



Universidad Nacional Mayor de San Marcos

Universidad del Perú. Decana de América

Facultad de Ingeniería de Sistemas e Informática

Escuela Profesional de Ingeniería de Sistemas

**Arquitectura de software de una aplicación orientada a
la gestión de prestación de servicios**

TESINA

Para optar el Título Profesional de Ingeniero de Sistemas

AUTORES

Lenin Richard ESPINOZA MENDOZA

Mirian Azucena PESEROS CASTRO

ASESOR

Zhing Fong LAM

Lima, Perú

2005



Reconocimiento - No Comercial - Compartir Igual - Sin restricciones adicionales

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

Usted puede distribuir, remezclar, retocar, y crear a partir del documento original de modo no comercial, siempre y cuando se dé crédito al autor del documento y se licencien las nuevas creaciones bajo las mismas condiciones. No se permite aplicar términos legales o medidas tecnológicas que restrinjan legalmente a otros a hacer cualquier cosa que permita esta licencia.

Referencia bibliográfica

Espinoza, L. & Peseros, M. (2005). *Arquitectura de software de una aplicación orientada a la gestión de prestación de servicios*. [Tesina de pregrado, Universidad Nacional Mayor de San Marcos, Facultad de Ingeniería de Sistemas e Informática, Escuela Profesional de Ingeniería de Sistemas]. Repositorio institucional Cybertesis UNMSM.

DEDICATORIA

A mi familia, la principal fuerza que me impulsa.

Lenin Richard Espinoza Mendoza

A mis padres por todo su apoyo brindado.

Mirian Azucena Peseros Castro

RESUMEN

ARQUITECTURA DE SOFTWARE DE UNA APLICACIÓN ORIENTADA A LA GESTION DE PRESTACION DE SERVICIOS

Lenin Richard Espinoza Mendoza

Mirian Azucena Peseros Castro

Marzo-2005

Asesor : Zhing Fong Lam
Grado : Ingeniero de Sistemas

El presente trabajo tiene como objetivo mostrar la forma de abordar la solución del problema de la definición de la arquitectura de una aplicación orientada a la gestión de prestación de servicios.

Diversas metodologías existen e intentan administrar la construcción de software. Hoy en día el software esta presente en cada campo del conocimiento, tenemos aplicaciones de software de todo tipo y para cada problema solucionable usando software.

Las principales empresas al construir y administrar software se encuentran en problemas, la inexperiencia al tratar de administrar el software y las practicas erradas al solucionar problemas son algunas de las razones.

Así, el presente trabajo muestra la creación de software de Gestión de Prestación de Servicios y como se aborda su solución orientándola a definir una arquitectura.

Palabras clave:

Ingeniería de Software
Arquitectura de Software
Metodologías Ágiles
Rational Unified Process

ABSTRACT

SOFTWARE ARCHITECTURE OF AN APLICACIÓN ORIENTED TO THE MANAGEMENT OF SERVICES GIVING

Lenin Richard Espinoza Mendoza

Mirian Azucena Peseros Castro

March-2005

Adviser : Zhing Fong Lam

Degree : System Engineer

The present work has as an objective show the way for take the solution of the problem of define the architecture of an application oriented to the management of services giving.

Different methodologies exist and try to manage the software building. Today the software is present in each camp of the knowledge; we have software applications of each kind and for each problem where the software can be used. The main enterprises when construct and manage software have problems, the inexperience when manage the software and bad practices when a problem appear are some of the reasons.

And so, the present work take the theme of the creation of software of Management of Services Giving and how the solution is take oriented to define his architecture.

Keywords:

Software Engineering
Software Architecture
Agile Methodologies
Rational Unified Process

INDICE

RESUMEN	3
ABSTRACT	4
INDICE	5
INTRODUCCION	8
1. PLANTEAMIENTO DEL PROBLEMA	9
1.1. Antecedentes y formulación del problema	9
1.2. Justificación en importancia	10
1.3. Definición del problema.....	11
Problemas en la construcción de software	11
1.4. Limitaciones y alcances	11
1.5. Variantes del problema	12
2. OBJETIVOS	13
2.1. Objetivos Generales.....	13
2.2. Objetivos Específicos.....	13
3. MARCO TEORICO CONCEPTUAL	14
3.1. Antecedentes de la Investigación	14
3.2. Bases teóricas y Definición de términos básicos	15
3.2.1 Definiciones de Ingeniería de Software	15
3.2.2 Definiciones de Arquitectura de software.....	25
3.2.3 Que no es la Arquitectura de software.....	32
3.2.4 Arquitectura de Software en el Contexto	34
3.2.5 Motivación de la arquitectura de software (¿Por qué necesitamos arquitecturas de software?)	36
3.2.6 Arquitecto de Software.....	40
3.2.7 Relación entre la Ingeniería de Software y la Arquitectura de Software	44

4.	METODOLOGIA DE LA INVESTIGACION	45
4.1.	Estado del arte: Aporte al conocimiento.....	45
4.2.	Parte central de la tesina	46
4.2.1	ALTERNATIVAS METODOLOGICAS	46
4.2.2	Predictivo contra Adaptable	47
4.3.	Las Metodologías.....	62
5.	METODOLOGIA A UTILIZAR	71
5.1.	Comparando Procesos: RUP, métodos Ágiles, y estándares de gobierno pesados	71
5.2.	Adoptando RUP en un proyecto	77
5.3.	Rol y contenido de la documentación Arquitectónica.....	82
5.3.1	Visión general arquitectónica y manual de referencia arquitectónica.....	82
6.	DOCUMENTACION DEL CASO PRACTICO.....	86
6.1.	Introducción	86
6.2.	Propósito.....	86
6.3.	Interfaces	87
6.4.	Requerimientos no funcionales.....	87
6.5.	Representación de la Arquitectura.....	87
6.6.	Vista de Casos de Uso.....	88
6.6.1	Actores.....	88
6.6.2	Casos de Uso	88
6.6.3	Diagrama de Actores	91
6.6.4	Diagrama de Clases	92
6.6.5	Diagrama de Paquetes	93
6.6.6	Descripción de los Componentes	93
6.7.	Vista Lógica de Procesos.....	94
6.7.1	Diagrama de Actividades	94
6.8.	Vista de Desarrollo.....	97
6.8.1	Modelo Conceptual	97
6.8.2	Diagrama de Clases	98
6.8.3	Diagrama de Secuencia.....	98
6.8.4	Diccionario de Clases	99

6.9. Vista de Implementación Física	102
7. CONCLUSIONES	103
8. REFERENCIAS BIBLIOGRAFICAS	104
8.1. Referencias bibliográficas de libros.....	104
8.2. Referencias bibliográficas de artículos.....	106
8.3. Referencias bibliográficas de páginas Web	107
8.4. Otras referencias.....	108
8.5. Referencias bibliográficas de tesis.....	109
LISTA DE FIGURAS.....	110
LISTA DE TABLAS	111
REFERENCIAS BIBLIOGRAFICAS	112

INTRODUCCION

Diversas metodologías existen e intentan administrar la construcción de software. Hoy en día el software esta presente en cada campo del conocimiento, tenemos aplicaciones de software de todo tipo y para cada problema solucionable usando software.

Las principales empresas al construir y administrar software se encuentran en problemas, la inexperiencia al tratar de administrar el software y las practicas erradas al solucionar problemas son algunas de las razones.

Así, el presente trabajo muestra la creación de software de Gestión de Prestación de Servicios y como se aborda su solución orientándola a definir una arquitectura. Se tiene como objetivo mostrar la forma de abordar la solución del problema de la definición de la arquitectura de una aplicación orientada a la gestión de prestación de servicios.

El propósito es mostrar que la aplicación de Arquitecturas de Software permite crear un producto de software eficientemente brindándonos un mapa conceptual de los problemas que este producto debe solucionar.

1. PLANTEAMIENTO DEL PROBLEMA

1.1. Antecedentes y formulación del problema

La arquitectura de software ha ganado mucha popularidad en el mundo académico y comercial durante la última década. Numerosos beneficios potenciales han sido atribuidos a las arquitecturas: habilidad para fácilmente asegurar las propiedades del sistema; soporte mejorado para la reutilización de software; desarrollo de sistemas confiables, distribuidos, y a gran escala en el tiempo y dentro del presupuesto y así sucesivamente. De esta forma se ha convertido en área de intensa investigación en la comunidad de ingeniería de software. Para asegurar estos beneficios, los investigadores han desarrollado un conjunto de notaciones de modelamiento que permiten a los arquitectos especificar precisamente la estructura, comportamiento, y propiedades de los aspectos críticos de un sistema. La precisión de los modelos arquitectónicos descansa directamente en la semántica subyacente de cada notación, que a su vez, esta basado en un método de especificación formal (p. Ej. maquinas de estado finito, procesos de comunicación secuencial).

Un gran número de notaciones de modelamiento de arquitectura y herramientas de soporte, como también nuevos estilos arquitectónicos han surgido. El enfoque de la arquitectura de software es ir desde toscos bloques de construcción a líneas de código y una completa estructura de interconexión. El enfoque explicito tiene tremendo potencial para mejorar el estado actual del desarrollo de software y aliviar muchos de sus problemas.

1.2. Justificación en importancia

El papel de la arquitectura de software es parecido al papel que juega la arquitectura en la construcción de edificios. El edificio se contempla desde varios puntos de vista: estructura, servicios, conducción de la calefacción, fontanería, electricidad, etc. Esto permite a un constructor ver una imagen completa antes de que comience la construcción. Análogamente, la arquitectura en un sistema software se describe mediante diferentes vistas del sistema en construcción.

El concepto de arquitectura software incluye los aspectos estáticos y dinámicos más significativos del sistema. La arquitectura surge de las necesidades de la empresa, como las perciben los usuarios y los inversores, y se refleja en los casos de uso. Sin embargo, también se ve influida por muchos otros factores, como la plataforma en la que tiene que funcionar el software (arquitectura hardware, sistema operativo, sistema de gestión de base de datos, protocolos para comunicaciones en red), los bloques de construcción reutilizables de que se dispone (por ejemplo, un marco de trabajo para interfaces gráficas de usuario), consideraciones de implantación, sistemas heredados, y requisitos no funcionales (por ejemplo, rendimiento, fiabilidad). La arquitectura es una vista del diseño completo con las características más importantes resaltadas, dejando los detalles de lado. Debido a que lo que es significativo depende en parte de una valoración, que a su vez, se adquiere con la experiencia, el valor de una arquitectura depende de las personas que se hayan responsabilizado de su creación. No obstante, el proceso ayuda al arquitecto a centrarse en los objetivos adecuados, como la comprensibilidad, la capacidad de adaptación al cambio, y la reutilización.

1.3. Definición del problema

Problemas en la construcción de software

Los problemas de Arquitectura y gestión de calidad de software pueden dar lugar a muchos problemas en el funcionamiento del mismo, aquí les presentamos algunos casos:

- En Julio del 2004, los diarios reportaban que un sistema de administración de la beneficencia publica en Canadá que costo varios cientos de millones de dólares no fue capaz de manejar un simple incremento en el porcentaje de ayuda después de ser puesto en producción.
- El departamento de vehículos motorizados de California gastó sobre \$43 millones de dólares en un sistema para unir los sistemas de registro de conductores y el registro de vehículos. El sistema fue abandonado sin ni siquiera haber sido usado.
- Un fallido esfuerzo de \$165 millones de dólares de American Airlines de vincular su software de reserva de pasajes con el sistema de reservaciones de Marriott, Hilton y Budget
- El aeropuerto de Denver computarizó su sistema de equipaje, lo que resultó en millones de dólares perdidos por la demora en la apertura del aeropuerto

1.4. Limitaciones y alcances

Se mostrara parte del proceso de la creación de software que es la definición de una arquitectura para la construcción del mismo, tomamos el tema de la creación del software de Gestión de Prestación de Servicios para enfocar las definiciones más eficientemente. Escogeremos una metodología, un proceso y daremos las pautas de creación de la arquitectura de la aplicación.

El software de Gestión de Prestación de Servicios esta orientado a empresas que brindan servicios externos y necesitan cuantificar los recursos utilizados al realizar estas labores, es importante pues hay una buena cantidad de empresas en el medio que brindan servicios a terceros.

Este trabajo primero mostrara los conceptos de Arquitectura de Software existentes, el papel del Arquitecto de Software, las definiciones y metodologías existentes, las empresas que han revisado este tema y como lo definen para luego pasar a la aplicación y definición de la arquitectura de la aplicación de Gestión de Prestación de Servicios.

Nos centramos además en la creación de arquitecturas de software comercial, hacemos mención de esto para tener una referencia mas adecuada.

1.5. Variantes del problema

La Arquitectura de Software puede ser aplicada a todo software que pueda ser desarrollado. Todo sistema de software tiene una arquitectura subyacente, aunque esta no haya sido documentada o la documentación no sea visible a todos.

2. OBJETIVOS

2.1. Objetivos Generales

- Mostrar de una forma amplia el concepto de Arquitectura de Software y su importancia dentro de la Ingeniería de Software.
- Utilización de Ingeniería de Software para definir una Arquitectura de Software
- Establecer un marco referencial para definición de Arquitecturas de Software.
- Mostrar las tendencias y cambios sufridos por los procesos de desarrollo software.

2.2. Objetivos Específicos

- Desarrollar la arquitectura de software de una Aplicación Orientada a la Gestión de Prestación de Servicios.

3. MARCO TEORICO CONCEPTUAL

3.1. Antecedentes de la Investigación

Describir arquitecturas no es un tema nuevo. Algunos libros han sido publicados sobre estilos arquitectónicos y patrones arquitectónicos [Por ejemplo: Shaw, Mary and Garlan, David, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996 y Buschmann, Frank, et al., Pattern-Oriented Software Architecture, A System of Patterns, John Wiley & Sons, 1996.].

Varios lenguajes de descripción arquitectónica (Architecture Description Language - ADL) han sido desarrollados, aunque estos son primariamente orientados hacia **descripciones formales** de interfaces de componentes y conectores.

Por otro lado UML enfatizan la importancia de documentar el alto nivel de la arquitectura del sistema, pero UML es un lenguaje **semi-formal** con muchas limitaciones [OOSPLA'99, Workshop #23: "Rigorous modeling and analysis with UML: Challenges and Limitations". Denver, Noviembre de 1999].

RUP que esta muy ligado a UML es un proceso de desarrollo de software que hace notar la importancia de la arquitectura, al estar centrado en esta, como lo mencionan en el libro de Rational Software Corporation (que ahora pertenece a IBM Corp).

Recientemente Microsoft Corporation ha entendido la importancia de la arquitectura de Software, esta por lanzar un proceso de desarrollo llamado MSF Agile donde también hace notar la importancia de la arquitectura, organiza eventos de Arquitectura de Software y acaba de lanzar un programa de certificación de Arquitectos de Software.

3.2. Bases teóricas y Definición de términos básicos

La Arquitectura de Software esta ligada íntimamente con la Ingeniería de Software, aquí mostramos los temas que consideramos mas saltantes de ambos y luego la relación existente.

3.2.1 Definiciones de Ingeniería de Software

Desarrollo de sistemas de software cuyo tamaño/complejidad justifica un equipo o equipos de ingenieros. (Construcción realizada por muchas personas de un software con muchas versiones)

El establecimiento y uso de principios de ingeniería válidos (métodos) para obtener sin demasiado gasto, software que es confiable y que trabaja en maquinas reales [Bauer, F. L. Software Engineering Information Processing, 1972 - Mr. Bauer fue uno de los principales organizadores de una conferencia de 1968 de la OTAN para extender el uso del término “ingeniería de software”].

La aplicación de una sistemática, disciplinada, cuantificable metodología para desarrollar, operar y mantener software; que es la aplicación de ingeniería de software. [IEEE Std 610-1990]

Ingeniería de software es la disciplina tecnológica y administrativa preocupada con la producción sistemática y mantenimiento de productos de software que son desarrollados y modificados en el tiempo, y dentro de las estimaciones de costo [Fairley, R. Software Engineering Concepts. New York: McGraw-Hill, 1985]

Ingeniería de software es la disciplina de la ciencia de computación preocupada en el desarrollo de aplicaciones de gran envergadura. Ingeniería de Software no solamente cubre los aspectos técnicos de construir sistemas de software,

también los asuntos de administración, como son la dirección de los equipos de programación, administración de tiempo y preparación de presupuestos.

La definición del SEI para la Ingeniería de Software desde 1990:

- Ingeniería es la aplicación sistemática de conocimiento científico en crear y construir soluciones con el costo adecuado a problemas prácticos al servicio de la humanidad.
- Ingeniería de Software es esta forma de ingeniería que aplica los principios de las ciencias de computación y matemáticas para lograr soluciones a problemas de software con el costo adecuado.

Alcance: Estudio del proceso de software, principios de desarrollo, técnicas y notaciones.

Meta: Producción de software de calidad, entregado en el tiempo pactado, dentro del presupuesto, satisfaciendo los requerimientos de los clientes y las necesidades de los usuarios.

Los ingenieros de software deben poseer un amplio rango de perfiles:

- Matemáticas
- Ciencias de computación
- Economía
- Administración
- Psicología

Por lo tanto, ingeniería de software no debe confundirse con la programación de software, el siguiente cuadro muestra algunas diferencias entre programación de software e ingeniería de software:

Programación de Software <> Ingeniería de Software	
Un solo desarrollador	Equipos de desarrollo Múltiples roles

Aplicaciones de “juguete” o simples	Sistemas complejos
Corto tiempo de vida	Largo, tiempo de vida indefinido
Un solo o pocos tomadores de decisiones Desarrollador = usuario	Múltiples tomadores de decisiones - desarrollador <> usuario - usuario <> cliente
El único sistema de su tipo	Familias de sistemas
Construido desde cero	Reutilización para amortizar costos
Mantenimiento mínimo	Mantenimiento es mayor al 60% del total costo de desarrollo

Tabla 1: Diferencias entre Programación de Software e Ingeniería de Software

La Ingeniería de Software tiene características únicas entre las ingenierías:

- El software es moldeable.
- la construcción de software requiere casi en su totalidad la participación humana.
- El software es intangible
- Los problemas de software son impredeciblemente complejos
- El software depende directamente del hardware
- El software casi siempre también depende de otro software
- Las soluciones de software requieren rigor inusual
- El software tiene naturaleza operacional discontinua

Aspectos económicos y administrativos

Producción de software = desarrollo + mantenimiento

Costos de mantenimiento > 60% de todos los costos de desarrollo

20% correctivo

30% adaptativo

50% perfectivo

Ciclo de vida del desarrollo de software

El ciclo de vida de desarrollo de software por lo regular presenta las siguientes etapas:

Requerimientos

Definición del problema -> Especificación de requerimientos

- Determinar que es exactamente lo que el cliente y el usuario quieren
- Desarrollar un contrato con el cliente
- Especificar que es lo que tiene que hacer el producto de software

Dificultades

- El cliente pide el producto erróneo
- El cliente es un iliterato en informática (computadoras/software)
- Las especificaciones son ambiguas, inconsistentes, incompletas

Verificación

- Revisiones de la especificación con el cliente
- Identificar ambigüedades, inconsistencias, temas incompletos
- verificar la viabilidad, y la posibilidad de realizar los test
- Plan de testeo/aceptación del sistema

Diseño

Especificación de requerimientos -> Diseño

- Alto nivel: descomponer el software en módulos con interfaces
- Detallado: desarrollar especificaciones de módulos (seleccionar algoritmos y tipos de datos)
- Mantener un registro de decisiones de diseño que facilite el seguimiento de los cambios
- Especifica como el producto de software debe realizar sus funciones

Dificultades

- Falta de comunicación entre los diseñadores de los módulos
- El diseño puede ser inconsistente, incompleto, ambiguo

Verificación

- Inspecciones de diseño intensivo para establecer la conformidad de los requerimientos
- Revisar las interacciones de los módulos
- Desarrollar un plan de test de integración

Implementación e integración

Diseño -> Implementación

- Implementar módulos; verificar que ellos encuentren sus especificaciones
- Combinar módulos de acuerdo al diseño
- Especificar como el producto de software hace su trabajo

Dificultades

- errores en la interacción de módulos
- orden de integración puede influenciar la calidad y productividad

Verificación

- revisiones extensivas de código para establecer la conformidad a los requerimientos y el diseño
- Revisar interacciones de módulos
- Desarrollar un plan de testeo de unidades (unit test plan): enfocado en módulos individuales
- Test sobre unidades (test on unit), integración (enfocado sobre interfaces de módulos) y planes de test de aprobación (enfocado sobre requerimientos)

Operación y mantenimiento

Operación->Evolución

- Mantener el software durante/después la operación del usuario
- Determinar el producto esta aun funcionando correctamente

Dificultades

- Diseño rígido
- Falta de documentación
- Cambio continuo de personal

Verificación

- Revisiones extensivas para asegurar cambios sin problemas y actualizaciones a la documentación
- Test para determinar que los cambios han sido implementados correctamente
- Test de regresión: asegurar que no existen cambios inadvertidos que puedan comprometer la funcionalidad del sistema

Principios de ingeniería de software

- Rigor y formalidad
- Separación de asuntos
- Modularidad
- Abstracción
- Anticipación al cambio
- Generalidad
- Escalabilidad
- Orden
- Heterogeneidad

Cualidades del Software (Qualities)

Las cualidades son metas en la práctica de la ingeniería de software

- Cualidades internas vs. Externas
 - Cualidades externas visibles al usuario: Fiabilidad, eficiencia, usabilidad.
 - Cualidades internas son preocupación de los desarrolladores
 - Ayudan a los desarrolladores a alcanzar cualidades externas
 - Verificabilidad, facilidad de mantenimiento, extensibilidad, capacidad de evolución, adaptabilidad
- Cualidades de producto vs. proceso
 - Cualidades del producto tiene que ver con objetos desarrollados
 - Facilidad de mantenimiento, fácil de entender, performance
 - Cualidades del proceso trata con la actividad de desarrollo
 - Los productos son desarrollados a través de procesos
 - Facilidad de mantenimiento, productividad, puntualidad

Algunas cualidades del software son:

- Exactitud
 - Calidad ideal
 - Establecido con respecto a la especificación de requerimientos
 - Absoluto
- Fiabilidad
 - Propiedad estadística
 - Probabilidad de que el software pueda operar como se espera sobre un periodo de tiempo dado
 - Relativo
- Robustez
 - Comportamiento razonable en circunstancias imprevistas
 - Subjetivo
 - Un requerimiento especificado es un asunto de exactitud, un requerimiento no especificado es asunto de robustez

- Usabilidad
 - Habilidad de los usuarios finales para usar fácilmente el software
 - Extremamente subjetivo
- Comprensibilidad
 - Habilidad de los desarrolladores para fácilmente entender los objetos producidos
 - Calidad interna del producto
 - Subjetivo
- Verificabilidad
 - Fácil de establecer las propiedades deseadas
 - Realizados en análisis formales o pruebas
 - Calidad interna
- Performance
 - Equivalente con eficiencia
 - Tasable por medidas, análisis y simulación
- Capacidad de evolucionar
 - Habilidad para adicionar o modificar funcionalidad
 - Direcciona los mantenimientos adaptativos y perfectivos
 - Problema: la evolución de la implementación es muy fácil
 - Evolución debe iniciar con requerimientos o diseño
- Reusabilidad
 - Habilidad de construir nuevo software desde piezas existentes
 - Debe ser planeado para reusabilidad
 - Ocurre en todos los niveles: desde personas a procesos, desde requerimientos hasta código
- Interoperabilidad
 - Habilidad de (sub)sistemas de software para cooperar con otros

- Fácilmente integrable con sistemas grandes
- Las técnicas comunes incluyen APIs, protocolos plug-in, etc.
- Escalabilidad
 - Habilidad de los sistemas de software para crecer en tamaño mientras se mantienen sus propiedades y cualidades
 - Asume la capacidad de mantenimiento y la capacidad de evolucionar
 - Meta del desarrollo basado en componentes
- Heterogeneidad
 - habilidad para componer un sistema desde las piezas desarrolladas en múltiples lenguajes de programación, sobre múltiples plataformas, por múltiples desarrolladores, etc.
 - Necesitado para volver a utilizar
 - Meta del desarrollo basado en componentes
- Portabilidad
 - Habilidad para ejecutar en nuevos ambientes con el menor esfuerzo
 - Puede ser planeado para este fin aislando los componentes dependientes del ambiente
 - Precisado por el surgimiento de sistemas altamente distribuidos (Ej. Internet)
 - Un aspecto de heterogeneidad

Evaluando las cualidades del software

- Las cualidades deben ser medibles
- La medición requiere que las cualidades sean precisamente definidas
- La mejora requiere mediciones acertadas
- Actualmente la mayor parte de las cualidades son informalmente definidas y son difíciles de evaluar

Proceso de Software

El proceso es un dialogo en el que se reúne conocimiento y se incluye en el software para convertirse en software. El proceso proporciona una interacción entre los usuarios y los diseñadores, entre los usuarios y las herramientas de desarrollo, y entre los diseñadores y las herramientas de desarrollo. Es un proceso interactivo donde la herramienta de desarrollo se usa como medio de comunicación, con cada iteración del dialogo se obtiene mayor conocimiento de las personas involucradas.

Cualidades del proceso de software

- El proceso es **confiable** si este consistentemente induce a productos de alta calidad
- El proceso es **robusto** si este puede acomodarse de manera imprevista a cambios de herramientas y ambientes
- El **performance** del proceso es la productividad
- El proceso es **evolutivo** si esta puede acomodar nuevas técnicas de gestión y organizacionales
- El proceso es **re-usable** si puede ser aplicado a través de proyectos y organizaciones

Axiomas de la Ingeniería de Software

- Adicionando desarrolladores a un proyecto puede resultar en un mayor atraso y costos acumulados
- Tensión básica de la ingeniería de software
 - Mejor, barato, rápido – debe elegir dos
 - Funcionalidad, escalabilidad, performance – debe elegir dos
- Mientras mas tiempo exista una falla
 - Es mas costoso detectar y corregir
 - Es menos probable que sea correctamente corregido
- Cerca del 70% de fallas detectadas en proyectos de software de gran escala son introducidos en las etapas de requerimientos y diseño

Detectar las causas de estas fallas tempranamente puede reducir los costos resultantes por un factor de 100 o más.

3.2.2 Definiciones de Arquitectura de software

Por muchos años, los ingenieros de software han estado empleando arquitecturas de software sin conocerlas. Los orígenes de la arquitectura de software yacen en problemas encontrados e identificados por investigadores y prácticos:

- Dificultades esenciales de la ingeniería de software

- Complejidad
- Cambio constante

Estos problemas se han tratado de solucionar con herramientas y técnicas como lenguajes de alto nivel, programación orientada a objetos, verificación de programas, etc.

- Características únicas de la programación de software complejo o muy grande

Se requiere estructurar grandes colecciones de módulos para construir el sistema. En la programación de software complejo las dinámicas interpersonales se convierten en críticas.

- Necesidad de la reutilización de software

La reutilización de software reduce el tiempo de desarrollo y costo, además de mejora la fiabilidad y las características.

La dificultad técnica es que no siempre existen componentes identificables, además de que en algunas ocasiones los componentes pueden ser muy toscos o muy finos. Los componentes pueden no proveer la exacta funcionalidad necesitada.

Los orígenes de la arquitectura de software también yacen en soluciones desarrolladas para tratar con estos problemas.

No hay una definición Standard o universalmente aceptada del termino, “Arquitectura de software” a pesar de que no existe escasez de definiciones tampoco. Las siguientes secciones intentan realizar una definición común tomando como base recientes trabajos de Arquitectura de Software. La primera sección incluye definiciones tomadas de trabajos mas recientes de arquitectura de software. La segunda sección provee las “definiciones clásicas”, un breve tour a través de las más importantes o que más han influido.

Definiciones modernas

Del libro Arquitectura de Software en Practica (2da Edición) (Bass, Clements, Kazman, Addison-Wesley 2003)

La arquitectura de software de un programa o sistema de cómputo es la estructura o estructuras del sistema que comprende elementos de software, las propiedades visibles externamente de estos elementos, y las relaciones entre ellos. Propiedades “Externamente visibles” se refiere a la suposición de que otros elementos pueden hacer de un elemento, como son servicios provistos, características de performance, manejo de errores, manejo de recursos compartidos, y mas. Vamos a ver algunas de las implicaciones de esta definición en más detalle.

Primero, arquitectura define elementos. La arquitectura incorpora información acerca de cómo los elementos se relacionan con el otro. Esto significa que la arquitectura específicamente omite cierta información acerca de los elementos que no se relaciona con sus interacciones. De esa manera, una arquitectura es primero una abstracción de un sistema que suprime detalles de elementos que no afectan como ellos usan, son usados por, se relacionan a, interactúan con otros elementos. En casi todos los sistemas modernos, los elementos interactúan con el otro por medio de las interfaces que ocultan detalles acerca de un elemento dentro de partes públicas y privadas. La arquitectura se preocupa con el lado público de esta división, los detalles privados de los elementos –detalles de implementación interna – no son de arquitectura.

Segundo, la definición deja claro que los sistemas pueden y deben comprender más de una estructura y ninguna estructura puede reclamar ser de manera irrefutable la arquitectura. Por ejemplo, todos los proyectos no triviales son divididos en unidades de implementación, a estas unidades son dadas responsabilidades específicas, y son la base de la asignación de trabajo para los equipos de programación. Este tipo de elementos puede comprender programas y datos que el software en otras unidades de implementación puede llamar o acceder, y los programas y datos son privados. En proyectos largos, los elementos pueden casi ser subdivididos por asignaciones a sub equipos.

Esto es un tipo de estructura algunas veces usados para describir un sistema. Esta es una estructura muy estática, en que esta enfocada sobre la manera en que la funcionalidad del sistema es dividido y asignado a los equipos de implementación.

Otras estructuras son mucho mas enfocadas sobre la manera en que los elementos interactúan con cada otra en tiempo de ejecución para llevar la función del sistema. Suponga que el sistema esta siendo construido como un conjunto paralelo de procesos. El conjunto de procesos que pueden existir en tiempo de ejecución, los programas en las varias unidades de implementación descritas previamente que son enlazadas previamente para formar cada proceso, y las relaciones de sincronización a lo largo de la forma de procesos algunas veces usadas para describir un sistema.

Tercero, la definición implica que cada sistema de software tiene una arquitectura por que cada sistema puede ser mostrado como un conjunto de elementos y relaciones en medio de ellos. En el caso más trivial un sistema es un solo elemento –poco interesante, y probablemente una arquitectura poco usable, pero una arquitectura a pesar de todo. Aun cuando cada sistema tiene una arquitectura este no significa necesariamente que la arquitectura es conocida para cualquiera. Desafortunadamente, una arquitectura puede existir independientemente de su descripción o especificación, que encumbra la

importancia de la documentación de la arquitectura y la reconstrucción de la arquitectura.

Cuarto, el comportamiento de cada elemento es parte de la arquitectura a tal punto como el comportamiento puede ser observado o juzgado desde el punto de vista de otro elemento. Este comportamiento es el que permite a los elementos interactuar con cada otro, que es una parte clara de la arquitectura. Esto no significa que el comportamiento exacto y la performance de cada elemento debe ser documentado en todas las circunstancias; pero para el alcance que de cómo el comportamiento de un elemento influencia en como otro elemento debe ser escrito para interactuar con este elemento o influencia la aceptabilidad del sistema como un todo, este comportamiento es parte de la arquitectura de software.

Finalmente, la definición es diferente como a si la arquitectura de un sistema es buena o mala, significando que la arquitectura puede permitir o prevenir el sistema desde encontrar su comportamiento, performance, y requerimientos de ciclo de vida. Asumiendo que nosotros no aceptamos prueba y error como la mejor forma para escoger la arquitectura para un sistema – esto es escogiendo una arquitectura al azar, construyendo el sistema desde este, y esperando para este lo mejor – esto le da importancia a la evaluación de la arquitectura.

ANSI/IEEE Std 1471-2000, Prácticas recomendadas para descripciones arquitectónicas de Sistemas de Software

Arquitectura es definida por la práctica recomendada como la organización fundamental de un sistema, que esta encarnado en sus componentes, sus relaciones con cada otro y el ambiente, y los principios que gobiernan su diseño y evolución. Esta definición tiene la intención para abarcar una variedad de usos del término arquitectura reconociendo sus fundamentales elementos comunes. Lo principal en medio de esta necesidad para comprender y controlar estos elementos del diseño de un sistema que captura la utilidad de un sistema, costo y riesgo. En algunos casos, estos elementos son componentes físicos del sistema y sus relaciones. En otros casos, estos elementos no son

físicos, pero en lugar de eso, componentes lógicos. En algunos otros casos, estos elementos soportan principios o patrones que crean estructuras organizacionales perdurables. Esta definición tiene la intención de abarcar los distintos, pero relacionados usos, mientras promueve una definición más rigurosa que constituye la organización fundamental de un sistema dentro de un dominio particular.

Definiciones clásicas

Booch, Rumbaugh, and Jacobson, 1999

Una arquitectura es un conjunto de decisiones significantes acerca de la organización de un sistema de software, la selección de los elementos estructurales y sus interfaces por los que el sistema esta compuesto, junto con su comportamiento como se especifica en la colaboración a lo largo de esos elementos dentro de subsistemas progresivamente largos, y el estilo arquitectónico que guía la organización, estos elementos e interfaces, sus colaboraciones y su composición.

Boehm, et al., 1995

Barry Boehm y sus estudiantes en el USC Center of Software Engineering escribieron que:

Una arquitectura de software comprende de:

- Una colección de software y componentes del sistema, conexiones y restricciones.
- Una colección de evaluaciones de riesgo necesarias
- Un argumento que demuestra que los componentes, conexiones y restricciones que si se implementa, pueden satisfacer la colección de evaluaciones de riesgo necesarias.

Soni, Nord y Hofmeister, 1995

Soni, Nord y Hofmeister de Siemens Corporate Research escribieron que, basado en las estructuras encontradas que prevalecen e influyen en un

ambiente de desarrollo de proyectos industriales que ellos estudiaron, la arquitectura de software tiene como mínimo 4 diferentes encarnaciones:

Dentro de cada categoría, las estructuras describen el sistema desde una diferente perspectiva:

- **La arquitectura conceptual** describe el sistema en términos de sus mayores elementos de diseño y sus relaciones entre ellos.
- **La arquitectura de interconexión** de módulos abarca dos estructuras ortogonales: descomposición funcional y capas.
- **La arquitectura de ejecución** describe la estructura dinámica de un sistema
- **La arquitectura de código** describe como el código fuente, archivos binarios, y librerías son organizadas en el ambiente de desarrollo.

Bredemeyer Consulting

Arquitectura de software es la estructura de nivel superior de un sistema de Software. Las propiedades importantes incluyen:

- Tiene un suficientemente amplio nivel de abstracción para que el sistema pueda ser visto como un todo
- La estructura debe soportar la funcionalidad requerida del sistema. De esta manera el comportamiento dinámico del sistema debe ser tomado en cuenta cuando se diseña la arquitectura.
- La estructura o arquitectura debe ser conforme a las cualidades del sistema (también conocido como requerimientos no funcionales). Estos incluyen requerimientos de performance, seguridad y confiabilidad asociadas con la funcionalidad actual, como con los requerimientos de flexibilidad o extensibilidad asociados cuando se acomoda la futura funcionalidad a un razonable costo de cambio. Estos requerimientos pueden estar en conflicto, y las alternativas de cambio son una parte esencial al diseñar una arquitectura.
- En el nivel de arquitectura todos los detalles de implementación están ocultos.

La Arquitectura de software es comúnmente definida en términos de componentes y conectores. Componentes se identifican y se les asigna responsabilidades, los componentes clientes actúan a través de interfaces. La interconexión de componentes especifica la comunicación y mecanismo de control, y soporta toda la interacción necesaria para lograr el funcionamiento del sistema.

Cuando creamos arquitecturas de software estamos preocupados en:

- **meta-arquitectura:** la visión arquitectónica, estilo, principios, comunicación clave y mecanismos de control, y conceptos que guíen al equipo de arquitectos en la creación de una arquitectura.
- **Vista arquitectónica:** Tal como la construcción de arquitecturas son mejor visualizadas en términos de un numero complementario de vistas o modelos, también lo son las arquitecturas de software. En particular, las vistas arquitectónicas (structural views) ayudan a documentar y comunicar la arquitectura en términos de componentes y sus relaciones, y son muy útiles en la evaluación de cualidades arquitectónicas como extensibilidad. Las vistas de comportamiento son útiles cuando se piensa en función de cómo los componentes interactúan para cumplir sus responsabilidades asignadas y evaluar el impacto de escenarios what-if sobre la arquitectura. Los modelos de comportamiento son especialmente útiles evaluando cualidades en tiempo de ejecución como performance y seguridad. Las vistas de ejecución (Execution views) ayudan en la evaluación de opciones de la distribución física y la documentación y comunicación de decisiones.
- **Patrones arquitectónicos** (architectural patterns): patrones estructurales como capas y cliente/servidor, y mecanismos como brokers y bridges.

- Los principios básicos de el diseño arquitectónico incluyen abstracción, separación de preocupaciones, posponer decisiones, y simplicidad, y técnicas relacionadas como ocultar interfaces y encapsulamiento.
- Principios de descomposición del sistema y buen diseño de interfaz de usuario.

3.2.3 Que no es la Arquitectura de software

Debe ser distinguida del diseño de bajo nivel (Ej. Diseño de componentes internos y algoritmos) y implementación, por un lado, y otro tipo de arquitecturas relacionadas por otro lado. Por ejemplo, arquitectura de software no es el modelo de información (datos), aunque este usa el modelo de información para obtener el tipo de información de los métodos en las interfaces. No es la arquitectura del modelo físico, incluyendo procesadores, redes, y parecidos en que el software debe correr. Sin embargo, usa esa información en la evaluación del impacto de elecciones arquitectónicas sobre la calidad del sistema como performance y confiabilidad. Más obviamente, quizás, no es tampoco la arquitectura de hardware de un producto a ser manufacturado. Mientras, cada una de estas arquitecturas tiene sus propios especialistas liderando sus diseños, estas arquitecturas impactan y son impactadas por la arquitectura de software, y donde sea posible, no deben ser diseñados aislados uno del otro. Este es el dominio del sistema arquitectónico.

La Arquitectura de Software de una forma sencilla es un diseño del sistema de Software, que:

- Maneja la complejidad
- Incrementa el potencial de reutilización y el potencial mercado de componentes.
- Abarca métodos formales

	Entrega	No Entrega
Arquitectura	- Requerimientos no funcionales - Alcance funcional y responsabilidades - Diseño clave y elecciones de productos - Diseño de alto nivel - Restricciones de diseño	- Prototipos - Requerimientos funcionales comprensivos o explicados ampliamente - Detallado análisis de datos - Sistemas construidos e implementados
Diseño	- Requerimientos funcionales y como ellos pueden ser conseguidos - Detallado Análisis de datos - Documentación del diseño del sistema - Sistemas construidos e implementados	- Visión de la Solución - Comprensivos y rastreables requerimientos no funcionales - Arquitectura de seguridad y gobierno

Tabla 2: Comparando Arquitectura y Diseño

3.2.4 Arquitectura de Software en el Contexto

¿Donde la Arquitectura de Software encaja en el proceso de desarrollo?

El diagrama siguiente muestra las actividades de arquitectura en el contexto de una simple aplicación o proceso de desarrollo del producto.

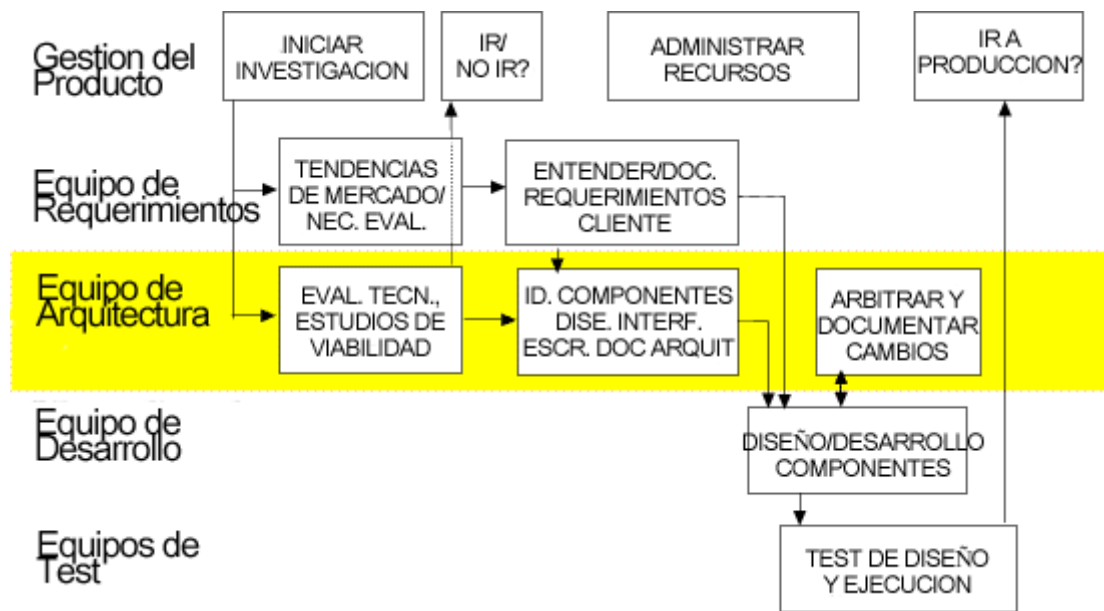


Ilustración 1: Arquitectura de Software en el Contexto

Más y más productos están siendo desarrollados incrementalmente – con múltiples revisiones, y cada revisión, múltiples ciclos de desarrollo. Esto es también conocido como desarrollo evolutivo. En este caso, una fase arquitectónica es un importante precursor para ciclos evolutivos, esto permite al sistema ser dividido en partes que son construidos incrementalmente. Un paso de actualización de arquitectura es una parte importante de cada pequeño ciclo de desarrollo, permitiendo aprender acerca de los requerimientos y el ajuste arquitectónico a ser incorporado dentro de la arquitectura.

Es importante también clasificar la Arquitectura de Software en diferentes niveles:

Arquitectura Empresarial: Definiendo toda la forma y función de los sistemas (negocios y IT) en la empresa (incluyendo socios y organizaciones que forman la extensión de la empresa), y proveyendo marcos de trabajo, estándares y guías para arquitecturas a nivel de proyecto. La arquitectura provista por la arquitectura a nivel empresarial permite el desarrollo de sistemas consistentes y apropiados en la empresa con la habilidad de trabajar juntos, colaborar e integrar donde y cuando es requerido.

Arquitectura a Nivel de Proyecto: Define la forma y función de los sistemas (negocios y IT) en un proyecto o programa, dentro del contexto de la empresa como un todo y no solo como individuales sistemas aislados. Este nivel de arquitectura de arquitectura a nivel de proyecto puede refinar, amoldarse y trabajar dentro de la definida Arquitectura Empresarial.

Arquitectura de Aplicación: Define la forma y función de las aplicaciones que pueden ser desarrolladas para entregar la funcionalidad requerida del sistema. Algo de esta arquitectura puede ser definida en la Arquitectura Empresarial y Arquitectura a Nivel de Proyecto (como estándares y guías) para asegurar mejores practicas y concordancia con la arquitectura completa.

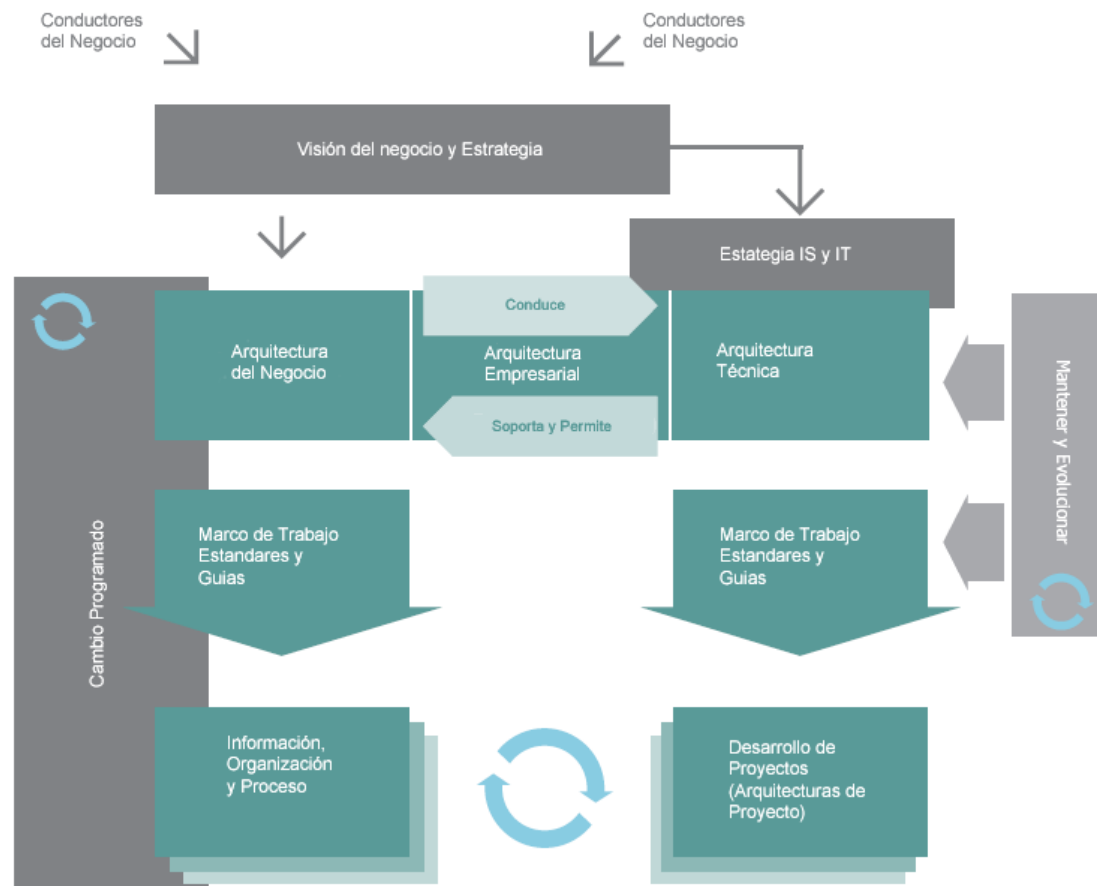


Ilustración 2: Alineamiento de Sistemas y Negocios

3.2.5 Motivación de la arquitectura de software (¿Por qué necesitamos arquitecturas de software?)

Una arquitectura exitosa forma la plataforma para la ventaja estratégica. En contraste, la falta de arquitectura ata a la organización inevitablemente al pasado. Nuestra herencia es una tortuosa trampa de sistemas casuales nacidos en un tiempo de maravillosa brujería pero de poca disciplina de sistemas. Estos sistemas heredados son caros y difíciles de cambiar, pero remplazarlos amenaza la misma vida de la organización.

Los frágiles sistemas monolíticos, y los largos e impredecibles tiempos de desarrollo, son síntomas de decaimiento arquitectónico que causa gran obstrucción al progreso organizacional. Para construir sistemas que encajen en

el ambiente, y adaptarnos a los cambios en el ambiente nosotros necesitamos arquitectura.

Si nosotros buscamos liderar a través de innovación, acercamiento al cliente o excelencia operacional, arquitectura es el fundamento esencial para agilidad, respuesta y efectividad. La Arquitectura direcciona la complejidad, dejando a la mente del equipo espacio para la innovación. Esto posibilita sistemas confiables desarrollados en “Internet time”, sistemas de eCommerce que escalan, y sistemas CRM que agradan a los clientes con experiencia personalizada.

La arquitectura sirve propósitos técnicos y organizacionales. En el lado de la organización, la arquitectura nos ayuda a:

- **Comunicar el diseño de alto-nivel:** Un número de tomadores de decisiones, necesitan entender el sistema de manera simple. Esto incluye ejecutivo de alto nivel, mucho del equipo funcional (Ej. Marketing, aseguramiento de calidad, y productos de aprendizaje o documentación de usuario), y puede incluir clientes también. Modelar el sistema en un nivel alto facilita la comunicación del diseño del sistema de alto nivel o arquitectura. La reducción del detalle hace fácil de captar la asignación de responsabilidades del sistema a las estructuras de alto nivel. Además, esto satisface las restricciones que, aunque aparentemente sean triviales, tienen importantes implicaciones de comunicación – con adecuados niveles y anidamientos, aun largos y complejas arquitecturas pueden ser transmitidas de manera sencilla.
- **Proveer el contexto del sistema:** Los desarrolladores (y el soporte futuro) también necesitan entender el sistema de manera genérica. En sistemas grandes, los desarrolladores no pueden entender eficazmente los detalles del sistema completo. Ellos necesitan un entendimiento detallado de la parte del sistema en que ellos van a trabajar. Pero sin entender las responsabilidades e interdependencias de las estructuras

de alto nivel, el desarrollo individual y la optimización de las subestructuras puede tender en un sistema poco óptimo.

- **Asignación del trabajo:** Donde la arquitectura descompone el sistema en subestructuras que son relativamente independientes, tienen responsabilidades claras, y se comunican unas con otras a través de un limitado número de interfaces bien definidas, el trabajo de desarrollo puede ser particionado de manera efectiva. Esto permite el trabajo de desarrollo paralelo para proceder con relativa independencia en los puntos de integración. Esto es especialmente importante en proyectos largos, o proyectos donde los equipos están geográficamente dispersos o subcontratistas son usados. El diseño de la arquitectura del sistema puede ser visto como el par del diseño de la arquitectura de la organización. Si esta dualidad es ignorada y la arquitectura organizacional no es compatible con la arquitectura del sistema, entonces esta puede influenciar y degradar la arquitectura del sistema.

En el lado técnico, la arquitectura nos permite diseñar mejores sistemas:

- **Encontrar los requerimientos y objetivos del sistema:** Los requerimientos funcionales y no funcionales pueden ser priorizados como “debe tener” vs. “altamente requerido” vs. “requerido”, donde “debe tener” identifica propiedades que el sistema debe tener para ser aceptable. Una arquitectura nos permite evaluar y hacer trueques en medio de requerimientos de diferente prioridad. Sin embargo las cualidades del sistema (también conocidos como requerimientos no funcionales) pueden ser comprometidos después en el proceso de desarrollo, muchos pueden no ser encontrados si no son explícitamente tomados en cuenta en el nivel arquitectónico.
- **Permitir distribución flexible/partición del sistema:** Una buena arquitectura permite una distribución flexible del sistema permitiendo al sistema y sus aplicaciones constituyentes a ser particionadas entre diferentes procesadores en muchas maneras diferentes sin tener que rediseñar las partes componentes.

- **Reducir los costos de mantenimiento y evolución:** Arquitectura puede ayudar a minimizar los costos de mantener y evolucionar un determinado sistema sobre su completo tiempo de vida anticipando los diferentes tipos de cambios que pueden ocurrir en el sistema, asegurando que todo el diseño del sistema pueda facilitar dichos cambios, y localizar tanto como sea posible los efectos de dichos cambios en los documentos de diseño, código, y otros productos de trabajo con el sistema. Esto puede ser conseguido por la minimización y control de las interdependencias de subsistemas.
- **Incrementar, rehusar e integrar con software heredado y de terceros:** Una arquitectura puede ser diseñada para permitir y facilitar la (re)utilización de ciertos componentes existentes, frameworks, librerías de clases, heredados o aplicaciones de terceros, etc.

3.2.6 Arquitecto de Software

El rol de Arquitecto de software es responsable de la arquitectura de software, que incluyen las decisiones técnicas claves que contienen el diseño completo e implementación para el proyecto.

La meta principal del arquitecto es asegurar el éxito completo del proyecto diseñando los fundamentos de la aplicación. Esto incluye la definición de la estructura organizacional de la aplicación y la estructura física de su distribución.

En estos propósitos, la meta del arquitecto es reducir la complejidad dividiendo el sistema en limpios y simples partes. La arquitectura resultante es extremadamente importante desde que no solo dicta de cómo el sistema debe ser construido al ir avanzando, también establece si la aplicación debe exhibir los muchos rasgos que son esenciales para el éxito del proyecto. Esto incluye su usabilidad, si este es confiable y mantenible, si encuentra los estándares de performance y seguridad y si puede evolucionar fácilmente al frente de los requerimientos cambiantes.

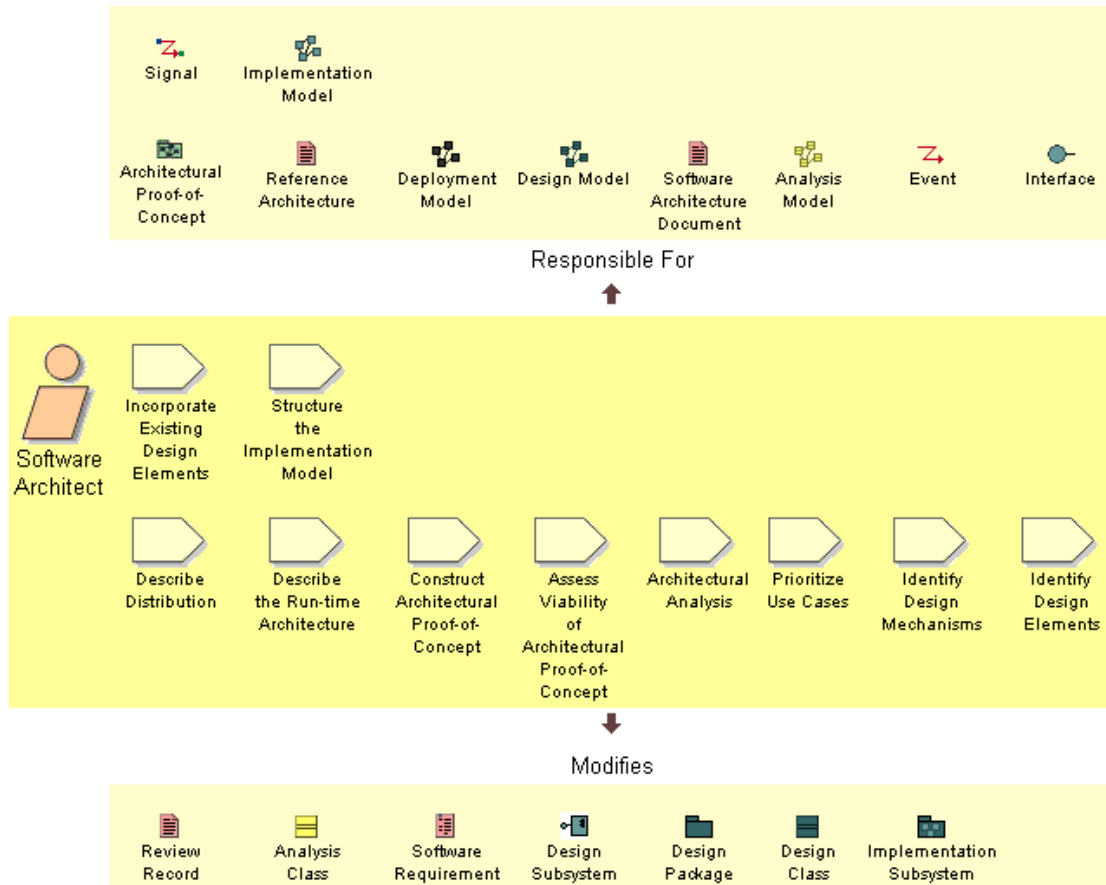


Ilustración 3: Artefactos y Responsabilidades del Arquitecto según RUP

Descripción

El arquitecto de software tiene la completa responsabilidad de manejar las mayores decisiones técnicas, expresados como la Arquitectura de Software. Esto típicamente incluye identificar y documentar los aspectos arquitectónicos significantes del sistema, incluyendo requerimientos, diseño, implementación, y “vistas” de despliegue del sistema.

El arquitecto es también responsable de proveer los fundamentos de estas decisiones, cubriendo las preocupaciones de los propietarios de la aplicación (stakeholders), eliminando riesgos técnicos, y asegurando que las decisiones sean efectivamente comunicadas, validadas y seguidas.

Perfil

“El arquitecto ideal debe ser una persona letrada, un matemático, familiarizado con estudios históricos, un estudiante diligente de la filosofía, al corriente con la música, no ignorante de la medicina, conocedor de la legislación civil, familiarizado con la astronomía y los cálculos astronómicos.” – Vitruvius, circa 25 AC.

En resumen, el arquitecto de software debe poseer madurez, visión y profunda experiencia que le permita captar los temas rápidamente y en forma educada, con juicio crítico en ausencia completa de información. Más específicamente, el arquitecto de software, o miembros del equipo de arquitectura, deben combinar estos perfiles:

- **Experiencia:** en ambos el problema de competencias, a través de un profundo entendimiento de los requerimientos, y el dominio de la ingeniería de software. Si hay un equipo, estas cualidades pueden ser obtenidas de los miembros del equipo, pero al menos un arquitecto de software debe proveer la visión global para el proyecto.
- **Liderazgo:** con el propósito de manejar el esfuerzo técnico entre varios equipos, y tomar las decisiones críticas bajo presión y hacer que estas decisiones se respeten. Para ser efectivo, el arquitecto de software y el administrador del proyecto deben trabajar cerradamente juntos, con el arquitecto de software conduciendo los asuntos técnicos y el administrador del proyecto conduciendo los asuntos administrativos. El arquitecto de software debe tener la autoridad para tomar las decisiones técnicas.
- **Comunicación:** para ganar confianza, para persuadir, para motivar y para guiar. El arquitecto de software no puede liderar por decretos, solo por el consenso del resto del proyecto. Para ser efectivos, el arquitecto de software debe ganar respeto del equipo del proyecto, el administrador del proyecto, el cliente y la comunidad usuaria, también del equipo de administración.

- Orientación a metas y proactividad: con un implacable enfoque a resultados. El arquitecto de software es la fuerza de conducción detrás del proyecto, no un visionario o un soñador. La carrera de un exitoso arquitecto de software es una larga serie de decisiones subóptimas hechas en la incertidumbre y bajo presión. Solo los que pueden enfocarse en hacer que las necesidades sean cubiertas con éxito en el ambiente del proyecto.

Desde un punto de vista profesional, el arquitecto de software también necesita abarcar las capacidades del ROL Diseñador (Responsable de diseñar una parte del sistema dentro de las restricciones de los requerimientos, arquitectura, y proceso de desarrollo para el proyecto). Sin embargo, de manera diferente que el diseñador, el arquitecto de software:

- Tiende a ser generalista más que especialista, conociendo muchas tecnologías a alto nivel más que pocas tecnologías en el nivel de detalle.
- Tomar extensas decisiones técnicas, y por otro lado amplio conocimiento y experiencia, como también perfiles de comunicación y liderazgo, son la clave.

Responsabilidades

Una lista típica de responsabilidades de un Arquitecto de Software es:

- Soportar la visión del negocio y proponer actividades para alcanzarlas
- Traducir entre requerimientos de negocios y de IT
- Comunicarse con los tomadores de decisiones, del negocio y de IT
- Evaluar los diferentes intereses
- Determinar las alternativas de solución
- Crear un viable y factible diseño
- Escoger soluciones
- Administrar cualidad y complejidad
- Mitigar riesgos
- Comunicar

Estrategia de asignación del rol

Si el equipo es lo suficientemente grande para garantizar un equipo de arquitectura, la meta es tener una buena mezcla de talentos, cubriendo un amplio espectro de experiencia y compartiendo un común entendimiento de procesos de ingeniería de software. El equipo de arquitectura no debe ser un comité de representantes de varios equipos, áreas o contratistas. El Arquitecto de software es una función de tiempo completo, con personal permanentemente dedicado a esto.

Para pequeños proyectos, una sola persona puede actuar como administrador del proyecto y arquitecto de software. Sin embargo, si es posible del todo, es mejor tener estos roles realizados por personas diferentes, a fin de asegurar que la presión de tiempo de un rol no cause que el otro rol sea descuidado.

3.2.7 Relación entre la Ingeniería de Software y la Arquitectura de Software

La Arquitectura de Software brinda a la Ingeniería de Software el diseño arquitectónico que guíe el desarrollo de un sistema. Tener una arquitectura robusta facilita el desarrollo en paralelo, minimiza la repetición de trabajos e incrementa la probabilidad de reutilización de componentes y el mantenimiento posterior del sistema. Este diseño arquitectónico sirve como una base sólida sobre la cual se puede planificar y manejar el desarrollo de software.

4. METODOLOGIA DE LA INVESTIGACION

4.1. Estado del arte: Aporte al conocimiento

El aporte de esta tesina al conocimiento es mostrar la elaboración de la Arquitectura de Software de una aplicación con la utilización de una metodología que no sea burocrática y que tenga tan solo el proceso suficiente como para que el esfuerzo en plantearla sea de ayuda.

El desarrollo de software es una actividad caótica, frecuentemente caracterizada por la frase "codifica y corrige". El software muchas veces se escribe con un mínimo un plan subyacente, y el diseño del sistema se adoquina con muchas decisiones a corto plazo. Esto realmente funciona muy bien si el sistema es pequeño, pero conforme el sistema crece llega a ser cada vez más difícil agregar nuevos aspectos al mismo. Además los errores (bugs) llegan a ser cada vez más frecuentes y más difíciles de corregir. La seña típica de tal problema es una larga fase de pruebas después de que el sistema ha sido "completado". Tal fase larga de pruebas hace estragos con los planes de pruebas y depurado llegando a ser imposible de ponerlo en el cronograma de trabajo.

Hemos tenido una alternativa desde hace mucho: **Metodología**. Las metodologías imponen un proceso disciplinado sobre el desarrollo de software con el fin de hacerlo más predecible y eficiente. Lo hacen desarrollando un proceso detallado con un fuerte énfasis en planificar inspirado por otras disciplinas de la ingeniería. Usaremos la metodología para encontrar el contexto en que debe crearse la arquitectura de nuestro sistema de software.

4.2. Parte central de la tesina

En el desarrollo de software las metodologías ofrecen la estrategia para desarrollar el proceso del software. Este modelo es importante e influye en gran manera al desarrollo de la Arquitectura de Software, ya sea por la cantidad de documentos que debe producir, el tiempo y las personas que participaran y los cambios que se realizaran conforme el proceso avance, por ello antes de desarrollar la Arquitectura vamos a seleccionar una metodología que nos permita trabajar de forma adecuada. Debe observar que la decisión en este caso depende de una serie de variables como el numero de personas que participan en el desarrollo (Asumimos un equipo de 2 personas), el tiempo, y la cantidad de ceremonia requerida.

4.2.1 ALTERNATIVAS METODOLOGICAS

Las metodologías ingenieriles han estado presentes durante mucho tiempo en el desarrollo de Software. No se han distinguido precisamente por ser muy exitosas. La crítica más frecuente a estas metodologías es que son burocráticas. Hay tanto que hacer para seguir la metodología que el ritmo entero del desarrollo se retarda.

Como una reacción a estas metodologías, un nuevo grupo de metodologías ha surgido en los últimos años: metodologías ágiles. El encanto de estas metodologías ágiles es su reacción a la burocracia de las metodologías monumentales. Estos nuevos métodos buscan un justo medio entre ningún proceso y demasiado proceso, proporcionando simplemente suficiente proceso para que el esfuerzo valga la pena.

El resultado de todo esto es que los métodos ágiles cambian significativamente algunos de los énfasis de los métodos ingenieriles. La diferencia inmediata es que son menos orientados al documento, exigiendo una cantidad más pequeña de documentación para una tarea dada. En muchas maneras son más bien

orientados al código: siguiendo un camino que dice que la parte importante de la documentación es el código fuente.

Sin embargo esto puede no ser el punto importante sobre los métodos ágiles. La falta de documentación es un síntoma de diferencias mucho más profundas:

- Los métodos ágiles son adaptables en lugar de predictivos. Los métodos ingenieriles tienden a intentar planear una parte grande del proceso del software en gran detalle para un plazo grande de tiempo, esto funciona bien hasta que las cosas cambian. Así que su naturaleza es resistirse al cambio. Para los métodos ágiles, no obstante, el cambio es bienvenido. Intentan ser procesos que se adaptan y crecen en el cambio, incluso al punto de cambiarse ellos mismos.
- Los métodos ágiles son orientados a la gente y no orientados al proceso. La meta de los métodos ingenieriles es definir un proceso que funcionará bien con cualquiera que lo use. Los métodos ágiles afirman que ningún proceso podrá nunca maquillar las habilidades del equipo de desarrollo, de modo que el papel del proceso es apoyar al equipo de desarrollo en su trabajo. Explícitamente puntualizan el trabajar a favor de la naturaleza humana y enfatizan que el desarrollo de software debe ser una actividad agradable.

4.2.2 Predictivo contra Adaptable

Separación de Diseño y Construcción

La inspiración usual para las metodologías han sido disciplinas como las ingenierías civil o mecánica. Tales disciplinas enfatizan que hay que planear antes de construir. Los ingenieros trabajan sobre una serie de dibujos que indican precisamente qué hay que construir y como deben juntarse estas cosas. Los dibujos se entregan entonces a un grupo diferente, a menudo una compañía diferente, para ser construidos. Se supone que el proceso de la construcción seguirá los dibujos. En la práctica los constructores se encuentran con algunos problemas, pero éstos son normalmente poco importantes.

Como los dibujos especifican las piezas y cómo deben unirse, actúan como los fundamentos de un plan de construcción detallado. Dicho plan define las tareas que necesitan hacerse y las dependencias que existen entre estas tareas. Esto permite un plan de trabajo y un presupuesto de construcción razonablemente predecibles. También dice en detalle cómo deben hacer su trabajo las personas que participan en la construcción. Esto permite que la construcción requiera menos pericia intelectual, aunque se necesita a menudo mucha habilidad manual.

Así que lo que vemos aquí son dos actividades fundamentalmente diferentes. El diseño, que es difícil de predecir y requiere personal caro y creativo, y la construcción que es más fácil de predecir. Una vez que tenemos el diseño, podemos planear la construcción. Una vez que tenemos el plan de construcción, podemos ocuparnos de la construcción de una manera más predecible. En ingeniería civil la construcción es mucho más costosa y tardada que el diseño y la planeación.

Así el enfoque de muchas metodologías es: queremos un plan de trabajo **predecible** que pueda usar gente del más bajo nivel. Para hacerlo debemos separar el plan de la construcción. Por consiguiente necesitamos entender cómo hacer el diseño de software de modo que la construcción pueda ser sencilla una vez que el plan esté hecho.

¿Qué forma toma este plan? Para muchos, éste es el papel de notaciones de diseño como el UML. Si podemos hacer todas las decisiones significativas usando UML, podemos armar un plan de construcción y entonces podemos dar planes a los programadores como una actividad de construcción.

Pero aquí surgen preguntas cruciales. ¿Es posible armar un plan que sea capaz de convertir el código en una actividad de construcción predecible? Y en tal caso, ¿es la construcción suficientemente grande en costo y tiempo para hacer valer la pena este enfoque?

Todo esto trae a la mente más preguntas. La primera es la cuestión de cuán difícil es conseguir un diseño UML en un estado que pueda entregarse a los programadores. El problema con un diseño tipo UML es que puede parecer muy bueno en el papel, pero resultar seriamente fallido a la hora de la programación. Los modelos que los ingenieros civiles usan están basados en muchos años de práctica guardados en códigos ingenieriles. Además los problemas importantes, como el modo en que juegan las fuerzas, son dóciles al análisis matemático. La única verificación que podemos hacer con los diagramas UML es la revisión cuidadosa. Mientras esto es útil trae errores al diseño que sólo se descubren durante la codificación y pruebas.

Otro problema es el costo comparativo. Cuando se construye un puente, el costo del esfuerzo en el plan es aproximadamente un 10% del total, siendo el resto la construcción. En software la cantidad de tiempo gastada codificando es mucho, mucho menor. McConnell sugiere que para un proyecto grande, sólo 15% del proyecto son código y pruebas unitarias, una inversión casi perfecta de las proporciones de la construcción del puente. Aun cuando se consideren las pruebas parte de la construcción, el plan es todavía 50% del total. Esto genera una pregunta importante sobre la naturaleza del diseño en software comparado con su papel en otras ramas de la ingeniería.

Esta clase de preguntas llevaron a Jack Reeves a sugerir que de hecho el código fuente es un documento de diseño y que la fase de construcción está en realidad en la compilación y el ligado. De hecho cualquier cosa que pueda tratar como construcción puede y debe automatizarse.

Estas ideas llevan a algunas conclusiones importantes:

- En software la construcción es tan barata que es casi gratis.
- En software todo el esfuerzo está en el diseño, de modo que requiere de personas creativas y talentosas.
- Los procesos creativos no se planean fácilmente, de modo que la previsibilidad bien puede ser una meta imposible.

Debemos ser muy cautos al usar la metáfora de la ingeniería tradicional para construir software. Es un tipo diferente de actividad y por ende requiere un proceso diferente.

La Impredecibilidad de los Requisitos

Los desarrolladores dicen cuando hay un proyecto problemático "el problema con este proyecto es que los requisitos cambian todo el tiempo". En el negocio de construcción de software los cambios en los requisitos son la norma, la pregunta es qué hacemos al respecto.

Una forma es tratar los requisitos cambiantes como el resultado de una pobre ingeniería de requisitos. La idea detrás de la ingeniería de requisitos es conseguir un cuadro totalmente entendido de los requisitos antes de empezar a construir el software, conseguir la firma del cliente sobre estos requisitos, y entonces preparar procedimientos que limiten los cambios de requisitos después de la firma.

Un problema con esto es que simplemente tratar de entender las opciones para los requisitos es duro. Es aun más duro porque la organización del desarrollo normalmente no proporciona la información del costo en los requisitos. Usted termina en la situación dónde quisiera tener un accesorio en su automóvil, pero el vendedor no puede decirle si agrega 10 al costo del automóvil, o 10,000. Sin una buena idea del costo, ¿cómo puede usted decidir si quiere pagar por ese accesorio?

La estimación es difícil por muchas razones. En parte porque el desarrollo de software es una actividad de diseño, difícil de planear y costar. En parte porque los materiales básicos cambian rápidamente. En parte por lo mucho que depende de los individuos involucrados, y los individuos son difíciles de predecir y cuantificar.

La naturaleza intangible del software también afecta. Es muy difícil saber qué valor aporta un rasgo de software hasta que se usa en realidad. Sólo cuando

se usa realmente una versión temprana de algún software se empieza a entender qué rasgos son valiosos y cuales no.

Esto lleva al punto irónico de que es de esperarse que los requisitos sean cambiables. Así, no sólo son cambiables los requisitos, sino que deben de serlo. Toma mucha energía conseguir que los clientes de software corrijan los requisitos.

Pero aun cuando se pudiera controlar todo esto y realmente se pudiera conseguir un conjunto exacto y estable de requisitos, probablemente aún no estamos a salvo. En la economía de hoy las fuerzas de negocios fundamentales cambian el valor de los rasgos de software demasiado rápido. El que podría ser un buen conjunto de requisitos ahora, no será tan bueno en seis meses.

Casi todo en el desarrollo de software depende de los requisitos. Si no se pueden obtener requisitos estables no se puede obtener un plan predecible.

¿Es Imposible la Previsibilidad?

Hay algunos desarrollos de software dónde la previsibilidad es posible. Organizaciones como el grupo de software del transbordador espacial de la NASA son un ejemplo selecto donde el desarrollo de software puede ser predecible. Requiere mucha ceremonia, mucho tiempo, un equipo grande y requisitos estables. Hay proyectos por ahí que son transbordadores espaciales. Sin embargo el software comercial no encaja en esa categoría. Para éste se necesita un tipo diferente de proceso.

Uno de los grandes peligros es pretender que se puede seguir un proceso predecible cuando no se puede. La gente que trabaja en metodologías no es buena en identificar condiciones límite: los lugares dónde la metodología pasa de apropiada en inapropiada. La mayoría de los metodologistas quieren que sus metodologías sean usadas por todos, de modo que no entienden ni

publican sus condiciones límite. Esto lleva a la gente a usar una metodología en malas circunstancias, como usar una metodología predecible en una situación imprevisible.

Hay una tentación fuerte para hacer eso. La previsibilidad es una propiedad muy deseable. No obstante creer que se puede ser predecible cuando no se puede, lleva a situaciones en donde las personas construyen un plan temprano, y entonces no pueden manejar la situación cuando el plan se cae en pedazos. Se acaba viendo el plan y la realidad flotando aparte. Durante algún tiempo podrá pretender que el plan es válido. Pero en algún punto la deriva es demasiada y el plan se cae en pedazos.

Así si está en una situación impredecible no puede usar una metodología predictiva. Significa que tantos modelos para controlar proyectos, y muchos de los modelos para llevar la relación con el cliente, no son ciertos más. Los beneficios de la previsibilidad son tan grandes, que es difícil dejarlos ir. Como en tantos problemas la parte más difícil está simplemente en comprender que el problema existe.

De cualquier modo dejar ir la previsibilidad no significa que hay que volver al caos ingobernable. Más bien hace falta un proceso que pueda dar control sobre la imprevisibilidad. De eso se trata la adaptabilidad.

Controlando un Proceso Imprevisible - Iteraciones

¿Así que cómo nos controlamos en un mundo imprevisible? La parte más importante y difícil, es saber con precisión dónde estamos. Necesitamos un mecanismo de retroalimentación que pueda decirnos con precisión cual es la situación a intervalos frecuentes.

La llave para obtener esta retroalimentación es el desarrollo iterativo. Esta no es una idea nueva. El desarrollo iterativo ha estado durante algún tiempo bajo muchos nombres: incremental, evolutivo, escenificado, espiral.... La clave del

desarrollo iterativo es producir frecuentemente versiones que funcionen del sistema final que tengan un subconjunto de los rasgos requeridos. Estos sistemas son cortos en funcionalidad, pero por otra parte deben ser fieles a las demandas del sistema final. Deben ser totalmente integrados y tan cuidadosamente probados como una entrega final.

El punto es que no hay nada como un sistema probado, integrado para traer una dosis poderosa de realidad en cualquier proyecto. Los documentos pueden esconder toda clase de fallas. El código no probado puede esconder bastantes defectos. Pero cuando las personas realmente se sientan delante de un sistema y trabajan con él, entonces las fallas se vuelven aparentes: tanto las relativas a bugs como las relativas a los requisitos mal entendidos.

El desarrollo iterativo también tiene sentido en los procesos predecibles. Pero es esencial en los procesos adaptables porque un proceso adaptable necesita poder tratar con los cambios en los rasgos requeridos. Esto lleva a un estilo de planear donde los planes a largo plazo son muy fluidos, y los únicos planes estables son a corto plazo hechos para una sola iteración. El desarrollo iterativo da un fundamento firme en cada iteración que puede usarse para basar los planes posteriores.

Una pregunta importante es cuánto debe durar una iteración. XP sugiere iteraciones de entre una y tres semanas. SCRUM sugiere un mes. Cristal estira aun más. La tendencia, es hacer cada iteración tan corta como se pueda manejar. Esto proporciona la retroalimentación más frecuente, así que sabe más a menudo donde se encuentra.

El Cliente Adaptable

Este tipo de proceso adaptable requiere un tipo diferente de relación con el cliente que las que se consideran a menudo, particularmente cuando el desarrollo es hecho por otra empresa. Cuando contrata una empresa separada para hacer el desarrollo del software, la mayoría de los clientes preferirán un

contrato a precio fijo. Dígale a los desarrolladores lo que quieren, negocie, acepte una oferta, y entonces la carga queda en la empresa de desarrollo.

Un contrato a precio fijo requiere requisitos estables y por tanto procesos predictivos. Los procesos adaptables y los requisitos inestables implican que no se puede trabajar con la noción usual de precio fijo. Tratar de encajar un modelo de precio fijo a un proceso adaptable acaba en una explosión muy dolorosa. La parte sucia de esta explosión es que el cliente queda herido tanto como la compañía de desarrollo de software. Después de todo el cliente no querría un software a menos que su negocio lo necesitara. Si no lo consigue su negocio sufre. Así aun cuando no pague nada a la compañía de desarrollo, todavía pierde. De hecho pierde más de lo que pagaría por el software (¿por qué habría de pagar el software si el valor comercial de ese software fuera menor?).

De modo que hay peligro para ambos lados al firmar un contrato a precio fijo en condiciones dónde un proceso predictivo no puede usarse. Esto significa que el cliente tiene que trabajar de otro modo.

Esto no significa que no se pueda fijar un presupuesto para software por adelantado. Lo que significa es que no se puede fijar el tiempo, precio y alcance. La manera ágil usual es fijar tiempo y precio y permitir que el alcance varíe de manera controlada.

En un proceso adaptable el cliente tiene mucho control a escala fina sobre el proceso de desarrollo de software. A cada iteración puede tanto verificar el progreso como alterar la dirección del desarrollo de software. Se necesita un amplio nivel de compromiso, que es esencial para lograr que un proceso adaptable funcione apropiadamente.

El beneficio importante para el cliente es un desarrollo de software mucho más sensible. Un sistema usable, aunque mínimo, puede entrar en producción

pronto. El cliente puede cambiar sus capacidades de acuerdo a los cambios en el negocio, y también aprender cómo se usa el sistema en realidad.

Una pieza tan importante como ésta es una visibilidad mayor sobre el verdadero estado del proyecto. El problema con los procesos predictivos es que la calidad del proyecto se mide por la conformidad con el plan. Esto dificulta a la gente señalar cuando la realidad y el plan divergen. El resultado común es un gran resbalón más tarde en el calendario del proyecto. En un proyecto ágil hay un constante rehacer del plan con cada iteración. Si las malas noticias están al acecho, tienden a aparecer más temprano, cuando aun se puede hacer algo al respecto. De hecho este control del riesgo es una ventaja clave del desarrollo iterativo. Los métodos ágiles van más allá manteniendo corta la duración de la iteración, pero también viendo estas variaciones como oportunidades.

Hay un aspecto importante en lo que constituye un proyecto exitoso. Un proyecto predictivo se mide a menudo por lo bien que coincide con el plan. Un proyecto que está a tiempo y en costo es un éxito. Esta medida no tiene sentido en un ambiente ágil. Para el agilista la cuestión es el valor de negocio - si el cliente consigue un software más valioso que el costo que puso en él. Un buen proyecto predictivo irá de acuerdo al plan, un buen proyecto ágil construirá algo diferente y mejor que lo que se esperaba en el plan original.

Poniendo a la Gente Primero

No es fácil ejecutar un proceso adaptable. En particular requiere un equipo muy eficaz de desarrolladores. El equipo necesita ser eficaz tanto en la calidad de los individuos como en la manera en que funcionan juntos en equipo. Hay también una sinergia interesante: no sólo la adaptabilidad requiere un equipo fuerte, la mayoría de los buenos desarrolladores prefieren un proceso adaptable.

Uno de los objetivos de las metodologías tradicionales es desarrollar un proceso donde las personas involucradas sean partes reemplazables. Con tal proceso se puede tratar a las personas como recursos que están disponibles en varios tipos. Se tienen un analista, algunos programadores, algunos verificadores, un gerente. Los individuos no son tan importantes, sólo los papeles lo son. De esa manera si usted planea un proyecto no importa qué analista y qué verificadores consiga, sólo hay que saber cuántos son para saber cómo afecta su plan el número de recursos.

Pero esto plantea una pregunta clave: ¿son las personas involucradas en el desarrollo de software partes reemplazables? Uno de los rasgos importantes de los métodos ágiles es el rechazo a esta afirmación.

Quizás el rechazo más explícito a las personas como recursos lo hace Alistair Cockburn. En su artículo Caracterización de las Personas como Componentes No Lineales de Primer Orden en el Desarrollo de Software, apunta a que los procesos predecibles requieren componentes que se comporten de manera predecible. Sin embargo las personas no son componentes predecibles. Además sus estudios de proyectos de software lo han llevado a concluir que la gente es el factor más importante en el desarrollo de software.

En el título, [de su artículo] me refiero a las personas como "componentes". Así es como se trata a la gente en la literatura de diseño de procesos / metodologías. El error de este enfoque es que las "personas" son altamente inconstantes y no lineales, con modos únicos de éxito y fracaso. Esos factores son de primer orden, factores no despreciables. El fracaso de los diseñadores de procesos y metodologías para responder a esto contribuye a la clase de trayectorias de proyectos no planeados que vemos a menudo. - [Cockburn, non-linear]

Decidir que las personas son primero es una decisión grande, que requiere mucha determinación. La noción de la gente como recursos es profundamente

inculcado en el pensamiento de negocios, teniendo sus raíces en el impacto del enfoque de La Dirección Científica de Frederick Taylor. En la administración de una fábrica, este enfoque Taylorista tiene sentido. Pero para un trabajo altamente creativo y profesional, como creo es el desarrollo de software, esto no se sostiene. (Y de hecho la fabricación moderna también se está saliendo del modelo Taylorista.)

Una parte clave de la noción Taylorista es que la gente que hace el trabajo no es la mejor gente para entender la mejor manera de hacer el trabajo. En una fábrica esto puede ser verdad por varias razones. En parte porque la mayoría de los obreros no son las personas más inteligentes o creativas, en parte porque hay una tensión entre la gerencia y los obreros en que la gerencia gana más dinero y los obreros menos.

La historia reciente nos demuestra cada vez más lo falso que es esto para el desarrollo de software. A las personas brillantes y capaces les atrae cada vez más el desarrollo de software, tanto por su ostentación como por ganancias potencialmente mayores. Cosas tales como opciones accionarias afirman el interés de los programadores en la compañía.

Cuando se quiere contratar y retener a gente capaz, hay que reconocer que son profesionales competentes. Como tales son los mejores para decidir cómo dirigir su trabajo técnico. La noción Taylorista de un departamento de planificación separado que decide cómo hacer las cosas sólo funciona si los planificadores entienden mejor cómo hacer bien el trabajo que aquéllos que lo hacen. Si usted tiene personas brillantes, motivadas que hacen bien el trabajo entonces eso no se sostiene.

Manejando un Proceso Orientado a la Gente

La orientación a la gente se manifiesta de varias maneras diferentes en los procesos ágiles, lo que lleva a efectos diferentes, no todos consistentes.

Uno de los elementos clave es la aceptación de un proceso en lugar de la imposición de un proceso. A menudo los procesos de software se imponen desde la gerencia. Como tales se les resiste a menudo, particularmente cuando la gerencia ha estado fuera del desarrollo activo un buen tiempo. Aceptar un proceso requiere compromiso, y como tal se necesita que todo el equipo se involucre activamente.

Esto termina con el resultado interesante de que sólo los desarrolladores pueden escoger seguir un proceso adaptable. Esto es particularmente cierto para la XP, que requiere mucha disciplina para ejecutarse. Aquí es donde Cristal es un complemento efectivo ya que apunta a una disciplina mínima.

Otro punto es que los desarrolladores deben poder tomar todas las decisiones técnicas. XP llega al corazón de esto cuando en su proceso de planeación establece que sólo los desarrolladores pueden estimar cuánto tiempo tomará hacer un trabajo.

Tal liderazgo técnico es un gran cambio para muchas personas en posiciones gerenciales. Tal acercamiento requiere compartir una responsabilidad donde desarrolladores y gerencia tienen un mismo lugar en la dirección del proyecto. Nótese el termino igual. La gerencia aun juega un papel, pero reconoce la pericia de los desarrolladores.

Una razón importante para esto es el ritmo del cambio de tecnología en nuestra industria. Después de unos años el conocimiento técnico se vuelve obsoleto. Esta vida media de habilidades técnicas no tiene parangón en cualquier otra industria. Incluso los técnicos tienen que reconocer que entrar a la gerencia significa que sus habilidades técnicas se marchitarán rápidamente. Los ex desarrolladores necesitan reconocer que sus habilidades técnicas desaparecerán rápidamente y necesitan confiar y depender en los desarrolladores actuales.

La Dificultad de Medir

Si tiene un proceso dónde las personas que dicen cómo hacer el trabajo son diferentes de las personas que realmente lo hacen, los líderes necesitan alguna manera de medir cuán eficaces son los que lo hacen.

Esto es particularmente pertinente al software debido a la dificultad de aplicar medidas al software. A pesar de nuestros mejores esfuerzos somos incapaces de medir las cosas más simples sobre el software, como la productividad. Sin buenas medidas para estas cosas, cualquier clase de control externo está condenado.

La introducción de gestión medida sin buenas medidas tiene sus propios problemas. Cuando se mide el rendimiento usted tiene que conseguir todos los factores importantes bajo medida. Cualquier cosa que se olvide tiene el resultado inevitable que los hacedores alterarán lo que hacen para producir las mejores medidas, incluso si eso claramente reduce la verdadera efectividad de lo que hacen. Este trastorno de la medida es el talón de Aquiles de la gestión basada en medidas.

La conclusión es que usted tiene que escoger entre la gestión basada en métricas y la gestión delegatoria (donde los hacedores deciden cómo hacer el trabajo). La gestión basada en métricas es más adecuada para el trabajo simple repetitivo, con bajos requisitos de conocimiento y rendimientos fácilmente medibles - exactamente lo contrario al desarrollo de software.

El punto de todo esto es que los métodos tradicionales han operado bajo la asunción de que la gestión basada en métricas es la manera más eficaz de administrar. La comunidad ágil reconoce que las características del desarrollo de software son tales que la gestión basada en métricas lleva el trastorno de la medida a niveles muy altos. Es realmente más eficaz usar un estilo delegatorio de administración que es el tipo de acercamiento que está en el centro del punto de vista agilista.

El Papel del Liderazgo de Negocio

Pero los técnicos no pueden hacer el proceso entero ellos solos. Necesitan una guía en las necesidades comerciales. Esto lleva a otro aspecto importante de los procesos adaptables: necesitan un contacto muy íntimo con los expertos del negocio.

Esto va más allá del compromiso de negocios en la mayoría de los proyectos. Los equipos ágiles no pueden existir con una comunicación ocasional. Necesitan un acceso continuo a los expertos del negocio. Además este acceso no es algo que se maneja a nivel gerencial, es algo que está presente para cada desarrollador. Como los desarrolladores son profesionales capaces en su propia disciplina, necesitan poder trabajar como iguales con otros profesionales de otras disciplinas.

Gran parte de esto, claro está, se debe a la naturaleza del desarrollo adaptable. Como la premisa entera del desarrollo adaptable es que las cosas cambian rápidamente, se necesita un contacto constante para avisar a todos de los cambios.

No hay nada más frustrante para un desarrollador que ver desperdiciarse su trabajo duro. Así que es importante asegurar que hay pericia de negocios de buena calidad que está disponible al desarrollador y de calidad suficiente para que el desarrollador pueda confiar en ella.

El Proceso Auto-Adaptable

Hasta ahora hemos hablado sobre la adaptabilidad en el contexto de un proyecto adaptando frecuentemente su software para enfrentarse a los requisitos cambiantes de sus clientes. No obstante hay otro ángulo de la adaptabilidad: el del proceso que cambia con el tiempo. Un proyecto que empieza usando un proceso adaptable no tendrá el mismo proceso un año

después. Con el tiempo, el equipo encontrará lo que funciona mejor para ellos, y alterará el proceso a su medida.

La primera parte de la auto-adaptabilidad son las revisiones regulares del proceso. Normalmente se hacen con cada iteración. Al final de cada iteración, haga una reunión corta y hágase las siguientes preguntas (Norm Kerth):

- ¿Qué hicimos bien?
- ¿Qué hemos aprendido?
- ¿Qué podemos hacer mejor?
- ¿Qué es lo que nos confunde?

Estas preguntas traerán ideas para cambiar el proceso en la siguiente iteración. De esta manera un proceso que empieza con problemas puede mejorar conforme el proyecto avanza, adaptándose mejor al equipo que lo usa.

Si la auto-adaptabilidad ocurre dentro de un proyecto, es aun más marcada en una organización. Estas retrospectivas involucran reuniones de 2-3 días y un facilitador entrenado. Ellas no solo dan aprendizaje al equipo, también dan aprendizaje a la organización entera.

Una consecuencia de la auto-adaptabilidad es que nunca se debe esperar encontrar una metodología corporativa única. En cambio cada equipo debe no simplemente escoger su propio proceso, debe también afinar activamente su proceso conforme avanza el proyecto. Mientras que los procesos publicados como experiencia de otros proyectos pueden actuar como una inspiración y referencia, la responsabilidad profesional de los desarrolladores es adaptar el proceso a la tarea en mano.

4.3. Las Metodologías

Varias metodologías encajan bajo el estandarte de ágil. Mientras todas ellas comparten muchas características, también hay algunas diferencias significativas. No podemos resaltar todos los puntos pero mencionamos varias de ellas.

XP (Programación Extrema)

De todas las metodologías ágiles, ésta es la que ha recibido más atención. Esto se debe en parte a la notable habilidad de los líderes XP, en particular Kent Beck, para llamar la atención. También se debe a la habilidad de Kent Beck de atraer a las personas a este acercamiento, y tomar un papel principal en él. De algunas maneras, sin embargo, la popularidad de XP se ha vuelto un problema, pues ha acaparado la atención fuera de las otras metodologías y sus valiosas ideas.

Las raíces de la XP yacen en la comunidad de Smalltalk, y en particular la colaboración cercana de Kent Beck y Ward Cunningham a finales de los 1980s. Ambos refinaron sus prácticas en numerosos proyectos a principios de los 90s, extendiendo sus ideas de un desarrollo de software adaptable y orientado a la gente.

El paso crucial de la práctica informal a una metodología ocurrió en la primavera de 1996. A Kent se le pidió revisar el progreso del proyecto de nómina C3 para Chrysler. El proyecto estaba siendo llevado en Smalltalk por una compañía contratista, y estaba en problemas. Debido a la baja calidad de la base del código, Kent recomendó tirar la base del código en su totalidad y empezar desde el principio. El proyecto entonces reinició bajo su dirección y subsecuentemente se volvió el buque insignia temprano y el campo de entrenamiento de XP.

La primera fase del C3 fue muy exitosa y comenzó a principios de 1997. El proyecto continuó desde entonces y después se encontró con dificultades, lo que resultó en la cancelación del desarrollo en 1999. (Lo cual prueba, que la XP no es garantía de éxito.)

La XP empieza con cuatro valores: Comunicación, Retroalimentación, Simplicidad y Coraje. Construye sobre ellos una docena de prácticas que los proyectos XP deben seguir. Muchas de estas prácticas son técnicas antiguas, tratadas y probadas, aunque a menudo olvidadas por muchos, incluyendo la mayoría de los procesos planeados. Además de resucitar estas técnicas, la XP las teje en un todo sinérgico dónde cada una refuerza a las demás.

Una de las más llamativas, así como inicialmente atractiva, es su fuerte énfasis en las pruebas. Mientras todos los procesos mencionan la comprobación, la mayoría lo hace con muy poco énfasis. Sin embargo la XP pone la comprobación como el fundamento del desarrollo, con cada programador escribiendo pruebas cuando escriben su código de producción. Las pruebas se integran en el proceso de integración continua y construcción lo que rinde una plataforma altamente estable para el desarrollo futuro.

En esta plataforma XP construye un proceso de diseño evolutivo que se basa en refactorar un sistema simple en cada iteración. Todo el diseño se centra en la iteración actual y no se hace nada anticipadamente para necesidades futuras. El resultado es un proceso de diseño disciplinado, lo que es más, combina la disciplina con la adaptabilidad de una manera que indiscutiblemente la hace la más desarrollada entre todas las metodologías adaptables.

La Familia de Cristal de Cockburn

Alistair Cockburn ha estado trabajando en metodologías desde que la IBM le encargó escribir sobre metodologías a inicios de los 90. Su acercamiento no es como la mayoría de los metodologistas. En lugar de partir solamente de su experiencia personal para construir una teoría de cómo deben hacerse las

cosas, él complementa su experiencia directa con la búsqueda activa de ver cómo trabajan los proyectos. Además él no teme alterar sus puntos de vista con base en sus descubrimientos.

Su libro, *Sobreviviendo Proyectos Orientados a Objetos*, fue su primer consejo en proyectos corrientes. Más recientemente Alistair escribió un libro de apreciación de desarrollo de software ágil que mira los principios subyacentes de este tipo de metodologías.

Desde ese libro él ha explorado más los métodos ágiles, viniendo con la familia de metodologías Crystal. Es una familia porque él cree que los tipos diferentes de proyectos requieren tipos diferentes de metodologías. Él mira esta variación a lo largo de dos ejes: el número de personas en el proyecto, y las consecuencias de los errores. Cada metodología encaja en una parte diferente de la reja, de modo que para un proyecto de 40 personas que puede perder dinero discrecionalmente tiene una metodología diferente a la de un proyecto vital de seis personas.

Los Crystal comparten con la XP una orientación humana, pero esta centralización en la gente se hace de una manera diferente. Alistair considera que las personas encuentran difícil seguir un proceso disciplinado, así que más que seguir la alta disciplina de la XP, Alistair explora la metodología menos disciplinada que aun podría tener éxito, intercambiando conscientemente productividad por facilidad de ejecución. Él considera que aunque Crystal es menos productivo que la XP, más personas serán capaces de seguirlo.

Alistair también pone mucho peso en las revisiones al final de la iteración, animando al proceso a ser auto perfeccionista. Su aserción es que el desarrollo iterativo está para encontrar los problemas temprano, y entonces permitir corregirlos. Esto pone más énfasis en la gente supervisando su proceso y afinándolo conforme desarrollan.

Código Abierto (Open Source)

El código abierto es un estilo de hacer software, no tanto un proceso. Sin embargo hay una manera definida de hacer las cosas haciendo en la comunidad de código abierto, y mucho de su enfoque es tan aplicable a los proyectos de código cerrado como a los de código abierto. En particular su proceso se engrana a equipos físicamente distribuidos, lo qué es importante porque la mayoría de los procesos adaptables exigen equipos locales.

La mayoría de los proyectos de código abierto tienen uno o más mantenedores. Un mantenedor es la única persona a la que se le permite integrar un cambio en el almacén de código fuente. Sin embargo otras personas pueden hacer cambios a la base del código. La diferencia importante es que estas otras personas necesitan enviar su cambio al mantenedor que entonces lo revisa y lo aplica a la base del código. Normalmente estos cambios son hechos en forma de archivos de parches que hacen este proceso más fácil. El mantenedor así es responsable de coordinar los parches y mantener la cohesión en el diseño del software.

Proyectos diferentes manejan el papel del mantenedor de diferentes maneras. Algunos tienen un mantenedor para el proyecto entero, algunos lo dividen en módulos y tiene un mantenedor por módulo, algunos rotan el mantenedor, algunos tienen múltiples mantenedores sobre el mismo código, otros tienen una combinación de estas ideas. La mayor parte de la gente de código abierto son de tiempo parcial, así que hay una duda en qué tan bien se coordina un equipo así para un proyecto de tiempo completo.

Un rasgo particular del desarrollo de código abierto es que la depuración es altamente paralelizable. Muchas personas pueden involucrarse en el depurado. Cuando encuentran un bug pueden enviar el parche al mantenedor. Esto es un buen papel para los no mantenedores ya que la mayor parte del tiempo se gasta en encontrar bugs. También es bueno para gente sin mucha destreza en programación.

El Desarrollo de Software Adaptable de Highsmith

Jim Highsmith ha pasado muchos años trabajando con metodologías predictivas. Él las desarrolló, instaló, enseñó, y concluyó que son profundamente defectuosas: particularmente para los negocios modernos.

En el corazón del ASD hay tres fases solapadas, no lineales: especulación, colaboración, y aprendizaje.

Highsmith ve la planificación como una paradoja en un ambiente adaptable, ya que los resultados son naturalmente imprevisibles. En la planificación tradicional, las desviaciones del plan son errores que deben corregirse. En un el ambiente adaptable, sin embargo, las desviaciones nos guían hacia la solución correcta.

En este ambiente imprevisible se necesita que las personas colaboren de la mejor manera para tratar con la incertidumbre. La atención de la gerencia es menor en lo que tiene que hacer la gente, y mayor sobre la comunicación alentadora para que las personas puedan proponer las respuestas creativas ellos mismos.

En ambientes predictivos, el aprendizaje se desalienta a menudo. Las cosas se ponen de antemano y entonces se sigue ese diseño.

En un ambiente adaptable, aprender desafía a todos - desarrolladores y sus clientes - a examinar sus asunciones y usar los resultados de cada ciclo de desarrollo para adaptar el siguiente.

-[Highsmith]

El aprendizaje como tal es un rasgo continuo e importante, uno que asume que los planes y los diseños deben cambiar conforme avanza el desarrollo.

Con este énfasis, el trabajo de Highsmith se enfoca directamente en fomentar las partes difíciles del desarrollo adaptable, en particular cómo fomentar la colaboración y el aprendizaje dentro del proyecto.

Scrum

Scrum ha estado durante algún tiempo en los círculos orientados a objetos. De nuevo se enfoca en el hecho de que procesos definidos y repetibles sólo funcionan para atacar problemas definidos y repetibles con gente definida y repetible en ambientes definidos y repetibles.

Scrum divide un proyecto en iteraciones (llamadas carreras de velocidad o sprints) de 30 días. Antes de que comience una carrera se define la funcionalidad requerida para esa carrera y entonces se deja al equipo para que la entregue. El punto es estabilizar los requisitos durante la carrera.

Sin embargo la gerencia no se desentiende durante la carrera corta. Todos los días el equipo sostiene una junta corta (quince minutos), llamada scrum, dónde el equipo discurre lo que hará al día siguiente. En particular muestran a los bloques de la gerencia: los impedimentos para progresar que se atraviesan y que la gerencia debe resolver. También informan lo que se ha hecho para que la gerencia tenga una actualización diaria de dónde va el proyecto.

La literatura de Scrum se enfoca principalmente en la planeación iterativa y el seguimiento del proceso.

Desarrollo Manejado por Rasgos

El Desarrollo Manejado por Rasgos (FDD por sus siglas en inglés) fue desarrollado por Jeff De Luca y el viejo gurú de la OO Peter Coad. Como las otras metodologías adaptables, se enfoca en iteraciones cortas que entregan funcionalidad tangible. En el caso del FDD las iteraciones duran dos semanas.

El FDD tiene cinco procesos. Los primeros tres se hacen al principio del proyecto.

- Desarrollar un Modelo Global
- Construir una Lista de los Rasgos
- Planear por Rasgo
- Diseñar por Rasgo
- Construir por Rasgo

Los últimos dos se hacen en cada iteración. Cada proceso se divide en tareas y se da un criterio de comprobación.

Los desarrolladores entran en dos tipos: dueños de clases y programadores jefe. Los programadores jefe son los desarrolladores más experimentados. A ellos se les asignan rasgos a construir. Sin embargo ellos no los construyen solos. Solo identifican qué clases se involucran en la implantación de un rasgo y juntan a los dueños de dichas clases para que formen un equipo para desarrollar ese rasgo. El programador jefe actúa como el coordinador, diseñador líder y mentor mientras los dueños de clases hacen gran parte de la codificación del rasgo.

Manifiesto para el Desarrollo de Software Ágil

Con tanta similitud entre estos métodos, los representantes de cada una de estas metodologías fueron invitados a un taller de dos días en Snowbird, Utah en febrero de 2001.

El resultado es un Manifiesto para el Desarrollo de Software Ágil, una declaración de los principios y valores comunes de los procesos ágiles. Hay también un deseo de colaborar más en el futuro, para animar más tanto a tecnólogos como a gente de negocios para usar y requerir acercamientos ágiles al desarrollo de software.

El manifiesto fue sólo eso, una publicación que actuó como un punto de partida para aquellos que compartían estas ideas básicas. Uno de los frutos del esfuerzo fue la creación de un cuerpo más longevo, la Alianza Ágil. La Alianza Ágil es una organización sin fines de lucro que busca promover el conocimiento y la discusión de todos los métodos ágiles.

¿Es RUP un método ágil?

Siempre que se discute métodos en la arena OO, inevitablemente salimos con el papel del Rational Unified Process. El Proceso Unificado fue desarrollado por Philippe Kruchten, Ivar Jacobson y otros de la Rational como el proceso complementario al UML. El RUP es un armazón de proceso y como tal puede acomodar una gran variedad de procesos.

Como resultado de esta mentalidad de armazón de procesos, el RUP puede usarse en un estilo muy tradicional de cascada o de una manera ágil. Como resultado se puede usar el RUP como un proceso ágil, o como un proceso pesado - todo depende de cómo lo adapte a su ambiente.

¿Debe usted irse a lo ágil?

El uso de un método ágil no es para todos. Hay que tener en cuenta varias cosas si se decide a seguir por este camino. Sin estas nuevas metodologías son extensamente aplicables y deben ser usadas por más personas de las que actualmente lo consideran.

En el ambiente actual, la metodología más común es codifica y corrige. Aplicar más disciplina que caos casi seguramente ayudará, y el acercamiento ágil tiene la ventaja de que es mucho menos de un paso que un método pesado. Mucha de la ventaja de los métodos ágiles es de hecho su peso ligero. Los procesos más simples son más probables de ser seguidos cuando uno no está acostumbrado a ningún proceso en absoluto.

Una de las limitaciones más grandes de estas nuevas metodologías es cómo manejan equipos más grandes. Como muchas nuevas tendencias, ellos tienden a ser usados primero a escala pequeña antes que a gran escala. También a menudo se han creado con énfasis en equipos pequeños. La XP explícitamente dice que está diseñada para equipos de no más de veinte personas.

Esperanzadoramente un mensaje que queda claro es que los acercamientos adaptables son buenos cuando sus requisitos son inciertos o volátiles. Si no tiene requisitos estables, entonces no está en la posición tener un plan estable y seguir un proceso planeado. En estas situaciones un proceso adaptable puede ser menos cómodo, pero será más eficaz. A menudo la barrera más grande aquí es el cliente. Es importante para el cliente entender que seguir un proceso predictivo cuando los requisitos cambian es arriesgado tanto para ellos como para el desarrollo.

Si va a tomar la ruta adaptable, necesita confiar en los desarrolladores e involucrarlos en la decisión. Los procesos adaptables cuentan en que usted confía en sus desarrolladores, de modo que si usted considera a sus desarrolladores de baja calidad y motivación, usted debe usar un acercamiento predictivo.

Para resumir. Los siguientes factores sugieren un proceso adaptable:

- Requisitos inciertos o volátiles
- Desarrolladores responsables y motivados
- Clientes que entienden y se involucrarán.

Estos factores sugieren un proceso predictivo

- Un equipo de más de cien
- Un precio fijo, o más correctamente un alcance o contrato fijo

5. METODOLOGIA A UTILIZAR

5.1. Comparando Procesos: RUP y métodos Ágiles

Discutiremos características y diferencias entre:

- Enfoques Agile como son Extreme Programming, Scrum, Dynamic Systems Development Method (DSDM), Crystal, y Adaptive Development
- RUP

Estos procesos serán comparados usando un “mapa de proceso” con dos dimensiones: baja ceremonia/Alta ceremonia y Cascada/Iterativo. Nosotros podemos usar este mapa para entender que debes pugnar en términos de mejoras en el proceso, para que puedas aplicar en RUP con el monto correcto de ceremonia y con el correcto nivel de desarrollo iterativo para encajar en las necesidades específicas de tu proyecto u organización.

¿Como podemos comparar procesos?

¿Como los procesos pueden ser comparados? Nosotros los hemos caracterizado mirando en las dos dimensiones discutidas abajo y mostrando nuestro mapa de proceso:

- Baja Ceremonia/Alta Ceremonia: en el eje horizontal. Baja ceremonia produce la documentación mínima de soporte y tiene un pequeño formalismo en el procedimiento de trabajo; Alta ceremonia tiene un soporte comprensivo de documentación y capacidad de rastreo conservado entre artefactos, Comité de control de cambios, y así.

- Cascada/Iterativo en el eje vertical. Cascada es un enfoque lineal con tardada integración y test. Iterativo es un desarrollo manejado por riesgos con implementación temprana de una arquitectura y temprana integración y testeo.

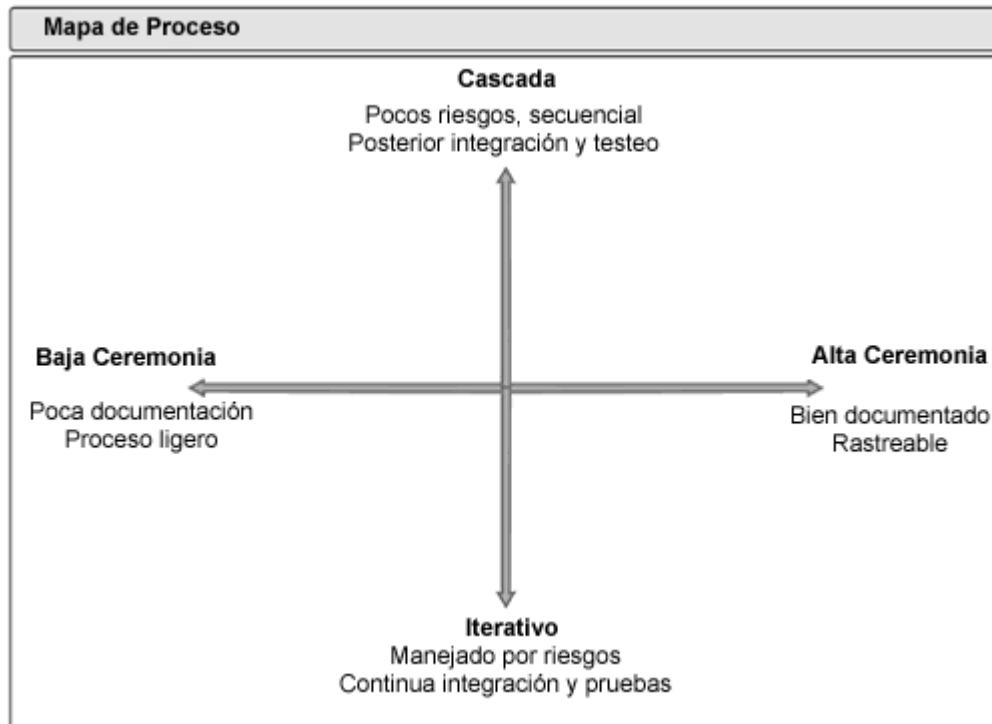


Ilustración 4: Mapa de Procesos para comparación de procesos. Por organización de procesos y frameworks de enfoques de procesos a lo largo de dos dimensiones – Baja Ceremonia/Alta Ceremonia y Cascada/Iterativo – puedes compararlos y analizar cuales son los mas adecuados para tu proyecto o organización.

Desarrollo Agile: Baja Ceremonia, enfoque iterativo

El movimiento Ágil se ha convertido incrementalmente popular, especialmente entre individuos y desarrolladores y pequeños equipos, y esta creciendo en popularidad. Algunos de los enfoques ágiles son XP, Scrum, y Adaptive Development. Notar que dejamos a propósito el enfoque RUP fuera de esta lista – iremos a RUP posteriormente.

El desarrollo Agile hace algunos sacrificios en términos de ceremonia y rigor, en retorno de flexibilidad y habilidad para adaptar y evolucionar ambientes de negocio. Esto toma gran valor en la producción de software trabajando más que creando cara y comprensiva documentación. En vez de trabajar desde un plan de desarrollo rígido, uno de las principales doctrinas de desarrollos ágiles es responder a los cambios que ocurren durante el proceso. Esto no significa que planes y documentación son poco importantes – ellos solo no son tan importantes como asegurar que el software trabaja o adapta planes a la realidad y cambiantes necesidades.

XP, Scrum, Crystal, y Adaptive Development crecieron en popularidad luego de 1990, pero ellos están contruidos primariamente sobre mejores prácticas que han estado por aquí por muchos años, como son desarrollo iterativo, integración continua, fuerte enfoque sobre software ejecutable antes que en productos derivados, refactoring, uso de estándares de codificación, e historias de usuarios. Como un ejemplo, desarrollo iterativo ha estado por aquí desde la mitad de 1980, y por muchos años Rational Software lo ha promovido activamente, el desarrollo iterativo con un fuerte enfoque sobre el software ejecutable más que por productos derivados. El movimiento Agile ha hecho un excelente trabajo promoviendo y popularizando estas mejores practicas y ha introducido algunos de sus propios como **pair programming** y **test-first design**. El movimiento Ágil ha también hecho algún excelente trabajo de minimizar la ceremonia que rodeaba el desarrollo de software, asegurando que el enfoque primario sea sobre el software y no sobre los artefactos de soporte.

Posiblemente uno de los Movimientos Agile con mas significativa contribución ha sido la aceptación de conceptos como son “procesos” y “mejores practicas” por campos de desarrollo que previamente estuvieron arriba de ellos. El proceso se ha convertido en excelente, y los desarrolladores reconocen que el valor de aprender de las experiencias de otros.

¿Entonces donde hacer procesos ágiles encajar en nuestro mapa de procesos? La siguiente figura pone todos los procesos ágiles en el lado izquierdo del cuadrante por que ellos son iterativos, enfoques con baja-ceremonia que producen mínima documentación.

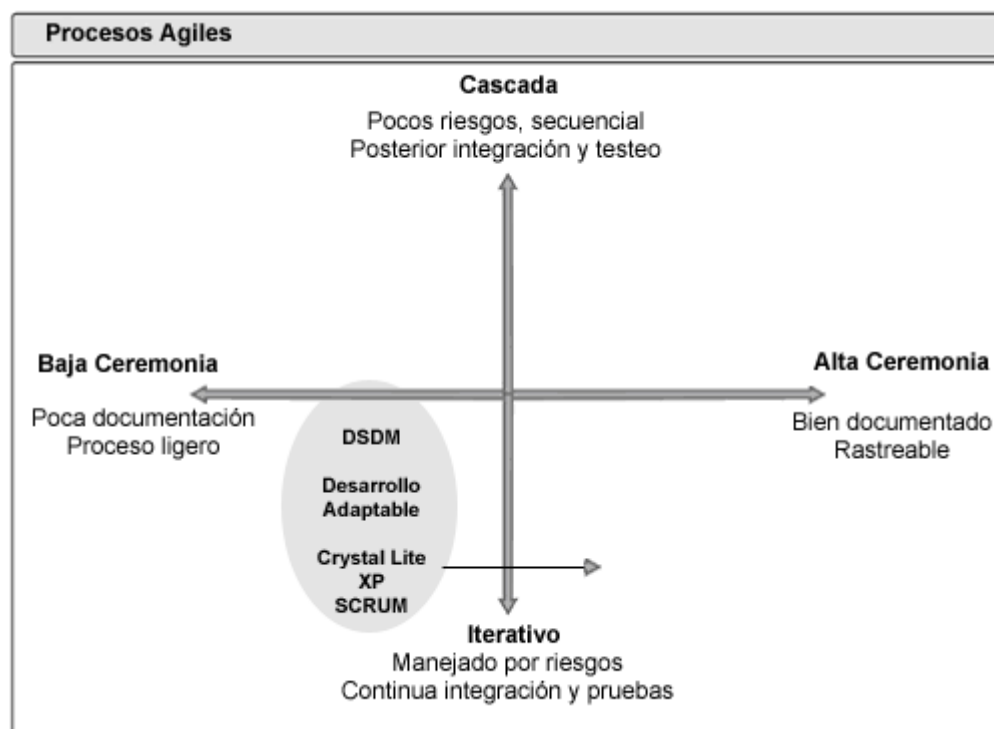


Ilustración 5: Procesos Ágiles sobre el Mapa de Procesos. Procesos de desarrollo Agile son caracterizados por un enfoque altamente iterativo usando un mínimo de documentación y formalismo, que los hace ideal para pequeños, menos complejos proyectos. Algunos enfoques Agile tienen adicionales directrices para proyectos mas complejos, haciéndolos con mayor ceremonia.

Desde que el movimiento Agile tiene unos pocos años de antigüedad, muchos de los enfoques Agile son nuevos y poco probados y aun no proveen mucha

guía sobre implementaciones prácticas. Consecuentemente, muchas organizaciones luchan con efectivamente aplicar estos métodos.

Algunos procesos Agile, como son XP, Scrum, y Crystal están ahora comenzando a utilizarse por grandes y mas complejos proyectos. Como los tamaños de los proyectos se incrementan, más y más guía es requerida para el éxito de la ejecución del proyecto. Si nosotros trasladamos esto dentro de nuestro mapa de procesos, veremos que usando procesos Agile para incrementalmente complejos proyectos requiere adicional guía y mas ceremonia, así llevando el proceso a la derecha que corresponde con alta ceremonia.

Cristal es una familia de procesos. La figura muestra donde Crystal Lite esta sobre el mapa, y la flecha indica donde se encuentra las versiones con mayor ceremonia de Cristal residen en el mapa. Cristal es una de algunos procesos Agile moviéndose hacia altas versiones de ceremonia.

RUP: Un enfoque iterativo con un Adaptable nivel de ceremonia

Racional Unified Process es un framework adaptable de procesos que permite a los usuarios a producir una amplia variedad de procesos, o Configuraciones RUP. Estas configuraciones pueden encajar donde sea sobre la escala Baja Ceremonia/Alta Ceremonia, dependiendo de las necesidades del proyecto. La siguiente figura muestra donde las diferentes configuraciones de RUP podrían encajar en el mapa de procesos. El RUP fuertemente promueve un iterativo, enfoque manejado por riesgos para el desarrollo de software con continuo testeo e integración. Hay, sin embargo, alguna flexibilidad a lo largo del eje Cascada/Iterativo, y algunas compañías aplican RUP en una manera de cascada –algunos generalmente aconsejan en contra.

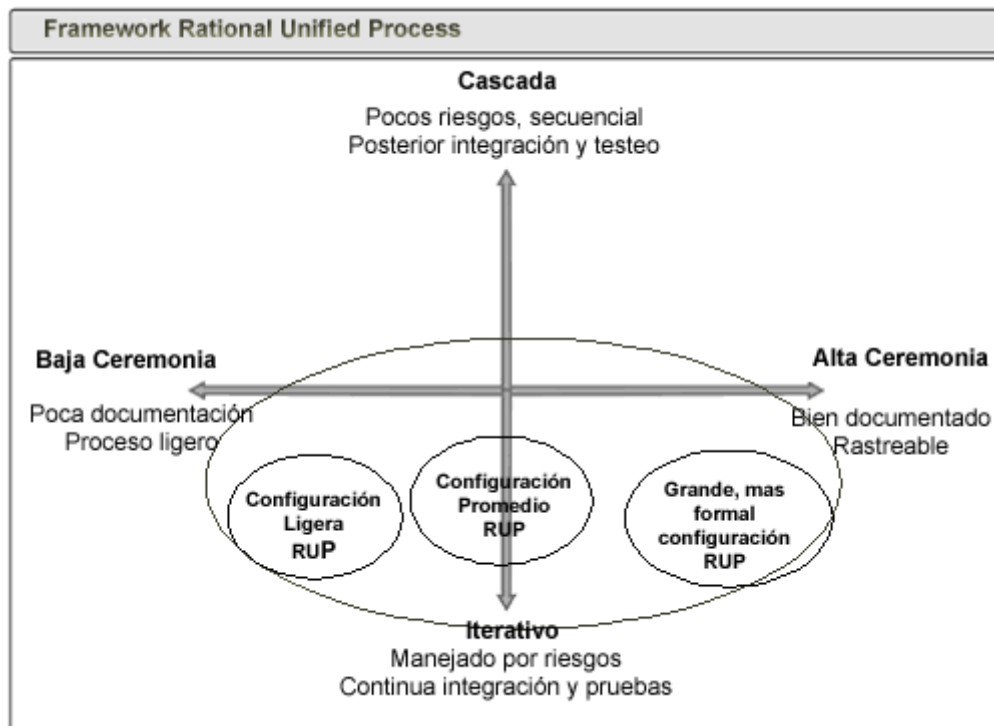


Ilustración 6: Configuraciones RUP sobre el mapa de Procesos. El framework RUP permite producir una variedad de configuraciones de procesos desde un muy bajo a un muy alto nivel de procesos de ceremonia.

La metodología a utilizar por mucho en este trabajo será RUP, en general tiene la suficiente flexibilidad para poder manejar cualquier tipo de proyecto en cualquier tipo de organización, con la cantidad de ceremonia que se desee (alta/baja ceremonia), utilizando un enfoque Cascada o Iterativo. La desventaja de las metodologías ágiles es que aun no ha sido probado en tantos proyectos como lo ha sido RUP y la ventaja de RUP es que puede aplicarse de manera Ágil limitando el número de artefactos a utilizar. Si bien no vamos a llegar a utilizar todo el proceso lo utilizaremos como una guía para poder obtener la arquitectura del sistema.

Las razones para introducir el RUP y las herramientas de soporte dentro de tu organización son para obtener beneficios de negocios cuantificados y resultados de proyectos mejorados. Para que la inversión valga la pena, la mejora de procesos a través de un uso efectivo del RUP y tecnologías asociadas debe finalmente llevarnos a tecnologías de alta calidad, bajo costo, o corto tiempo de desarrollo. Un error común es lanzarse con una versión pesada

de RUP, donde tienes también muchos artefactos que necesitan ser producidos por el equipo de proyecto. Esto puede impedir que alcances los resultados de negocios esperados. Cuando sea posible, debes minimizar el número de artefactos a ser producidos sin comprometer sobre calidad.

Adoptar el RUP significa que necesitas cambiar la manera de tu trabajo, y el cambio es difícil. Necesitas tener un claro entendimiento de la motivación del cambio, que cambio tiene sentido, que cambios son menos prioritarios, como introducir cambio y como medir éxito en términos de mejorar los resultados del proyecto.

Para alcanzar resultados mejorados de los proyectos, algunas veces necesitaras no solo una mejora en los procesos, también herramientas de soporte. Las organizaciones que no entienden la relación entre procesos y herramientas pueden fallar en alcanzar algún beneficio tangible de su esfuerzo de mejora de procesos.

5.2. Adoptando RUP en un proyecto

Es fácil implementar RUP en el inicio de un proyecto para evitar adicionar la complejidad de transición desde el proceso actual al RUP en la mitad del proyecto. Los siguientes 5 pasos pueden ayudarte cuando introduces el producto RUP en el inicio de tu proyecto.

1. **Valorar:** valorar que problemas tu equipo ha estado enfrentando en el pasado, que impacto ellos han tenido en tu negocio, y que asuntos son de alta prioridad de negocio para tener una solución en el futuro. Claramente articular los beneficios esperados para el negocio, idealmente a través de casos de negocio.
2. **Plan:** planear las actividades, como adaptación del proceso, entrenamiento (que son, lecciones estructuradas) y tutoría (que es, un

miembro experto del proyecto trabajando lado a lado con el equipo del proyecto haciendo entrenamiento en el trabajo), y comprar las herramientas requeridas. Enfocarse en el entrenamiento just-in-time para asegurar estudiantes motivados, si aprendes rápido podrás poner el conocimiento en practica rápidamente.

3. **Configurar y adaptar:** Hacer la configuración y adaptación necesaria del proceso, ambientes de herramientas, y material de entrenamiento. Hacer solo adaptaciones que son motivadas desde una perspectiva de negocios, que es, esta alineado con los casos del negocio.
4. **Ejecutar:** Distribuir el proceso y herramientas en tu proyecto, entrenar y guiar a los miembros del proyecto, y ejecutar tu proyecto. Enfocarse en los resultados del equipo del proyecto, y usar tutores para asegurar que la utilización del proceso y herramientas están alineados con los objetivos del negocio.
5. **Evaluar:** evaluar la performance del proyecto y comparar este a las expectativas como se expresa en el caso del negocio. Identificar las oportunidades para futuras mejoras.

Lo siguiente es un breve boceto de lo que es hecho durante un proyecto:

- **Antes de iniciar el proyecto:** Antes de que el proyecto realmente inicie, las personas clave necesitan ser entrenadas – usualmente ingenieros de procesos, arquitectos, líderes de equipo, y administradores de proyecto. Estas personas necesitan instalar el proyecto, reclutar la gente correcta, y sacar adelante los planes correctos para la Concepción y más allá. Típicamente también necesitas instalar una infraestructura básica para la concepción, como con herramientas para administración de requerimientos y seguimiento de cambios y defectos. Un tutor puede ayudar con la producción del caso de desarrollo, entrenamiento, y planeamiento del proyecto y proceso de implementación.

- **Fase de Concepción:** Durante esta fase, típicamente enfocarse como mejorar la manera que manejas los requerimientos (disciplina de Requerimientos) y como administras el proyecto (disciplina de Administración de Proyectos). Debes establecer una infraestructura de Administración de Configuración (CM) para la Elaboración, y debes querer hacer entrenamiento JIT en estas áreas. Un tutor puede realizar el entrenamiento JIT, como también trabajar lado a lado con los miembros del equipo para mostrarles como trabajar con las disciplinas de Requerimientos y Administración del Proyecto. Un tutor puede ayudarte en establecer un sano ambiente CM.
- **Fase de elaboración:** Hacia el final de la fase de Elaboración, todos los procesos y herramientas están en el lugar. La mayor cantidad de partes críticas de esta fase es a menudo Análisis, Diseño y Administración de configuración por que necesitas una línea base para tus requerimientos y arquitectura, necesitas tener una Configuración funcionando bien. Administrar la infraestructura que permita desarrollar equipos que trabajan en paralelo. Concepción y elaboración son dos fases en que tener un mentor es la parte mas critica. Un mentor guía gente en cuales son las tareas más esenciales y cuales son las mejores prácticas para adoptar. El tutor realizar entrenamiento JIT para acelerar el aprendizaje y evitar la sobrecarga de información. Durante la fase de Elaboración puedes testar la arquitectura y funcionalidad clave; por otro lado necesitas asegurarte que el ambiente de test esta funcionando y corriendo. También necesitas tomar especial atención para efectivamente administrar el desarrollo iterativo, dado que típicamente no es obvio como desarrollar la arquitectura de manera incremental.
- **Fase de construcción:** No hay nuevos procesos o herramientas introducidas en esta fase. El enfoque aquí es producir el producto; por otro lado, el ambiente de desarrollo debe ser estable. En la fase de Construcción, la motivación es obtener un nuevo equipo de miembros en el proyecto por velocidad. Un tutor ayuda a sintonizar el proceso,

introducir nuevos miembros al equipo al proceso, y mantener el equipo de proyecto enfocado sobre los resultados del proyecto.

- **Fase de transición:** No hay nuevos procesos o herramientas introducidas. En la fase de transición, el enfoque nos lleva desde mejoras específicas al proceso a post-mortem del proyecto, permitiendo mejorar el proceso y herramientas para futuros proyectos. Experiencias del proyecto actual son obtenidas, sumariadas, y empaquetadas en una forma que futuros proyectos puedan usarlo.

Adoptando RUP en pequeños proyectos

Cuando se adopta el producto RUP en pequeños proyectos con quizás un pequeño grupo de gente, usas el mismo enfoque como para proyectos grandes. La diferencia no es el concepto del proceso o que estas haciendo, mas bien en la formalidad en que tus decisiones documentadas y el rigor del enfoque usado cuando se transfiere el conocimiento.

Vamos a ir a través de 5 pasos y apuntar algunas de las diferencias:

- **Medición:** Tal como para proyectos grandes, necesitas entender que deseas alcanzar para introducir el RUP, y que necesitas decidir los ítems de alta prioridad a donde enfocarse. Puedes, sin embargo, probablemente no documentar esto en un caso del negocio, pero mas bien ponerlo en un acuerdo verbal con el equipo del proyecto o escribir un memo usado como una base para decisiones de inversión.
- **Plan:** Necesitas aun un plan de cómo introducir el proceso de mejora. Desde que el esfuerzo es pequeño, no necesitas documentar el plan tan rigurosamente como para un proyecto grande.
- **Configurar y Adaptar:** Configurar el RUP, pero evitar construir plug ins estructurados desde que la mayoría de ellos son mas como esfuerzo no deseado. Hacer la adaptación mínima a las herramientas del ambiente.

Antes que adaptar el material de entrenamiento, usar un mentor para guiarte en la aplicación del proceso y herramientas para tu proyecto y para proveerte con un entendimiento de cómo debe diferenciarse desde que fue enseñado en alguna clase formal de entrenamiento.

- **Ejecutar:** distribuir el proceso y herramientas en tu proyecto, entrenar y hacer tutoría a los miembros del equipo, y correr tu proyecto. La principal diferencia entre grandes y pequeños proyectos es que en proyectos grandes puedes confiar más fuertemente en mentores para transferir el conocimiento (en vez de tener que depender exclusivamente en el entrenamiento formal), pero puedes aun necesitar un par de gente a tomar el entrenamiento.
- **Evaluar:** Tener una reunión de equipo para evaluar la performance del proyecto, y comparar con las expectativas dentro del proyecto. Identificar las oportunidades para futuras mejoras.

5.3. Rol y contenido de la documentación Arquitectónica

5.3.1 Visión general arquitectónica y manual de referencia arquitectónica

Esta referencia puede ser usada para producir dos tipos diferentes de documentación arquitectónica: visión general arquitectónica y manual de referencia arquitectónica

Una **visión general arquitectónica** Una visión general arquitectónica es idealmente producida tempranamente en el ciclo de desarrollo y sirve como punto de partida para el desarrollo. Una visión arquitectónica debe estar en un alto nivel de abstracción. Todas las funcionalidades y componentes de la arquitectura deben ser descritas pero las descripciones pueden carecer de detalle y precisión por que ellos usan un lenguaje natural más que notaciones formales.

Un **manual de referencia arquitectónico** describe una arquitectura en una detallada y precisa manera. Diferente de una visión general arquitectónica, un manual de referencia arquitectónico no es escrito en un punto particular de tiempo. Mas que un manual de referencia debe ser un documento vivo que es construido de forma colaborativa por el equipo de desarrollo como el desarrollo avanza. Como este desarrolla el manual de referencia puede ser utilizado para seguir el progreso del desarrollo de software y evaluar el impacto de requerimientos de cambios propuestos. Cuando se completa el manual de referencia es la base para el mantenimiento del sistema y mejoras. El manual de referencia debe ser actualizado cada vez que ocurran cambios para que siempre refleje la arquitectura actual.

Un manual de referencia necesita ser completo en el sentido que cada faceta del la arquitectura debe ser documentado. Aunque un manual de referencia puede tender a ser detallado y técnico en naturaleza el nivel de abstracción y precisión varia entre arquitecturas y proyectos. La arquitectura dentro de grandemente granulados componentes debe ser documentada en un separado

manual de referencia de componentes. El nivel de plenitud, formalidad, detalle y precisión usado para un particular aspecto de la arquitectura puede depender de los riesgos técnicos o de negocio.

Una documentación arquitectónica debe contener las siguientes secciones:

1. Introducción

Información general acerca del documento de arquitectura, otros documentos relacionados

2. Propósito del sistema

Propósito, características funcionales y cualitativas del sistema desde un punto de vista de caja negra.

2.1 Contexto : descripción del contexto del sistema y del problema que resuelve

2.2 Interfaz del sistema: servicios provistos por el sistema

2.3 Requerimientos no funcionales: principios de la arquitectura, cualidades de servicio, restricciones externas

3. Estructura

Vista lógica de la estructura estática de la arquitectura en términos de sus componentes, sus interconexiones, y las interfaces y operaciones ofrecidas por los componentes

3.1 Visión General : diagramas de arquitectura para la estructura completa, comentarios conteniendo racional, estilo arquitectónico, restricciones arquitectónicas y alternativas arquitectónicas

3.2 Componentes :

- Para cada componente: descripción de los componentes en términos de sus responsabilidades, interfaces que ofrece, componentes de colaboración, posibilidades de parametrización, y restricciones

3.3 Interfaces

Para cada componente de interfaz: descripción de la interfaz, sus operaciones y restricciones sobre el orden de operaciones, opcionalmente especificación de las operaciones

4. Comportamiento dinámico

Especificación del comportamiento del sistema, colaboración de los componentes para conseguir el comportamiento del sistema

4.1 Escenarios: diagrama(s) de arquitectura

Para cada escenario:

- Especificación del escenario como operación del sistema o especificación de caso de uso para algunos o todos los servicios provistos por el sistema
- Modelo de interacción de componentes del escenario

4.2 Mecanismos (opcional): explicación de y modelos de interacción para mecanismos importantes

5. Otras vistas

5.1 Vista de procesos (opcional) : diagramas de arquitectura de los procesos mostrando el mapeo de componentes a procesos

5.2 Vista de distribución (opcional): diagramas de arquitectura de la arquitectura de hardware mostrando el mapeo de componentes, procesos y archivos de código para nodos de hardware

5.3 Secciones adicionales (opcional): secciones para modelos adicionales y puntos de vista no especificados en esta plantilla

6. Framework conceptual

Definición conceptos específicos del dominio del problema y términos y sus relaciones

7. Conclusión: ventajas y limitaciones de arquitecturas con respecto a requerimientos no funcionales, problemas abiertos.

6. DOCUMENTACION DEL CASO PRÁCTICO

6.1. Introducción

El presente documento mostrará la forma de elaboración de la arquitectura de Software para un sistema de Gestión de Prestación de Servicios. En nuestra actualidad la mayoría de las empresas se dedican al rubro de prestación de servicios. Estas empresas tienen a su cargo un sin número de clientes, con los cuales realizan varios tipos de proyectos.

Los proyectos demandan un sin número de tareas, tiempo estimado de la finalización de dichas tareas pero lo más importante la administración del personal que estará conformando cada uno de estos proyectos, es decir controlar el número de horas y gastos reportados a cada uno de estos proyectos.

El personal que integra cada uno de estos proyectos posee diferentes niveles de experticia, por lo mismo su tiempo de colaboración en el mismo varía en base a ello.

La presente documentación nos mostrará el desarrollo la arquitectura de software basado en RUP para ello desarrollaremos los casos de uso principales y sus respectivos diagramas.

6.2. Propósito

El propósito de este sistema es poder administrar eficientemente los proyectos para los clientes que pueda poseer el negocio. En base a ello poder administrar los siguientes puntos:

- Administrar la relación de clientes.
- Administrar la relación de Proyectos por Cliente.
- Administrar las tarifas para el personal según el nivel de experticia.
- Generar reportes que indiquen el total de honorarios y gastos generados en un determinado proyecto.

6.3. Interfaces

El sistema propuesto no tiene diseñado interfaces.

6.4. Requerimientos no funcionales

Cualidades.

- El sistema permitirá registrar tanto las horas y gastos reportados por un personal para un determinado proyecto.
- El sistema permitirá generar tarifas por personal para una determinada categoría en un rango de tiempo específico.

Restricciones

- Los reportes Gerenciales previstos a desarrollar deberán ser extraídos de una tabla resumen, productos de un proceso batch.
- Dado que el sistema es cliente servidor, si se desea migrar a una versión web los reportes gerenciales presentarían un alto tiempo de respuesta para la generación de los mismos.

6.5. Representación de la Arquitectura

La arquitectura del siguiente caso práctico esta representado por las recomendaciones dadas por *Rational Unified Process (RUP)*. La especificación del sistema en UML ha sido dividida en las siguientes vistas:

- Vista de Casos de Uso
Describe los actores y sus casos de Uso del sistema, la cual representa las necesidades del usuario y describe el flujo de proceso.
- Vista Lógica de Procesos

Describe cada uno de los componentes a utilizar en el sistema.

- Vista de Desarrollo

6.6. Vista de Casos de Uso

6.6.1 Actores

La siguiente lista muestra los actores (Ver Ilustración 7) a utilizar en el desarrollo del sistema:

- RRHH
- Secretaria
- Finanzas
- Personal

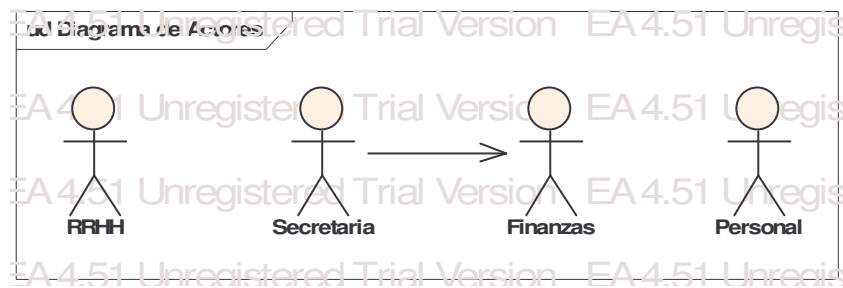


Ilustración 7: Actores

6.6.2 Casos de Uso

- a. Registrar Cliente. (Ver Ilustración 8)

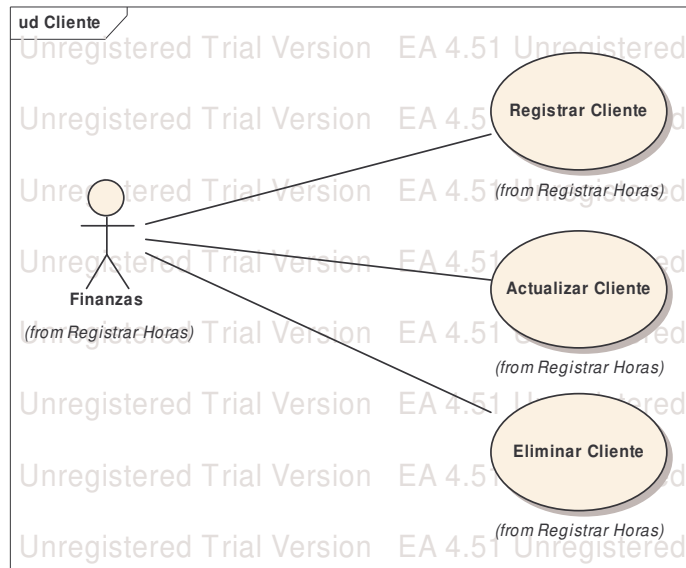


Ilustración 8: Caso de Uso – Registrar Cliente

b. Registrar Proyecto. (Ver Ilustración 9)

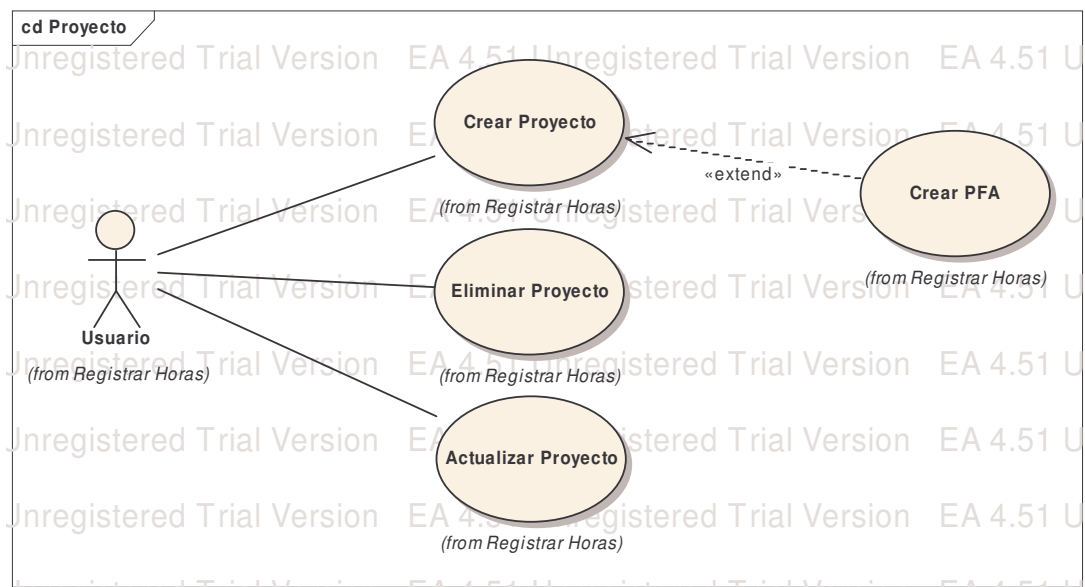


Ilustración 9: Caso de Uso – Registrar Proyecto

c. Registrar Horas y Gastos (Ver Ilustración 10)

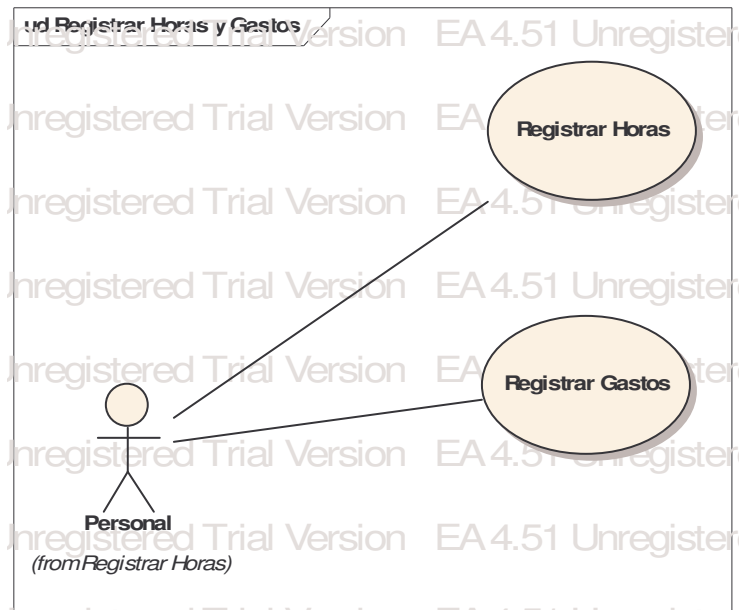


Ilustración 10: Caso de Uso – Registrar Horas

d. Registrar Usuario (Ver Ilustración 11)

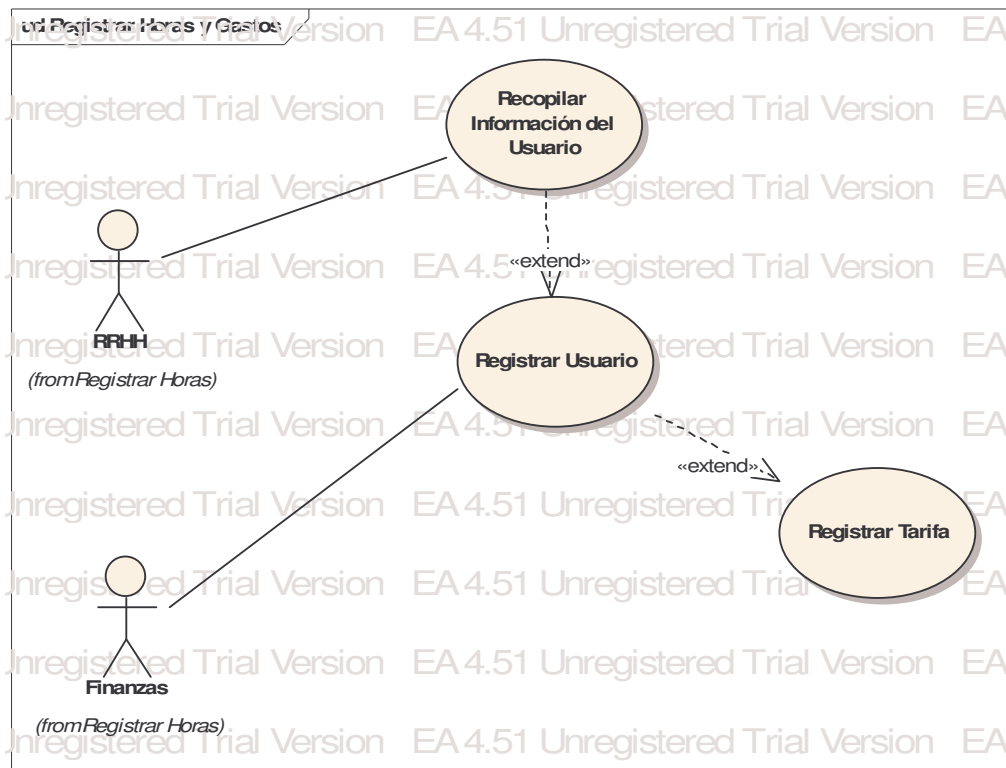


Ilustración 11: Caso de Uso – Registrar Usuario

6.6.3 Diagrama de Actores

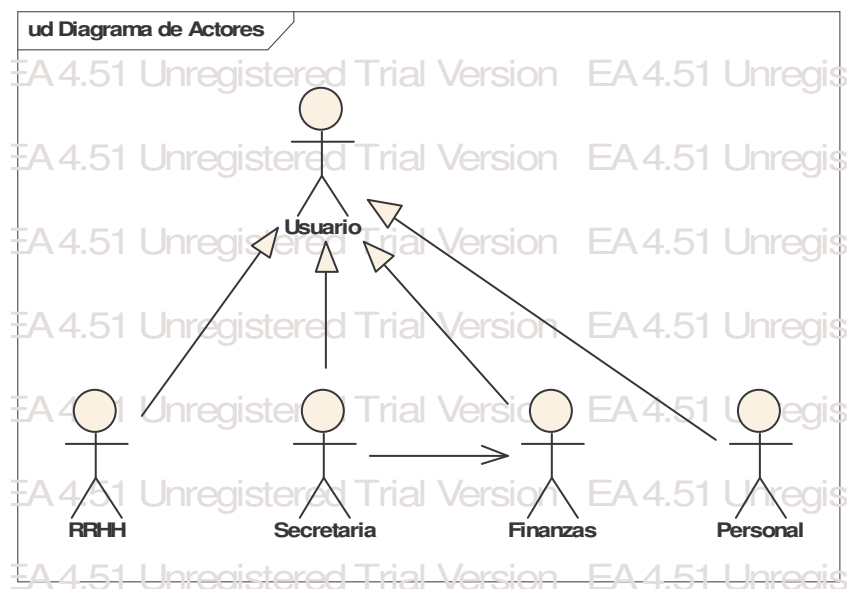


Ilustración 12: Diagrama de Actores

6.6.4 Diagrama de Clases

El sistema de Gestión de Prestación de Servicios presenta varios repositorios de información tales como:

- Clientes
- Proyecto
- PFA
- Personal
- Horas
- Gastos
- Tarifas

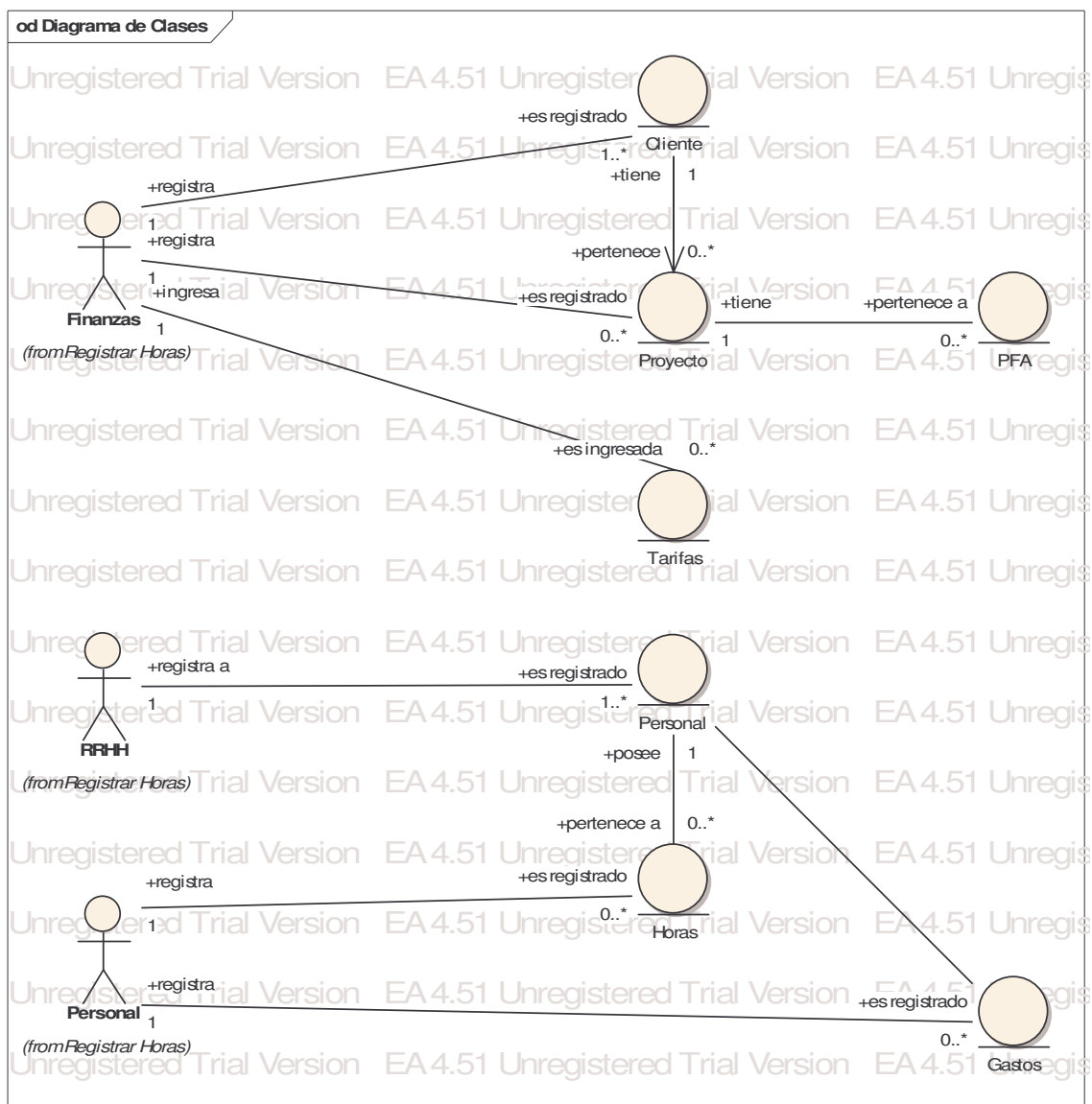


Ilustración 13: Diagrama de Clases

6.6.5 Diagrama de Paquetes

PARa el sistema de Gestión de Prestación de Servicios se definen 4 paquetes principales (Ver Ilustración 14):

- Módulo de Cliente
- Módulo de Proyecto
- Módulo de RRHH
- Módulo de Registro, en el cual se registran horas y Gastos.

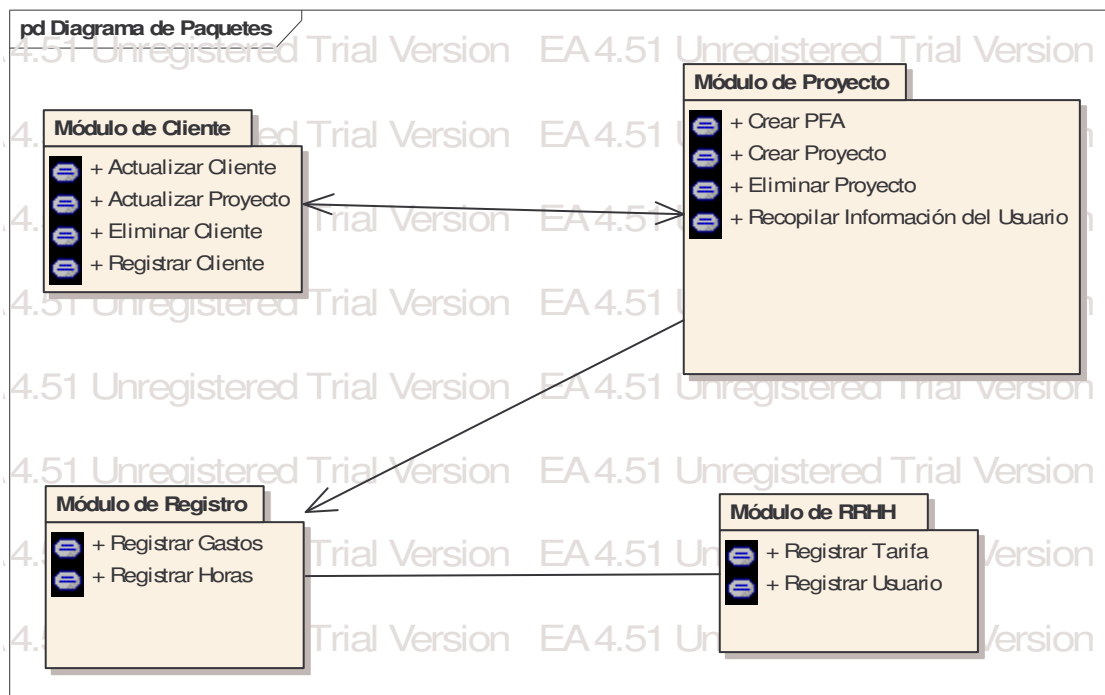


Ilustración 14: Diagrama de Paquetes

6.6.6 Descripción de los Componentes

En este caso práctico no se utilizaría.

6.7. Vista Lógica de Procesos

6.7.1 Diagrama de Actividades

- Registrar Cliente (Ver Ilustración 15)

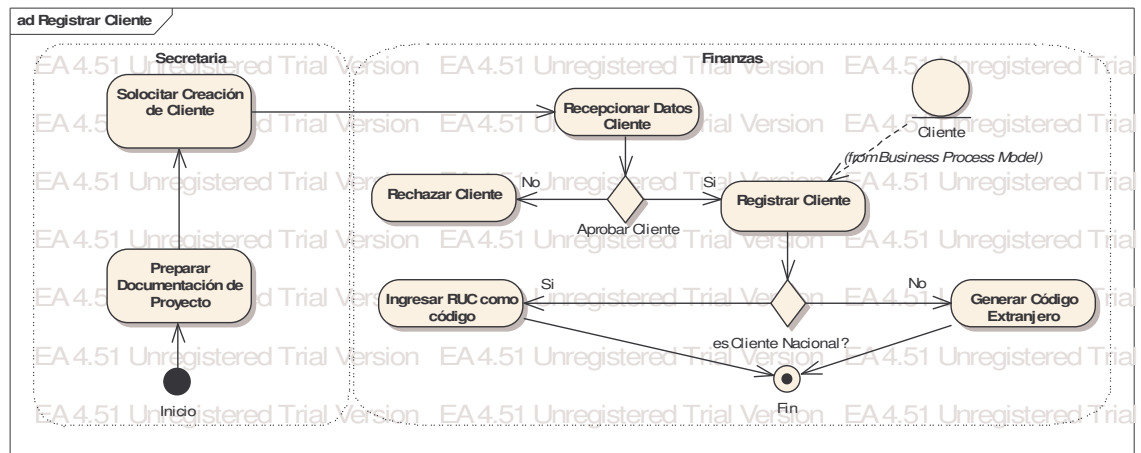


Ilustración 15: Diagrama de Actividades – Registrar Cliente

- Registrar Proyecto (Ver Ilustración 16)

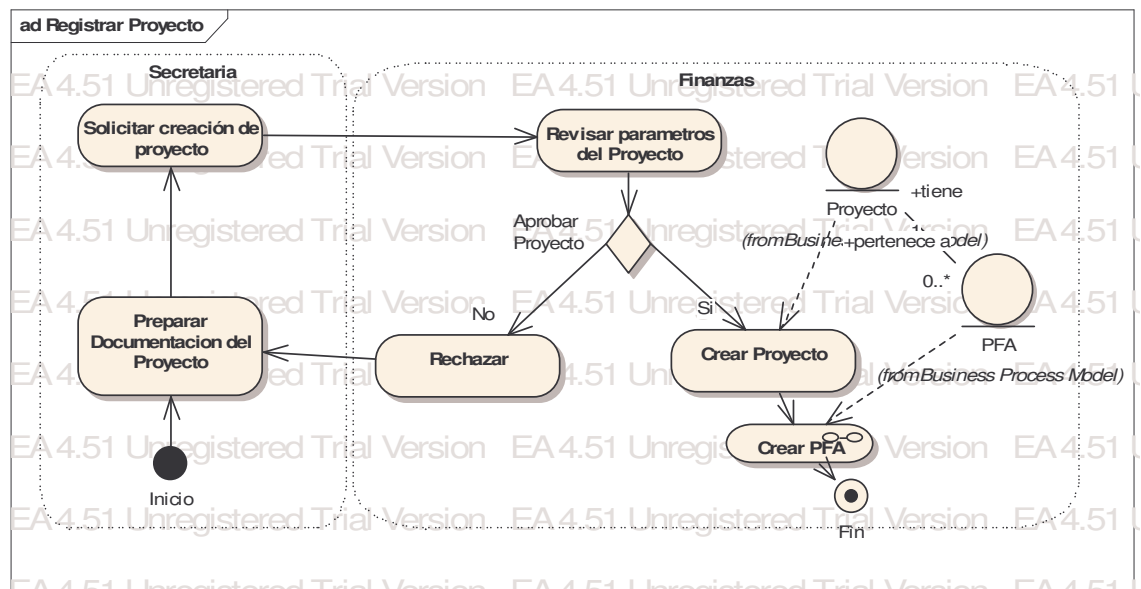


Ilustración 16: Diagrama de Actividades – Registrar Proyecto

- Registrar Horas (Ver Ilustración 17)

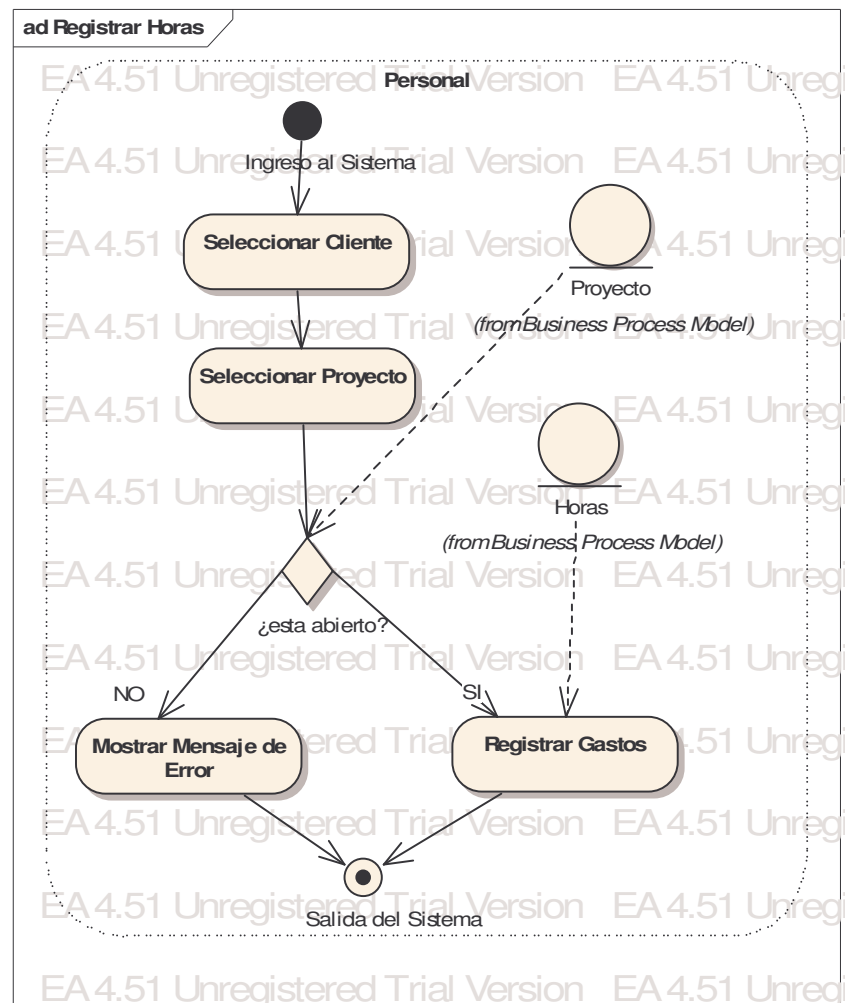


Ilustración 17: Diagrama de Actividades – Registrar Horas

- Registrar Gastos (Ver Ilustración 18)

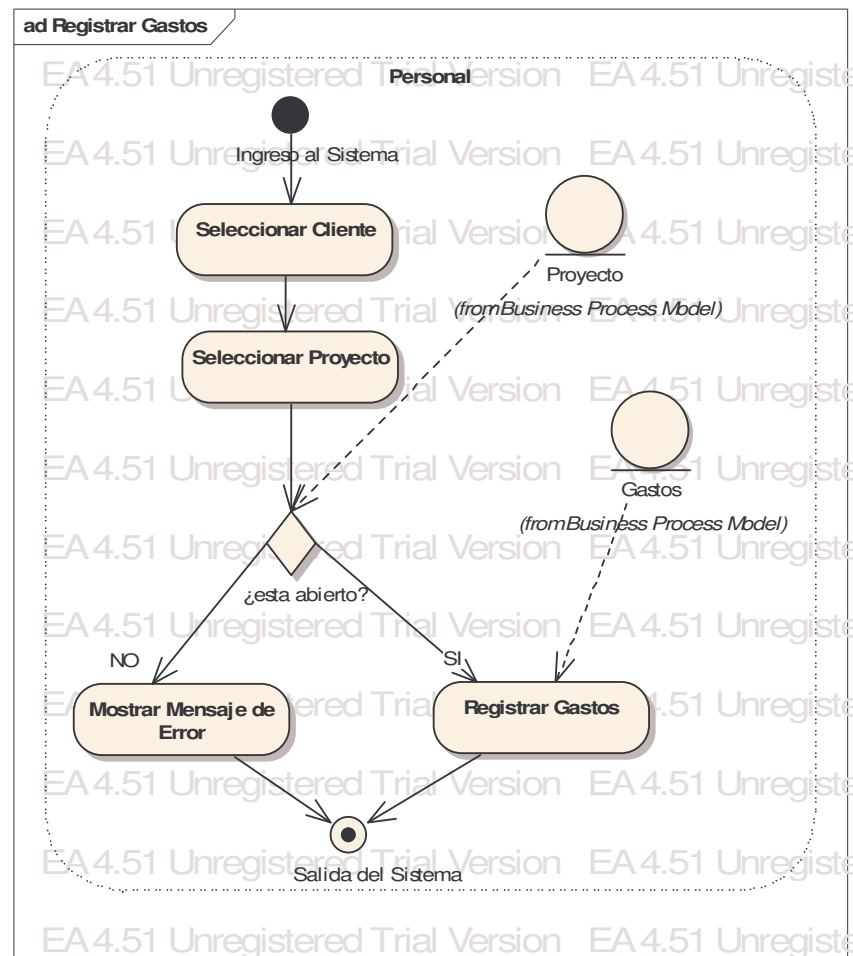


Ilustración 18: Diagrama de Actividades – Registrar Gastos

- Registrar Personal (Ver Ilustración 19)

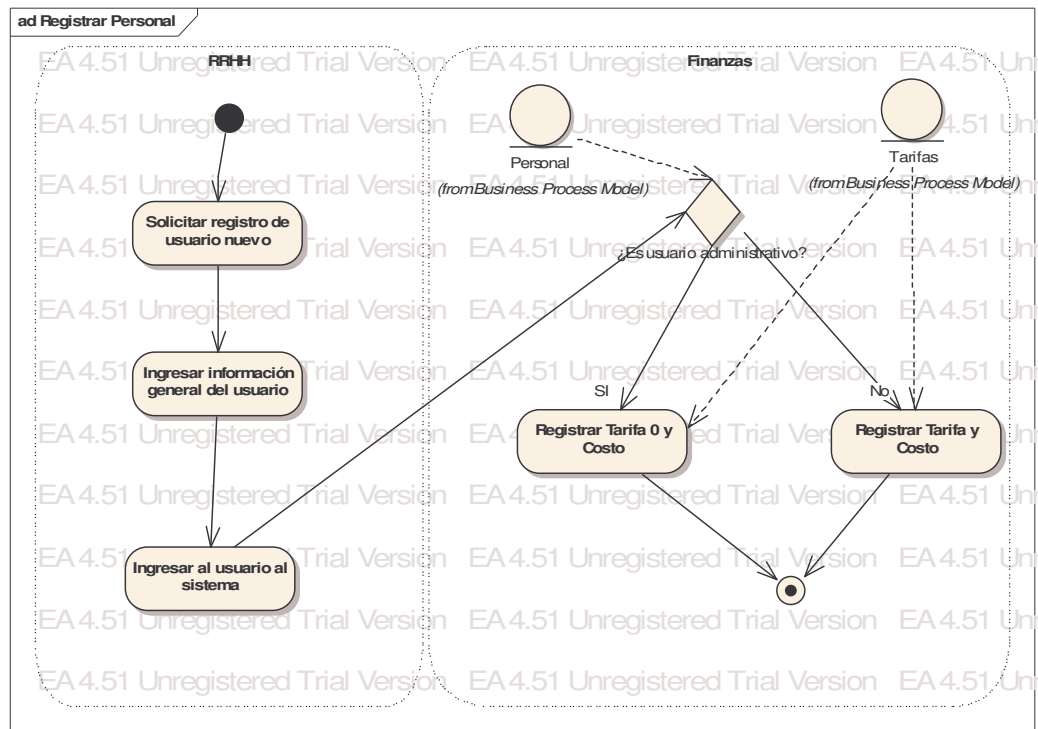


Ilustración 19: Diagrama de Actividades – Registrar Personal

6.8. Vista de Desarrollo

6.8.1 Modelo Conceptual

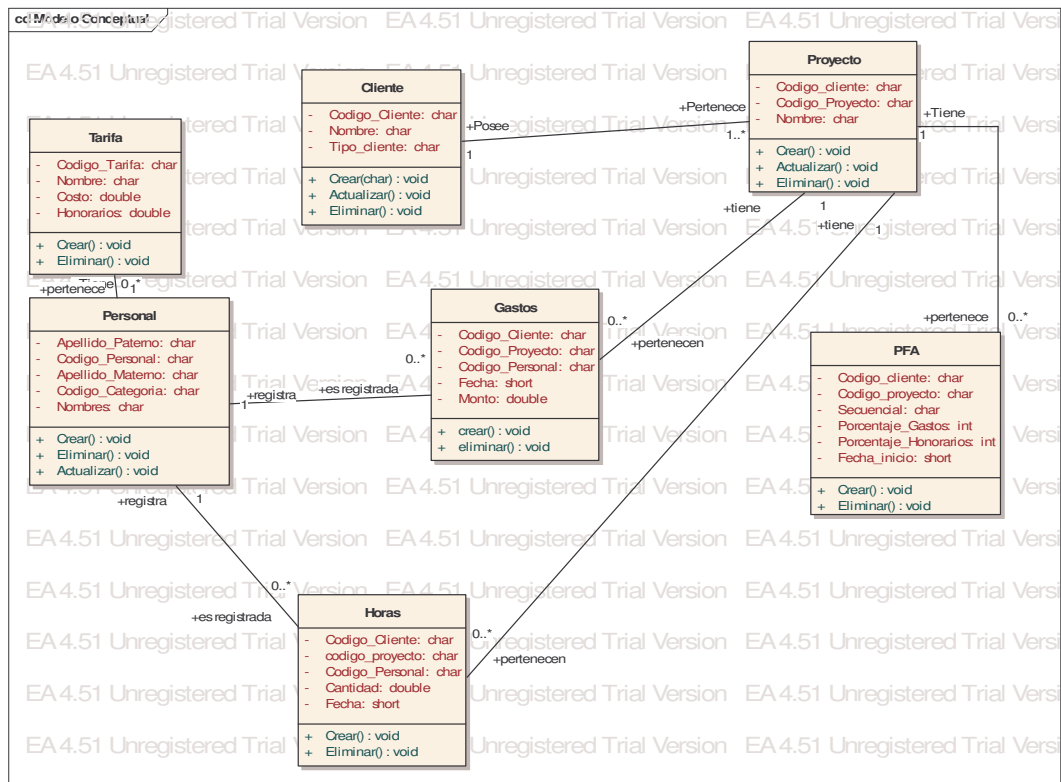


Ilustración 20: Modelo Conceptual

6.8.2 Diagrama de Clases

En el siguiente diagrama (Ver Ilustración 21) se muestra la información de las clases correspondiente a los paquetes de:

- Módulo de Clientes
- Módulo de Proyectos

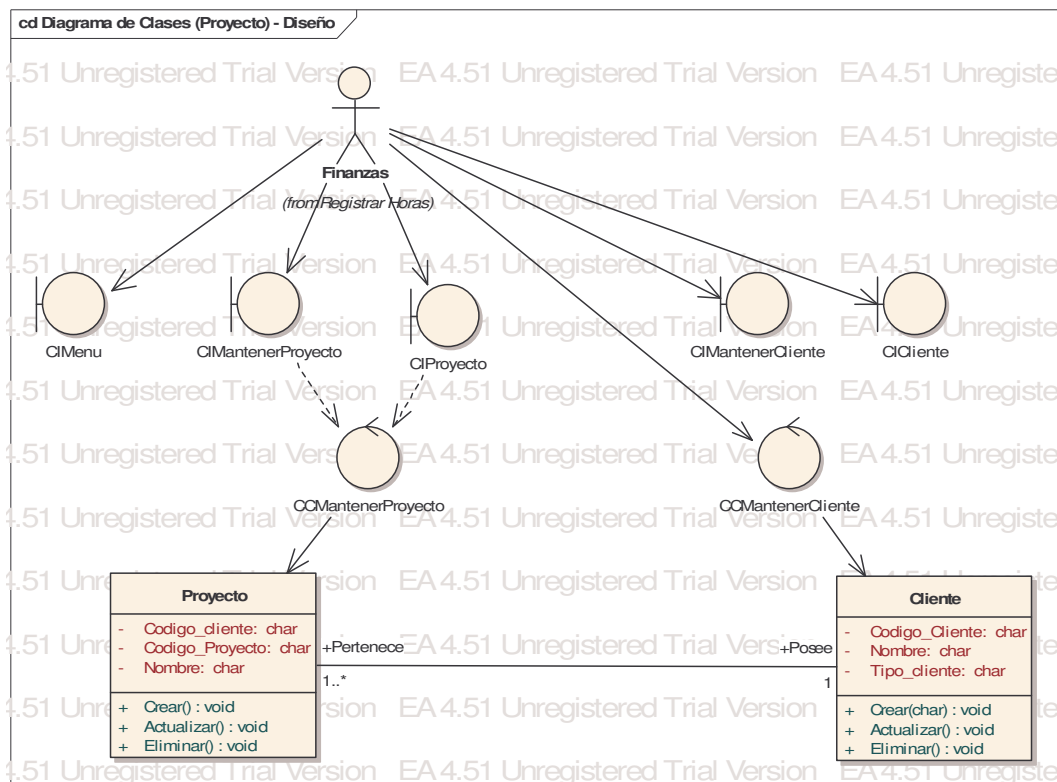


Ilustración 21: Diagrama de Clases – Etapa de Diseño

6.8.3 Diagrama de Secuencia

- Registrar Cliente (Ver Ilustración 22)

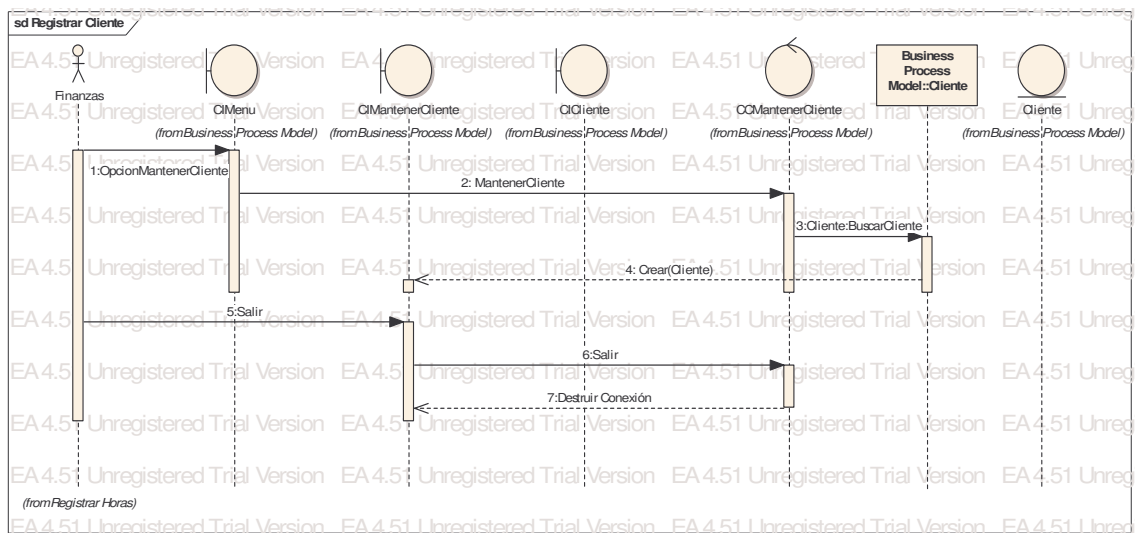


Ilustración 22: Diagrama de Secuencia – Registrar Cliente

- Registrar Proyecto (Ver Ilustración 23)

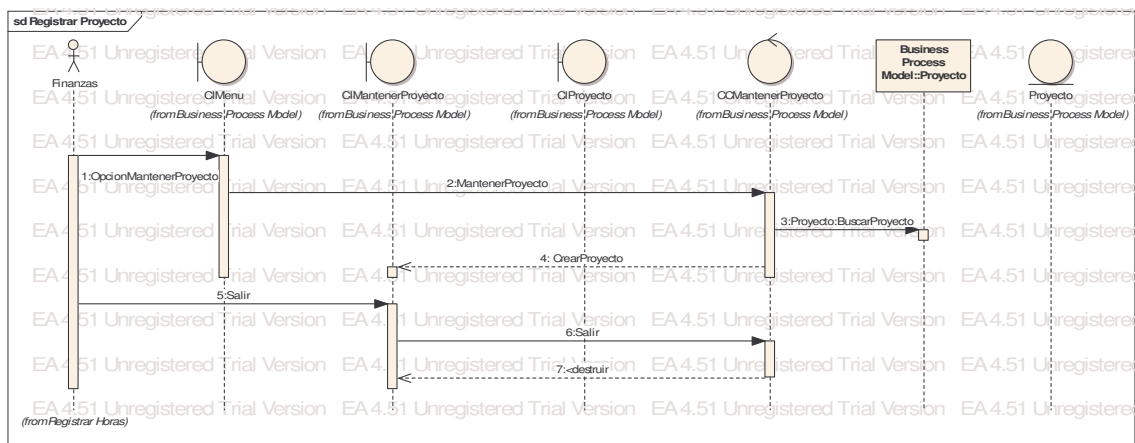


Ilustración 23: Diagrama de Secuencia – Registrar Proyecto

6.8.4 Diccionario de Clases

Aquí se describirá la información a las entidades definidas en el sistema de Gestión de Prestación de Servicios.

- Cliente
- Proyecto
- Personal
- Tarifas
- Horas

- Gastos

Nombre	Cliente		
Tipo	Entidad		
Descripción	Almacena la información a guardar sobre los clientes que solicitan la realización de un proyecto.		
Atributo	Tipo de Dato	Visibilidad	Valor Inicial
Codigo_cliente	Char	Privado	
Nombre	Char	Privado	
Tipo_Cliente	Char	Privado	

Nombre	Proyecto		
Tipo	Entidad		
Descripción	Almacena la información a guardar sobre los proyectos a realizar.		
Atributo	Tipo de Dato	Visibilidad	Valor Inicial
Codigo_cliente	Char	Privado	
Codigo_proyecto	Char	Privado	
Nombre	Char	Privado	

Nombre	Personal		
Tipo	Entidad		
Descripción	Almacena la información del personal que registrará sus horas y gastos en el sistema.		
Atributo	Tipo de Dato	Visibilidad	Valor Inicial
Codigo_personal	Char	Privado	
Apellido_paterno	Char	Privado	
Apellido_Materno	Char	Privado	
Nombres	Char	Privado	
Codigo_categoria	Char	Privado	

Nombre	Tarifa		
Tipo	Entidad		
Descripción	Almacena la información de las tarifas para el personal que labora en los diferentes proyectos.		
Atributo	Tipo de Dato	Visibilidad	Valor Inicial
Codigo_tarifa	Char	Privado	
Nombre	Char	Privado	
Honorarios	Double	Privado	
Costo	Double	Privado	

Nombre	Horas		
Tipo	Entidad		
Descripción	Almacena la información a guardar referente a las horas involucradas para los diferentes proyectos.		
Atributo	Tipo de Dato	Visibilidad	Valor Inicial
Codigo_cliente	Char	Privado	
Codigo_Proyecto	Char	Privado	
Codigo_personal	Char	Privado	
Cantidad	Double	Privado	
Fecha	Datetime	Privado	

Nombre	Gastos		
Tipo	Entidad		
Descripción	Almacena la información a guardar referente a los gastos involucrados para los diferentes proyectos.		
Atributo	Tipo de Dato	Visibilidad	Valor Inicial
Codigo_cliente	Char	Privado	
Codigo_Proyecto	Char	Privado	
Codigo_personal	Char	Privado	

Monto	Double	Privado	
Fecha	Datetime	Privado	

6.9. Vista de Implementación Física

El sistema de Gestión de Prestación de Servicios se basa en la plataforma Cliente Servidor. Dicha plataforma será la utilizada para el desarrollo de nuestro sistema planteado.

Para ello se requiere de un servidor de Base de Datos, Un Servidor de Componentes y finalmente las estaciones de trabajo (Ver Ilustración 24).

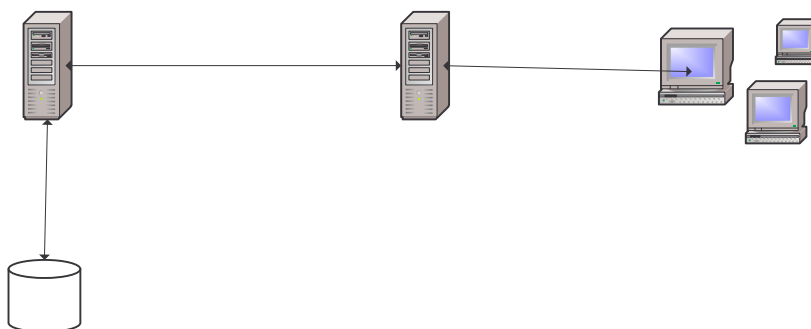


Ilustración 24: Diagrama Cliente Servidor

7. CONCLUSIONES

- La utilización de Arquitecturas de Software permite tener una visión clara de la estructura o estructuras del sistema que comprende elementos de software.
- La Arquitectura de Software debe ser distinguida del diseño de bajo nivel (Ej. Diseño de componentes internos y algoritmos) y implementación.
- Una arquitectura exitosa forma la plataforma para la ventaja estratégica. En contraste, la falta de arquitectura ata a la organización inevitablemente al pasado.
- El Arquitecto de software es responsable de la arquitectura de software, que incluyen las decisiones técnicas claves que contienen el diseño completo e implementación para el proyecto.
- La metodología utilizada en el caso práctico esta basado en un proceso ágil: RUP con baja ceremonia. Dicha metodología hace referencia a 4 vistas:
 - Vista de Casos de Uso
 - Vista Lógica de Procesos
 - Vista de Diseño
 - Vista de Implementación Física
- En general la utilización de Arquitecturas de Software permitirá tener una visión clara de los objetivos de negocio y tecnológicos dentro de una organización.

8. REFERENCIAS BIBLIOGRAFICAS

8.1. Referencias bibliográficas de libros

Rumbaugh, James y otros

1999. El proceso Unificado de desarrollo de Software. Madrid: Pearson Education, 2000

Pressman, Roger S.

2002. Ingeniería de Software: Un enfoque práctico. Madrid: McGraw-Hill/Interamericana de España, 2002

Shmuller, Joseph

1999. Aprendiendo UML en 24 Horas. Madrid: Pearson Education, 2000

Booch, Grady y otros

1999. El Lenguaje Unificado de Modelado. Madrid: Addison Wesley Iberoamericana, 2000

Per, Krol y otros

2003. The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP: Addison Wesley, 2003

Bass Clements, Kazman

2003. Arquitectura de Software en Práctica. USA: Addison Wesley, 2003

Microsoft Corporation

2003. Analyzing requirements and Defining Microsoft .NET Solutions Architectures, Exam 70-300. USA: Microsoft Press, 2003

8.2. Referencias bibliográficas de artículos

Malan and Creary

1999. Software Architecture in the Context. Fusion Newsletter. April, 2003

Sundblad, Sten & Sundblad, Per

2003. Architecture of an Autonomous Application, Volume18. Number 7. MSDN Magazine. Redmond, Washington.

Booch, Grady

2002. Grady Booch Discusses .NET and the Art of Software Development, Volume 17. Number 2. MSDN Magazine. Redmond, Washington.

Microsoft Corporation

2004. Enterprise Architecture Design and the Integrated Architecture Framework, Volume 1. Number 1. Journal. Redmond, Washington.

Microsoft Corporation

2004. Enterprise Architecture Design and the Integrated Architecture Framework, Volume 1. Number 3. Journal. Redmond, Washington.

IEEE Software

1995. Architectural Blueprint – The “4+1” View Model of Software Architecture, Number 12. IEEE Software. Vancouver, BC. Canada.

Hewlett Packard

2000. A Template for Documenting Software and Firmware Architectures, Version 1.3. Hewlett Packard. Vancouver. Canada.

8.3. Referencias bibliográficas de páginas Web

Fowler, Martin

2003. The New Methodology.

<http://www.martinfowler.com/articles/newMethodology.html#N10058>

:2003-04-01

Lasso, Adrián

2002. Arquitectura de Software

<http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art110.asp>

:2005-02-01

Medvidovic, Nenad

2002. Software Architectures

http://sunset.usc.edu/classes/cs612_2000/: 2002-04-17

Reynoso, Carlos Billy

2004. Arquitectura de Software

<http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art110.asp>

:2003-05-10

Platt, Michael

2002. Microsoft Architecture Overview.

<http://msdn.microsoft.com/architecture/default.aspx?pull=/library/en-us/dnea/html/eaarchover.asp> : 2002-07-01

Sun Microsystems

2002. Jini Technology Architectural Overview.

<http://wwws.sun.com/software/jini/whitepapers/architecture.html>

:2002-10-01

Carnegie Mellon, Software Engineering Institute

2004. Essays on Software Architecture.

<http://www.sei.cmu.edu/architecture/essays.html> : 2002-06-07

Bredemeyer Consulting

2004. Software Architecture Resources.

<http://www.bredemeyer.com/papers.htm> : 2004-01-27

8.4. Otras referencias

IBM Software Corporation

2005. Rational Unified Process 6.13

Microsoft Corporation

2005. Microsoft Solutions Framework MSF Agile Beta

8.5. Referencias bibliográficas de tesis

Carranza Avalos, Zlatiel

1998. Método de Análisis Y Diseño de Software. Tesis de Maestría.

Escuela de Postgrado, Facultad de Ingeniería de Sistemas, Universidad de Lima. Lima.

Tekinerdogan, Bedir

2000. Synthesis-Based Software Architecture Design. Tesis de PhD.

Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands.

Piekarski, Wayne

2004. Interactive 3d Modelling in Outdoor Augmented Reality Worlds. Tesis de PhD.

School of Computer and Information Science, University of South Australia, Adelaide, South Australia.

LISTA DE FIGURAS

<i>Ilustración 1: Arquitectura de Software en el Contexto</i>	34
<i>Ilustración 2: Alineamiento de Sistemas y Negocios</i>	36
<i>Ilustración 3: Artefactos y Responsabilidades del Arquitecto según RUP</i>	41
<i>Ilustración 4: Mapa de Procesos para comparación de procesos.</i>	72
<i>Ilustración 5: Procesos Ágiles sobre el Mapa de Procesos.</i>	74
<i>Ilustración 6: Configuraciones RUP sobre el mapa de Procesos.</i>	76
<i>Ilustración 7: Actores</i>	88
<i>Ilustración 8: Caso de Uso – Registrar Cliente</i>	89
<i>Ilustración 9: Caso de Uso – Registrar Proyecto</i>	89
<i>Ilustración 10: Caso de Uso – Registrar Horas</i>	90
<i>Ilustración 11: Caso de Uso – Registrar Usuario</i>	91
<i>Ilustración 12: Diagrama de Actores</i>	91
<i>Ilustración 13: Diagrama de Clases</i>	92
<i>Ilustración 14: Diagrama de Paquetes</i>	93
<i>Ilustración 15: Diagrama de Actividades – Registrar Cliente</i>	94
<i>Ilustración 16: Diagrama de Actividades – Registrar Proyecto</i>	94
<i>Ilustración 17: Diagrama de Actividades – Registrar Horas</i>	95
<i>Ilustración 18: Diagrama de Actividades – Registrar Gastos</i>	96
<i>Ilustración 19: Diagrama de Actividades – Registrar Personal</i>	97
<i>Ilustración 20: Modelo Conceptual</i>	97
<i>Ilustración 21: Diagrama de Clases – Etapa de Diseño</i>	98
<i>Ilustración 22: Diagrama de Secuencia – Registrar Cliente</i>	99
<i>Ilustración 23: Diagrama de Secuencia – Registrar Proyecto</i>	99
<i>Ilustración 24: Diagrama Cliente Servidor</i>	102

LISTA DE TABLAS

<i>Tabla 1: Diferencias entre Programación de Software e Ingeniería de Software</i>	<i>17</i>
<i>Tabla 2: Comparando Arquitectura y Diseño</i>	<i>33</i>

REFERENCIAS BIBLIOGRAFICAS

Bass Clements, Kazman

(2003) Arquitectura de Software en Práctica. USA: Addison Wesley, 2003

Booch, Grady

(2002) Grady Booch Discusses .NET and the Art of Software Development, Volume 17. Number 2. MSDN Magazine. Redmond, Washington.

Bredemeyer Consulting

(2004) Software Architecture Resources.

<http://www.bredemeyer.com/papers.htm> : 2004-01-27

Carranza Avalos, Zlatiel

(1998) Método de Análisis Y Diseño de Software. Tesis de Maestría. Escuela de Postgrado, Facultad de Ingeniería de Sistemas, Universidad de Lima. Lima.

Carnegie Mellon, Software Engineering Institute

(2004) Essays on Software Architecture.

<http://www.sei.cmu.edu/architecture/essays.html> : 2002-06-07

Fowler, Martin

(2003) The New Methodology.

<http://www.martinfowler.com/articles/newMethodology.html#N10058> : 2003-04-01

Lasso, Adrián

- (2002) Arquitectura de Software
<http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art110.asp> : 2005-02-01

Malan and Creary

- (1999) Software Architecture in the Context. Fusion Newsletter. April, 2003

Medvidovic, Nenad

- (2002) Software Architectures
http://sunset.usc.edu/classes/cs612_2000/:2002-04-17

Microsoft Corporation

- (2003) Analyzing requirements and Defining Microsoft .NET Solutions Architectures, Exam 70-300. USA: Microsoft Press, 2003

Booch, Grady y otros

- (1999) El Lenguaje Unificado de Modelado. Madrid: Addison Wesley Iberoamericana, 2000

Piekarski, Wayne

- (2004) Interactive 3d Modelling in Outdoor Augmented Reality Worlds. Tesis de PhD.
School of Computer and Information Science, University of South Australia, Adelaide, South Australia.

Platt, Michael

- (2002) Microsoft Architecture Overview.
<http://msdn.microsoft.com/architecture/default.aspx?pull=/library/en-us/dnea/html/eaarchover.asp> : 2002-07-01

Reynoso, Carlos Billy

(2004) Arquitectura de Software

<http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art110.asp> : 2003-05-10

Rumbaugh, James y otros

(1999) El proceso Unificado de desarrollo de Software. Madrid: Pearson Education, 2000

Rumbaugh, James y otros

(1999) El Lenguaje Unificado de Modelado. Manual de Referencia. Madrid: Pearson Education, 2000

Shmuller, Joseph

(1999) Aprendiendo UML en 24 Horas. Madrid: Pearson Education, 2000

Sun Microsystems

(2002) Jini Technology Architectural Overview.

<http://www.sun.com/software/jini/whitepapers/architecture.html> :
2002-10-01

Sundblad, Sten & Sundblad, Per

(2003) Architecture of an Autonomous Application, Volume18. Number 7. MSDN Magazine. Redmond, Washington.

Tekinerdogan, Bedir

(2000) Synthesis-Based Software Architecture Design. Tesis de PhD.

Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands.