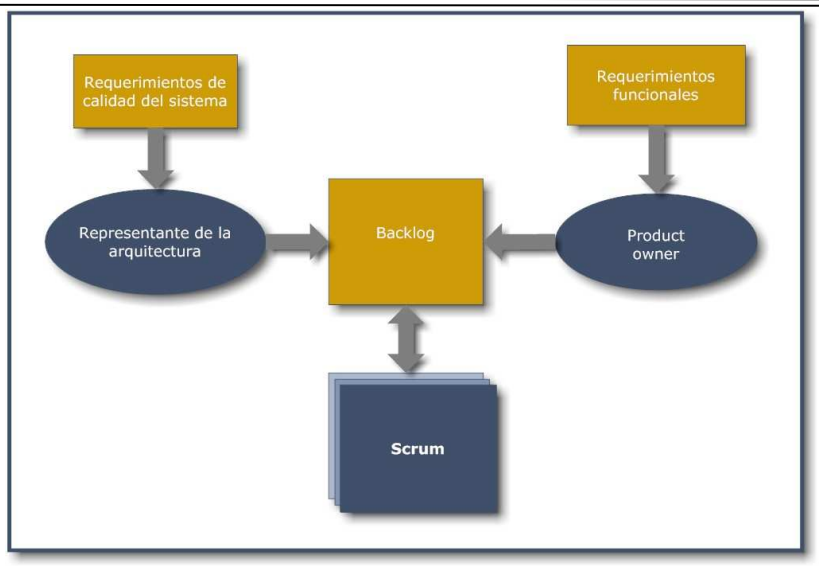

Figura 4-6 Transformación de los requerimientos en tareas de desarrollo para *Scrum*, involucrando tanto al *product owner* como al representante de la arquitectura [Faber].

En la experiencia del autor del artículo [Faber], comenta que sin el representante de la arquitectura, las tareas concernientes a la funcionalidad del sistema, suelen ser la de mayor prioridad, lo cual dirige a altos esfuerzos de refactorización cuando las deficiencias en la calidad del sistema llegan a ser evidentes. Los cambios en la calidad del sistema suelen ser más caros de hacer que los cambios en la funcionalidad. Por lo anterior, es importante asegurarse de que la calidad del sistema está siendo abordada adecuadamente y tan pronto como sea posible.

Todos es comunicación: más que la tecnología, la comunicación es la clave para el éxito del proyecto. Debido a que el arquitecto es el responsable de la calidad del sistema, el debe organizarse y comunicarse con todos los roles y *stakeholders* del proyecto. Por lo anterior el arquitecto requiere las siguientes características:

- La habilidad de construir una red de comunicación efectiva.
- El arquitecto debe ser un comunicador apasionado.
- Requiere dirigir la atención para convencer a todos, de que, comunicar los asuntos arquitectónicos es muy importante y previsto.

Los arquitectos que trabajan con muchos equipos, necesitan organizar su comunicación. Un arquitecto hace cumplir la comunicación entre arquitectos y provee la plataforma para las decisiones arquitectónicas de los asuntos generales.

Roland Faber [Faber], dice que en su experiencia han encontrado que la comunicación arquitectónica debe ser constantemente estimulada.

Alcances del método de Faber

Al implementar el método propuesto en [Faber], se tiene como resultado las siguientes mejoras:

- La cantidad de código de la aplicación se decremento, porque el marco de trabajo utilizado a cumplió con las necesidades funcionales de la aplicación.
- La refactorización del marco de trabajo aumento y como consecuencia se tuvo un mejor número de rompimiento de reglas arquitectónicas.
- Se obtuvo una integración temprana de los prototipos y código parcial del producto en un esqueleto del sistema.
- Se alcanzo una consistente arquitectura de aplicación en el proyecto, a través de una directa participación de los arquitectos.
- El reuso de conceptos y código se incremento, ya que los arquitectos dieron soporte directo al desarrollo.
- Se evito diseño el uso de enfoques de diseño de aplicaciones costosas mediante el establecimiento de la comunicación con los arquitectos.
- Se gano un conocimiento temprano, sobre el mejoramiento del marco de trabajo para las siguientes entregas del sistema; debido a la experiencia de desarrollo de la aplicación directa de los arquitectos.

4.2.3.2 Método de puntos de interacción entre arquitectura y agilidad

James Madison [Madison 00] dice que la arquitectura es la encargada de establece la tecnología de pila, crear patrones de diseño, mejorar los atributos de calidad y ayuda a la comunicación con todas las partes interesadas (*stakeholders*)".

Madison ve a la arquitectura ágil como un enfoque que usa técnicas agiles para dirigirse hacia una buena arquitectura. Además, dice que para obtener una arquitectura ágil se requiere un arquitecto que entienda el desarrollo ágil, interactué con el equipo en puntos bien definidos, que influya en ellos usando habilidades críticas fácilmente adaptables de su experiencia arquitectónica con otros enfoques y que aplique las funciones arquitectónicas que sean independiente de la metodología del proyecto.

Puntos de interacción de la arquitectura

En la Figura 4-7 se muestra el marco de trabajo hibrido de *Scrum*, Extreme Programming y de prácticas de gestión de proyectos secuenciales que Madison ha encontrado efectivas para guiar el trabajo arquitectónico en 14 proyectos ágiles [Madison 00].

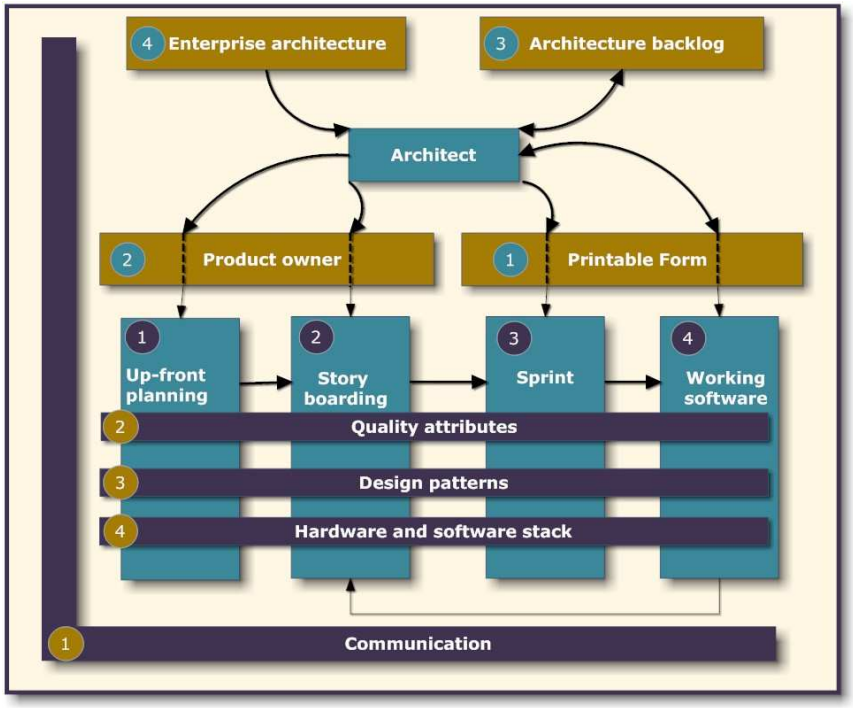

Figura 4-7 Marco de trabajo hibrido para guiar una arquitectura ágil [Madison 00]

La Tabla 35 explicar cada uno de los elementos expuestos en el marco de trabajo híbrido de James Madison.

Descripción de los elementos en el marco de trabajo híbrido

Categoría	Elemento	Descripción
Puntos de interacción	*1. Up-front planning*	Establecer la dirección de la arquitectura, de la misma manera que en una proyecto secuencial.
	2. Storyboarding	Estructura las necesidades del negocio, el trabajo de la arquitectura y poner a todos al tanto.
	3. Sprint	Construir la funcionalidad como partes del equipo, cuando la participación directa sea valiosa.
	4. Working software	Revisar lo que actualmente se está entregando, para medir el estado de la arquitectura.
Habilidades críticas	*1. Sprintable form*	Dividir el trabajo de la arquitectura en pequeñas y medibles unidades.
	2. Product owner	Cuantificación de la arquitectura en términos claros del valor de negocio.
	3. Architecture backlog	Dar seguimiento de los asuntos de interés para la arquitectura y el equilibrio con las prioridades del negocio.
	4. Enterprice architecture	Conocer ampliamente el panorama de la arquitectura y el uso de cada proyecto para avanzar.
Función de la arquitectura	*1. Communication*	Mantener a todos los *stakeholders* informados sobre el valor y estado de la arquitectura.
	2.Quality attributes	Medir la Mantenibilidad, escalabilidad, extensibilidad y similares del sistema.
	3. Design patterns	Hacer el esquema de las estructuras que dan forma al trabajo de implementación.
	4. Hardware and software stack	Escoger el hardware y software apropiados para el proyecto.

Tabla 35 Descripción de los elementos del marco de trabajo híbrido [Madison 00]

La Tabla 36 explica las intervenciones de las cuatro funciones de la arquitectura en los cuatro puntos de interacción ágiles propuestos por Madison.

Aplicación de las funciones de la arquitectura en puntos de interacción ágil

Función de la arquitectura	Punto de interacción			
	Up-front planning	***storyboarding***	***Sprint***	***Working software***
Comunicación	▪ Entender los objetivos del negocio. ▪ Obtener la entrada del equipo técnico. ▪ Comunicar la dirección general con todos.	▪ Facilitar activamente sesiones para establecer el storyboarding. ▪ Trabajar las historias de usuario concernientes a la arquitectura, en particular las que se muestran en las siguientes casillas:	▪ Asistir a las reuniones diarias. ▪ Construir funcionalidad como un medio para comprender mejor el sistema. ▪ Mentor y asistir tanto como la experiencia lo permita.	▪ Asistir a la revisión del *sprint.* ▪ Revisión de la documentación. ▪ Abogado de la refactorización para promover el valor de la arquitectura con el equipo y el *product owner.*
Atributos de calidad	▪ Establecer intervalos de referencia para los atributos de calidad. ▪ Establecer cuáles son los atributos dominantes durante el análisis de ventajas y desventajas.	▪ Añadir historias de usuario para mejorar atributos específicos, incluyendo la refactorización.	▪ Plasmar los atributos en el código, explícitamente y como una norma para el trabajo de la construcción. ▪ Participar en el diseño o en la construcción para mejorar los atributos de calidad.	▪ Verificar que la solución liberada se ajusta a los intervalos de referencia preestablecidos. ▪ Ajustar los intervalos de referencia si el trabajo de la construcción indica una necesidad para el ajuste.
Patrones de diseño	▪ Elegir los patrones de diseño más importantes. ▪ Hacer un esquema general de las interacciones entre los patrones que son significantes.	▪ Añadir historias de usuario para construir patrones de diseño, incluyendo la refactorización.	▪ Detallar los patrones de diseño elegidos. ▪ Participar en la construcción de los patrones de diseños que son críticos para el sistema.	▪ Verificar que el patrón de diseño liberado es válido. ▪ Ajustar el patrón de diseño tanto como el trabajo de la construcción lo permita.
Pila del hardware y software	▪ Reutilizar la pila corporativa. ▪ Crear prototipos tempranos para verificar los supuestos. ▪ Planear cuidadosamente; los cambios en el hardware y software que son inherentemente no ágiles.	▪ Añadir historias de usuario para validar el software y el hardware.	▪ Validar el hardware y software elegido en *sprints* tempranos. ▪ Cambiar la pila del hardware y software pronto y rápido si esta lo necesita.	▪ Verificar el hardware y el software mediante la continua liberación de funcionalidad al negocio. ▪ Desplegar en otros entornos de forma rutinaria.

Tabla 36 Aplicación de las funciones de la arquitectura en puntos de interacción ágil [Madison 00]

A continuación se describe brevemente cada una de las etapas del marco híbrido.

Up-Front Planning

Toda función arquitectónica comienza en un proyecto ágil con un "*up-front planning",* tal y como se hace en cualquier otro proyecto, independiente de la metodología.

Para esta etapa, el arquitecto tienes las siguientes actividades:

- Hacer las principales decisiones sobre el hardware y el software que se utilizara, sobre todo usando estándares corporativos existentes.
- Establecer patrones de diseño importantes nivel detallado y ampliamente.
- Identificar la oportunidad para reutilizar componentes o servicios.
- Generar diagramas con alto nivel de abstracción.
- Crear un bosquejo de los atributos de calidad, tanto técnicos como de negocio, y la línea base de sus ventajas y desventajas.
- Establecer los canales de comunicación mediante reuniones con los stakeholders, para entender sus asuntos de interés y compartir la dirección técnica general con ellos.

Madison comenta que hasta este momento, las tareas son similares a las que se harían en un enfoque no ágil. La excepción es que la dirección de la arquitectura debería incluir un rango de opciones en lugar de una solución especifica. A su vez, Madison haciendo cita a (M. Poppendieck y T.Poppendieck, 2003) comenta que un conjunto de posibles soluciones para la arquitectura sería aceptable, basándose en el supuesto de que el conocimiento empírico reunido por todos los participantes, mientras construyen el sistema, hará evidentemente mejores opciones.

Storyboarding and Backlogs

Se construye el *producto/sprint backlog,* con el arquitecto como un clave stakeholder. El arquitecto debe atender las primeras sesiones del *storyboarding* y contribuir con las historias de usuario que sean significantes o influyan a la dirección de la arquitectura. El arquitecto debe atender el curso del *storyboarding* entre los *sprints* para que contribuyan a la arquitectura las historias de usuario o corregir las deviaciones indeseables. El arquitecto debe trabajar con el *product owner* para priorizar estas historias con las historias de usuario del negocio y construirlas en conjunto con la funcionalidad del negocio en *sprints.*

Sprint Participation

Madison haciendo cita a (V. Subramaniam y A. Hunt, 2006) dice que “escribir código es la manera más poderosa de asegurarse de que el arquitecto comprende completamente la arquitectura que está produciendo”.

Para que lo anterior resulte favorable, Madison dice que se requiere que el arquitecto colabore fuertemente con el equipo durante el *sprint,* comprendiendo los objetivos y ayudando con los desafiantes problemas de diseño.

Working software

Después de cada sprint, el equipo y el *product owner* deben presentar el trabajo de software en una revisión formal del *sprint;* de esta mañanera todos los *stakeholders,* uno de ellos el arquitecto, puedan observar todo el progreso y proveer una retroalimentación. El *sprint review* tiende a durar sólo algunas horas, por lo que el arquitecto de software debería comenzar la revisión del trabajo de software varios días antes de la revisión oficial.

Además, Madison comenta que un proyecto de software bien ejecutado, requiere entregas iterativas de documentación con el trabajo de software, incluyendo la documentación arquitectónica (el código no documentado y la funcionalidad del sistema no debería de ser considerado trabajo de software). Otro punto que menciona Madison es, que la revisión del documento de arquitectura conforme va surgiendo en cada *sprint,* es una forma útil de revisión de la arquitectura.

A continuación se explican las habilidades que el arquitecto debe tener dentro del marco híbrido.

Saber Descomponer en forma *Sprintable*

El desarrollo ágil requiere que el *product owner* descomponga las historias de usuario hasta que estas sean lo suficientemente pequeñas como para ser ejecutadas en un *sprint*, pero sin dejar de ser lo suficientemente sustancial como para mostrar el valor del negocio. Del mismo modo, el equipo técnico se descompone las historias de usuario en una forma que pueda ser eficientemente construir dentro de los *sprints.*

La contribución del arquitecto en la descomposición consiste de identificar los límites de la importancia arquitectónica y el trabajo con el *Product Owner* y el equipo técnico para asegurar que la descomposición general del trabajo sigua a esos límites.

Saber apoyar al *Product Owner*

El camino, desde la descomposición del problema, hasta el trabajo del software, corre a través del *Product Owner,* quien requiere del arquitecto para promover el valor del trabajo de la arquitectura con él. Los dos aspectos más críticos de esto se refieren a la creación y refactorización de el diseño del sistema e indicando el valor de los atributos de calidad en términos del valor del negocio. Si el *Product Owner* no comprende el valor del trabajo de la arquitectura, continuamente se obtendrá una baja prioridad en el *Product Backlog* para este trabajo, y como resultado se tendrá una arquitectura deficiente.

Ejemplo de cómo se pueden ver a los atributos de calidad en términos de valor para el negocio:

Mantenibilidad: La Mantenibilidad resulta en la construcción más rápida de funcionalidad de negocio que se hará en *sprints* posteriores, con mejoras más rápidas en los cambios para la vida del sistema, dando lugar a una mayor velocidad al mercado.

Madison haciendo cita a (M. Fowler, 2003) dice: “la promoción más importante tiene lugar en los primeros *sprints*, cuando hay un alto valor en la construcción de los componentes de la arquitectura y que son difícil de revertir.”

Crear el *backlog* de la arquitectura

Madison comenta respecto a la importancia de priorizar correctamente el *product backlog.* El dice que al igual que con todos los usos del *backlog,* el trabajo será ordenado por prioridad; y si no hay suficiente tiempo o dinero, puede que exista trabajo que pudiera no hacerse. Lo anterior puede

resultar en una arquitectura muy comprometida. Ciertamente si el trabajo de la arquitectura recibe una baja prioridad que nunca se llegue a hacer, entonces la arquitectura se degradara.

Por lo anterior Madison comenta que para mantener un enfoque claro sobre la arquitectura y para facilitar la puntuación de la arquitectura, un separado pero conectado *product backlog* llamado "*architecture backlog"* debería realizar un seguimiento del trabajo de la arquitectura.

Sobre el *architecture backlog* Madison dice que ésta debe ser mantenida por el arquitecto, comunicada tanto al *product owner,* como al equipo en el tiempo y lugar apropiado, y sus artículos movidos hacia el *backlog* principal de acuerdo con el juicio del *product owner* que es influenciado por el arquitecto.

Por último, Madison dice que la agilidad y la arquitectura no son opuestas. El desarrollo ágil le da al arquitecto la oportunidad de trabajar muy de cerca con el equipo técnico y del negocio para guiar continuamente al sistema en la dirección de una arquitectura correcta.

4.2.3.3 Método de desarrollo de una arquitectura ágil de Microsoft

Microsoft [MicrosoftGuide], publicó una guía para hacer un diseño de arquitectura ágil, la cual proporciona las directrices detalladas que se deben seguir para la construcción de la arquitectura de una aplicación, que está siendo desarrollada bajo un enfoque ágil. A continuación se explica brevemente los puntos importantes de esta guía:

Entrada para el diseño de la arquitectura:

- Casos de uso y escenarios de uso
- Requisitos funcionales
- Requisitos no funcionales (atributos de calidad tales como rendimiento, seguridad y confiabilidad)
- Requisitos tecnológicos
- Ambiente de despliegue
- Restricciones

El diseño debe producir la siguiente salida:

- Casos de usos significantes para la arquitectura
- Puntos Críticos de Arquitectura
- Arquitecturas candidatas
- Esbozos de Arquitectura

La guía recomienda los pasos mostrados en la Figura 4-8.

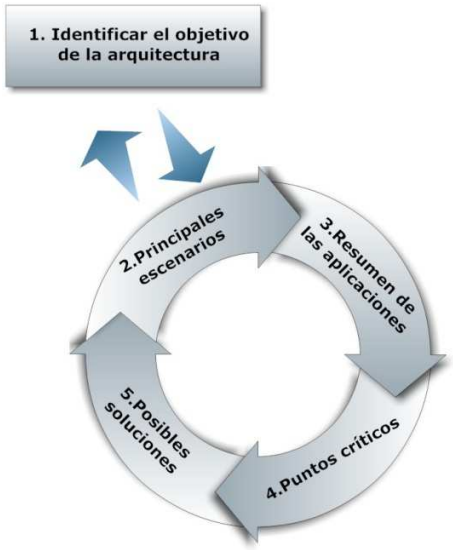

Figura 4-8 Pasos del método para una arquitectura ágil

A continuación se describe brevemente cada uno de ellos:

Paso 1 Identificar el objetivo de la Arquitectura: Los objetivos claros, ayudan a poner en buenos términos a la arquitectura, y ayuda a centrarse en la solución de los problemas en el diseño. Los buenos objetivos ayudan a determinar cuándo se ha terminado, y cuando estamos listos para pasar a la siguiente etapa.

Paso 2 Principales escenarios: Durante esta etapa, se utilizan los escenarios clave para mostrar el diseño a los principales *stakehoders* y para evaluar las posibles arquitecturas cuando estén listas.

Paso 3 Visión general de la aplicación: Entender el tipo de aplicación, la arquitectura de despliegue, los estilos de arquitectura y tecnologías para conectar su diseño al mundo real en que la aplicación será ejecutada.

Paso 4 Puntos Críticos: Identificar los puntos críticos sobre la base de los atributos de calidad y del cuadro de la arquitectura. Estas son áreas donde los errores suceden más a menudo cuando se diseña una aplicación.

Paso 5 Posibles soluciones: Crear una arquitectura candidata o un esbozo arquitectónico y evaluarlo contra sus escenarios principales, los puntos críticos y las restricciones del despliegue.

La guía recomienda hacer los pasos anteriores en forma iterativa. A continuación se detallas los pasos:

Paso 1 Identificar el objetivo de la arquitectura.

Según JD Meier, Director Gerente de Programa de patrones y prácticas de Microsoft, el objetivo de este paso es determinar "cuánto tiempo y cuánta energía gastar en etapas posteriores, así como dirigir su esfuerzo global". El resultado del paso 1 debería ser:

- Construir un prototipo
- Identificar los principales riesgos técnicos
- Probar los potenciales caminos
- Compartir modelos y entendimientos

Paso 2 Escenarios Principales.

Los mejores escenarios son identificados por los siguientes criterios para los casos de uso, de acuerdo con JD Meier:

- Son importantes para el éxito y la aceptación de la aplicación.
- Ejercitan lo suficiente el diseño como para ser útil en la evaluación de la arquitectura.

Paso 3 Visión general de la aplicación.

Una visión general de la aplicación es necesaria para introducir detalles de la vida real y dejar el diseño más concreto, y se crea a través de los siguientes pasos:

- **Determinar el tipo de su aplicación**. En primer lugar, determinar qué tipo de aplicación se está construyendo. Por ejemplo, si se trata de una aplicación móvil, un cliente rico, una aplicación de Internet de gran porte, un mensaje de autobús, una aplicación web o una combinación de estas.
- **Comprender las restricciones impuestas de despliegue**. Comprender el entorno de despliegue y determinar qué impacto tendrá en su arquitectura.
- **Identificar los estilos arquitectónicos más importantes**. Determinar el estilo arquitectónico que se usará en el diseño. Por ejemplo, si se va a construir una arquitectura orientada a servicios, cliente-servidor, arquitectura en capas, un bus de mensajes, o alguna combinación de estas.
- **Determinar las tecnologías relevantes**. Finalmente, identificar las opciones de las tecnologías pertinentes basadas en el tipo de aplicación y otras restricciones, y determinar cuáles tecnologías pueden utilizarse en la arquitectura.

Es importe mencionar que, la guía publicada en [MicrosoftGuide], ofrece consejos para todos los pasos mencionados anteriormente.

Paso 4 Puntos críticos.

En este paso, se identifican los puntos críticos en la arquitectura de aplicación para entender las áreas en las que es probable que ocurran errores. Los puntos críticos pueden ser organizados en

torno a los atributos de calidad. Una larga lista de puntos críticos presentados por la guía incluyen: disponibilidad, interoperabilidad, durabilidad, confiabilidad, seguridad, etc.

Paso 5 Posibles soluciones.

Después de identificar los principales puntos críticos, el primer proyecto de arquitectura puede ser producido. Después de eso, se regresa al paso 2, para validar las posibles arquitecturas y, a continuación, se siguen los pasos 3 a 5 para generar una nueva arquitectura candidata. El proceso se repite iterativamente mejorando con cada iteración.

4.3 Conclusiones

De la revisión bibliográfica expuesta en este capítulo, se puede afirmar que a los métodos ágiles aún les falta mucho para ser una alternativa de desarrollo confiable y aplicable en cualquier contexto y dominio. Su evolución deberá comprender prácticas de análisis, síntesis y evolución arquitectónicas. Las prácticas arquitectónicas fortalecen a cualquier enfoque de desarrollo y como tal, no se debería de prescindir de ellas bajo ninguna circunstancia; como consecuencia, las prácticas de arquitectura de software son independientes del enfoque utilizado, más sin embargo, la manera de implementarlas es inherente al enfoque de desarrollo elegido, lo que resulta en una adaptación acorde al enfoque.

También podemos concluir que, la arquitectura de software es un medio en el cual, todos los involucrados en el proyecto verán plasmados sus asuntos de interés con respecto al sistema, estarán de acuerdo y seguirán hasta que sus necesidades acordes al valor del negocio cambien. Por lo que es importante documentar lo necesario respecto a la arquitectura.

Podemos ver que han existido intentos de unificar las prácticas arquitectónicas dentro de métodos ágiles, obteniendo exitosos resultados en la implementación de su método. Lo anterior, gracias a que se demostró que la gran mayoría de los usos de la arquitectura vienen a fortalecer y complementar el cumplimiento de los cuatro valores ágiles.

Por último, se concluye que las prácticas arquitectónicas no son opuestas a las establecidas por los métodos ágiles, más bien, las complementan y que es, el resultado del análisis del contexto del sistema, lo que dictaminará la cantidad de prácticas arquitectónicas necesarias en el proyecto.

En el capítulo 5 se propone un marco teórico para incluir a la arquitectura en los métodos ágiles, el cual toma muchas de las prácticas investigadas en este capítulo.

5 Marco conceptual de arquitectura ágil

En este capítulo se definen los elementos que componen al marco conceptual de arquitectura ágil, de aquí en adelante nos referiremos a esté marco conceptual como MCAA. El MCAA es un marco de trabajo híbrido qué combina las actividades de gestión de proyectos de *Scrum* [ScrumOrg], con las prácticas de desarrollo de *Extreme Programming (XP)* [BeckKent]. El MCAA reúne varias prácticas enfocadas en el desarrollo de una arquitectura inicial dentro de los métodos ágiles, en particular, se emplean prácticas definidas por el *Software Engieenering Institute* [SEI2010] que han sido utilizadas por practicantes de métodos ágiles.

Durante este capítulo se explica cómo las prácticas arquitectónicas, definidas aquí, apoyan tanto a las actividades de *Scrum* como a las prácticas de *Extreme Programming.*

5.1 Alcance

5.1.1 Campo de aplicación

Él modelo MCAA es aplicable a cualquier empresa, organización, departamento o proyecto que sea dirigido por un proceso de desarrollo ágil.

Él modelo MCAA provee una guía para adaptar las prácticas arquitectónicas dentro de los métodos ágiles, así como también introduce el rol de arquitecto de software ágil.

Usando el MCAA, se obtienen los siguientes beneficios

- Soporte al cumplimiento de los valores y principios ágiles.
- Un sistema que cumpla con el valor de negocio y los atributos de calidad para cada iteración.
- Manejo de la complejidad inherente al contexto del proyecto.
- Menor esfuerzo en la refactorización.
- Manejo de activos de arquitectura.
- Comprensión temprana del proyecto en desarrollo.

5.2 Audiencia

Él modelo MCAA está dirigido, más no limitado, para grupos de trabajo ágil que:

- desarrollen sistemas complejos de software, que por la complejidad del proyecto sea necesario incluir prácticas arquitectónicas inherentes a él.

- estén desarrollando sistemas con nuevas tecnologías y prácticas arquitectónicas aun no probadas.
- desarrollen sistemas críticos de software; donde los atributos de calidad tengan que cumplirse rigurosamente.

5.3 Conceptos básicos

La Tabla 37 y Tabla 38 muestran los conceptos básicos utilizados a lo largo de este capítulo.

Actividad	La actividad representa un grupo de tareas. [SPEM 2008]
Atributos de calidad	Definen los requerimientos de la aplicación en términos de escalabilidad, disponibilidad, facilidad de cambio, portabilidad, usabilidad, desempeño, etc [IanGordon].
Directriz arquitectónica	Una directriz arquitectónica es cualquier requerimiento funcional, restricción de diseño, requerimiento de atributo de calidad u otro que tenga un impacto sobre la estructura de una arquitectura [Rob Wojcik].
Producto de trabajo	Es el elemento del contenido del método que es usado, modificado y producido por las tareas definidas [SPEM 2008].
Product Owner	En *Scrum*, es la persona responsable de gestionar el *Product backlog* con el fin de maximizar el valor del proyecto. El *product owner* representa a todos los interesados en el proyecto. [RedmondWashington]
Requerimientos arquitectónicos	También llamados requerimientos significantes para la arquitectura o casos de uso arquitectónicos, son esencialmente los requerimientos de calidad y los requerimientos no funcionales para la aplicación. Los requerimientos no funcionales están divididos en tres categorías: restricciones de técnicas, restricciones de negocio y atributos de calidad [IanGordon].
Restricciones técnicas	Restringen las opciones de diseño, mediante la especificación de ciertas tecnologías que la aplicación debe usar. Usualmente no son negociables [IanGordon].
Restricciones de negocio	También son restricción es de diseño, pero por razones del negocio y no técnicas [IanGordon].
Rol	El rol es un elemento del contenido de un método, que define un conjunto de habilidades relacionadas, competencias y responsabilidades. Los roles son usados tanto en la definición de tareas para definir quién las realiza, como para definir un conjunto de productos de trabajo de los cuales ellos son responsables. [SPEM 2008]

Tabla 37 Conceptos básicos (1/2)

SAEM	Software Architecture Evaluation Method
Táctica arquitectónica	Una táctica arquitectónica es un conjunto de decisiones de diseño que influyen en las propiedades de los atributos de calidad de un sistema. Por ejemplo, la táctica *Ping-Echo* para detectar fallas puede ser empleada durante el diseño para influir en la propiedad de "disponibilidad" de un sistema. La táctica de *ocultamiento de información* puede ser empleada durante el diseño para influir en la propiedad de "modificabilidad" de un sistema [Rob Wojcik].
Tarea	Es el elemento del contenido de un método y la definición de trabajo que define el trabajo que se está realizando mediante instancias de la definición de Roles. Una tarea está asociada con la entrada y salida de productos de trabajo. [SPEM 2008]
Trade-off	Cuando hay un compromiso entre dos o más atributos de calidad debido a que estos se contraponen, hacer un *trade-off* significa resolver la contraposición haciendo un balance de acuerdo con la prioridad de cada uno de estos atributos de calidad. Por ejemplo, respaldar una base de datos incrementa la confiabilidad, pero esto consume recursos lo que afecta el desempeño; por lo tanto, hay una contraposición entre confiabilidad y el desempeño [PaulClements].

Tabla 38 Conceptos básicos (2/2)

5.4 Marco conceptual de arquitectura ágil

La estructura del modelo MCAA se muestra en la Figura 5-1. De esta figura podemos identificar tres principales capas: capa conceptual del cliente, capa conceptual ágil y capa conceptual de arquitectura.

Figura 5-1 Estructura del Marco Conceptual de Arquitectura Ágil (MCAA)

La capa con mayor prioridad y que aparece en la parte superior del MCAA, es la **capa conceptual del cliente**. En ella se expresan las necesidades del cliente en términos de valor para su negocio, el ambiente bajo el cual operan y las expectativas que tienen con respecto a un proceso de desarrollo de software que se adapte a sus necesidades en el mercado. Para dar soporte al cumplimiento de las necesidades para el negocio, se define la **capa conceptual ágil**; en ella se establecen los valores y principios que guían a las actividades de desarrollo de software ágil, como: la gestión del proyecto y las actividades de diseño e implementación del software. Cabe mencionar que, durante la definición de las actividades y prácticas ágiles, no se considera el soporte, rol, actividades y productos de trabajo que las prácticas arquitectónicas aportan. Es hasta la definición de la capa conceptual de arquitectura que, se definirán estos elementos. Por lo tanto, para denotar qué una actividad o práctica (en el caso de *Extreme Programming*) ágil es soportada y reforzada con las prácticas arquitectónicas, se le agregará el símbolo  . La **capa conceptual de arquitectura** es la base del MCAA y es la encargada de dar soporte a la capa conceptual ágil; en ella se definen las actividades arquitectónicas que fortalecen el cumplimiento de los valores y principios ágiles, así como también, sus actividades y prácticas.

A continuación se describe a detalle el contenido del MCAA.

5.5 Capa conceptual del cliente:

Para comprender la importancia de la arquitectura de software dentro de los métodos ágiles, es importante entender su importancia y valor que esta representa para el negocio del cliente. Por lo tanto, en la capa conceptual del cliente se exponen los siguientes puntos:

- Las necesidades a las cuales el negocio del cliente se ve comprometido debido a la globalización y mercados abiertos.
- Las circunstancias en la que el negocio está sumergido; esto es, su ambiente en el mercado.
- Las expectativas que tienen con respecto a un proceso de desarrollo de software ágil que se adapte a su modelo de negocio orientado al cliente.

La globalización y los mercados abiertos han provocado competencia entre los negocios, tanto en la venta de productos como de servicios, por lo tanto, se necesita adoptar prácticas nuevas que permitan la competitividad y supervivencia en el mundo globalizado

La necesidad de competir con empresas de clase mundial hace muy necesaria e imprescindible la búsqueda de un alto desempeño en las operaciones de un negocio, así como la entrega de productos y servicios de gran calidad [Donadio 2004].

Para que una empresa que tiene éxito hoy, y lo continúe teniendo mañana, se requiere grandes retos y adoptar prácticas que ayuden a lograr una particularidad que tenga efecto en la decisión de compra de los clientes. Para ello la evolución de la tecnología y del internet pone elementos y

medios a disposición de las organizaciones para desarrollar mejores estrategias que aseguren su permanencia en el mercado.

Las estrategias orientadas al comercio electrónico y los negocios electrónicos son cada vez más utilizadas, por lo que en la actualidad hay más empresas dispuestas a realizar inversiones en su desarrollo.

Con el fin de ilustrar el ambiente en el cual se ven sumergidos los negocios actuales, se expone la Figura 5-2, en ella podemos observar que la tecnología es sin duda, uno de los factores que intervienen en la operación del negocio.

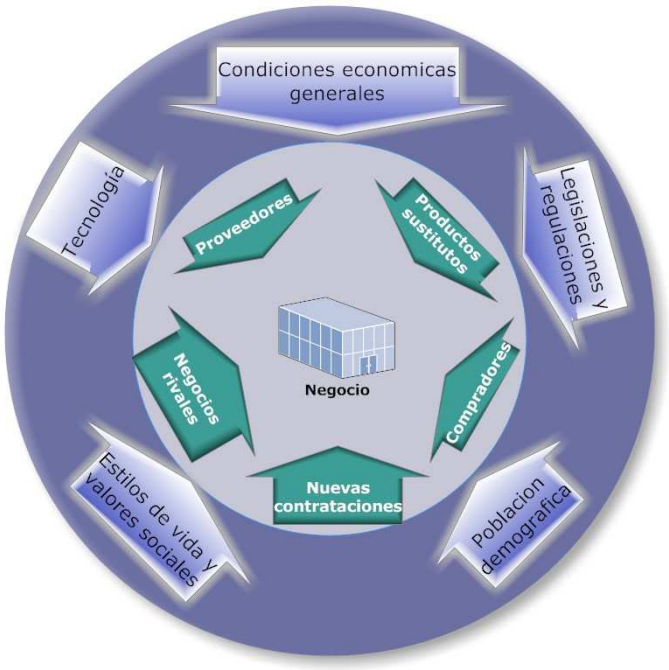

Figura 5-2 Elementos que influyen en el ambiente de un negocio [Thompson 2007]

La Tabla 39 resume las condiciones bajo las cuales operan los negocios actuales y lo que esperan de un proceso de desarrollo de software.

Condiciones	Lo que espera el cliente de un proceso de software
Empresas que operan en un entorno global que cambia rápidamente.	Que acepte requerimientos fluctuantes. Además los cambios no deberán de ser costosos.
Responder frente a nuevas oportunidades y mercados, al cambio en las condiciones económicas, así como el surgimiento de productos y servicios competitivos.	Que responda al cambio lo más pronto posible; esto quiere decir que el proceso deberá aceptar cambios o nuevos requerimientos aun en etapas avanzadas del proyecto y producir software con valor para el negocio.
Empresas que necesitan software que se desarrolle rápidamente para aprovechar las actuales oportunidades, con la finalidad de responder ante la amenaza competitiva.	Se requiere de procesos que produzcan software que de valor al negocio rápidamente, con el fin de incorporarlo en su modelo de negocio y volverse más competitivos.

Tabla 39 Condiciones bajo las cuales operan los negocios actuales y lo que esperan de un proceso de desarrollo de software

El panorama de las necesidades de un negocio surge a partir del análisis de la cadena de valores del negocio. Con el fin de comprender a la capa conceptual del cliente y el problema que deben de resolver los nuevos procesos de desarrollo de software la Figura 5-3 muestra la cadena de valores del negocio empresarial.

.

Figura 5-3 Esquema de la cadena de valor [PearceRobinson 2007]

A continuación se definen conceptos relevantes que están en torno a la cadena de valores:

Cadena de valor: el termino **cadena de valor** describe una forma de mirar un negocio como una cadena de actividades que transforman insumos en productos que los clientes valoran.

1. **Actividades primarias:** Las actividades primarias son aquellas en las que están envueltas físicamente con la creación del producto, comercialización y transferencia a los compradores, y después brindarles soporte. Como actividades primarias tenemos:
   - **Logística interna**: Son actividades, costos y activos asociados con la obtención de combustible, energía, materias primas, partes de componentes, mercancías y artículos de consumo de los proveedores.
   - **Operaciones:** Son actividades, costos y activos asociados a la conversión de insumos en el producto final (producción, ensamblado, empaquetado, mantenimiento de equipo, instalaciones, operaciones, aseguramiento de la calidad, protección del ambiente).
   - **Logística externa:** Son actividades, costos y activos que tratan con la distribución física del producto para los compradores (almacenamiento de productos terminados, procesamiento de pedidos, levantamiento de pedidos y empaquetado, envío).
   - **Marketing y ventas:** Son actividades, costos y activos asociados con el esfuerzo en ventas, publicidad y promoción, investigación de mercado y planeación.
   - **Servicios:** Son actividades, costos y activos relacionados para proveer asistencia a los compradores, tales como, instalación, piezas de repuesto, mantenimiento y reparación, asistencia técnica, preguntas y reclamaciones del comprador.

2. **Actividades de soporte:** Son las actividades que asisten a un negocio al proveer infraestructura o insumos que permitan a las actividades primarias llevarse a cabo de manera continua. Como actividades de soporte o secundarias tenemos:

   - **Administración general:** Son actividades, costos y activos relacionados con la administración general, contaduría y finanzas, asuntos legales y regulatorios, seguridad y protección.
   - **Gestión de recursos humanos:** Son actividades, costos y activos asociados con el reclutamiento, despidos, entrenamiento, desarrollo y compensación de todo tipo de personal.
   - **Investigación, tecnología y desarrollo de sistemas:** Son actividades, costos y activos relacionados con la producción de investigación y desarrollo, el proceso de investigación y desarrollo, proceso de mejora del diseño, diseño de equipo, desarrollo de software para computadoras, sistemas de telecomunicaciones, diseño asistido por computadora e ingeniería, nuevas capacidades de una base de datos y soporte a sistemas.
   - **Contrataciones:** Son actividades, costos y activos asociados con la compra y suministro de materias primas, suministros, servicios y *outsourcing* [PearceRobinson 2007] necesario para soportar el negocio y sus actividades.

De la cadena de valores podemos observar que el desarrollo de software se encuentra dentro de las actividades de soporte, por lo tanto, todo negocio que tenga un enfoque orientado al cliente, requiere de actividades de soporte que también estén orientadas al cliente. En el desarrollo de software sólo existen dos enfoques de desarrollo, el dirigido por un plan o tradicionales que están orientados al producto y los métodos ágiles o ligeros, que están orientados al cliente.

Él modelo MCAA se enfoca en los métodos orientados al cliente y el soporte que las prácticas arquitectónicas les brinda para el cumplimiento de las necesidades del cliente y su negocio.

El valor debe ser definido perfectamente desde la perspectiva del cliente y no sólo atendiendo a los intereses del negocio, por lo que la creación de valor está determinada por una activa participación del cliente [Donadio 2004].

Por lo tanto, la capa conceptual ágil y la capa conceptual de arquitectura deberán proveer actividades que permitan dar soporte al cumplimiento de las condiciones anteriormente descritas.

5.6 Capa conceptual ágil

5.6.1 Objetivo

La capa conceptual ágil está enfocada en aquellas actividades, productos de trabajo, valores y principios que dan soporte a la capa conceptual del cliente para satisfacer sus necesidades.

Esta capa tiene los siguientes objetivos:

- Dar cumplimiento y soporte a las necesidades de la capa conceptual del cliente.
- Establecer los valores y principios que rigen al proceso de desarrollo ágil; quien es el responsable de satisfacer al cliente mediante entregas tempranas, rápidas y con valor para su negocio.
- Establecer las actividades, productos de trabajo y roles ágiles que dan soporte a la capa conceptual del cliente.
- Capturar las necesidades del cliente, lo que el sistema debe hacer; el servicio que ofrece y las restricciones de su operación.

5.6.2 Valores ágiles

Esta subcapa da soporte a las siguientes capas/subcapas explicadas en la Tabla 40:

Capa	Subcapas	Soporte
Capa conceptual del cliente	Todas	Mediante el seguimiento de los cuatro valores ágiles durante un proceso de desarrollo, es posible manejar los requerimientos fluctuantes y la entrega inmediata de software con valor para los negocios que operan en ambientes globalizados con mercados abiertos.

Tabla 40 Soporte para la capa conceptual del cliente

Los valores ágiles son los expuestos en [ManifiestoAgil]:

1. Individuos e interacciones sobre procesos y herramientas.
2. Software funcionando sobre documentación extensiva.
3. Colaboración con el cliente sobre negociación contractual.
4. Respuesta ante el cambio sobre seguir un plan.

5.6.3 Principios Ágiles

Esta subcapa da soporte a las siguientes capas/subcapas explicadas en la Tabla 41.

Capa	Subcapas	Soporte
Capa conceptual del cliente	Todas	Al apoyar en la comprensión de los cuatro valores ágiles descritos en la subcapa “Valores ágiles”, ayuda indirectamente a la capa conceptual del cliente.
Capa conceptual ágil	Valores ágiles	Los cuatro valores ágiles pueden llevarse a la práctica aplicando los principios agiles. Los principios agiles ayudan a entender mejor la esencia de los valores del manifiesto ágil.

Tabla 41 Soporte para la capa conceptual del cliente y capa conceptual ágil

A continuación se nombran los 13 principios ágiles [ManifiestoAgil]:

1. Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
2. Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
3. Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
4. Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
5. Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
7. El software funcionando es la medida principal de progreso.
8. Los procesos Ágiles promueven el desarrollo sostenible.
9. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
10. La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
11. La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
12. Las mejores arquitecturas, requisitos y diseños emergen de equipos auto organizados.
13. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

Los principios ágiles se resumen en la Tabla 42 [Sommerville 2011]:

Principio	Descripción
Participación del cliente	Los clientes deben intervenir estrechamente durante el proceso de desarrollo. Su función consiste en ofrecer y priorizar nuevos requerimientos del sistema y evaluar las iteraciones del mismo.
Entrega incremental	El software se desarrolla en incrementos y el cliente especifica los requerimientos que se van a incluir en cada incremento.
Personas, no procesos	Tienen que reconocerse y aprovecharse las habilidades del equipo de desarrollo. Debe permitirse a los miembros del equipo desarrollar sus propias formas de trabajar sin procesos establecidos.
Adoptar el cambio	Esperar a que cambien los requerimientos del sistema, de este modo, diseñar el sistema para adoptar dichos cambios.
Mantener simplicidad	Enfocarse en la simplicidad tanto en el software a desarrollar como en el proceso de desarrollo. Siempre que sea posible, trabajar de manera activa para eliminar la complejidad del sistema.

Tabla 42 Resumen de los principios ágiles

5.6.4 Gestión ágil

Está subcapa se enfoca en las actividades y productos de trabajo que dan soporte a la capa conceptual del cliente. A continuación se detalla las capas/subcapas a la que esta subcapa brinda soporte:

Capa	Subcapas	Soporte
Capa conceptual del cliente	– Todas	En esta capa se establecen las actividades que permiten capturar, gestionar y revisar las necesidades del cliente en términos del valor del negocio, para que de esta manera se de soporte a los objetivos de la capa conceptual del cliente.
Capa conceptual ágil	– Valores ágiles. – Principios ágiles.	Las actividades descritas aquí son basadas en los principios y valores ágiles. Principalmente son las actividades de gestión descritas para *Scrum*, en las cuales el cliente está involucrado, y por lo tanto, se enfatiza en la entrega de valor para el negocio.

Tabla 43 Soporte para la capa conceptual del cliente y la capa conceptual ágil

La Tabla 44 muestra los roles involucrados en la capa conceptual ágil.

Rol	Abreviación
Product owner	PO
Scrum master	SM
Scrum team	ST

Tabla 44 Roles involucrados

A continuación se describen las actividades, productos de trabajo y roles involucrado[8].

5.6.4.1 Actividades

Las actividades identificadas para esta capa se muestran en la Tabla 45 y Tabla 46:

Actividad	Descripción
Crear Visión del sistema	Un proyecto de *Scrum* comienza con el desarrollo de la visión del sistema. El PO es responsable de encontrar el financiamiento del proyecto para hacer que se obtenga lo que indica la visión del sistema, de tal manera que se maximice el retorno de inversión (ROI). El PO crea y prioriza el Product Backlog. Los cambios en el Product Backlog reflejan los requerimientos cambiantes del negocio y qué tan rápido o tan lento el ST puede transformarlos en funcionalidad [RedmondWashington]. Para construir el *product backlog* el PO elabora las historias de usuario.
Release planning meeting	El propósito de la planificación de la entrega es establecer un plan y unas metas que los Equipos *Scrum* y el resto de las organizaciones puedan entender y comunicar. La planificación de la entrega responde a las preguntas: "¿Cómo podemos convertir la visión en un producto ganador, de la mejor manera posible? ¿Cómo podemos alcanzar o mejorar la satisfacción del cliente deseada y el Retorno de la Inversión?". Esta planificación de entrega por lo general no requiere más de un 15-20% del tiempo consumido por una organización para construir un plan de entrega tradicional. [ScrumOrg]
Sprint planning meeting	Es una reunión que se realiza al inicio de cada Sprint y tiene una duración de 8 horas. La reunión se divide en dos segmentos de 4 horas cada uno. Durante el primer segmento, el PO presenta la lista de requerimientos (product backlog) al equipo, con la lista de tareas priorizadas. El equipo y el product owner colaboran para determinar la cantidad del product backlog que se puede convertir en funcionalidad durante el próximo Sprint. Al final del primer segmento el equipo se compromete a este product backlog. Durante el segundo segmento de la reunión, el equipo hace planes de cómo va a cumplir con este compromiso, detallando su labor en el *sprint backlog* [RedmondWashington]. La reunión se restringe a un bloque de tiempo de ocho horas para un Sprint de un mes. Para Sprints más cortos, se debería reservar para esta reunión un tiempo proporcionalmente menor, aproximadamente el 5% de la longitud total del Sprint (por ejemplo, para un Sprint de dos semanas sería una Reunión de Planificación de cuatro horas). [ScrumOrg]

Tabla 45 Actividades para la gestión ágil (1/2)

[8] Debido a que las prácticas descritas aquí son tomadas de *Scrum*, no se discutirá su integración dentro de métodos ágiles. Sin embargo, para la capa conceptual de arquitectura, se hará hincapié en el soporte que sus actividades brindan en el desarrollo ágil.

Actividad	Descripción
Sprint	Son 30 días consecutivos, durante el cual el ST trabaja para convertir el *product backlog* elegido en un incremento de la funcionalidad del producto final [RedmondWashington].
Daily Scrum meeting	Es una reunión corta, de 15 minutos, que se lleva a cabo diario. En ella, el *Scrum team* sincroniza su trabajo y su progreso, además de reportar cualquier impedimento al *Scrum master* [RedmondWashington].
Sprint Review Meeting	Es una reunión de 4 horas y ocurre al final de cada Sprint. El equipo muestra al PO y a otros interesados en el proyecto, lo que fue posible alcanzar durante el Sprint. Sólo se muestra el incremento de funcionalidad del producto que se encuentra finalizado [RedmondWashington]. Esta es una reunión restringida a un bloque de tiempo de cuatro horas para un Sprint de un mes. Para Sprints de menor duración, hay queasignar proporcionalmente menos tiempo de la longitud total para esta reunión (por ejemplo, para dos semanas, la Revisión del Sprint sería de dos horas); esta reunión no debe consumir más de 5% del total del Sprint. [ScrumOrg]
Sprint retrospective meeting	Es una reunión de tres horas al final de cada iteración y dirigida por el *Scrum master*, en la cual, el *Scrum team* discute el *sprint* concluido y determinan que podría ser modificado para que el siguiente *sprint* sea más productivo o agradable [RedmondWashington]. Es una reunión restringida a un bloque de tiempo de tres horas para *sprints* de un mes (asignar tiempo proporcionalmente menor para *sprints* de longitud menor). [ScrumOrg].

Tabla 46 Actividades para la gestión ágil (1/2)

5.6.4.2 Productos de trabajo

Los productos de trabajo identificados para esta capa se muestran en la Tabla 47.

Producto de trabajo	Descripción
Product Backlog	El Product Backlog es una lista de requerimientos funcionales y no funcionales que, cuando se conviertan en funcionalidad, cumplirán con la visión del sistema [RedmondWashington]. Los elementos del product backlog son las historias de usuario descritas por el PO.
Sprint Backlog	Es una lista de tareas que definen el trabajo de un equipo para un Sprint. La lista surge durante el Sprint. Cada tarea identifica a los responsables que realizarán el trabajo y el monto del tiempo estimado para concluir dichas tareas. [RedmondWashington]
Release Burdown	El gráfico de *Burndown* de la Entrega, registra la suma del esfuerzo restante estimado del *Product Backlog* a lo largo del tiempo. El esfuerzo se estima en cualquier unidad de trabajo que el Equipo *Scrum*, y la organización, hayan decidido. La unidad de tiempo que se utiliza generalmente es el Sprint. [ScrumOrg]
Sprint Burdown	El *sprint burdown* es una gráfica que muestra la cantidad de trabajo restante por hacer del *sprint backlog* a lo largo de un *sprint*. [ScrumOrg]
Incremento	Es la funcionalidad del producto que es desarrollado por el ST durante cada sprint. [RedmondWashington]
Historias de usuario	Es la descripción de lo que el cliente quiere que el sistema haga. Las historias de usuario tienen el propósito similar al de los casos de uso, pero no son lo mismo. Las historias de usuario son usadas para hacer estimaciones de tiempo durante la planeación de una entregar. También pueden ser utilizadas en lugar de la documentación de requerimientos muy grandes. Las historias de usuario son escritas por el PO y representan algo que el sistema necesita para ellos [BeckKent], [DonWells].

Tabla 47 Productos de trabajo

5.6.4.3 Relación entre actividades, productos de trabajo y roles involucrados

La Tabla 48 muestra las actividades anteriormente expuestas con sus respectivos productos de trabajo.

Producto de trabajo(entrada)	Actividad	Producto de trabajo(Salida)	Rol
	Crear visión del sistema	Product backlog Historias de usuario	PO, ST, SM
Product backlog Historias de usuario	**Release planning meeting**	Plan general	PO, SM
Product backlog	**Sprint Planning Meeting**	Sprint backlog	PO, ST, SM
Sprint backlog	**Sprint**	Incremento	ST,SM
Sprint Burdown	**Daily Scrum meeting**	Sprint backlog Sprint Burdown	ST,SM
Sprint backlog Incremento	**Sprint Review Meeting**	Release Burdown	PO, ST, SM
	Sprint retrospective meeting		PO, ST, SM

Tabla 48 Actividades, productos de trabajo y roles

La Figura 5-4 Flujo de actividades para la gestión ágilFigura 5-4 exponen las principales actividades y productos de trabajo de *Scrum* [ScrumOrg], y por lo tanto, las principales actividades y productos de trabajo de la capa conceptual ágil para la gestión del desarrollo de software.

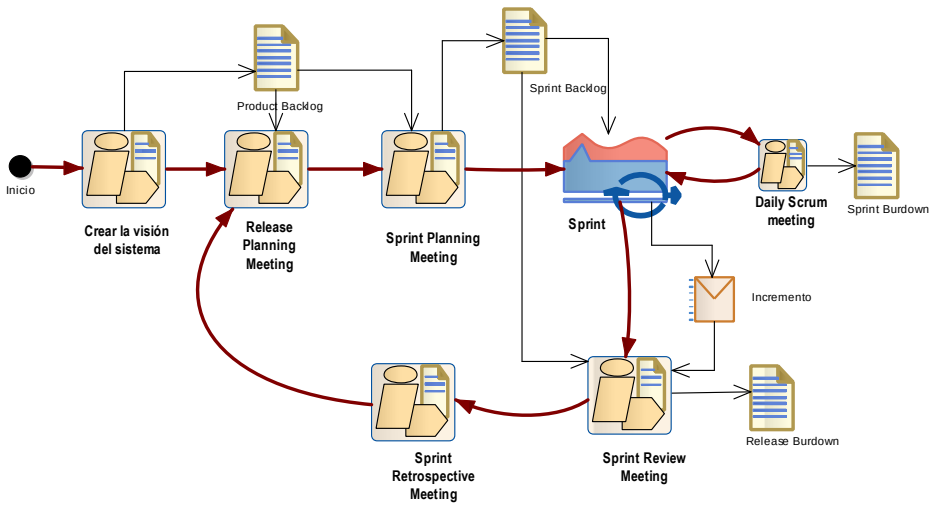

Figura 5-4 Flujo de actividades para la gestión ágil

5.6.5 Prácticas ágiles de desarrollo de software.

La subcapa de "prácticas ágiles de desarrollo", se enfocan en el diseño y construcción del incremento de software. Para esta subcapa se utilizan algunas de las prácticas definidas dentro de *Extreme Programming* (XP). Las prácticas empleadas son las definidas en la Tabla 49 y Tabla 50.

Práctica	Descripción
Metáfora del sistema	Los equipos XP desarrollan una visión común sobre cómo funciona el programa, que llamamos la "metáfora". La metáfora es una descripción evocativa simple sobre cómo funciona el programa. [XPRonaldJeffries]
Diseño simple	Los equipos de XP construyen software sobre un diseño simple. Empiezan simple, y a través de las pruebas de programación y las mejoras al diseño, lo mantienen así. Cualquier equipo XP mantiene al diseño para que sea el justo y necesario para cumplir la funcionalidad actual del sistema. No hay desperdicio, y el software siempre está listo para lo que sigue. [XPRonaldJeffries]
Refactorización	*Extreme Programming* se enfoca en entregas por cada iteración. Para lograr esto a lo largo de todo el proyecto, el software debe estar bien diseñado. La alternativa es retrasarse hasta detenerse por completo. Es por esto que XP utiliza un proceso de mejora continua del diseño llamado *Refactoring*, sacado del libro de Fowler "Refactoring: Improving the Design of Existing Code". El proceso de *refactoring* se enfoca en eliminar la duplicación (una clara señal de diseño pobre) y en incrementar la cohesión del código, disminuyendo el acoplamiento. [XPRonaldJeffries]
Programación en pares	En XP todo el software productivo se escribe en pareja, dos programadores sentados lado a lado en una misma computadora. Esta práctica asegura que todo el código productivo fue revisado por al menos otro programador, y genera mejores diseños, mejores pruebas y mejor código. [XPRonaldJeffries]

Tabla 49 Prácticas para el desarrollo de software (1/2)

Práctica	Descripción
Propiedad colectiva	En un proyecto XP, cualquier pareja de programadores puede mejorar cualquier porción de código en cualquier momento. Todos los programadores se responsabilizan por el código, cualquiera puede cambiar cualquier función. [XPRonaldJeffries]
Estándares de codificación	Los equipos XP usan un estándar de código en común, de manera que el código del sistema se vea como si fuera escrito por una única persona muy competente. No importa mucho el estándar en sí mismo: lo importante es que el código se vea familiar, para permitir la propiedad colectiva. [XPRonaldJeffries]
Integración continua	Los equipos mantienen integrado al sistema todo el tiempo**.** Los equipos XP realizan construcciones muchas veces por día. [XPRonaldJeffries]
TDD	El desarrollo dirigido por pruebas (TDD, por sus siglas de Test-Driven Development) es un enfoque al diseño de programas donde se entrelazan el desarrollo de pruebas y el de código. En esencia, el código se desarrolla incrementalmente, junto con una prueba para ese incremento. No se avanza hacia el siguiente incremento sino hasta que el código diseñado pasa la prueba. [Sommerville 2011]

Tabla 50 Prácticas para el desarrollo de software (2/2)

La Tabla 51 muestra algunos de los productos de trabajo que podría emplearse, no obstante la lista puede variar de acuerdo con las necesidades del proyecto.

Producto de trabajo	Descripción
Pruebas de aceptación del cliente	Las pruebas de aceptación son pruebas de caja negra [DonWells] para el sistema. Cada prueba de aceptación representa algún resultado del sistema. Son creadas durante la generación de las historias de usuario por el cliente, quien especifica los escenarios para los cuales se considerará que una historia de usuario estará implementada correctamente. [DonWells]
Pruebas unitarias	Las pruebas unitarias son el proceso de probar componentes del programa, como métodos o clases de objetos. Las funciones o los métodos individuales son el tipo más simple de componente. [Sommerville 2011]
Tarjetas CRC	Es una tarjeta estándar con índice que está dividida en tres secciones: la primera indica el nombre de la clase que la tarjeta representa, la segunda es una lista de responsabilidades de la clase y la última contiene los nombres de las otras clases con las que ésta colabora para cumplir con sus responsabilidades. [ScottAmbler]

Tabla 51 Productos de trabajo para el desarrollo de software

La Figura 5-5, una modificación de [DonWells], muestra la realización de las prácticas de XP, explicadas en la Tabla 49 y Tabla 50, que se realizan durante la etapa de desarrollo del software.

Éstas ocurren durante un sprint para conseguir el incremento de software que será entregado para él cliente.

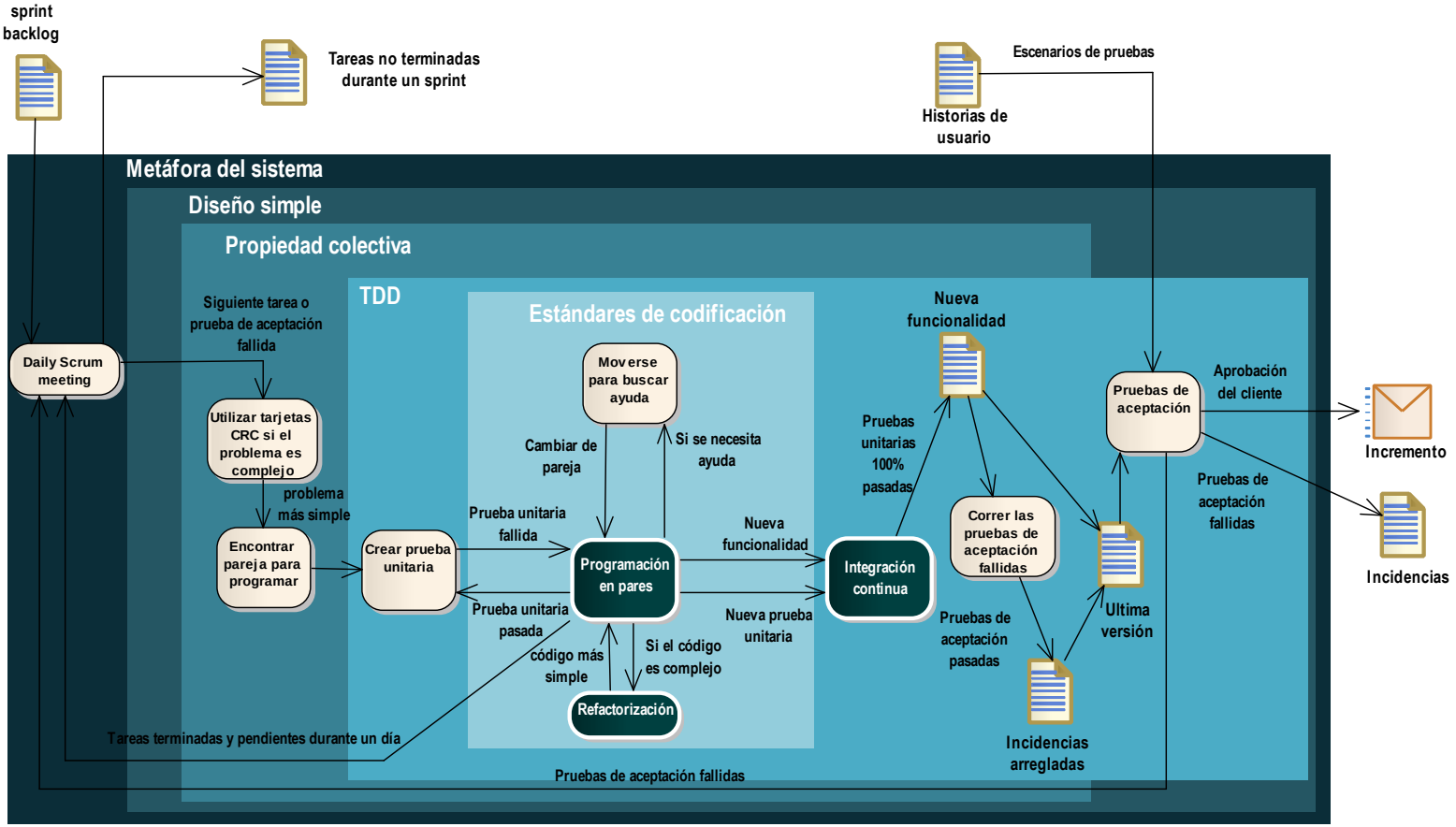

Figura 5-5 Desarrollo del software durante un sprint

Es importante mencionar que, hasta ahora, se ha mostrado una manera de integrar las prácticas de XP, dentro de *Scrum*. Sin embargo, estas se pueden variar de acuerdo con las necesidades del proyecto.

Lo que a continuación sigue es introducir las actividades arquitectónicas dentro de este MCAA.

5.7 Capa conceptual de arquitectura

5.7.1 Objetivo

La capa conceptual de arquitectura tiene por objetivo principal brindar soporte a la capa conceptual ágil, y está enfocada en aquellas actividades y productos de trabajo relacionados con las prácticas arquitectónicas.

Esta es la capa que se incorpora en la base de la pirámide del MCAA para dar soporte a la capa conceptual ágil. La capa de arquitectura, como se ha mencionado en el capítulo 4, no está definida dentro de procesos de desarrollo que siguen un enfoque ágil. Por lo que el MCAA aporta la incorporación de esta capa.

La capa conceptual de arquitectura tiene su razón de existir debido a la necesidad de incorporar prácticas de análisis, diseño y evaluación de arquitecturas dentro de procesos de desarrollo de software dirigidos por un enfoque ágil (ver capitulo 4), [Falessi], [Babar2009], [Madison 00]. La incorporación de las prácticas arquitectónicas viene inherente al contexto del proyecto; por lo que es importante considerar una versión ágil de las mismas para aplicarlas en ambientes ágiles [Farhan], [Abrahamsson], [Nord].

Es importante mencionar que la capa conceptual de arquitectura no viene a sustituir las prácticas de desarrollo ágiles existentes (como las expuestas en [BeckKent]), más bien, viene a trabajar en conjunto con ellas, ayudar a mantener los atributos de calidad del sistema en desarrollo y dar soporte al cumplimiento de los valores y principios ágiles. Lo anterior se puede apreciar en la Figura 5-6.

Figura 5-6 Soporte de la capa conceptual de arquitectura a las actividades de gestion y desarrollo ágil

La capa conceptual de arquitectura está conformada por las siguientes subcapas:

- Arquitecto
- Arquitectura
- Diseño arquitectónico

Cada una de las subcapas, mencionadas anteriormente, le agregan un valor y soporte a las actividades y prácticas ágiles definidas en la "capa conceptual ágil", por lo tanto, para denotar el valor y soporte que la capa conceptual de arquitectura aporta a las actividades/prácticas ágiles, se agregará el símbolo  a éstas.

5.7.2 Arquitecto

La subcapa "Arquitecto", es la encarga de definir las habilidades que debe cubrir un arquitecto ágil, la manera en que debe apoyar para asegurar el cumplimiento de los valores y principios ágiles y su definición dentro de las actividades de gestión y desarrollo.

El arquitecto dentro de un proceso de desarrollo ágil de software, deberá actuar como un proveedor de servicios, tanto para los desarrolladores de la aplicación como para los clientes [Faber]; además de ser un buen comunicador. Para una comunicación intensiva, el arquitecto deberá participar muy de cerca en el trabajo de desarrollo de la aplicación, incluyendo programar. Lo anterior dependerá del contexto del sistema [Faber].

La Tabla 52, Tabla 53, Tabla 54 y Tabla 55 enuncia las características que deberá cumplir un arquitecto ágil (AA), las cuales son similares a las características que debe cumplir un arquitecto tradicional [Peter Eels], excepto en ciertos detalles que se explican en la columna de "Agregado ágil". Por lo tanto, las características con las que debe de contar el arquitecto ágil son:

Arquitecto ágil = Arquitecto tradicional + Agregado ágil.

Características	Arquitecto ágil	
	Arquitecto tradicional	Agregado ágil
El papel del arquitecto puede ser realizado por un equipo	El rol del arquitecto puede estar conformado no sólo por una persona, sino, por un equipo que cumplan en conjunto con las características que se presentan aquí.	El *Scrum team*, arquitecto ágil y el *Scrum master* pueden ser los encargados de toda la arquitectura.
El arquitecto es un líder técnico	El arquitecto es un líder técnico, lo que significa que, además de tener conocimientos técnicos, el arquitecto exhibe cualidades de liderazgo. El liderazgo puede ser caracterizado tanto en términos de la posición en la organización (es el jefe técnico) como en las cualidades que el arquitecto exhibe.	Como líder técnico el arquitecto ágil debe ser un mentor, modelador y solucionador de problemas, concentrándose en el código y la coordinación interna del equipo de desarrollo [Abrahamsson]. Además, deberá crear una cierta atmosfera de confianza y motivación para el éxito del proyecto. [Faber]
El arquitecto entiende el proceso de desarrollo de software	El arquitecto debe tener una apreciación de todo el proceso de desarrollo de software, ya que este proceso garantiza que todos los miembros del equipo trabajen de manera coordinada. Un buen proceso define los roles involucrados, las actividades realizadas y los productos de trabajo creados. Dado que el arquitecto está involucrado diario con los miembros del equipo, es importante que el arquitecto entienda sus funciones y responsabilidades para darles guía cuando sea necesario.	Debe tener un conjunto de capacidades y conocimientos sobre el proceso de desarrollo ágil de software; cuando escucha atentamente y cuando provee guía al equipo. Además, debe tomar diferentes roles para interactuar con los desarrolladores y otros *stakeholders*. Por un lado, deben comprender profundamente los requerimientos de calidad del sistema. Por otro lado, debe guiar el desarrollo al balance correcto entre los atributos de calidad. Dependiendo la situación, el arquitecto ágil escucha cuidadosamente o dirige firmemente las actividades del equipo [Faber].

Tabla 52 Características del arquitecto ágil (1/4)

Características	Arquitecto ágil	
	Arquitecto tradicional	Agregado ágil
El arquitecto tiene un conocimiento del dominio del negocio	Además de tener una comprensión del desarrollo de software, también es muy conveniente que el arquitecto tenga una comprensión del dominio[9] del negocio.	Al tener el conocimiento sobre el dominio del negocio, el arquitecto ágil debe ser capaz de dibujar requerimientos del negocio en la forma de *storyboard*[10], explicar las restricciones técnicas para el negocio y reafirmar las necesidades del negocio en términos técnicos para el equipo [Madison 00]. Además, como proveedor de servicios, el arquitecto ágil debe darse cuenta que la arquitectura no es final ni estática, sino un objeto de cambio como la comprensión del sistema que se incrementa durante su desarrollo. Para mantenerse al día, el arquitecto debe actualizar continuamente sus conocimientos sobre el dominio de aplicación [Faber].
El arquitecto tiene conocimientos tecnológicos	Ciertos aspectos de la arquitectura claramente requieren un conocimiento de la tecnología, un arquitecto, por lo tanto, debe mantener un cierto nivel de habilidades tecnológicas. Debido a que la tecnología cambia rápidamente, es necesario que el arquitecto se mantenga al tanto de esos cambios.	No se agrega característica

Tabla 53 Características del arquitecto ágil (2/4)

[9] Un dominio es un área de conocimiento o actividad caracterizada por un conjunto de conceptos entendibles por los profesionales en esa área. (UML User Guide 1999)

[10] Él *storyboard* o guión gráfico, en el campo del desarrollo del software, es una técnica utilizada para presentar y describir, eventos interactivos con el usuario, particularmente, en las interfaces de usuario y páginas electrónicas (http://en.wikipedia.org/wiki/Storyboard).

Características	Arquitecto ágil	
	Arquitecto tradicional	Agregado ágil
El arquitecto tiene conocimientos de diseño	El arquitecto debe poseer fuertes habilidades de diseño, ya que la arquitectura contempla las principales decisiones de diseño. Tales decisiones representarán las principales decisiones sobre el diseño de toda la estructura del sistema, entre estas decisiones de diseño encontramos: la elección de un patrón particular, una táctica arquitectónica, etc.	El arquitecto ágil debe saber crear el *Backlog* de la arquitectura [Madison 00] y asegurar que la refactorización no tenga efectos negativos en la arquitectura del sistema [Babar2009].
El arquitecto tiene conocimientos de programación	Los desarrolladores de un proyecto representan uno de los grupos más importantes con los que el arquitecto deberá interactuar. Después de todo, ellos producen el software final. La comunicación entre el arquitecto y los desarrolladores puede ser efectiva sólo si el arquitecto es considerado importante dentro del trabajo de los desarrolladores. Por lo tanto, el arquitecto debe tener un cierto nivel de habilidades para programar.	El arquitecto ágil, al ser el stakeholder de la calidad del sistema, debe actuar como maestro programador, quien se asegurará de que la arquitectura del sistema sea entendida por los desarrolladores y la calidad de todo el código del sistema sea preservada. [Faber]
El arquitecto es un buen comunicador	La comunicación es una de las habilidades más importantes del arquitecto. El arquitecto tiene que tener habilidades para hablar, escribir y exponer. El arquitecto debe ser un buen oyente y observador, así como un buen orador.	El arquitecto debe estimular el intercambio con sus compañeros, para que de esta manera se obtenga una rápida retroalimentación [Faber].

Tabla 54 Características del arquitecto ágil (3/4)

Características	Arquitecto ágil	
	Arquitecto tradicional	Agregado ágil
El arquitecto toma decisiones	Un arquitecto que es incapaz de tomar decisiones en un entorno desconocido, donde no hay tiempo suficiente para explorar todas las alternativas posibles y donde existe una presión para entregar el trabajo deseado, es poco probable que tenga éxito. Los arquitectos exitosos reconocen la situación, en lugar de tratar de cambiarla. La incapacidad para tomar decisiones poco a poco socavara el proyecto y el equipo del proyecto perderá la confianza en el arquitecto.	El arquitecto ágil debe ser el creador y poseedor de las grandes decisiones, centrándose en la coordinación externa [Abrahamsson] y ser el responsable de decidir sobre el tiempo y cantidad de la refactorización que se realizará [Babar2009]. Además el arquitecto ágil deberá tomar decisiones sobre la forma de descomponer el trabajo en formas manejables para cada sprint (*sprintable form)* [Madison 00].
El arquitecto es consciente de las políticas de la organización	El arquitecto de éxito no es sólo técnico. También, son políticamente astutos y conscientes de que el poder reside en una organización. Este conocimiento se utiliza para asegurar que las personas adecuadas se comunican con él.	No se agrega característica
El arquitecto es un negociador	Dadas las muchas dimensiones del diseño de la arquitectura, el arquitecto interactúa con muchos de los *stakeholders* del proyecto. Algunas de estas interacciones requieren de habilidades para la negociación. Por ejemplo, el arquitecto tiene que minimizar el riesgo tan pronto como sea posible en el proyecto, una manera de remover un riesgo es volver a redefinir los requerimientos, de tal manera que los *stakeholders* y el arquitecto lleguen a un acuerdo mutuo. Esta situación requiere que el arquitecto sea un negociador efectivo.	El arquitecto ágil, como negociador, deberá saber apoyar al *Product Owner* en la priorización de las historias de usuario durante la creación del *product backlog* [Madison 00]. Durante la creación del *product backlog* el arquitecto ágil deberá expresarle al *product owner* la importancia de ciertas tareas que son relevantes para estabilizar la arquitectura del sistema, sobre algunas prioridades del negocio, y que esta prioridad más adelante permitirá reducir tiempos de refactorización originando que las entregas puedan ser más rápidas.

Tabla 55 Características del arquitecto ágil (4/4)

5.7.3 Arquitectura

La subcapa "Arquitectura", explican los usos de la arquitectura de software y su relevancia para el desarrollo ágil [Falessi], (ISO/IEC 42010). La Tabla 56 presentan los usos de la arquitectura de software que fueron considerados relevantes dentro de las actividades de desarrollo ágil de acuerdo con la encuesta realizada en [Falessi] y el soporte que brinda a la capa conceptual ágil.

Categoría	Usos y soporte de la arquitectura de software para la capa conceptual ágil ISO/IEC 42010
Comunicación	Para la comunicación entre las organizaciones involucradas en el desarrollo, producción, operación y mantenimiento del sistema.
	Para comunicar las características y diseños de un sistema para los clientes potenciales.
	Para la comunicación entre los clientes, los compradores y desarrolladores como parte de las negociaciones del contrato.
Diseño y análisis de la arquitectura	Como entrada para subsecuentes diseños de sistemas y actividades de desarrollo.
	Para analizar y evaluar diferentes arquitecturas.
	Para dar soporte a la revisión, análisis y evaluación del sistema.
Documentación	Para documentar supuestos hechos por el arquitecto acerca del sistema y la intensión de su uso y ambiente.
	Para documentar puntos de flexibilidad o limitaciones dentro del sistema para futuros requerimientos.
	Como documentación para el desarrollo y mantenimiento.
Ampliación de los métodos ágiles	Para apoyar la ampliación de las prácticas ágiles a los grandes proyectos.
Mantenimiento	Para ayudar a la planificación de la transición de una arquitectura de software heredada a una nueva arquitectura de software.
	Como la especificación para un grupo de sistemas que comparten un conjunto de características (ejemplo, líneas de productos).
	Para soporte operacional y de infraestructura; administración de la configuración y reparación; re diseño y mantenimiento de un sistema, subsistemas y sus componentes
	Para establecer criterios para las implementaciones de certificación para la conformidad de la arquitectura de software.

Tabla 56 Usos de la arquitectura de software dentro de un proceso de desarrollo ágil.

5.7.4 Diseño arquitectónico

Las prácticas arquitectónicas conforman el corazón de la capa conceptual de arquitectura, en ellas se encuentran definidas las actividades que ayudan al análisis, diseño, evaluación y evolución de la arquitectura. Esta sección se enfoca, principalmente, en las prácticas arquitectónicas definidas en el [SEI2010].

En equipos ágiles, el diseño que emerge es un producto proveniente de las historias de usuario. Pero la arquitectura depende, para su forma y calidad, de la experiencia del equipo de desarrollo. Las actividades centradas en la arquitectura, pueden informar y regularizar el proceso de desarrollo, enfatizando en los atributos de calidad y enfocándose de manera temprana en las decisiones arquitectónica. En situaciones donde los requerimientos cambian rápidamente y se ha garantizado un enfoque ágil, los conceptos arquitectónicos pueden mejorar el proceso de diseño de un sistema que conocerá sus requerimientos de manera temprana [Nord].

Por otro lado, los desarrolladores en XP aumentan el sistema incrementalmente. Cuando un sistema no soporta la nueva funcionalidad, entonces se hace una refactorización en el diseño. Por lo tanto, el primer sprint juega un rol crucial en la definición de toda la estructura de un sistema, ya sea implícitamente (la arquitectura emerge después de implementar la primera ronda de historias de usuario) o explícitamente [Nord]. Dentro de este marco conceptual se define el sprint 0, en el cual se dedican los esfuerzos para el diseño de la arquitectura.

Las actividades arquitectónicas, expuestas aquí, se muestran agrupadas en dos categorías: actividades para la obtención de requerimientos arquitectónicos y actividades para el diseño arquitectónico detallado.

En lo que resta de esta sección, se dedican los esfuerzos en la adaptación de estas actividades dentro del MCAA. Más tarde, se presentará el proceso descrito en SPEM 2.0, mismo que sigue todas las actividades presentadas en el MCAA.

5.7.4.1 Obtención de los requerimientos arquitectónicos

En esta categoría se encuentran las actividades para la obtención de los requerimientos arquitectónicos (los atributos de calidad, restricciones técnicas y del negocio) y elementos para el diseño y evaluación de la arquitectura de software. La Figura 5-7 muestra el soporte que brindan los requerimientos arquitectónicos para la obtención de la estructura arquitectónica.

Figura 5-7 Requerimientos arquitectónicos como la base fundamental para el diseño de la estructura arquitectónica.

Los requerimientos arquitectónicos son los encargados de moldear la estructura del sistema de software que darán soporte a las necesidades del negocio. Esta idea se muestra en la Figura 5-8.

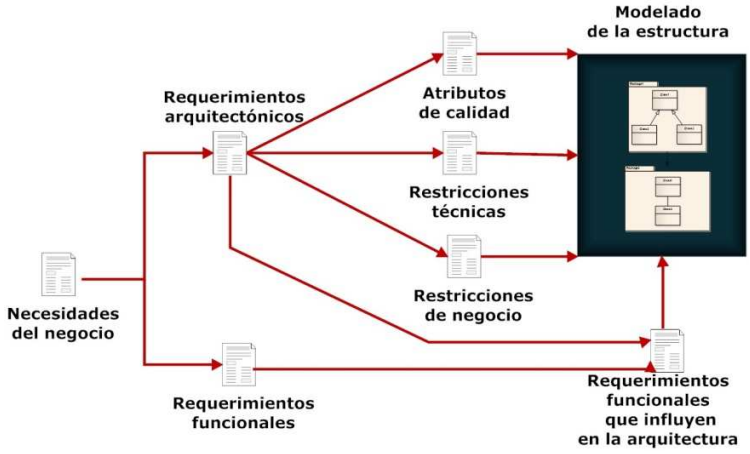

Figura 5-8 Modelado de la estructura del sistema

La Tabla 57 describe las Características del QAW, dentro del ciclo de desarrollo ágil, para la captura de los requerimientos arquitectónicos.

QAW (Quality Attribute Workshop)	
Descripción	El taller de atributos de calidad (QAW) es un método sencillo que involucra a los *stakeholders* del sistema desde una etapa temprana en el ciclo de vida del proyecto, con la finalidad de descubrir oportunamente los atributos de calidad del sistema de software. El QAW fue desarrollado para complementar el método de análisis de *tradeoff* de la arquitectura (ATAM) y proveer una manera de identificar importantes atributos de calidad y clarificar los requerimientos del sistema antes de crear la arquitectura del sistema. [QAW]
Cuando en el ciclo de desarrollo ágil	El QAW, se realiza a principios del desarrollo del proceso ágil, durante la producción de historias de usuario, ayudando a mostrar los requerimientos de atributos de calidad en la forma de escenarios. Por lo que es apropiado antes de la iteración cero de un proyecto ágil [Babar2009], [Kanwal]. En las iteraciones subsecuentes, los desarrolladores pueden colaborar con el cliente para obtener y afinar los escenarios adicionales que se vallan a requerir [Nord]. El QAW es similar al FAW (taller de análisis de características) propuesto en [Babar2009] para la iteración cero en un proyecto ágil.
Ventajas y soporte para la capa conceptual ágil	Las ventajas son [Nord]: – El QAW compromete a los stakeholders del sistema de manera temprana en el ciclo para encontrar los requerimientos de calidad que manejaran la construcción del sistema. – El QAW está basado en el sistema, enfocándose en los *stakeholders* y se hace antes de la creación de la arquitectura del software. – Los stakeholders tienden a enfocarse en la funcionalidad y no en los atributos de calidad. Los escenarios de los atributos de calidad son muy similares a las historias de usuario, con la diferencia de que los primeros permiten considerar a los atributos de calidad además de la funcionalidad. – También, este método mejora la planeación y el proceso de la generación de historias de usuarios, enfatizando en las metas del negocio, stakeholders y el papel de los atributos de calidad en el diseño de la arquitectura. – Las metas del negocio son obtenidas y refinadas durante el QAW. – Los escenarios pueden ayudar a determinar lo que se encuentra fuera o dentro del alcance del sistema y puede dirigir a la creación o refinamiento del diagrama del contexto del sistema o su equivalente. Además, la generación de los escenarios ayuda a crear los casos de uso. – El QAW ayuda a los *stakeholders* a descubrir y priorizar los atributos de calidad. – Los escenarios pueden ayudar al cliente para preparar las pruebas de aceptación que crecerán con el producto. – Muchos de los clientes no saben cómo construir los casos de pruebas. El escenario de atributos de calidad puede dar información sobre que probar, en caso de no haber recibido ayuda para construir estos casos de prueba.
Más información:	[PaulClements]

Tabla 57 Características del QAW dentro del ciclo de desarrollo ágil

La Tabla 58 muestra los roles involucrado en el método QAW.

Rol	Realizado por	Descripción
Líder QAW	SM	– Se asegura que los pasos del método QAW se lleven a cabo durante el taller. – Dirige y modera las discusiones – Se asegura de que el método sea realizado en tiempo y forma, y que se produzcan los artefactos requeridos del taller sean producidos.
Escritor QAW	AA	– Se encarga de capturar los principales escenario, su priorización, los escenarios refinados y cualquier asunto relevante que valla emergiendo durante el taller.

Tabla 58 Roles involucrados en el método QAW

La Tabla 59 es una adaptación de [RobertL] para mostrar el valor agregado que añade el QAW a las actividades específicas de: crear visión del sistema, sprint planning meeting y TDD.

Actividad/Práctica ágil +A	Valor agregado durante el QAW
Crear vision del sistema, *release planning meeting y sprint planning meeting* .	Las historias de usuario son complementadas con la información de los atributos de calidad. Estos son descritos en términos de escenarios, los cuales proveen un mayor detalle al ser descompuestos en sus seis partes.
Interacción con el *product owner*	Ayuda a la comunicación con el cliente, permitiendo analizar los requerimientos de manera temprana.
TDD	Los escenarios pueden ser utilizados para evaluar el diseño y proveer una entrada para el análisis durante las pruebas.

Tabla 59 Valor agregado por el QAW a la capa conceptual ágil.

Los elementos que participan el QAW y adaptados de [QAW], con base a [RobertL], se muestran en la Figura 5-9.

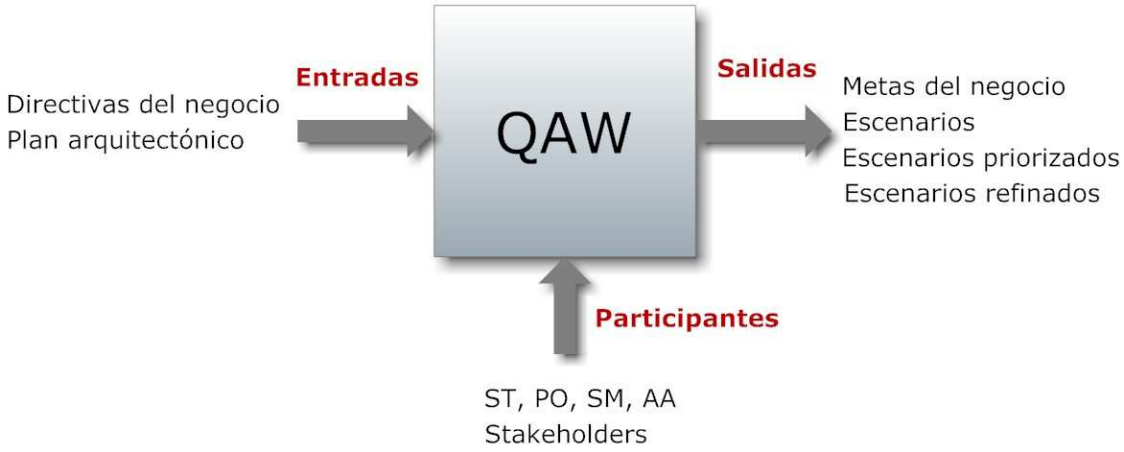

Figura 5-9 Elementos del QAW para el marco ágil

5.7.4.2 Diseño arquitectónico detallado

En esta categoría se encuentran las actividades que se aplican de manera iterativa para el diseño detallado y documentación de la arquitectura de software. Estas actividades terminan cuando quedan satisfechos los principales requerimientos de atributos de calidad.

Las actividades para el diseño detallado son: método ADD, ATAM y CBAM descritos a continuación.

En la Tabla 60 se describe el método ADD [Nord], [RobertL].

ADD (Atrribute-Driven Design)	
Descripción	El método ADD (*Attribute-Driven Design*) define a la arquitectura de software mediante un proceso de diseño; el cual está basado en escenarios de atributos de calidad priorizados que el software debe cumplir. El método ADD sigue un proceso de descomposición a lo largo de varias etapas. Durante cada etapa, se eligen las diferentes tácticas arquitectónicas y patrones, que satisfagan al conjunto de escenarios de atributos de calidad.
Cuando en el ciclo de desarrollo ágil	El método ADD puede y debería ser aplicado en los primeros sprints de un proyecto ágil Dentro de este marco conceptual, la aplicación del ADD ocurre en el Sprint 0.
Ventajas y soporte para la capa conceptual ágil	– Ayuda a identificar los asuntos más importantes que se deben de atender, con el fin de asegurarse de que el diseño está en camino de conocer los principales atributos de calidad y entregar valor para el cliente. – El método ADD se enfoca en lo que algunas veces los desarrolladores de XP ignoran; "la estructura general del sistema que los atributos de calidad forman". Esta atención debería ocurrir en las primeras iteraciones y recurrir a ella en iteraciones más adelante, como un cambio sustancial en la arquitectura del software.
Más información:	[PaulClements]

Tabla 60 Características del ADD dentro del ciclo de desarrollo ágil

La Tabla 61 es una adaptación de [RobertL] para mostrar el valor agregado que añade el método ADD a las actividades específicas de: crear visión del sistema, sprint planning meeting, metáfora del sistema, diseño simple y refactorización.

Actividad/Práctica ágil +A	Valor agregado por el método ADD
Crear 174ision del sistema , *release planning meeting y sprint planning meeting*	Se construye el árbol de utilidad para identificar las directrices arquitectónicas; las cuales son útiles de conocer en el momento de elegir la priorización de las historias de usuario.
Metáfora del sistema	El método ADD define la arquitectura de software paso a paso en términos de la descomposición de módulos, vistas de concurrencia y despliegue.
Diseño simple	El método ADD, proporciona una arquitectura suficiente para garantizar que el diseño cumple con los requerimientos de calidad. Lo anterior permite lidiar con cualquier riesgo que se pueda venir. Cualquier otra decisión arquitectónica, no es tomada en cuenta durante el método ADD.
Refactorización	La refactorización, influenciada por los atributos de calidad, es beneficiada por la elección de las tácticas arquitectónicas durante el método ADD.

Tabla 61 Valor agregado por el ADD a la capa conceptual ágil

Los elementos que participan en el método ADD y adaptados de [PaulClements], [Rob Wojcik] con base a [RobertL], se muestran en la Figura 5-10.

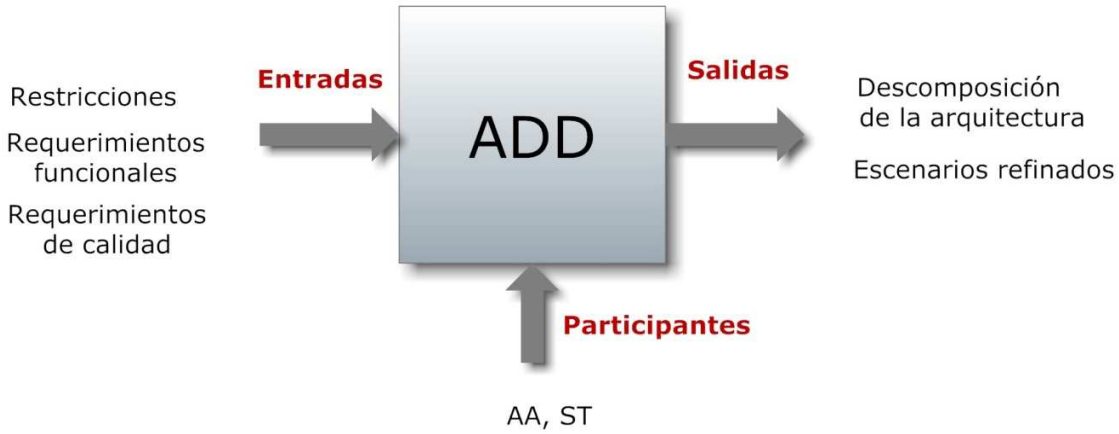

Figura 5-10 Elementos del método ADD para el marco ágil

Análisis temprano: ATAM y CBAM

Se puede utilizar el método ATAM y CBAM para hacer un seguimiento temprano de los riesgos técnicos y de negocio en el proceso y para ayudar a priorizar las historias de usuarios para la siguiente entrega. El método ATAM y CBAM ayudan a los practicantes de métodos ágiles a entender como las decisiones de diseño, que se hacen mientras se crea la arquitectura de un sistema complejo, interactuarán [Nord].

En la Tabla 62 se describen estos dos métodos.

<table>
<tr><th colspan="3">ATAM/CBAM
(Architecture Trade-off Analysis Method/Cost Benefit Analysis Method)</th></tr>
<tr><th colspan="2">ATAM</th><th>CBAM</th></tr>
<tr><td colspan="2">– El propósito del ATAM es la de evaluar las consecuencias de las decisiones de diseño a la luz de los requerimientos de atributos de calidad y las metas del negocio [RobertL], [Nord].
– El ATAM, provee a los desarrolladores un marco de trabajo para comprender los trade-offs y riesgos técnicos a los que se enfrentaran cuando se hagan las decisiones de diseño.
– El ATAM provee ayuda a los stakeholders para responder a las preguntas que ayuden a descubrir problemas potenciales sobre las decisiones arquitectónicas.
– Los riesgos descubiertos pueden ser el centro de atención de las actividades; por ejemplo, futuros diseño, futuros análisis y prototipos.</td><td>– El CBAM ayuda al AA a considerar el ROI (retorno de inversión) de cualquier decisión arquitectónica y proveer guía a los trade-offs económicos.
– El CBAM toma los análisis de decisiones arquitectónicas realizadas durante el ATAM y ayuda a hacerla parte del roadmap para el diseño y evolución del software mediante la asociación de prioridades, costos y beneficios con cada decisión arquitectónica.</td></tr>
<tr><td>Cuando en el ciclo de desarrollo ágil</td><td colspan="2">El ATAM/CBAM se puede hacer en diferentes etapas del ciclo de desarrollo ágil de acuerdo con las circunstancias del proyecto. A continuación se describen algunas de las circunstancias expuestas en [RobertL] de manera general y más tarde se especificarán las actividades ágiles que se ven involucradas:
– Se hace durante el sprint 0.
– Al final de cada Sprint, siempre que exista un análisis de las cualidades arquitectónicas para asegurar que la arquitectura este completa.
– Al final de cada Sprint si se quiere tener un control de daños cuando las cosas han ido mal.
– Finalmente, una evaluación puede realizarse antes de la entrega de software, en particular para hacer un inventario de activos reusables para un nuevo producto o para la evolución del software.</td></tr>
<tr><td>Ventajas y soporte para la capa conceptual ágil</td><td colspan="2">El ATAM (Architecture Trade-off Analysis Method) provee una guía detallada para el análisis del diseño y de esta manera obtener una retroalimentación temprana sobre el riesgo. [Nord]</td></tr>
<tr><td>Más información:</td><td colspan="2">[PaulClements]</td></tr>
</table>

Tabla 62 Características de ATAM/CBAM dentro del ciclo de desarrollo ágil

La Tabla 63 es una adaptación de [RobertL] para mostrar el valor agregado que añade el ATAM/CBAM a las actividades específicas de: crear visión del sistema, sprint planning meeting, interacción con el *product owner* y refactorización.

Actividad/Práctica ágil +A	Valor agregado por el método ATAM/CBAM
Crear vision del sistema , *release planning meeting y sprint planning meeting*	Las historias de usuario son complementadas con la información que proveen los atributos de calidad cuando son descompuestos en la forma de escenarios. Las estrategias arquitectónicas referentes al ROI (retorno de la inversión), los escenarios priorizados y refinados, contribuyen con información para el *product owner* y el arquitecto ágil durante la elección de las historias de usuario que serán implementadas en cada sprint. En otras palabras, los escenarios ayudan a la construcción del *sprint backlog*.
Interacción con el *product owner*	La comunicación con el *product owner* se mejora durante cada taller de evaluación.
Refactorización	El ATAM contribuye con los productos de trabajo necesarios para la comprensión del diseño antes de la refactorización. Por su lado, el CBAM provee información del costo que un cambio conllevaría al elegir determinada estrategia de refactorización con respecto al valor que trae para el cliente.

Tabla 63 Valor agregado por el ATAM/CBAM a la capa conceptual ágil

Los elementos que participan en el ATAM/CBAM con base a [RobertL], se muestran en la Figura 5-11 .

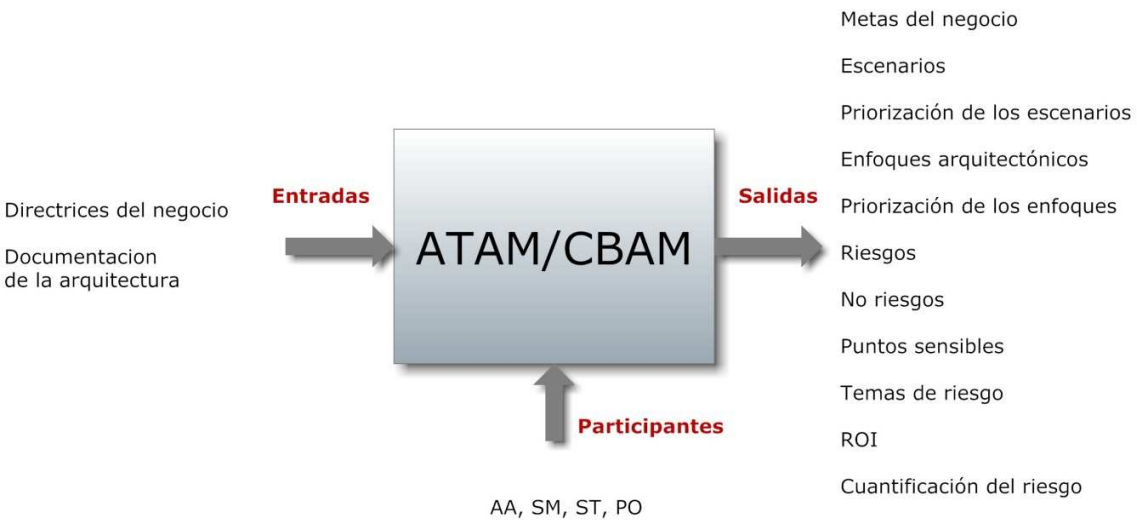

Figura 5-11 ATAM/CBAM para el marco ágil

Usar los métodos anteriores resulta en un enfoque centrado en la arquitectura: La arquitectura conecta las metas del negocio para la implementación, los atributos de calidad del diseño y las actividades centradas en la arquitectura para manejar el ciclo de vida del sistema de software. Estos métodos hacen el desarrollo de software más fácil y más consistente. [Nord]

Tanto el QAW y el ADD, como se pudo ver anteriormente, pueden emplearse sin omitir ninguno de sus pasos dentro de un proceso de desarrollo de software ágil. Sin embargo, no ocurre lo mismo con ATAM/CBAM. ATAM/CBAM necesitan una adaptación dentro de los métodos ágiles, dicha adaptación se hace al dividir las fases de ATAM, y todos sus pasos, dentro de las diferentes actividades de *Scrum*, identificando que actividades de ATAM/CBAM pueden ser mapeadas en las actividades que ya existen para *Scrum*, un mapeo similar lo podemos ver en [Kanwal] y [Farhan]. La Tabla 69, Tabla 70 y Tabla 71 muestran la distribución de las actividades de ATAM/CBAM dentro de *Scrum*.

Revisión del diseño intermedio

La arquitectura de software suele consistir de componentes muy complejos. Si esos componentes intermedios son inapropiados, la arquitectura no puede ser determinada. ARID es un método ligero para la evaluación del diseño, el cual puede ser utilizado para examinar un diseño cuando está siendo desarrollado y antes de que sea entregado. En la Tabla 64 se describe el método ARID [Nord], [RobertL].

ARID (Active Reviews for Intermediate Designs)	
Descripción	El método ARID combina la revisión activa de diseño con ATAM, creando una técnica para revisar los diseños parcialmente terminados. Las revisiones se enfocan en ver si el diseño es suficiente para darle soporte al equipo de desarrollo que lo usará. El método ARID ayuda a encontrar los asuntos y problemas que dificultan el uso exitoso del diseño una vez que este es concebido.
Cuando en el ciclo de desarrollo ágil	En método ARID se puede utilizar para la revisión de la arquitectura durante un Sprint 0 después del método ATAM para un análisis más detallado. También, se emplean antes de la definición o redefinición del diseño.
Ventajas y soporte para la capa conceptual ágil	El método ARID ayuda a los desarrolladores a entregar versiones del sistema de manera iterativa y con valor para el cliente.
Más información:	[Paul C]

Tabla 64 Características de ARID dentro del ciclo de desarrollo ágil

La Tabla 65 es una adaptación de [RobertL] para mostrar el valor agregado que añade el método ARID a las actividades específicas de: integración continua, entregas pequeñas y TDD.

Actividad/Práctica ágil +A	Valor agregado por el método ARID
Integración continua	Ayuda a identificar los desajustes de la arquitectura al nivel de interfaz lo más pronto posible.
Entregas pequeñas	Se enfoca en una porción de la arquitectura, permitiendo evaluar sólo lo que se necesite entregar.
TDD	Se hace una evaluación temprana dentro del ciclo de desarrollo del software. En lugar de probar el código, el diseño detallado de la interfaz es probado mediante los escenarios de atributos de calidad.

Tabla 65 Valor agregado por ARID a la capa conceptual ágil

Los elementos que participan en el método ARID y adaptados de [PaulClements], [Rob Wojcik] con base a [RobertL], se muestran en la Figura 5-10.

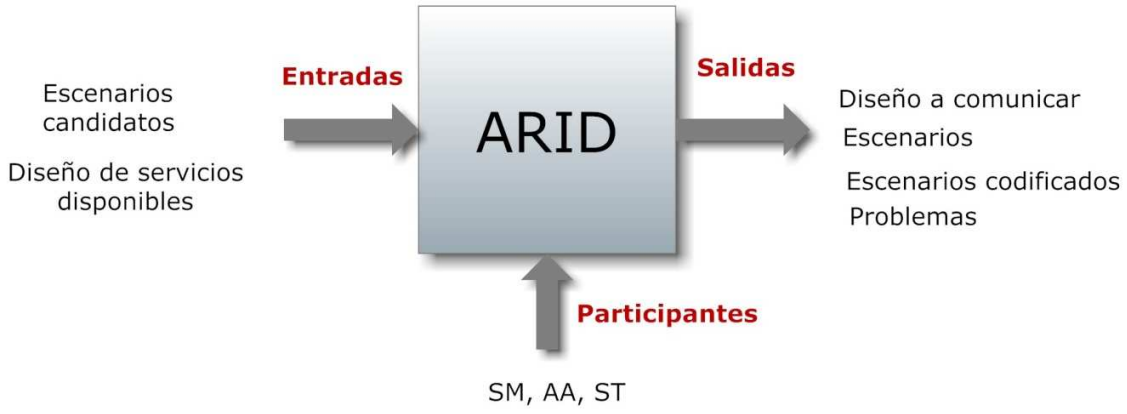

Figura 5-12 Elementos del método ARID para el marco ágil

Por último, en la Tabla 66 se muestra un resumen del valor agregado por las actividades de la capa conceptual de arquitectura a las actividades de: Crear la visión del sistema, *sprint planning meeting,* diseño, análisis y pruebas.

Actividades ágiles y actividades arquitectónicas	
Actividad/Práctica ágil	**Valor agregado por las actividades arquitectónicas**
Crear visión del sistema, *release planning meeting y Sprint planning meeting*	Las metas del negocio determinan los atributos de calidad utilizando QAW – Las historias de usuario se complementan con los escenarios de atributos de calidad, que capturan los asuntos de los stakeholders referentes a los requerimientos de atributos de calidad. – Los escenarios ayudan al AA para comunicar los requerimientos de atributos de calidad a los desarrolladores, y que estos influyan en el diseño. – El refinamiento y priorización de los escenarios, le dan al cliente y a los desarrolladores la información adicional para ayudarles a escoger sus historias para cada *Sprint* .
Diseño	Los atributos de calidad dirigen el diseño, utilizando el ADD – El enfoque step-by-step para definir la arquitectura de software, complementa el diseño incremental; el nivel de detalle es flexible. La arquitectura permite una mejor planeación para que los desarrolladores puedan hacer mejores estimaciones sobre el impacto que pueden causar los nuevos requerimientos. – Los desarrolladores crean la arquitectura suficiente para asegurar que el diseño va a producir un sistema que satisfaga los requerimientos de atributos de calidad y para mitigar los riesgos. – Las tácticas arquitectónicas ayudan a la refactorización, la cual es dirigida por las necesidades de los atributos de calidad (tales como hacerlo más rápido o más seguro)
Análisis y pruebas	El análisis del diseño provee una temprana retroalimentación al utilizar ATAM o CBAM – El ST puede utilizar los escenarios para evaluar el diseño y proveer una entrada para el análisis durante las pruebas. – La evaluación de la arquitectura tiene una noción semejante al triaje[11] para producir tanta información como sea necesaria y priorizar sobre la base de la importancia del negocio y la dificultad de los esfuerzos arquitectónicos. – La evaluación arquitectónica provee una temprana retroalimentación para comprender los trade-offs técnicos, el riesgo y el ROI (Retorno de inversión) de las decisiones arquitectónicas. El riesgo está relacionado con las decisiones técnicas y las metas del negocio, dada la justificación de los desarrolladores para invertir recursos para mitigarlos.

Tabla 66 Valor agregado de las actividades arquitectónicas a las actividades de: crear visión del sistema, *sprint planning meeting, diseño, análisis y pruebas.*

[11] Triaje (del francés *triage*) este término se emplea para la selección de pacientes en distintas situaciones y ámbitos. En situación normal se privilegia la atención del paciente más grave, el de mayor prioridad.

La Tabla 67 y Tabla 68 muestran un resumen de las influencias que las prácticas arquitectónicas tienen sobre las actividades definidas en la capa conceptual ágil. La Figura 5-13 muestra el mapeo de QAW, ADD, ATAM/CBAM y ARID en el MCAA durante el sprint 0 y la figura muestra el mapeo de QAW, ADD, ATAM/CBAM, ARID y las prácticas XP en el MCAA para los *sprints* subsecuentes.

Gestión ágil	Práctica de desarrollo	QAW	ADD	ATAM/CBAM	ARID
Crear la visión del sistema	N/A	Comprender los requerimientos y mejorar las historias de usuarios.	Con el árbol de utilidad se prioriza el *product backlog*.	Proporciona información importante para el PO y el AA para la priorización del product backlog.	
	Metáfora del sistema	Ayuda a crear la metáfora del sistema.	Define la estructura paso a paso en vistas de concurrencia y despliegue.	Evalúa la metáfora	
Release planning meeting	N/A	Ayuda a mejorar las estimaciones para la entrega.	Con el árbol de utilidad se mejora las priorización del *product backlog.*	Los escenarios ayudan a planear la entrega.	
Sprint planning meeting	N/A			Los escenarios ayudan a la construcción del sprint backlog.	
Sprint 0..n	N/A	Sus productos de trabajo son utilizados durante el sprint 0	El ADD es realizado durante el sprint 0	El ATAM/CBAM es realizado durante el sprint 0 y cada vez que se necesite.	El ARID es realizado para evaluar diseño intermedio.
					Evalúa sólo lo que se tiene que entregar.
	TDD	Ayuda a la evaluación del diseño y el análisis durante las pruebas.			Aporta una evaluación temprana dentro del ciclo de desarrollo del software.

Tabla 67 Valor agregado de las actividades arquitectónicas a la capa conceptual ágil (1/2)

Gestión ágil	Práctica de desarrollo	QAW	ADD	ATAM/CBAM	ARID
Sprint 0..n	Diseño simple		Proporciona una arquitectura suficiente para el cumplimiento de los requerimientos de calidad		
	Refactorización		Es beneficiada por la elección de las tácticas arquitectónicas.	Contribuye con los productos de trabajo necesarios para la comprensión del diseño antes de la refactorización	
	Programación en pares	Sus productos de trabajo son empleados durante el desarrollo.	El diseño creado es empleado por los programadores.		Permite evaluar diseños específicos, en lugar de toda la arquitectura.
	Propiedad colectiva		Sus productos de trabajo ayudan a encontrar fallas y		
	Estándares de codificación				
	Integración continua				Identifica los desajustes de la arquitectura a nivel de interfaz.
Sprint review meeting					
Sprint Retrospective					

Tabla 68 Valor agregado de las actividades arquitectónicas a la capa conceptual ágil (2/2)

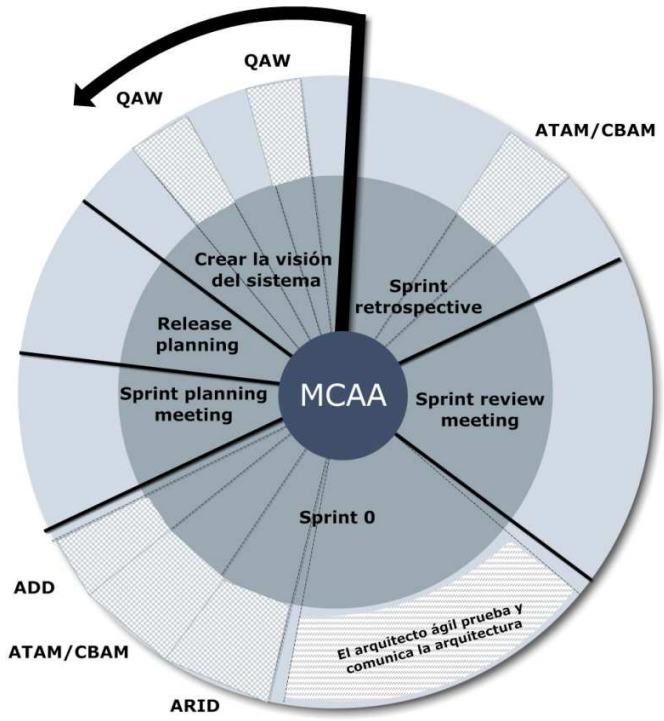

Figura 5-13 Mapeo de QAW, ADD, ATAM/CBAM y ARID en el MCAA durante el sprint 0

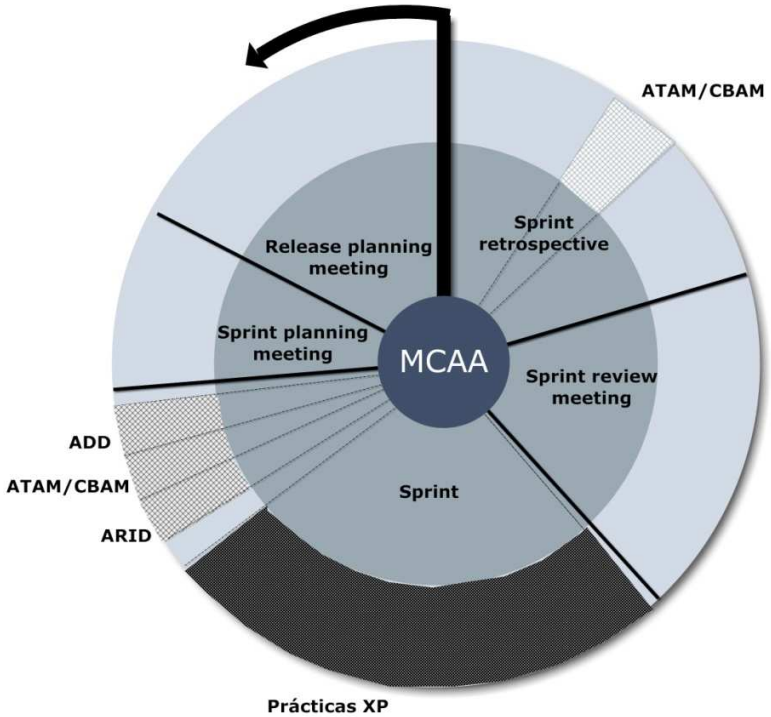

Figura 5-14 Mapeo de QAW, ADD, ATAM/CBAM, ARID y prácticas XP en el MCAA para sprints subsecuentes

La Tabla 69 , Tabla 70 y Tabla 71, muestran los pasos de las actividades arquitectónicas (tercera columna) y la actividad arquitectónica (cuarta columna) que se realiza durante una actividad de gestión ágil (primera columna), además de las actividades de desarrollo que ésta afecta (segunda columna).

Gestión ágil	Actividad de desarrollo	Pasos de la actividad arquitectónica	Actividad arquitectónica
Crear la visión del sistema	N/A	9. Presentación e introducción.	QAW
		10. Presentación de la misión del negocio.	
		11. Presentación del plan arquitectónico.	
	Crear Metáfora del sistema.	12. Identificación de las directrices arquitectónicas.	
		13. Lluvia de escenarios de atributos de calidad.	
		14. Consolidación de los escenarios de atributos de calidad.	
		15. Establecer la prioridad de los escenarios de atributos de calidad.	
		16. Refinamiento de los escenarios de atributos de calidad.	
	N/A	Fase 0 Preparación y evaluación	ATAM/CBAM
		Fase 1 Evaluación inicial	
		10. Presentar el ATAM	
		11. Presentar las directivas del negocio	
Sprint release meeting	N/A	N/A	N/A
Sprint planning meeting			
Sprint	N/A	9. Confirmar que existe la suficiente información sobre los requerimientos.	ADD
		10. Elegir un elemento del sistema para descomponerlo	
		11. Identificar directrices arquitectónicas que pueden ser candidatas.	
		12. Elegir el concepto de un diseño que satisfaga las directrices arquitectónicas.	
		13. Instanciar los elementos arquitectónicos y asignar responsabilidades.	
		14. Definir las interfaces para los elementos instanciados.	
		15. Verificar y refinar los requerimientos, y hacerlos restricciones para los elementos instanciados.	
		16. Repetir desde el paso 2 si es necesario.	

Tabla 69 Actividades arquitectónicas durante el ciclo de vida del proyecto (1/3)

Gestión ágil	Actividad de desarrollo	Pasos de la actividad arquitectónica	Actividad arquitectónica
Sprint		**ATAM**	ATAM/CBAM
		Fase 1 Evaluación inicial (continuación)	
		12. Presentar la arquitectura	
		13. Identificar los enfoques arquitectónicas	
		14. Generar el árbol de utilidad	
		15. Análisis de los enfoques arquitectónicos	
		Fase 2 Evaluación completa	
		16. Lluvia de ideas y establecimiento de la prioridad de los escenarios de atributos de calidad.	
		17. Análisis de los enfoques arquitectónicos	
		Fase 3 Seguimiento	
		18. Presentar los resultados.	
		CBAM	
		1. Reunir los escenarios de atributos de calidad.	
		2. Refinar los escenarios de atributos de calidad.	
		3. Priorización de escenarios de atributos de calidad.	
		4. Asignar utilidad.	
		5. Identificar estrategias arquitectónicas para los escenarios de atributos de calidad seleccionados y determinar el nivel de respuesta para todos los escenarios de atributos de calidad afectados.	
		6. Determinar el valor de la utilidad esperada de la estrategia arquitectónica utilizando interpolación.	
		7. Calcular el beneficio obtenido de la estrategia arquitectónica elegida.	
		8. Elegir estrategias arquitectónicas basándose en el ROI.	
		9. Validar las estrategias seleccionadas usando la intuición.	

Tabla 70 Actividades arquitectónicas durante el ciclo de vida del proyecto (2/3)

Gestión ágil	Actividad de desarrollo	Pasos de la actividad arquitectónica	Actividad arquitectónica
Sprint		1. Identificar los revisores.	ARID
		2. Preparar la presentación del diseño.	
		3. Preparar los escenarios de atributos de calidad.	
		4. Prepararse para las reuniones de revisión/ preparar el material.	
		5. Presentar el método ARID	
		6. Presentar el diseño	
		7. Lluvia de ideas y priorizar los atributos de calidad.	
		8. Realizar la revisión.	
		9. Presentar conclusiones.	
	TDD	Todas las actividades de desarrollo son influenciadas por los productos de trabajo producidos durante las etapas de diseño de la arquitectura inicial. Estas actividades se ven influenciadas de acuerdo con los establecido en la Tabla 59, Tabla 61, Tabla 63, Tabla 65 y Tabla 67	QAW ADD ATAM/CBAM ARID
	Diseño simple		
	Refactorización		
	Programación en pares		
	Propiedad colectiva		
	Estándares de codificación		
	Integración continua		
Sprint review	N/A	N/A	N/A
Sprint retrospective		Fase 0 Preparación y evaluación	ATAM/CBAM
		Fase 1 Evaluación inicial	
		1. Presentar el ATAM	
		2. Presentar las directivas del negocio (en caso de haber cambiado)	

Tabla 71 Actividades arquitectónicas durante el ciclo de vida del proyecto (3/3)

5.8 Proceso de desarrollo con el MCAA

Con el fin de ayudar a comprender la integración de las prácticas arquitectónicas dentro del ciclo de desarrollo del MCAA, en esta sección se explican las diferentes fases por las que se pasa durante el ciclo de desarrollo del software. La Figura 5-15 muestra el flujo general de actividades

desarrolladas a lo largo de una versión de software, enmarcando en color azul las actividades de "obtención de los requerimientos arquitectónicos" y "Diseño arquitectónico detallado".

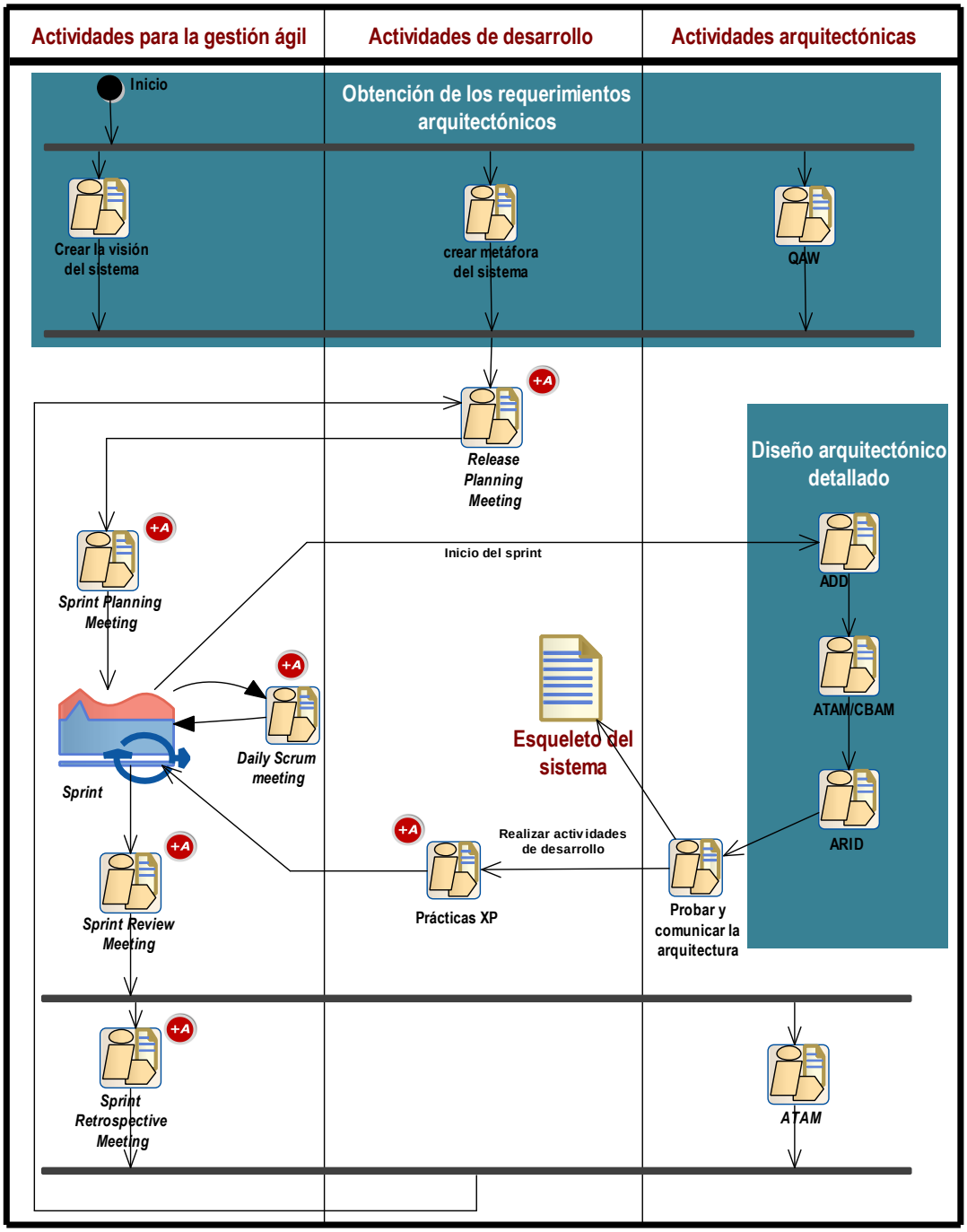

Figura 5-15 Flujo de actividades con las prácticas arquitectónicas.

5.8.1 Obtención de los requerimientos arquitectónicos

Durante la creación de la visión del sistema, el arquitecto ágil define el contexto del sistema y con ello, determina la cantidad de prácticas arquitectónicas necesarias para el proyecto. En ésta etapa se puede ejecutar la tarea de "estudio de activos de arquitectura", con ella se motiva a la reutilización. Se ejecuta en paralelo el taller de atributos de calidad (QAW, ver sección 5.7.4.1), del cual saldrán las principales directrices arquitectónicas. Los pasos del QAW, descritos en la Tabla 69, son ejecutados durante la captura de los requerimientos del sistema (ver Figura 5-8), es decir, durante la producción de las historias de usuario. Mediante el QAW, se captura de manera oportuna los atributos de calidad del sistema en la forma de escenarios de atributos de calidad, clarificando los requerimientos antes de crear el esqueleto del sistema. A su vez, la creación de la metáfora del sistema, es construida a un alto nivel de abstracción. Es importe mencionar que, la metáfora del sistema deberá incluir los primeros bosquejos de la arquitectura, también conocidos como: *spikes* arquitectónicos. Los *spikes* arquitectónicos viene a sustituir a la "prueba del concepto de la arquitectura" (*Architecture Proof-of-Concept*) de métodos tradicionales.

La metáfora del sistema, construida durante la creación de la visión del sistema, no brinda una guía detallada que se pueda emplear durante la implementación del software; por lo tanto, será hasta el termino del sprint 0 qué se especifique una arquitectura inicial o esqueleto del sistema, que pueda servir de guía para la implementación.

La creación del *product backlog* ocurre durante la creación de la visión del sistema. Para construir correctamente el *product backlog,* el arquitecto ágil deberá apoyar al *product owner* (ver características del arquitecto ágil en la sección 5.7.2 más atrás) en la priorización de las historias de usuario para construir el *product backlog,* expresándole al *product owner* la importancia de ciertas tareas que son relevantes para la arquitectura y que deberán tener mayor prioridad. La Figura 5-16 muestra la colaboración del arquitecto ágil durante la producción del *product backlog.*

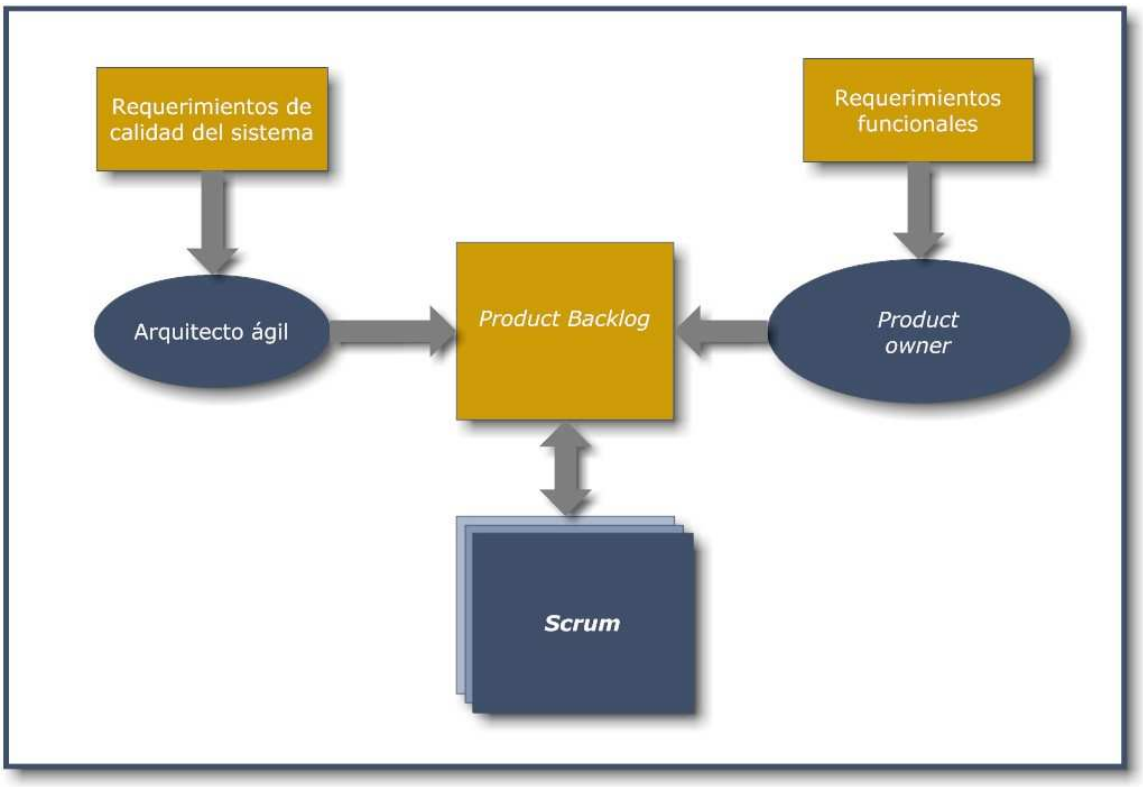

Figura 5-16 Creación del *product backlog*

5.8.2 Diseño arquitectónico detallado

Una vez terminada la captura de los requerimientos arquitectónicos y la construcción del *product backlog,* se procede a ejecutar el sprint 0 (ver Figura 5-13,Tabla 69, Tabla 70,Tabla 71) del proyecto. Durante esta etapa el arquitecto ágil se encargará de: las principales decisiones sobre el hardware y software que se empleara durante el proyecto, establecer importantes patrones de diseño, identificar los componentes o servicios que se pueden reutilizar, generar diagramas con un alto nivel de abstracción, clarificar las principales directrices arquitectónicas encontradas durante la fase de captura de los requerimientos arquitectónicos y establecer canales de comunicación con los *stakeholders;* esto se logra mediante reuniones con los *stakeholders*, con el fin de entender y conocer sus asuntos de interés, así como también, compartir la dirección técnica del proyecto con ellos.

El diseño arquitectónico detallado, es alcanzado mediante el desarrollo de las prácticas de diseño (método ADD), análisis y evaluación de la arquitectura (métodos ATAM/CBAM Y ARID). Estos métodos proporcionarán una arquitectura que servirá de **esqueleto para el sistema**, la cual deberá ser probada y comunicada por el arquitecto ágil antes de terminar el sprint 0.

Una vez terminado el sprint 0, la ejecución de los *sprints* subsecuentes se hace de una manera semejante a la de cualquier proyecto *Scrum*. La diferencia se encuentra cuando el cliente decide incluir un nuevo requerimiento ó algún cambio de cualquier índole. Cuando esto ocurre, es necesario hacer una evaluación sobre el impacto que este nuevo requerimiento pueda tener sobre la arquitectura. Por lo regular, los requerimientos de una nueva funcionalidad no tendrían efectos relevantes sobre la arquitectura; es decir, los requerimientos de alguna funcionalidad no suelen ser directrices arquitectónicas y por lo tanto, no impactarían en la arquitectura del sistema. Sin embargo, cuando algún nuevo requerimiento es detectado como una nueva directriz arquitectónica (por ejemplo, un cambio en un atributo de calidad), será necesario hacer una refactorización a nivel de estructura, ejecutando las prácticas arquitectónicas que sean necesarias durante el comienzo del *sprint*.

Durante el *sprint planning meeting el product owner* descompone las historias de usuarios hasta que estas sean lo suficientemente pequeñas para ser ejecutadas en un *sprint.* Por su parte, el arquitecto de software contribuye con la descomposición de las historias de usuario mediante la identificación de los límites de la arquitectura, por lo que trabaja con el *product owner* y el *Scrum team* para asegurar de que toda la descomposición del trabajo sigue los límites de ésta. La descomposición, anteriormente mencionada, es conocida como "descomposición en forma manejable para cada *sprint" o sprintable form;* por ejemplo, sí se ha elegido una arquitectura multicapa, los límites de la arquitectura están asociados a cada una de las capas que la componen, por lo tanto, la descomposición y ejecución de cada tarea, durante un sprint, deberán hacerse tomando en cuenta el tiempo y momento en que cada una de estas capas pueden ser construidas (por ejemplo, la capa de persistencia no puede ser construida, en su totalidad, sin contar con el diseño de la base de datos).

De acuerdo con la Tabla 59, Tabla 61, Tabla 63, Tabla 65 y Tabla 67, las actividades arquitectónicas (QAW, ADD, ATAM/CBAM y ARID) apoyan a las prácticas de desarrollo de software como se muestra en la Figura 5-17. En esta figura podemos ver que las prácticas arquitectónicas son la base y soporte de las prácticas XP. Sus productos de trabajo son empleados durante el desarrollo del sistema, y cada práctica XP se apoya una de la otra, por lo que todas son influenciadas por las prácticas arquitectónicas.

Figura 5-17 Influencia de las prácticas arquitectónicas en las prácticas de desarrollo de XP.

La Figura 5-18 es una ampliación de la Figura 5-17, en ella se muestra la ejecución de un sprint que es influenciado por las prácticas arquitectónicas desarrolladas durante el sprint 0. A diferencia de la Figura 5-5, esta figura muestra los nuevos requerimientos arquitectónicos que puedan ir surgiendo durante el *sprint.*

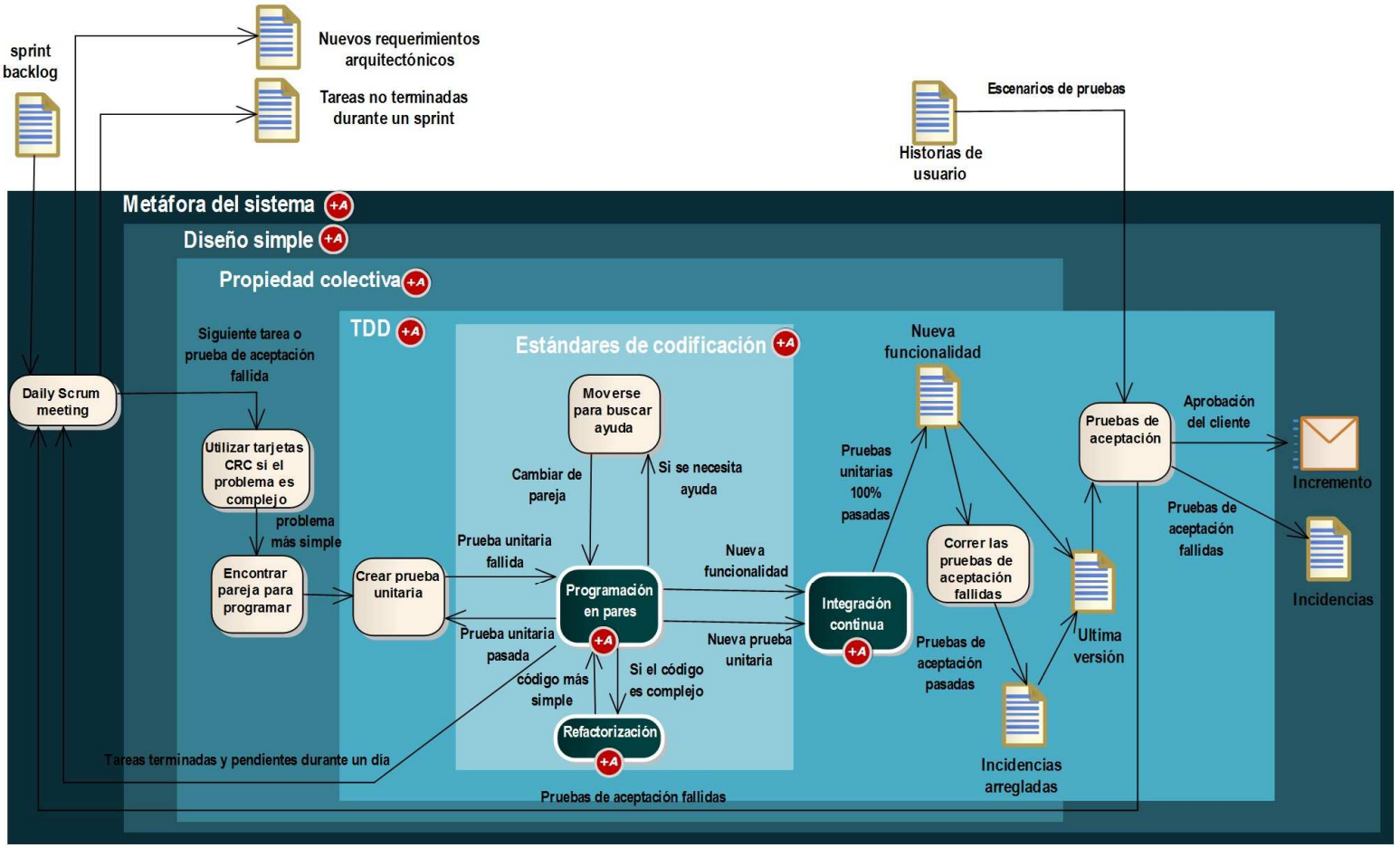

Figura 5-18 Desarrollo del software durante un *sprint* con arquitectura

6 Conclusiones y trabajo a futuro

6.1 Conclusiones

El objetivo principal de este trabajo, fue integrar las prácticas del diseño arquitectónico y el rol del arquitecto de software dentro de los métodos ágiles; en especial, dentro de *Scrum y Extreme Programming*. Para lograr esta integración, se realizó una investigación sistemática para identificar el rol y las prácticas relacionadas con la arquitectura de software que son utilizadas dentro de los métodos ágiles. En base a esta investigación sistemática, se definió un marco híbrido para la integración de las prácticas arquitectónicas dentro los métodos ágiles antes mencionados.

A continuación se presentan las conclusiones generales a las que se llegó:

- El rol del arquitecto de software es muy importante dentro de cualquier proyecto de desarrollo de software, donde su principal responsabilidad es la de cuidar los atributos de calidad que el sistema debe mostrar.
- Tanto la arquitectura como los métodos ágiles no son opuestos, más bien, la arquitectura, y sus prácticas, apoyan en el cumplimiento de los valores y principios ágiles.
- Los métodos centrados en la arquitectura, son construidos sobre los conceptos y técnicas que los practicantes de un enfoque ágil pueden integrar.
- Las prácticas arquitectónicas fomentan la construcción de software altamente reutilizable, permitiendo obtener beneficios importantes para la entrega oportuna y con valor para el cliente en proyectos futuros, mediante la adopción de la práctica “estudio de activos de arquitectura”.
- Los métodos centrados en la arquitectura pueden añadir valor a los métodos ágiles, mediante el énfasis en los atributos de calidad y su rol en el moldeado del diseño de la arquitectura, además de hacer posible adaptar métodos ágiles usando un enfoque hibrido para manejar el diseño de sistemas más grandes y complejos.
- Los desarrolladores ágiles, perciben a la arquitectura de software tan importante y de apoyo para los valores ágiles. Por lo que consideran que las prácticas arquitectónicas deberían de estar integradas en los métodos ágiles.
- Los métodos provistos por el SEI para el análisis, diseño y evaluación de la arquitectura, son de gran importancia y brindan mucho valor a los métodos ágiles.
- Los métodos ágiles no son una alternativa fiable para la construcción de sistemas críticos o sistemas más complejos. De hecho se considera que no son adecuados cuando se trabaja en un dominio desconocido, con un nuevo cliente o con soluciones no probadas. Por lo tanto, para que los métodos ágiles sean más robustos, se requiere que estos adopten

prácticas que sean inherentes al contexto del sistema. Muchas de estas prácticas son provistas por las prácticas arquitectónicas.

- Las prácticas arquitectónicas incrementan la comunicación entre los *stakeholders* del proyecto.
- Se definieron las competencias que deberá cubrir un arquitecto ágil.
- Se propuso y se obtuvo la validación, por dos expertos en arquitecturas de software, de un marco híbrido para la integración de las prácticas arquitectónicas dentro de los métodos ágiles.

6.2 Trabajo a futuro

A continuación se muestran algunas propuestas para la continuidad del presente trabajo:

- Validar el MCAA en la práctica dentro de un desarrollo de software real.
- Identificar y proponer herramientas que puedan ser empleadas dentro del MCAA, para ayudar a ejecutar las prácticas arquitectónicas.
- Hacer una comparativa entre el MCAA y otros marcos híbridos que manejen otras prácticas arquitectónicas.
- El MCAA está basado en prácticas arquitectónicas propuestas por el SEI, por lo que se recomienda hacer una integración de otras prácticas diferentes a las del SEI.
- El MCAA basa sus actividades de desarrollo en las prácticas de XP, por lo que sería recomendable utilizar algún otro método ágil que se enfoque en el desarrollo, por ejemplo, FDD.
- Proponer paquetes de puesta en operación para fomentar el uso del MCAA.

7 Referencias bibliográficas

[Abrahamsson] Abrahamsson, P.; Babar, M.A.; Kruchten, P.: *Agility and Architecture: Can They Coexist?* Software, IEEE , vol.27, no.2, pp.16-22, Marzo-Abril, 2010.

[AlexanderWolf] Alexander Wolf: *Succeedings of the Second International Software Architecture Workshop*. (ISAW-2). ACM SIGSOFT Software Engineering, 1997.

[AlianzaAgil] Scrum Alliance. [citado por última vez el 9 de febrero del 2011] http://www.agilealliance.org

[Babar2009] Babar, M.A.: *An exploratory study of architectural practices and challenges in using agile software development approaches*. Software Architecture, 2009 & European Conference on Software Architecture. WICSA/ECSA 2009. Joint Working IEEE/IFIP Conference on , vol., no., pp.81-90, 14-17 Septiembre, 2009.

[BeckKent] Beck, Kent: *Extreme Programming explained*. Reading, Mass: Addison-Wesley. Longman, Inc. 2000.

[Boehm2002] Boehm, B.: *Get ready for agile methods, with care.* Computer , vol.35, no.1, pp.64-69, Enero, 2002.

[Boehm95] Barry Boehm: *Engineering Context (for Software Architecture).* Invited talk, First International Workshop on Architecture for Software Systems. Seattle, Washington, Abril de 1995.

[ChristineHofmeister00] Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran, and Pierre America. 2007. *A general model of software architecture design derived from five industrial approaches*. J. Syst. Softw. 80, 1. Enero, 2007.

[ClementsSzyperski] Clements Szyperski, Dominik Gruntz and Stephan Murer: *Component Software Beyond Object-Oriented Programming*, Addison Wesley, ACM press 2002.

[CraigLarman] Craig Larman: *Agile and iterative development a manager´s guide*. Addison Wesley, 2004

[Dennis Mancl] Dennis Mancl, Steven Fraser, Bill Opdyke, Ethan Hadar, and

Irit Hadar. 2009. *Architecture in an agile world.* In Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA '09). ACM, New York, NY, USA, 719-720.

[Donadio 2004] Antonio Donadío medaglia, Elena Dieck, Bertha García, Dolores Lankenau, Imelda Valdes: *Negocios en ambientes computacionales*, McGraw-hill, 2004

[DonWells] Página oficial de James Donovan Wells. [citado por última vez el 22 de abril del 2011] http://www.extremeprogramming.org/

[Ethan Hadar] Ethan Hadar and Gabriel M. Silberman: *Agile architecture methodology: long term strategy interleaved with short term tactics*. In Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA Companion '08). ACM, New York, NY, USA, 641-652, 2008.

[Faber] Faber, R.: *Architects as Service Providers*. Software, IEEE, vol.27, no.2, pp.33-40, Marzo-Abril, 2010.

[Falessi] Falessi, D.; Cantone, G.; Sarcia, S.A.; Calavaro, G.; Subiaco, P.; D'Amore, C.: *Peaceful Coexistence: Agile Developer Perspectives on Software Architecture*. Software, IEEE , vol.27, no.2, pp.23-25, March-April 2010

[Farhan] Farhan, S.; Tauseef, H.; Fahiem, M.A.: *Adding Agility to Architecture Tradeoff Analysis Method for Mapping on Crystal.* Software Engineering, 2009. WCSE '09. WRI World Congress on, vol.4, no., pp.121-125, 19-21. Mayo, 2009.

[Gar00] David Garlan: *Software Architecture: A Roadmap*. En Anthony Finkelstein, The future of software engineering, ACM Press, 2000.

[GarlanShaw011994] Mary Shaw y David Garlan: *Characteristics of Higher Level Languages for Software Architecture*. SEI. Diciembre, 1994.

[GarlanShaw1994] David Garlan and Mary Shaw: *An Introduction to Software Architecture*, SEI. Enero, 1994.

[Highsmith 01] Highsmith, Jim y Highsmith, james A: Agile Software Development Ecosystems. Boston, Addison-Wesley, 2002.

[HighsmithASD] 2003. Cockburn, J. Highsmith: *Agile software development, the people factor*. Computer, 2001,

Vol: 34(11), pp.131-133

[Hofmeister] Hofmeister, C., et al.: *A general model of software architecture design derived from five industrial approaches*. Journal of System and Software, 2007. 80(1): pp. 106-126.

[IanGordon] Gordon, Ian: *Essential Software Architecture*. Springer 2006

[IEEE 1471 2000] *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE Std 1471-2000 , vol., no., pp.i-23, 2000

[IshamM2008] Isham, M.: *Agile Architecture IS Possible You First Have to Believe!* Agile, 2008. AGILE '08. Conference , vol., no., pp.484-489, 4-8 Aug. 2008

[Ivar Jacobson] Ivar Jacobson, Grady Booch, James Rumbaugh: *The Unified Software Development Process*. Addison-Wesley, 2008.

[John K] John K. Bergey, Matthew J. Fisher: Use of the Architecture Tradeoff Analysis Method (ATAM) in the Acquisition of Software-Intensive Systems. Technical Note CMU/SEI-2001-TN-009. Software Engineering Institute, 2001.

[KaiQian] Qian, Kai, et al., Software Architecture and Design Illuminated, Jones and Bartlett Illuminated Series, Sudbury 2010.

[Kanwal] Kanwal, F.; Junaid, K.; Fahiem, M.A.: *A Hybrid Software Architecture Evaluation Method for FDD – An Agile Process Model*. Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on , vol., no., pp.1-5, 10-12 Dec.

[Kruchten 01] Kruchen, P: Mommy, Where Do Software Architectures Come From?. 1st International Workshop on Architectures for Software Systems, Seattle, 1995

[Lane1990] Thomas G. Lane : *Studying Software Architecture Through Design Spaces and Rules*. Technical Report CMU/SEI-90-TR-18 ESD-90-TR-219, noviembre, 1990.

[Lawrence G] Lawrence G. Jones, Anthony J. Lattanze: Using the Architecture Tradeoff Analysis Method to Evaluate a Wargame Simulation System: A Case Study. Technical Note CMU/SEI-2001-TN-022. Software Engineering Institute, 2001.

[Madison 00] Madison, J.: *Agile Architecture Interactions*. Software, IEEE , vol.27, no.2, pp.41-48, Marzo-Abril, 2010.

[ManifiestoAgil] Página oficial del manifiesto ágil. [citado por última vez el 13 de junio del 2011] http://www.agilemanifesto.org

[MicrosoftGuide] Guía para el desarrollo ágil de software . [citado por última vez el 22 de junio del 2011] http://apparch.codeplex.com/

[MingHuo] Ming Huo; Verner, J.; Liming Zhu; Babar, M.A.: *Software quality and agile methods*. Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International , vol., no., pp. 520- 525 vol.1, 28-30 Sept. 2004

[Nord] Nord, R.L.; Tomayko, J.E.: *Software architecture-centric methods and agile development*. Software, IEEE , vol.23, no.2, pp. 47- 53, Marzo-Abril, 2006.

[Paul C] Paul C. Clements: *Active Reviews for intermediate Designs.* Technical Note CMU/SEI-2000-TN-009. Software Engineering Institute, 2000.

[PaulClements] Lean Bass, Paul Clements, Rick Kazman: *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley 2003.

[PaulClements01] PaulClements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford: *Documenting Software Architecture views and beyond.* Addison Wesley, SEI series in software engineering, 2002.

[PaulCLindaNorth96] Paul C. Clements y Linda M. Northrop: *Software architecture: An executive overview*. Technical Report, CMU/SEI-96-TR-003, ESC-TR-96-003. Febrero de 1996.

[PearceRobinson 2007] John A. Pearce II, Richard B. Robinson Jr: *Strategic Management*. Decima edición, McGraw-hill, 2007

[Peter Eels] Peter Eels, Peter Cripps: *The process of software architecting.* Addison-Wesley, 2010

[PeterEeles] Peter Eeles: Understanding Architectural Assets. In Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008) (WICSA '08). IEEE Computer Society, Washington, 2008.

[PhilippeKruchten2010] Philippe Kruchten: *Software architecture and agile software development: a clash of two cultures?*. In Proceedings of

the 32nd ACM/IEEE International Conference on Software Engineering – Volume 2 (ICSE '10),
Vol. 2. ACM, New York, NY, USA, 497-498, 2010.

[QAW] Quality Attribute Workshop. Technical Report CMU/SEI 200 2003-tr-016. Software Engineering Institute, 2003.

[RedmondWashington] Schwaber, Ken: *Agile Project Management with Scrum.* Redmond Washington : Microsoft Press, 2004.

[RichardN2010] Richard N. Taylor, Nenad Medvidovic y Eric M. Dashofy: *Software Architecture foundations, Theory and Practice*. Wiley 2010

[Rick Kazman] Rick Kazman, Robert L. Nord, Mark Klein: A Life-Cycle View of Architecture Analysis and Design Methods. Technical Note CMU/SEI-2003-TN-026. Software Engineering Institute, 2003.

[Rob Wojcik] Rob Wojcik, Felix Bachman, Len Bass, Paul Clements, Paulo Merson, Robert Nord, Bill Wood. "Attribute-Driven Design" Technical Report CMU/SEI-2006-TR-023. Software Engineering Institute, 2006.

[RobertL] Robert L., Nord James, E. Tomayko, Rob Wojcik: *Integrating Software-Architecture-Centric Methods into Extreme Programming (XP)*. Technical Note CMU/SEI-2004-TN-036. Software Engineering Institute, 2004.

[ScottAmbler] Scott Ambler: *Agile modeling: Effective practices for extreme programming and unified process*. Wiley Computer Publishing. 2002

[scrumAlliance] Página dedicada a Scrum, [citado por última vez el 16 de febrero del 2011] http://www.scrumalliance.org/

[scrumAlliance01] Scrum in a nutshell, Douglas E. Shimp. [citado por última vez el 23 de febrero del 2011] http://www.scrumalliance.org/profiles/29-douglas-e-shimp

[ScrumOrg] Página oficial de Scrum, [citado por última vez el 12 de julio del 2011] http://www.scrum.org/

[SEI2010] Página 201ficial del *Software Engineering Institute* [citado por última vez el 2 de julio del 2011]
http://www.sei.cmu.edu/

[Sommerville 2011] Ian sommerville: *Software Engineering*. Addison-Wesley, 2011

[SPEM 2008] Página oficial de SPEM [citado por última vez el 7 de marzo del 2011] http://www.omg.org/spec/SPEM/2.0/

[SquaRE2009] ISO/IEC FCD 25010: *Systems and software engineering – System and software product Quality Requirements and Evaluation (SquaRE) System and software quality models,* 2009.

[Stephen R] Stephen R. Palmer, John M. Felsing: A practical Guide to Feature-Driven Development. The coad series, 2002

[Steven Fraser] Steven Fraser, Ethan Hadar, Irit Hadar, Dennis Mancl, Grenville (Randy) Miller, and Bill Opdyke. 2009. Architecture in an agile world. In Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications (OOPSLA '09). ACM, New York, NY, USA, 841-844.

[Thompson 2007] Arthur A. Thompson, A. J. Strickland, John E.Gamble: *Crafting and Executing Strategy*. 15th edition, McGraw-hill, 2007.

[XPRonaldJeffries] Página 202ficial de Ronald E. Jeffries [citado por última vez el 29 de junio del 2011] http://xprogramming.com/

[XPRonaldJeffries01] Jeffries, R., Anderson, A., Hendrickson, C.: *Extreme Programming Installed*. Addison-Wesley, 2001.