# INSTITUTO UNIVERSITARIO AERONÁUTICO
# FACULTAD DE CIENCIAS DE LA ADMINISTRACIÓN

## PROYECTO DE GRADO – TRABAJO FINAL

**AÑO: 2013**

# REINGENIERÍA
# DE APLICACIONES LOCALES A APLICACIONES DISTRIBUIDAS

### SISTEMA DE GESTIÓN ADMINISTRATIVA CON EL PATRON MVC

**AUTOR:**
REYNOSO, Ricardo
**TUTOR:**
DIAZ, Daniela

# RESUMEN

Gran cantidad de sistemas y programas funcionan en forma práctica y eficiente, pero limitados a un alcance geográfico puntual, uniendo servidores y clientes por una conexión LAN (Local Area Network). La evolución constante de la tecnología, hace necesaria la adecuación de los sistemas y programas antes referidos.

Los cambios y avances tecnológicos una vez incorporados a los sistemas y programas ya existentes, deben afectar lo menos posible para mantener en un mínimo el rechazo en los usuarios finales.

En experiencias anteriores a este nuevo desafío he tenido que pasar de capas tecnológicas tales como diseños en forma de texto y ejecución secuencial y estructurada (siguiendo un orden establecido), a una programación guiada por eventos y con interfaces gráficas, lo cual implicó un verdadero cambio, con resultados satisfactorios, y por tanto me alienta a este nuevo proyecto.

El traspaso de una tecnología a otra hace eje en la utilización de bases de datos robustas que perduran al paso de las distintas versiones, como repositorio de los datos informáticos.

La meta del trabajo es migrar a las aplicaciones distribuidas, las cuales tienen importantes ventajas: se comunican globalmente por internet, pueden usar como soporte de transmisión redes de ancho de banda reducido, tales como enlaces radioeléctricos y además permiten ofrecer servicios web a terceros.

Opto por las metodologías ágiles que se fundan principalmente, en dos pilares: a) preferir lo simple y adaptable a lo predictivo y b) centrarse en las personas y no en los procesos. Por eso utilizo Programación Extrema (XP), que permite obtener versiones en corto tiempo y así posibilita una realimentación continua. Esto permite corregir errores tempranamente durante el desarrollo de la investigación y la implementación de los resultados obtenidos.

En la primera parte del trabajo defino la situación y el problema generado en los sistemas de información y gestión de una institución, debido al paso del tiempo. La desactualización se produce por el avance del hardware y por los nuevos requisitos de interoperabilidad de los sistemas informáticos.

En la segunda parte del proyecto doy el marco teórico de la reingeniería de sistemas para trabajar con "código heredado", de manera que la transformación a nuevos sistemas se haga con la mayor eficacia y funcionalidad, y el menor costo para todas las partes involucradas.

En la tercera parte realizo la investigación en base a los sistemas actualmente en funcionamiento, y en primer lugar trato de modificar la arquitectura de manera de no interferir con el normal funcionamiento, debido a los inconvenientes que eso provoca en los usuarios finales del aplicativo. Luego avanzo en ideas generales, manteniendo compatibilidades tecnológicas (Windows, Visual Studio.Net, Microsoft SQL Server, etc.) con los desarrollos en VFP y MSSQL. Posteriormente tomo decisiones en la arquitectura de comunicación y pongo la investigación en el terreno de cómo serán los futuros escenarios de los sistemas en funcionamiento.

En la cuarta parte muestro las distintas herramientas que utilizo para implementar los resultados obtenidos de esta investigación. Muchas de estas herramientas están en constante evolución de manera que el escenario es bien dinámico y lo desarrollado en esta parte, deberá tener de actualizaciones constantes.

Por último en la quinta parte muestro un desarrollo arquitectónico del sistema que responde al patrón MVC y una implementación del sistema social, administrativo y de gestión de la institución mutual derivada de los resultados obtenidos en este proyecto.

# AGRADECIMIENTOS:

A Dios por sobre toda las cosas.

A mi familia por su apoyo en esta tarea, y en forma infinita a mi esposa Marcela por su ayuda y comprensión.

A toda la gente del IUA que durante estos años tuvieron su atención para conmigo.

A los tutores y compañeros con quienes compartí mis horas en las aulas del Liceo Aeronáutico en Funes (Rosario- Sta. Fe)

A los tutores Daniela y Gustavo por sus positivos aportes a este trabajo.

# DEDICATORIA:

Dedico este trabajo a Marcela, por la suma de cualidades que me contagian en cada momento haciendo que todo sea posible.

También dedico a mis padres Milta y Leandro, que ya no están pero los sigo admirando por sus personalidades y los valores que me supieron legar y que aún hoy siguen alimentando mi espíritu.

A mis hijos Guillermo y Santiago a los que quiero transmitir y contagiar las ganas de superarse en todo momento y hacer el esfuerzo en cada acto de sus vidas para lograr mejores resultados.

# PROYECTO DE GRADO

# 1.0 INTRODUCCIÓN

La tecnología informática ocupa un papel importante en el desarrollo de una organización, debido a que cada vez más se depende de los sistemas de información como soporte de las distintas operaciones y tareas. Es común que exista una brecha entre los aspectos organizacionales y los sistemas de información existentes. Disminuir esa distancia es muy importante para lograr un mejor entendimiento de ambas partes con el fin de brindar un mejor y más eficiente servicio al usuario (cliente), y lograr los beneficios de una informatización creciente y en continuo desarrollo.

Con el paso de los años, evolución tecnológica mediante, los sistemas de información se han hecho cada vez más complejos. Hoy es normal que una organización cuente con bases de datos relacionales para gestionar grandes volúmenes de datos, con sistemas con interfaces gráficas complejas.

Pero las nuevas exigencias pasan por unir en tiempo real distintos puntos geográficos. Es el caso de la interrelación con otros sistemas en terceras organizaciones tales como proveedores y clientes. Estos nuevos requerimientos generan nuevas aplicaciones, nuevas tecnologías y conceptos evolucionados en continuo cambio. La consolidación de todas ellas para lograr un resultado esperado es una tarea para nada sencilla.

El problema más crítico en cualquier tipo de empresa (comercial, de servicios, estatal, etc.) es implementar los cambios requeridos en sus procesos de negocios de manera rápida. Para ello es necesario transformar un ambiente de negocios inflexible y estático en una plataforma flexible y ágil.

Este es uno de los mayores desafíos que debe vencer la tecnología de la información (IT): su capacidad para reflejar los cambios, tanto, en los negocios como en las herramientas de aplicación. Siempre fue motivo de preocupación para los responsables del área IT, el mantenimiento y actualización de los sistemas informáticos. Desde ya es imposible obviarlo pues las exigencias crecientes de los negocios así lo imponen. Pero minimizar los costos materiales y operativos es una preocupación que pone límite a cualquier proyecto de este tipo.

Solucionar el problema, es decir lograr una integración y coordinación de los negocios con nuevas herramientas para transformar lo ya realizado (todos los sistemas que están en marcha y operativos), incorporar nuevas funciones, sin perder la confiabilidad, seguridad, simplicidad y considerando el factor humano como parte del problema (cliente y usuario), significa encontrar un modelo arquitectónico funcional que lo resuelva. Su utilización debe ser amigable, que entusiasme, produciendo una retroalimentación positiva con las experiencias del usuario.

La evolución del software comenzó con aplicaciones concentradas que resolvían necesidades específicas pero no podían comunicarse entre ellas. Como cada una de esas aplicaciones debía ser autosuficiente, muchas de ellas hacían las mismas cosas, pero cada una a su modo. Debido a esto los diversos sistemas de IT de las empresas hoy no pueden acceder o procesar los datos desde un sistema a otro, o desde un entorno informático alejado físicamente del sistema central. La situación creada a partir de estructuras arquitectónicas rígidas se traduce en una incapacidad para evolucionar tecnológicamente y adaptarse a los cambios que el negocio requiere.

**San Martín Mutual, Social y Biblioteca**, es una organización que nació como una entidad deportiva. Prestaba y presta dentro de un espacio físico una infraestructura para la práctica de ciertos deportes. Sus integrantes se vinculan a través de la condición de socios, que en asamblea eligen su comisión directiva. Para suministrar el servicio financiero se transformó en una mutual, además anexó la biblioteca pública José Mármol. La mutual es la encargada de encauzar operaciones financieras de sus socios, tales como ahorro variable (cajas de ahorro), ahorro a término (plazo fijo), ayuda económica (préstamos), gestión de valores (cheques), etc.

La evolución del negocio hizo que en la parte informática se solucionaran los problemas con las herramientas del momento, en forma sectorizada.

Cuando se planteó la posibilidad de abrir primero un punto de atención al público en otro lugar físico, más una sucursal en otra localidad vecina, como también la descentralización de las tareas propias de administración deportiva, se puso como requisito a tener en cuenta, pensar en una concentración de datos que permita un funcionamiento integrado.

La idea original del proyecto, es lograr un sistema distribuido en cuanto al acceso a funcionalidades, pero con un acceso unificado a los datos. Para lograr esto en un principio, se pensó en formar varios puntos que trabajen en forma comunicada, de manera de tener en cualquier momento la totalidad de la información, fundamentalmente para validar las operaciones que se realicen. Luego eso mudó a una arquitectura web con distribución de información a través de servicios.

Uno de los ejes centrales es encontrar una forma de trabajo que esté abierta al cambio, siempre presente en este tipo de actividad. Por eso la búsqueda estuvo en un punto medio entre un desarrollo documentado en extremo, definido al mínimo detalle y un desarrollo sin control, en el que resulte imposible prever sus resultados.

Para entender cómo está compuesta la organización y ver las distintas partes que se interrelacionan, a continuación, muestro el organigrama de toda la institución:

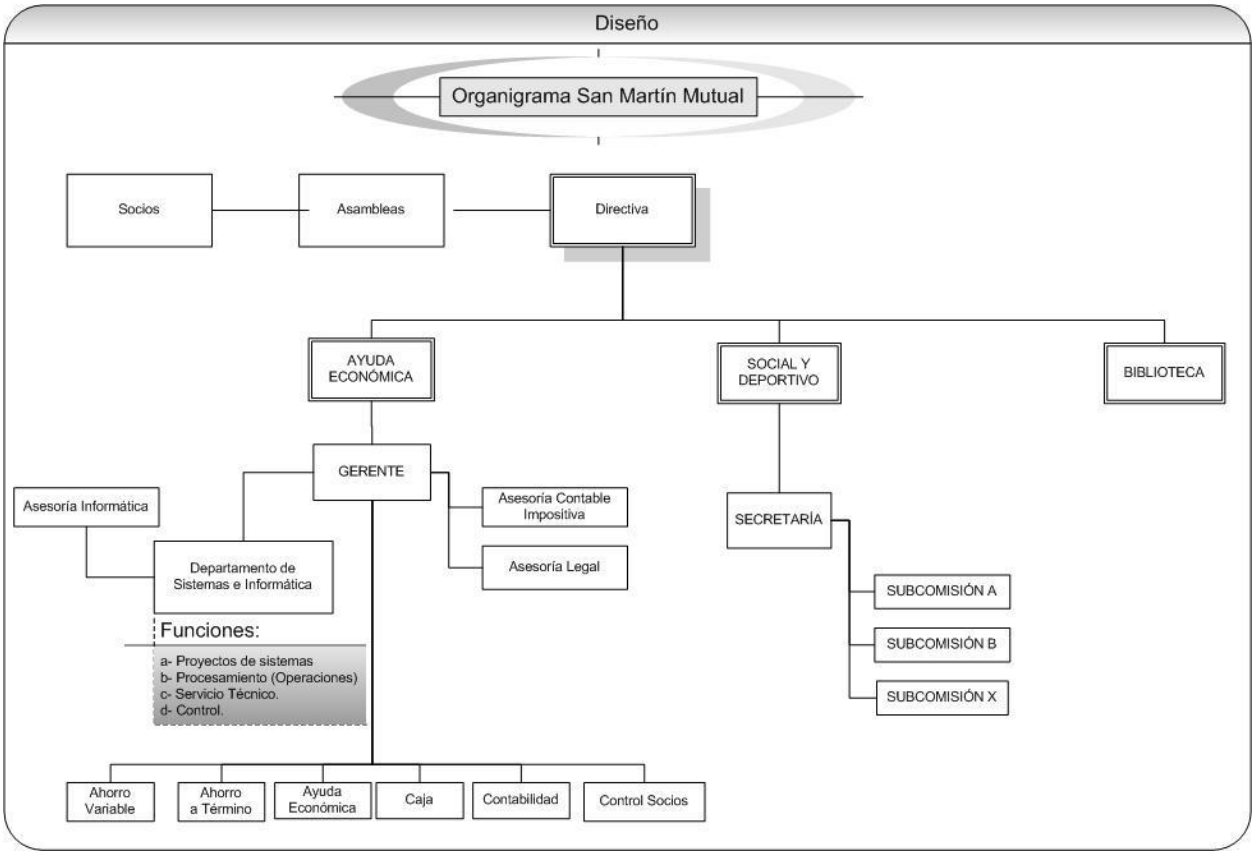


**Fig. 1.1: Organigrama San Martín Mutual, Social y Biblioteca**

# 2.0 – SITUACIÓN PROBLEMÁTICA

En la actualidad muchos sistemas que fueron desarrollados sin aplicar ninguna práctica de ingeniería de software, siguen forzosamente operativos debido a que son vitales para el funcionamiento de las organizaciones. Una solución a este problema es la reingeniería de software. Pretende extraer de esos sistemas obsoletos conocimientos que permitan crear un nuevo sistema fiable, eficiente y de fácil mantenimiento, que los reemplace sin dificultad.

Los sistemas instalados son muy importantes y significan horas de desarrollo y programación. El criterio es resguardar la inversión y mudarla en forma ordenada y con el menor impacto negativo, tanto operativo como económico. Se debe transformar los sistemas actuales en sistemas distribuidos, que sean operables desde distintos puntos físicamente separados. Para ello se debe utilizar como medio unificador la estructura de internet (banda ancha, es decir 1Mbit/s o mayores velocidades).

Una de las tareas a realizar, en la organización en estudio, consiste en conectar funcionalmente el sistema financiero (la mutual) con la sede central del club (organización). En este caso las instalaciones deportivas y sociales, están a una distancia aproximada de 500 metros de la mutual. Debo además prever futuras sucursales financieras y sociales que puedan llegar a estar en otras localidades vecinas (ubicadas a distancias mayores a 20 km), y todo esto debe conectarse e interactuar con el sitio web de la institución. Estos puntos pueden manejar sistemas que utilicen en su implementación lenguajes distintos a los de la sede central pero deben usar los servicios web dados por dicha sede central.

El primer problema a resolver es el medio físico de unión entre los puntos. La forma con más consenso es usar los servicios de internet como vehículo de transporte. Pero en el caso de la sede social se puede optar por un enlace de radio punto a punto a 5MHz con las medidas de seguridad adecuadas.

La Figura muestra visualmente el despliegue.

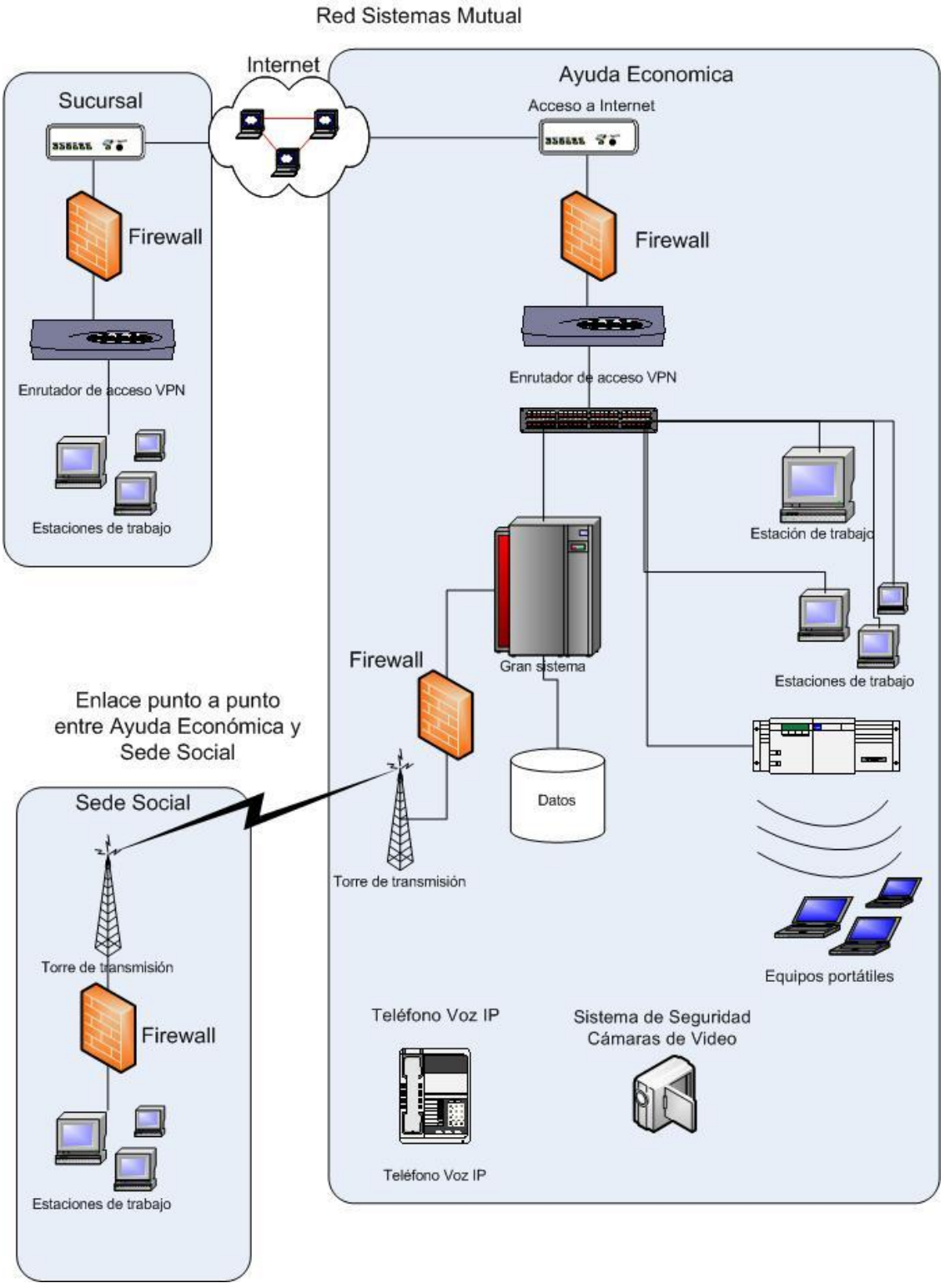


**Fig. 2.1: Despliegue Red San Martín Mutual**

Los sistemas que actualmente están en funcionamiento, cubren un cierto grado de funcionalidades, cosa que se debe mantener e incrementar, pero además se debe ser cuidar

el proceso de transformación y dejar abierta la posibilidad de incorporar nuevas disciplinas deportivas y servicios financieros.

El problema aquí planteado es tomar una decisión correcta en cuanto a una arquitectura determinada, a las tecnologías asociadas y a las herramientas de construcción, sin dejar de lado los principios básicos de un buen software: fácilidad de mantenimiento, alto grado de confiabilidad (incluyendo la fiabilidad, protección y seguridad), eficiencia y facilidad de uso.

## 2.1 – Problema

La institución tiene un sistema informático que está operativo, es confiable, es conocido y cubre una parte significativa de sus necesidades funcionales. El problema es que este sistema no permite distribuir geográficamente los puntos de acceso a la información y dificulta o impide la relación entre los subsistemas propios y de terceros.

El objetivo es encontrar un patrón arquitectónico que, además de resolver el problema actual, permita dar solución a futuras necesidades.

## 2.2 – Antecedentes del Problema

**San Martín Mutual, Social y Biblioteca** comenzó como **Club Atlético San Martín** el 6 de junio de 1909. Su transformación a mutual en 1986 trajo como consecuencia nuevas necesidades, entre las cuales era prioritaria la automatización de los procesos de gestión. Eso planteó un escenario de informatización creciente.

No existía una estrategia global y a largo plazo. Los subsistemas se fueron desarrollando para cubrir las necesidades administrativas de los socios y las distintas disciplinas deportivas y luego los requerimientos de la actividad financiera y la gestión contable. No tenían ninguna conexión entre sí.

En un primer momento (desde 1987 hasta 1990) los trabajos estaban tercerizados, con los inconvenientes que esto conlleva, entre ellos dos muy importantes: la falta de inmediatez y los problemas de seguridad de los datos.

Considerando estas y otras cuestiones se incorporó hardware propio, con las limitaciones económicas que en ese momento se tenían. Los equipos eran clones de PC con Intel 286, que tenían sistema operativo Xenix.

Xenix fue una derivación del sistema Unix, desarrollado por Santa Cruz Operation. Como en ese momento del desarrollo informático de nuestro país, comenzaba a irrumpir Microsoft con su sistema operativo MS-DOS en máquinas PC, una de las opciones de programación era el Fox, que permitía su desarrollo en ambos sistemas operativos.

Fox era una implementación con mayor potencia que el famoso DBase. Fox también competía con Clipper. Una de las ventajas de Fox era que tenía su propia base de datos nativa. La versión para Xenix (Unix) se llamaba Foxplus. Muy práctico era intercambiar desarrollos de uno a otro sistema operativo.

Con el paso del tiempo Xenix perdió peso en el mercado de manera que la distribución en distintos puestos de trabajo se tuvo que resolver a través de máquinas PC con sistema operativo MS-DOS y para unirlas se utilizó el software Lantastic. Como medio físico la conexión era a través de cable coaxil, con conectores BNC.

Un importante paso fue la aparición de Windows 95 con su entorno gráfico, que aportó mejor desempeño. Como con este SO se podían implementar redes, esa fue la solución de conexión entre PC.

El IDE de desarrollo que mantenía compatibilidad con los anteriores se mudó a Visual FoxPro que era Fox con un entorno gráfico. Aquí el desarrollo de sistemas tuvo un gran punto de inflexión, las pantallas solo texto se reconvirtieron en diseños visuales guiados por eventos. Pero siempre manteniendo la estructura cliente/servidor donde una de las PC hacía de server.

El próximo paso, marcado por la compra de Visual FoxPro por Microsoft fue incorporar como sistema operativo Windows 98 y luego Millenium. El cambio notable fue la incorporación de Windows Server 2000 como servidor principal.

En 2004 se incorporó Windows XP en los clientes y Windows Server 2003. Distintas opciones se fueron agregando al software de manera de cubrir las necesidades que la gestión financiera requería.

Los datos fueron mudados desde las bases nativas de VFP a MSSQL 2005 pues el volumen de datos ameritaba una herramienta con mayor potencia. De esta forma se desarrolló toda la interfaz de comunicación con SQL para su utilización.

En una etapa última previa a este trabajo, los sistemas habían alcanzado la versión final de VFP9SP2. Esto es debido a que Microsoft dio por terminada la evolución de VFP como software de desarrollo, y dejó de actualizarlo.

Se ve así cómo el esquema de trabajo se limita a un solo punto geográfico y está desarrollado para interactuar desde una LAN sin poder extenderse geográficamente.

Llegado a este punto es que se plantea su adecuación a un sistema distribuido y así arranca la investigación desarrollada en este trabajo.

# 3.0 – OBJETIVOS Y ALCANCES

- Establecer las ventajas y debilidades de la estructura arquitectónica actual.
- Desarrollar una arquitectura que resuelva los problemas actuales y facilite futuras actualizaciones tecnológicas.
- Realizar la reingeniería de los sistemas de manera que manteniendo las funcionalidades básicas se pueda avanzar a una modernización de ellos con capacidad para interconectarse entre propios y terceros.
- Adaptar un sistema que inicialmente se usaba en varias PC fijas con una separación física de pocos metros, a su utilización desde cualquier punto donde exista internet o señal WiFi que pueda ser captada desde SmartPHone, Tablet, NoteBook, etc. o simplemente una PC de escritorio.
- Desarrollar las interfaces de usuario de modo que cualquier explorador web sea suficiente para ejecutar el aplicativo, haciéndolo independiente de los sistemas operativos.
- Lograr que los cambios implementados causen el menor efecto negativo posible, sobre los usuarios finales.
- Ampliar la base de funcionalidades y prestaciones actuales de acuerdo al constante incremento de las exigencias a que está sometida la mutual desde los organismos de control.
- Extraer las enseñanzas de este trabajo, y documentarlas para su aplicación futura.

# 4.0 – DELIMITACIÓN DEL PROYECTO

El constante desarrollo tecnológico en el campo informático hace que resulte poco útil en una investigación de estas características definir el hardware utilizado. De manera que este trabajo no establecerá una configuración exacta de las PC y su red de conexiones.

Los límites de la aplicación de los sistemas a desarrollar estarán dados por las restricciones que afectan la disponibilidad de internet, condicionada principalmente por el ancho de banda de los enlaces.

En la documentación del proyecto solo se incluirán las historias de usuarios, que en la terminología de UML se llaman casos de uso.

No se pretende en este trabajo profundizar el tema de seguridad informática. En cuanto a la utilización por parte de los usuarios, no ahondaré en procesos de identificación y niveles de permisos para ejecutar tal o cual ítem del sistema. En cuanto a la transmisión de la información, no detallaré lo atinente a los temas de seguridad que se establecen para los protocolos de comunicación utilizados (http, tcp/ip, etc.).

En este trabajo no estableceré el software de base para las máquinas clientes, dada la independencia del sistema operativo; solo estableceré la necesidad de utilizar un browser de última versión para aprovechar todas las funcionalidades disponibles.

# 5.0 – APORTES DEL PROYECTO

- Aporta una forma de resolver los problemas de actualización de sistemas y hacer reingeniería dentro de un contexto de informatización creciente y cambiante.
- El punto más importante es profundizar el análisis del modelo de datos y del proceso de negocio hasta lograr que la base de datos pueda ser leída desde cualquier arquitectura. Una vez logrado esto se pueden implementar los servicios de infraestructura. Esto puede ser tomado como modelo de una forma suave de cambios para los programadores de VFP y VB, que persisten en ese tipo de programación.
- La investigación llevada adelante para este trabajo permitió adquirir los conocimientos necesarios para adoptar como arquitectura base el patrón MVC.
- Sirve como guía para organizar nuevos proyectos similares y generar modelos para la toma de decisiones. Alienta la reutilización. Adquiere relevancia económica y de gestión, pues su éxito traerá muchas ventajas tanto en las estructuras actuales como futuras de la institución mutual. Los beneficiarios directos serán los operadores de los sistemas, como así también los clientes de la entidad.
- Logra una mayor integración entre los distintos sistemas aunque los idiomas (lenguajes de programación) sean distintos.

# 6.0 - MARCO TEÓRICO

Consideré importante iniciar el trabajo con un estudio del aspecto teórico; conocer a fondo el tema me daría bases firmes para la toma de decisiones desde el punto de vista práctico.

Mi intención es mostrar a través de este capítulo las ideas generales sobre sistemas informáticos, su desarrollo y aplicación, así como la interrelación de disciplinas que significa la ingeniería de sistemas. Las personas como partícipes fundamentales de estas tareas, son tema prioritario. También se detallan los efectos del paso del tiempo en esta área de la tecnología y la respuesta que da la ingeniería, para encontrar soluciones a este problema a través de la reingeniería, que propone cambiar a nuevos procedimientos y formas, los que en este momento están en plena aplicación.

## 6.1 - Antecedentes Teóricos

### 6.1.1 Sistemas:

Como introducción al tema considero conveniente comenzar hablando de sistemas. El término sistemas es universalmente usado, sin embargo en el ámbito de este trabajo está referido a aquellos que incluyen computadoras, ayudan a la comunicación y a la información. Así siguiendo a Sommerville se lo puede definir diciendo que:

*"Un sistema es una colección de componentes interrelacionados que trabajan conjuntamente para cumplir algún objetivo"*. **(1)**

Esta definición es general e incluye un amplio espectro de sistemas. Si se restringe el ámbito de aplicación de la palabra sistemas únicamente a aquellos que incluyen software, se los puede dividir en dos categorías:

- *Sistemas técnicos informáticos*: incluyen componentes de hardware y software pero no procesos ni procedimientos. Se usan con un fin determinado, como una herramienta, pero el conocimiento de ese fin no pertenece al sistema. Ejemplo: procesadores de texto, clientes de correo, etc.
- *Sistemas socio-técnicos*: comprenden uno o más sistemas técnicos, incluyendo el conocimiento de cuál es el objetivo a alcanzar. Significa que se han definido los

procesos operativos, incluyendo las personas como parte del sistema. Ejemplo: la fabricación de un auto, etc.

Dentro de esta categoría, resultan de interés aquellos sistemas socio-técnicos que incluyen hardware, software y personas.

Los profesionales (en este caso los ingenieros en sistemas) tienen responsabilidades y no sólo respecto a las obligaciones de la profesión, sino también respecto a la sociedad que integran.

### 6.1.2 Ingeniería de Sistemas

La ingeniería de sistemas es la actividad que tiene por objeto especificar, diseñar, implementar, validar, utilizar y mantener los sistemas socio-técnicos. Los ingenieros de sistemas no solo tratan con el software, sino también con el hardware y las interacciones del sistema con los usuarios y el contexto. Por eso las distintas definiciones de sistemas informáticos varían, principalmente, por ubicar al usuario o las personas dentro o fuera de ellos. Aunque parece simple es muy importante definir a la hora de considerar una equivocación humana, si a la misma se la considera como un error del sistema o no.

Existen diferencias entre **la ingeniería de sistemas y el desarrollo de software**, fundamentalmente en dos aspectos:

1 – Implicancias interdisciplinarias: La ingeniería de sistemas conjuga muchas disciplinas con equipos de personas de diferentes conocimientos, terminologías y convenciones. Lo que lleva a que no siempre la mejor solución desde el punto de vista técnico lo sea también desde un punto de vista del sistema en su totalidad.

2 – Capacidad de adaptación y corrección: Una vez que se tomó una decisión en un sistema cuesta mucho cambiarla. Cosa que sí se puede lograr en materia de software que permite cambios menos costosos cuando varían los requerimientos.

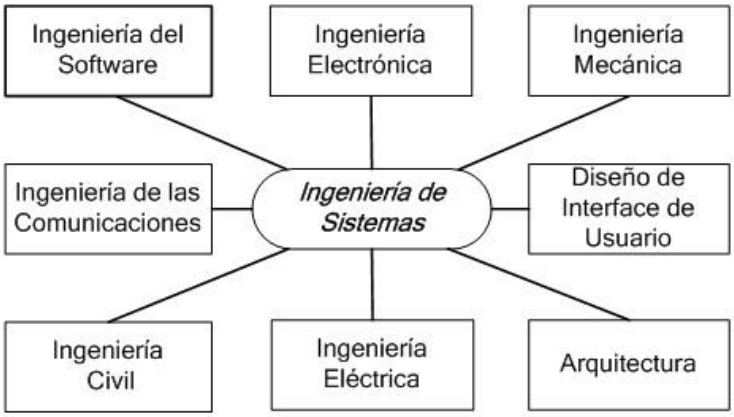


**Fig. 6.1: Disciplinas relacionadas con la ingeniería de sistemas**

### 6.1.3 Requerimientos de los sistemas

Se refiere a qué es lo que el sistema debe hacer:

1 – Requerimientos funcionales: las funciones básicas que el sistema debe proporcionar se definen en un primer nivel abstracto, dejando los detalles para el subnivel de componentes del sistema.

2 – Propiedades del sistema: son las propiedades emergentes no funcionales del sistema, tales como disponibilidad, rendimientos, seguridad, etc.

3 – Características que no debe mostrar: a veces es tan importante especificar lo que el sistema no debe hacer, como especificar lo que debe hacer.

### 6.1.4 Diseño de los sistemas

El diseño del sistema se centra en proporcionar las funcionalidades del sistema a través de sus componentes. Las actividades que se realizan son:

1 – Dividir requerimientos.

2 – Identificar subsistemas.

3 – Asignar requerimientos a los subsistemas.

4 – Especificar la funcionalidad de los subsistemas.

5 – Definir las interfaces de los subsistemas.

Para muchos sistemas existen varios diseños posibles que cumplen con los requerimientos. Estos comprenden una amplia gama de soluciones que combinan hardware, software y operaciones humanas. Las decisiones técnicas que satisfacen los

requerimientos deben adecuarse a los criterios de la organización y al marco jurídico vigente.

### 6.1.5 Modelado de los sistemas

Durante la actividad de requerimientos y diseño de los sistemas, éstos se pueden modelar como un conjunto de componentes y sus relaciones. Esto puede ser llevado a formas gráficas en un modelo arquitectónico de los sistemas, que permite dar una visión general de la estructura y desarrollo. La utilización de técnicas de modelización es útil en la medida que pueda separar entre lo *que hace* un objeto y el *cómo lo hace*. Esto permite en todo momento, prever futuras mejoras, no solo a partir de los requerimientos, sino a medida que vayan apareciendo nuevas necesidades, generadas fundamentalmente para logar mayor eficiencia en los sistemas.

La arquitectura de sistemas se representa como diagramas de bloques, y su interrelación. Los diagramas de bloques se utilizan para sistemas de cualquier tamaño, apoyados en la correcta división de subsistemas. Siempre el modelo arquitectónico de sistemas fue utilizado para identificar componentes de hardware y software, dando funcionalidades a cada componente. Esto depende de factores técnicos específicos y no técnicos.

### 6.1.6 Desarrollo e integración de los sistemas

El desarrollo de los sistemas significa la aplicación en forma práctica de lo especificado en el diseño del mismo.

El desarrollo de un sistema, implica normalmente recurrir a distintos subsistemas que forman parte de un todo. Una vez desarrollados los subsistemas, de forma independiente se procede a llevar adelante el proceso de integración del sistema. Este proceso de integración puede hacerse de todos los subsistemas al mismo tiempo o uno a uno, lo cual es mejor ya que facilita la localización de los errores. Una vez lograda la integración, se pasa a la prueba del sistema y a las tareas tendientes a solucionar los problemas que van surgiendo, lo cual puede llevar semanas o meses.

### 6.1.7 Implementación de los sistemas

La implementación, puesta en marcha y uso de los sistemas están influenciados por las políticas y procedimientos de la organización y por la cultura del trabajo vigente. Los usuarios del sistema son personas que forman parte de la gestión en curso y se relacionan con personas que están dentro y fuera de la organización.

Al llevarse a cabo la tarea de implementación de un sistema, muchas veces aparecen factores humanos y organizacionales no tenidos en cuenta al comienzo y que generan una revisión del diseño del sistema. Será necesario tenerlos en cuenta y tomar medidas correctivas; en caso contrario se podría sufrir el rechazo de usuarios y directivos que se relacionan con el sistema.

Estos factores son muy difíciles de predecir a través de la tarea de ingeniería.

### 6.1.8 Evolución de los sistemas

Durante la vida útil de un sistema importante se producen modificaciones para corregir errores en los requerimientos originales y para implementar nuevos requerimientos que surgen. Se suma la evolución tecnológica y la mayor capacidad de procesamiento de los equipos. Esta evolución trae inconvenientes que deben ser solucionados teniendo en cuenta entre otros, los siguientes criterios:

- Los cambios deben responder a cuestiones técnicas y de negocio, contribuyendo al objetivo del sistema.
- Cuando se modifica un subsistema, si el acoplamiento no es bajo se pueden tener que modificar varios componentes relacionados.
- Cuando se perdió el razonamiento original se deben tomar decisiones de diseño en el contexto actual.
- Con el paso del tiempo las estructuras sufren un proceso de obsolescencia lógico que multiplica los costos del cambio.
- Los sistemas que se han desarrollado, con el tiempo pasan a depender de tecnologías hardware y software que fueron quedando obsoletas. Si estos sistemas tienen un papel fundamental dentro de la organización, son conocidos como

sistemas heredados. A la organización le gustaría reemplazar los sistemas heredados, pero introducir un sistema nuevo trae aparejado un riesgo muy alto.

## 6.2 - El software. Su evolución

El origen del desarrollo del software comienza en 1948, cuando corrió el primer programa. Se trata por tanto de una disciplina que recién comienza frente a otras que son centenarias. En un primer momento, con pocos conocimientos, se empiezan a desarrollar las primeras aplicaciones, que dan respuestas a nuevas necesidades; aquí se hablaba de desarrollo de software y no de ingeniería de software, término que comienza a utilizarse a partir de 1968, fecha en la cual también se comienza a hablar de crisis de software para referirse a la falta de inmediatez y precisión para poder coordinar el desarrollo de complejos sistemas que requerían el trabajo en conjunto de equipos y disciplinas diferentes en la construcción de sistemas únicos.

Esta crisis fue el resultado de la introducción de la tercera generación del hardware. El hardware dejó de ser un impedimento para el desarrollo de la informática; redujo los costos y mejoró la calidad y eficiencia en el software producido. La crisis se caracterizó por los siguientes problemas:

- Imprecisión en la planificación del proyecto y estimación de los costos.
- Baja calidad del software.
- Dificultad de mantenimiento de programas con un diseño poco estructurado, etc.

A raíz de esto se desarrollaron tres áreas de conocimiento que se revelaron como estratégicas para hacer frente a la crisis del software:

- **Ingeniería del software**: este término fue usado para definir la necesidad de contar con una disciplina científica que permita aplicar un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software.
- **Gestión predictiva de proyectos**: es una disciplina formal de gestión, basada en la planificación, ejecución y seguimiento a través de procesos sistemáticos y repetibles.

- **Producción basada en procesos**: se crean modelos de procesos basados en la producción industrial. La calidad del resultado depende básicamente de la calidad de los procesos.

De las tres áreas de conocimiento arriba mencionadas, por un lado, la gestión predictiva de proyectos establece como criterios de éxito obtener el producto definido en el tiempo previsto y con el costo estimado. Para ello, se asume que el proyecto se desarrolla en un entorno estable y predecible. Por otro lado, la producción basada en procesos emula modelos industriales e ingenieriles que surgieron en otros ámbitos y con otros desencadenantes. Lo cierto es que, ni los productos de software se pueden definir por completo a priori, ni son totalmente predecibles, ni son inmutables. Además, los procesos aplicados a la producción industrial no tienen el mismo efecto que los aplicados al desarrollo de software, ya que en un caso se aplican sobre máquinas y en otro, sobre personas.

La ingeniería de software, dio una respuesta más adecuada a la crisis al tener en cuenta el tiempo de vida del producto software. Antes, la vida útil de un producto acabado era muy larga; durante este tiempo, generaba beneficios a las empresas, para las que era más rentable este producto que las posibles novedades pero, a partir de los ochenta, esta situación empieza a cambiar. La vida de los productos es cada vez más corta y una vez en el mercado, son novedad apenas unos meses, quedando fuera de él enseguida. Esto obliga a cambiar la filosofía de las empresas, que se deben adaptar a este cambio constante y basar su sistema de producción en la capacidad de ofrecer novedades en forma permanente.

Fue así que dentro de la ingeniería de software comienzan a aparecer modelos de procesos para el desarrollo de software.

### 6.2.1 Modelos de procesos de software

Un modelo de procesos de software es una descripción simplificada de un proceso de software que presenta una visión de ese proceso. Estos modelos pueden incluir actividades que son parte de los procesos y productos de software y el papel de las personas involucradas en la ingeniería del software. El uso de un proceso inadecuado del software puede reducir la calidad o la utilidad del mismo y/o incrementar sus costos. La mayor parte de los modelos de procesos de software se basan en uno de los tres modelos generales o paradigmas de desarrollo de software:

**1. El enfoque en cascada**. Considera las distintas actividades necesarias para el desarrollo de software como fases separadas, tales como la especificación de requerimientos, el diseño de software, la implementación, las pruebas, etcétera. Después de que cada etapa queda definida se continúa con el desarrollo de la siguiente etapa.

**2. Desarrollo interactivo**. Este enfoque entrelaza las actividades de especificación, desarrollo y validación. Un sistema inicial se desarrolla rápidamente a partir de especificaciones muy abstractas. Éste se perfecciona basándose en las solicitudes del cliente para producir un sistema que satisfaga las necesidades de dicho cliente. El sistema puede entonces ser entregado. De forma alternativa, se puede reimplementar utilizando un enfoque más estructurado para producir un sistema más sólido y de fácil mantenimiento.

**3. Ingeniería de software basada en componentes:** Esta técnica supone que las partes del sistema existen. El proceso de desarrollo del sistema se enfoca en la integración de estas partes, más que en desarrollarlas desde el principio.

Se pasa a considerar en detalle cada uno de los dos primeros modelos ya que el tercero supone la existencia de sistemas que se preservarán y se deben unir a otros.

### 6.2.2 Modelo en cascada

Este es el más básico de todos los modelos y ha servido como bloque de construcción para los demás paradigmas de ciclo de vida. Está basado en el ciclo convencional de una ingeniería y su visión es muy simple: el desarrollo de software se debe realizar siguiendo una secuencia de fases.

Cada etapa tiene un conjunto de metas bien definidas y las actividades dentro de cada una contribuyen a la satisfacción de metas de esa fase o quizás a una subsecuencia de metas de la misma. El arquetipo del ciclo de vida abarca las siguientes actividades:

**Ingeniería y análisis del sistema**: debido a que el software es siempre parte de un sistema mayor, el trabajo comienza estableciendo los requisitos de todos los elementos del sistema y luego asignando algún subconjunto de estos requisitos al software.

**Análisis de los requisitos del software:** la ingeniería de software debe centrarse en comprender y resolver las funcionalidades, el correcto desempeño y las interfaces requeridas en su ámbito.

**Diseño:** el proceso de diseño traduce los requisitos en una representación del software con la calidad requerida antes de que comience la codificación.

**Codificación:** el diseño debe escribirse de una manera interpretable para la máquina. Si el diseño se realiza de una manera adecuada, la codificación puede realizarse automáticamente.

**Prueba:** luego de llevar a código el diseño, comienzan las pruebas del programa que aseguren que una entrada definida, produce los resultados requeridos.

**Mantenimiento:** el software sufrirá cambios después de que se entrega al cliente. Los cambios ocurrirán porque: se han encontrado errores, el software debe adaptarse a cambios del entorno externo (sistema operativo o dispositivos periféricos) o el cliente requiere ampliaciones funcionales o de rendimiento.

En el modelo de cascada se ve una ventaja evidente y radica en su sencillez, ya que sigue los pasos intuitivos necesarios a la hora de desarrollar el software. Pero el modelo se aplica en un contexto, así que se debe atender también a él y saber que:

- Los proyectos reales raramente siguen el flujo secuencial que propone el modelo. Siempre hay iteraciones y se crean problemas en la aplicación del paradigma.
- Normalmente, al principio, es difícil para el cliente establecer todos los requisitos explícitamente. Sin embargo este ciclo de vida clásico así lo requiere y tiene dificultades en acomodar posibles incertidumbres que pueden existir al comienzo de muchos productos.

- El cliente debe tener paciencia. Hasta llegar a las etapas finales del proyecto no estará disponible una versión operativa del programa. Un error importante que no sea detectado hasta que el programa esté funcionando, puede ser desastroso.

### 6.2.3 Modelos interactivos

**Metodologías ágiles**

El agilismo es una respuesta a los fracasos y las frustraciones del modelo en cascada. Aplicar prácticas de estimación, planificación e ingeniería de requisitos, no son malas en sí mismas ni su método de aplicación es incorrecto, lo que sucede es que no es una práctica adecuada.

En 2001, 17 representantes de nuevas metodologías y críticos de los modelos de mejora basados en procesos se reunieron, convocados por Kent Beck, para discutir sobre el desarrollo de software. De allí nació el manifiesto ágil que se compone de cuatro principios. Es pequeño pero cargado de significado:

"Estamos descubriendo mejores maneras de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros.

A través de esta experiencia hemos aprendido a valorar:

- **Individuos e interacciones** sobre *procesos y herramientas.*
- **Software que funciona** sobre *documentación exhaustiva.*
- **Colaboración con el cliente** sobre *negociación de contratos.*
- **Responder ante el cambio** sobre *seguimiento de un plan.*

Esto es, aunque los elementos a la derecha tienen valor, nosotros valoramos por encima de ellos los que están a la izquierda."

*Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas.*

Tras este manifiesto se encuentran 12 principios de vital importancia para entender su filosofía:

- La máxima prioridad es satisfacer al cliente a través de entregas tempranas y continuas de software valioso.
- Los requisitos cambiantes son bienvenidos, incluso en las etapas finales del desarrollo. Los procesos ágiles aprovechan el cambio para ofrecer una ventaja competitiva al cliente.
- Se entrega software que funciona frecuentemente, entre un par de semanas y un par de meses. De hecho es común entregar cada tres o cuatro semanas.
- Las personas del negocio y los desarrolladores deben trabajar juntos diariamente a lo largo de todo el proyecto.
- Se construyen proyectos en torno a individuos motivados, dándoles el lugar y el apoyo que necesitan y confiando en ellos para hacer el trabajo.
- El método más eficiente y efectivo de comunicar la información hacia y entre un equipo de desarrollo es la conversación cara a cara.
- La principal medida de avance es el software que funciona.
- Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deben poder mantener un ritmo constante.
- La atención continua a la excelencia técnica y el buen diseño mejoran la agilidad.
- La simplicidad es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de la auto organización de los equipos.
- A intervalos regulares, el equipo reflexiona sobre cómo ser más eficaces, a continuación mejoran y ajustan su comportamiento en consecuencia.

Las metodologías ágiles se basan en que el software es propenso a errores por naturaleza de sus creadores, y les hacen tomar medidas para minimizar sus efectos negativos desde el principio. Reconoce que el ser humano no es perfecto y se equivoca con

frecuencia, entonces propone técnicas que aportan confianza a pesar de ello. La automatización de procesos es uno de sus pilares. Al igual que en el modelo tradicional, existen fases de análisis, desarrollo y prueba, pero en lugar de estar en forma consecutiva (cascada) están solapadas. Esta combinación de etapas que se ejecutan repetidas veces se denominan iteraciones. El modelo de cascada se transforma en varios ciclos (iteraciones) que se alimentan con nuevos requerimientos, perfeccionando los temas abordados en cada iteración, adecuando los detalles técnicos. Y en cada una de ellas se habla con el *cliente/usuario* para analizar requerimientos. Se escriben pruebas automatizadas, se escriben líneas de código nuevas y se mejora el código existente. Al cliente se lo hace participar de los resultados después de cada iteración para comprobar su aceptación y corregir los detalles que se estimen oportunos o incluso modificar la planificación.

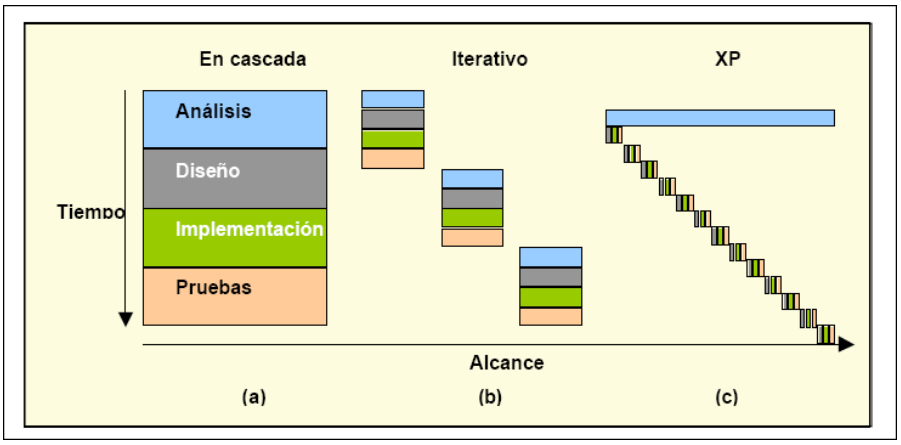


**Fig. 6.2: Alcance de las Iteraciones**

## 6.3 - Reingeniería de los sistemas. Reingeniería del software

Con el paso del tiempo los sistemas de software se ven afectados y para seguir siendo útiles inevitablemente deben sufrir modificaciones. Estos cambios pueden ser realizados dentro de un mantenimiento de rutina o dentro de una adecuación más profunda que implica un proceso de reingeniería. Esto permite colocar al software a tono con los avances del hardware y poder así aprovechar al máximo su potencial.

### 6.3.1 Sistemas de software heredados

Se denomina sistemas heredados a los sistemas socio-técnicos informáticos que se desarrollaron con herramientas y criterios anteriores y que debido a la evolución

tecnológica sufren irremediablemente el proceso de obsolescencia. Esto incluye software, hardware, procesos y procedimientos.

Los sistemas heredados son a menudo sistemas de negocio crítico, que se mantienen porque es demasiado arriesgado o costoso reemplazarlos. Si en una organización, su sistema financiero o contable, del cual dependen otros procedimientos, es reemplazado con un resultado negativo en su funcionalidad, esto traería consecuencias difíciles de evaluar. También afectaría a los integrantes de la organización y a los clientes causando un deterioro a nivel organizativo.

### 6.3.2 Mantenimiento y reingeniería

La reingeniería y la fase de mantenimiento del ciclo de vida de un sistema están muy relacionados. El mantenimiento se llevará a cabo una vez que la implementación del sistema software haya concluido y se haya entregado al cliente, y puede estar determinado por nuevas necesidades del usuario o bien por la detección de errores.

Según ANSI-IEEE, el mantenimiento del software es la modificación de un producto software, después de su entrega al cliente, para corregir defectos, para mejorar el rendimiento u otras propiedades deseables, o para adaptarlo a un cambio de entorno. Por lo tanto, el mantenimiento del software se puede definir como el conjunto de medidas que hay que tomar para que el sistema siga trabajando correctamente. Existen 4 tipos de mantenimiento:

- **Correctivo**: el cual tiene como objetivo localizar y eliminar los posibles defectos de los programas.
- **Adaptativo:** cuyo objetivo es modificar un programa para adaptarlo a los cambios hardware o software en el entorno en el que se ejecuta. Puede ser, desde un pequeño cambio, hasta una reescritura de todo el código, y es cada vez más habitual debido a la actualización frecuente de los sistemas operativos.
- **Perfectivo**: consistente en el conjunto de actividades para mejorar o añadir nuevas funcionalidades requeridas por el cliente.
- **Preventivo**: que consiste en la modificación del software para mejorarlo en cuanto a la calidad y al mantenimiento, sin alterar sus especificaciones funcionales.

Los recursos invertidos en mantenimiento se incrementan a medida que se genera más software. Esto se debe a la gran cantidad de software antiguo cuya creación se produjo con restricciones de tamaño y espacio de almacenamiento y con herramientas desfasadas, a las migraciones continuas de plataformas o sistemas operativos, a las modificaciones, correcciones, mejoras y adaptaciones que el software experimenta con el tiempo y a las nuevas necesidades de los usuarios. Además, estos cambios, generalmente se han realizado sin técnicas de reingeniería o ingeniería inversa, dando como resultado sistemas con las estructuras de datos mal diseñadas, mal codificadas, con lógica defectuosa y con escasa documentación.

Existen diferentes dificultades a la hora de realizar el mantenimiento. La primera, y más importante, es que la mayor parte del software está formado por código antiguo heredado, desarrollado hace tiempo con técnicas y herramientas en desuso, y ha sufrido varias actividades de mantenimiento. Después de estos cambios, los programas son menos estructurados, lo que provoca un incremento en el tiempo de compresión de los mismos y una documentación desfasada. Los sistemas mantenidos son cada vez más difíciles de cambiar. Debido a estos problemas, se producen una serie de efectos secundarios sobre el código, los datos y la documentación, como por ejemplo: modificación o eliminación de subprogramas, etiquetas o identificadores; modificación de los formatos de registros o archivos; modificación de definición de variables globales; nuevos mensajes de error no documentados o modificación de las interfaces de usuario.

Las soluciones al problema del mantenimiento se pueden dividir en dos grupos: soluciones de gestión y soluciones técnicas (éstas últimas son a su vez de dos tipos: herramientas y métodos). Las herramientas sirven para soportar de forma efectiva los métodos, y se refieren a la tecnología usada. Y los principales métodos son:

- **Reingeniería**: examen y modificación del sistema para reconstruirlo en una nueva forma.
- **Ingeniería inversa**: análisis de un sistema para identificar sus componentes y las relaciones entre ellos, así como para crear representaciones del sistema en otra forma o en un nivel de abstracción más elevado.

- **Reestructuración de software**: consiste en la modificación del software para hacerlo más fácil de entender y cambiar o menos susceptible de incluir errores en cambios posteriores.

### 6.3.3 Proceso de reingeniería de software

La ventaja fundamental que presenta la reingeniería es**: la reducción del riesgo**, ya que si hay una aplicación que funciona se conocen sus resultados y, por tanto, **ya se dispone de una especificación del sistema.**

No en todos los sistemas es adecuado realizar un proceso de reingeniería. Antes de tomar esa decisión se debe determinar si el sistema tiene un gran valor de negocio y por tanto es conveniente que se aplique reingeniería.

**Evolución de los sistemas heredados**

Las organizaciones que cuentan con un presupuesto limitado para mantener y actualizar sus sistemas heredados tienen que decidir cómo obtener los mejores beneficios de su inversión. Esto significa que tienen que realizar evaluaciones realistas de sus sistemas heredados y a continuación decidir cuál es la estrategia más adecuada para la evolución de estos sistemas.

¿Cómo encarar el problema de la evolución de los sistemas heredados?. La teoría nos muestra cuatro opciones estratégicas que permiten defender la inversión realizada:

1- Desechar completamente el sistema. Esta opción debería elegirse cuando el sistema no constituye una contribución efectiva que justifique su continuidad.

2 - Dejar el sistema sin cambios y continuar con un mantenimiento regular. Esta opción debería elegirse cuando el sistema todavía es necesario y es muy estable y los usuarios del sistema solicitan pocos cambios.

3 - Hacer reingeniería del sistema para mejorar su mantenimiento. Esta opción debería elegirse cuando la calidad del sistema se ha degradado por los cambios continuos y cuando dichos cambios todavía son necesarios. Este proceso puede incluir el desarrollo de nuevos componentes de interfaz para que el sistema original pueda trabajar con otros sistemas más nuevos.

4 - Reemplazar todo o parte del sistema por un nuevo sistema. Esta opción debería elegirse cuando otros factores, como un nuevo hardware, implican que el sistema antiguo no puede continuar en funcionamiento o cuando desarrollar el nuevo sistema tenga un costo razonable.

Naturalmente, estas opciones no son exclusivas, por lo que, pueden aplicarse diferentes opciones a distintas partes del sistema. Cuando se está evaluando un sistema heredado, éste tiene que verse desde una perspectiva de negocio y desde una perspectiva técnica.

- Desde una perspectiva de negocio, tiene que decidirse si el negocio realmente necesita del sistema.
- Desde una perspectiva técnica, tiene que evaluarse la calidad de la aplicación y del software y hardware que dan soporte al sistema.

A continuación, debe utilizarse una combinación del valor del negocio y de la calidad del sistema para decidir qué hacer con el sistema heredado.

### 6.3.4 Fases en la reingeniería del software

Hay varias formas de seguir los procesos de reingeniería. Una forma simple sería definirlo como el proceso de ingeniería inversa seguido de un proceso de ingeniería. Es el proceso de recuperar el diseño del sistema a partir del código fuente para luego volver a aplicar un ciclo de vida de software tradicional. La diferencia entre reingeniería y nuevo desarrollo de software es el punto de partida. El nuevo desarrollo de software comienza con una especificación del sistema e implica el diseño e implementación de un nuevo sistema. La reingeniería comienza con un sistema existente y el proceso de desarrollo para su reemplazo se basa en comprender y transformar el sistema original.

Véase la figura 6.3. El primer paso es comprender el software existente, donde el diseño del sistema se recupera desde su código fuente con actividades como: análisis de dependencias, comprensión del programa, detección, extracción y almacenamiento del diseño. El segundo paso incluye todas las actividades que se realizan para transformar el software existente en un software diferente, más fácil de mantener. Entre ellas están: descomposición, reestructuración, remodularización, redocumentación, etc.

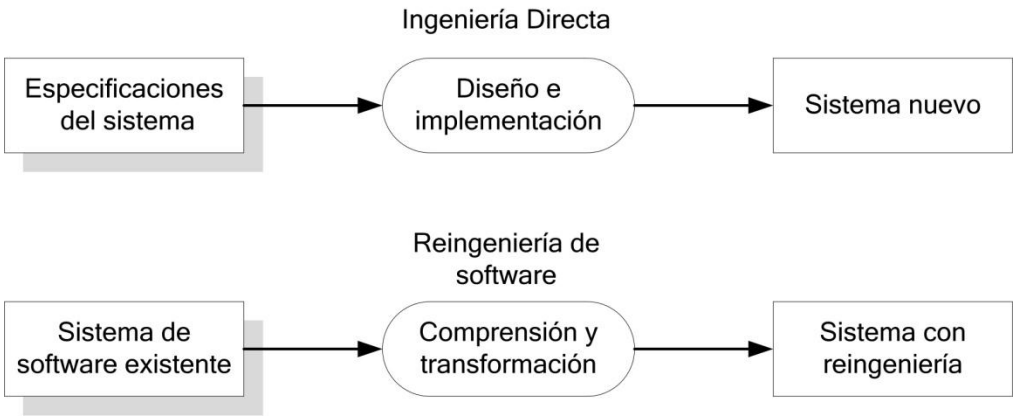


**Fig. 6.3: Proceso básico de Ingeniería Directa y Reingeniería.**

Según Sommerville, **(2)** la reingeniería de software comprende: la redocumentación del sistema, la reorganización y reestructura del sistema, la traducción del sistema a un lenguaje de programación más moderno, la modificación y actualización de la estructura y los valores de los datos del sistema. Un posible proceso de reingeniería, más completo, sería el que se recoge en la figura 6.4.

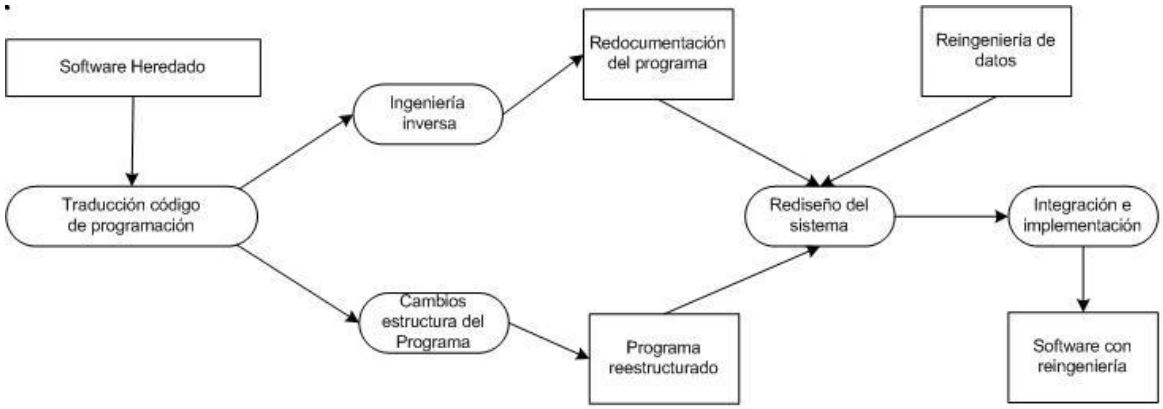


**Fig. 6.4: Proceso de Reingeniería**

En el proceso de reingeniería se podrían distinguir las siguientes fases:

**Conversión del código fuente**: El programa se convierte a una versión más moderna del lenguaje en que estaba codificado o a un lenguaje diferente.

**Ingeniería inversa**: Es el proceso de analizar el software con el objetivo de recuperar su diseño y especificación, partiendo de los resultados hasta llegar al código inicial que lo generó.

**Mejora de la estructura del programa**: Se analiza y modifica la estructura de control del programa para hacerlo más fácil de leer y comprender.

**Modularización del programa**: Es el proceso de reorganizar un programa de manera que partes relacionadas se integren de forma conjunta.

**Reingeniería de datos**: Se trata de analizar y reorganizar las estructuras, e incluso a veces, los valores de los datos de un sistema para hacerlos más comprensibles. Si la funcionalidad del sistema no cambia, la reingeniería de datos no es necesaria.

### 6.3.5 El caso de este proyecto

Las anteriores no son fases que tengan que desarrollarse todas necesariamente, sino que dependiendo de los casos podrán figurar unas u otras.

Se puede desarrollar un modelo de reingeniería del software que sitúe al usuario como colaborador principal en la tarea de especificar los requisitos del sistema. Este sería el caso analizado en este trabajo y me inclino por resumir en estas fases los procedimientos utilizados:

**Definición del problema:** se identifican objetivos, límites, beneficios, riesgos, estimaciones de tiempos, etc., estableciendo una imagen real de lo que existe realmente ahora y de lo que se quiere obtener en el futuro.

**Estudio del código antiguo**: partiendo del código fuente, se obtienen un conjunto de documentos que ayudan a posteriores fases de la metodología. Esto está limitado por los costos materiales que implica su concreción.

**Viabilidad del proyecto**: análisis de los factores que harán posible el proyecto.

**Rediseño de especificaciones**: se busca conseguir, que las especificaciones representen de forma real la visión futura deseada del sistema.

**Estudio de las tecnologías y creación de prototipos**: de aquellas partes que puedan dar problemas, o solamente de aquellas que vayan a cambiar sustancialmente de la original.

**Planificación de la implementación**: consiste en diseñar la forma y modo en que se va a migrar de una herramienta a otra.

**Perfeccionamiento**: realizar cambios en la nueva aplicación que aumenten la calidad del sistema.

# 7.0 – PROCESO DE INVESTIGACIÓN

En este capítulo trato de transcribir en forma teórica y práctica las horas dedicadas a la implementación y prueba de un prototipo de cada uno de los esquemas arquitectónicos aquí planteados y la evolución hacia un esquema final que se ajusta a la tecnología actual.

La tarea arriba descripta no fue fácil ni rápida, requirió de mucho tiempo, esfuerzo y dedicación, en un escenario cambiante y que incorpora nuevas y más poderosas herramientas, en forma permanente y continua. Pero finalmente opté, luego de encontrarla, por una arquitectura que promete ser estable de cara al futuro, principalmente por estar basada en una tecnología muy difundida y exitosa como es la internet y sus protocolos de comunicaciones.

## 7.1 - Sistemas en estudio

La arquitectura existente al inicio de la investigación desarrollada para este trabajo, era la siguiente: los clientes (terminales de operaciones) eran PC con Windows, el server era Windows Server y la base de datos MS SQL, todos unidos por sistemas desarrollados en VFP9. Esto se puede ver en la siguiente figura:

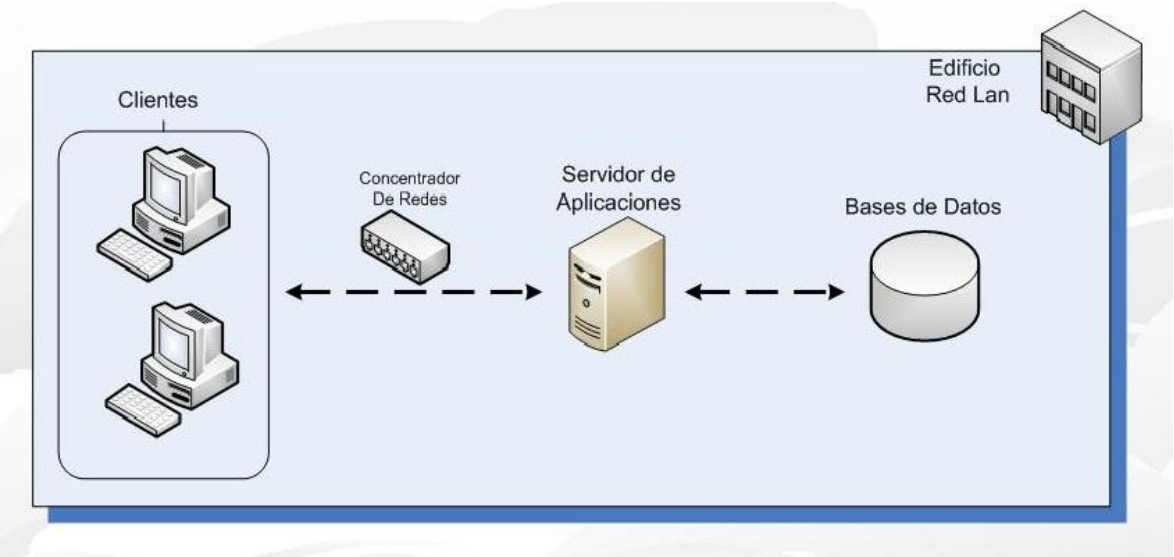


**Fig. 7.1: Arquitectura inicial.**

La idea inicial era defender la inversión existente y hacer un paso gradual hacia nuevas "formas", pensando en el factor humano como usuario de las nuevas tecnologías, dado que se produce en general una actitud de oposición al cambio. Por otra parte siempre

se debe tener en cuenta la necesidad de sacar el mejor beneficio desde el punto de vista del negocio.

La evolución tecnológica que permite comunicar los sistemas y su información sin barreras de distancias ni lenguajes de desarrollo, apuntó a la implementación de SOA (Arquitectura Orientada a Servicio).

En primer lugar se consideró a los web services como piezas claves para esta arquitectura.

Al comenzar el estudio de opciones para este trabajo, se planteó como objetivo a conseguir la siguiente arquitectura, reflejada en este esquema:

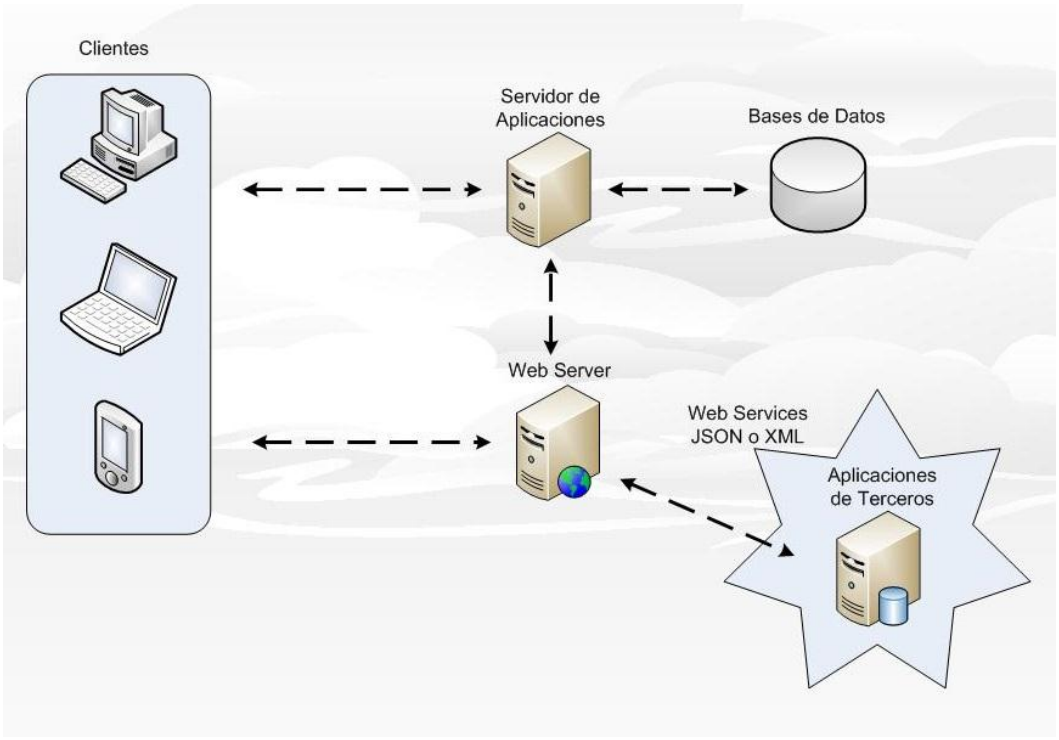


**Fig. 7.2: Arquitectura con servicios web**

Se investigó en varias líneas que en sí mismas formaron un camino, que permitió llegar a la solución propuesta. Se necesitaba dar servicio de HTTP por puerto 80 para sistema web, pero también proveer servicios a otros sistemas en distintos lenguajes e implementaciones.

Y como se ve en la figura siguiente el objetivo era lograr esto con una arquitectura web que luego se explicará:

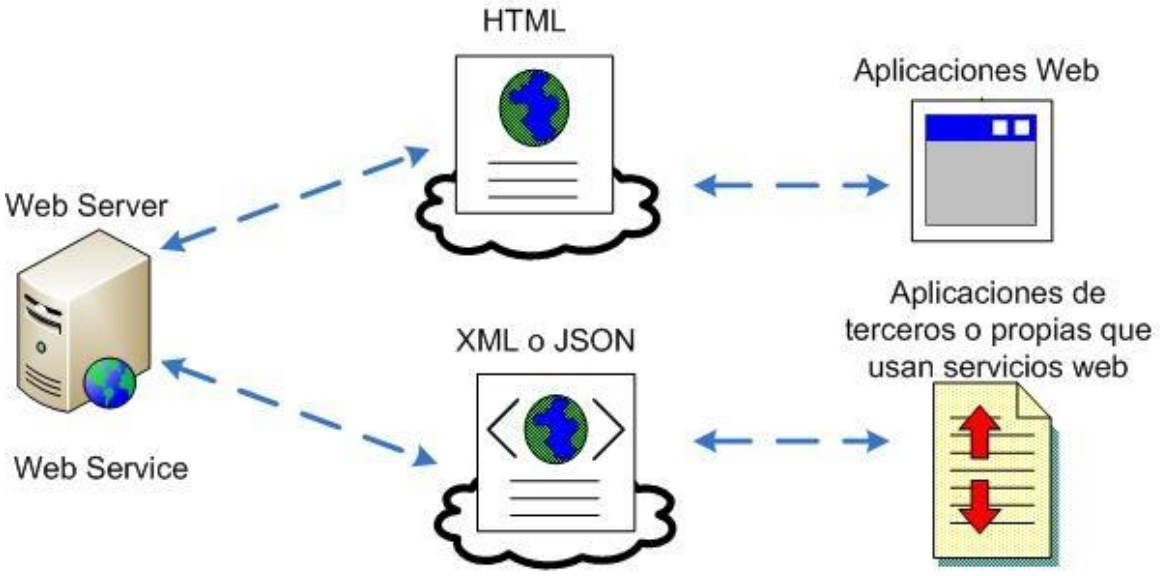


**Fig. 7.3: Aplicaciones Web y WebService**

## 7.2 - Sistemas VFP con servicios web .NET

En primer lugar como el sistema estaba desarrollado en VFP9 y base de datos SQL se buscó utilizar .Net como elemento a incorporar pero a su vez sin abandonar VFP. Se pensó en crear un servicio web sobre IIS donde un componente COM+ desarrollado en VFP era la referencia del WS .Net. Del lado del cliente una DLL hecha con .Net leía sobre el puerto 80 y servía de datos al front end VFP. Pero como se tenía un web service corriendo se podía usar esto como fuente de otros sistemas más evolucionados sin perder la programación anterior. La Fig. Nº 7.4 muestra los distintos elementos involucrados.

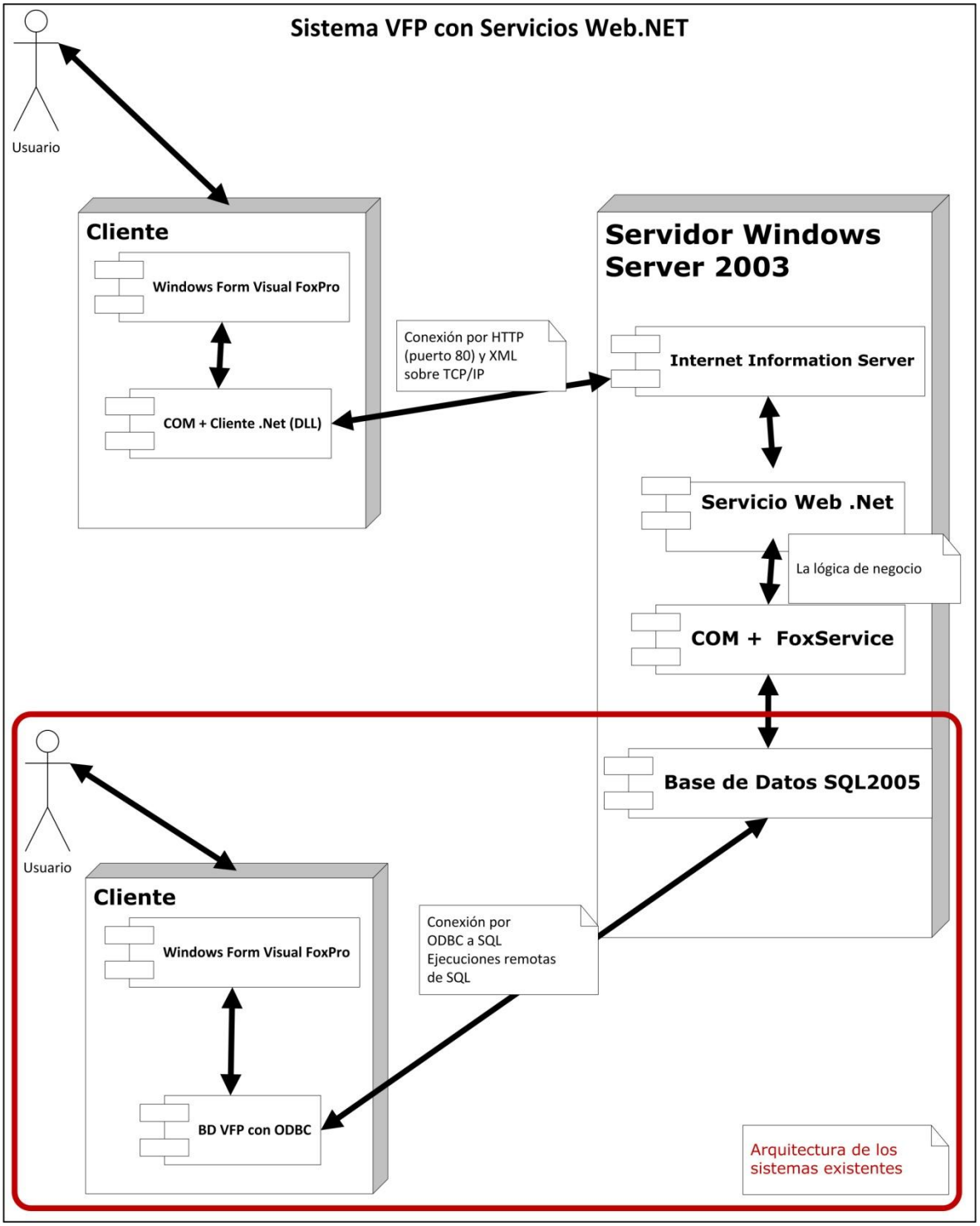


**Fig. 7.4: Sistema VFP con servicios Web.Net**

Aquí se ve cómo es necesario formar dos capas de conversión para mantener un front end en VFP. Esto trae la degradación del rendimiento. Un COM+(DLL) hace de intermediario entre la base de datos y el servicio .Net. Si bien VFP puede consumir servicio web directamente, técnicamente puede tener problemas al leer objetos de negocios

que no son naturales para VFP. Entonces hay que agregar un objeto COM+(DLL) que lea el servicio web y lo traduzca para ser consumido por VFP. El resultado será que se siguen desarrollando nuevas funcionalidades con el viejo VFP. Pero lo positivo será que ya se está acostumbrado a trabajar con él.

A su vez lo arriba detallado permite alimentar una nueva programación web forms en C# usando el framework Visual Studio. De esta manera se logra integrar modificaciones de VFP sobre los programas operativos y encarar nuevos subsistemas C#.

Aquí se debe explicar qué es un servicio web y su evolución.

## 7.3 - Servicios web

Un **servicio** es una unidad de funcionalidad que se expone al mundo exterior. La **orientación a servicios** es, entonces, un conjunto de principios y prácticas recomendadas para desarrollar aplicaciones basadas en la utilización de servicios.

Los servicios pueden ser locales o remotos, haber sido desarrollados utilizando cualquier tecnología e incluso funcionar asíncronamente. Pero los lenguajes y, tecnologías utilizados para desarrollar un servicio son detalles internos de su implementación, que permanecen encapsulados. Los servicios se comunican únicamente siguiendo patrones claramente definidos.

Se denomina **cliente** de un servicio a la parte que consume la funcionalidad de éste. El cliente también puede haber sido creado con cualquier tecnología, y puede ser cualquier tipo de aplicación o librería, aplicación o servicio que se ejecuta localmente, aplicación o servicio web, etc.

Los clientes y los servicios interactúan intercambiando **mensajes**, que pueden ser transferidos directamente o a través de un intermediario.

Los servicios web permiten la integración de las aplicaciones de distintos orígenes, y ayudan a la reutilización de sus componentes.

MSDN define a los servicios web de la siguiente forma: **(3)**

*"Un servicio web XML es una entidad programable que proporciona un elemento de funcionalidad determinado, como lógica de aplicación, al que se puede tener acceso desde*

*diversos sistemas potencialmente distintos mediante estándares de internet muy extendidos, como XML y HTTP".*

Un poco más claro queda en el siguiente párrafo:

*"Un servicio web XML puede ser utilizado internamente por una aplicación o bien ser expuesto de forma externa en internet para ser utilizado por varias aplicaciones. Dado que es posible el acceso a los servicios web XML a través de una interfaz estándar, éstos permiten el funcionamiento de una serie de sistemas heterogéneos como un conjunto integrado".*

Actualmente existen dos tendencias de arquitectura en cuanto a servicios web:

- SOAP (Especificaciones WS-I, WS-*)

- REST (Servicios RESTful)

SOAP está basado especialmente en los mensajes SOAP, que es un formato de los mensajes que utilizan XML y HTTP como protocolo de transporte (como los servicios-web ASMX o servicios WCF).

REST está muy orientado a la URI, al direccionamiento de los recursos basándose en la URL de HTTP y por lo tanto los mensajes a intercambiar son mucho más sencillos y ligeros que los mensajes XML de SOAP.

## 7.4 - El protocolo SOAP

SOAP, es una especificación para intercambio de información estructurada en la implementación de servicios web, recomendada por la W3. Originalmente "SOAP" era un acrónimo de "Simple Object Access Protocol", pero la versión 1.2 (2003) eliminó el acrónimo.

SOAP es un concepto tecnológico basado en la ***sencillez*** y la ***flexibilidad*** que hace uso de ***tecnologías y estándares comunes*** para conseguir las promesas de la ubicuidad de los servicios, la transparencia de los datos y la independencia de la plataforma.

SOAP empezó como un protocolo de invocación remota basado en XML, llamado **XML-RPC**. A partir de éste se obtuvo en septiembre de 1999 la versión 1.0 de SOAP.

Este protocolo está definido en tres partes:

- Un sobre de mensaje, que define qué habrá en el "cuerpo" o contenido del mensaje y cómo procesarlo.
- Un conjunto de reglas de serialización para expresar instancias de tipos de datos de aplicación.
- Una convención para representar llamadas y respuestas a métodos remotos.


**Fig. 7.5: Estructura paquete SOAP Message**

Un mensaje SOAP se utilizará tanto para solicitar una ejecución de un método de un servicio web remoto como para dar respuesta, conteniendo la información solicitada. Debido a que el formato de los datos es XML (texto, con un esquema, pero texto, al fin y al cabo), puede llegar a integrarse en cualquier plataforma tecnológica. SOAP es interoperable. El estándar básico de SOAP es "SOAP WS-I *Basic Profile"*.

### 7.4.1 -Especificaciones WS-*

Servicios web básicos (como SOAP WS-I, *Basic Profile*) no ofrecen más que comunicaciones entre el servicio web y las aplicaciones cliente que lo consumen. Sin embargo, los estándares de servicios web básicos (*WS-Basic Profile*), fueron simplemente el inicio de SOAP.

Las aplicaciones empresariales complejas y transaccionales requieren de muchas más funcionalidades y requisitos de calidad de servicio (QoS) que simplemente una comunicación entre cliente y servicio web. Los puntos o necesidades más destacables de las aplicaciones empresariales pueden ser aspectos como:

- Seguridad avanzada orientada a mensajes en las comunicaciones con los servicios, incluyendo diferentes tipos de autenticación, autorización, cifrado, firma, etc.
- Mensajería estable y confiable.
- Soporte de transacciones distribuidas entre diferentes servicios.
- Mecanismos de direccionamiento y ruteo.
- Metadatos para definir requerimientos como políticas.
- Soporte para adjuntar grandes cantidades de datos binarios en llamadas/respuestas a servicios (imágenes y/o "attachments" de cualquier tipo).

Para definir todas estas "necesidades avanzadas", la industria (diferentes empresas como Microsoft, IBM, HP, Fujitsu, BEA, VeriSign, SUN, Oracle, CA, Nokia, CommerceOne, Documentum, TIBCO, etc.) ha definido unas especificaciones teóricas que establecen cómo deben funcionar los "aspectos extendidos" de los servicios web.

A todas estas "especificaciones teóricas" se las conoce como las **especificaciones WS.***. (El "*" se refiere a que son muchas las especificaciones de servicios web avanzados, como *WS-Security*, *WS-SecureConversation*, WS-AtomicTransactions, etc.). Si se quiere conocer en detalle estas especificaciones, se puede consultar los estándares en: **http://www.oasis-open.org**.

En definitiva, esas especificaciones WS-* definen teóricamente los requerimientos avanzados de las aplicaciones empresariales. El siguiente esquema muestra a alto nivel las diferentes funcionalidades que trata de resolver WS.*.

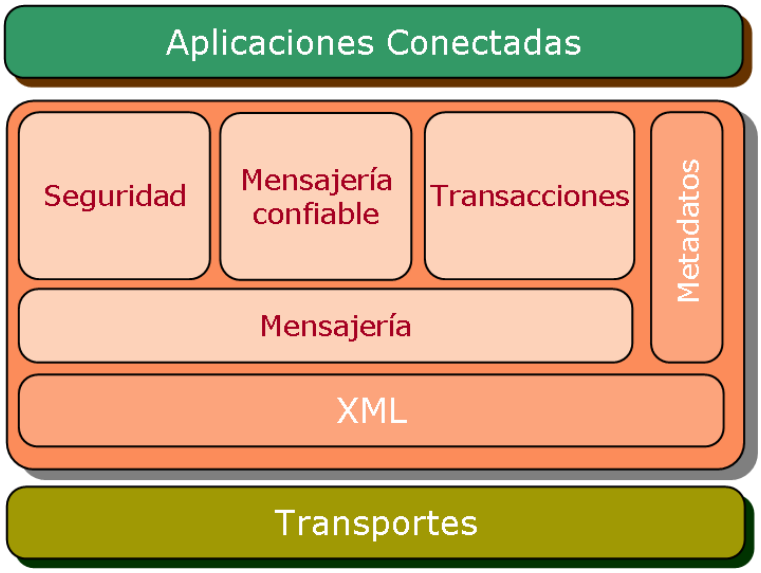


**Fig. 7.6: Esquema con las diferentes funcionalidades para resolver WS.*.**

Todos los módulos centrales (seguridad, mensajería confiable, transacciones y metadatos) son precisamente las funcionalidades de las que carecen los servicios web XML básicos y lo que define precisamente WS.*.

Las **especificaciones WS.*** están asimismo formadas por subconjuntos de especificaciones:

- WS-Security.

- WS-Messaging.

- WS-Transaction.

- WS-Reliability.

- WS-Metadata.

Estos a su vez se vuelven a dividir en otros subconjuntos de especificaciones aún más definidas, que son prácticamente todo un mundo, o sea que no es algo pequeño y limitado que simplemente extiende un poco más los servicios web básicos.

Los servicios web basados en estas especificaciones son **servicios web avanzados**.

### 7.4.2 - Interoperabilidad de servicios web

Por interoperabilidad se conoce a la capacidad de comunicación entre diferentes plataformas de servicios web. Aún cuando los servicios web XML fueron desarrollados utilizando especificaciones públicas (XML, SOAP, etc.) para alcanzar el máximo grado de

interoperación entre servicios web en plataformas diferentes, las ambigüedades de esas especificaciones y las diferencias entre implementaciones, han demorado el desarrollo de servicios web ampliamente interoperables.

Para resolver estos problemas, se creó un grupo de trabajo llamado Web Services Interoperability Group (**WS-I**), cuyo sitio Web es **http://www.ws-i.org/**, que ha desarrollado la especificación **Basic Profile** (BP). Esta especificación elimina las ambigüedades, describe claramente las limitaciones y establece un conjunto de recomendaciones para que los servicios web puedan interoperar de manera óptima entre plataformas, sistemas operativos y lenguajes de programación.

Actualmente, las plataformas de servicios web más populares cumplen con la especificación WS-I BP (Basic Profile) como **servicios web básicos.**

Durante el proceso de investigación de este trabajo esta opción de servicios web fue evolucionando hacia un servicio WCF Windows Communication Foundation.

En este punto de la investigación se empezaba a aclarar la idea de una arquitectura SOA (arquitectura orientada a servicios), creando una capa superior que incluyera componentes que prestaran servicios web (SOAP, XML, HTTP). Para ello se incorporó como herramientas a utilizar WCF y WPF (Windows Presentation Foundation).

### 7.4.3 WCF –Windows Communication Foundation

WCF es una librería incorporada a .NET 3.0 para promover la utilización de arquitecturas orientadas a servicios (SOA) y facilitar el desarrollo y consumo de todo tipo de servicios. WCF es la implementación de Microsoft de un conjunto de estándares de la industria, que definen la interacción entre servicios, la representación de los datos que estos intercambian y la gestión de los protocolos involucrados, lo que garantiza la interoperabilidad con otros lenguajes y plataformas.

En WCF, todos los mensajes son mensajes SOAP. Se debe observar que los mensajes son independientes de los protocolos de transporte, a diferencia de los servicios web ASP.NET, los servicios WCF pueden comunicarse utilizando toda una variedad de transportes y no solo HTTP. Los clientes WCF pueden interoperar con servicios no-WCF.

### 7.4.4 WPF – Windows Presentation Foundation

Es una tecnología de Microsoft que permite el desarrollo de interfaces en Windows tomando características de aplicaciones Windows y de aplicaciones web. WPF ofrece una amplia infraestructura y potencia gráfica con la que es posible desarrollar aplicaciones visualmente óptimas. Facilita la interacción con páginas que incluyen animación, audio, video, documentos, etc. Usa el lenguaje declarativo XAML (Extensible Application Markup Language) y los lenguajes de programación .Net para separar la interfaz de interacción de la lógica del negocio, propiciando una arquitectura modelo vista controlador para el desarrollo de las aplicaciones.

## 7.5 – Sistema WCF – WPF.NET

Se investigó WCF (Windows Communication Foundation) como generador de web services. Luego se pensó en un nuevo front end en WPF consumiendo esos web services y trabajando en paralelo con la antigua programación. Siempre con la misma base de datos SQL como persistencia de datos.

En la figura que sigue está representado el esquema de la arquitectura investigada de la cual se hizo un prototipo.

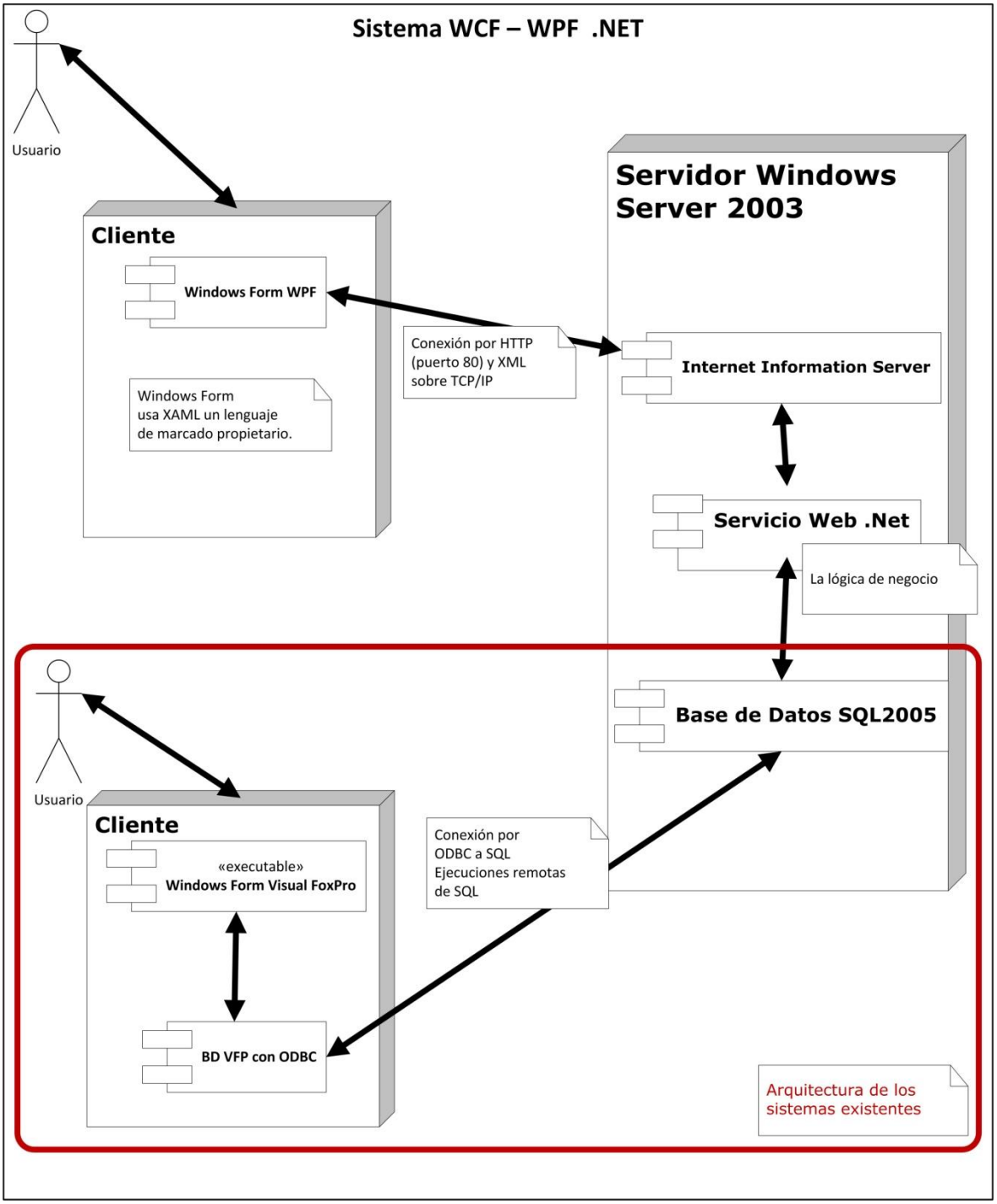


**Fig. 7.7: Sistemas WCF-WPF .Net**

En el transcurso de la investigación surgió como inconveniente insalvable la siguiente información, que se convierte en un punto de inflexión:

**¿Adiós, SOA, adiós?**

**Lunes 15 de noviembre de 2010**


*La semana pasada la Web Services Interoperability Organization (más conocida como WS-I) anunció en un comunicado de prensa que da por cumplida su labor con la reciente aprobación de las versiones originales de los estándars Basic Profile 1.2 y 2.0, y el Reliable Secure Profile, e inicia la transición de su labor de difusión y soporte de estos temas a OASIS.*

*El comunicado y algunos saludos de parte de los integrantes de este comité de parte de la industria suenan a un final laudatorio, pero personalmente me suena más a un final realista de un cuerpo cuyo impacto fue mayor en los presupuestos invertidos por algunas empresas que en el avance general de la industria.*

*La Arquitectura Orientada a Servicios (SOA) ha cumplido muy pocas de sus promesas, y como muchas otras de estas iniciativas de estilo comité entre varios participantes, siempre pareció llegar un poco tarde. Mientras tanto, fuera de los confines de los sectores más corporativos, los web services han dejado el complejo y problemático camino de SOAP y sus descendientes y se han volcado ampliamente al paradigma REST, donde la interoperabilidad nunca fue un problema, y unos pocos principios fundamentales y su aplicación práctica definieron el camino sin necesidad de organizaciones especiales.*

*Irónicamente, la WS-I se despide haciendo gala de un pobre entendimiento de la web, publicando este comunicado como PDF, casi como evitando que los buscadores ayuden a divulgar sus últimos pasos.*

*Por Martin Salias (http://www.codeandbeyond.org/2010/11/adios-soa-adios.html)*

## 7.6 El camino es hacia REST

### 7.6.1 - Fundamento de REST

**¿Qué es REST?**, REST fue introducido por *Roy Fielding* en una ponencia, **(4)** donde describió un "estilo de arquitectura" de sistemas interconectados. Así mismo, REST es un acrónimo que significa "*Representational State Transfer*".

¿Por qué se le llama "*Representational State Transfer*"? La web es un conjunto de recursos. Un recurso es un elemento de interés. Por ejemplo, se puede definir un tipo de recurso, que podría ser, una lista de inmuebles de Rosario. En este caso, los clientes

pueden acceder a dicho recurso con una URL como:

**http://www.inmobiliariapepe.com.ar/rosario/departamentos/1**

Para acceder a dicho recurso, se devolverá una **representación** de dicho recurso (por ejemplo: DepartamentosUnAmbiente.htm). La **representación** sitúa a la aplicación cliente en un **estado**. El resultado del cliente accediendo a un enlace dentro de dicha página HTML será otro recurso accedido. La nueva representación situará a la aplicación cliente en otro estado. Así pues, la aplicación cliente cambia (**transfiere**) de estado con cada representación de recurso. En definitiva "*Representational State Transfer*".

O sea, los objetivos de REST son plasmar **las características naturales de la web que han hecho que internet sea un éxito**. Son precisamente esas características las que se han utilizado para definir REST. REST no es un estándar, es un estilo de arquitectura. Es difícil crear una especificación REST, porque REST es solamente un estilo de arquitectura. No se puede "empaquetar" un estilo, solo se puede comprender y diseñar los servicios web según ese estilo.

Sin embargo, aunque REST no es un estándar en sí mismo, sí que está basado en estándares de internet:

- HTTP
- URL
- XML/HTML/PNG/GIF/JPEG/etc.(representaciones de recursos)
- Text/xml, text/html, image/gif, etc. (tipos MIME)

**La URI en REST**

Como concepto fundamental, lo más importante en REST es la URI (URI es una forma más técnica de decir URL, por lo que es mejor decirlo así). La URI es muy importante en REST porque basa todas las definiciones de acceso a servicios web en la sintaxis de una URI. Para que se entienda, aquí van varios ejemplos de URIs de servicios web basados en REST. Como se verá, es autoexplicativo según su definición, y ese es uno de los objetivos de REST, simplicidad y autoexplicativo, entonces se entienden bien las URIs de ejemplo:

**http://www.midominio.com/Proveedores/Todos/**

**http://www.midominio.com/Proveedores/2050**

**http://www.midominio.com/Proveedores/Ramirez/Jose**

**Simplicidad**

La simplicidad es uno de los aspectos fundamentales en REST. Se persigue la simplicidad en todo, desde la URI hasta en los mensajes XML proporcionados o recibidos desde el servicio web. Esta simplicidad es una gran diferencia si lo comparamos con SOAP, que puede tener gran complejidad en sus cabeceras.

El beneficio de dicha simplicidad es poder conseguir un buen rendimiento y eficiencia (aun cuando se esté trabajando con estándares no muy eficientes, como HTTP y XML), pero al final, los datos (bits) que se transmiten son siempre los mínimos necesarios. Se tiene algo ligero, por lo que el rendimiento será bastante óptimo. Por el contrario, como contrapartida si se elige algo bastante simple, habrá muchas cosas complejas que sí se pueden implementar con estándares SOAP y que con REST es imposible, por ejemplo, estándares de seguridad, firma y cifrado a nivel de mensajes, transaccionalidad que englobe diferentes servicios web y un largo etc. de funcionalidades avanzadas, que sí se definen en las especificaciones WS-* basado en SOAP.

Pero el objetivo de REST no es conseguir una gran o compleja funcionalidad, sino conseguir un mínimo de funcionalidad necesaria en un gran porcentaje de servicios web en internet, que sean interoperables, que simplemente se transmita la información y que sean servicios web muy eficientes.

Es un contraste notable entre un mensaje REST devuelto por un servicio web y un mensaje SOAP WS-* que puede llegar a ser muy complejo y por lo tanto también más pesado, reduciendo la eficiencia de la comunicación.

**URLs lógicas versus URLs físicas**

Un recurso es una entidad conceptual. Una representación es una manifestación concreta de dicho recurso. Por ejemplo: http://www.miempresa.com/clientes/00345

La anterior URL es una URL lógica, no es una URL física. Por ejemplo, no hace falta que exista una página HTML para cada cliente de este ejemplo.

Un aspecto de diseño correcto de URIs en REST es que no se debe revelar la tecnología utilizada en la URI/URL. Se debe poder tener la libertad de cambiar la implementación sin impactar en las aplicaciones cliente que lo están consumiendo.

**Características base de los servicios web REST**

- Cliente-Servidor: estilo de interacción "pull". Métodos complejos de comunicaciones tipo Full-Duplex o Peer-to-Peer quedan lejos de poder ser implementadas con REST. REST es para servicios web sencillos.
- *Stateless* (sin estados): cada petición que hace el cliente al servidor debe contener toda la información necesaria para entender y ejecutar la petición. No puede hacer uso de ningún tipo de contexto del servidor.
- *Cache*: para mejorar la eficiencia de la red, las respuestas deben poder ser clasificadas como "*cacheables*" y "no *cacheables*".
- Interfaz uniforme: todos los recursos son accedidos con un interfaz genérico (ejemplo: HTTP GET, POST, PUT, DELETE), sin embargo, el interfaz más importante o preponderante en REST es GET (como las URLs mostradas de ejemplo anteriormente). GET es considerado "especial" para REST.
- El tipo de contenido (content-type) es el modelo de objetos, tales como imágenes, XML, JSON, etc.
- Recursos nombrados: el sistema está limitado a recursos que puedan ser nombrados mediante una URI/URL.
- Representaciones de recursos interconectados: las representaciones de los recursos se interconectan mediante URLs, esto permite a un cliente progresar de un estado al siguiente.

**Principios de diseño de servicios web REST**

1. La clave para crear servicios web en una red REST (por ejemplo: la web en internet) es identificar todas las entidades conceptuales que se desea exponer como servicios.

2. Crear una URI/URL para cada recurso. Los recursos deben ser nombres, no verbos. Por ejemplo, no se debe utilizar esto:

http://www.miempresa.com/clientes/obtenercliente?id=00452

Es incorrecto el verbo "obtenercliente". En lugar de eso, se debe poner solo el nombre, así: http://www.miempresa.com/clientes/cliente/00452

3. Categorizar los recursos de acuerdo a si las aplicaciones cliente pueden recibir una representación del recurso, o si las aplicaciones cliente pueden modificar (añadir) al recurso. Para lo primero, se debe poder hacer accesible el recurso con un HTTP GET, para lo segundo, se deben poder hacer accesibles los recursos con HTTP POST, PUT y/o DELETE.

4. Las representaciones no deben ser islas de información. Por eso se deben implementar enlaces dentro de los recursos para permitir a las aplicaciones cliente el consultar más información detallada o información relacionada.

5. Diseñar para revelar datos de forma gradual. No revelar todo en una única respuesta de documento. Proporcionar enlaces para obtener más detalles.

6. Especificar el formato de la respuesta utilizando un esquema XML o datos en JSON.

### 7.6.2 - REST VS. SOAP

REST (*Representational State Transfer*) y SOAP **representan dos estilos** bastante diferentes para implementar una arquitectura de servicios distribuidos. Técnicamente, REST es un patrón de arquitectura construido con verbos simples que encajan perfectamente con HTTP. Si bien, aunque los principios de arquitectura de REST podrían aplicarse a otros protocolos adicionales a HTTP, en la práctica, las implementaciones de REST se basan completamente en HTTP.

La principal diferencia entre estos dos enfoques es la forma en la que se mantiene el estado del servicio. Se refiere a un estado muy diferente al estado de sesión o aplicación. En realidad se trata de los diferentes estados por los que una aplicación pasa durante su vida. Con SOAP, el movimiento por los diferentes estados se realiza interactuando con un extremo único del servicio que puede encapsular y proporcionar acceso a muchas operaciones y tipos de mensaje.

Por el contrario, con REST, se permite un número limitado de operaciones y dichas operaciones se aplican a recursos representados y direccionados por URIs (direcciones

HTTP). Los mensajes capturan el estado actual o el requerido del recurso. REST funciona muy bien con aplicaciones web donde se puede hacer uso de HTTP para tipos de datos que no son XML. Los consumidores del servicio interactúan con los recursos mediante URIs de la misma forma que las personas pueden navegar e interactuar con páginas web mediante URLs (direcciones web).

Desde un punto de vista tecnológico, estas son algunas ventajas y desventajas de ambos:

Ventajas de SOAP:

- Bueno para datos (las comunicaciones son estrictas y estructuradas).

- Dispone de proxies fuertemente tipados gracias a WSDL.

- Funciona sobre diferentes protocolos de comunicaciones.

Desventajas de SOAP:

- Los mensajes de SOAP no son "cacheables".

- No se puede hacer uso de mensajes SOAP en JavaScript (para AJAX debe utilizarse REST).

Ventajas de REST:

- Gobernado por las especificaciones HTTP por lo que los servicios actúan como recursos, igual que imágenes o documentos HTML.

- Los datos pueden bien mantenerse de forma estricta o de forma desacoplada (no tan estricto como SOAP).

- Los recursos REST pueden consumirse fácilmente desde código JavaScript (AJAX, etc.).

- Los mensajes son ligeros, por lo que el rendimiento y la escalabilidad que ofrece es muy alta. Lo cual es importante para internet.

- REST puede utilizar bien XML o JSON como formato de los datos.

Desventajas de REST:

- Es difícil trabajar con objetos fuertemente tipados en el código del servidor.

- Solo funciona normalmente sobre HTTP.

- Las llamadas a REST están restringidas a verbos HTTP (GET, POST, PUT, DELETE, etc.).

Aun cuando ambas aproximaciones (REST y SOAP) pueden utilizarse para diferentes tipos de servicios, la aproximación basada en REST es normalmente más adecuada para servicios distribuidos accesibles públicamente (internet) o en casos en los que un servicio pueda ser accedido por consumidores desconocidos. SOAP se ajusta, por el contrario, mucho mejor a implementar interfaces entre las diferentes capas de una arquitectura de aplicación.

## 7.7 - MVC – Modelo Vista Controlador

Aunque de forma algo simplista, se puede definir MVC como un patrón arquitectural que describe una forma de desarrollar aplicaciones software separando los componentes en tres grupos (o capas):

- El **Modelo** que contiene una representación de los datos que maneja el sistema, su lógica de negocio y sus mecanismos de persistencia.
- La **Vista**, o interfaz de usuario, que compone la información que se envía al cliente y los mecanismos de interacción con éste.
- El **Controlador**, que actúa como intermediario entre el modelo y la vista, gestionando el flujo de información entre ellos y las transformaciones para adaptar los datos a las necesidades de cada uno.

**MVC** son las siglas de Modelo-Vista-Controlador. Se trata de un modelo muy maduro y que ha demostrado su validez a lo largo de los años en todo tipo de aplicaciones, sobre multitud de lenguajes y plataformas de desarrollo.

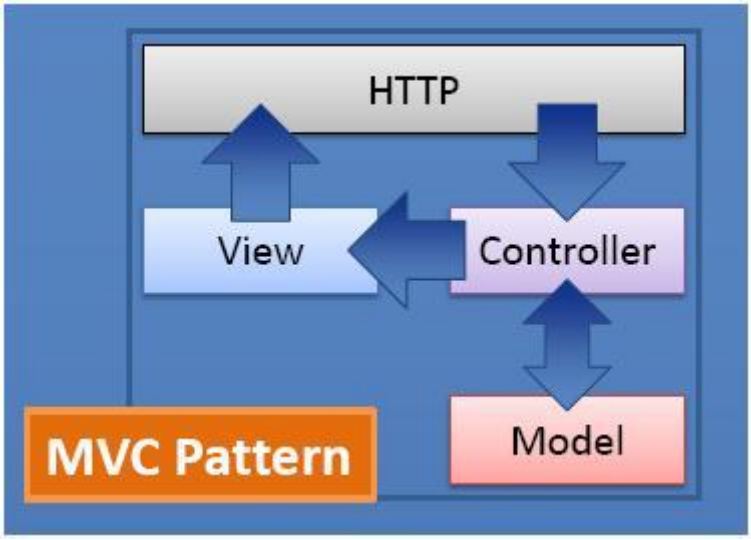


**Fig. 7.8: Patrón MVC**

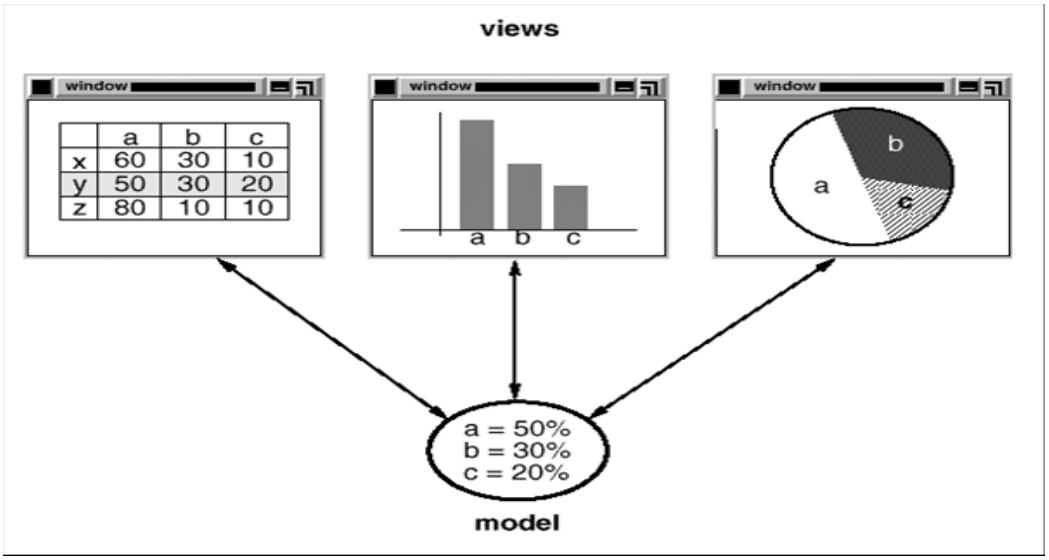


**Fig. 7.9: Ejemplo práctico del patrón MVC**

Algunas características de MVC:

- Clara separación de responsabilidades entre interfaz, lógica de negocio y de control, que además provoca parte de las ventajas siguientes.
- Facilidad para la realización de pruebas unitarias de los componentes, así como de aplicar desarrollo guiado por pruebas (TDD).
- Simplicidad en el desarrollo y mantenimiento de los sistemas.
- Reutilización de los componentes.

- Facilidad para desarrollar prototipos rápidos.
- Sencillez para crear distintas representaciones de los mismos datos.
- Los sistemas son muy eficientes, y a la postre más escalables.

Pero también se pueden citar algunos inconvenientes:

- Tener que ajustarse a una estructura predefinida, lo que a veces puede incrementar la complejidad del proyecto. De hecho, hay problemas que son más difíciles de resolver, o al menos cuestan algo más de trabajo, respetando el patrón MVC.
- Al principio puede implicar cierto esfuerzo adaptarse a esta filosofía, sobre todo a desarrolladores acostumbrados a otros modelos más cercanos al escritorio, como webforms.
- La distribución de componentes obliga a crear y mantener un mayor número de archivos.

Luego de todo el análisis realizado en este capítulo, estoy en condiciones de concluir que el patrón MVC fue el elegido porque respondía a los requerimientos planteados al comienzo de este trabajo.

En primer lugar porque resolvía el tema de la independencia de la distribución geográfica para la aplicación. Es un patrón muy natural para internet a diferencia de otras opciones en especial el webforms de la ASP.Net. Esta última opción es algo forzada para la web puesto que mantiene el estado llevando y trayendo en cada comunicación los datos necesarios para mantener los estados.

En segundo lugar MVC, resultó exitoso también porque con esa estructura se pueden prestar servicios web no solo a programas propios sino a programas de terceros que así lo requieran. Con la aparición de MVC API, se facilitó notablemente la prestación de esta funcionalidad.

Como arquitectura en cuanto a servicios web la opción estuvo dada hacia REST, ya que la aplicación SOAP dejó de tener soporte para fututos desarrollos, por el contrario REST se torna cada vez más popular y es una forma exitosa de comunicarse en internet.

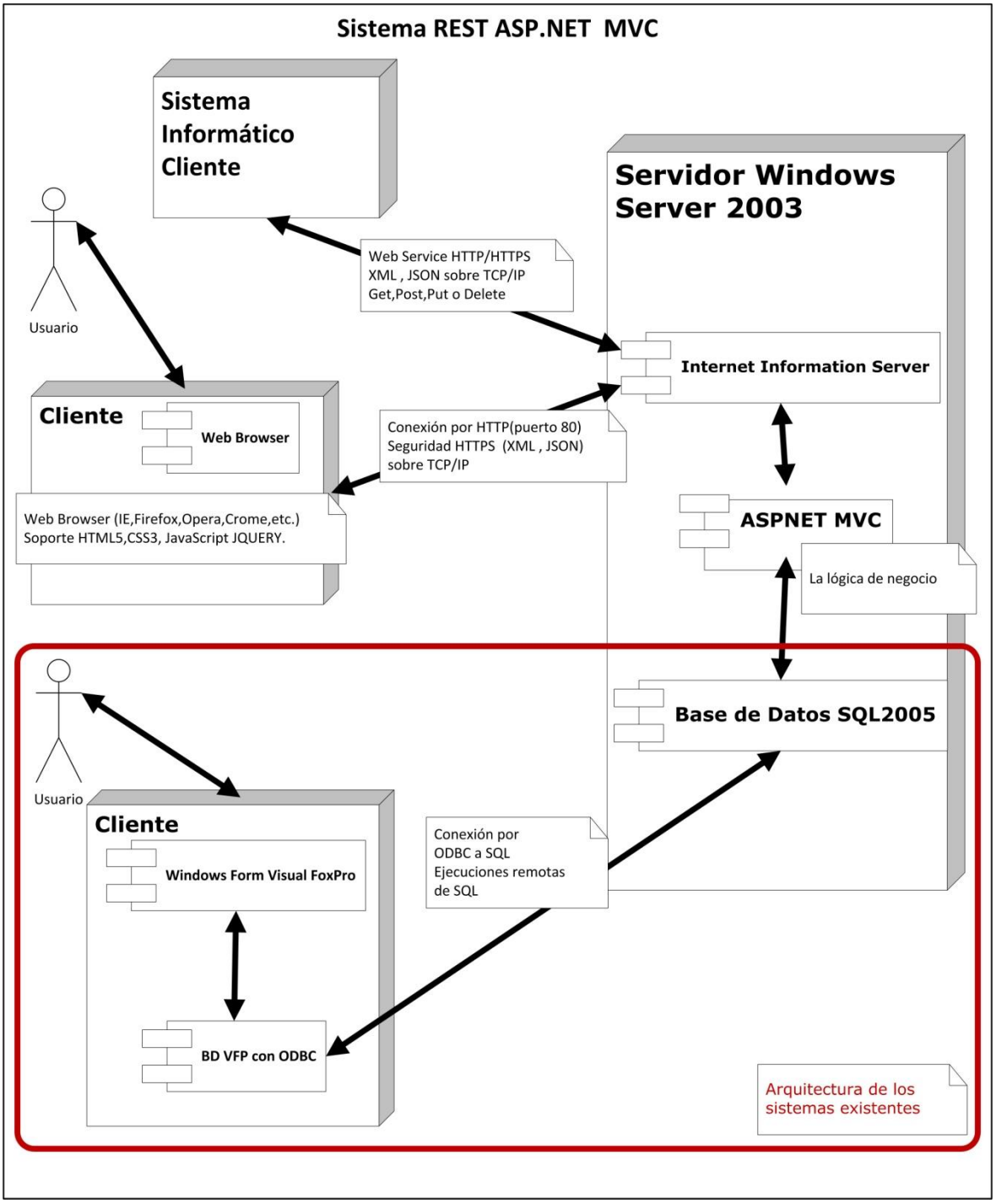


**Fig. 7.10: Arquitectura ASP.NET MVC**

# 8.0 – TECNOLOGÍAS UTILIZADAS

Una vez encontrada la arquitectura más conveniente, tuve que elegir herramientas. Las herramientas más destacadas de las que resultaron elegidas, son analizadas en este capítulo donde describo y fundamento el por qué de su elección, acentuando que si bien en algunos casos existen desde hace tiempo, sus principios cobraron importancia recientemente.

## 8.1- XP Programación Extrema

Programación Extrema (XP) es una metodología ágil creada por Kent Beck. Propone un conjunto de valores, principios y prácticas que permiten desarrollar rápidamente software de alta calidad en el menor tiempo posible.

Kent Beck en su libro *Extreme Programming Explained: Embrace Change* afirma que es una metodología ágil para que un equipo de desarrolladores, de tamaño pequeño a medio, genere software frente a requisitos imprecisos o muy cambiantes.

XP es extremo en el sentido que lleva al extremo doce de las mejores prácticas "bien conocidas" (o tradicionales) de desarrollo de software. Presenta la ventaja de reducir el costo del proyecto, bajar los riesgos y aceptar los cambios del negocio.

Esta metodología promueve como pilares cuatro valores:

**Comunicación:** los desarrolladores necesitan intercambiar información e ideas sobre el proyecto con los directivos y los clientes de forma confiable y fácil. La información debe fluir de manera continua y rápida.

**Sencillez:** siempre que sea posible hay que elegir soluciones simples.

**Retroalimentación:** en todos los niveles las personas deberían obtener una retroalimentación muy rápida sobre lo que hacen. Los clientes, los directivos y los desarrolladores tienen que alcanzar una comprensión común de la meta del proyecto y también acerca del estado actual del proyecto.

**Valentía:** cada persona implicada en el proyecto debería de tener el valor (y el derecho) de expresar su valoración sobre el mismo. Todos deberían de tener el valor de ser abiertos y dejar que a su vez todos examinen e incluso modifiquen su trabajo.

### 8.1.1 - Las prácticas y su aplicación

XP está compuesto por un conjunto de prácticas estrechamente relacionadas entre sí que aplicadas todas juntas son capaces de reforzar las debilidades que algunas de ellas pueden presentar si se aplican en forma separada. Estas prácticas en sí mismas no son nuevas, ya que muchas han demostrado ser las mejores de las utilizadas en la historia del desarrollo de software; la novedad radica en llevarlas al extremo y en utilizarlas todas juntas.

A continuación, se describe brevemente cada práctica y su aplicación en este trabajo.

**1 - El juego de la planificación**

Consiste en determinar rápidamente el ámbito de la siguiente versión, combinando las prioridades del negocio y las estimaciones técnicas. Los planes se deben actualizar cada vez que la realidad los sobrepase.

Luego se desarrollará una planificación en XP sobre este proyecto.

**2 - Versiones pequeñas**

Primero se pone en producción rápidamente un sistema sencillo con los requisitos más importantes del negocio, de tal forma que el sistema tenga sentido como un todo. A partir de allí se liberan nuevas versiones en ciclos muy cortos.

La ventaja de las versiones pequeñas es la rapidez con la que se puede poner en producción el sistema y obtener retroalimentación por parte del cliente. Esta retroalimentación ayuda al cliente a detectar nuevas funcionalidades para escribir historias y mejorar las ya existentes.

La primera versión del sistema tendrá como objetivo desarrollar la administración de socios, sus registros, el aporte mensual, la participación en las actividades deportivas, etc.

En esta primera versión se observa la respuesta del usuario sobre la nueva forma de programar y se registran sus consecuencias. Los pros y las contras reciben el análisis y ponderación del equipo de desarrollo.

Una vez que se tiene la primera versión del sistema, se hará la segunda versión agregando gestión financiera con la mayoría de sus servicios.

En una tercera versión, se perfeccionarán los puntos con varios informes y se agregarán contabilidad y auditoría.

**3 - Metáfora**

La metáfora es una visión común, entre el cliente y el equipo de desarrollo, de cómo funciona el sistema. En el mejor de los casos es una descripción sencilla pero fuerte.

No siempre es fácil de encontrar una metáfora que describa en pocas palabras cómo funciona el sistema, pero de todas formas se usará un sistema de nombres para asegurar que todos entiendan cómo funciona el mismo y sepan dónde buscar una funcionalidad cuando necesiten trabajar sobre ella. Este conjunto de nombres actúa como vocabulario para hablar sobre el dominio del problema, ayudando a la nomenclatura de clases y métodos del sistema.

Los conceptos centrales que se utilizan y que están presentes en muchas de las historias son: socio, disciplina, servicio mutual.

- El caso de socio: es claro, es la persona que pertenece a una entidad abstracta. Es la representación dentro del sistema de las personas que forman parte de **San Martín Mutual Social y Biblioteca**.

- Disciplina: hace referencia a los datos propios de cada actividad en la entidad deportiva.

-Servicio mutual: son las características del tipo de servicio financiero prestado por la mutual.

**4 - Diseño sencillo**

En cualquier momento dado, el sistema deberá ser diseñado de forma tan simple como sea posible. La complejidad extra se eliminará tan pronto como sea descubierta.

Al momento de implementar una funcionalidad, la pregunta obligada es: ¿qué es lo más simple que podría funcionar? Introducir complejidad innecesaria es contraproducente, ya que al escribir pruebas se torna más difícil y prolonga el tiempo necesario para tener una versión lista.

Las prácticas en XP se apoyan entre sí: se refactoriza para lograr un diseño sencillo (por ejemplo cuando hay código duplicado) y las pruebas aseguran que no se cometan errores en el camino.

**5 - Pruebas**

La producción de código está respaldada por la creación de pruebas de unidad. De esta forma, el sistema puede ser modificado sin temor. Las pruebas aseguran que el sistema se sigue comportando de la misma forma. Así, esta práctica es fundamental para poder aceptar el cambio.

Para hacer pruebas se utilizó el *framework* de *testing* llamado Visual Studio Unit Test, el cual está incluido en la librería estándar de Visual Studio y pertenece a la familia de herramientas xUnit.

El sistema de Tests (Pruebas) es un punto a explicar.

**6 - Recodificación**

La recodificación o refactorización (como se la llamó antes) consiste en reescribir ciertas partes del código para aumentar su legibilidad y hacerlo de fácil mantenimiento, eliminar duplicación de código o hacerlo más sencillo, sin modificar su comportamiento. Las pruebas garantizan que en la recodificación no se ha introducido ningún fallo.

Esto suele suceder cuando se modifican algunas clases y sus métodos, o propiedades. Visual Studio tiene la facilidad de hacer refactorización por nosotros; eso ayuda al proceso.

La refactorización, sin temor a perder funcionalidades o introducir errores es posible gracias a las pruebas automatizadas. Así se ve que, las distintas prácticas se refuerzan entre sí.

**7- Programación en parejas**

Las tareas de desarrollo son llevadas a cabo de a dos personas en un mismo escritorio. Así se obtiene código de mayor calidad (ya que se lo puede revisar y discutir mientras se lo escribe). Esta mayor calidad es más importante que la posible pérdida de productividad que se pueda suponer *a priori*.

Además fortalece las prácticas de pruebas y recodificación.

La forma más común de llevar a cabo esta práctica consiste en que uno de los programadores escriba el código mientras el otro piensa más estratégicamente.

**8 - Propiedad colectiva del código**

En vez de dividir la responsabilidad en el desarrollo de cada módulo en grupos de trabajo distintos, esta práctica promueve que todos los miembros puedan corregir y extender cualquier parte del proyecto.

Se utiliza una herramienta para que ayude a seguir esta práctica: un servidor de control de versiones (Git Extensions), el cual guardará los distintos cambios que se hacen y el historial de cada archivo que se define afectado por el control de código fuente. Esto también permite trabajar sin temor a perder alguno de los cambios hechos ya que con este sistema siempre se puede volver a alguna de las revisiones anteriores.

Esta práctica es reforzada por el uso de estándares de codificación.

**9 - Integración continua**

El sistema debe ser integrado cada vez que se termina una tarea asegurándose de que ésta pase todas las pruebas.

Cuando los autores de código son varios, los cambios producidos por cada uno deben quedar reflejados en la versión central. Se necesita asegurar que el sistema esté en funcionamiento y todas las pruebas pasen en todo momento.

**10 - Horas semanales**

Esta práctica establece que no se debe trabajar más horas semanales de las planificadas y que no se deben hacer horas extras dos semanas seguidas.

Esto quiere decir que durante la mayoría del tiempo que dure el desarrollo se trabajará el tiempo planificado. Tal vez llegando al final de una versión se haga un esfuerzo extra (trabajando más horas), pero se debe evitar que esto se convierta en el estado normal del proyecto; una persona descansada realiza su trabajo mejor que una cansada.

**11 - Cliente in-situ**

Esta práctica establece que un representante del cliente debe trabajar junto al equipo de desarrollo. Esta persona debe ser alguien que utilizará el sistema cuando esté en

producción. En este caso los empleados de la mutual y del club son plenamente conscientes de la importancia de su participación en el proyecto para lograr mejores resultados.

**12 - Estándares de codificación**

En el caso analizado en este trabajo se usará StyleCop. Esto controla que las formalidades sean respetadas por todos los autores de código, facilitando que cada uno pueda entender el código del otro.

## 8.2 - Tests

Aplicar pruebas unitarias y refactoring permite reducir los tiempos cuando es necesario realizar cambios en el código, hacer mantenimiento a un software o desarrollarlo en forma más rápida y mejor.

El refactoring es una técnica para revisar el código fuente, alterando su estructura interna sin cambiar su comportamiento externo, esto es, se mejora el código sin perder su funcionalidad. Se aplica siempre que se vea acoplamiento, complejidad innecesaria y en general siempre que se vea algo que puede ser mejorado.

Para poder aplicar el refactoring es necesario apoyarse en pruebas unitarias. Las pruebas unitarias son código que prueba otro código. Este tipo de pruebas aseguran que el código está cumpliendo con los requerimientos dados.

Las pruebas unitarias se deben aplicar sobre las partes más pequeñas y comprobables de una aplicación, esto es, en los métodos de las clases.

**Características de las pruebas unitarias**

**Automatizable**: no debería requerirse una intervención manual.

**Completas**: deben cubrir la mayor cantidad de código.

**Repetibles o Reutilizables**: no se deben crear pruebas que sólo puedan ser ejecutadas una vez.

**Independientes**: la ejecución de una prueba no debe afectar a la ejecución de otra.

Las pruebas unitarias están conformadas por una estructura triple A, estas son:

**Arrange (Inicialización)**: donde se prepara el contexto de la prueba y se inicializan variables y demás parámetros.

**Action (Ejecución)**: ejecuta la funcionalidad que se desea probar.

**Assert (Comprobación)**: se comprueba que lo que se ejecutó hace lo que se desea.

Utilizar pruebas unitarias y refactoring de manera que , antes de empezar a escribir código funcional, se escriban primero las pruebas y luego su implementación constituye el TDD, Test Driven Development. Es una técnica de diseño e implementación de software incluida dentro de la metodología XP. El nombre *Diseño Dirigido por Ejemplos* hubiese sido el más apropiado para esta técnica.

TDD es una técnica para diseñar software que se centra en tres pilares fundamentales:

- La implementación de las funciones justas que el cliente necesita y no más.
- La minimización del número de defectos que llegan al software en fase de producción.
- La producción de software modular, altamente reutilizable y preparado para el cambio.

TDD es buena técnica para que el código tenga una cobertura de test muy alta, pero además es una herramienta de diseño que convierte al programador en desarrollador.

No se trata de escribir múltiples pruebas, sino de diseñar adecuadamente según los requisitos.

En este trabajo no se utilizó TDD pero sí se recurrió a métodos de pruebas unitarias e integrales por tratarse de un trabajo de reingeniería, donde las funcionalidades ya están escritas y desarrolladas.

## 8.3 – JQuery

Antes de hablar de JQuery comenzaré hablando de JavaScript.

Los programas web corren (se ejecutan) en uno de dos lugares: en el servidor web que aloja el sitio visitado (server side o back-end) o en la computadora del visitante (client side o front-end), más precisamente en el navegador web. JavaScript es un lenguaje de

programación client side, que se utiliza principalmente para crear páginas web dinámicas. Una página web dinámica es aquella que incorpora efectos como texto que aparece y desaparece, animaciones, acciones que se activan al pulsar botones y ventanas con mensajes de aviso al usuario.

Técnicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas para ejecutarlos. En otras palabras, los programas escritos con JavaScript se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.

A pesar de su nombre, JavaScript no guarda ninguna relación directa con el lenguaje de programación Java. Legalmente, JavaScript es una marca registrada de la empresa Sun Microsystems.

**JQuery** es una biblioteca de Java Script, creada inicialmente por John Resig, que permite simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM, manejar eventos, desarrollar animaciones y agregar interacción con la técnica AJAX a páginas web. Fue presentada el 14 de enero de 2006 en el Bar Camp NYC.

**JQuery** es software libre y de código abierto, posee un doble licenciamiento bajo la licencia MIT y la licencia pública general de GNU v2, permitiendo su uso en proyectos libres y/o propietarios. JQuery , al igual que otras bibliotecas, ofrece una serie de funcionalidades basadas en JavaScript que de otra manera requerirían de mucho más código, es decir, con las funciones propias de esta biblioteca se logran grandes resultados en menos tiempo y espacio.

JQuery está añadido como otro paquete de desarrollo al IDE Visual Studio y se usa junto con los frameworks ASP.NET y ASP.NET MVC.

## 8.4 – CSS

CSS es un lenguaje de hojas de estilos creado para controlar el aspecto o presentación de los documentos electrónicos definidos con HTML y XHTML. CSS es la mejor forma de separar los contenidos y su presentación y es imprescindible para crear páginas web complejas. Separar la definición de los contenidos y la definición de su aspecto presenta numerosas ventajas, ya que obliga a crear documentos HTML/XHTML bien definidos y con significado completo (también llamados *"documentos semánticos"*).

Además, mejora la accesibilidad del documento, reduce la complejidad de su mantenimiento y permite visualizar el mismo documento en infinidad de dispositivos diferentes.

Al crear una página web, se utiliza en primer lugar el lenguaje HTML/XHTML para *marcar* los contenidos, es decir, para designar la función de cada elemento dentro de la página: párrafo, titular, texto destacado, tabla, lista de elementos, etc.

Una vez creados los contenidos, se utiliza el lenguaje CSS para definir el aspecto de cada elemento: color, tamaño y tipo de letra del texto, separación horizontal y vertical entre elementos, posición de cada elemento dentro de la página, etc.

Las hojas de estilos aparecieron poco después que el lenguaje de etiquetas SGML, alrededor del año 1970. Desde la creación de SGML, se observó la necesidad de definir un mecanismo que permitiera aplicar de forma consistente diferentes estilos a los documentos electrónicos.

El gran impulso de los lenguajes de hojas de estilos se produjo con el boom de internet y el crecimiento exponencial del lenguaje HTML para la creación de documentos electrónicos. La guerra de navegadores y la falta de un estándar para la definición de los estilos dificultaban la creación de documentos con la misma apariencia en diferentes navegadores.

La versión de CSS que utilizan todos los navegadores de hoy en día es CSS 2.1 y CSS 3.0.

El trabajo del diseñador web siempre está limitado por las posibilidades de los navegadores que utilizan los usuarios para acceder a sus páginas. Por este motivo es imprescindible conocer el soporte de CSS en cada uno de los navegadores más utilizados del mercado.

Internamente los navegadores están divididos en varios componentes. La parte del navegador que se encarga de interpretar el código HTML y CSS para mostrar las páginas, se denomina motor. Desde el punto de vista del diseñador CSS, la versión de un motor es mucho más importante que la versión del propio navegador.

## 8.5 - HTML5

El HTML5 (HyperText Markup Language, versión 5) es la quinta revisión del lenguaje de programación "básico" de la World Wide Web, el HTML. Se empezó con HTML 2.0 (No hubo versión 1.0) y aterrizó en el relativamente estable HTML 4.01 en 1999. A continuación, la sintaxis se hizo más estricta, sobre la base de las reglas de XML, y se tuvo XHTML 1.0.

XHTML 2.0 fue de corta duración, ya que el objetivo de una web basada en XML se separó de lo que los desarrolladores y diseñadores estaban haciendo en el mundo real. Y luego, después de luchas internas y mucho alboroto entre las potencias que deciden estas especificaciones, aparece HTML5.

Esta nueva versión pretende reemplazar al actual (X)HTML, corrigiendo problemas con los que los desarrolladores web se encuentran, así como rediseñar el código actualizándolo a nuevas necesidades que demanda la web de hoy en día.

A diferencia de otras versiones de HTML, los cambios en HTML5 comienzan añadiendo semántica y accesibilidad implícitas, especificando cada detalle y borrando cualquier ambigüedad. Se tiene en cuenta el dinamismo de muchos sitios web (facebook, tweeter, etc), cuyo aspecto y funcionalidad son más semejantes a aplicaciones web que a documentos.

**Mejor estructura**

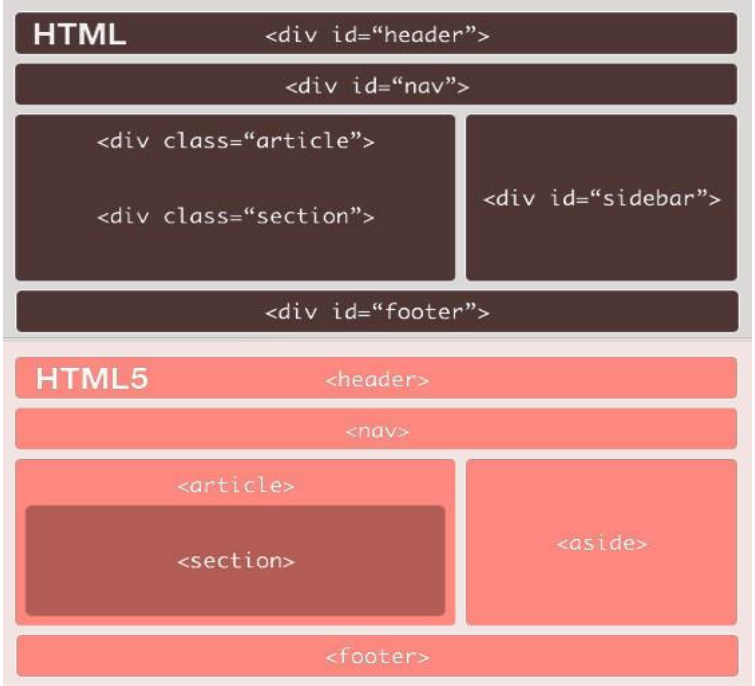


**Fig. 8.1: Estructura HTML 5**

Actualmente es abusivo el uso de elementos DIV para estructurar una web en bloques. El HTML5 brinda varios elementos que perfeccionan esta estructuración estableciendo qué es cada sección, eliminando así DIV innecesarios. Este cambio en la semántica hace que la estructura de la web sea más coherente y fácil de entender por otras personas. Así los navegadores podrán darle más importancia a distintas secciones de la web, facilitándoles además la tarea a los buscadores, así como cualquier otra aplicación que interprete sitios web. Las webs se dividirán en los siguientes elementos:

- <section></section> -
- <article></article> -
- <aside></aside> -
- <header></header> -
- <nav></nav> -
- <footer></footer>

**Diferencias entre HTML y HTML5**

En HTML5 existen nuevos tipos de input que son usados para ingresar datos en los formularios y que implican mejoras con respecto a HTML.

El elemento input adquiere gran relevancia al ampliarse los elementos que se permitirán en el “type”.

· <input type="search"> para cajas de búsqueda.

· <input type="number"> para adicionar o restar números mediante botones.

· <input type="range"> para seleccionar un valor entre dos valores predeterminados.

· <input type="color"> seleccionar un color.

· <input type="tel"> números telefónicos.

· <input type="url"> direcciones web.

· <input type="email"> direcciones de email.

· <input type="date"> para seleccionar un día en un calendario.

· <input type="month"> para meses.

· <input type="week"> para semanas.

· <input type="time"> para fechas.

· <input type="datetime"> para una fecha exacta, absoluta y tiempo.

· <input type="datetime-local"> para fechas locales y frecuencia.

**Otros elementos muy interesantes**

<audio> y <video> - Nuevos elementos que permitirán incrustar un contenido multimedia de sonido o de video, respectivamente. Es una de las novedades más importantes e interesantes en este HTML5, ya que permite reproducir y controlar videos y audio sin necesidad de aplicaciones externas como Flash.

El comportamiento de estos elementos multimedia será como el de cualquier elemento nativo y permitirá insertar en un video, enlaces o imágenes.

<embed> - Se emplea para contenido incrustado que necesita Flash. Es un elemento que ya reconocen los navegadores, pero ahora al formar parte de un estándar, no habrá conflicto con <object>.

<canvas> - Este es un elemento complejo que permite que se generen gráficos al hacer dibujos en su interior. Es utilizado en Google Maps y en un futuro permitirá a los desarrolladores crear aplicaciones muy interesantes.

## 8.6 – ASP.NET MVC Framework

Este framework es un marco de trabajo que ha sido creado por Microsoft con el objeto de ayudar a desarrollar aplicaciones que sigan la filosofía MVC sobre ASP.NET. Además del conjunto de librerías (ensamblados) que proporcionan las nuevas funcionalidades a nivel de API, incluye plantillas y herramientas que se integran en Visual Studio para facilitar un poco las cosas. Visual Studio 2012 ya incorpora ASP.NET MVC 4 de serie.

ASP.NET MVC es una forma de crear aplicaciones para la web, basada en ASP.NET de una forma radicalmente distinta a los formularios web:

- No existe el postback.
- No hay viewstate.
- No hay evento.
- El diseñador visual deja de tener sentido.
- Como consecuencia no hay controles de servidor, al menos en la forma Webforms.
- No es necesario utilizar los archivos code-behind de las páginas .aspx.
- Las páginas no siguen complejos ciclos de vida; de hecho, el proceso de una petición es infinitamente más simple que en Webforms.
- Se controla totalmente el código de marcado generado.
- También se tiene control absoluto sobre las URLs de acceso a la aplicación.
- Se basa en muchos aspectos en el concepto convención sobre configuración.
- Se integra con Ajax de forma natural, sin artificios como los UpdatePanels y similares.
- Favorece la introducción de buenas prácticas como la inversión de control o inyección de dependencias.

Realmente, ASP.NET MVC es una tecnología muy distinta y que requiere que los desarrolladores tengan que acostumbrarse a pensar de otra manera y dedicar tiempo a aprenderla para sacarle partido.

Existen otros frameworks MVC para ASP.Net, como MonoRail, Maverick.Net, FubuMVC y muchos otros.

MVC y Webforms de ASP.NET, son dos filosofías diferentes para conseguir lo mismo, páginas web.

La tecnología de Webforms es muy útil para asemejar el desarrollo de aplicaciones web a las de escritorio, ocultando la complejidad derivada del entorno desconectado y *stateless* (sin conservación de estado) del protocolo HTTP a base de complejos *roundtrips*, *postbacks* y *viewstates*, lo que nos permite crear de forma muy productiva formularios impresionantes y que el funcionamiento de la aplicación esté guiado por eventos, como si se estuviera programando Winforms.

Sin embargo, esta misma potencia a veces hace que las páginas sean pesadas y de difícil mantenimiento, además de dificultar enormemente la realización de pruebas automatizadas. Y por no hablar de comportamientos extraños cuando se intenta intervenir en el ciclo de vida de las páginas, por ejemplo para la carga y descarga de controles dinámicos.

Dado que el framework MVC está creado sobre ASP.NET, será posible utilizar páginas maestras, codificar las vistas en un .aspx utilizando C# o VB.NET, usar los mecanismos de seguridad internos, control de caché, gestión de sesiones, localización, etc.

ASP.NET MVC requiere un conocimiento más profundo del entorno web y sus tecnologías subyacentes, puesto que a la vez que ofrece un control mucho más riguroso sobre los datos que se envían y reciben desde el cliente, exige una mayor responsabilidad por parte del desarrollador, ya que deberá encargarse él mismo de mantener el estado entre peticiones, diseñar las vistas, crear las hojas de estilo apropiadas, e incluso los scripts.

Cada día existen más componentes para ASP.NET MVC de compañías dedicadas a la creación de herramientas de programación, y generados desde la propia comunidad de desarrolladores, que ayudan a ser más productivos.

También hay que tener en cuenta la compatibilidad tan sencilla del framework MVC con soluciones basadas en cliente, como JQuery y su interminable colección de plugins, entre los que se pueden encontrar elementos de interfaz para prácticamente cualquier necesidad.

**Madurez del framework**

Aunque en sus principios la adopción de ASP.NET MVC framework podía suponer ciertos riesgos debido a su escasa madurez, hoy en día ya no es así.

Gracias a la disponibilidad del código fuente y su relativa simplicidad interna, los errores que pudieran detectarse en él podrían ser rápidamente subsanados.

La madurez también se hace patente en la cantidad y calidad de información disponible. ASP.NET MVC, cuenta con una comunidad de desarrolladores bastante entusiasta y cada vez mayor.

**Los beneficios de ASP.NET MVC**

Las ventajas de la arquitectura MVC se manifiestan al adoptar este framework:

- La separación de aspectos impuesta por el patrón MVC obligará a tener un código más limpio y estructurado, independizando totalmente la interfaz de la lógica de navegación y, por supuesto, de la de negocio.
- De la misma forma, esta división facilita el trabajo en equipo, pues permite el avance en paralelo en las distintas capas.
- Si se usan pruebas unitarias, o si se opta por utilizar una metodología de desarrollo guiado por pruebas (TDD), ASP.NET MVC será muy útil. La separación de aspectos citada anteriormente facilita la creación de pruebas específicas para los componentes de cada capa de forma independiente, así como el uso de técnicas avanzadas (*mocking*, inyección de dependencias).
- Las friendly URLS, o direcciones amigables, es un beneficio directo del uso del framework MVC de Microsoft. Esto es mérito del tratamiento del Routing. Si se adopta esta tecnología por defecto tendremos esta ventaja, más otras como SEO, REST, claridad en direcciones, etc.

- Al final, el software será de más fácil mantenimiento; el hecho de que los componentes estén separados y bien estructurados simplificará dicha tarea.
- El conjunto de convenciones en cuanto a la estructura de proyectos y de nombrado y disposición de elementos facilitará el desarrollo una vez que sean asimiladas.

**El tipo de sistema**

A la hora de plantearse adoptar MVC es imprescindible tener en cuenta el tipo de proyecto en el que se va a trabajar. No es lo mismo desarrollar un sitio web colaborativo destinado a un gran número de usuarios, como Facebook, donde el control fino sobre la entrada y salida es crucial para asegurar aspectos como la escalabilidad, cumplimiento de estándares, o accesibilidad, que crear una aplicación de gestión que utilizará un grupo relativamente reducido de usuarios desde una intranet corporativa.

En ambos casos se trata de crear sistemas web, pero los objetivos, requisitos y restricciones a considerar son muy diferentes.

Para el primer caso, ASP.NET MVC es una magnífica opción. La simplicidad de la arquitectura MVC hace que el ciclo de vida de las páginas de este framework sea mucho más sencillo que el de los Webforms, y la ausencia de automatismos y persistencia de estado aligera en gran medida el peso y complejidad de las páginas, lo cual redundará muy positivamente en el rendimiento del sistema.

Por último, la facilidad de introducción de funcionalidades Ajax hace de ASP.NET MVC una opción muy apropiada para estos sistemas con estilo “2.0”.

El funcionamiento de ASP.NET MVC es en muchos aspectos bastante más sencillo que la tecnología Webforms.

La inyección de dependencias (DI, *Dependency Injection*) es una técnica que permite realizar aplicaciones cuyos componentes se encuentran muy desacoplados entre sí, lo que flexibiliza el diseño y, por ejemplo, facilita la realización de pruebas unitarias. Se trata de un patrón muy fácilmente implementable y que puede aportar muchos beneficios.

La inversión de control (IoC, *Inversion of Control*), relacionada con el concepto anterior, permite modificar en determinadas circunstancias el flujo de ejecución, cediendo la responsabilidad de realizar tareas específicas a componentes especializados externos a la

aplicación. El caso más típico es el uso de software IoC (llamados contenedores) para delegar a ellos la instanciación de clases, lo que permite modificar la implementación concreta de la clase de forma externa, siempre que se cumpla un contrato (normalmente, una interfaz).

También es habitual encontrar referencias a técnicas para facilitar las pruebas unitarias, como la construcción de *mocks*, *stubs*, o *fakes*, que conceptualmente no son más que objetos falsos cuyo comportamiento y datos se preconfiguran explícitamente para poder probar de forma más sencilla los métodos.

**Ajax con MVC**

De hecho, ASP.NET MVC funciona muy bien con librerías de scripting tales como JQuery.

La limpieza de la filosofía MVC hace posible que sea realmente sencillo realizar desde el cliente llamadas a los controladores mediante scripting con el objeto de obtener datos, actualizar porciones de contenido de la página con el marcado de la vista correspondiente, o, en definitiva, interactuar con el servidor.

Algunas ventajas de MVC con JQuery:

- JQuery y algunos de sus plugins vienen incluidos de serie en las plantillas de proyectos ASP.NET MVC, facilitando su uso desde las aplicaciones.
- Se mejoró la integración del Visual Studio agregando intellisense a la escritura de código.
- JQuery se valoriza más como script de implementación en el cliente al contar con el respaldo de Microsoft.

### 8.6.1 - Tipo de tecnologías para implementar el modelo

La adopción del modelo es bien sencilla: **ASP.NET MVC es completamente independiente del modelo**, es el desarrollador el que elige cómo implementarlo siempre que se respeten los límites impuestos por el patrón MVC.

Teniendo en cuenta las responsabilidades del modelo, habitualmente se encontrarán en esta capa:

- Entidades de negocio.

- Clases de lógica empresarial, que implementan los procesos y reglas de negocio.
- Mecanismos de acceso a datos, por ejemplo usando directamente las clases de ADO.NET, o mejor aún, un ORM que aísle de la persistencia, como Linq2Sql, Entity framework o NHibernate.

### 8.6.2 - Tipos de tecnologías para utilizar en las vistas

El objetivo de las vistas es componer el interfaz de usuario y los mecanismos de interacción con el usuario. Lo habitual, por tanto, será utilizar algún **lenguaje de marcado, como XHTML ó HTML, CSS y Javascript, complementado con bloques de código de servidor** que se ejecutará en el momento de *renderizar* la página.

También se utiliza la tecnología Ajax para enviar u obtener información desde el servidor, siempre mediante llamadas a acciones definidas en el controlador, que permitirán crear interfaces más dinámicas.

Otra posibilidad interesante que aprovecha y demuestra la flexibilidad de la arquitectura de ASP.NET MVC framework, es la utilización de motores de vistas distintos al estándar. MVC3 y MVC4 soportan el motor aspx e incluye **Razor** que tiene un código mucho más simple y legible.

Razor minimiza el número de pulsaciones de teclas necesarias para crear una vista, y permite un flujo de desarrollo rápido y fluido. Esto permite una sintaxis más compacta y expresiva convirtiéndola en limpia, rápida y divertida de escribir.

Existen multitud de motores ligeros (NHaml, Spark, Brail, NVelocity…), cada uno con su propio lenguaje de marcas y convenciones, que permiten la definición de vistas a partir de plantillas.

**Lo más importante: ASP.NET MVC Framework es software libre.** A principios de abril de 2009 se comenzó a distribuir oficialmente el código fuente de ASP.NET MVC con licencia MS-PL (Microsoft Public License), un modelo de licencia aprobado por la OSI (Open Source Initiative) que permite el uso del software en aplicaciones comerciales y no comerciales.

Esto, entre otras ventajas, supuso la rápida adopción del framework MVC en las plataformas .NET alternativas, como el célebre proyecto Mono, con el que es totalmente

compatible. Incluso MonoDevelop dispone de herramientas y ayudas específicas para aplicaciones ASP.NET MVC.

El código fuente de ASP.NET MVC se publica en Codeplex, prácticamente de forma simultánea a las distintas revisiones que son lanzadas.

## 8. 7 - GIT

Git es un sistema de control de código fuente distribuido. A diferencia de TFS, VSS o SVN que son centralizados, en Git cada desarrollador tiene su propia copia entera del repositorio *en local*. Los cambios son propagados entre repositorios locales y pueden (o no) sincronizarse con un repositorio central.

La diferencia fundamental con TFS ó VSS es que en esos está claro *quién es el repositorio* (el servidor). Eso **no** existe en Git. En Git *cada usuario es el repositorio* y se sincronizan cambios *entre repositorios (usuarios)*. Opcionalmente puede usarse *un repositorio central* pero no es obligatorio.

Los cambios en Git *se propagan* a través de estas operaciones:

- **commit:** Envia los datos del *working directory* al repositorio *local.*
- **push:** Sincroniza los cambios del repositorio *local* a otro repositorio remoto.
- **fetch:** Sincroniza los cambios de un repositorio remoto al repositorio *local*.
- **pull:** Sincroniza los cambios de un repositorio remoto al working directory.
- **checkout:** Actualiza el *working directory* con los datos del repositorio *local*.

La siguiente imagen lo clarifica muy bien:

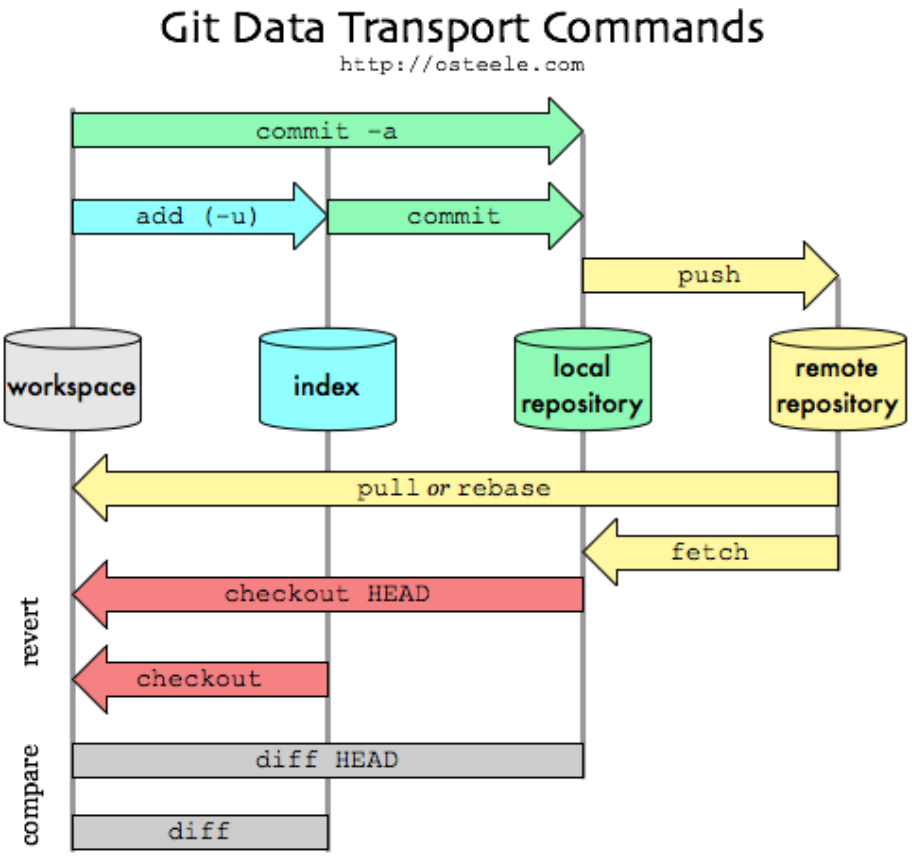


**Fig. 8.4: Estructura Repositorio GIT**

Operaciones de Git (imagen original de Oliver Steele en http://www.gitready.com/beginner/2009/01/21/pushing-and-pulling.html).

# 9.0 – EL SISTEMA DE SOFTWARE A DESARROLLAR

Finalmente llegué a una implementación de la solución buscada que permite mostrar en detalle las características de la arquitectura elegida. En este capítulo expongo la puesta en funcionamiento del sistema no sólo en forma de prueba en desarrollo, sino en forma concreta en el ámbito de los usuarios. Todo esto sin dar de baja a los sistemas anteriores, que siguen funcionando normalmente y en paralelo con el nuevo desarrollo, lo cual era uno de los objetivos planteados al comienzo del trabajo.

## 9.1 - Estructura de la aplicación web en ASP.NET MVC

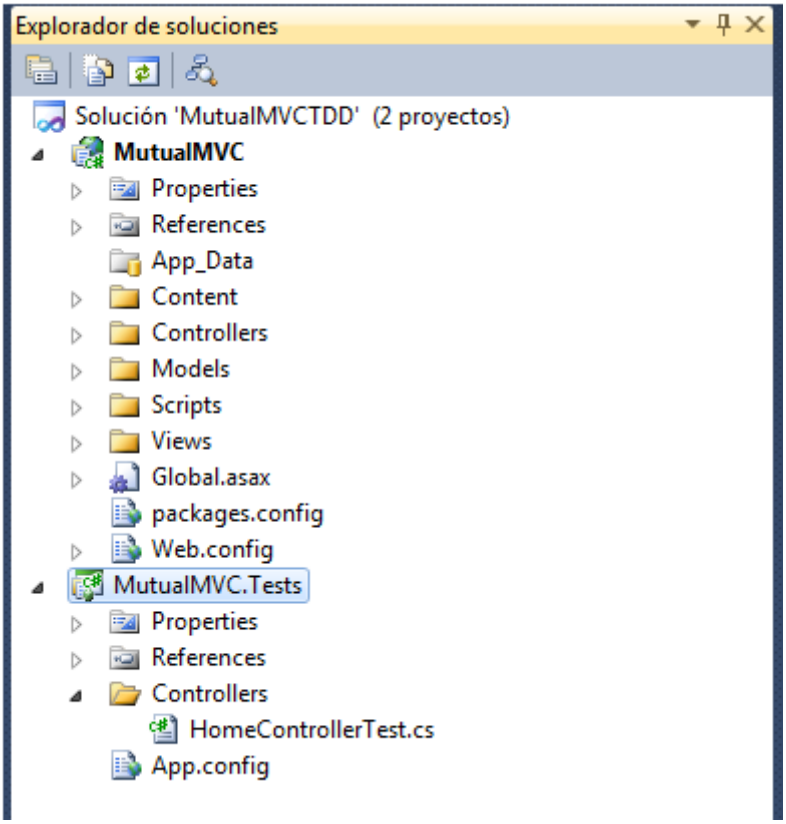


**Fig. 9.2: Estructura de un proyecto dentro de la solución**

- **/Content**: esta carpeta está pensada para almacenar los recursos de la aplicación, como imágenes, estilos (CSS), etcétera.
- **/Controllers**: almacena todos los controladores del proyecto.
- **/Models:** todas las clases relacionadas con la capa de datos.
- **/Scripts**: contiene los scripts que se ejecutan del lado del cliente.

- **/Views**: todas las vistas de todos los controladores. Dentro de la misma hay una carpeta por controlador.
- **/App_Data**: se pueden almacenar los archivos de datos.

## 9.2 – Principios bases del diseño a seguir

Existen al momento de diseñar un software una serie de principios fundamentales que maximizan la facilidad de mantenimiento y la calidad y minimizan en contraposición la baja productividad, favoreciendo una arquitectura que permita la reusabilidad y la extensión en el tiempo. Estos principios son:

**Cohesión:** los componentes (módulos, clases, vistas, acceso a datos, etc.) tienen una responsabilidad específica, de manera que ningún componente cumpla funciones que se solapen con otras partes, ni ninguna función básica deje de estar cubierta.

Todos los componentes lógicos tienen una responsabilidad dentro del sistema. Se debe pensar pues en el nivel de cohesión de una aplicación no como una suma de componentes, sino como el conjunto de los mismos.

**Acoplamiento:** en la ingeniería del software, el acoplamiento entre módulos, clases o cualquier entidad lógica es el grado de dependencia entre ellos. El factor fundamental es minimizar los puntos de interacción y cuanto más estándar sea la relación entre entidades, mayor reaprovechamiento se podrá hacer de ellas.

**Encapsulamiento:** consiste en abstraer determinadas funciones para que, además de ser reutilizables, los diseñadores puedan sacar el máximo provecho de un componente, sin tener que saber la complejidad de su diseño interno.

**Principios SOLID:**

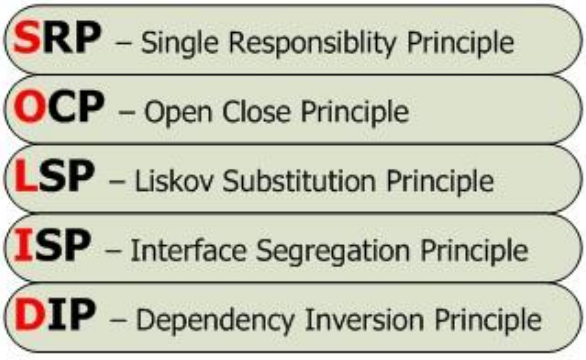


**Fig.9.1: Principios SOLID**

Los principios SOLID son una guía a seguir durante la fase de desarrollo para facilitar el mantenimiento de las aplicaciones y tratar de eliminar el impacto de las inevitables modificaciones que éstas sufren durante su ciclo de vida, además de facilitar el uso de las unidades de testeo, entre otras ventajas.

Son principios de diseño básicos para poder crear aplicaciones robustas y sostenibles, creados por **Robert C. Martin**, y es un acrónimo de acrónimos, es decir, una combinación de varios principios ya existentes.

- **S-RP:** Single Responsability Principle: especifica que una clase (o un método) solamente tiene que tener un propósito concreto, es decir, tener una única responsabilidad o un motivo por el que cambiar. Esto se reduce en clases más pequeñas y más fáciles de entender.
- **O-CP:** Open Closed Principle: se basa en delegar la responsabilidad a la clase. Si existen una serie de acciones que dependen del subtipo de una clase, es más fácil que esa funcionalidad esté implícita en la clase, y que las subclases las reimplementen (o las especifiquen) mediante polimorfismo, una de las características más comunes de los lenguajes orientados a objetos. Esto se resume en que una clase debe estar abierta a extensión y cerrada a cambios.
- **L-SP:** Liskov Substitution Principle: explica que todas las clases que hereden de una superclase, por decirlo así, no deben replicar funcionalidad ya implementada en esta clase, con lo cual la clase base se deberá poder sustituir por las subclases en cualquier región del código.
- **I-SP:** Interface Segregation Principle: propone dividir las interfaces en algo que tenga más sentido, en vez de dejar operaciones de la interfaz por implementar, en cierta manera sería una aplicación del SRP a las interfaces, teniendo en cuenta que una clase puede implementar varias interfaces de manera simultánea. No se debe forzar a clientes a implementar métodos no necesarios.
- **D-IP:** Dependency Inversion Principle: permite conseguir un desacoplamiento de las clases en el código, de tal manera que si una clase emplea otras clases, la inicialización de los objetos venga dada desde fuera. Además, si se puede

sustituir las clases por interfaces, se puede extender a otro objeto que implemente la interfaz, sin variar el parámetro.

También hay otros principios a seguir, los más importantes son:

**No repetirse (DRY)**: una funcionalidad específica se debe implementar en un único componente; esta misma funcionalidad no debe estar implementada en otros componentes.

**Minimizar el diseño de arriba hacia abajo**. (Diseñar solamente lo que es necesario, no realizar sobreingeniería, y evitar el efecto **YAGNI** (En inglés-*slang*: ***Y**ou **A**in´t **G**onna **N**eed **I**t*).

Estos principios llevados a la codificación se pueden ver en esta parte del código donde se escribe la interface del repositorio de cliente:

```
public interface IMutualRepository
{
    Soc_clu1 CreateSoc_clu1(Soc_clu1 soc_clu1ToCreate);
    void DeleteSoc_clu1(Soc_clu1 soc_clu1ToDelete);
    Soc_clu1 EditSoc_clu1(Soc_clu1 soc_clu1ToEdit);
    Soc_clu1 GetSoc_clu1(int Id);
    IEnumerable<Soc_clu1> ListSoc_clu1(string socioSearch);
}
```

Luego se desarrolla la clase que implementa esta interface, que utiliza el contexto de datos que provee Entity Framework sobre MSSQL:

```
public class MutualRepository : IMutualRepository
{
    private MVC4MutualContext context = new MVC4MutualContext();
    // ...
    public IEnumerable<Soc_clu1> ListSoc_clu1(string socioSearch)
    {
     if (!string.IsNullOrEmpty(socioSearch))
     {
         List<Soc_clu1> lp = context.soc_clu1.Where(x => x.nom_soc.ToUpper()
             .Contains(socioSearch.ToUpper())).ToList();
         return lp;
     }else {
        List<Soc_clu1> lp = context.soc_clu1.ToList();
         return lp;
        }
    }
    // GET: /Socios/Details/5
    public Soc_clu1 GetSoc_clu1(int Id)
    {
         Soc_clu1 soc_clu_1 =  context.soc_clu1.Single(x => x.soc_clu1ID == Id);
         return soc_clu_1;
    }
```

Una capa intermedia entre el controlador y el repositorio se llama service y está definida a través de la interface:

```
namespace MVC4Mutual.Areas.Social.Models
{
    public interface IMutualService
    {
        bool CreateSoc_clu1(Soc_clu1 soc_clu1ToCreate);
        bool DeleteSoc_clu1(Soc_clu1 soc_clu1ToDelete);
        bool EditSoc_clu1(Soc_clu1 soc_clu1ToEdit);
        Soc_clu1 GetSoc_clu1(int Id);
        IEnumerable<Soc_clu1> ListSoc_clu1(string socioSearch);
    }
```

Cuando se quiere implementar esta interface en una clase concreta se ve en el siguiente código como se usa la interface del repositorio para la utilización de los datos:

```
public partial class MutualService:IMutualService
{
    private IValidationDictionary _validationDictionary;
    private IMutualRepository _repository;

    public MutualService(IValidationDictionary validationDictionary)
        : this(validationDictionary, new MutualRepository())
    { }

    public MutualService(IValidationDictionary validationDictionary, IMutualRepository repository)
    {
        _validationDictionary = validationDictionary;
        _repository = repository;
    }

    # region  Soc_clu1
    // Validacion del socio
    public bool ValidateSoc_clu1(Soc_clu1 soc_clu_1)...
    public bool CreateSoc_clu1(Soc_clu1 soc_clu_1)...
    public bool DeleteSoc_clu1(Soc_clu1 soc_clu_1)...
    public bool EditSoc_clu1(Soc_clu1 soc_clu_1)...
    public Soc_clu1 GetSoc_clu1(int Id)...
    public IEnumerable<Soc_clu1> ListSoc_clu1(string socioSearch)...
    # endregion
```

En el constructor de la clase se usa una instancia del IMutualRepository() de manera que será muy fácil luego cambiar la clase a usar, ya que ésta puede ser una implementación que use otro tipo de base de datos. Lo importante será que cumpla con el contrato dado por la definición de la interface IMutualRepository().

Gráficamente podemos ver las distintas capas lógicas, sus interrelaciones y dependencias:

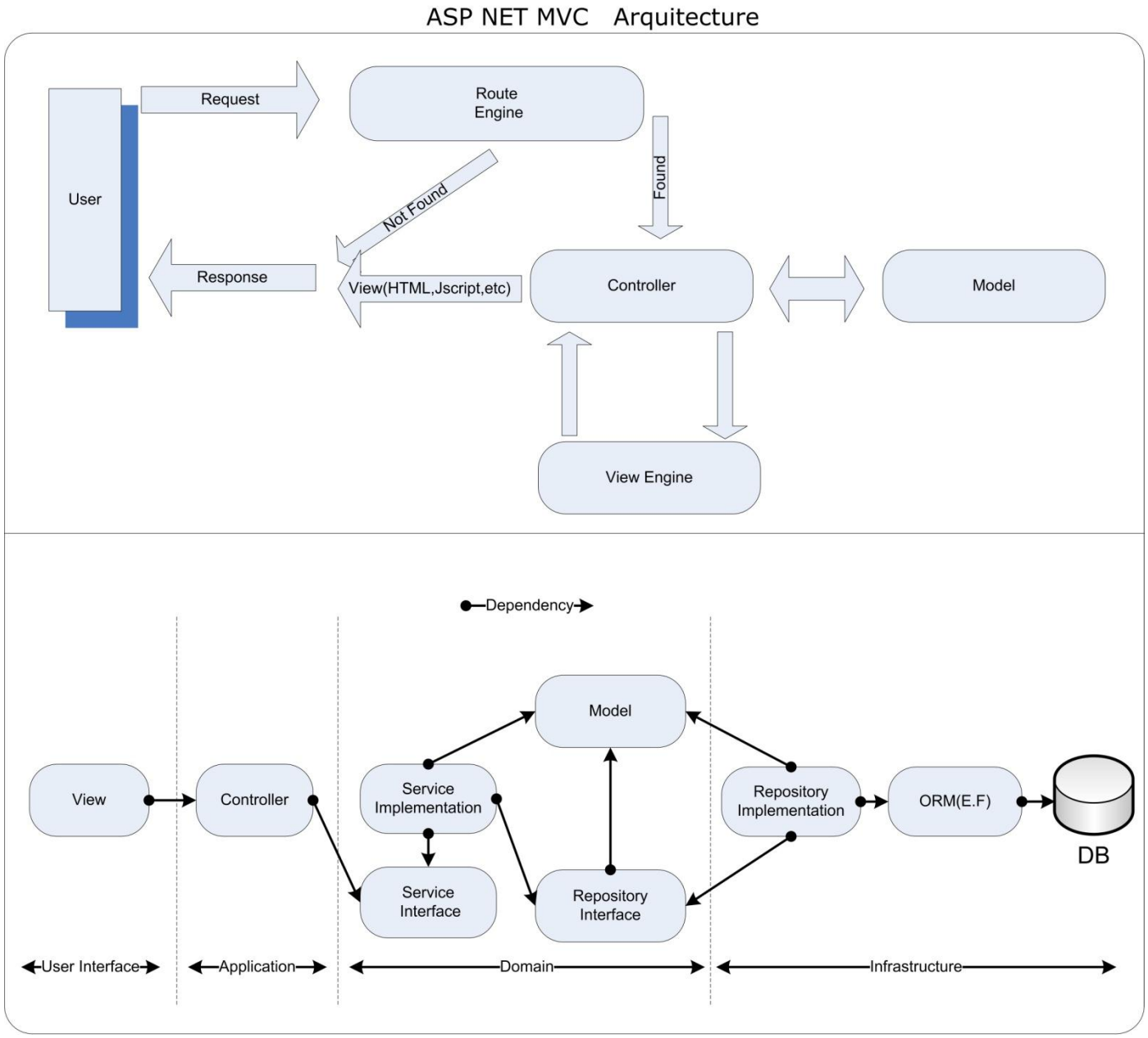


**Fig.9.2: ASP NET MVC Arquitectura**

Dividir una aplicación en capas separadas que desempeñan diferentes roles y funcionalidades ayuda a mejorar el mantenimiento del código, permite diferentes tipos de despliegue y sobre todo proporciona una clara ubicación y delimitación de dónde debe estar cada tipo de componente funcional e incluso cada tipo de tecnología.

Beneficios del uso de capas:

- Mantener los sistemas o introducir modificaciones es mucho más fácil porque las funciones están localizadas y además al estar las capas poco acopladas entre ellas y con alta cohesión interna es posible variar de una forma sencilla diferentes combinaciones y/o implementaciones de capas.
- Permitir que otras soluciones puedan reutilizar las funcionalidades expuestas por las diferentes capas.
- Los desarrollos distribuidos son más sencillos de implementar si se ha subdividido la aplicación previamente en diferentes capas lógicas.
- Si se hace coincidir las capas lógicas con diferentes niveles físicos (un servidor de datos, un servidor web, etc.), se mejora la escalabilidad del sistema.
- Permitir implementar pruebas sobre los componentes en forma individual.

## 9.3 – Cuestiones prácticas de las aplicaciones web

La web trajo beneficios y dificultades. Las aplicaciones que corren sobre ella deben lograr maximizar los beneficios y superar las dificultades. Es lo que este trabajo se propuso lograr, además de potenciar las siguientes pautas:

- Facilidad de comunicación.
- Integración de diferentes lugares físicos.
- Integración de diferentes lenguajes.
- Confiabilidad (seguridad en los informes).
- Rendimiento.
- Productividad en el desarrollo.

La web ha evolucionado de ser una enorme biblioteca de solo lectura a ser una plataforma para hacer negocios. Las tecnologías web siguen evolucionando y permiten mejorar la experiencia del usuario y usabilidad. Las *aplicaciones web* son un escalón más en este proceso.

Los siguientes puntos son criterios prácticos a tener en cuenta para que las diferencias entre una aplicación de escritorio y una aplicación web se minimicen. El diseño de una aplicación web (no un sitio web), es más exitoso si se lo toma como una aplicación de escritorio. Sin embargo, se deben entender claramente las limitaciones del browser y tenerlas en cuenta en el proceso:

- **Diferenciación entre web site y aplicación:** una aplicación web por lo general es lanzada desde el website de la empresa. Para distinguir la aplicación del sitio web, la transición de uno a otro debe ser clara. El usuario tiene que sentir que no está más en la web.
- **Abrir la aplicación en pantalla completa:** el hecho de abrir la aplicación en pantalla completa hace que el usuario sienta que está en una aplicación de escritorio. Pantalla completa no significa que hay que tapar los elementos del sistema operativo, sino que sea una nueva ventana y totalmente maximizada.
- **Esconder los menús y controles del browser:** el usuario percibe el browser como una ventana a la web. En el contexto de una aplicación el objetivo es que el usuario vea la ventana como una aplicación independiente del browser.
- **No usar múltiples tabs o ventanas del browser:** la recomendación es usar una única ventana de manera tal que el usuario no se pierda entre ventanas abiertas de la misma aplicación.
- **Utilizar elementos web conocidos:** los usuarios utilizan el conocimiento adquirido en la web. Es bueno construir sobre la base de ese conocimiento, usando elementos web como el hyperlink, áreas clickeables, etc.
- **Utilizar tamaños constantes en fuentes, tablas y otros elementos visuales:** para hacer consistente el uso de la aplicación desde diferentes máquinas y diferentes dimensiones.
- **Atajos de teclado:** permitir ejecutar acciones con atajos de teclado para usuarios expertos.
- **Utilizar overlays:** el *overlay* es un efecto que simula un popup pero sin abrir otra ventana. Se maneja como un panel y se muestra en la página con una

transparencia que deja ver lo que hay atrás. Esto se puede usar para funciones como *Login* o *Help* para que el usuario sienta "que sus datos no desaparecieron".

- **Animaciones y tiempo de carga:** informar al usuario el progreso mientras se cargan los datos.
- **Limitar el uso de animaciones/arte:** el arte puede ser placentero a los ojos, pero en la mayoría de los casos distrae al usuario de la aplicación. El diseño visual es importante pero en una aplicación lo más importante es la funcionalidad. El diseño debe estar supeditado a la funcionalidad.

## 9.4 - Sitio web que tenga este menú e implemente estos servicios:

### A - Sistema de socios

```
MENU - ACTUAL
"VUELCOS"
     "ABM SOCIOS"
"CONSULTAS"
      "PADRON SOCIOS"
"INFORMES"
      "PADRON SOCIOS"        do form i_pasoc1
      "CUOTAS SOCIOS"                              do form i_cuotas1
      "TOTALES P/CATEGORIA"                        do form i_tcat1
      "CUOTAS VENCIDAS"                            do form i_atras1
      "\-"
      "CUOTAS TENIS"                               do form i_cuoten1
      "CUOTAS FUTBOL"
      "CUOTAS BASKET"
      "CUOTAS GIMNASIA"
"IMPRESIONES"
     "ENCABEZADO LIBRO FOLIADO"                    do form i_encfol1
     "LIBRO FOLIADO IMPRESIÓN"                     do form i_libfol1
"A.B.M AUXILIARES"
     "CUOTAS ATRASADAS"
     "BORRADO CUOTAS ATRASADAS"
"PARAMETROS"
     "PARAMETROS GRALES."
"F I N"
```

**Servicios Web Rest XML o JSON:**
1 - Padrón de socios
2 – Socio datos

### B - Sistema gestión financiera - Menú:

```
* ******************************************************
* * Menu Definition
```

```
* ********************************************************
"AHORRO VARIABLE"
   "DEBITO/CREDITO"                  do form aho_vue_doc1
   "CORRRECIONES"
      "ACTUALIZACIÓN NO CONFORME"    do form aho_act_ncf1
      "MOVIMIENTOS CONFORME"         do form aho_mod_cnf1
      "MOVIMIENTOS NO CONFORME"      do form aho_mod_ncf1
   "CALCULO INTERESES"               do form aho_cal_int1
   "CONSULTAS E IMPRESIONES"
      "RESUMEN DE CUENTAS"           do form aho_res_cta1
      "MOVIMIENTOS DIARIOS"          do form aho_mov_dia1
      "SALDOS CUENTAS"               do form p_salaho1
      "INTERESES Y SALDOS"           do form p_sal_intaho1
      "IMPRESIÓN CHEQUES"            do form imp_che1
      "\-"
   "CUENTAS SOCIOS"
      "CAJAS DE AHORRO"              do form aho_abm_cta1
      "ALTA DE SOCIOS"               do form abm_soc1
   "\-"
   "RETENCIONES DGI"
      "NUMERACIÓN RETENCIÓN CHEQUES" do form aho_n_rche1b
      "\-"
      "IMPRESION RETENCIONES"        do form aho_i_rche1b
      "LISTADO RETENCIONES"          do form aho_i_lrche1b
"AHORRO A TERMINO"
      "AHORRO A TERMINO-DOCUMENTOS"  do form pfi_vue_doc1
      "\-"
      "INFORMES"                     do form pfi_i_lispfi1
      "INTERESES DEVENGADOS"         do form pfi_i_lisint1
"PRESTAMOS"
   "PRESTAMOS-DOCUMENTOS"            do form pre_vue_doc1
   "RECIBO PRESTAMO"                 do form pre_vue_can1
   "\-"
   "CALCULO CUOTAS"                  do form pre_cal_cuo1
   "\-"
   "INFORMES PRESTAMOS"
      "PRESTAMOS LEGAJO"             do form pre_i_docemi1
      "PRESTAMOS ACTIVOS"            do form pre_i_lisact1
      "CANCELACIONES LEGAJO"         do form pre_i_liscan1
   "INFORMES A TERCEROS"
      "SEGURO DE VIDA"               do form pre_i_sdv1
"BANCOS"
   "BOLETAS DE DEPOSITOS"            do form bol_dep1
   "\-"
   "CARTERA DE CHEQUES"              do form ban_car_che1
   "\-"
   "A.B.M. BANCOS"                   do form abm_bco1
   "A.B.M. CODIGOS POSTALES"         do form abm_c_p1
   "A.B.M DIAS CANJE"              do form abm_cje1
"C A J A"
   "RECIBOS"                       do form pre_vue_rec1
```

```
   "DEBITOS/CREDITOS"              do form p_debcre1
   "\-"
   "INFORME TESORERIA"             do form caj_vue_leg1
   "PLAN DE CUENTAS"               do form caj_leg_cta1
   "INFORMES"
      "INFORME LEGAJO"             do form caj_imp_leg1
      "INFORME CONTABLE"           do form caj_imp_asi1
"PARAMETROS"
   "PARAMETROS GRALES."
      "PARAMETROS EMPRESA"         do form par_emp1
   "DIAS FERIADOS"                 do form par_dia_fer1
   "PARAMETRO MOVIMIENTOS"         do form par_mov_ayu1
   "FECHA LEGAJO"                  do form par_fec_leg1
   "IMPRESORA PREDETERMINADA"      do form imp_pre1
   "\-"
   "A.B.M. SOCIOS"                 do form abm_soc1
   "A.B.M. ENTIDAD"              do form abm_ent1
   "\-"
   "BACKUP"                        do form adm_zip1

"FIN"
```

## 9.5 - Planificación del desarrollo

En este título se describe cómo se planificó el desarrollo del sistema utilizando la metodología XP (ver la sección 8.1) para construir de forma ágil un sistema de información que cumpla los requisitos del cliente y le agregue valor a su negocio.

Las premisas básicas y esenciales de la "filosofía XP" son:

- Hacer código rápido y testeable en primer lugar. Si queda tiempo, realizar los diagramas y documentación.
- Llevar código sencillo de menor a mayor, para ir robusteciéndolo.
- Mantener las cosas simples. Los "adornos" si sobra tiempo.
- Atacar lo más importante ahora.

### 9.5.1 – Planeamiento

La mayor diferencia entre las metodologías clásicas y XP (Programación Extrema), es la forma en que se expresan los requisitos de negocio. En lugar de documentos de Word, son ejemplos que fueron detallados por el *cliente/usuario*.

Para ello se hacen las reuniones con el *cliente/usuario* (aquí representado por la gerencia administrativa) tomando notas del funcionamiento actual y las mejoras que se

necesitan. Reuniones que sirven para escribir, priorizar y ordenar las historias de usuarios, que seguirán durante todo el transcurso del desarrollo del proyecto.

La idea es hacer emerger los mejores diseños y con las óptimas arquitecturas del análisis funcional de cada historia de usuario.

**Proceso XP**

El ciclo de vida de los proyectos XP puede verse como una sucesión de definiciones por parte del *cliente/usuario* y una aceptación y continuo desarrollo por parte de los programadores. Esto se trata que ocurra en el más breve lapso de tiempo procurando que el cliente pueda corregir y ajustar el rumbo. Este ciclo atraviesa por los siguientes pasos:

1- El cliente decide el valor de negocio que quiere implementar.

2- Se estima el esfuerzo necesario.

3- El cliente selecciona qué construir.

4- Se construye la selección.

**Historias de usuarios**

Una historia de usuario posee similitudes con un caso de uso, salvando ciertas distancias. Por hacer una correspondencia entre historias de usuario y casos de uso, se puede decir que el título de la historia se corresponde con el del caso de uso tradicional. Sin embargo, la historia no pretende definir el requisito. Escribir una definición formal incurre en el peligro de la imprecisión y la malinterpretación, mientras que contarlo con ejemplos ilustrativos, transmite la idea sin complicaciones. Cada historia de usuario contiene una lista de ejemplos que cuentan lo que el cliente quiere, con total claridad y ninguna ambigüedad. El enunciado de una historia es tan solo una frase en lenguaje humano, de alrededor de cinco palabras, que resume qué es lo que hay que hacer.

Quiero agregar el pensamiento de Carlos BLE JURADO **(5)** que a continuación transcribo:

*"Saltar de casos de uso a crear un diagrama de clases modelando entidades, es en mi opinión, peligroso cuanto menos. Los diagramas nos pueden ayudar a observar el problema desde una perspectiva global, de manera que nos aproximamos al dominio del cliente de una manera más intuitiva. Pueden ayudarnos a comprender el dominio hasta*

*que llegamos a ser capaces de formular ejemplos concretos. En cambio, representar elementos que formarán parte del código fuente mediante diagramas, es una fuente de problemas. Traducir diagramas en código fuente, es decir el modelado, es en cierto modo opuesto a lo que se expone en este libro. Para mí,* ***la única utilidad que tiene el UML es la de representar mediante un diagrama de clases, código fuente existente****. Es decir, utilizo herramientas que autogeneran diagramas de clases, a partir de código, para poder echar un vistazo a las entidades de manera global pero nunca hago un diagrama de clases antes de programar. Mis entidades emergen a base de construir el código conforme a ejemplos".*

### 9.5.2 - Primera versión

Story Point (SP) se define como la cantidad de trabajo que puede realizar un programador ideal en una semana trabajando 20 horas por semana.

En base al menú anterior y sus formularios de ingresos, también se suman los conceptos del cliente y usuarios, se escriben las primeras historias para el sistema, empezando por el padrón de socios. Escribir estas historias significa interactuar con los usuarios y operadores del sistema, de manera que fue producto de diálogo e intercambio de ideas con los empleados de la mutual y socios de la misma.

Luego, se estima el tiempo que llevaría completar cada historia medida en Story Point (SP). Como una iteración no debe durar más de 3 semanas, una historia tampoco debe hacerlo. Para poder estimar la cantidad de Story Point se tuvieron que utilizar criterios que se refieren más a la experiencia que a un sistema de medidas preestablecido.

El desarrollo basado en métricas es más adecuado para el trabajo simple repetitivo, con bajos requisitos de conocimiento y rendimientos fácilmente medibles, exactamente lo contrario al desarrollo de software.

Una vez que se considera que las historias con las que se cuenta son suficientes como para armar la primera versión del sistema, se le asigna a cada una un riesgo (alto, medio o bajo) en función de la dificultad que se tendría para implementarla de acuerdo a la experiencia.

A su vez, el cliente les asignó a las historias un valor (alto, medio o bajo) dependiendo de lo crítica que es la misma para el funcionamiento de su negocio; las

características sin las cuales el sistema no puede funcionar adquieren el valor alto, mientras que aquellas que sería bueno tener, pero no son fundamentales, adquieren el valor bajo.

El siguiente es el primer grupo de historias escritas, con sus estimaciones y valores.

Cada historia tiene un identificador (número), un título y una descripción del requerimiento.

**Historia 1: ABM socios**

El sistema debe permitir cargar, modificar y eliminar un socio definido por su tipo y número de documento.

El sistema debe otorgar número de socio y categoría, debe guardar la fecha de ingreso y asignar qué tipo de cuota societaria paga o no. El número de socio tiene la forma:{NÚMERO} que es un valor secuencial independiente para cada socio/persona:

**Los datos a cargar para un socio son:**

Número de socio.
Categoría: las categorías de socios (menor, mayor, vitalicio, etc.).
Condición: paga o no, y valor de la cuota.
Nombre y apellido: identifica a la persona.
Sexo: femenino /masculino.
Estado civil: casado/soltero/etc.
Domicilio.
Localidad.
Teléfono.
Fecha de Ingreso: la fecha de ingreso como socio permite calcular la antigüedad.
Fecha de nacimiento: permite referencias sociales.
Activo/Autorizado: condición para operar con la ayuda económica.

**Criterios de aceptación:**

Quiero dar de alta un socio.
Quiero controlar si ya estuvo cargado a través de un número de documento.
Quiero controlar que tenga número de CUIT/CUIL válido.
Debo tener los datos fundamentales (a definir) para dar de alta.
Estimación cliente/usuario: Alto

Riesgo: Alto

Story Points: 3

**Historia 2: ABM Categoría**

Las distintas categorías en que se clasifica la masa societaria.

Estimación cliente: Bajo

Riesgo: Bajo

Story Points: 1

**Historia 3: ABM Condición**

La condición se refiere a cuánto es la cuota social, el socio puede no pagar y es independiente de la categoría. De acuerdo a la condición se valoriza el importe.

Estimación cliente: Alta

Riesgo: Medio

Story Points: 0,5

**Historia 4: ABM Disciplina**

Las distintas disciplinas que se practican en el club (fútbol, tenis, basket, etc.). Quienes las practican deben ser identificados y se les debe aplicar o no una cuota contribución por esa disciplina.

Estimación cliente: Medio

Riesgo: Medio

Story Points: 1

**Historia 5: Genera cuotas societarias**

El sistema debe emitir las cuotas de socio con los importes que le corresponde pagar a cada uno.

Estimación cliente: Medio

Riesgo: Bajo

Story Points: 0,5

**Historia 6: Informe padrón de socio y consultas**

Se debe poder consultar por el total de socios, por las distintas categorías, condición, disciplinas, por nombre, por número de socio, por documento, etc.

Estimación cliente: Medio

Riesgo: Medio

Story Points: 1

**Historia 7: Usuarios y permisos**

El sistema debe ser multiusuario. Se deben poder definir distintos roles. Los roles son un conjunto de privilegios. A cada usuario se le podrá asignar uno o más roles. Cada privilegio permite realizar una acción sobre el sistema.

Estimación cliente: Medio

Riesgo: Alto

Story Points: 2

**Historia 8: Sistema web y software libre**

El sistema debe ser web y debe estar implementado sobre IIS 6 o 7, MSSQL, debe poder correr sobre cualquier browser en cualquier sistema operativo del lado cliente.

Estimación cliente: Alto

Riesgo: Bajo

Story Points: 1

**Historias para la primera versión**

Se definió la velocidad del equipo de desarrollo en 5 SP. Se debe recordar que la velocidad es la cantidad de SP que debería esperar el cliente por iteración (cada 3 semanas).

Con esta información se determinó el ámbito, es decir, las historias que entrarían en la primera versión. La metodología establece que una versión no debería durar más de tres meses. Por lo tanto se decidió que la primera versión estaría compuesta por 2 iteraciones. Esto da un total de 10 SP.

La primera versión debe alcanzar para completar la estructura global del sistema. Es por esto que aunque se tenga la libertad de elegir las historias que se desean, éstas no deben sumar más de 10 SP, para que, de esta forma el objetivo de tener un esqueleto que pueda funcionar se cumpla.

Las historias de las restantes versiones podrán ser elegidas libremente. En general, se trata de incluir en cada versión las historias que tengan mayor valor para el usuario y (en lo posible) menor riesgo para los desarrolladores como se verá a continuación.

En la siguiente tabla de doble entrada se aprecia la distribución de las historias de acuerdo a las variables riesgo y valor. Las historias marcadas en **negrita** son las que se eligieron para la primera versión.

| Estimación *Cliente/usuario* | | Riesgo | | |
|---|---|---|---|---|
| | | Bajo | Medio | Alto |
| | Alto | **8**- | **3**- | **1**- |
| | Medio | **5**- | 4- 6- | 7- |
| | Bajo | 2- | | |

**Tabla 9.1: Comparación riesgo y valoración cliente primera versión**

**Iteraciones**

Se dividen las historias de la primera versión en dos iteraciones de no más de 5 SP cada una (que corresponde a la velocidad declarada) como se muestra a continuación:

**Historias de la primera iteración**

- Historia 8 sistema web 1 SP
- Historia 3 ABM condición 0,5 SP
- Historia 1 ABM socio 3 SP
- Historia 5 genera cuota societaria 0,5 SP

Total 1ª iteración 5,0 SP

**Historias de la segunda iteración**

- Historia 4 ABM disciplinas 1 SP
- Historia 6 informes padrón 1 SP
- Historia 7 usuarios y permisos 2 SP
- Historia 2 ABM categorías 1 SP

Total 2ª iteración 5,0 SP

### 9.5.3 - Segunda versión

**Historia 9: Informes libros foliados**

Los estatutos societarios de una mutual exigen tener anualmente pasado a actas el listado de socios.

Estimación cliente: Bajo

Riesgo: Bajo

Story Points: 1

**Historia 10: Historial socios**

Se desea llevar un control histórico de la relación de los socios con el club, en base al número de documento, para saber si ya fue socio, desde qué fecha y hasta qué fecha, en qué disciplina participó, desde y hasta, y los datos de cumplimiento de pago de la cuota societaria.

Estimación cliente: Alto

Riesgo: Alto

Story Points: 4

**Historia 11: Control de pago cuotas**

Llevar un control de pagos y reflejarlo en cada uno de los socios.

Estimación cliente: Medio

Riesgo: Medio

Story Points: 2

**Historia 12: Web services padrón socios**

Sobre HTTPS se debe poder obtener la consulta de la totalidad de socios con los datos de cada uno. De forma que se obtiene el recurso:

HTTP GET ➔ http://mutual/padron

HTTP GET ➔ http://mutual/socio/125

Donde 125 es el número de socio.

Estimación cliente: Medio

Riesgo: Alto

Story Points: 3

| Estimación *Cliente/usuario* | | Riesgo | | |
|---|---|---|---|---|
| | | Bajo | Medio | Alto |
| | Alto | | | **10**- |
| | Medio | | 11- | 12- |
| | Bajo | **9**- | | |

**Tabla 9.2: Comparación riesgo y valoración cliente segunda versión**

**Historias de la primera iteración**

- Historia 10 historial socios 4 SP
- Historia 9 informes libros foliados 1 SP

Total 1ª iteración 5,0 SP

**Historias de la segunda iteración**

- Historia 11 control de pagos 2 SP
- Historia 12 Web services 3 SP

Total 2ª iteración 5,0 SP

# 10.0 CONCLUSIONES

El software es una parte vital de los sistemas informáticos: permite explotar el hardware y obtener los máximos beneficios. El desarrollo de software siempre ha estado en un proceso de evolución, y como toda actividad dinámica, las primeras etapas siempre resultan difíciles debido a que no se tienen las herramientas adecuadas, ni los conocimientos, ni las estrategias para realizar bien los desarrollos. Es por eso que muchas veces los sistemas se convierten en partes inamovibles de las organizaciones, pero el tiempo les hace perder actualidad, quitando la posibilidad de utilizar las nuevas funcionalidades.

El hardware evoluciona constantemente, poniendo en el mercado las nuevas tecnologías a un costo accesible. Entonces, si es posible su incorporación como soporte del software, esto trae como consecuencia la necesidad de replantear y lograr nuevos beneficios actualizando los sistemas e incorporando nuevas herramientas y estrategias que están disponibles en correspondencia con el nuevo hardware. Obtener lo mejor de los sistemas actuales e incorporarlos en los nuevos, es parte fundamental de la reingeniería.

El problema que se planteó al principio de este trabajo fue encontrar la forma de establecer los lineamientos básicos esenciales para lograr que las aplicaciones existentes de una organización, que eran sistemas locales se pudieran llevar a una arquitectura distribuida con soporte sobre internet y ejecutada desde browsers independientes del sistema operativo.

Luego de un proceso de investigación, desarrollo y estudio de las distintas alternativas de comunicación, siempre manteniendo una línea tecnológica, llegué a la conclusión de la necesidad de establecer un sistema basado en aplicaciones web, junto con sus servicios.

El patrón arquitectónico se fijó en MVC que es natural para internet como medio de soporte, y además permite obtener un mayor rendimiento de las aplicaciones, en comparación a otras formas de hacer un desarrollo web.

Durante el estudio tuve que tomar determinaciones que fueron decisivas para la vida útil de las aplicaciones. De una correcta elección de la arquitectura y del software que la implementa junto con las herramientas aplicadas, dependerá que los sistemas continúen

prestando sus servicios, sin estar fuera de actualidad; es decir volverse obsoletos con rapidez. Un ejemplo destacable y definitorio fue la elección de REST en vez de SOAP.

Elegí también una metodología de desarrollo ágil que permite que pequeños grupos de trabajo se puedan concentrar en las tareas de construir software usando prácticas de fácil adopción y un entorno ordenado que facilita un mejor desempeño de los involucrados en la realización del proyecto. Opté por XP (Extreme Programing) porque tiene como ventajas: reducir el costo del proyecto, bajar los riesgos y aceptar los cambios derivados de los requisitos variables.

Como herramienta de desarrollo escogí ASP.NET MVC que si bien es promovida por Microsoft, de hecho se integra perfectamente con Visual Studio. Su utilización es bajo licencia libre, estando el código disponible para su modificación. Es muy importante que se mencione que cuenta con todas las herramientas de mayor éxito en la web.

Con este trabajo se realizaron contribuciones en dos direcciones: por un lado, al dar respuesta a una necesidad creciente de interacción desde cualquier punto geográfico tanto para **San Martín Mutual Social y Biblioteca** como para sistemas de terceros que deban intercambiar información con ésta; y por otro lado se dieron los primeros pasos para que todas esas horas de trabajo representadas por programaciones cliente/servidor que están corriendo actualmente, puedan mudar a nuevas formas con un mínimo de impedimentos, y en una dirección adecuada que mira a la persistencia en el tiempo. La evolución tecnológica pasa factura en forma implacable si se elije un camino equivocado.

# 11.0 GLOSARIO

**ADO.NET:** Servicio de datos.

**ANSI-IEEE:** Norma de estandarización eléctrica y electrónica.

**AJAX:** Asynchronous Java Script and XLM. (Java Script asíncrono y XML).

**API:** Application Programming Interface.

**ASMX:** Página web.

**ASP.NET:** Framework para aplicaciones web de Microsoft.

**C#:** Lenguaje de programación.

**COM:** Componente DLL para comunicación.

**CPU:** Unidad de procesamiento.

**CSS:** Cascading Style Sheet.

**DLL:** Componente compilado de un programa para su ejecución.

**DOM:** Document Object Model.

**GIF:** Graphics Interchange Format. (Formato gráfico).

**HTML:** Lenguaje de codificación para intercambio de información.

**HTTP:** Protocolo de comunicación en internet.

**IDE:** Integrated Development Environment. (Entorno desarrollo integrado).

**IIS:** Internet information server.

**IP:** Dirección de red.

**IT:** Tecnología de la información.

**JSON:** Java Script Object Notation. (Datos en Java Script).

**LAN:** Local Area Network (Red de área local).

**MIME:** Multipurpose Internet Mail Extensions.

**MSDN:** Red de desarrolladores de Microsoft.

**MS-DOS:** MicroSoft Disk Operating System.

**MSSQL:** Microsoft SQL Server,

**MVC:** Patrón de diseño Modelo Vista Controlador.

**OASIS:** Organization Advancement of Structured Information Standards.

**ODBC:** Standard de acceso a base de datos.

**ORM:** Object-Relating Mapping.

**PC:** Computadora de escritorio. Ordenador.

**PDF:** Portable Document Format.

**PNG:** Portable Network Graphics.

**QoS:** Quality of Service (Calidad de servicio).

**RAD:** Rapid Application Development.(Desarrollo rápido de aplicaciones).

**REST:** Representational State Transfer.

**SOA:** Arquitectura orientada a servicios.

**SOAP:** Especificación para implementación de servicios web.

**SQL 2005:** Software de bases de datos versión año 2005.

**SVN:** Sistema de control de versiones.

**TDD:** Desarrollo guiado por pruebas.

**URI:** Uniform Resource Identifier.

**URL:** Uniform Resource Locator.

**VB:** Visual Basic.

**VFP:** Visual FoxPro.

**VFP9:** Versión 9 del IDE Visual FoxPro.

**VFP9SP2:** Versión 9 del IDE Visual FoxPro Service Pack 2.

**VPN:** Red privada virtual.

**VS:** Visual Studio.

**WCF:** Windows Communication Foundation.

**WPF:** Window Presentation Foundation.

**WSDL:** Web Services Description Language. (Descriptor servicios web).

**WS-I:** Web Services Interoperability.

**XHTML:** HTML expresado como XML válido.

**XML:** Codificación de datos para utilizar sobre internet.

**XP:** eXtreme Programming. (Metodología de desarrollo software).

# 12.00 REFERENCIAS BIBLIOGRÁFICAS

**(1)** SOMMERVILLE Ian, "Ingeniería del Software", Séptima Edición, Pearson Educación SA, Madrid España 2005. Capitulo 2 "Sistemas socio-técnicos" Página 20.

**(2)** SOMMERVILLE Ian, "Ingeniería del Software", Séptima Edición, Pearson Educación SA, Madrid España 2005. Capitulo 21 "Evolución del software" Página 459.

**(3)** http://msdn.microsoft.com/es-es/library/w9fdtx28%28v=vs.80%29.aspx

**(4)** FIELDING Roy T., "Architectural Styles and the Design of Network-based Software Architectures", PhD thesis, UC Irvine, 2000, http://roy.gbiv.com/pubs/dissertation/top.htm

**(5)** BLE JURADO Carlos, "Diseño Ágil con TDD", iExpertos, 2010. Capitulo 3 "Desarrollo Dirigido por Tests de Aceptación (ATDD)". Página 65.

# 13.0 BIBLIOGRAFÍA

**[01**] SOMMERVILLE Ian, "Ingeniería del Software", Séptima Edición, Pearson Educación SA, Madrid España 2005

**[02**] PRESSMAN Roger, "Ingeniería del Software", Sexta Edición, Mc.Graw Hill Interamericana, Madrid España, 2008

**[03]** GAMMA Erich, HELM Richard, JOHNSON Ralph, VLISSIDES John, "Patrones de Diseño" , Pearson Educación SA, Madrid España, 2003

**[04**] PRIOLO Sebastián, "Métodos Ágiles", Banfield Loma de Zamora, Manual Users, 2009

**[05]** "MICROSOFT APPLICATION ARCHITECTURE GUIDE 2nd Edition", Microsoft Corporation, 2009

**[06]** LARMAN Craig, "UML y Patrones" 2nd Edición, Pearson Education, Madrid, 2003

**[07**] SINAY Damián, "Web Services con C#", Banfield Loma de Zamora, Manual Users, 2006

**[08**] ASTEASUAIN, Fernando, "UML", Banfield Loma de Zamora, Manual Users, 2009

**[09**] MINERA Francisco, "XML La guía total", Banfield Loma de Zamora, Manual Users, 2009

**[10]** BLE JURADO Carlos, "Diseño Ágil con TDD", iExpertos, 2010

**[11**] DE LUCA Damián, "WEBMASTER PROFESIONAL", Bueno Aires, 2011

**[12**] DE LUCA Damián, "HTML 5", Bueno Aires, 2011

# 14.0 REFERENCIAS WEB

**[01]** ASP.NET MVC http://www.asp.net/mvc

**[02]** ASP.NET MVC3 http://www.asp.net/mvc/mvc3

**[03]** JQuery http://www.jquery.com

**[04]** Martin Fowler La nueva Metodología.

http://www.programacionextrema.org/articulos/newMethodology.es.html

**[05]** Git Extensions http://sourceforge.net/projects/gitextensions/

**[06]** Web Data Service http://msdn.microsoft.com/es-ar/data/bb931106

**[07]** StyleCop 4.6 http://stylecop.codeplex.com/releases/view/64494

**[08]** http://es.wikipedia.org/wiki/Jquery

**[09]** Martin Salias (http://www.codeandbeyond.org/2010/11/adios-soa-adios.html)