UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

PROPUESTA DE UNA METODOLOGÍA PARA LA CONSTRUCCIÓN DE SISTEMAS INFORMÁTICOS EN UN ÓRGANO DE IMPARTICIÓN DE JUSTICIA

INFORME DE TRABAJO PROFESIONAL QUE PARA OBTENER EL TÍTULO DE INGENIERO EN COMPUTACIÓN PRESENTA:

ERIK LEÓN MENDOZA

ASESOR: M.I. AURELIO ADOLFO MILLÁN NÁJERA

CIUDAD UNIVERSITARIA, MARZO DEL 2015

AGRADECIMIENTOS

A mis padres, cuyo esfuerzo me permitió convertirme en un profesionista.

A todos los involucrados en la Universidad Nacional Autónoma de México, cuyo trabajo diario permiten el progreso de nuestra Nación mediante la formación de estudiantes o investigación científica.

A la Maestra Ana Juárez Ariza, que depositó su confianza en mí para poder participar en proyectos de gran importancia nacional; una oportunidad soñada para un recién egresado.

A mis colaboradores en la Suprema Corte de Justicia de la Nación, que me permitieron recibir parte de su conocimiento; además de su no menos importante amistad.

A mi equipo de desarrollo, análisis y pruebas; un grupo envidiable de personas talentosas que hicieron mi estancia mucho más grata con su amistad.

A mi familia y amigos, por todo su apoyo durante tantos años.

... a Tekton, por regresarme a la luz en tiempos de oscuridad.

PREFACIO

Al terminar mis estudios universitarios, me incorporé al mundo laboral esperando que los alrededor de 13 años que tenía de experiencia escribiendo código en diversos lenguajes de programación me serviría para dar este salto en mi vida, sin embargo no fue suficiente. El conocimiento que había adquirido en la academia y por auto-enseñanza era limitado, había mucho por aprender.

La arquitectura del primer sistema en que colaboré era abrumadora, no había visto algo así antes. Las colaboraciones en equipo eran de una forma distinta, algo difícil de emular en las instituciones de educación. Había que enfrentarse a problemas más allá de las cuestiones técnicas. Siendo mi primer trabajo, lo más sensato que se me ocurrió fue seguir órdenes al pie de la letra, realizar de la mejor manera las responsabilidades que se me encomendaron.

Al transcurrir varios meses, sentía que mi colaboración era limitada; que estaba desperdiciando mi potencial. Me di a la tarea de analizar cómo se podían mejorar los proyectos que estábamos realizando, ¿cómo se podía trabajar de forma más eficiente? e incluso ¿cómo aprovechar nuevas tecnologías?. Sin embargo no lo dejé sólo en un análisis, lo fui convirtiendo en propuestas que en poco tiempo se convirtieron en realidad.

¿De qué se trata este reporte laboral?

La forma en la que mi equipo y yo dimos vida a los proyectos seguía una metodología que no se podía encontrar en su totalidad en un libro, era una combinación de procedimientos y prácticas ya existentes en otras metodologías que se adaptaron a nuestras necesidades y dieron excelentes resultados. Son esos procedimientos y prácticas los que describo en este documento, con el objetivo de hacer más corta la brecha existente entre los conocimientos adquiridos en la academia y lo necesario para colaborar en proyectos de gran tamaño.

Debe considerarse este documento como una herramienta de referencia, una recopilación de información de varias fuentes con comentarios adicionales; que en combinación dan como resultado una nueva metodología que fue utilizada no únicamente en un sistema, sino en varios y de diversa naturaleza; siempre obteniendo resultados satisfactorios. Sin embargo, dicha metodología no es una guía para seguir de forma precisa; necesita de ajustes que se adapten a la infinidad de problemas que se pueden enfrentar al desarrollar sistemas informáticos.

Organización del reporte laboral

- Capítulo 1, Introducción. Una breve descripción del origen del problema planteado y que dio como resultado la metodología descrita a lo largo del documento. También se

mencionan algunos antecedentes que denotan la importancia de encontrar una solución exitosa.

- Capítulo 2, Desarrollo.
    - Ingeniería de software. Se describe un marco teórico de importancia fundamental para la construcción de cualquier sistema informático.
    - Recolección y análisis de requerimientos. Algunas recomendaciones de cómo abordar la etapa inicial del sistema, con el objetivo de minimizar problemas en el futuro.
    - Modelado y diseño de la arquitectura. Denota principios básicos que se recomiendan seguir independientemente de la arquitectura elegida para su próxima implementación. A su vez, se hace mención de algunas de las arquitecturas más comunes en la actualidad, y que fueron utilizadas para resolver el problema planteado en el primer capítulo.
    - Desarrollo seguro de aplicaciones. Probablemente el tema de mayor importancia. Hoy en día los mayores problemas en el desarrollo de sistemas informáticos conciernen a la seguridad; problema que tiene como origen la carencia o ausencia de buenas prácticas. Se proporcionan algunas recomendaciones al respecto que tienen importancia vital en momentos previos a la etapa de implementación, así como en las etapas siguientes.
    - Implementación del software. Se proporcionan conceptos básicos necesarios para una implementación adecuada, así como recomendaciones pertinentes de acuerdo a las diversas capas que comúnmente son utilizadas en la implementación de sistemas informáticos.
- Por último, comentario acerca de mi participación profesional relacionada a la realización de este documento, así como algunas conclusiones.

IMPORTANTE. El lector notará la ausencia de ejemplos prácticos que ilustren la metodología descrita a lo largo de este documento; lo cual obedece a la naturaleza confidencial de los proyectos en los que fue aplicada la metodología. Todo lo descrito a lo largo de los diversos capítulos recibió un tratamiento especial para evitar revelar detalles sensibles que pudieran representar riesgos a la institución dueña de las aplicaciones.

CONTENIDO

1. INTRODUCCIÓN ... - 1 -
1.1 Antecedentes ... - 1 -
1.2 La Dirección General de Tecnologías de la Información ... - 3 -
1.3 Problemática y objetivo ... - 3 -
2. DESARROLLO ... - 4 -
2.1 Ingeniería de software ... - 4 -
2.1.1 Proceso de software ... - 5 -
2.1.2 Atributos del buen software ... - 6 -
2.1.3 Modelos de proceso de software ... - 10 -
2.1.4 Iteración de procesos ... - 13 -
2.1.5 Actividades del proceso ... - 13 -
2.2 Recolección y análisis de requerimientos ... - 17 -
2.3 Modelado y diseño de la arquitectura ... - 18 -
2.3.1 Importancia ... - 18 -
2.3.2 Principios del diseño de la arquitectura ... - 19 -
2.3.3 Principios del diseño orientado a objetos ... - 19 -
2.3.4 Arquitecturas cliente-servidor ... - 22 -
2.3.5 Componentes de software para arquitecturas cliente-servidor ... - 23 -
2.3.6 Arquitectura basada en componentes ... - 24 -
2.3.7 Arquitectura en capas ... - 25 -
2.4 Desarrollo seguro de aplicaciones ... - 26 -
2.4.1 Medidas fundamentales para una seguridad efectiva ... - 27 -
2.4.2 Un nuevo enfoque de cara a la seguridad ... - 27 -
2.5 Implementación del software ... - 28 -
2.5.1 Desarrollo seguro de software ... - 28 -
2.5.2 Security Development Lifecycle ... - 28 -
2.5.3 Almacenamiento de datos ... - 32 -
2.5.4 Acceso a datos ... - 36 -
2.5.5 Capa de negocios o dominio ... - 39 -
2.5.6 Capa de presentación ... - 42 -
2.6 Consolidación de la metodología propuesta ... - 46 -
2.6.1 Pilares de la metodología ... - 48 -
PARTICIPACIÓN PROFESIONAL ... - 50 -
CONCLUSIONES ... - 51 -
BIBLIOGRAFÍA ... - 52 -

1. INTRODUCCIÓN

1.1 Antecedentes

La división de poderes es uno de los elementos imprescindibles en la organización del Estado. Tiene por objeto evitar el abuso del poder y preservar los derechos del hombre. De esta forma, se separan las funciones de los órganos públicos en tres categorías generales: legislativas (Poder Legislativo), administrativas (Poder Ejecutivo) y jurisdiccionales (Poder Judicial).

En México el Poder Judicial de la Federación (PJF) es el guardián de la constitución, el protector de los derechos fundamentales y el árbitro que dirime las controversias, manteniendo el equilibrio necesario que con base en un estado de derecho.

El Poder Judicial de la Federación (PJF), conforme al artículo 1° de su Ley Orgánica, se integra y ejerce por:

- La Suprema Corte de Justicia de la Nación
- El Tribunal Electoral
- Los Tribunales Colegiados de Circuito
- Los Tribunales Unitarios de Circuito
- Los Juzgados de Distrito
- El Consejo de la Judicatura Federal

La Suprema Corte de Justicia de la Nación (SCJN) es el Máximo Tribunal Constitucional del país y cabeza del Poder Judicial de la Federación. Tiene entre sus responsabilidades defender el orden establecido por la Constitución Política de los Estados Unidos Mexicanos; mantener el equilibrio entre los distintos Poderes y ámbitos de gobierno, a través de las resoluciones judiciales que emite; además de solucionar, de manera definitiva, asuntos que son de gran importancia para la sociedad. En esa virtud, y toda vez que imparte justicia en el más alto nivel, es decir, el constitucional, no existe en nuestro país autoridad que se encuentre por encima de ella o recurso legal que pueda ejercerse en contra de sus resoluciones.

La impartición de justicia en el estado mexicano requiere de un estricto seguimiento de los casos que ingresan al Poder Judicial de la Federación para su registro, estudio, propuesta de resolución y fallo. Por ello, los órganos del PJF deben integrar los elementos jurídicos, administrativos y tecnológicos que coadyuven a la plena gestión de los asuntos que les competen.

El siglo XX fue escenario de avances sin precedentes en la ciencia y la tecnología. En cuanto al manejo de información, es incuestionable que la humanidad se ha beneficiado de ese desarrollo. Aunque en nuestro país algunos sectores de la población no disfrutan de tales beneficios, gradualmente crece el número de usuarios; por lo general, la limitación económica es el principal obstáculo para que el total de la población se vea beneficiada por tales beneficios.

En la transición del siglo XX al siglo XXI, inmerso en las tendencias de la globalización mundial, nuestro país se suma al aprovechamiento de las tecnologías de información y comunicación, las cuales permiten procesar y administrar grandes volúmenes de información, enviar y recibir datos, así como consultarlos en dispositivos externos e independientes. Las sociedades alrededor del mundo generan grandes cantidades de información que mediante las tecnologías de información se ha buscado almacenar y explotar en beneficio de los seres humanos, permitiendo acortar barreras entre diferentes naciones, dando como resultado una revolución cultural.

Consciente de la importancia de las herramientas informáticas, el PJF ha sido pionero en la utilización de equipos electrónicos, lo que ha implicado la capacitación permanente de su personal, el diseño de estrategias para su óptimo aprovechamiento, además del desarrollo de software propio. El beneficio de estas tecnologías en la impartición de justicia es inobjetable.

El uso de las tecnologías de la información permite la gestión de la información como un proceso que incluye operaciones como extracción, manipulación, tratamiento, depuración, conservación, acceso y/o colaboración de la información (adquirida a través de diferentes fuentes) y que gestiona el acceso y los derechos de los usuarios sobre la misma.

Una aproximación a un sistema de gestión de los asuntos que competen al PJF debe considerar la naturaleza de los mismos, su origen, su jurisdicción, su materia, el tipo de recurso y otras características que la ciencia jurídica les atribuye.

Considerando lo anterior, es posible conceptualizar una secuencia de procedimientos jurídicos y administrativos para cada expediente que ingresa a uno de los órganos del PJF En esta definición conceptual, es preciso recurrir a los usuarios de los procesos referidos para que coadyuven en la plena sistematización de dichos procedimientos.

Las áreas usuarias son de índole diversa, desde las estrictamente administrativas hasta las eminentemente jurídicas, por ello los perfiles son diferentes y diferentes son los enfoques de los tramos de gestión del proceso.[1]

[1] Suprema Corte de Justicia de la Nación. (Enero del 2015). *Portal web de la Suprema Corte de Justicia de la Nación*. Obtenido de www.supremacorte.gob.mx, con comentarios adicionales por parte de Erik León Mendoza.

1.2 La Dirección General de Tecnologías de la Información

La misión de la Dirección General de Tecnologías de la Información, de conformidad con el artículo 140 del Reglamento Interior de la Suprema Corte de Justicia de la Nación, es:

I. Proponer los lineamientos en materia de tecnología de la información, a fin de mantener a la vanguardia la infraestructura informática y de redes de comunicación de la Suprema Corte.

II. Planear, diseñar, mantener y supervisar la operación de los sistemas de información que requieran los órganos que integran a la Suprema Corte.

III. Elaborar estudios técnicos en materia de infraestructura, recursos, desarrollos y sistemas tecnológicos, a fin de determinar las necesidades correspondientes.

IV. Proporcionar los servicios de mantenimiento a las redes, sistemas, equipo informático, comunicación y digitalización de los órganos de la Suprema Corte y, en su caso, a otros órganos del Poder Judicial.

V. Mantener comunicación constante con las Ponencias, con la Secretaría General de Acuerdos y con la Subsecretaría General de Acuerdos a fin de que la informática jurídica responda de manera eficaz y eficiente a las necesidades tecnológicas de la Suprema Corte.

VI. Llevar el control administrativo de las asignaciones presupuestales a su cargo, en coordinación con la Dirección General de Presupuesto y Contabilidad.

VII. Ejecutar y actualizar los mecanismos de seguridad informática y vigilar su adecuado funcionamiento. [2]

1.3 Problemática y objetivo

Anterior a la fecha 1 de diciembre de 2012, la Suprema Corte de Justicia de la Nación contaba con un sistema para el control y seguimiento de expedientes desarrollado por la Dirección General de Informática en el año de 1995, sobre la plataforma de desarrollo Visual Basic 2 y base de datos Access con componentes altamente acoplados. La plataforma ya no contaba, para la fecha mencionada previamente, con soporte por parte del proveedor de la herramienta.

[2] Suprema Corte de Justicia de la Nación. (Enero del 2015). *Portal web de la Suprema Corte de Justicia de la Nación*. Obtenido de www.supremacorte.gob.mx

Debido a la forma en que fue construido el sistema informático y el tipo de tecnología que se utilizó para ser implementado, el sistema causaba las siguientes situaciones problemáticas:

- Alto esfuerzo para el mantenimiento por el tipo de tecnología implementado y la falta de documentación.
- Dependencia del poco personal que conoce el sistema.
- Fallas técnicas significativas.
- Disminución considerable de sus niveles de servicio.
- Pérdidas de información y múltiples suspensiones de operación.
- El volumen de datos causaba ineficiencia.
- Redundancia de funcionalidades para soportar ambiente operativo y de consulta.
- Transacciones sin tolerancia a fallas.

Dentro de la SCJN, las áreas: Presidencia, Ponencias, Secretaría General de Acuerdos, Subsecretaría General de Acuerdos, Secretaría General de Acuerdos de la Primera Sala y Secretaría de Acuerdos de la Segunda Sala; son principalmente quienes requerían mejorar y acelerar la gestión, seguimiento y difusión electrónica de expedientes judiciales. Así es como el Sistema de Informática Jurídica (SIJ) de la SCJN fue diseñado para ser un sistema de información que proporcionara el cumplimiento de los objetivos buscados por las áreas mencionadas previamente además de permitir un fácil mantenimiento y con la flexibilidad de expansiones que complementaran al sistema de acuerdo a los requisitos en el flujo de trabajo que la SCJN tiene día a día.

2. DESARROLLO

2.1 Ingeniería de software

El software es un producto que involucra a distintas personas para su diseño y construcción; no es un trabajo sólo de programadores, sino que se requiere de varios perfiles dentro del equipo involucrado. A pesar de la existencia de varios perfiles, todas esas personas involucradas pueden ser definidas bajo el término ingeniero de software, ya que el propósito de cada una es darle vida a dicho software, el cual engloba programas que se ejecutan dentro de una computadora de cualquier tamaño y arquitectura, documentos en formas virtuales o físicos, y datos que combinan texto y números pero también incluye información en imágenes, video y audio. Todo software construido es virtualmente utilizado por el mundo industrializado en diferentes campos, ya sea de manera directa o indirecta; por ello es muy importante su adecuada construcción, ya que afecta de forma cercana cada aspecto de nuestras vidas y se ha vuelto dominante en nuestro comercio, cultura y actividades diarias.

La construcción del software es como la construcción de cualquier producto exitoso, mediante un proceso que guía a un resultado de alta calidad y que cumple con las necesidades de la gente que lo utilizará.

La ingeniería del software es una disciplina que comprende todos los aspectos de la producción de software; comprende las formas prácticas para desarrollar y entregar un software útil.

Desde el punto de vista del ingeniero de software, el producto del trabajo son los programas, documentos y datos que conforman el software; pero desde el punto de vista del usuario, es la información resultante que de alguna forma convierte el mundo del usuario en uno mejor; que en la mayoría de los casos ese mundo mejor es representado por una herramienta de trabajo que mejora la productividad del usuario o le permite realizar el trabajo de una manera más cómoda.[3]

2.1.1 Proceso de software

Cuando se construye un producto o sistema es importante seguir una serie de pasos previsibles; un mapa que nos ayude a crear un resultado de alta calidad y en tiempo. Para el caso de la construcción de sofware, ese mapa es llamado 'proceso de software'.

Un proceso de software es un conjunto de actividades y resultados asociados que al final producen un producto de software. Estas actividades son llevadas a cabo por los ingenieros de software (que pueden ser ingenieros en computación, licenciados en informática, etcétera), los cuales deben adaptar el proceso a sus necesidades y seguirlo para lograr el resultado deseado. Además de las personas directamente responsables, la gente que requiere el software (el usuario) juega un rol en el proceso; un rol cuya magnitud de importancia radica en si es un software hecho a la medida o si es un software más genérico, pero para ambos casos dicho rol es uno de los más importantes.

Cabe destacar la trascendencia de tener y seguir este proceso, ya que proveé de estabilidad, control y organización en una actividad que, si dejamos sin control, puede tornarse caótica. En un nivel detallado, el proceso que adoptemos depende del tipo de software que estemos construyendo, ya que el uso del mismo proceso puede no ser adecuado para distintos tipos de proyectos.

Existe un número de mecanismos de valoración del proceso de software que permiten determinar la madurez del proceso. Sin embargo; la calidad, puntualidad y viabilidad a largo plazo del producto que se construye, son los mejores indicadores de la eficacia del proceso que se está utilizando.

[3] (Somerville, 2005), con comentarios adicionales por parte de Erik León Mendoza

Son cuatro las actividades fundamentales que son comunes para todos los procesos de software:

1. *Especificación del software.* Los clientes y/o ingenieros definen el software a producir y las restricciones sobre su operación.
2. *Desarrollo del software.* El software se diseña y programa.
3. *Validación del software.* El software se valida para asegurar que es lo que el cliente requiere.
4. *Evolución del software.* El software se modifica para adaptarlo a los cambios requeridos por el cliente y/o el mercado.[4]

2.1.2 Atributos del buen software

Los productos de software cuentan con cierto número de atributos que reflejan su calidad. El conjunto específico de atributos que se espera depende obviamente de su aplicación, sin embargo, podemos intentar generalizar con los atributos mostrados en la figura 1, los cuales tienen las características esenciales de un buen software.

Mantenibilidad	El software debe escribirse de tal forma que pueda evolucionar para cumplir las necesidades de cambio de los clientes. Éste es un atributo crítico debido a que el cambio en el software es una consecuencia inevitable de un cambio en el entorno de negocios. Sin duda alguna este atributo es el más difícil de lograr.
Confiabilidad	La confiabilidad del software tiene un gran número de características, incluyendo la fiabilidad, protección y seguridad. El software confiable no debe causar daños físicos o económicos en el caso de una falla del sistema, y por supuesto, no debe causar la pérdida de información.
Eficiencia	El software no debe hacer que se malgasten los recursos del sistema, como la memoria y los ciclos de procesamiento. Por lo tanto, la eficiencia incluye tiempos de respuesta y de procesamiento, utilización de la memoria, etcétera.
Usabilidad	El software debe ser fácil de utilizar, sin esfuerzo adicional, por el usuario para quien está diseñado. Esto significa que debe tener una interfaz de usuario apropiada y una documentación adecuada.

Figura 1. Atributos de un buen software.

Acerca de la mantenibilidad

A medida que los sistemas se usan, surgen nuevos requerimientos, por lo cual es importante mantener la utilidad del sistema realizando los cambios necesarios para adaptarlo a las nuevas necesidades. Un software mantenible es aquel que puede ser adaptado para cumplir con los

[4] (Somerville, 2005), con comentarios adicionales por parte de Erik León Mendoza

nuevos requerimientos dentro de un costo razonable y con una baja probabilidad de introducir errores nuevos en el sistema cuando se realizan los cambios correspondientes.

Para lograr un alto grado de mantenibilidad, no es suficiente con utilizar apropiadamente los paradigmas de programación, patrones de diseño y buenas prácticas en cuanto a codificación; gran parte de lo necesario para lograr tal grado recae sobre el diseño del software, por lo que se debe poner especial cuidado en el modelado del proceso que eventualmente dará vida a una pieza de software.

Acerca de la confiabilidad

Refiere al grado de confianza que tendrá el usuario para operar el sistema, de tal forma que cumpla con lo que se espera de él y que no fallará bajo un uso normal.

La confiabilidad cuenta con cuatro dimensiones principales que se muestran en la figura 2.

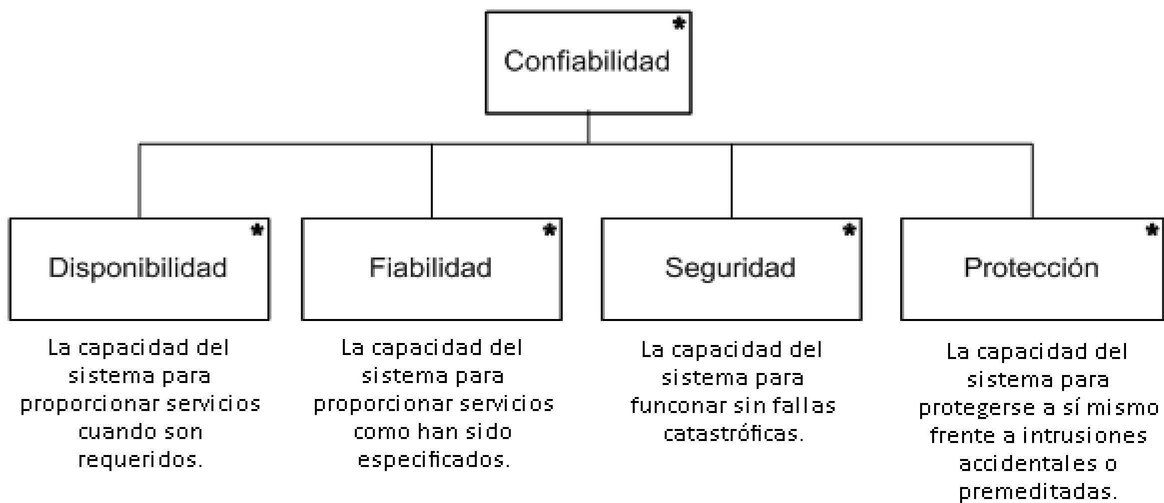

Figura 2. Dimensiones de la confiabilidad.

Disponibilidad. Es la probabilidad de que el sistema esté en disposición de funcionar, proporcionando los servicios que el usuario solicite. No sólo depende del sistema, sino también del tiempo necesario para reparar los defectos que hicieron que el sistema dejara de estar disponible.

Fiabilidad. Es la probabilidad de que el sistema funcione correctamente tal y como se ha especificado.

En la figura 3 podemos encontrar algunos términos comunes con respecto a la fiabilidad.

Falla del sistema (System failure)	Evento que tiene lugar en algún instante cuando el sistema no funciona como esperan sus usuarios.
Error del sistema (System error)	Estado erróneo del sistema que puede dar lugar a un comportamiento del mismo inesperado por sus usuarios.
Defecto del sistema (System fault)	Característica de un sistema que puede dar lugar a un error del sistema. Por ejemplo, un fallo en la ejecución al inicializar una variable puede hacer que dicha variable tenga un valor incorrecto cuando sea usada.
Error humano o equivocación (Mistake)	Comportamiento humano que tiene como consecuencia la introducción de defectos en el sistema.

Figura 3. Términos concernientes a la fiabilidad.

Considerando los términos anteriores, podemos mencionar 3 enfoques que nos ayudarán a mejorar la fiabilidad:

- *Evitación de defectos.* Minimizar la posibilidad de cometer equivocaciones o en su defecto detectarlas antes de que causen efectos negativos en el sistema.
- *Detección y eliminación de defectos.* Verificar y validar el sistema para encontrar defectos y corregirlos antes de la utilización del sistema.
- *Tolerancia a defectos.* Asegurar que los defectos en el sistema no conducen a errores y que éstos nos provocan fallas del funcionamiento.

<u>Seguridad.</u> Para garantizar que el funcionamiento del sistema se realizará sin fallos que conlleven consecuencias o efectos catastróficos debemos tener en cuenta las situaciones contempladas en la figura 4.

Accidente (o percance)	Evento o secuencia de eventos no planificados que provocan muerte o lesiones, daño a las propiedades o al entorno. Un ejemplo de un accidente es una máquina controlada por un ordenador que lesiona a su operador.
Contingencia	Una condición con el potencial de causar o contribuir a un accidente. Un ejemplo de contingencia es un fallo de funcionamiento de un sensor que detecta un obstáculo delante de una máquina.
Daño	Medida de la pérdida resultante de un percance. El dato puede variar desde varias personas muertas como resultado de un accidente, hasta pérdida parcial de información.
Gravedad de la contingencia	Evaluación del peor daño posible que podría resultar de una contingencia en particular. La gravedad de la contingencia puede variar desde catastrófica, en donde muchas personas mueren, a menor, en donde resultan solamente daños menores.
Probabilidad de la contingencia	La probabilidad de la ocurrencia de eventos que provocan una contingencia. Los valores de probabilidad tienden a ser arbitrarios, pero varían desde probable (por ejemplo, una probabilidad de ocurrencia del 1%) hasta improbable (situaciones no concebibles en las que probablemente ocurra la contingencia).
Riesgo	Es una medida de la probabilidad de que el sistema provoque un accidente. El riesgo se evalúa considerando la probabilidad de la contingencia, la gravedad de la contingencia y la probabilidad de que una contingencia cause un accidente.

Figura 4. Situaciones que deben evitarse para garantizar seguridad.

Protección. Refleja la capacidad que tiene el sistema de evitar posibles ataques externos que pueden tener origen accidental o intencional.

Los métodos para asegurar la disponibilidad, fiabilidad y seguridad se valen del hecho de que el sistema operacional del software es el mismo que se instaló originalmente. Si dicho sistema instalado se ha visto comprometido de alguna forma, entonces los argumentos para la fiabilidad y la seguridad originalmente establecidos dejan de ser ciertos. El software puede entonces corromperse y comportarse de forma impredecible.

Los ataques externos mencionados con anterioridad pueden causar daños como los siguientes:

- *Denegación de servicios.* El sistema puede entrar en un estado en que no estén disponibles sus servicios normales, lo cual afecta la disponibilidad.
- *Corrupción de programas o datos.* Los componentes del sistema pueden ser alterados de forma no autorizada, lo cual puede afectar el comportamiento y por lo tanto, la fiabilidad y seguridad del sistema.
- *Revelación de información confidencial.* La información gestionada por el sistema puede ser confidencial, y un ataque externo puede exponerla a personas no autorizadas.

Al igual que con la seguridad, para la protección se deben tener en cuenta las situaciones descritas en la figura 5.[5]

Exposición	Posible pérdida o daño en un sistema informático. Un ejemplo puede ser la pérdida o daño de los datos o la pérdida de tiempo y esfuerzo si es necesaria una recuperación del sistema después de una violación de protección.
Vulnerabilidad	Debilidad en un sistema informático que se puede aprovechar para provocar pérdidas o daños.
Ataque	Aprovechamiento de la vulnerabilidad de un sistema. Generalmente, se produce desde fuera del sistema y con una intención deliberada de causar algún daño.
Amenazas	Circunstancias que potencialmente pueden provocar pérdidas o daños. Se pueden entender como una vulnerabilidad del sistema que está expuesto a un ataque.
Control	Medida de protección que reduce la vulnerabilidad del sistema. La encriptación podría ser un ejemplo de un control que reduce una vulnerabilidad de un sistema de control de acceso deficiente.

Figura 5. Situaciones a considerar para la protección.

2.1.3 Modelos de proceso de software

Con anterioridad hice mención acerca de los procesos de software. Dichos procesos se pueden representar de manera abstracta mediante un modelo, elegido en base a la naturaleza del proyecto y del sistema, además de los métodos y herramientas a ser utilizadas.

Citando de una manera general, los siguientes son algunos modelos de proceso de software comunes y los cuales he utilizado a lo largo de mis años laborando en la industria:

1. *El modelo en cascada*, considera las actividades fundamentales del proceso de especificación, desarrollo, validación y evolución, y los representa como fases separadas del proceso, tales como la especificación de requerimientos, el diseño del software, la implementación, las pruebas, etcétera.
2. *Desarrollo evolutivo*. Este enfoque entrelaza las actividades de especificación, desarrollo y validación. Un sistema inicial se desarrolla rápidamente a partir de especificaciones abstractas. Éste se refina basándose en las peticiones del cliente para producir un sistema que satisfaga sus necesidades.
3. *Ingeniería del software basada en componentes*. Este enfoque se basa en la existencia de un número significativo de componentes reutilizables. El proceso de desarrollo del sistema se enfoca en integrar estos componentes en el sistema más que en desarrollarlos desde cero.

[5] (Somerville, 2005)

A continuación hago mención de algunos detalles importantes acerca de los modelos de proceso que considero son más utilizados en los sistemas hechos a la medida actualmente. Los sistemas que se han creado en la SCJN siguen alguno de estos modelos o incluso una combinación de ellos.

2.1.3.1 Desarrollo evolutivo

El desarrollo evolutivo se basa en la idea de desarrollar una implementación inicial, exponiéndola a los comentarios del usuario y refinándola a través de las diferentes versiones hasta que se desarrolla un sistema adecuado. Las actividades de especificación, desarrollo y validación se entrelazan en vez de separarse, con una rápida retroalimentación entre éstas, como se ilustra en figura 6.

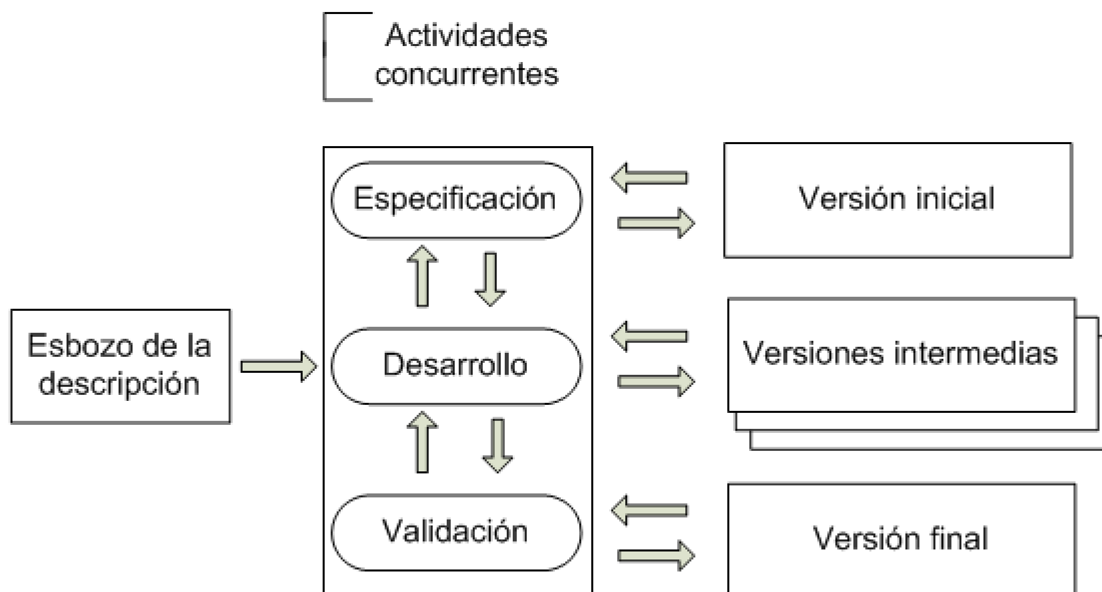

Figura 6. Etapas del desarrollo evolutivo.

Existen dos tipos de desarrollo evolutivo:

1. *Desarrollo exploratorio*, donde el objetivo del proceso es trabajar con el cliente para explorar sus requerimientos y entregar un sistema final. El desarrollo empieza con las partes del sistema que se comprenden mejor. El sistema evoluciona agregando nuevos atributos propuestos por el cliente.
2. *Prototipos desechables*, donde el objetivo del proceso de desarrollo evolutivo es comprender los requerimientos del cliente y entonces desarrollar una definición mejorada de los requerimientos para el sistema. El prototipo se centra en experimentar con los requerimientos del cliente que no se comprenden del todo.

En la producción de sistemas, un enfoque evolutivo para el desarrollo de software suele ser más efectivo que el enfoque en cascada, ya que satisface las necesidades inmediatas de los clientes. La ventaja de un proceso del software que se basa en un enfoque evolutivo es que la especificación se puede desarrollar de forma creciente. Tan pronto como los usuarios desarrollen un mejor entendimiento de su problema, éste se puede reflejar en el sistema de software. Sin embargo, desde una perspectiva de ingeniería y de gestión, el enfoque evolutivo tiene dos problemas:

1. *El proceso no es visible*. Los administradores tienen que hacer entregas regulares para medir el progreso. Si los sistemas se desarrollan rápidamente, no es rentable producir documentos que reflejen cada versión del sistema.
2. *A menudo los sistemas tienen una estructura deficiente*. Los cambios continuos tienden a corromper la estructura del software. Incorporar cambios en él se convierte cada vez más en una tarea difícil y costosa.

Para sistemas pequeños y de tamaño medio (digamos, hasta 500,000 líneas de código), el enfoque evolutivo de desarrollo es el mejor. Los problemas del desarrollo evolutivo se hacen particularmente agudos para sistemas grandes y complejos con un periodo de vida largo, donde diferentes equipos desarrollan distintas partes del sistema. Es difícil establecer una arquitectura del sistema estable usando este enfoque, el cual hace difícil integrar las contribuciones de los equipos.

Para sistemas grandes, se recomienda un proceso mixto que incorpore las mejores características del modelo en cascada y del desarrollo evolutivo. Esto puede implicar desarrollar un prototipo desechable utilizando un enfoque evolutivo para resolver incertidumbres en la especificación del sistema. Puede entonces reimplementarse utilizando un enfoque más estructurado. Las partes del sistema bien comprendidas se pueden especificar y desarrollar utilizando un proceso basado en el modelo en cascada. Las otras partes del sistema, como la interfaz de usuario, que son difíciles de especificar por adelantado, se deben desarrollar siempre utilizando un enfoque de programación exploratoria.

2.1.3.2 Ingeniería de software basada en componentes

La ingeniería del software basada en componentes es el proceso de definir, implementar e integrar componentes independientes débilmente acoplados que en su totalidad formen el sistema. Este modelo se ha convertido en una importante aproximación de desarrollo del software debido a que los sistemas de software son cada vez más grandes y más complejos, y los clientes demandan software más confiable que sea desarrollado más rápidamente. La única forma en la que podemos tratar con la complejidad y entregar software más rápidamente es reutilizar componentes de software en vez de reimplementarlos.[6]

[6] (Somerville, 2005)

2.1.4 Iteración de procesos

Los cambios son inevitables en todos los proyectos de software grandes. Los requerimientos del sistema cambian cuando el negocio que procura el sistema responde a las presiones externas. Las prioridades de gestión cambian; cuando se dispone de nuevas tecnologías, cambian los diseños y la implementación. Esto significa que el proceso del software no es un proceso único; más bien, las actividades del proceso se repiten regularmente conforme el sistema se rehace en respuesta a peticiones de cambios.

Existen dos modelos de procesos que han sido diseñados explícitamente para apoyar la iteración de procesos:

1. *Entrega incremental*. La especificación, el diseño y la implementación del software se dividen en una serie de incrementos, los cuales se desarrollan por turnos.
2. *Desarrollo en espiral*. El desarrollo del sistema gira en espiral hacia fuera, empezando con un esbozo inicial y terminando con el desarrollo final del mismo.[7]

Basándome en mi experiencia en el desarrollo de software a la medida, recomendaría el uso de un modelo de procesos de entrega incremental. Algunas de las ventajas de hacerlo de dicha forma son las siguientes:

- Los clientes no tienen que esperar a que el sistema esté completo para sacar provecho de su utilización.
- Los clientes pueden utilizar las entregas iníciales como prototipos para obtener experiencia de la funcionalidad del sistema. Por ejemplo, los prototipos desarrollados pueden ser utilizados para la capacitación del cliente en la utilización del software; de esta forma se puede tener retroalimentación para poder mejorar la funcionalidad y contar con una mejor experiencia de usuario.
- Existe un bajo riesgo de un fallo total del proyecto. Aunque es posible que se encuentren errores en incrementos al sistema, es normal que éste se entregue de forma satisfactoria.
- Debido a que los servicios de más alta prioridad son los que se entregan primero es inevitable que la funcionalidad más importante del sistema sea la que tenga más pruebas. Esto se traduce en menos probabilidad de que el cliente encuentre fallos de funcionamiento del sistema.

2.1.5 Actividades del proceso

Existen cuatro actividades básicas de un proceso de software: especificación, desarrollo, validación y evolución. El cómo se llevan a cabo dichas actividades depende del tipo de software, de las personas responsables de darle vida al software y de la estructura organizacional implicadas.

[7] (Somerville, 2005)

2.1.5.1 *Especificación del software*

Se refiere al proceso de comprensión y definición de que servicios se requieren del sistema, así como las restricciones del mismo. Esta etapa es crítica en todo el proceso debido a que los errores cometidos inevitablemente ocasionarán errores al momento del diseño e implementación del sistema.

En la figura 7 se muestra en general cómo es la etapa de especificación del software.

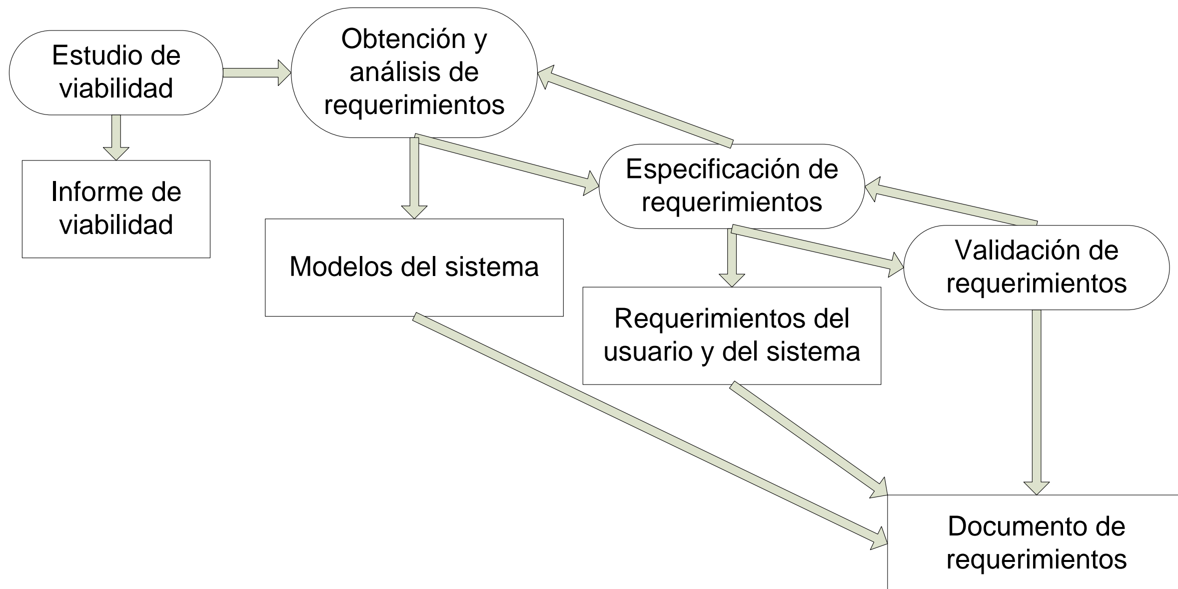

Figura 7. Fases de especificación del software.

En la figura 7 se pueden reconocer cuatro fases principales:

1. *Estudio de viabilidad.* Una estimación de si las necesidades del usuario se pueden satisfacer con las tecnologías actuales (software y hardware). En el estudio se analizará si el sistema es rentable desde un punto de vista de negocios y si es viable su desarrollo dentro de las restricciones que impone el presupuesto previsto.
2. *Obtención y análisis de requerimientos.* Mediante la observación de sistemas existentes, discusión con los usuarios potenciales, análisis de tareas, etc.; se pueden obtener los requerimientos del sistema. Esto puede implicar el desarrollo de modelos y prototipos del sistema que ayudarán a comprender el sistema a especificar.
3. *Especificación de requerimientos.* Se debe traducir la recopilación de información, resultado del análisis, en un documento que define un conjunto de requerimientos.

4. *Validación de requerimientos.* Comprobar la veracidad, consistencia y completitud de los requerimientos. Es inevitable que en esta fase se encuentren errores en el documento de requerimientos, los cuales se procederá a corregir.

2.1.5.2 Diseño e implementación del software

Durante esta etapa se convertirá una especificación de software en un sistema ejecutable. Siempre implica procesos de diseño y programación de software pero, si se utiliza un enfoque de desarrollo evolutivo, también implica un refinamiento de la especificación del software.

Un diseño de software es una descripción de la estructura del software que se implementará, los datos procesados por el sistema, las interfaces entre los componentes del sistema e incluso, los algoritmos utilizados.

El resultado final de toda la etapa de diseño son especificaciones precisas de los algoritmos y estructuras de datos que deben implementarse.

La figura 8 ilustra de manera general el proceso de diseño.

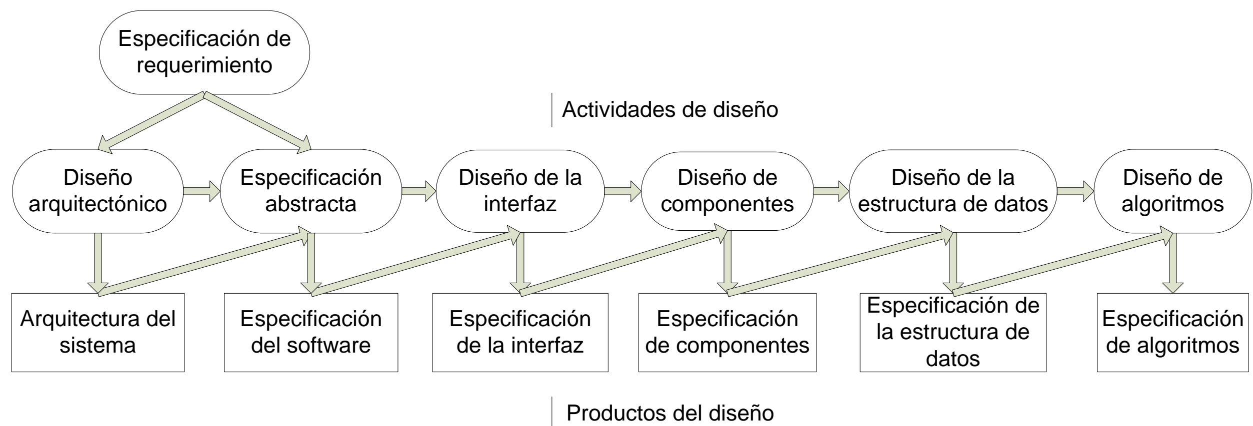

Figura 8. Proceso de diseño de software.

2.1.5.3 Validación del software

La validación del software significa que debemos mostrar que el sistema se ajusta a la especificación con que fue diseñado y que cumple las expectativas del usuario.

Para evitar probar el sistema como una simple unidad, se divide la prueba en etapas que nos permiten detectar defectos en el sistema desde etapas iniciales del proceso.

Las etapas de prueba son las siguientes:

- *Prueba de componentes o unidades.* Cada componente se prueba de forma independiente a otros componentes.
- *Prueba del sistema.* Los componentes se integran para formar el sistema. Se debe probar que la interacción de los distintos componentes se de de forma correcta.
- *Prueba de aceptación.* La prueba final antes de poner el sistema en funcionamiento. Se hacen las pruebas con los datos que el cliente proporciona, es decir, con datos reales.

La figura 9 muestra el proceso de validación del software, que como puede verse es un proceso de retroalimentación entre las distintas etapas.

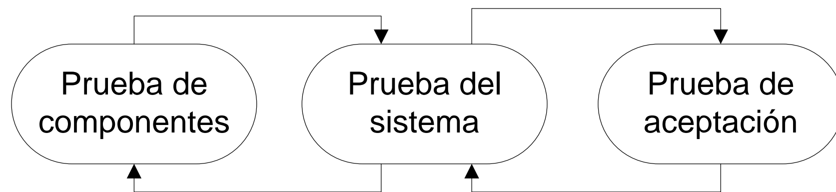

Figura 9. Proceso de validación de software.

2.1.5.4 Evolución del software

Una de las ventajas de los sistemas de software es que son muy flexibles, lo cual permite hacer cambios en cualquier momento durante o después del desarrollo. Aún cuando sean cambios significativos, dichos cambios son muchos más económicos que cuando se trata, por ejemplo, de cambios en el hardware.

Más que dos procesos separados, es más realista considerar a la ingeniería de software como un proceso evolutivo en el cual el software cambia continuamente durante su periodo de vida como respuesta a los requerimientos cambiantes y necesidades del usuario.

En la figura 10 se muestra el proceso evolutivo mencionado.

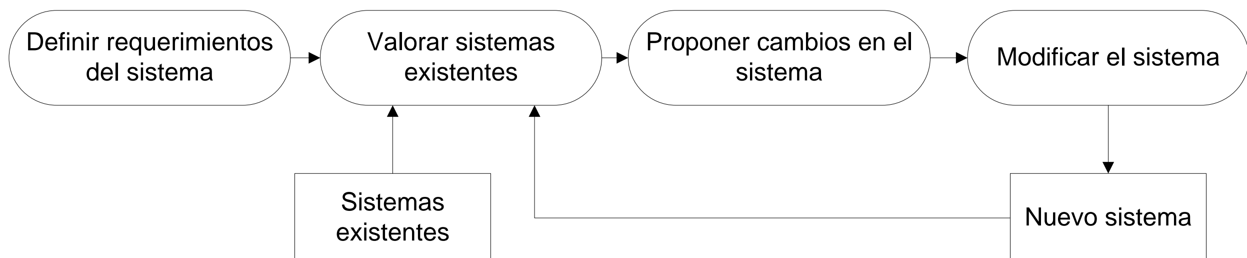

Figura 10. Proceso evolutivo del software.

Hasta este momento hemos abordado las 4 actividades que mencioné se contemplan en el proceso de software. Contamos con un bosquejo que nos da una idea de lo que se tiene que contemplar para construir el software, sin embargo no es tarea fácil, ya que cada una de estas cuatro actividades representa una variedad de retos que generalmente no tienen una solución general y se tendrá que usar el criterio para tomar las mejores decisiones e incluso, algunas decisiones en las que no se tiene otra opción.[8]

2.2 Recolección y análisis de requerimientos

Un proyecto de software que está hecho a la medida (es decir, para cumplir una función y un mercado muy específicos) tendrá imprescindiblemente un cliente que nos tiene que especificar cuál es la misión del producto y de que formas se alcanzará dicha misión. De entrada suena a tarea fácil, pero resulta más complicado de lo que se puede esperar debido a que las ideas generalmente se dejan en un término abstracto y no son concurrentes a una misma y exacta visión entre el cliente y los creadores del software.

En general los textos acerca de lo relacionado con requerimientos nos dan las recomendaciones de lo que se debería hacer, por lo cual me gustaría resaltar algunas recomendaciones de lo que NO se debe hacer o no es recomendable hacer para evitar un posible fracaso:

- Asumir un comportamiento, una acción o un resultado de algún proceso que se está modelando en una pieza de software sin tener una confirmación del cliente de que es la forma correcta de hacerlo, puede provocar que se tenga que rehacer algunas partes de la implementación o peor aún, del diseño de la aplicación.
- Cuando se está construyendo un sistema basado en alguno existente, el hecho de querer deducir el comportamiento y funcionalidad del sistema existente sin recibir retroalimentación del cliente puede llegar a convertirse en el peor error cometido. Incluso de tal forma se pueden estar heredando errores existentes en dicha versión.
- Al utilizar un esquema de prototipos desechables, el forzar al ciclo de vida a ser demasiado corto provoca la toma de decisiones apresuradas tanto en diseño como en la implementación, que pueden resultar costosas a largo plazo.
- La separación de los requerimientos del usuario y de los del sistema debe ser clara. Sin embargo, ambos requerimientos deben de ayudar o complementarse para cumplir el objetivo deseado.
- No existe razón para asumir que el usuario necesitará una funcionalidad que no ha sido solicitada explícitamente. Así mismo, las sugerencias para mejorar la usabilidad del sistema deben hacerse con precaución.

[8] (Somerville, 2005), con comentarios adicionales por parte de Erik León Mendoza

Al querer plasmar la recolección y análisis de requerimientos en un documento, se suelen sugerir algunas herramientas de modelado como, por ejemplo, UML. Sin embargo, al plasmar los requerimientos bajo diagramas de este tipo se suelen perder algunos detalles significantes para el cumplimiento pleno del requerimiento. Mi recomendación es trabajar en un documento con explicación más detallada, donde se plasme de una manera más amplia y clara cada uno de los requerimientos. No hay que olvidar que esta documentación de requerimientos es la base para la gente encargada de diseñar e implementar la solución, y en la mayoría de los casos dichas personas no han tenido contacto con el cliente; así que el documento debe dejar claro los requerimientos del usuario de tal forma que no exista pérdida de información.

2.3 Modelado y diseño de la arquitectura

La arquitectura de una aplicación de software es el proceso de definir una solución estructurada que cumpla con todos los requerimientos técnicos y operacionales, al mismo tiempo que se optimizan atributos comunes concernientes a la calidad como lo son el rendimiento, la seguridad y la manejabilidad. Dicha arquitectura envuelve una serie de decisiones basadas en un amplio rango de factores, y cada una de esas decisiones tiene un impacto en el éxito de la aplicación.

2.3.1 Importancia

El software debe ser construido bajo fundamentos sólidos. El ignorar escenarios claves y las consecuencias a largo plazo de las decisiones tomadas pueden poner en riesgo la aplicación. A pesar de que las herramientas actuales permiten simplificar la construcción de nuevo software, éstas no reemplazan la necesidad de diseñar una aplicación cuidadosamente con base a escenarios específicos y requerimientos. Cuando hablamos de riesgo en la aplicación, hablamos de que dicha aplicación puede ser inestable o no es apta para soportar requerimientos de negocio existentes (o futuros requerimientos), entre otros riesgos. Para evitar en lo más posible los riesgos, el diseño debe tener consideración del usuario, el sistema (en cuestión de infraestructura), y los objetivos del negocio.

Podemos mencionar como principales objetivos de una arquitectura los siguientes:

- Exponer la estructura del sistema al mismo tiempo que se ocultan los detalles de la implementación.
- Contempla todos los casos de uso y los escenarios.
- Maneja tanto los requerimientos funcionales como los de calidad.[9]

[9] (Microsoft, 2009)

2.3.2 Principios del diseño de la arquitectura

Debemos asumir que el diseño creado para la arquitectura de una aplicación evolucionará con el tiempo y que no podemos saber todo lo necesario para tener un diseño que se adapte a futuras necesidades. Mientras se aprenda más de las reglas de negocio y de las pruebas contra requerimientos reales, el diseño generalmente tendrá la necesidad de evolucionar durante la etapa de implementación. La arquitectura debe ser creada con esta evolución en mente para que tenga la capacidad de adaptarse a los requerimientos que no se conocen al inicio del proceso de diseño.

Principios clave

Tener en mente algunos principios clave ayudará a crear una arquitectura que minimice los costos de mantenimiento y que promueva la usabilidad y extensibilidad. Algunos de dichos principios son:

- *Separación de los asuntos de interés*. Dividir la aplicación en distintas características cuya funcionalidad se traslape lo menos posible. El factor importante en este principio es minimizar los puntos de interacción para lograr una alta cohesión y un bajo acoplamiento. Sin embargo, si esta separación no se hace de forma adecuada, se puede tener como resultado características demasiado complejas.
- *Principio de responsabilidad simple*. Cada componente o módulo debe ser responsable de sólo una característica específica o funcionalidad.
- *Principio de mínimo conocimiento.* Un componente u objeto no debe conocer los detalles internos de otros componentes u objetos.
- *No repetirse asimismo.* Funcionalidad específica se debe implementar en sólo un componente, dicha funcionalidad no debe duplicarse en cualquier otro componente.
- *Minimizar el diseño anticipado.* Solo diseñar lo que es necesario.[10]

2.3.3 Principios del diseño orientado a objetos

Previo a la existencia de la orientación a objetos, cualquier programa era el resultado de la interacción de módulos y rutinas. La programación era procedimental, lo cual significa que existía un flujo principal de código el cual determinaba los distintos pasos que se debía completar.

La orientación a objetos permite visualizar un programa como el resultado de la interacción de objetos, cada uno de éstos con datos y comportamiento propios.

[10] (Microsoft, 2009)

Algunos principios del diseño orientado a objetos se mencionan a continuación.

Principio de responsabilidad simple.

Se refiere a que una clase debe tener una, y sólo una, razón para cambiar. Esa razón corresponde a la definición de responsabilidad. Si podemos pensar en más de una razón para cambiar una clase quiere decir que esa clase tiene más de una responsabilidad. La figura 11 muestra un ejemplo que no cumple con este principio.

El Principio de responsabilidad simple es uno de los principios más sencillos pero uno de los más difíciles de obtener de forma correcta. Reunir responsabilidad es algo que hacemos con naturalidad. Encontrar y separar esas responsabilidades es mucho de lo que el diseño de software significa.

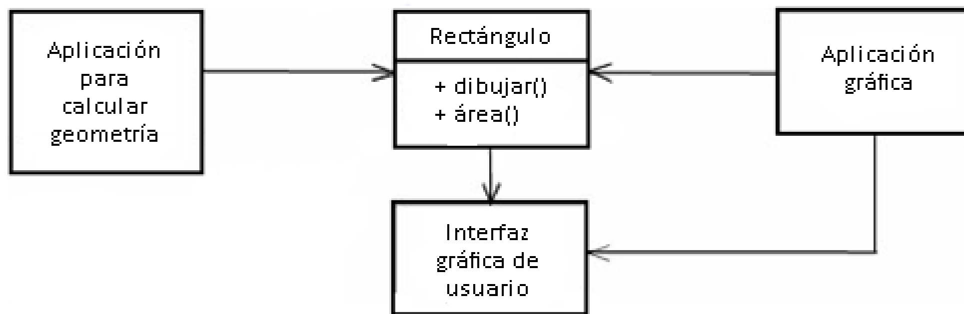

Figura 11. Se puede distinguir que el objeto rectángulo cuenta con dos responsabilidades.

Principio abierto/cerrado

Este principio nos dice que las entidades de software (clases, módulos, funciones, etc.) deben estar abiertas a extensión, pero cerradas a la modificación. Aunque se puede modificar una clase para reparar un defecto, si se desea agregar un nuevo comportamiento entonces la clase debe extenderse. Lo anterior ayuda a mantener el código apto para su administración y pruebas.

La abstracción es la herramienta que necesitaremos para hacer posible que el comportamiento de los objetos sea modificado sin cambiar su código fuente. Debemos hacer uso de clases base y derivación de clases para sacar ventaja de la abstracción.

En muchas formas, el principio abierto/cerrado es el 'corazón' del diseño orientado a objetos. Apegarse a este principio es lo que conlleva a los grandes beneficios de las tecnologías orientadas a objetos: flexibilidad, reusabilidad y mantenibilidad. Sin embargo, el simple uso de un lenguaje orientado a objetos no significa hacer uso del principio; se necesita cierta dedicación para encontrar en que partes se debe aplicar la abstracción, que generalmente son las partes que exhiben mayor cambio.

Principio de sustitución Liskov

En términos de programación computacional, si S es un subtipo de T, entonces los objetos de tipo T pueden ser reemplazados con objetos de tipo S sin alterar propiedad alguna; entonces hablamos de que se cumple el Principio de sustitución de Liskov.

La importancia de este principio se vuelve obvia cuando se consideran las consecuencias de violarlo. Supongamos que tenemos una función *f* que toma como argumento una referencia a una clase base *B*. Supongamos también que cuando algún derivado *D* de *B* causa un mal comportamiento cuando se encuentra dentro de la función. Dadas estas circunstancias entonces podemos decir que *D* viola el principio de sustitución de Liskov. Claramente *D* es frágil en la presencia de *f.*

Principio de segregación de interfaz

La intención del principio de segregación de interfaz es crear un software que sea más fácil de darle mantenimiento. Este principio alienta el bajo acoplamiento para que de esa forma se tenga un sistema con facilidad de cambiar. Las interfaces que son muy largas deben ser divididas en otras más pequeñas y específicas para que las clases cliente solo necesiten saber de métodos que usan, ninguna clase cliente debe ser forzada a depender de métodos que no utiliza.

Principio de inversión de dependencias

Podemos resumir en dos puntos el principio de inversión de dependencias:

- Los módulos de alto nivel no deben depender de los de bajo nivel. Ambos deben depender de abstracciones.
- Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones.

El uso de la palabra ‘inversión’ en el nombre del principio se debe a que la mayoría de los métodos tradicionales de desarrollo de software tienden a crear estructuras en las cuales los módulos de alto nivel dependen de otros de bajo nivel y en los cuales las políticas dependen de los detalles. Ciertamente uno de los objetivos de esos métodos es definir la jerarquía que describe como los módulos de alto nivel hacen las llamadas a los módulos de bajo nivel. En un programa con buen diseño orientado a objetos la estructura de dependencias está “invertida” con respecto a la estructura normalmente usada en los métodos procedimentales tradicionales.[11]

[11] (Martin & Martin, 2006)

2.3.4 Arquitecturas cliente-servidor

Las tecnologías de hardware, software, bases de datos y redes han contribuido a arquitecturas de cómputo distribuidas y cooperativas. Un sistema raíz (mainframe, o sistema central) sirve como repositorio para datos corporativos o empresariales. El sistema raíz está conectado a servidores (generalmente estaciones de trabajo o computadoras personales poderosas) que juegan un rol dual. Los servidores actualizan y hacen peticiones sobre los datos que se encuentran en el sistema raíz, además de comunicar a las computadoras personales mediante una red local (LAN).

Tipos de implementaciones

- *Servidores de archivos.* El cliente realiza una petición de registros de un archivo. El servidor transmite esos registros al cliente a través de la red.
- *Servidores de bases de datos.* El cliente manda una petición de datos al servidor. Estas peticiones son transmitidas como mensajes a través de la red. El servidor procesa las peticiones y encuentra la información requerida, y regresa la información sólo al cliente.
- *Servidores transaccionales.* El cliente hace una petición que invoca a procedimientos remotos en el servidor. Los procedimientos remotos son un grupo de declaraciones SQL. Una transacción ocurre cuando una petición resulta en la ejecución de un procedimiento remoto con el resultado transmitido de vuelta al cliente.
- *Servidores de grupo colaborativo.* Cuando los servidores proveen un grupo de aplicaciones que permiten la comunicación entre clientes (y los usuarios que los usan) mediante texto, imágenes, boletines, video y otras representaciones; se dice que existe una arquitectura de grupo colaborativo.[12]

La figura 12 muestra un ejemplo de arquitectura cliente-servidor.

[12] (Pressman, 2009)

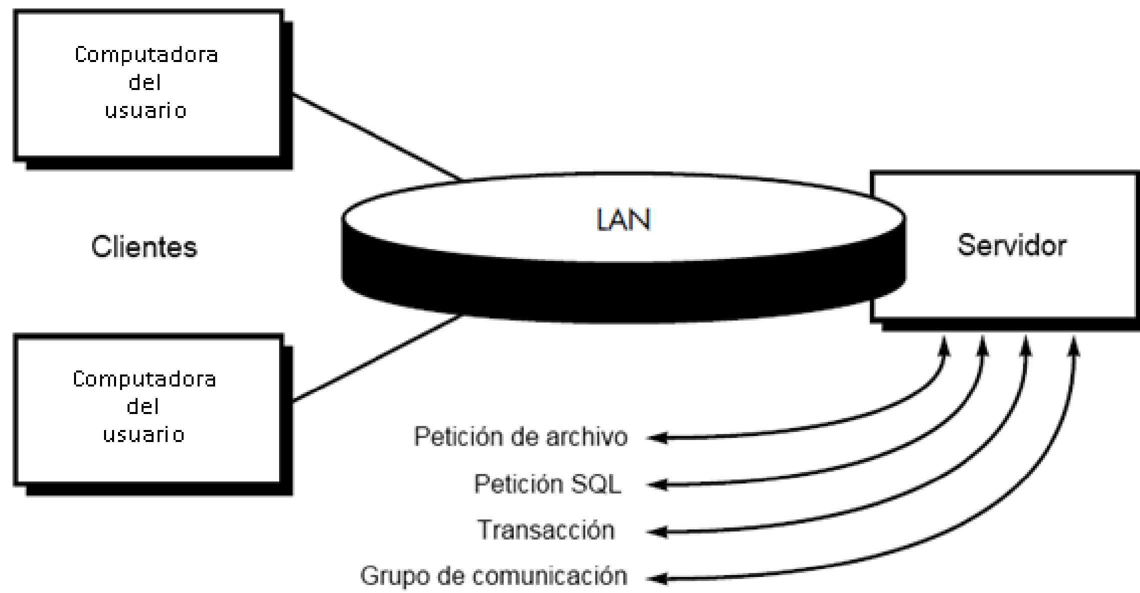

Figura 12. Ejemplo de una arquitectura cliente-servidor.

2.3.5 Componentes de software para arquitecturas cliente-servidor

En lugar de ver al software como una aplicación monolítica a ser implementada en una máquina, el software que es apropiado para una arquitectura C/S tiene distintos subsistemas que pueden estar alojados en el cliente, el servidor, o distribuidos entre ambas máquinas:

- *Subsistema de Interacción del usuario / Presentación*. Este subsistema implementa todas las funciones que típicamente se asocian con una interfaz gráfica de usuario.
- *Subsistema de aplicación.* Implementa los requerimientos definidos por la aplicación dentro del contexto del dominio en el cual la aplicación opera. Por ejemplo, una aplicación de negocios puede producir una variedad de reportes impresos basados en una entrada de datos numérica, cálculos, información de una base de datos, y otras consideraciones. En este caso, la aplicación de software debe dividir que algunos componentes residan en el cliente y otros del lado del servidor.
- *Subsistema de administración de base de datos.* Realiza manipulación de datos y administración de estos requerida por una aplicación. Estos procesos pueden ser tan simples como la transferencia de un registro o tan completos como el procesamiento de sofisticadas transacciones.[13]

[13] (Pressman, 2009)

2.3.6 Arquitectura basada en componentes

Una arquitectura basada en componentes se concentra en la descomposición del diseño en componentes funcionales o lógicos individuales. Dichos componentes cuentan con interfaces de comunicación bien definidas, conteniendo métodos, eventos y propiedades.

Los componentes deberían tener al menos las siguientes características:

- *Reutilizables.* Diseñados para ser utilizados en diferentes escenarios y diferentes aplicaciones; sin embargo, algunos componentes pueden ser diseñados para una tarea en específico.
- *Reemplazables.* Fácilmente substituibles con componentes similares.
- *No son específicos al contexto.* Diseñados para operar en diferentes ambientes y contextos. La información específica, como el estado de los datos, debe ser pasado al componente en lugar de ser incluido dentro del componente o accedido por éste.
- *Extensible.* Un componente puede ser extendido para proveer un nuevo comportamiento.
- *Encapsulado.* Se exponen interfaces que permiten al objeto que haga uso del componente usar su funcionalidad, y no revelar los detalles de los procesos internos o cualquier variable o estado interno.
- *Independiente.* Deben existir dependencias mínimas de otros componentes. De esta forma los componentes pueden ser modificados sin afectar otros componentes.

Tipos comunes de componentes utilizados en aplicaciones pueden ser cuadrículas de información y botones (frecuentemente nos referimos a ellos como controles), auxiliares y utilidades que proveen un grupo específico de funciones.

Los siguientes son los beneficios principales de utilizar una arquitectura basada en componentes:

- *Facilidad de despliegue.* Al haber nuevas versiones disponibles, se pueden remplazar las versiones existentes sin impacto en los otros componentes o el sistema entero.
- *Reducción de costos.* El uso de componentes de otras compañías permite extender esfuerzos en el desarrollo y en el mantenimiento.
- *Facilidad de desarrollo.* La implementación de componentes con interfaces bien conocidas que proveen una funcionalidad definida permiten un desarrollo sin impactar otras partes del sistema.
- *Reutilizable.* El uso de componentes reutilizables significa que pueden ser utilizados incluso en otros sistemas o en distintas aplicaciones.
- *Mitigación de la complejidad técnica.* A través de un contenedor de componentes y sus servicios.[14]

[14] (Pressman, 2009)

2.3.7 Arquitectura en capas

Una arquitectura en capas se basa en la agrupación de funcionalidad relacionada dentro de una aplicación en distintas capas que son colocadas verticalmente una encima de otra. La funcionalidad dentro de cada capa está relacionada por un rol o responsabilidad común. La comunicación entre capas es explícita y sin excesivo acoplamiento. Estructurando la aplicación en capas adecuadamente ayuda a soportar una fuerte separación de cuestiones que en consecuencia, permite el soporte de flexibilidad y mantenimiento.

Este estilo de arquitectura ha sido descrito como una pirámide invertida de reutilización donde cada capa agrega las responsabilidades y abstracciones de la capa directamente debajo de ésta.

Las capas de una aplicación pueden residir en la misma ubicación física o pueden ser distribuidas en ubicaciones físicas diferentes. Los componentes en cada capa se comunican con los componentes de otras capas mediante interfaces bien definidas.

Los principios comunes para diseños que usan el estilo de arquitectura en capas son:

- *Abstracción.* Se abstrae la vista del sistema como un todo mientras se provee suficiente detalle para entender los roles y responsabilidades de capas individuales y las relaciones entre ellas.
- *Encapsulación.* No se necesitan suposiciones acerca del tipo de datos, métodos y propiedades, o implementación durante el diseño, ya que esas características no son expuestas en las fronteras de las capas.
- *Clara definición funcional.* La separación entre la funcionalidad en cada capa es clara. Las capas superiores como la capa de presentación mandan comandos a capas inferiores, como la capa de negocios y la de datos; y puede reaccionar a eventos en esas capas, permitiendo el flujo de datos en ambas direcciones entre las capas.
- *Alta cohesión.* Los límites de responsabilidad para cada capa están bien definidos; se garantiza que cada capa contiene funcionalidad directamente relacionada a las tareas de dicha capa, lo cual ayuda a maximizar la cohesión dentro de la capa.
- *Reutilizable.* Las capas inferiores no tienen dependencias de las capas superiores, lo cual permite potencialmente ser utilizadas en otros escenarios.
- *Amplio acoplamiento.* Comunicación entre las capas está basada en la abstracción y eventos.

Los principales beneficios del estilo de arquitectura en capas son:

- *Abstracción.* Las capas permiten realizar cambios a un nivel abstracto. Se puede incrementar o aumentar el nivel de abstracción que se utiliza en cada capa de la estructura jerárquica.
- *Aislamiento.* Permite aislar las actualizaciones de tecnología de capa para reducir los riesgos y minimizar el impacto en todo el sistema.
- *Administración.* La separación de cuestiones ayuda a identificar dependencias además de organizar el código en secciones a las que se les puede dar mejor mantenimiento.

- *Rendimiento.* Distribuir las capas en múltiples capas físicas puede proveer escalabilidad, tolerancia a fallos y rendimiento.
- *Reusabilidad.* Los roles promueven la usabilidad. Algunos componentes de las capas pueden ser compatibles para su uso en otros sistemas.
- *Pruebas.* Se incrementa la capacidad de pruebas debido a que las interfaces de las capas están bien definidas.[15]

2.4 Desarrollo seguro de aplicaciones

A pesar de que los riesgos de seguridad en aplicaciones de software han evolucionado drásticamente con la inclusión de nuevas tecnologías en nuestra vida cotidiana, las estrategias de seguridad para combatir dichos problemas se han quedado atrás. En otras palabras, muchas organizaciones utilizan estrategias y técnicas que son viejas o incluso obsoletas para combatir los riesgos y vulnerabilidades actuales. Como ejemplo, puedo mencionar que intrusos sofisticados son capaces de traspasar las defensas en el perímetro (hablando de la infraestructura en donde se aloja nuestra aplicación) y perpetrar ataques que en la mayoría de ocasiones son difíciles de detectar.

La información es poder. En la actualidad, la información digital tiene gran relevancia debido a que representa datos sensibles que no deben ser del conocimiento de terceros; por lo cual las medidas de seguridad que utilicemos tanto en nuestra infraestructura como en el propio diseño e implementación deben ser no sólo parte indispensable de la construcción de un software, sino que incluso es la base de la que debemos partir para evitar la mayor cantidad de riesgos.

Si queremos tomar acciones con respecto a la seguridad, debemos empezar por considerar un modelo de información segura que tenga como base el conocimiento que se tiene sobre amenazas, recursos, así como de los motivos y objetivos potenciales de los atacantes. Por este motivo, se requiere que la organización que tendrá posesión y que hará uso del software identifique sus propiedades más valiosas y que dé prioridad a la protección de éstas. Los incidentes de seguridad deben ser vistos como riesgos del negocio que no siempre pueden ser prevenidos, pero sin duda pueden ser manejados dentro de un nivel aceptable.

[15] (Microsoft, 2009)

2.4.1 Medidas fundamentales para una seguridad efectiva

Una seguridad efectiva requiere de numerosas medidas técnicas, políticas y de aquellas que involucran a las personas que son parte de la organización. A continuación mencionaré diez medidas claves que he utilizado y que nos pueden ayudar a estar más cerca de lograr un software seguro:

- Una política de seguridad plasmada en un escrito.
- Respaldos y planes de continuidad en el negocio.
- Colección y retención mínima de información personal, con restricciones de acceso físico a los registros que contienen datos personales.
- Medidas tecnológicas fuertes para la prevención, detección y encriptación.
- Un inventario preciso de donde están los datos personales de empleados y clientes siendo colectados, transmitidos y almacenados; incluyendo a terceros que tengan acceso a dicha información.
- Valoración de los riesgos internos y externos acerca de la privacidad, seguridad, confidencialidad e integridad de los registros electrónicos o físicos.
- Monitoreo del programa de privacidad de datos.
- Revisión de antecedentes personales.
- Programa de entrenamiento para concientizar a los empleados acerca de la seguridad.
- Requerir a los empleados y terceros cumplir con las políticas de seguridad.

2.4.2 Un nuevo enfoque de cara a la seguridad

Las medidas tradicionales de seguridad no nos llevarán muy lejos en la actualidad al tener un entorno con demasiados riesgos por hacer frente. Un modelo de seguridad más actual debe alinearse a otras recomendaciones:

La seguridad es imperativa. Se debe entender la exposición y el impacto potencial en el negocio asociado con la operación en un ecosistema global. Una estrategia integral de seguridad debe ser el pivote en el modelado del negocio; la seguridad ya no es más un simple reto en el mundo de las tecnologías de la información.

Las amenazas de seguridad son riesgos para el negocio. Nos debemos anticipar a esas amenazas. Hay que conocer las vulnerabilidades y ser capaces de identificar y manejar los riesgos asociados.

Proteger la información que es realmente importante. Se deben priorizar los recursos para proteger la información valiosa. Claro que antes se debe identificar cuál es esa información valiosa.

Conciencia acerca de la acción. Crear una cultura de seguridad que empieza desde los altos ejecutivos y se extiende hasta todos los empleados.

2.5 Implementación del software

Sin lugar a dudas la implementación de una pieza de software es la que toma más tiempo en el proceso de construcción. Si bien el tener un buen análisis de requerimientos y un correcto diseño de la arquitectura son claves para obtener un sistema exitoso, se necesita de tomar decisiones correctas en la implementación para evitar riesgos de seguridad y un mal funcionamiento de la aplicación.

La existencia de infinidad de tecnologías entre las que se puede elegir para llevar a cabo la implementación, puede causar duda de cuál es la ideal. Sin embargo, es muy probable que la decisión se base en diversos factores políticos, económicos o de acuerdo a la experiencia con desarrollos realizados previamente.

2.5.1 Desarrollo seguro de software

La importancia de la seguridad en sistemas informáticos es indiscutible como lo comenté previamente. Más allá de las consideraciones que se toman al momento de diseñar la aplicación, debemos seguir ciertas prácticas dentro del proceso de implementación que refuercen las características de seguridad contempladas en el diseño y que el producto resultado de dicha implementación sea seguro de acuerdo a nuestras necesidades o estándares.

Como caso de ejemplo, he utilizado un proceso de desarrollo de software seguro llamado Security Development Lifecycle (SDL) que fue creado por la compañía Microsoft. Podemos resumir que SDL es un proceso que nos ayuda a garantizar seguridad dentro del proceso de implementación de una pieza de software. Es importante recalcar que el uso de un proceso como SDL no significa que se obtendrá un software cien por ciento seguro, sin embargo se reducirán el número de riesgos y vulnerabilidades.

2.5.2 Security Development Lifecycle

En términos simples, SDL es una colección de actividades obligatorias en materia de seguridad que están ordenadas y agrupadas conforme a las etapas clásicas o comunes del desarrollo de software. Algunas de las actividades pueden realizarse sin depender de las otras y obtener ciertos beneficios, pero conforme se hace uso de SDL y se adquiere experiencia, claramente se puede notar que siguiendo la totalidad de actividades se pueden obtener beneficios de seguridad verdaderamente notables.

En la figura 13 se muestran las actividades agrupadas como se ha mencionado.

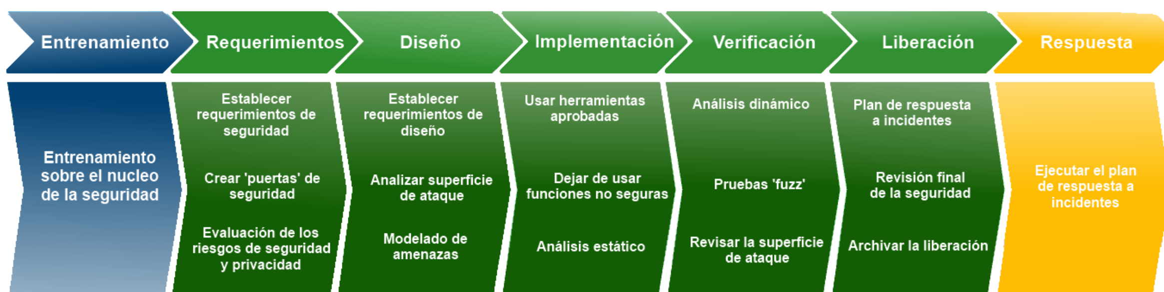

Figura 13. Actividades del Security Development Lifecycle.

Si se desea se pueden agregar actividades a discreción conforme se requiera para cumplir con los requerimientos de seguridad que se tengan.

Dado que personalmente considero a la seguridad como uno de los elementos más importantes a tener en cuenta en la implementación de sistemas (especialmente en aplicaciones enterprise) quiero entrar en más detalle acerca de las diversas actividades a manera de describirlas de forma general.

Entrenamiento

Todos los miembros implicados en el desarrollo del software deben recibir entrenamiento apropiado para permanecer informados acerca de lo básico en seguridad y de las tendencias de seguridad y privacidad.

Algunos de los conceptos fundamentales que se deben contemplar en el entrenamiento son los siguientes:

- Diseño seguro
  - Reducción de la superficie de ataque
  - Defensa en profundidad
  - Principio de privilegios mínimos
  - Seguridad predeterminada
- Modelado de amenazas
- Código seguro
  - Sobrecarga de los buffers
  - Errores de aritmética
  - Ataques Cross-site usando scripts
  - Inyección de código SQL
  - Criptografía débil
- Pruebas de seguridad
  - Diferencias entre pruebas de seguridad y pruebas funcionales
  - Evaluación de riesgos

- Privacidad
    - Tipos de datos sensibles
    - Mejores prácticas para diseñar privacidad

Requerimientos

Requerimientos de seguridad: Un aspecto fundamental en el desarrollo de un sistema seguro es considerar la seguridad y privacidad como lo primero por intentar resolver. El análisis de requerimientos debe de realizarse al mismo tiempo que se está concibiendo el proyecto y se debe incluir la especificación de mínima seguridad requerida.

Puertas de seguridad: Cuando se usa el término de puertas de seguridad, se hace referencia a los mínimos niveles aceptables de calidad en seguridad y privacidad. Definir estos criterios desde el inicio del proyecto nos puede ayudar a entender de mejor forma los riesgos asociados con problemas de seguridad y permite identificar y reparar los posibles errores en la etapa de desarrollo. Conforme la etapa de desarrollo avanza, se pueden hacer mejoras teniendo como base esos mínimos niveles para lograr mejor calidad en la pieza de software.

Evaluación de los riesgos de seguridad y privacidad: Son procesos obligatorios que identifican los aspectos funcionales del software que necesitarán una revisión más profunda y exhausta. Se recomienda que la evaluación considere la siguiente información:

- Partes del proyecto que requieren modelos de amenazas.
- En dónde se requerirán revisiones del diseño de seguridad.
- Partes que requerirán pruebas de penetración (se recomienda que estas pruebas se realicen por un equipo diferente al involucrado en el proyecto).
- Analizar si deberían existir más pruebas para mitigar los riesgos de seguridad.

Diseño

Requerimientos de diseño: La confianza en el diseño de la aplicación debe comenzar desde las etapas tempranas del ciclo de vida del software. La mitigación de problemas de seguridad y privacidad es menos costosa cuando se realiza durante las etapas iniciales. Es muy importante tener en cuenta que algunas aplicaciones se implementan basándose en diseño y código ya existentes previamente, por lo cual se debe hacer un análisis de los riesgos de seguridad existentes en esas versiones y si esos riesgos están limitados por detalles tecnológicos, es decir, que la tecnología usada para la implementación tiene un riesgo de seguridad que no puede ser reparado o mitigado.

Reducción de la superficie de ataque: Reducir las oportunidades dadas a atacantes potenciales para vulnerar el sistema es uno de los objetivos al momento de diseñar la aplicación. Algunos ejemplos de cómo lograrlo son el acceso restringido a los servicios del sistema o aplicar principios de privilegios mínimos; entre otros.

Modelado de amenazas: Es usado en ambientes donde los riesgos de seguridad son significativos. Permite considerar, documentar y discutir las implicaciones de seguridad que tienen los diseños en un contexto del ambiente operacional planeado para su funcionamiento. También se consideran, como parte del modelado de amenazas, los problemas de seguridad a nivel de los componentes de la aplicación, Este modelo debe ser trabajado por todos los perfiles involucrados en la creación del software (administradores, desarrolladores, testers, etc.) y representa la tarea primaria en cuanto a análisis de seguridad en la etapa de diseño.

Implementación

Usar herramientas aprobadas: Los equipos involucrados en el desarrollo deben definir y publicar una lista de herramientas aprobadas para su uso, así como su respectiva evaluación de seguridad. En términos generales, se debe hacer el esfuerzo por utilizar las versiones más recientes de las herramientas para tomar ventaja de la protección a riesgos de seguridad detectados y corregidos. Es una tarea difícil debido a que el uso de tecnologías nuevas suele generar desconfianza debido a la estabilidad incierta que se pudiera tener.

Dejar de usar funciones no seguras: Muchas funciones e interfaces de programación de aplicaciones (API's, por sus siglas en idioma inglés) comúnmente usadas, no son seguras de cara al ambiente de amenazas actuales. Una vez hecho el análisis, se determinará cuáles de éstas serán prohibidas para su uso. Se recomienda tener medios para analizar código y buscar funciones que no deben utilizarse y que sean reemplazadas por sus versiones seguras.

Análisis estático: El análisis del código fuente nos permite conocer si las políticas de codificación se están siguiendo. La utilización de herramientas automatizadas que lleven a cabo esta tarea suele ser insuficiente, y no debería reemplazar una revisión manual del código. Los equipos de desarrollo deben decidir acerca de la frecuencia con que debería realizarse el análisis estático.

Verificación

Análisis dinámico: Para asegurar que la aplicación funcione de manera en la que fue diseñada es necesario tener una verificación en momento de ejecución. Entre las actividades del análisis se deben realizar tareas de monitoreo sobre corrupción de memoria, problemas de privilegios de usuarios y otros problemas de seguridad críticos.

Pruebas Fuzz: Este tipo de pruebas son una forma de análisis dinámico en el cual se induce la falla del sistema introduciendo datos mal formados o aleatorios. De esta manera se pueden encontrar errores en el diseño o implementación de la aplicación.

Revisión de la superficie de ataque: El desviarse significativamente de las especificaciones de diseño y funcionales es algo muy común conforme se avanza entre las etapas del proyecto, por lo cual es crítico realizar revisiones cuando el código haya sido completado. La importancia de que no existan cambios en las especificaciones recae en que un cambio puede introducir nuevos riesgos en la seguridad y que muy probablemente no se contemplaron de forma inicial.

Liberación

Plan de respuesta a incidentes: El plan debe contemplar, al menos, los siguientes puntos:

- Un plan que identifique a los grupos de ingeniería, mercadeo, comunicaciones y administración para, en caso necesario, sean contactados en caso de la presencia de la emergencia.
- Contactos con autoridad para tomar decisiones que puedan ser llamados en cualquier momento de las 24 horas del día, los 7 días de la semana.
- Planes para cuando se hereda código de otros grupos dentro de la organización.
- Planes para código de terceros que se ha licenciado.

Revisión final de seguridad: Previo al lanzamiento de la aplicación, es necesaria una revisión de todas las actividades concernientes a la seguridad que se han realizado hasta el momento. Esta actividad es realizada por un asesor de seguridad en colaboración con el equipo o equipos involucrados en la creación del proyecto. Cabe destacar que esta revisión no tiene como objetivo hacer pruebas a profundidad para corregir los errores, es más bien una revisión a los modelos de amenazas, excepciones y rendimiento de acuerdo a lo contemplado inicialmente.

Archivar la liberación: La liberación del sistema debe estar condicionada al cumplimiento del proceso de seguridad llevado a cabo durante todo el ciclo de la creación del software. Toda la información y datos deben de ser almacenados para su probable uso en el posterior mantenimiento del sistema. Se deben incluir todas las especificaciones, código fuente, binarios, modelos de amenaza, documentación, planes de respuesta; entre otros elementos que se consideren necesarios.[16]

2.5.3 Almacenamiento de datos

Lo más común que podemos encontrar en cuanto a almacenamiento de datos es el uso del modelo relacional, que nos facilita representar problemas reales y administrar los datos dinámicamente. El uso de tablas para representar los datos puede parecer simple en primera instancia, pero para evitar problemas posteriores se deben tener consideraciones importantes en el momento de diseñar dichas tablas y las relaciones con otras tablas.

Tablas

Las tablas son la parte más importante de la capa de almacenamiento de datos, ya que representan la información que será procesada por el sistema. En el momento de creación de las tablas, se debe haber diseñado prácticamente en su totalidad el proceso de software, de lo contrario nos podemos encontrar con muchos problemas que requerirán el crear tablas nuevamente, una tarea bastante ardua, en especial cuando ya existen datos en las tablas.

[16] (Microsoft, 2015) Sitio web de Security Development Lifecycle

Características de una tabla relacional

- Una tabla se percibe como una estructura bidimensional compuesta de filas y columnas.
- Cada fila de la tabla representa una entidad única dentro del grupo de entidades.
- Cada columna de la tabla representa un atributo, y cada columna tiene un nombre distinto.
- Cada intersección fila/columna representa un valor único.
- Todos los valores en una columna deben seguir el mismo formato de datos.
- Cada columna tiene un rango específico de valores conocido como el dominio del atributo.
- El orden de las filas y las columnas es irrelevante para el sistema de gestión de la base de datos.
- Cada tabla debe tener un atributo o combinación de atributos que definen a cada fila como única.

Normalización

El hecho de utilizar el modelo relacional aplicado a nuestras bases de datos no es suficiente para evitar la redundancia de datos, es por ello que hacemos uso de la normalización, un proceso para evaluar y corregir la estructura de las tablas con el fin de minimizar esas redundancias y de ese modo eliminar anomalías en los datos.

La normalización trabaja a través de una serie de escenarios que son llamados formas normales. Los primeros tres escenarios son descritos como primera forma normal (1FN), segunda forma normal (2FN) y tercera forma normal (3FN). Desde un punto de vista estructural, la 2FN es mejor que la 1FN, y la 3FN es mejor que la 2FN. Para la mayoría de los propósitos de diseño de las bases de datos, la 3FN es suficiente dentro del proceso de normalización. De acuerdo a los estándares de la SCJN, la 3FN fue elegida al normalizar las tablas.

A pesar de que la normalización es muy importante para el diseño de las bases de datos, no debemos asumir que el nivel más alto de normalización es siempre el más deseable. Generalmente, entre más alta la forma normal, se necesitará hacer uso de más tablas para consultar los datos deseados y esto podría provocar un desempeño más lento al momento de responder a las demandas del usuario.

Las características principales de las primeras formas normales son mencionadas a continuación.

- 1ra Forma Normal (1FN)
  - Eliminación de grupos repetidos. En otras palabras, cada intersección fila-columna representa un solo valor, no un grupo de valores.
  - Tablas independientes para conjuntos de datos relacionados.
  - Identificación, mediante una clave principal, de cada conjunto de datos relacionados.

- 2da Forma Normal (2FN)
  - Es una primera forma normal.
  - No hay dependencias parciales, es decir, todos los datos que no son clave principal deben depender únicamente de la clave principal.
- 3ra Forma Normal (3FN)
  - Es una segunda forma normal.
  - No contiene dependencias transitivas.

Procedimientos almacenados

Un procedimiento almacenado es una colección nombrada de sentencias SQL, que son almacenadas en una base de datos. Una de las mayores ventajas de los procedimientos almacenados es que pueden ser usados para encapsular y representar transacciones del negocio. Existen dos claras ventajas de utilizar procedimientos almacenados:

- Reducen sustancialmente el tráfico de red e incrementan el rendimiento. Dado que se almacenan en el servidor, no hay transmisión individual de sentencias SQL en la red. El uso de los procedimientos almacenados incrementa el rendimiento del sistema ya que se ejecutan como una única transacción, que es ejecutada localmente en el sistema de gestión de bases de datos.
- Ayudan a reducir la duplicación de código, lo cual se refleja en minimizar la posibilidad de error y reducen el costo del desarrollo de la aplicación y de su mantenimiento.

Es muy importante evitar introducir reglas de negocio dentro de los procedimientos, ya que debería de ser la capa de negocios la que se encargue de dichas reglas. Por lo anterior, es recomendable que los procedimientos sólo realicen tareas simples de creación, lectura, actualización y eliminación de datos; sin mayor complejidad.

Funciones

Las funciones dentro de un manejador de bases de datos son herramientas muy útiles que pueden resolver tareas específicas en el procesamiento de la información, sobretodo en la aplicación de un formato determinado a nuestros datos para obtener los datos representados de acuerdo a nuestra necesidad. Se puede hacer uso de las funciones integradas del manejador de bases en conjunto con funciones creadas por nosotros para obtener las mayores ventajas posibles.

Vistas

Una vista es una tabla virtual que está basada en una consulta de tipo SELECT. La consulta puede contener columnas, columnas calculadas, alias, y funciones de una o más tablas. Algunas de las principales características de las vistas son:

- Las vistas se actualizan dinámicamente. Es decir, la vista es recreada cada vez que es invocada.

- Las vistas proveen un nivel de seguridad en la base de datos debido a que éstas pueden restringir a los usuarios a sólo consultar filas y columnas específicas de una tabla. Por ejemplo, el uso de esquemas en la base de datos puede restringir el acceso al usuario para utilizar sólo el esquema deseado.

El uso de vistas puede facilitar la segmentación de tareas dentro de nuestras consultas, uno de los principios de diseño que se buscan para mantener lo más simple posible la implementación.

Índices

Un índice es un arreglo metódico que se utiliza para acceder a filas en una tabla.

Los índices, bajo un ambiente de bases de datos relacionales, trabajan como los índices de un libro. Desde un punto de vista conceptual, un índice está compuesto por una clave índice y un grupo de apuntadores. La clave índice es, en efecto, el punto de referencia del índice. Formalmente, un índice es un arreglo ordenado de claves y apuntadores. Cada clave apunta a una ubicación de los datos identificados con esa clave.

Un sistema de gestión de base de datos utiliza los índices para varios propósitos. Para nuestro caso, es utilizado para recuperar los datos más eficientemente, además de que los índices nos permiten ordenar los datos por un atributo en específico; por ejemplo, para los atributos que son de tipo cadena de texto, el índice nos permite ordenar los datos alfabéticamente. Cabe destacar que la mayoría de los atributos existentes en las tablas son de tipo texto, por lo cual es sumamente importante contar con índices en la base de datos.

Full-Text

Una búsqueda full-text está diseñada para realizar búsquedas lingüísticas (basadas en el lenguaje) dentro de texto y documentos almacenados en las bases de datos. Con opciones como búsqueda basada en palabras o en frases, características del lenguaje, habilidad para indexar documentos en sus formatos nativos (por ejemplo, documentos de Office y PDF), términos generados por inflexión y tesauro, relevancia, eliminación de palabras ruidosas, etcétera. La búsqueda full-text provee un poderoso grupo de herramientas de búsqueda basada en texto.

Búsqueda fonética (Soundex)

La búsqueda fonética incluye métodos para codificar palabras en sus equivalentes fonéticos. Está implementada mediante algoritmos que intentan aproximar la pronunciación de palabras. En el siglo XX, Robert Rusell y Margaret O'Dell patentaron un sistema de indexación basada en el sonido como Soundex. El sistema Soundex fue diseñado para indexar los registros del censo estadounidense del siglo XIX. Muchos algoritmos fonéticos modernos están basados, hasta cierto grado, en Soundex.

Por supuesto que Soundex tiene varios defectos, pero ninguno de ellos es su simplicidad, la cual fue su atractivo principal antes de la era de las computadoras. La codificación Soundex fue diseñada para que fuera realizada por personal con variantes niveles de educación. Las reglas

tuvieron que ser formuladas con simplicidad, fácil de memorizar, y capaces de ser implementadas con las herramientas más rudimentarias (por ejemplo, papel y lápiz).

Para ejemplificar, SQL Server de Microsoft incluye una versión del algoritmo Soundex que puede ser llamado mediante la función SOUNDEX. Cuando se pasa una cadena como parámetro a dicha función, se regresa un código fonético de 4 caracteres, que consiste de un caracter alfabético seguido de 3 números que representan un grupo de letras que son pronunciadas con similitud.

Queries

Los queries son las operaciones primarias de manipulación de datos, son el mecanismo necesario para traducir las peticiones de la aplicación en acciones de creación, actualización, eliminación y lectura en la base de datos. Dado que los queries son esenciales, deben ser optimizados para maximizar el rendimiento y tasa de transferencia en la base de datos. Se deben considerar los siguientes consejos cuando se usen queries:

- Utilizar queries parametrizados para mitigar problemas de seguridad y reducir la posibilidad de que sucedan ataques del tipo Inyección SQL. No se debe usar la concatenación de cadenas, formadas por información que el usuario ha introducido al sistema, para construir queries dinámicos. En resumen, es recomendable el uso de procedimientos almacenados para alojar estos queries.
- Considerar el usar objetos para construir los queries. Por ejemplo, implementar el patrón "objeto query" o usar el soporte para query parametrizado que proporcionan algunas librerías de acceso a datos. También considerar la optimización del esquema de datos en la base para la ejecución de los queries.

Cuando se construyan queries dinámicos de SQL, si es el caso necesario, evitar mezclar la lógica de procesamiento de negocios con la lógica usada para generar la declaración SQL, ya que ello puede llevar a código difícil de mantener y corregir.

2.5.4 Acceso a datos

Consideraciones generales del diseño

La capa de acceso a datos debe cumplir con los requerimientos de la aplicación, trabajar de forma eficiente y segura, y ser fácil de mantener y extender conforme la capa de negocios lo requiera. Para diseñar esta capa puede tomarse a consideración lo siguiente:

- *Elección de una tecnología apropiada para el acceso a datos.* La elección depende del tipo de datos que manejaremos con la aplicación y como manipularemos dicha información dentro de ella.
- *Utilizar abstracción para implementar una interfaz con bajo acoplamiento.* Puede ser logrado definiendo componentes que traduzcan las peticiones a un formato que sea entendido dentro de la capa.

- *Encapsular la funcionalidad de acceso a datos dentro de la capa.* La capa debe ocultar los detalles del acceso a la fuente de datos. Debe ser responsable de manejar las conexiones, procesar peticiones, y mapear entidades de la aplicación a estructuras de datos provenientes de la fuente.
- *Decidir cómo mapear las entidades de la aplicación a estructuras de datos.* Se debe identificar una estrategia para poblar las entidades de negocios o las estructuras de datos provenientes de la fuente y hacerlas disponibles para la capa de negocios.
- *Decidir cómo se manejarán las conexiones.* Como una regla, la capa debe crear y manejar todas las conexiones a la fuente de datos requeridas por la aplicación. Se debe elegir un método apropiado para almacenar y proteger la información acerca de la conexión.
- *Determinar cómo se manejarán las excepciones de manejo de datos.* La capa debe atrapar y manejar todas las excepciones asociadas con la fuente de datos y las operaciones de creación, lectura, actualización y eliminación. Las excepciones concernientes a los datos, el acceso a éstos y errores por exceso de tiempo de respuesta; deben ser manejados en esta capa y pasados a otras capas si es que las fallas afectan la funcionalidad o respuesta de la aplicación.

La capa de acceso a datos desempeña el rol de comunicación e interacción con la capa de almacenamiento de datos y con la capa de negocios, lo cual debe realizarse de forma segura y eficiente.

Conexiones

Las conexiones a las fuentes de datos son una parte fundamental de la capa de acceso a datos. Todas las conexiones deben ser manejadas por esta capa. Crear y administrar las conexiones consume recursos valiosos tanto en la capa de acceso a datos como en la fuente de estos. Para maximizar el rendimiento y seguridad, se deben tener las siguientes consideraciones en el diseño de las conexiones:

- En general, abrir las conexiones tan tarde como se pueda y cerrarlas lo más pronto posible. Nunca hay que dejar conexiones abiertas por periodos excesivos.
- Realizar las transacciones a través de una conexión simple cuando sea posible.
- Tomar ventaja del agrupamiento de conexiones (connection pool) utilizando un modelo de subsistema de seguridad confiable y evitar el uso de entidades individuales si es posible.
- Considerar el manejo de situaciones donde la conexión a la fuente de datos se pierde o expira.

Formato de los datos

Elegir el formato de datos apropiado nos proporciona interoperabilidad con otras aplicaciones y facilita comunicaciones serializadas a través de diferentes procesos y máquinas físicas. El formato de los datos y la serialización son también importantes para permitir el almacenamiento y

recuperación del estado de la aplicación por medio de la capa de negocios. Algunas consideraciones que se pueden tener al diseñar el formato de los datos son:

- Considerar el uso de XML para interoperabilidad con otros sistemas y plataformas, o cuando se trabaja con estructuras de datos que cambian con el tiempo.
- Cuando se deben transferir datos a través de fronteras físicas, considerar los requerimientos de serialización e interoperabilidad.

Transacciones

Una transacción es un intercambio de información secuencial y actos asociados que son tratados como una unidad atómica para satisfacer una petición y garantizar la integridad de la base de datos.

Una transacción también se considera como completa sólo si toda la información y las acciones están completas, y los cambios asociados en la base de datos son permanentes. Las transacciones soportan el deshacer (rollback) las acciones en la base de datos si surgiera un error, lo cual ayuda a preservar la integridad de los datos en la base.

Es importante identificar el modelo de concurrencia apropiado y determinar cómo administrar las transacciones. Se puede elegir para la concurrencia entre un modelo optimista y uno pesimista. Con la *concurrencia optimista,* los datos no tienen bloqueos y las actualizaciones requieren código para verificar, usualmente comparando contra una estampa de tiempo, que los datos no han cambiado desde que fueron recuperados. Con la *concurrencia pesimista*, los datos son bloqueados y no se puede actualizar la información a menos que los bloqueos sean liberados.

Considerar lo siguiente cuando se diseñan las transacciones:

- Considerar los límites de las transacciones, para que los reintentos y composición sean posibles; y habilitar las transacciones sólo cuando sean necesarias. Los queries pueden no necesitar una transacción explicita, pero debemos asegurarnos que estamos conscientes del comportamiento default de nuestra base de datos en cuanto a las transacciones y el nivel de aislamiento.
- Mantener las transacciones lo más cortas como sea posible para minimizar la cantidad de tiempo en la que existen los bloqueos. Se debe evitar utilizar bloqueos para transacciones de larga duración o bloquear durante el acceso a datos compartidos, lo cual puede bloquear el acceso a datos por otro código. Hacer uso excesivo de bloqueos exclusivos puede ocasionar deadlocks.
- Usar un nivel apropiado de aislamiento, lo cual define cómo y cuándo se permiten cambios por otras operaciones. Un nivel de aislamiento alto ofrece alta consistencia de los datos con costo de concurrencia. Un nivel de aislamiento bajo mejora el rendimiento con costo de consistencia.
- Cuando no se pueda aplicar un commit o un rollback, o si se usa una transacción de larga duración, implementar métodos de compensación para revertir los datos almacenados a sus estados previos en caso de que una operación dentro de la transacción falle.

A pesar de que se hace mención de las transacciones en este capítulo acerca del acceso a datos, es conveniente que el control de las transacciones se encuentre en la capa de negocios, ya que debe ser el negocio el que dicte el inicio y fin de una transacción.[17]

2.5.5 Capa de negocios o dominio

Consideraciones generales del diseño

Al diseñar la capa de negocios (a veces también llamada capa de dominio), el objetivo del arquitecto de software es minimizar la complejidad separando las tareas en diferentes áreas. Por ejemplo, reglas de procesamiento en el negocio, flujos de trabajo, y entidades de negocio; representan diferentes áreas. Dentro de cada área, los componentes diseñados deben centrarse en el área específica, y no deben incluir código relacionado con otras áreas. Considérense los siguientes consejos acerca del diseño de la capa de negocios:

- Decidir si se necesita una capa de negocios. Siempre es buena idea utilizar una capa de negocios separada, donde es posible mejorar la mantenibilidad de la aplicación. La excepción podría ser aplicaciones que tienen pocas o ninguna regla de negocio (sólo validación de los datos).
- Identificar las responsabilidades y consumidores de la capa de negocios. Esto ayudará a decidir que tarea debe cumplir la capa de negocios y cómo debe ser expuesta dicha capa. Usar la capa para procesar reglas de negocio complejas, transformar datos, aplicar políticas y para validación. Si la capa será usada por la capa de presentación y/o una aplicación externa, se puede elegir exponer la capa mediante un servicio web.
- No mezclar diferentes tipos de componentes en la capa de negocios. Utilizar una capa de negocios para evitar mezclar el código de presentación y de acceso a datos con el código de lógica de negocios, y simplificar las pruebas de la funcionalidad de negocios. También se debe usar una capa de negocios para centralizar funciones comunes de lógica del negocio y promover la reutilización.
- Reducir los viajes de ida y vuelta cuando se accede a capas de negocio remotas. Si la capa de negocios se encuentra en una capa física diferente y separada de los clientes con los cuales debe interactuar, se debe considerar implementar una fachada de aplicación basada en mensajes remotos o una capa de servicios.
- Evitar un fuerte acoplamiento entre capas. Usar principios de abstracción para minimizar el acoplamiento cuando se creen interfaces para la capa de negocios. Algunas técnicas de abstracción que se pueden utilizar incluyen el uso de interfaces de objetos públicos, definiciones de interfaces comunes, clases base abstractas o mensajes.

[17] (Microsoft, 2009), con comentarios adicionales por parte de Erik León Mendoza

Caching

El diseño de una estrategia apropiada para caching en la capa de negocios es importante para el rendimiento y la respuesta de la aplicación. El caching se puede usar para optimizar búsqueda de datos, evitar idas y vueltas de información en la red, y evitar procesos duplicados e innecesarios. Como parte de dicha estrategia, se debe decidir cuándo y cómo cargar la información en memoria caché. Para evitar retrasos en el sistema cliente, se recomienda realizar la carga de cache asincrónicamente o utilizando un proceso de batch. Considérense los siguientes puntos para la planeación de una estrategia de caching:

- Considerar el caching con información estática que será reutilizada regularmente dentro de la capa de negocios, pero evitar la información que es volátil. También se debe considerar la información que no puede ser obtenida de la base de datos de manera rápida y eficiente, evitando un largo volumen de información que puede alentar el proceso. Se debe utilizar caching con lo mínimo requerido.
- El caching debe realizarse en un formato que esté listo para ser utilizado dentro de la capa de negocios.
- Evitar, si es posible, utilizar caching con información sensible o diseñar un mecanismo para proteger los datos sensibles en el cache.

Acoplamiento y cohesión

Al diseñar los componentes para la capa de negocios, debemos asegurarnos que éstos son altamente cohesivos y que implementan un acoplamiento flexible entre las capas de la aplicación. Esto ayuda a mejorar la escalabilidad de nuestro sistema. Las siguientes son algunas consideraciones al respecto:

- Evitar dependencias circulares. La capa de negocios sólo debe conocer acerca de la capa de datos y no de la capa de presentación o de aplicaciones externas que acceden directamente a la capa de negocios.
- Utilizar abstracción para implementar una interfaz acoplada flexiblemente. Esto se puede lograr con componentes de tipo interfaz, definiciones comunes de interfaz, o abstracción compartida donde componentes concretos dependen de abstracciones y no de otros componentes concretos.
- El diseño dentro de la capa de negocios debe ser para un rígido acoplamiento a menos que el comportamiento dinámico requiera un acoplamiento flexible.
- Diseñar para alta cohesión. Los componentes deben contener solo funcionalidad específica relacionada con dicho componente. Se debe evitar siempre mezclar el acceso lógico a los datos con la lógica de negocios.
- Considerar utilizar interfaces basadas en mensajes para exponer los componentes de negocios y así reducir el acoplamiento y permitir, si se requiere, que estos componentes se encuentre en una capa física diferente.

Patrón Fachada

El patrón estructural de fachada, proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

Estructurar un sistema en subsistemas ayuda a reducir la complejidad. Un típico objetivo de diseño es minimizar la comunicación y dependencias entre subsistemas. Un modo de lograr esto es introduciendo un objeto fachada que proporcione una interfaz única y simplificada para los servicios más generales del subsistema.

Es conveniente utilizar el patrón de fachada cuando:

- Queremos proporcionar una interfaz simple para un subsistema complejo. Los subsistemas suelen volverse más complicados a medida que van evolucionando. La mayoría de los patrones, cuando se aplican, dan como resultado más clases y más pequeñas. Esto hace que el subsistema sea más reutilizable y fácil de personalizar, pero eso lo hace más difícil de usar.
- Existan muchas dependencias entre los clientes y las clases que implementan una abstracción. Se introduce una fachada para desacoplar el subsistema de sus clientes y de otros subsistemas, promoviendo así la independencia entre subsistemas y la portabilidad.
- Deseamos dividir en capas nuestros subsistemas. Se usa una fachada para definir un punto de entrada en cada nivel del subsistema. Si éstos son dependientes, se pueden simplificar las dependencias entre ellos haciendo que se comuniquen entre sí únicamente a través de sus fachadas.

Manejo de excepciones

Es recomendable diseñar una estrategia de manejo de excepciones centralizada para que las excepciones sean capturadas y arrojadas consistentemente en la capa de datos. Se debe poner particular atención a las excepciones que se propagan a través de las capas. Se deben diseñar las excepciones no manejadas para evitar la exposición de información sensible de la aplicación. Para lo anterior, deben tomarse las siguientes consideraciones cuando se diseñe la estrategia de manejo de excepciones:

- Identificar las excepciones que deben ser capturadas para que no se den situaciones de dudosa fiabilidad del sistema. Por ejemplo, los 'deadlock' y problemas de conexión pueden frecuentemente ser resueltos dentro de la capa de datos. Sin embargo, algunas excepciones, como las violaciones de concurrencia, deben ser presentadas al usuario para una resolución.
- Diseñar una estrategia de propagación de excepciones que sea apropiada. Por ejemplo, permitir que las excepciones se propaguen fuera de las fronteras de las capas donde pueden ser registradas y transformadas como sea requerido para pasar a la siguiente

capa. Se debe considerar incluir un identificador del contexto para que las excepciones relacionadas puedan ser asociadas a través de las capas.

- Considerar la implementación de un proceso de repetición para las operaciones donde existen errores en la fuente de los datos o cuando ocurren timeouts, pero solo si es seguro hacerlo.
- Asegurarse de limpiar los recursos y estados después de que ocurre una excepción.
- Diseñar apropiadamente estrategias de registro y notificación para errores críticos y excepciones.

2.5.6 Capa de presentación

Consideraciones generales del diseño

Existen varios factores claves que deben ser considerados cuando se diseña la capa de presentación. Se recomiendan los siguientes principios para asegurarse de que el diseño cumple con los requerimientos de la aplicación:

- Elegir el tipo de aplicación apropiado. El tipo de aplicación que se elija tendrá un considerable impacto en las opciones que se tienen para la capa de presentación. Se debe determinar si se implementará una aplicación cliente inteligente, un cliente web, o una aplicación de Internet. La decisión debe estar basada en los requerimientos de la aplicación, así como en las restricciones organizacionales y de infraestructura.
- Elegir una tecnología de interfaz de usuario (IU) apropiada. Los diferentes tipos de aplicaciones proveen diferentes grupos de tecnologías que pueden ser utilizadas para desarrollar la capa de presentación. Cada tipo de tecnología tiene distintas ventajas que pueden afectar la habilidad para crear un diseño de capa de presentación adecuado.
- Diseñar una separación de funcionalidades. Utilizar componentes de IU dedicados que se enfoquen en el dibujado, despliegue e interacción del usuario. Considerar la utilización de componentes dedicados a la lógica de la presentación para administrar el procesamiento de la interacción del usuario donde ésta sea compleja. También se debe considerar utilizar entidades dedicadas de presentación para representar la lógica del negocio y los datos en una forma que sea fácilmente consumible por la IU y los componentes de la capa de presentación.
- Considerar directrices de interfaces humanas. Es recomendable implementar directrices de la organización para diseñar la IU, donde se incluyan factores como la accesibilidad, localización, y usabilidad cuando se diseña la capa de presentación. Se pueden revisar directrices de IU establecidas para la interactividad, usabilidad, compatibilidad del sistema, cumplimiento de estándares, y patrones de diseño; basándonos en el tipo de cliente y las tecnologías que fueron elegidas.
- Apegarse al principio de diseño. Antes de diseñar la capa de presentación, se debe entender al cliente. Utilizar sondeos, estudios de usabilidad y entrevistas para

determinar el mejor diseño de la presentación para cumplir con los requerimientos del usuario.

Comunicación

Es recomendable manejar las peticiones con un periodo largo de respuesta teniendo en cuenta la capacidad de respuesta en tiempo por parte del sistema, así como la mantenibilidad y capacidad de pruebas del código. Considerar lo siguiente cuando se diseñe el procesamiento de las peticiones:

- Considerar la utilización de operaciones asíncronas o hilos de trabajo para evitar el bloqueo de la IU en acciones que tomen mucho tiempo. Proveer retroalimentación al usuario acerca del progreso de la acción en curso que está tomando mucho tiempo en ser completada. Considerar el permitir al usuario el cancelar la operación.
- Evitar mezclar el procesamiento de la IU con la lógica de dibujado.
- Cuando se realicen llamadas costosas a fuentes remotas, como las peticiones a la base de datos, considerar si es posible y conveniente realizar varias peticiones pequeñas o una sola petición con mayor volumen de información. Si el usuario requiere un largo volumen de datos para completar una tarea, considerar obtener sólo lo necesario para mostrar y empezar la tarea, para entonces obtener de manera incremental el resto de los datos utilizando un hilo de fondo o a petición del usuario (paginación de los datos y virtualización de la IU son algunos ejemplos).

Navegación

Se debe diseñar una estrategia de navegación para que los usuarios puedan navegar fácilmente a través de las pantallas o páginas, de tal forma que se pueda separar la navegación del procesamiento de la IU. Garantizar que se muestren hipervínculos o controles de navegación en una forma consistente en toda la aplicación de forma que se reduzca la confusión del usuario y para ocultar la complejidad de la aplicación. Considérense las siguientes observaciones:

- Diseñar barras de herramientas y menús para ayudar al usuario a encontrar la funcionalidad proporcionada por la IU.
- Considerar la utilización de tutoriales que expliquen la navegación a través de la IU.
- Determinar cómo preservar el estado de la navegación entre sesiones si es necesario.
- Evitar la duplicación de lógica para los manejadores de eventos en la navegación, y evitar que los caminos de la navegación sean poco flexibles.

Validación

Es crítico para la seguridad de la aplicación diseñar una estrategia de entrada efectiva de datos y de validación de éstos. Se deben determinar las reglas de validación para los datos recibidos desde otras capas y de componentes de terceros, así como de la base de datos.

Se debe tener conciencia de las fronteras en la aplicación para validar cualquier dato que cruce dichas fronteras. Considérense los siguientes puntos en cuanto al diseño de una estrategia de validación:

- Validar todos los datos recibidos dentro de la capa de acceso a datos. Se debe garantizar un manejo correcto de valores nulos, además de filtrar caracteres inválidos.
- Considerar el propósito de los datos cuando se diseña la validación.
- Regresar mensajes de error que informen si una validación falla.

Experiencia de usuario

Una buena experiencia de usuario puede hacer la diferencia entre una aplicación usable y una que no lo es. El rendimiento percibido es más importante que el rendimiento real. Por ejemplo, a los usuarios puede no importarles esperar más para que una página cargue si han obtenido retroalimentación con una estimación de en cuánto tiempo será cargada, y si este tiempo de espera no interfiere con sus actividades. En otras situaciones, un pequeño retraso (incluso fracciones de segundo) para algunas acciones de la IU, pueden hacer sentir a la aplicación como no responsiva. Se debe considerar estudios de usabilidad, encuestas y entrevistas para entender lo que el usuario requiere y espera de la aplicación; y diseñar de forma que se logre una IU eficiente con estos resultados en mente. Tómense en cuenta las siguientes consideraciones cuando se diseñe la experiencia de usuario:

- No diseñar interfaces sobrecargadas o muy complejas. Proveer un camino claro a través de la aplicación para cada escenario clave del usuario, y considerar usar colores y animaciones no invasivos para atraer la atención del usuario cuando existan cambios importantes en la IU, como por ejemplo cambios de estado en la información visualizada.
- Proveer información de ayuda y mensajes de error informativos, sin exponer datos sensibles.
- Para acciones que pueden tomar tiempo en completarse, debe intentarse evitar el bloqueo a las acciones del usuario. Como mínimo, proveer retroalimentación acerca del progreso de la acción, y considerar si el usuario debe tener la posibilidad de cancelar el proceso.
- Considerar conferir poder al usuario otorgándole flexibilidad y configuración de la IU, permitiéndole personalización.

El usuario mantiene el control

- Definir modos de interacción a manera de no forzar al usuario a acciones innecesarias o no deseadas.
- Proveer una interacción flexible.
- Permitir que la interacción del usuario sea interrumpible y pueda deshacer cambios.
- Hacer más eficiente la interacción de acuerdo al avance en las habilidades del usuario.

- Permitir que la interacción sea modificable en cuanto a preferencias del usuario.
- Ocultar al usuario detalles técnicos.
- Diseñar interacción directa con objetos que aparecen en la pantalla.

Reducir que el usuario recurra a su memoria

- La interfaz debe ser diseñada para reducir el requerimiento de recordar acciones o resultados pasados.
- Las características por default deben ser apropiadas para el usuario promedio, pero un usuario debe ser capaz de especificar preferencias individuales.
- Definir accesos rápidos intuitivos.
- El diseño visual de la interfaz debe estar basado en una metáfora del mundo real.
- Revelar información progresivamente. La interfaz debe ser organizada jerárquicamente; esto es, información acerca de una tarea, un objeto o algún comportamiento debe ser presentado primeramente en un nivel alto de abstracción. Los detalles deben ser presentados después de que el usuario indica su interés.

Hacer la interfaz consistente

- Permitir al usuario colocar la tarea actual en un contexto significativo.
- Mantener la consistencia a través de la familia de aplicaciones.

Principios de diseño de las interfaces de usuario

La figura 14 nos muestra una breve descripción de los principios de diseño que se recomiendan seguir al construir interfaces de usuario.

Familiaridad del usuario	La interfaz debe utilizar términos y conceptos obtenidos de la experiencia de las personas que más utilizan el sistema.
Uniformidad	Siempre que sea posible, la interfaz debe ser uniforme en el sentido de que las operaciones comparables se activen de la misma forma.
Mínima sorpresa	El comportamiento del sistema no debe causar sorpresa a los usuarios.
Recuperabilidad	La interfaz debe incluir mecanismos para permitir a los usuarios recuperarse de los errores.
Guía de usuario	Cuando ocurran errores, la interfaz debe proporcionar retroalimentación significativa y características de ayuda sensible al contexto.
Diversidad de usuarios	La interfaz debe proporcionar características de interacción apropiadas para los diferentes tipos de usuarios del sistema.

Figura 14. Principios de diseño de interfaces de usuario.

Categorización de los usuarios

- *Novatos.* Sin conocimiento sintáctico del sistema y pequeño conocimiento semántico de la aplicación o de uso de la computadora en general.
- *Usuarios intermitentes.* Conocimiento semántico de la aplicación razonable pero relativamente baja habilidad en el uso de la interfaz.
- *Usuario frecuentes.* Buen conocimiento semántico y sintáctico de la aplicación. Es común que busquen modos de interacción abreviados o accesos rápidos.

Consideraciones del rendimiento

Para maximizar el rendimiento de la capa de presentación en la aplicación se recomienda considerar lo siguiente:

- Diseñar la capa de presentación cuidadosamente para que contenga la funcionalidad requerida para entregar al cliente una experiencia de usuario rica y de rápida respuesta. Por ejemplo, garantizar que la capa de presentación se encuentra habilitada para validar la entrada del usuario de una manera rápida, sin recurrir al uso de otras capas. Lo anterior puede requerir que las reglas de validación de los datos sea representada en la capa de presentación.
- La interacción entre la capa de presentación y la capa de negocios debe ser asíncrona. Esto evita la posibilidad de latencia alta o conectividad intermitente que afecten adversamente la usabilidad y respuesta de la aplicación.
- Considerar el caching de los datos que se desplegarán al usuario.[18]

2.6 Consolidación de la metodología propuesta

Hasta este momento hemos propuesto una serie de buenas prácticas que se recomiendan seguir a lo largo del proceso de construcción del software; en este capítulo intentaremos formalizar las características y uso de la metodología propuesta.

Previamente a describir la metodología, mencionaré las situaciones que motivaron su origen y en las que se adecua su uso:

- Existen entornos en los que las tomas de decisiones más importantes son realizadas por personas ajenas a la administración del proyecto de software; decisiones motivadas por factores políticos, económicos y/o sociales que finalmente determinan los recursos económicos y, mucho más importante, el tiempo disponible para la construcción del software. Debido a ello, el tiempo disponible para construir en su totalidad el software no es determinado por un análisis de requerimientos y la estimación del tiempo requerido

[18] (Microsoft, 2009), con comentarios adicionales por parte de Erik León Mendoza

para completar las tareas tanto del mismo análisis, como las de diseño, implementación y validación. La ventana de tiempo disponible desde el inicio del proyecto hasta su entrega puede estar determinada mucho antes de siquiera empezar con la especificación del software.

- En una situación ideal, el grupo de trabajo involucrado en la construcción del software durante todas sus etapas estará dedicado únicamente a un proyecto de software. Sin embargo, existen ocasiones en las que dicho equipo tendrá que realizar actividades concurrentes de distintos proyectos de software. La escases de tiempo y recursos humanos se vuelven críticos en dichas situaciones.
- No hay que perder los pies del piso. El desarrollo de software no es magia. Se debe ser responsable sea cualquier rol que se desempeñe en el ciclo de creación de software, incluso el rol de cliente. A lo largo de la historia de creación de software, de cualquier clase de índole, se han cometido errores costosísimos, no sólo en cuestión económica sino también en pérdida de vidas. Suena exagerado, pero es cierto. Muchos de esos errores se dieron por una mala administración del proyecto, generalmente en tener expectativas irreales. Si un análisis (por supuesto bien realizado) concluye que se necesita determinado tiempo para construir el software; suponer que se puede realizar en menor tiempo con la premisa de presionar al equipo de trabajo (o incluso algo peor, como amenazas enfocadas a la pérdida del empleo), horarios excesivos de trabajo, etcétera; es totalmente irresponsable. Es realmente lamentable que se piense que trabajar bajo dichas condiciones es sinónimo de profesionalidad. Como cliente tienes que ser comprehensivo, si eres líder del proyecto no puedes prometer cosas por el deseo de quedar bien. Mucho del éxito del software depende de tener un gran equipo de trabajo que no solamente cuente con capacidades técnicas destacables, sino que cuenten con un ambiente de trabajo favorable.
- Tenemos que aprender a priorizar. ¿Qué pasa en la situaciones que mencionamos en el primer punto, cuando la fecha de entrega es inamovible y el tiempo necesario para completar el software sobrepasa dicha fecha? Irremediablemente en ocasiones se tiene que sacrificar algo, y la decisión final es del cliente. ¿Se prefiere seguridad o mantenibilidad? ¿Se prefiere experiencia de usuario o eficiencia? La clave en esta situación siempre será la honestidad con los clientes.
- El uso de métodos de desarrollo de software rápido generalmente tienen la falta de documentación del software como un problema en común. Lo más recomendable en dicha situación es utilizar una implementación orientada al dominio, es decir, tener una capa de dominio (también conocida como capa de negocios) que sea la base de el software. Si lo hacemos de forma correcta la implementación hablará por sí misma y la falta de documentación no será un problema mayúsculo.
- La combinación de reglas de negocio muy extensas y/o complejas con desarrollo rápido de software pueden dar como resultado un alto riesgo, ya que los miembros del equipo responsable de la construcción del software no solo necesitan de conocimientos técnicos, sino también de reglas de negocio que implican una curva de aprendizaje costosa en

esfuerzo y tiempo; un lujo que no se puede tener cuando el tiempo es lo que falta. En conclusión: cuida a tu equipo, podrían ser más valiosos que la implementación misma.

- Las decisiones importantes no se toman por únicamente una persona. Parece que este punto se contradice con el primero, sin embargo hago mayor hincapié en las decisiones técnicas y de diseño. Es muy conveniente involucrar a demás miembros del equipo; uno puede poseer mucho conocimiento, pero eso no exime de cometer errores. El dicho "Dos cabezas piensan mejor que una", también aplica en la construcción de software.

La metodología propuesta cuenta con las etapas comunes de otras metodologías existentes, me refiero a las etapas de especificación, desarrollo, verificación y evolución. Sin embargo, el mayor énfasis se tiene en la etapa de desarrollo, comprendiendo lo concerniente al diseño e implementación de la solución. La razón puede verse obvia al analizar detenidamente las situaciones descritas con anterioridad. La falta de suficiente tiempo generalmente afectará a que algunas de las etapas del proceso reciba menos atención. Si bien la especificación y la validación del software son importantísimas, las omisiones o errores cometidos en estas etapas se verán siempre reflejados en la etapa de desarrollo, ya que es ahí donde se hacen las correcciones. Un buen diseño e implementación pueden soportar las deficiencias de otras etapas.

Lo anterior nos da la razón de por qué a lo largo de este documento se hizo mayor énfasis en las recomendaciones para la etapa de desarrollo. Recalco que si bien son las consideraciones y principios fundamentales, no debemos limitarnos a lo presentado a lo largo de cada capítulo.

Si bien la metodología se enfoca principalmente en la etapa de desarrollo, existen elementos que deben existir a lo largo de todo el proceso de construcción del software. A esos elementos los he llamado pilares de la construcción de software, cada uno indispensable para la metodología propuesta.

2.6.1 Pilares de la metodología

- **Equipo humano de creación de software**. En las situaciones que dieron origen a la metodología mencionadas anteriormente se hizo notar la importancia del equipo de trabajo. Ténganse las siguientes consideraciones al respecto:
    - Selecciona correctamente a tu equipo de trabajo. Muchas veces uno se deja llevar por un curriculum que se ve impresionante, sin embargo uno puede decepcionarse después de un tiempo. Las personas encargadas de conseguir al personal pueden ser totalmente ajenas a las formas de trabajo y características técnicas de cada perfil involucrado en la creación del software. En medida de lo posible hay que involucrar a una persona con el mismo perfil o similar para recibir una opinión acerca de las capacidades de los candidatos. Sin embargo se tiene que tener cuidado en que dicha persona de apoyo a la elección no actúe por intereses propios, se desearía que fuera un consejo imparcial.

- No todos tienen las mismas capacidades. No todos somos iguales, cada quien destaca más en algún campo. Algunas personas pueden poseer el mismo conocimiento pero trabajar a velocidades distintas. Se debe conocer al equipo de trabajo para potencializar su aprovechamiento.
- La comunicación es la base de resolución de problemas. Tener un equipo con personas talentosas técnicamente suele no ser suficiente si la comunicación entre ellos es deficiente. Es una consideración que se tiene que tener en cuenta al elegir a los miembros del equipo.

- **Desarrollo seguro de software**. En el documento se dedicó un capítulo completo para hablar del tema, pero me gustaría volver a hacer énfasis. La seguridad del software ya no es una característica deseable, es un DEBE TENERSE, es parte del núcleo del software. La seguridad es el mayor problema de los productos de software en la actualidad.
- **El cliente es parte del equipo**. En el mundo de creación de software actual el cliente ya no únicamente es participe desempeñando el rol de consumidor, ahora es un creador. Lo anterior no solamente sucede en la creación de software a la medida, también en el de propósito general. Las reseñas a los productos de software han sido de gran ayuda en términos de retroalimentación. Hago un recordatorio sobre lo mencionado en el capítulo acerca de los buenos atributos de software: la medida de que tan bueno es el software depende de la satisfacción del cliente.

La figura 15 muestra un modelo de la metodología propuesta.

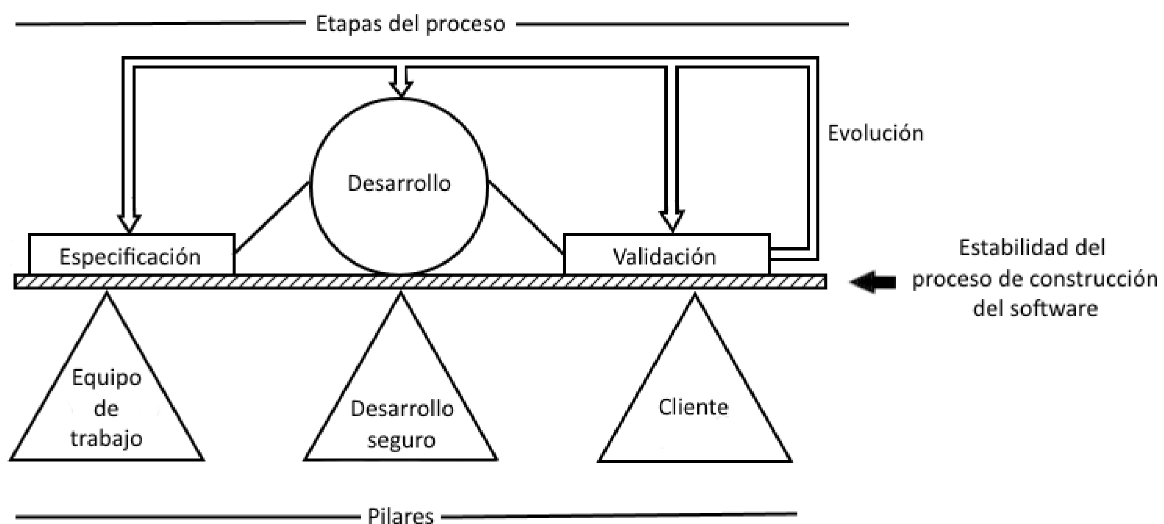

Figura 15. Modelo de la metodología propuesta.

PARTICIPACIÓN PROFESIONAL

La metodología descrita anteriormente fue conceptualizada y utilizada durante mi participación en los proyectos de la Suprema Corte de Justicia de la Nación en un periodo de 4 años y 2 meses. En ese periodo desempeñe las funciones y responsabilidades asignadas a los perfiles de Ingeniero de Software C (8 meses), Ingeniero de Software A (1 año) y finalmente como Arquitecto de Soluciones Tecnológicas (2 años y medio).

Algunas de esas funciones y responsabilidades las describo a continuación:

- Liderar la arquitectura, diseño, implementación y pruebas de las soluciones tecnológicas.
- Guiar los esfuerzos en el área de desarrollo.
- Análisis técnico y conceptual de las necesidades del usuario.
- Comunicar los conceptos de negocio y los requerimientos al equipo de desarrollo.
- Implementar soluciones de software utilizando diversos enfoques (cliente-servidor, aplicaciones distribuidas, aplicaciones de escritorio, aplicaciones web, servicios de Windows) y diversas tecnologías y lenguajes de programación.

Gracias al esfuerzo de los miembros de mi equipo, fue posible poner a disponibilidad del cliente los siguientes sistemas de la SCJN: Sistema de Informática Jurídica (SIJ), Módulo de Intercomunicación (MINTER) y el Sistema Electrónico del Poder Judicial de la Federación (SEPJF).

CONCLUSIONES

La construcción de los sistemas informáticos utilizados en la SCJN continua hasta la fecha, ya sea en forma de nuevos módulos como parte de sistemas ya existentes, actualizaciones o incluso sistemas nuevos que pueden o no colaborar con los ya existentes. Esta construcción, que implica análisis, diseño, implementación y pruebas; siempre se hace persiguiendo el objetivo primordial que es facilitar la impartición de justicia.

El proceso de expedientes dentro de la SCJN se ha vuelto más eficiente en tiempo como en uso de recursos físicos, citando como ejemplo la reducción en el uso de papel. Día a día se buscan mejorar las herramientas tecnológicas para una mejor productividad y satisfacción del usuario; no se trata sólo de automatizar procesos utilizando tecnología actual, sino maximizar su integración con las personas que la utilizan. Dicho objetivo se ha logrado siguiendo la metodología aquí descrita, sin embargo no se limita exclusivamente a lo plasmado en este reporte, y más importante, esta metodología continua en desarrollo paralelamente al proceso de construcción de sistemas informáticos en la SCJN. Si profundizara en cada uno de los temas aquí descritos el resultado sería uno o más libros de cientos de páginas, sin embargo se intentó darle un tratamiento general al proceso para obtener un resumen de lo más importante.

Si bien el estudio de una carrera profesional en ocasiones no te proporciona todas las herramientas necesarias para una participación plena en el mundo laboral, te da la formación necesaria en términos de capacidad de análisis y resolución de problemas de diversa índole. Eso es lo que recibí de la Facultad de Ingeniería, por lo cual estoy agradecido.

Durante mi colaboración de más de 4 años he logrado obtener una gran experiencia, pero sobretodo, una enorme gratificación al ser parte del equipo que ha logrado poner en marcha sistemas de información que son importantes no solo dentro de la SCJN sino también para el resto de los órganos de justicia en el resto del país e incluso para cualquier ciudadano mexicano.

BIBLIOGRAFÍA

Martin, R., & Martin, M. (2006). *Agile principles, patterns and practices in C#.* Prentice Hall.

Microsoft. (2009). *Microsoft application architecture guide, patterns & practices.*

Microsoft. (2015). *Sitio web de Security Development Lifecycle (SDL)*. Obtenido de Sitio web de Security Development Lifecycle (SDL): http://www.microsoft.com/security/sdl/default.aspx

Pressman, R. (2009). *Software engineering, a practitioner's approach.* Mc Graw Hill.

Somerville, I. (2005). *Software engineering.* Pearson.

Suprema Corte de Justicia de la Nación. (2015). *Portal web de la Suprema Corte de Justicia de la Nación*. Obtenido de www.supremacorte.gob.mx