ARQUITECTURAS DE
APLICACIONES ENTERPRISE

Director:

Dr. Gustavo H. Rossi
(gustavo@lifia.info.unlp.edu.ar)

Alumnos:

Alfredo Fidani – 02074/1
(alfredo.fidani@gmail.com)

Matias Butti– 2013/7
(matias.butti@gmail.com)

TABLA DE CONTENIDOS

AGRADECIMIENTOS

A mis padres y hermanos como muestra de mi cariño y agradecimiento, por todo el amor y el apoyo brindado.

A mi familia y amigos por estar siempre a mi lado.

Matías

A mi familia que me dio el apoyo y la posibilidad necesaria para mi formación profesional y como persona.

Fido

Al laboratorio por la formación académica y profesional y en particular a Gustavo por ser el guía de nuestra formación.

Y un agradecimiento especial a la Lic. Marta Vázquez por sus aportes y constante apoyo para la finalización de este trabajo.

LISTADO DE ANEXOS

Anexo Documento de Visión.doc

PARTE I – INTRODUCCIÓN

Los desarrollos de software de gran escala y las arquitecturas de software han concentrado la mayor atención en la última década incluyendo grandes compañías y pequeñas firmas de software en el mundo. Ha crecido el reconocimiento de la importancia del diseño y organización del sistema previo a la construcción como así también de principios de diseño, de desarrollo y patrones reusables.

Si bien se desarrollan herramientas para facilitar el diseño, frameworks para potenciar el reuso, y se utilizan patrones de diseño para la construcción de sistemas, desafortunadamente el número de proyectos que fracasan es alto. El **caos** que se reporta en el 2004[1] establece que el porcentaje de proyectos exitosos no supera el 34% y que 15% es el porcentaje de proyectos que fracasan. El 51% de los proyectos superan el presupuesto estimado, no cumplen las fechas comprometidas o carecen de funcionalidad crítica y requerimientos importantes.

Lo expuesto ocurre, entre otros motivos, por no definir en la etapa inicial del proyecto una arquitectura que permita cumplir con los requerimientos no funcionales y que acompañe el desarrollo para asegurar la calidad y maximizar la productividad.

El término arquitectura, además de tener una gran variedad de definiciones, suele ser utilizado para referirse a distintos aspectos de un sistema.

Sobre la base de la bibliografía y de nuestra experiencia en equipos de arquitectura de software, usaremos el término para referirnos al *soporte* necesario para la construcción de los casos de uso, que ayude a garantizar el cumplimiento de los requerimientos no funcionales, asegure la calidad y maximice la productividad.

Es frecuente en la industria del software, que el alcance de la arquitectura se restrinja a la definición de tecnologías a utilizar y a la elaboración de pautas plasmadas en varios documentos. De esta manera, los responsables de la arquitectura de software suelen trabajan solo en etapas previas al comienzo del desarrollo.

---

[1] S.H. Kaisler, F. Armour, M. Valivullah, "Enterprise Architecting: Critical Problems", IEEE Proceedings 38va Conferencia Internacional en Hawaii en Ciencias de Sistemas, 2005

Las mencionadas pautas son una parte muy importante de la arquitectura pues definen el conjunto de patrones y abstracciones coherentes que proporcionan el marco de referencia para guiar la construcción del software. De todas maneras, consideramos que el ***soporte*** que debe brindar la arquitectura de software incluye, además, otros elementos necesarios para ordenar y estandarizar el desarrollo, maximizar la productividad y simplificar el mantenimiento. Entre estos elementos aparecen:

- Pautas de diseño y desarrollo que aseguran la calidad del producto
- Pautas que garanticen el cumplimiento de los requerimientos no funcionales
- Diseño y Desarrollo de Componentes que resuelven problemas comunes.
- Diseño y Desarrollo de frameworks que abstraen funcionalidad común.
- Actualizaciones tecnológicas
- Evaluación de frameworks existentes en la comunidad que puedan ser de utilidad en el proyecto.
- Evaluación de herramientas
- Soporte técnico a los desarrolladores
- Capacitación al equipo de desarrollo
- Resolución de problemas de performance

Algunas de estas actividades culminarán antes de comenzar el desarrollo pero otras acompañarán al equipo a lo largo de la etapa de construcción. Es por esto que, las tareas de los responsables del equipo de arquitectura, no terminarán cuando comienza la construcción de los casos de uso sino que continuarán hasta avanzado el proyecto.

El problema de disponer de una arquitectura, teniendo en cuenta todos los aspectos antes mencionados, es aún más complicada en aplicaciones Enterprise de gran tamaño por razones que se explicarán a lo largo de la tesis.

La presente tesis tiene entonces por objetivo, estudiar la definición de arquitecturas en aplicaciones Enterprise de gran tamaño (incluyendo la definición del marco de arquitectura y las actividades arriba mencionadas) para poder luego definir una metodología que describa los pasos a seguir para su definición y construcción.

Los aportes de esta tesis serán entonces:

- Brindar distintas soluciones a los problemas técnicos que debe resolver el marco de arquitectura en aplicaciones enterprise de gran tamaño, analizando ventajas y desventajas de cada solución.

- Proponer una metodología para la definición y construcción de una arquitectura definiendo claramente las responsabilidades de un equipo de arquitectura de software.

- Definir la arquitectura de un caso de estudio para mostrar, en un caso concreto, la metodología propuesta, el marco definido que muestre las decisiones técnicas tomadas y el acompañamiento que haría la arquitectura a lo largo del desarrollo para simplificarlo y mejorar su productividad.

La tesis se divide en 5 capítulos. En el capítulo 2 se hace una presentación de las características de las aplicaciones de tipo Enterprise. En el capítulo 3 presentamos el concepto de arquitectura de software, los estilos arquitectónicos y los patrones arquitecturales que ayudarán en la definición del marco teórico de arquitectura. En el capítulo 4 se presenta una metodología que describe los pasos a seguir para definir una arquitectura. En el último capítulo se presenta el caso de estudio donde se aplica la metodología y se muestran decisiones técnicas de arquitectura.

Enfoque inicial del problema

Al momento de traer soluciones a los problemas presentados, trabajaremos con el paradigma orientado a objetos que creemos tiene la gran ventaja de permitir, en este tipo de sistemas, expresar los diseños utilizando modelos muy cercanos a los conceptos que aparecen en el problema real.

Para referirnos a una funcionalidad del sistema lo haremos bajo el término *caso de uso*.

Con respecto a la tecnología, elegimos Java como lenguaje de programación. Con este lenguaje mostraremos soluciones a problemas que se vayan presentando a lo largo de la tesis.

Tomaremos como caso testigo un sistema para la Secretaría de Hacienda de la Nación que utilizaremos como referencia en cada ejemplo y como caso de estudio en el capítulo 5.

PARTE II – Aplicaciones Enterprise

Las Aplicaciones Enterprise representan una importante herramienta de soporte para las organizaciones. Serán las encargadas de relacionar la información, los recursos humanos, la tecnología, los procesos de negocio y los elementos de la infraestructura de la empresa, para cumplir con los objetivos de la organización.

Una Aplicación Enterprise modela el funcionamiento del proceso de la empresa. Su principal objetivo es la automatización de los procesos de negocio de la organización. Si bien existen distintos tipos de organizaciones con distintos procesos de negocio, en esta tesis estudiaremos los sistemas que modelan los procesos de negocio **administrativos** de una organización.

Para analizar de forma completa las características de este tipo de sistema, buscaremos resultados en el análisis de sistemas de **gran tamaño** o que representen organizaciones donde los procesos de negocio sean complejos y estratificados. Esto involucra organizaciones con un gran número de personas que desarrollan sus tareas a través del sistema, con gran dispersión geográfica y realizando tareas de procesos complejos.

Las aplicaciones enterprise brindan soporte funcional a las organizaciones con el objetivo de mejorar la productividad y la eficiencia de las actividades de la misma.

Las características generales de las aplicaciones enterprise se describen a continuación:

- Persistencia de datos

  Los datos que se van generando a través de las distintas operaciones que brinda el sistema deben ser almacenados La persistencia de los datos asegura que los mismos queden disponibles para futuras operaciones.

  Los datos deben soportar cambios de hardware y migraciones a nuevas estructuras de persistencia.

  Considerando que trabajamos con entornos orientados a objetos y que en la actualidad la industria sigue utilizando base de datos relacionales, es imprescindible trabajar con un *mapeador objeto*

relacional que permita "traducir" las clases en tablas (y viceversa) y los atributos en campos de las tablas (y viceversa).

- Grandes volúmenes de datos

  Los procesos administrativos generan importantes volúmenes de datos. Es por esto que mucha de las operaciones que brindan las aplicaciones enterprise generan gran cantidad de información. De hecho, estas aplicaciones suelen proveer la posibilidad de ejecutar las operaciones de forma masiva y de forma batch, generando más información en menos tiempo y con menos trabajo.

  Como consecuencia de la gran cantidad de datos que manejan este tipo de aplicaciones, suelen ser un buen recurso donde aplicar *minería de datos* para descubrir información implícita de gran ayuda para la organización.

- Acceso de datos concurrente

  Debido a que las aplicaciones enterprise son multiusuario, la información será accedida por varios usuarios a la vez. Durante este acceso concurrente se deben garantizar las 4 propiedades ACID: Atomicidad, Consistencia, Aislamiento y Durabilidad.

  - Atomicidad. Una transacción debe ser tratada como una unidad de procesamiento. Por lo tanto, todas las acciones de la transacción se realizan o ninguna de ellas se lleva a cabo. Asimismo, si una transacción se interrumpe por una falla, sus resultados parciales deben ser desechados.

  - Consistencia. Los recursos del sistema deben estar en un estado consistente y no-corruptos al comenzar la transacción y finalizada su ejecución. Las transacciones no deben violar las restricciones de integridad de un sistema.

  - Aislamiento. Una transacción en ejecución no puede revelar sus resultados a otras transacciones concurrentes antes de su completitud. Más aún, si varias transacciones se ejecutan concurrentemente, los resultados deben ser los mismos que si ellas se hubieran ejecutado de manera secuencial (seriabilidad).

- Durabilidad. Se debe asegurar que una vez que una transacción finaliza, sus resultados son permanentes y no pueden deshacerse.

- Presentación de los datos de diferentes maneras

  Es necesario presentar los datos de diferentes maneras considerando los distintos perfiles de los usuarios y las distintas operaciones que provee el sistema.

- Integración con otros sistemas

  Es muy común en sistemas enterprise consumir servicios que ofrecen otros sistemas. Esta interacción podrá ser con otros sistemas de la misma organización, con sistemas de otras organizaciones que proveen servicios y con aplicaciones que tienen como fin vender servicios a aplicaciones enterprise entre otro tipos de integración.

  Hoy en día, el uso de Web Services facilita la integración de aplicaciones merced a la estandarización de protocolos y formatos utilizados para la comunicación e interpretación de los pedidos y respuestas.

- Lógica de negocio muy compleja y evolutiva

  Para clarificar la diferencia con respecto a la lógica entre aplicaciones enterprise y no enterprise, daremos algunos ejemplos de ambas.

  Un sistema para la gestión del presupuesto Nacional, Manejo de pacientes en un Hospital y Manejo de clientes son consideradas aplicaciones enterprise.

  Por otro lado, sistemas operativos, compiladores y juegos no son consideradas aplicaciones enterprise.

Requerimientos funcionales de aplicaciones enterprise de gran tamaño

De la operatoria diaria de la organización surgen ciertas necesidades funcionales generales que la aplicación debe contemplar. Las aplicaciones empresariales orientadas a empresa de envergadura tienen características que detallaremos a continuación.

- *La información debe estar integrada*. La información generada por cada área de la organización es un insumo para alguna otra y por ende se debe administrar de forma centralizada para facilitar su disponibilidad para quien la necesite.

  Esto requiere en principio:

  - La definición de los procesos en forma centralizada - centralización normativa
  - La estandarización de la información de uso común - codificaciones por ejemplo

- *Descentralización operativa*. Las distintas áreas o sucursales operan en forma descentralizada siguiendo normas definidas de manera centralizada

- *Multimoneda*: Las gestiones requieren la utilización de recursos financieros en distintas monedas en el mundo globalizado.

- *Manejo de objetos del negocio de gran tamaño*: El sistema debe soportar que sus operaciones puedan trabajar sobre objetos que, por modelar conceptos del dominio que disponen de mucha información, son naturalmente de gran tamaño. Por ejemplo, una factura puede tener cientos de ítems.

- *Log de operaciones*: Llevar registro de cada una de las operaciones que fueron ejecutadas por los usuarios finales, con el fin de realizar auditorias, controles y seguimientos.

- *Definición de perfiles y niveles de información*. Se debe permitir la definición de perfiles, dominios de información y funcionalidad con los que se pueden trabajar en cada uno de los perfiles configurados.

Requerimientos no funcionales de aplicaciones de gran tamaño

Como consecuencia de las características inherentes a los sistemas Enterprise de gran tamaño, de los requerimientos funcionales y de los procesos que se deben cubrir en este tipo de sistemas, aparecen como parte fundamental del problema, requerimientos no funcionales que también deben ser considerados en la solución:

- Performance: La performance mide distintos tiempos que se describen a continuación:
  - Tiempo de respuesta: tiempo que tarda el sistema en procesar un pedido externo (por ejemplo a partir de presionar un botón)
  - Tiempo de respuesta al usuario: tiempo que tarda el sistema en responder al usuario (aunque posiblemente no haya terminado de procesar). En un proceso asíncrono, aún con un tiempo de respuesta alto, se podrá bajar el tiempo de respuesta al usuario si se da una respuesta inmediata indicando que su proceso fue encolado.
  - Tiempo de latencia: Es el tiempo mínimo de cualquier respuesta.
  - Throughput: Cuanto trabajo se puede hacer en un determinado tiempo. En aplicaciones enterprise suelen medirse transacciones por segundo.

  La performance cumple un rol muy importante en cualquier aplicación. Este requerimiento, frecuentemente marca el éxito o fracaso del proyecto. La aplicación necesitará ejecutar la mayoría de las transacciones con un tiempo de respuesta adecuado, de manera de no degradar la calidad de servicio percibida por los usuarios. Son éstos los que determinan la aceptación del sistema.

  El gran esfuerzo del equipo de desarrollo se puede ver deslucido por problemas de performance pues, generalmente, produce disconformidad en los usuarios del sistema

  Si bien lo expresado respecto del tiempo de respuesta aplica a cualquier tipo de aplicación, alcanzar buena performance en Aplicaciones Enterprise es más complicado que en otro tipo de

aplicaciones. En las que nos ocupan, como causa principal, aparecen otros requerimientos no funcionales que compiten con la performance. Por ejemplo, requerimientos muy estrictos respecto a la seguridad que se traducen en administrar los datos organizados por dominios. Esto implica verificar en cada acceso a los datos, los permisos del usuario. Analizaremos más adelante este problema, cuando veamos posibles soluciones.

Algunos factores que pueden afectar la performance son

A nivel físico: el acceso a la base de datos, el enlace de red utilizado y la capacidad de procesamiento de los servidores involucrados.

A nivel diseño: la arquitectura, las pautas de desarrollo definidas, la buena utilización de los frameworks, la optimización de consultas a BD, entre otros.

Tomar decisiones sobre la performance es una tarea difícil. Es importante considerarla en cualquier decisión de diseño, tanto a nivel de arquitectura como a nivel de diseño y desarrollo de la aplicación.

De todas maneras, aún habiéndola tenido en cuenta en las etapas tempranas de diseño e inclusive en la definición de la arquitectura, los problemas de performance pueden producirse y el equipo debe estar preparado para poder afrontarla. No alcanzará solo con analizar el diseño sino que se necesitarán herramientas de medición y *"profiling"*[2] para poder descubrir un posible cuello de botella, un algoritmo ineficiente, un método invocado varias veces de manera innecesaria o cualquier otra causante del problema.

- Capacidad y Escalabilidad: Las aplicaciones enterprise deben ser capaces de crecer en escala para adecuar el servicio al crecimiento de la demanda.

  Si bien es frecuente realizar tests para verificar que se cumplan los tiempos de respuesta requeridos para la funcionalidad implementada, es importante realizar pruebas de carga para simular

[2] Profiling: Análisis e información de la ejecución de procesos, en formato de gráficos o tablas que detallan y representan características distintivas y cuantitativas.

un ambiente donde se reproduzca la carga real. De esta manera, se podrá detectar el umbral de degradación del sistema, anticipando el problema con margen suficiente para su análisis y solución.

- Usabilidad: Otro de los requerimientos muy importantes, por tener impacto directo en el usuario final, es la usabilidad. La usabilidad define las pautas que tienen como objetivo garantizar una interacción cómoda, simple y que se ajuste a la operatoria diaria del usuario con el sistema.

  Las mencionadas pautas, serán quienes guíen el diseño de las interfaces gráficas de la aplicación.

  Para definir las pautas de usabilidad es importante conocer, además de la funcionalidad que se quiere proveer, la actividad diaria que está automatizando dicha funcionalidad para que sea natural su uso. Otro punto importante es determinar las distintas *categorías de usuarios*[3] que pueden usar el sistema. Frecuentemente ocurre que el sistema se releva con expertos del dominio que no serán los únicos usuarios del sistema. Existe una relación entre caso de uso y categoría de usuario. Esto no debe perderse de vista en la definición la interfaz.

  Teniendo en cuenta los puntos mencionados, las Aplicaciones Enterprise deberían incluir las siguientes características en lo que respecta a la interacción con el usuario:

  - Look & Feel
    - Personalización del entorno de trabajo. No a todos los usuarios la misma organización del entorno de trabajo les resulta cómoda. Es por esto que se debe brindar flexibilidad para que un usuario pueda armar su ambiente según sus preferencias. Por ejemplo, deberá poder definir el formato de sus ventanas y sus reportes.
  - Productividad del usuario final

[3] Profesionales, Data entries, Analistas, Gerentes

- Criterios de usabilidad. Es importante que participe un especialista de usabilidad en el equipo para definir estos criterios. El grupo de desarrollo no tiene esta formación y por ende desatiende este criterio haciendo necesario brindarle pautas al respecto. Un especialista sí tiene en mente qué cosas pueden facilitar la tarea del usuario y qué cosas pueden hacer el sistema poco amigable.

- Funciones programables: posibilidad de mapear funciones a teclas. Algunos usuarios prefieren trabajar con el Mouse pero muchos otros prefieren hacerlo con el teclado para mejorar la productividad. En este caso, algo tan simple de implementar como los *shortcuts* (teclas rápidas) puede evitar demoras en la carga masiva de datos de los usuarios.

- Ayudas en Línea sensibles al contexto

  - Es importante la ayuda para los usuarios finales. Esta ayuda debe estar disponible desde la aplicación para dar autonomía al usuario. Debe ser contextual, permitiendo resolver problemas a distintos niveles: circuitos o procesos, casos de uso específicos o el llenado de un campo en particular.

- Mensajes de Error

  - Ante un error, es muy importante informar claramente al usuario lo acontecido. No basta con informar que hubo un problema al ingresar un comprobante, sino que se debe indicar en que ítem del mismo se produjo el error.

    También es necesario distinguir entre un error y un alerta. Un error impide continuar con la operación; un alerta puede requerir la confirmación del usuario para continuar con la operación o simplemente informar.

    Los errores internos de una operación deben capturarse y manejarse evitando mostrar errores incomprensibles al usuario.

- Adaptabilidad al cambio de reglas de negocio: Esta necesidad puede producirse por la poca experiencia de los analistas y de los usuarios en el dominio que se está modelando, o por la variabilidad inherente de las reglas del negocio que se modela.

- Portabilidad a través de plataformas: Las aplicaciones Enterprise deben tener la capacidad de operar bajo una variedad de plataformas y sistemas operativos.

  Si bien no es objetivo de este capítulo hablar del marco de arquitectura de este tipo de aplicaciones, cabe mencionar (para poder estudiar la portabilidad) que dispondremos de clientes y servidores de aplicación.

  Del lado del servidor de aplicación, la portabilidad podría surgir como una necesidad del equipo de desarrollo. Ellos serán quienes administran los servidores y por diferentes razones (económicas, disponibilidad, conocimiento, administración, entre otras) podrían sugerir cambios de plataformas o sistemas operativos en los equipos que alojan la parte *server* de la aplicación. Desarrollar, entonces, un producto que se pueda adaptar a estos cambios durante el desarrollo y la puesta en producción es un punto a tener en cuenta..

  En aplicaciones de cliente *standalone*, la portabilidad del cliente también es importante dado que los usuarios pueden disponer de diferentes ambientes y sistemas operativos.

  A continuación se ejemplifica una necesidad real de portabilidad de la aplicación cliente en el caso de ejemplo de esta tesis. "El cliente standalone de la aplicación fue diseñado para trabajar con el sistema operativo Windows pues todos los usuarios finales disponían de ese SO. Luego de varias pruebas de carga se detectó que ciertos usuarios no podrían instalar la aplicación en sus PCs por problemas de recursos. Se decidió, entonces, utilizar un emulador gráfico para estos usuarios. Este emulador disponible en la instalación corría bajo Linux. Era necesario entonces poder correr el cliente en Linux. El requerimiento original de portabilidad del cliente, garantizó la posibilidad de llevar a cabo esta decisión sin mayores inconvenientes. "

En el caso de aplicaciones Web, la portabilidad es responsabilidad del Web Server pues la portabilidad del usuario final la brindan los browsers[4].

- **Seguridad:** Las aplicaciones Enterprise deberán proveer una infraestructura de alta seguridad, acorde al nivel de exigencia de la industria u organización, que proteja al sistema contra las vulnerabilidades de las aplicaciones de gran escala. Este tipo de arquitectura debe contar con las siguientes consideraciones:

    - Contraseña única (Single Sign-On): La implementación de este tipo de validación de usuarios permite a los mismos un acceso transparente a través de todos los módulos que posea el sistema.

    - Administración centralizada de usuarios: Se debe considerar un único punto de administración de usuarios y permisos, que a su vez permita definir "administradores locales o parciales" por cada una de las posibles sucursales a los que se les asignará un subconjunto de usuarios, roles y datos sobre los que ejercer sus funciones dentro de la empresa u organización. Asimismo la administración de los permisos sobre los datos debe ser centralizada y almacenada de forma unificada.

    - Encriptación: Las aplicaciones Enterprise deben hacer uso de técnicas de criptografía para asegurar la protección de la información sensible del sistema, como así también encriptaciones a nivel de transmisiones de datos en la red.

- Alta Disponibilidad: El sistema desarrollado sobre la mencionada arquitectura debe ser capaz de cumplir con la exigencia de disponibilidad que requiere la empresa u organización. Para aplicaciones de tipo Enterprise comúnmente es de 7 por 24 para las principales componentes de software del sistema transaccional, no incluyéndose subsistemas de consulta e históricos.

- Mantenibilidad:

  La arquitectura sobre la que se desarrolla el sistema Enterprise debe separar claramente las capas de presentación y de negocio, de esta manera, el producto ha de tener la habilidad de cambiar fácilmente

[4] Se pueden presentar diferencias en el look & feel debido a que los browsers no suelen adherirse a estándares

la apariencia y percepción, permitiendo así la fácil adecuación a los distintas sucursales o áreas en las que se utiliza el sistema. Con una división modular y basada en componentes de cada una de las capas, se permite la posibilidad y facilidad de realizar adaptaciones, adecuaciones, modificaciones y extensiones ante los sucesivos cambios que suelen tener las empresas y organizaciones.

- Hot deploy: Es necesario que se contemple en la arquitectura la posibilidad de realizar reinstalaciones de parte del sistema sin afectar al sistema completo, ante emergencias que no permitan sacar de línea el sistema.

- Autonomía del usuario: Se debe garantizar la autonomía del usuario permitiendo que defina circuitos que se adecuen a su ubicación geográfica, a sus gestiones y a sus dimensiones, sin requerir intervención del área de sistemas, y facilitando el tratamiento de información fuera de línea y con distinto nivel de agregación (por ejemplo Datawarehouses).

- Descentralización de operaciones: Debe permitir la habilitación y inhabilitación de funciones en cada proceso de forma parametrizada desde la casa central a medida que las sucursales crecen en rendimiento y madurez.

- Modos de Operación

  Modo normal

  Es el modo en el que los usuarios del sistema podrán disponer de toda la funcionalidad. Dentro de este modo se cuenta con la posibilidad de trabajar en forma interactiva y en forma desatendida (batch) para el procesamiento masivo de datos.

  Modo mantenimiento

  En este modo se encuentra el sistema cuando se realizan modificaciones de infraestructura, hardware, software de base o software de aplicación, brindando la posibilidad de deshabilitar

temporalmente los componentes del sistema afectados en su funcionalidad por estas modificaciones.

- Uso de Red

  En el gráfico a continuación se presentan los componentes físicos de una Aplicación Enterprise y sus relaciones a través de la red.

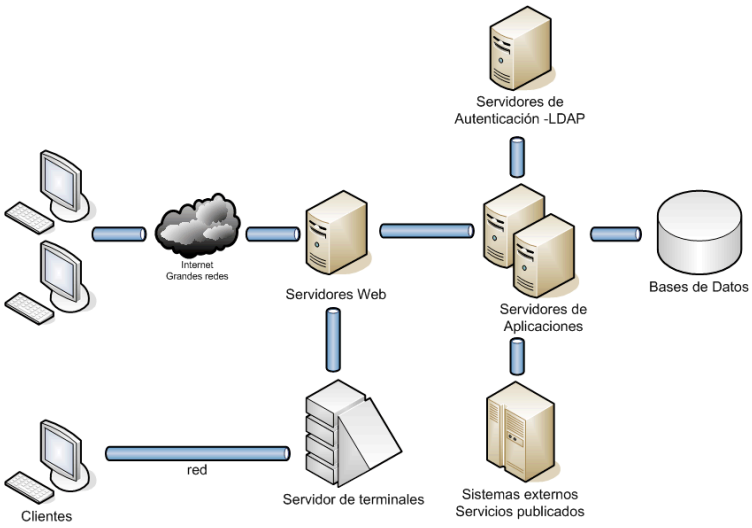


Ilustración 1 - Elementos físicos de una Aplicación Enterprise

Considerando que la interacción con sistemas externos y que la comunicación entre los distintos servidores de la arquitectura física (Servidores Web, Servidores de Aplicación y Bases de Datos) se hará a través de la red, es necesario privilegiar en el diseño y programación la optimización del uso de la red.

También es muy importante optimizar el uso de red desde el cliente al servidor. Existen estrategias de diseño y de arquitectura que permiten conseguir esta optimización, sin apartarse de un buen modelo de objetos[5]. Abordaremos este tema en secciones posteriores.

[5] En el patrón *Remote Facade* del libro "**Patterns of Enterprise Aplicación Architecture"** de Martin Fowler se presenta una solución al problema mencionado que mejora eficazmente el uso de red.

- Facilidad de Instalación

  Es necesario definir un proceso repetible para la instalación de la aplicación. Este proceso permitirá automatizar, con herramientas, la instalación del sistema en los distintos ambientes en etapas de desarrollo y producción. En el primer caso, se requieren múltiples ambientes, a saber:

  - Ambiente de Desarrollo
  - Ambiente de Aceptación de los usuarios
  - Ambiente de Capacitadores
  - Ambiente de Prueba de carga

  En el segundo caso, se requieren:

  - Ambiente de Preproducción
  - Ambiente de Producción

  El proceso no solo deberá definir las actividades relacionadas con el deploy de la aplicación sino también pautas que deben ser cumplidas durante el desarrollo. Este proceso contempla la automatización de tareas sobre la base datos y tareas de configuración de datos de instalación. Para que la automatización sea posible, el equipo de desarrollo debe cumplir con estas pautas preestablecidas.

  Para automatizar el proceso se debe disponer de herramientas que faciliten la tarea de *deploy* y registro de *logs* para su posterior verificación.

  Adicionalmente, para atender las emergencias de la instalación, se debe desarrollar un proceso que permita realizar un *hot deploy* manteniendo la disponibilidad del sistema en producción.

  Para atender las instalaciones, tanto emergencias como programadas, en el cliente se debe proveer un mecanismo de distribución automática de las versiones. En el caso de Java, el uso de JNLP, para poder publicar las nuevas versiones, permite

que el usuario siempre ejecute la última versión del cliente instalada.

- Integración con sistemas legados

  Una aplicación Enterprise se construye de forma modular. Para darle visibilidad al sistema y servicio al cliente, los módulos se van instalando progresivamente.

  Si el sistema reemplaza un sistema existente, surge como consecuencia de la decisión de la instalación progresiva, un proyecto que llamaremos convivencia. Este proyecto se ocupa de definir estrategias de migración de datos y de comunicación entre los sistemas legado y el nuevo.

PARTE III: MARCO TEÓRICO DE ARQUITECTURA

3.1 Que es la arquitectura de software?

Sobre la base de la bibliografía y de nuestra experiencia en equipos de arquitectura de software, usaremos el término para referirnos al ***soporte*** **necesario para la construcción de los casos de uso, que garantice el cumplimiento de los atributos de calidad, asegure la calidad y maximice la productividad.**

Ciertos autores y parte de la comunidad consideran como alcance de la arquitectura de software, lo que nosotros damos a llamar *marco teórico de arquitectura*. Si bien la definición del marco teórico es un paso muy importante en la definición de la arquitectura, no es el único para conseguirla.

Considerando la importancia del marco teórico de arquitectura, dedicaremos el resto del capítulo a abordar este tema. También abordaremos los estilos y patrones de arquitectura que son el medio para llegar al reuso durante la definición del marco teórico.

Una vez clarificado el concepto de marco teórico, y ya en el próximo capítulo, abordaremos el resto de las actividades que conciernen a la arquitectura de aplicación ordenándolas en una metodología.

3.2 Que es el marco teórico de la arquitectura de software?

El marco teórico define la estructura global del sistema, los elementos de la arquitectura y la forma en que se comunican estos elementos.

El marco teórico de arquitectura es el conjunto de patrones y abstracciones coherentes que guían la construcción del software para un sistema de información.

El marco teórico de la arquitectura de software de una aplicación se selecciona y diseña a partir del documento de visión.

El documento de visión debe incluir como mínimo

- Propósito del sistema
- Descripción de los interesados y usuarios
- Descripción global del sistema

- Perspectiva del sistema
- Resumen de la funcionalidad: Definición de muy alto nivel de los procesos que deben ser cubiertos con el sistema.
- Supuestos y dependencias con otros sistemas

- Características funcionales del producto
  - Características generales
  - **Requerimientos funcionales provenientes de los casos de uso significativos para la arquitectura.**
- Restricciones
  - Perfiles de usuario
  - De interfaz
- **Atributos de calidad: También conocido como requerimientos no funcionales. Ya se describieron en el capítulo anterior.**

A partir de aquí llamaremos *línea base de requerimientos* (LBR) al conjunto de requerimientos significativos para la arquitectura más el conjunto de atributos de calidad.

Para aclarar el concepto de marco teórico de arquitectura, tomamos una definición de Clements [Cle96a] que, si bien él la utilizó para definir arquitectura de software, consideramos que es más apropiada para definir solo el marco teórico de la arquitectura de software.: *"El marco teórico de arquitectura es, a grandes rasgos, una vista del sistema que incluye los elementos*[6] *principales del mismo, el comportamiento de esos elementos según se la percibe desde el resto del sistema y las formas en que los elementos interactúan y se coordinan para alcanzar la misión del sistema. "*

Cualquier decisión tomada en el marco teórico de arquitectura, por ser una decisión temprana en el proyecto que incide en el desarrollo, es muy costosa de modificar posteriormente. Es por este impacto que debe ser cuidadosamente definida.

[6] Es importante mencionar que a todas las definiciones le hemos cambiado el termino componente por elemento para no confundir con el termino de componente que se da a nivel implementación (COM, CORBA Component Model, EJB).

Decía Parnas[7] que las decisiones tempranas de desarrollo serían las que probablemente permanecerían invariantes en el desarrollo ulterior de una solución. Esas "decisiones tempranas" constituyen de hecho lo que hoy se llamarían decisiones arquitectónicas. Como escriben Clements y Northrop [CN96] en todo el desenvolvimiento ulterior de la disciplina permanecería en primer plano esta misma idea: la estructura es primordial (structure matters), y la elección de la estructura correcta ha de ser crítica para el éxito del desarrollo de una solución.

3.3 Un poco de historia

Cada vez que se narra la historia de la arquitectura de software (o de la ingeniería de software, según el caso), se reconoce que en un principio, hacia 1968, Edsger Dijkstra, de la Universidad Tecnológica de Eindhoven en Holanda y Premio Turing 1972, propuso que se establezca una estructuración correcta de los sistemas de software antes de lanzarse a programar, escribiendo código de cualquier manera [Dij68a]. Dijkstra, quien sostenía que las ciencias de la computación eran una rama aplicada de las matemáticas y sugería seguir pasos formales para descomponer problemas mayores, fue uno de los introductores de la noción de sistemas operativos organizados en capas que se comunican sólo con las capas adyacentes y que se superponen "como capas de cebolla". Inventó o ayudó a precisar además docenas de conceptos: el algoritmo del camino más corto, los stacks, los vectores, los semáforos, los abrazos mortales. De sus ensayos arranca la tradición de hacer referencia a "niveles de abstracción" que ha sido tan común en la arquitectura subsiguiente. Aunque Dijkstra no utiliza el término arquitectura para describir el diseño conceptual del software, sus conceptos sientan las bases para lo que luego expresarían Niklaus Wirth [Wir71] como stepwise refinement y DeRemer y Kron [DK76] como programming-in-the large (o programación en grande), ideas que poco a poco irían decantando entre los ingenieros primero y los arquitectos después.

En la década de 1980, los métodos de desarrollo estructurado demostraron no escalar suficientemente y fueron dejando el lugar a un nuevo paradigma, el de la programación orientada a objetos. En teoría, parecía posible modelar el dominio del problema y el de la solución en un lenguaje de implementación. La investigación que llevó a lo que después sería el diseño orientado a objetos puede remontarse incluso a la década de 1960 con Simula, un lenguaje de programación de simulaciones. Este fue el primero que proporcionaba tipos de

[7] David Lorge Parnas (nacido el 10 de Febrero de 1941) es un pionero en la ingeniería de software, que desarrolló el concepto de diseño modular el cual es la base de la programación orientada a objetos. Parnas es además un defensor del realismo tecnológico.

datos abstractos y clases. Años más tarde, Smalltalk reafirmó la existencia del paradigma con un lenguaje puramente orientado a objetos. Paralelamente, hacia fines de la década de 1980 y comienzos de la siguiente, la expresión arquitectura de software comienza a aparecer en la literatura para hacer referencia a la configuración morfológica de una aplicación.

Sin embargo, la arquitectura de software como disciplina bien delimitada, data de hace poco tiempo. El primer texto que vuelve a reivindicar las abstracciones de alto nivel, reclamando un espacio para esa reflexión y augurando que el uso de esas abstracciones en el proceso de desarrollo pueden resultar en "un nivel de arquitectura de software en el diseño" es uno de Mary Shaw [Shaw84] seguido por otro llamado "Larger scale systems require higher level abstractions" [Shaw89]. Se hablaba entonces de un nivel de abstracción en el conjunto; todavía no estaban en su lugar los elementos de juicio que permitieran reclamar la necesidad de una disciplina y una profesión particulares. El primer estudio en que aparece la expresión "arquitectura de software" en el sentido en que hoy lo conocemos es sin duda el de Perry y Wolf [PW92]; ocurrió tan tarde como en 1992, aunque el trabajo se fue gestando desde 1989. En él, los autores proponen concebir la AS por analogía con la arquitectura de edificios, una analogía de la que luego algunos abusaron [WWI99], otros encontraron útil y para unos pocos ha devenido inaceptable [BR01]. El artículo comienza diciendo exactamente:

"El propósito de este paper es construir el fundamento para la arquitectura de software. Primero desarrollaremos una intuición para la arquitectura de software recurriendo a diversas disciplinas arquitectónicas bien definidas. Sobre la base de esa intuición, presentamos un modelo para la arquitectura de software que consiste en tres componentes: elementos, forma y fundamentos(rationale). Los elementos corresponden a procesamiento, datos o conexión. La forma se define en términos de las propiedades de, y las relaciones entre, los elementos, es decir, restricciones operadas sobre ellos. Los fundamentos proporcionan una base subyacente para la arquitectura en términos de las restricciones del sistema, que lo más frecuente es que se deriven de los requerimientos del sistema. Discutimos los elementos del modelo en el contexto tanto de la arquitectura como de los estilos arquitectónicos...."

Para la caracterización de lo que sucederá en la década siguiente ellos formulan esta otra frase que ha quedado inscripta en la historia mayor de la especialidad:

La década de 1990, creemos, será la década de la arquitectura de software. Usamos el término "arquitectura" en contraste con "diseño", para evocar nociones de codificación, de abstracción, de estándares, de entrenamiento formal (de los arquitectos de software) y de estilo. Es tiempo de re-examinar el papel de la arquitectura de software en el contexto más amplio del proceso de

software y de su administración, así como señalar las nuevas técnicas que han sido adoptadas.

Dando cumplimiento a las profecías de Perry y Wolf, la década de 1990 fue sin duda la de la consolidación y diseminación de la AS en una escala sin precedentes. Las contribuciones más importantes surgieron en torno del instituto de ingeniería de la información de la Universidad Carnegie Mellon (CMU SEI). En la misma década surge también la programación basada en elementos, que en su momento de mayor impacto impulsó a algunos arquitectos mayores, como Paul Clements [Cle96b], a afirmar que la AS promovía un modelo que debía ser más de integración de componentes pre-programados que de programación.

Un segundo gran tema de la época fue el surgimiento de los patrones, cristalizada en dos textos fundamentales, el de la Banda de los Cuatro en 1995 [Gof95] y la serie POSA desde 1996 [BMR+96]. El primero de ellos promueve una expansión de la programación orientada a objetos, mientras que el segundo desenvuelve un marco ligeramente más ligado a la AS. Este movimiento no ha hecho más que expandirse desde entonces. El creador de la idea de patrones fue Christopher Alexander, quien incidentalmente fue arquitecto de edificios; Alexander desarrolló en diversos estudios de la década de 1970 temas de análisis del sentido de los planos, las formas, la edificación y la construcción, en procura de un modelo constructivo y humano de arquitectura, elaborada de forma que tenga en cuenta las necesidades de los habitantes [Ale77]. El arquitecto (y puede copiarse aquí lo que decía Fred Brooks) debe ser un agente del usuario.

En el siglo XXI, la AS aparece dominada por estrategias orientadas a líneas de productos y por establecer modalidades de análisis, diseño, verificación, refinamiento, recuperación, diseño basado en escenarios, estudios de casos y hasta justificación económica, redefiniendo todas las metodologías ligadas al ciclo de vida en términos arquitectónicos. Todo lo que se ha hecho en ingeniería debe formularse de nuevo, integrando la AS en el conjunto.

3.4 Pasos para definir el marco teórico de arquitectura es necesario:

1-Analizar el documento de visión

2- Definir la estructura global del sistema apoyándose en los estilos y los patrones de arquitectura.

1-Analizar el documento de visión

La línea base de requerimientos se traduce en los requerimientos funcionales que afectan a la arquitectura de aplicación y en los *atributos de calidad* (también conocidos como requerimientos no funcionales) como los que se describieron en el capítulo anterior.

En la sección *"Requerimientos no funcionales de aplicaciones de gran tamaño"*, describimos cada uno de los atributos de calidad. En esta sección retomaremos el tema, para destacar algunas cuestiones a considerar para la toma de decisiones arquitectónicas sobre la base de estos atributos de calidad. Usaremos algunos de los atributos mencionados en dicha sección para ejemplificar.

- **No existe la arquitectura "silver bullet"**

  No existe la arquitectura que sea la mejor solución para cualquier tipo de aplicación. No alcanza que el marco de arquitectura sea: "se utilizará una arquitectura en capas". Tampoco es una buena alternativa lanzarse a utilizar una arquitectura que funcionó bien en otra aplicación sin hacer un previo análisis. La arquitectura depende de los atributos de calidad y de algunos requerimientos funcionales de la aplicación que se va a construir.

  Para ejemplificar lo expresado en el párrafo anterior, tomemos el caso del atributo de calidad "Adaptabilidad al cambio de reglas de negocio". El factor común de las reglas de negocio en las aplicaciones enterprise es la forma y frecuencia en la que cambian, dado que:

  - Experimentan cambios, dado que la realidad cambia y el negocio se adapta a ella adecuando sus reglas.
  - El usuario no siempre define inicialmente las reglas correctamente.

  Podemos tener una aplicación con reglas muy complejas y muy variables, pero cada modificación a una de estas reglas debe ser evaluada por el equipo de análisis, luego pasar al quipo de desarrollo y testing para recién llevarla a producción. Por otro lado, en el caso de estudio, se presenta la necesidad de que ciertas reglas no se caracterizan por su variación frecuente, pero tienen la característica de que pueden ser modificadas por los propios usuarios. En la arquitectura de esta aplicación, será necesario

entonces, considerar en la definición del marco de arquitectura la posibilidad de que ciertas reglas de negocio se puedan modificar en tiempo de ejecución por usuarios con permisos determinados.

- **Los atributos de calidad pueden competir entre sí**

  Muchas veces es necesario hacer una priorización entre los distintos atributos de calidad porque compiten entre sí en al gún momento. En algunos casos se debe dejar de cumplir alguno para poder cumplir con otro.

  Como ejemplo tomemos la performance. Es frecuente que en ciertas decisiones que surjan por mejoras de performance vayan en desmedro de la mantenibilidad futura del sistema.

  Podemos graficar este problema ubicando en ambos extremos de una recta, la performance y los atributos de mantenibilidad como muestra la siguiente figura.

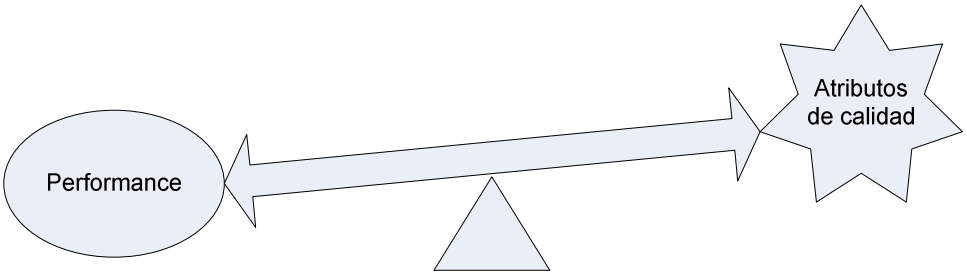


Ilustración 2 - Atributos de calidad vs performance

Para cada decisión que ponga en juego esta relación es necesario hacer una priorización. Para una decisión sobre un elemento de la arquitectura que necesita de extremada performance y en el cual la flexibilidad no es un punto muy importante pues es altamente probable que no vaya a cambiar, es necesario ser pragmático e ir por la opción performante. De todas maneras proponemos que, en cada decisión, se empiece tratando de dejar la solución al problema del extremo izquierdo y que se vaya corriendo la vara hasta el punto que me garantice la performance requerida. Aún así es extremadamente importante tener siempre en cuenta la performance esperada en cada decisión.

La tecnología y herramientas de programación que materializan el modelo de este paradigma no están preparadas para garantizar tiempos de respuesta adecuados para procesos que involucran un elevado número de elementos, por ejemplo 1000000 de objetos en memoria. Conociendo de antemano este requerimiento, la arquitectura debe definir pautas que deben ser respetadas y seguidas por el equipo de desarrollo.

- **Impacto de las modificaciones de arquitectura en las aplicaciones que la utilizan**

  Muchas veces, una decisión de arquitectura puede implicar una migración costosa en las aplicaciones ya desarrolladas. No se puede obviar la consideración de este costo cuando se toman decisiones de modificación de la arquitectura. Mediante un análisis detallado y una negociación con las áreas de desarrollo de las aplicaciones, se debe encontrar el momento en el que una decisión que implica migración se implementa en la arquitectura.

2-Definir la estructura global del sistema apoyándose en los estilos y los patrones de arquitectura.

El objetivo es definir y describir la arquitectura de una manera conceptual.

En esta definición se debe determinar:

- El estilo de la arquitectura
- Los elementos que componen la arquitectura
- Las responsabilidades de cada elemento de la arquitectura
- Consideraciones a tener en cuenta en la futura implementación de cada elemento
- La interacción entre los elementos
- Las restricciones de comunicación entre los elementos
- Pautas de uso del marco teórico

La arquitectura conceptual define **cómo un caso de uso debe ser diseñado sobre la arquitectura**.

Tomemos un ejemplo sencillo de la definición de un marco teórico para clarificar el concepto.

La arquitectura será distribuida entre clientes y servidores.

- *El cliente se organiza utilizando el patrón MVC.*
- *El servidor se organiza utilizando las siguientes capas:*
  - *Capa de servicios*
  - *Capa de lógica de dominio*
  - *Capa de acceso a datos.*

Algunas pautas

- *Un servicio es responsable de coordinar un caso de uso pero no de implementar la lógica de dominio*
- *Las consultas se organizan en la capa de acceso a datos implementando el patrón Data Access Object. Esto significa que las consultas se agrupan en distintos DAOs. Los DAOs no son instanciables; se deben conseguir a partir de factories encargados de tal función*

Las pautas son una parte muy importante en la definición del marco teórico. Son las encargadas de explicar cómo se debe diseñar sobre el marco teórico definido.

En el caso de estudio abordaremos una definición completa del marco teórico de su arquitectura.

La documentación del marco teórico tiene distintos interesados. Es por esto que deberán realizarse guías dirigidas a cada uno de ellos.

¿Quiénes son los interesados en conocer el marco teórico de arquitectura?

- Equipo de arquitectura: El documento orientado al equipo de arquitectura, debe describir cada una de las decisiones que se tomaron respecto al marco teórico. También es importante describir las causas que descartaron las

soluciones alternativas, de manera de poder entender en un futuro el por qué de cada decisión.

- Equipo de implementación: El documento orientado al equipo de implementación, funciona como una guía de uso del marco teórico de arquitectura. Más adelante, veremos que es necesario complementar esta documentación con la incluida en el marco de arquitectura completo (este incorpora tecnología y frameworks)

- Gerencia del proyecto: Debe aprobar las decisiones de arquitectura como responsable del proyecto y para ello necesita que las decisiones estén debidamente fundamentadas.

- Otros equipos no técnicos (análisis, testing): Si bien no serán usuarios directos de la arquitectura, es importante que conozcan el marco teórico para contextualizar su trabajo.

No es tema de esta tesis dedicarse a los distintos lenguajes y notaciones que existen para describir los marcos teóricos de arquitectura pero sí es necesario volver a recalcar la importancia de documentar y mantener actualizada dicha documentación. Una buena referencia es [KRUTCHEN], propuesta de Rational para la documentación utilizando distintas vistas de la arquitectura.

Al diseñar el marco teórico de arquitectura, es importante no reinventar la rueda y apoyarse en estilos arquitectónicos, patrones y soluciones exitosas para su definición. Analizaremos a continuación los estilos arquitectónicos, haciendo foco en el estilo n-layered que es el más utilizado en aplicaciones Enterprise. Luego abordaremos los patrones de arquitectura según la visión de Martin Fowler, visión que compartimos.

3.5 Estilos arquitectónicos

Como punto de partida se dispone de estilos de arquitectura ya definidos. Se conocen los requerimientos no funcionales que estos satisfacen.

El pensamiento de Parnas sobre familias de programas, en particular, anticipa ideas que luego habrían de desarrollarse a propósito de los estilos de arquitectura:

"Una familia de programas es un conjunto de programas (no todos los cuales han sido construidos o lo serán alguna vez) a los cuales es provechoso o útil considerar como grupo. Esto evita el uso de conceptos ambiguos tales como "similitud funcional" que surgen a veces cuando se describen dominios. Por ejemplo, los ambientes de ingeniería de software y los juegos de video

no se consideran usualmente en el mismo dominio, aunque podrían considerarse miembros de la misma familia de programas en una discusión sobre herramientas que ayuden a construir interfaces gráficas, que casualmente ambos utilizan."

En función de las características de la aplicación se elige inicialmente un estilo arquitectónico. A partir de este marco se trabaja para definir el marco de la arquitectura de aplicación.

Un estilo define:

- ✓ Glosario: tipos de elementos y tipos de conectores que participan. Ejemplos de elementos: cliente, servidor, base de datos, filtro, layer.
- ✓ Restricciones de combinación

Las arquitecturas complejas o compuestas resultan del agregado o la composición de estilos más elementales.

A continuación describiremos tres de los estilos arquitectónicos más conocidos para luego comenzar con la presentación del **estilo n-layered que es el que prevalece en las aplicaciones enterprise.**

Pipes and filters

- Un elemento denominado filtro lee datos de su entrada, les aplica algún procesamiento y genera datos para enviarlos a su salida.
- Los conectores de filtros se denominan *pipes* y son los encargados de llevar la salida de un filtro a otro.

Se menciona a continuación el ejemplo de un sistema donde una arquitectura de *pipes and filters* sería una buena elección.

Para el ejemplo, se estudió un caso de un sistema de un banco en Portugal. La lógica del negocio está implementada en un mainframe. El intercambio de información con el mainframe se debe hacer a través de un sistema de mensajes asíncronos (como podría ser un IBMMQSeries de IBM). El formato del mensaje es propietario y está definido por la aplicación del mainframe. Es muy costoso modificar el formato porque hay gran cantidad de clientes que está utilizando ese formato para la comunicación, lo cual implicaría una migración costosa en las aplicaciones cliente. Llamaremos a este formato M.

Aparecen varios nuevos clientes que generan información en distintos formatos y que necesitan intercambiar esta información con el sistema del mainframe.

Una buena alternativa es utilizar un *message broker* [8] que se encargue de intermediar entre cualquier cliente y el mainframe, para lograr el intercambio de información haciendo la traducción que corresponda.

Como ejemplos de clientes podemos mencionar:

- Un Banco: la información la deja en un archivo dentro de un directorio compartido, donde el **broker** puede leer. El formato del archivo es un formato propietario y tampoco está decidido a cambiarlo porque ese mismo formato lo utilizan para otro tipo de comunicaciones.
- Aplicación web: Front-end que recibe los datos para una operación, se comunica con un mainframe y muestra el resultado en un browser.
- Usuarios con dispositivos móviles: Desde este tipo de clientes también es necesario intercambiar información con el mainframe. Esta información llegaría al broker en formato WML (wireless markup language)
- web service: Un cliente pide consumir un web service para el intercambio de información. La información llegará en este caso en el cuerpo de un mensaje SOAP (protocolo estandar para escribir un mensaje que representa la ejecución de un servicio)

Una posible solución al broker sería disponer de un traductor para cada formato F distinto. Cada traductor es responsable de la traducción de F a M y viceversa. El problema es que, como el formato M es propietario, es muy probable que no sea fácil generar e interpretar este mensaje para cada uno de los clientes.

Una mejor solución sería que el broker defina un lenguaje XML que sea más fácil de generar e interpretar. En este caso, una arquitectura con estilo *pipes and filters* evitaría la complicación del formato F. La traducción a F se hará

[8] **Message broker** es un programa intermediario que traduce un mensaje desde el protocolo formal del emisor hasta el protocolo formal del receptor en una telecomunicación por red donde los programas se comunican mediante mensajes formalmente definidos.

solo una vez desde y hacia el lenguaje XML definido. Luego, se tendrán que definir traductores de los formatos de entrada al lenguaje XML. Pero esta traducción será mas simples y existirán herramientas para tal fin, por ejemplo XSL para convertir del formato WML que generó el cliente móvil a otro tipo de XML (el del broker). A continuación se muestra un gráfico de cómo quedaría la arquitectura *pipe and filters* con los diferentes clientes presentados.

Cuando aparezca un nuevo filtro, alcanzara con generar un traductor que sepa convertir el formato del nuevo cliente a alguno de todos los formatos intermedios.

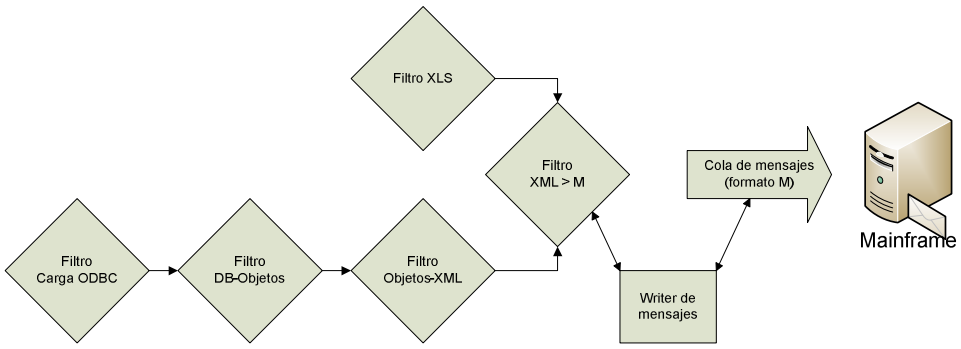


Ilustración 3 - Arquitectura Pipes and Filter

Publisher-subscriber (invocación implícita)

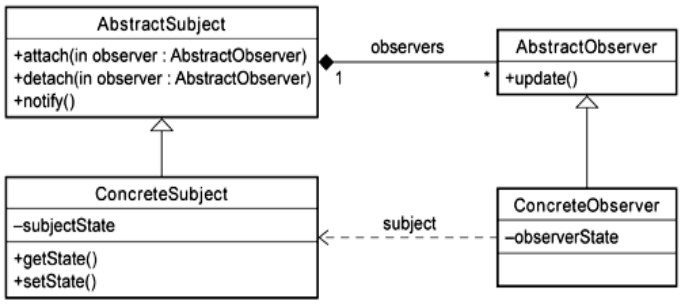
Es un modelo en el cual existen elementos de la arquitectura interesados en enterarse de ciertos eventos que pueden ocurrir en otro elemento de la arquitectura (al que denominaremos *publisher).*

La solución tradicional sería que el *publisher* avise explícitamente a cada uno de los interesados la ocurrencia de un evento, para que el interesado actúe en consecuencia. Para esto, el *publisher* será el responsable de conocer los eventos de interés de cada elemento. Este conocimiento explícito produce un gran acoplamiento del *publisher* hacia los interesados y se da una responsabilidad incorrecta al *publisher*, pues no es él quien debe mantener la información de interés en los eventos.

Para evitar los problemas mencionados, se invierte el conocimiento. Los elementos interesados (a partir de ahora *subscribers*) se "suscriben" a los eventos

de interés del *publisher*. Cuando se produce un evento, el *publisher* simplemente avisa que ocurrió el evento y los *subscribers* se enteran a través del soporte del mecanismo de observación.

Resumiendo, participan dos tipos de elementos en este estilo: los publishers y los subscribers. Estos elementos se relacionan a través de eventos.



Ventajas

- ✓ Simplicidad
- ✓ Desacoplamiento: El mecanismo evita tener que modificar el subject ante la aparición de nuevos *subscribers*. Serán los *subscribers* los responsables de subscribirse a los eventos de interés.
- ✓ Puede mejorar eficiencia, eliminando la necesidad de polling por ocurrencia de evento

Desventajas

- ✓ No se puede utilizar entre cualquier par de elementos de la arquitectura. Para poder implementar el mecanismo, es necesario que existe una relación entre el publisher y el subscriber que se mantenga en el tiempo. Por ejemplo, no es tarea trivial lograr una arquitectura que conecte, mediante un mecanismo de Publisher-subscriber, un cliente web (publisher) y un servidor corriendo en un web Server (subscriber)
- ✓ Pobre comprensibilidad: Puede ser difícil conocer qué pasará en respuesta a una acción pues existen invocaciones que no se ven reflejadas directamente en el código
- ✓ Pueden producirse fallas que dejen suscriptores sin recibir el mensaje

Como ejemplo, podemos presentar un caso desarrollado y presentado en un concurso de estudiantes organizado por IBM en 2002. El concurso consistía en presentar ideas de aplicaciones médicas que utilicen web services.

La situación del caso de estudio era un sistema para un hospital. La característica era que el desarrollo comenzaba por el módulo de ingresos de pacientes. Luego, se irían incorporando otros módulos como los de facturación, internación, obras sociales y médicos, que actuaban ante en eventos que se producían en el módulos de pacientes. Por ejemplo, ante el evento "ingreso de paciente" el sistema de facturación debía registrar el ingreso del paciente y generar el recibo correspondiente

Se propuso entonces una arquitectura Publisher-subscriber entre los distintos módulos de la aplicación. La comunicación de la ocurrencia de un evento se hacía a través de web services, pues los sistemas podían estar implementados en distintos lenguajes y plataformas.

Esta arquitectura permitía agregar nuevos módulos evitando modificar el módulo de pacientes siendo que éste no cambio la funcionalidad.

Blackboard (pizarrón)

El concepto de la arquitectura blackboard o pizarrón surgió en el campo de la inteligencia artificial hace mas de una década. Su propósito es concentrar y compartir un problema entre múltiples *agentes*. El nombre de arquitectura blackboard o pizarrón evoca la metáfora en la cual un grupo de expertos frente a un pizarrón colaboran en la resolución de un problema complejo.

El pizarrón se utiliza como repositorio central para compartir la información. Esta información representa **hechos, asunciones y deducciones** hechas por los expertos en el transcurso de la búsqueda de solución de un problema. Cada experto aporta desde su conocimiento, una estrategia distinta para resolver el problema. Un moderador controla la **tiza** y escribe sobre la pizarra, determinando qué información, brindada por los expertos, aporta a la solución del problema.

Una sesión de *resolución* comienza con la escritura de una especificación de problema por el moderador, junto con los hechos relevantes que puedan aportar a la solución del problema. Los expertos analizan el problema y hacen los aportes que puedan sobre la base de los conocimientos específicos de cada uno, requiriendo la atención del moderador para cada aporte. El moderador

elige, entre todos los expertos, la contribución mas prometedora y la escribe en la pizarra. Este proceso continúa de forma iterativa hasta que el problema es resuelto.

En términos mas precisos, el **pizarrón** podría ser una base de datos que representa la memoria del sistema de resolución de problemas. Los **expertos** son subsistemas modulares denominados **fuente de conocimientos**, que representan los distintos puntos de vistas, estrategias y tipos de conocimiento de cómo resolver el problema. Este paradigma de resolución de problemas incluye:

- Sistemas basados en reglas
- Redes neuronales
- Sistemas de lógica difusa
- Algoritmos genéticos
- Sistemas legados y/o tradicionales

Un sistema de control representa la función del moderador, comprendido por detectores de eventos y coordinadores de agenda, que controla la interacción entre la pizarra, fuentes de conocimiento y fuentes externas como información de usuarios y otros sistemas de control.

Los detectores de eventos actualizan la información de la pizarra provista de las fuentes externas. El coordinador de la agenda elige el experto que puede escribir en la pizarra.

En el ambiente académico, los sistemas con modelos arquitectónicos de tipo backboard son muy eficientes, pero únicos y artesanales, requiriendo grandes esfuerzos en investigación, diseño, desarrollo y mantenimiento.

Los sistemas con arquitecturas Blackboard poseen los siguientes características:

1. **Integración de fuentes de conocimientos dispares**, manejados de manera transparente por el sistema de control.
2. **Modularidad**. Existe una independencia ente cada una de las fuentes de conocimientos facilitando el desarrollo y mantenimiento.
3. **Flexibilidad**. Con arquitecturas Blackboard los sistemas se adaptan fácilmente a los requerimientos cambiantes.

4. **Reuso**. Obtenida de las siguientes maneras

    a. La independencia de las fuentes de conocimiento hacen posible la construcción de otros sistemas utilizando las mismas fuentes.

    b. Sistemas legados pueden ser preservados e incorporados como parte del conocimiento base.

    c. El Blackboard en si mismo es una aplicación independiente que puede ser utilizada en otros dominios.

5. **Extensibilidad.** Nuevas fuentes de conocimiento pueden ser desarrolladas y agregadas sin impactar en el sistema existente.

Para resolver un problema de un dominio específico, se necesitan necesariamente las fuentes de conocimiento que representan la experticia en el dominio. Si se necesita interacción con fuentes externas, será necesario poner en práctica mecanismos de comunicación específicos para cada fuente.

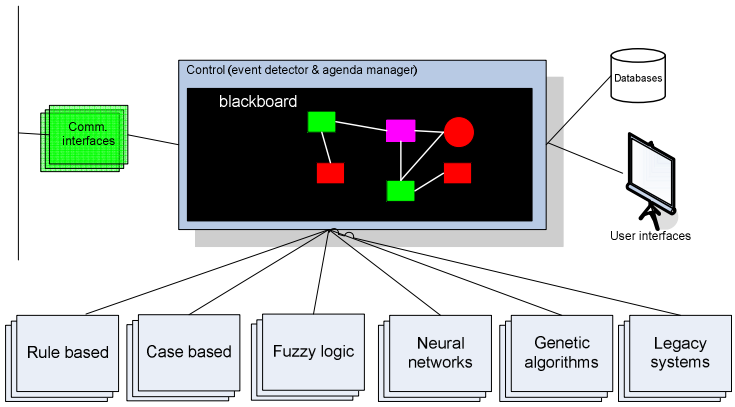


Ilustración 4 - Arquitectura Blackboard

Layered (en capas)

Los elementos de este tipo de arquitecturas son layers. Si bien existen varias traducciones de la palabra, utilizaremos el término en inglés para evitar confusiones.

Cada layer tiene una responsabilidad bien definida que, para llevarla a cabo, requiere comunicación con las capas subsiguientes. La organización de los layers se puede mostrar tanto horizontalmente como verticalmente. En el

primer caso, una layer se puede comunicar con los layers de abajo. En el segundo caso, un layer se comunica con los layers inmediatamente a su derecha. Esta última es la que elegimos para realizar los gráficos pertinentes.

Si bien en la teoría, un layer no debería poder interactuar directamente con otro layer si no está inmediatamente a su lado (arquitecturas layered opacas), en la práctica no siempre ocurre por cuestiones de performance o simplicidad. El hecho de que no sea un modelo puro de layers atentará contra la mantenibilidad. Es necesario hacer un análisis de costo-beneficio y definir pautas claras de cuando es posible sobrepasar capas.

Las principales ventajas de este estilo son:

- Desacoplamiento: Se puede desarrollar un layer aún sin disponer de los layers con los que debe colaborar. Alcanza tan solo con conocer el "contrato" de dichos layers. El contrato será el conjunto de servicios que ofrece un layer. Este desacoplamiento favorece la división de trabajo en el equipo de desarrollo y el testing aislado.
- Es posible cambiar implementaciones de layers (con el mismo contrato) sin modificar las restantes.
- Si se respeta el desacoplamiento y se mantiene el modelo lo mas opaco que se pueda, un cambio en el contrato de dos layers no debería afectar a los demás. Por ejemplo, supongamos que se tienen los layers A, B y C. Si cambia el contrato entre B y C no debería tener que modificar A (asumiendo una arquitectura opaca donde A no consume servicios de C).
- Reuso: Un mismo layer puede ser utilizado o consumido por layers para distintos propósitos.

La performance, si no se tiene en cuenta en las decisiones arquitectónicas, puede ser un problema en este tipo de arquitecturas. El estilo layered estimula la creación de múltiples layers para obtener todos los beneficios que trae consigo el desacoplamiento de las distintas partes del sistema. Hay que evitar el sobre-diseño de layers que puede traer más problemas que soluciones. Además, como se mencionó anteriormente, en ciertos casos es inevitable que la mantenibilidad "ceda un poco de pista" para dar paso a una mejora de la performance.

A continuación mostraremos la evolución de este estilo arquitectónico. Comenzaremos desde el momento en que no tenían sentido pensar en layers para llegar a arquitectura n-layered, pasando por arquitectura conocidas como la de 2 y 3 layers.

Durante el desarrollo de sistemas batch, sistemas que solo interactuaban con archivos y no tenían interacción hombre máquina, no existía la necesidad de layers.

Dos layers

A partir de los años 90, comenzó a surgir la idea de layers con los sistemas client-server. Estos sistemas se consideraban "sistemas de dos layers": el layer de cliente y el layer de servidor como muestra la siguiente figura.

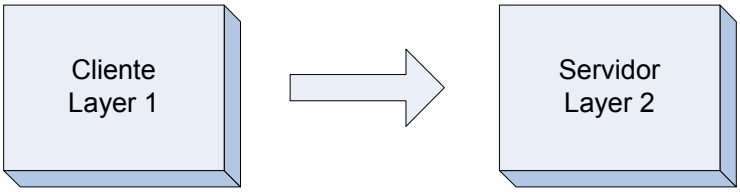


Ilustración 5 - Arquitectura Layered - 2 Layers

En los primeros sistemas dos layers, los clientes no solo implementaban la GUI sino también la lógica de la aplicación. El servidor, generalmente, se encargaba solo de la persistencia de los datos. Por lo general era una base de datos. Los datos, podían ser consultados y modificados desde el cliente utilizando algún lenguaje de consultas y de ABM (SQL por ejemplo).

Para sistemas que solo tenían que implementar operaciones de consultas, altas, bajas y modificaciones, este esquema funcionaba bien y era muy productivo. Muchos de los ambientes de desarrollo brindaban herramientas WYSIWIG (What You See Is What You Get) que facilitaban la construcción de los clientes en este tipo de arquitecturas. Por ejemplo, proveían la posibilidad de realizar fácilmente *bindings* entre los *widgets* de la GUI con campos de la tabla. Este binding, luego de configurado, sincronizaba el dato que mostraba el widget con el valor que tenía el campo en la BD. Algunos ejemplos: ambientes de desarrollo para Delphi y Visual Basic.

El problema surgió cuando se quiso utilizar la misma estrategia (aquella de tener la lógica implementada en el cliente), aún cuando esta lógica se hizo mucho más complicada. Una herramienta que fue diseñada para desarrollar GUI e interactuar con datos era ahora utilizada para escribir reglas, validaciones y cálculos complejos. Como consecuencia, la lógica de las GUIs comenzó a ser cada vez más compleja, con código replicado, difícil de mantener y de testear. Además, cuando surgió la idea de poder presentar la misma aplicación en distintos clientes (cliente de escritorio y cliente web por ejemplo)[9], la solución era reescribir toda la lógica en ambos cliente, solución que implicaba replicar código y complicar el mantenimiento.

Para evitar algunos de los mencionados problemas, se buscaron alternativas en las cuales la lógica se escriba del lado del servidor y los clientes solo tengan que consumirla para poder dibujar las GUIs.

La primera idea que surgió, considerando que en ese momento el servidor era en general una base de datos, fueron los *stored procedures*.

Este esquema incitaba más a la modularización que el anterior, pues estaba pensado para escribir la lógica y no para dibujar GUIs. Además, se resuelve el problema de múltiples clientes que necesitan la misma lógica.

Si bien se resuelven algunos de los problemas que presentaba la primera idea de arquitecturas dos layers, el lenguaje que ofrece la BD es poco natural para expresar las complicadas reglas, validaciones y cálculos. La base de datos fue diseñada para persistir los datos y no para escribir algoritmos complicados que consuman estos datos.

3 layers

Al mismo tiempo que se fue haciendo popular el esquema client-server, el paradigma orientado a objetos empezaba a nacer.

Considerado como un paradigma muy expresivo y natural para diseñar e implementar la lógica de dominio, fue la causa del surgimiento de un nuevo layer que permita separar la lógica, de los datos. Este layer es denominado "layer de dominio".

No es obligatorio el uso del paradigma orientado a objetos en este layer, pero es el más utilizado. Principalmente en aplicaciones Enterprise, en las cuales abundan validaciones, cálculos y reglas complicadas.

[9] Cliente escritorio es una aplicación o un subsistema que accede de manera remota a otras aplicaciones o a un servidor.

Con este nuevo layer nace la arquitectura de tres layers, que se muestra en la siguiente figura

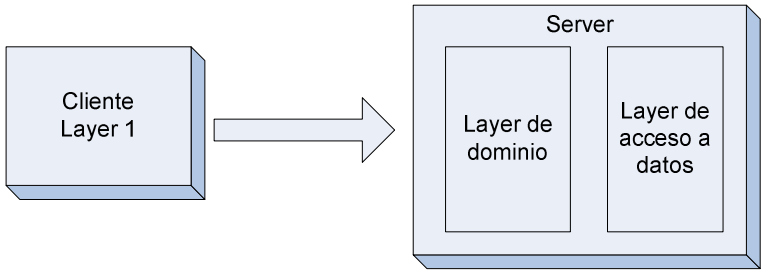


Ilustración 6 - Arquitectura Layered- 3 Layers

A continuación se describen las responsabilidades de cada una de las capas

Layer de presentación: Interfaces gráficas responsables de mostrar información e interpretar la interacción del usuario para ejecutar los comandos correspondientes que serán quienes consuman la lógica de dominio. Es responsable de hacer validaciones básicas de cliente.

Layer de dominio: Lógica de dominio. Es responsable de realizar validaciones de datos ingresados por el usuario y de ejecutar la lógica de dominio correspondiente, que en aplicaciones Enterprise frecuentemente incluirá cálculos complejos y reglas del dominio complicadas.

Layer de dataSource: Comunicación con el mecanismo de persistencia que se utilice. En general una base de datos. Si fuera una base de datos relacional, este layer será el responsable de realizar el mapeo objeto-relacional para lograr mapear los objetos en tablas y los atributos en campos de esas tablas.

N layers

A medida que los sistemas se tornaron más exigentes con sus arquitecturas, nuevos layers fueron surgiendo. Este surgimiento de nuevos layers permitió clarificar y distinguir la responsabilidad de cada elemento de una arquitectura en layers, además de reforzar el desacoplamiento necesario para favorecer el mantenimiento y extensión de las aplicaciones. Es así como surgen las arquitecturas n-layered.

Durante el caso de estudio se propondrá una arquitectura N-layered para la aplicación en cuestión, mostrando las responsabilidades de cada uno de estos layers.

3.6 Patrones de arquitectura

Definición

Un patrón codifica conocimiento específico acumulado por la experiencia en un dominio. Es una solución a un problema en un contexto

Christopher Alexander, 1977: "Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, y luego describe el núcleo de la solución a ese problema, de tal manera que puedes usar esa solución un millón de veces más, sin hacer jamás la misma cosa dos veces."

Alexander era arquitecto. Como ejemplo de patrones proponía: galería, paseo, patio compartido, columnata, estacionamiento.

Existen distintos niveles de patrones como lo muestra la siguiente tabla:

	Descripción	***Problemas que resuelve***	***Soluciones***	***Fase de Desarrollo***
Patrones de Arquitectura	Relacionados a la interacción de elementos de la arquitectura	Problemas arquitectónicos, adaptabilidad a requerimientos cambiantes, performance, modularidad, acoplamiento	Patrones de llamadas entre objetos (similar a los patrones de diseño), decisiones y criterios arquitectónicos, empaquetado de funncionalidad	Diseño inicial
Patrones de Diseño	Conceptos de ciencia de computación en general, independiente de aplicación	Claridad de diseño, multiplicación de clases, adaptabilidad a requerimientos cambiantes, etc	Comportamiento de factoría, Clase-Responsabilidad-Contrato (CRC)	Diseño detallado
Patrones de Análisis	Usualmente específicos de aplicación o industria	Modelado del dominio, completitud, integración y equilibrio de objetivos múltiples, planeamiento para capacidades adicionales comunes	Modelos de dominio, conocimiento sobre lo que habrá de incluirse (p. ej. *logging* & reinicio)	Análisis

Patrones de Proceso o de Organización	Desarrollo o procesos de administración de proyectos, o técnicas, o estructuras de organización	Productividad, comunicación efectiva y eficiente	Armado de equipo, ciclo de vida del software, asignación de roles, prescripciones de comunicación	Planeamiento
Idiomas	Estándares de codificación y proyecto	Operaciones comunes bien conocidas en un nuevo ambiente, o a través de un grupo. Legibilidad, predictibilidad.	Sumamente específicos de un lenguaje, plataforma o ambiente	Implementación, Mantenimiento, Despliegue

Serán los patrones arquitecturales, combinado con los estilos arquitectónicos y la experiencia en otros proyectos los que nos permitan reusar soluciones que facilitarán el armado del marco de arquitectura.

Existen varias categorizaciones de patrones de arquitectura. Una de las más difundidas y utilizadas es la que brinda Martin Fowler en su libro *Patterns of Enterprise Applications* [PEA-Fowler]. A continuación se lista las categorías que propone y los patrones más importantes de cada una.

Patrones de Arquitectura de Aplicaciones de tipo Enterprise

Patrones que resuelven distintos problemas asociados a las capa de una Aplicación de tipo Enterprise. [Martin Fowler eaaCatalog]
La presentación se basa en el catálogo de Martin Fowler.
http://www.martinfowler.com/eaaCatalog/

Patrones estructurales de mapeo Objeto-Relacional

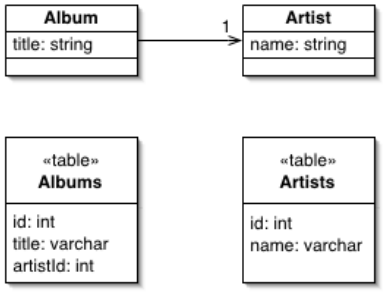
El rol que posee la capa de acceso a datos de una arquitectura es de comunicar los objetos de dominio con la base de datos. Siendo que la mayoría de las bases de datos son relacionales, es necesario contar con patrones que nos faciliten esta traducción.

La separación que provee una capa de acceso a datos permite trabajar de manera aislada con la lógica de dominio dejando afuera el diálogo que tiene el dominio con la base de datos. Si bien todo desarrollador debe conocer como es esta interacción, esta separación permite que expertos en bases de datos puedan optimizar el acceso y forma de interpretar los daros (SQL generados) para

lograr una mayor performance. De manera análoga, los diseñadores que modelen la capa de dominio podrán diseñar representado la realidad sin tener que preocuparse cómo y quién maneja de la persistencia.

Identity Field

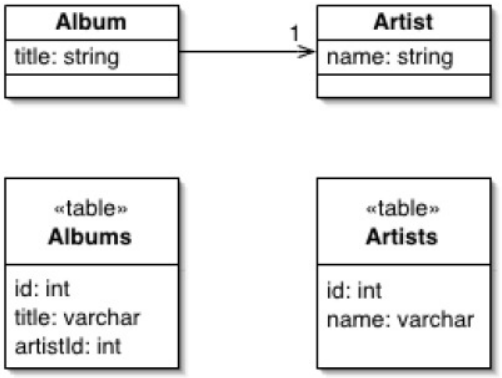
Guarda un campo *ID* de la base de datos en un objeto para mantener la relación univoca que existe entre un objetos y su representación de una fila en una tabla de una base de datos.



Las bases de datos relacionales, distinguen una fila de otra utilizando una identidad llamada *primary key*.
Los objetos en memoria quedan identificados mediante un atributo de identidad *Itentity Field* que se corresponde con la *primary key* que posee la tupla en la base de datos.

Foreign Key Mapping

Mantiene una relación entre objetos cuando existe una relalcion de *foreign key* en la base de datos.



Los objetos en memoria se referencian con otros directamente por sus referencias en memoria. Para persistir estas relaciones en la base de datos, es necesario guardar estas referencias. Por lo que *Foreign Key Mapping* representa esta referencia a un objeto, a un *foreign key* en la base de datos.

Patrones de modelado de la complejidad del Dominio

Al momento de interactuar con la lógica de dominio nos encontramos con problemas de acceso, coordinación y responsabilidades de comportamiento.

Nos referimos a los elementos de software que atenderán los pedidos solicitados por algún cliente para la ejecución de la lógica de dominio, la coordinación de las actividades que se disparan con la solicitud y los responsables de ejecutar las reglas del negocio y lógica de dominio.

Los siguientes patrones presentan alternativas para simplificar lo antes expuesto.

Transaction Script

Es la forma más simple de ordenar la lógica de dominio.

Organiza la lógica de negocio a través de procedimientos dónde cada uno maneja un simple *request* (pedido) desde la capa de presentación.

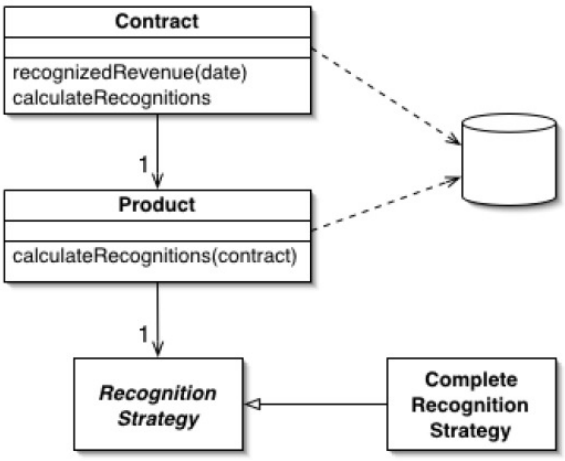
```
recognizedRevenue(contractNumber: long, asOf: Date) : Money
calculateRevenueRecognitions(contractNumber long) : void
```

La mayoría de la lógica que contienen las aplicaciones puede pensarse como una serie de transacciones. Una transacción puede ver cierta información organizada de una manera particular, otra puede hacer cambios sobre ésta. Cada interacción entre un cliente y un servidor contiene cierta lógica, en algunos casos puede ser simple como visualizar información y en otros puede incorporar una cierta cantidad de pasos de validaciones y cálculos.

Un *Transaction Script* organiza toda esa lógica como un solo porcedimiento, conectándose directamente a la base de datos, o a través de un delgado *wrapper* de ésta. Cada transacción tiene su propio *Transaction Script*, aunque algunas tareas comunes pueden volcarse en subprocedimientos.

Domain Model

Un modelo de objetos del dominio que incorpora comportamiento y datos.

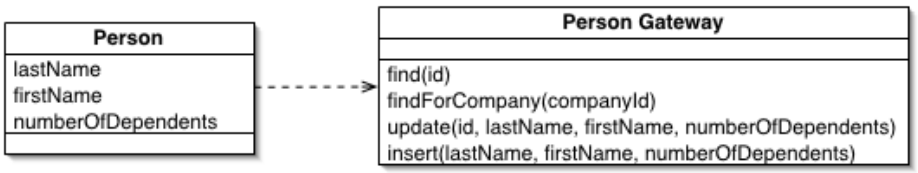


Cuando la lógica de negocio es muy compleja, y depende de muchos factores, los objetos son los indicados para modelarlo. Un Modelo de Dominio crea una red de objetos interconectados, donde cada objeto representa algo significativo e individual.

Patrones arquitecturales de acceso a datos

Table Data Gateway

Un objeto que actúa como *Gateway* a la base de datos. Una instancia que maneja todas las filas en una tabla.

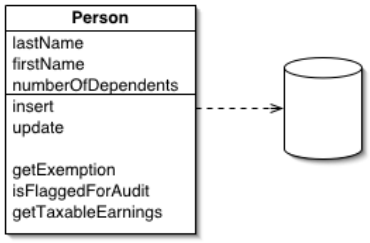


Mezclar código SQl en la lógica de la aplicación puede causar numerosos problemas. Los desarrolladores no suelen codificar SQL como lo harian los

administradores de una base de datos. Por esta razón es que es necesario centralizar en pocos lugares la utilización de este código.
Table Data Gateway contiene todo el SQL para acceder a una sola tabla o vista. Los demás elementos de la aplicación invocan sus métodos para acceder a los datos.

Active Record

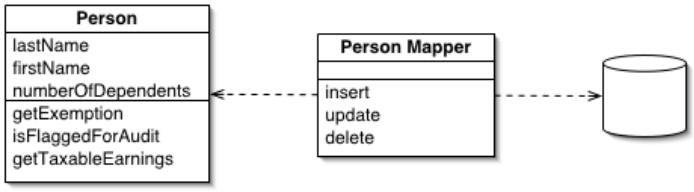
Un objeto que hace de wrapper a una fila de una tabla o vista de la base de datos encapsulando el acceso a los datos y agregando la logica del dominio a sus datos.



Un objeto que posee tanto datos como comportamiento y muchos de sus datos son persistentes y necesitan ser almacenados en una base de datos. *Active Record* pone la lógica de acceso a datos en el objeto de dominio, permitiendo que cada objeto de dominio sepa como cargarse y guardarse desde y hacia la base de datos.

Data Mapper

Una capa de Mappers mueve los datos entre objetos y una base de datos mientras mantiene la independecia entre ambas.



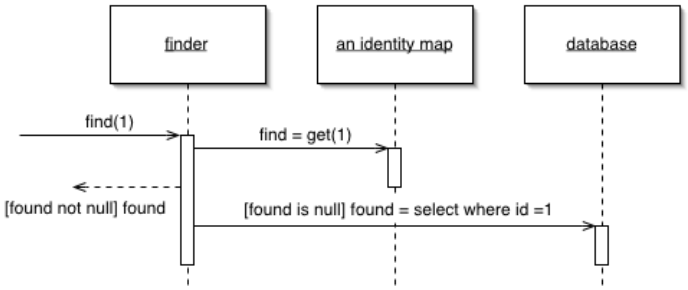
La estructura de los objetos en memoria y los datos persistidos en bases de datos relacionales es bastante diferente. Las partes de los objetos que representan colecciones y herencia no están presentes en las bases relacionales. La representación de la realidad del modelado de comportamiento del modelo de

objetos no siempre encaja con el modelado de la base. Si se lleva el conocimiento de la estructura de la base de datos a los objetos, perderíamos independencia ya que un cambio en la estructura de persistencia impactaría directamente en el modelado de objetos.
Data Mapper es una capa de software que separa los obejtos en memoria de su representación en la base de datos. Su responsabilidad es transferir los datos entre los dos y aislarlos a uno del otro. Con Data Mapper los objetos en memoria no necesitan saber que existe una base de datos, ni saber de código SQL, ni como son persistidos.

Patrones de comportamiento objeto-relacional

Identity Map

Se asegura que cada objeto sea cargado solo una vez manteniendo cada objeto cargado en un mapa. Cuando requiere un objeto busca en en el mapa si ya lo tiene cargado.



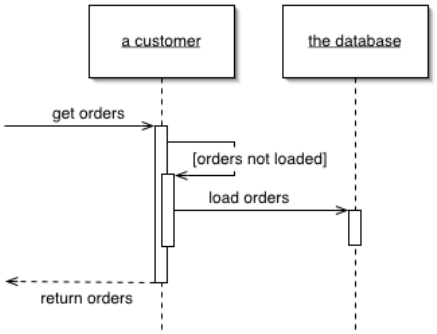
La recuperación de objetos desde la base de datos a memoria tiene su trasfondo peligroso. Si no se procura mantener una identificación correcta de los objetos que se tienen en memoria se podría caer en el problema de hacer una carga de un mismo registro de la base de datos en dos objetos distintos en memoria. Asociado a esto aparecen dos problemas no menores, problemas de performance, ya que se pueden hacer lecturas innecesarias a la base de datos

Un *Identity Map* mantiene un registro de todos los objetos que fueron cargados de la base de datos dentro de una transacción de negocio. Cuando se desea un objeto, se verifica que no este en *Identity Map* para ver si ya fue cargado.

Un problema que puede ocurrir es que se puede cargar la información de un mismo registro, de la base de datos, como dos objetos distintos. Cuando se actualizan ambos objetos con alguna modificación se sufrirá una demora innecesaria en la actualización de estos datos en la base de datos ya que generaría 2 sentencias de insert con la misma información para la misma tabla.
Asociado a esto existe un problema de performance, ya que si se cargan muchas veces un objeto en memoria, se necesitan numerosas peticiones a la base de datos para la recuperación de datos que ya existen en memoria.

Lazy Load

Un objeto que no contiene los datos solicitados, sabe cómo obtenerlos.



Lazy Load actúa en el proceso de la búsqueda de objetos relacionados a otro objeto. Permite la recuperación de relaciones de objetos, a demanda, y de manera "perezosa" haciendo que la recuperación sea efectiva luego de la verificación de que la solicitud de recuperación. De no ser necesaria la búsqueda de estos datos, se consigue un ahorro de tiempos.

Patrones de base

Layer Supertype

Una clase que es la base de todas las clases en su capa.
Todo el comportamiento común a los objetos de dominio se detallan en una clase de dominio padre de todas, que generaliza el comportamiento.

Patrones de concurrencia

Optimistic Offline Lock

Previene conflictos de concurrencia en transacciones de negocio, detectando un conflicto y realizando un rollback de la transacción.

Patrones metadatos en mapeo objeto-relacional

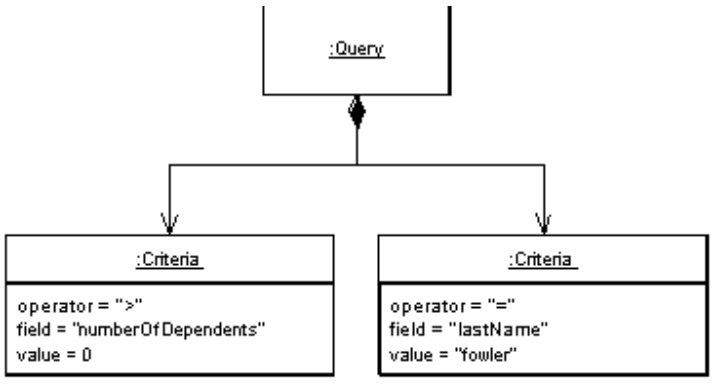
Metadata Mapping

Mantiene informacion del mapeo objeto-relacional como meta información

Para evitar duplicar la información de la correspondencia de los campos de las tablas de la base datos con los atributos de los objetos de dominio, se utiliza un mapeador de esta información para que pueda ser interpretada y mantenida de una manera más sencilla.

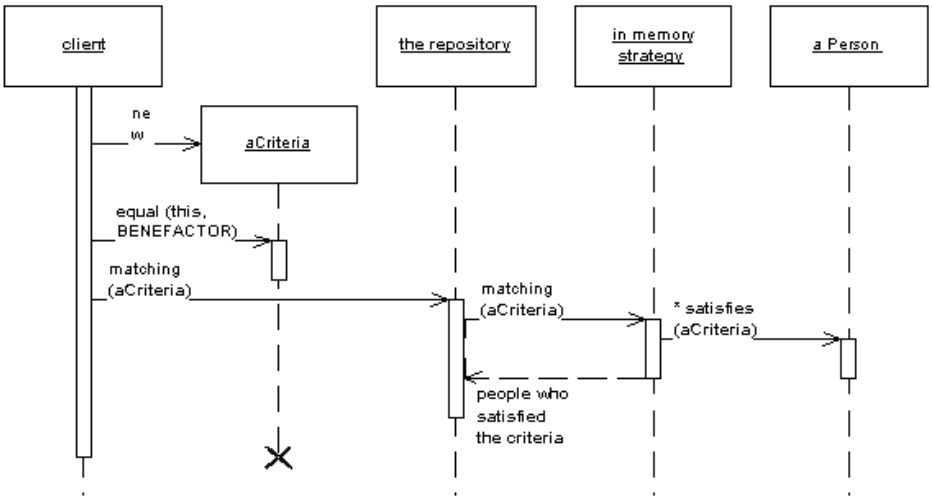
Query Object

Es un objeto que representa una consulta a la base de datos



Repository

Es un mediador entre los objetos de dominios y la capa de mapeo de datos.

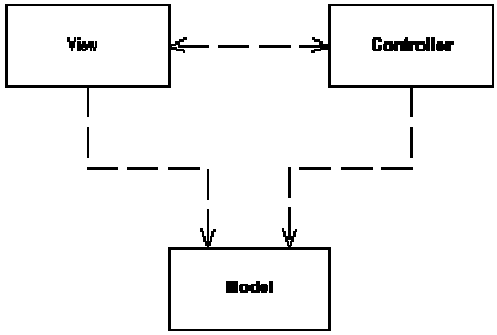


En sistemas con un modelo de dominio complejo se utilizan capas como las que proveen los Data Mappers que aíslan a los objetos de dominio de los detalles de la base de datos. En este tipo de sistemas es beneficioso tener otra capa de acceso a datos por sobre la capa de mapeos en donde se concentre la construcción de consultas. Esto resulta casi necesario en dominios en donde existe un gran número de objetos de dominios o se utilizan muchas consultas.
Un repositorio actúa de mediador entre los objetos de dominio y la capa de mapeo, actuando como si el dominio fuera una gran colección de objetos que están en memoria.

Patrones de presentación web

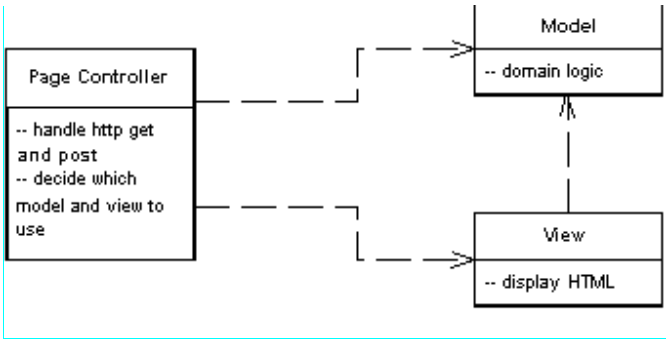
Model View Controller

Separa la interaccion de la interfaz de usuario en tres roles distintos.



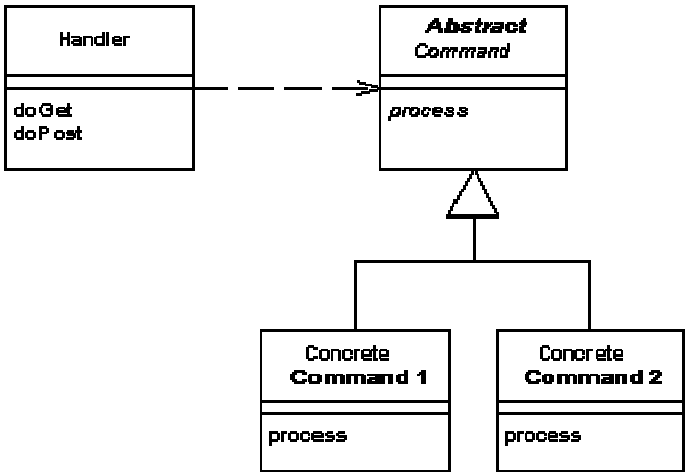
Page Controller

Un objeto que maneja el **request** de una acción especifica o una pagina en un sitio web.



Front Controller

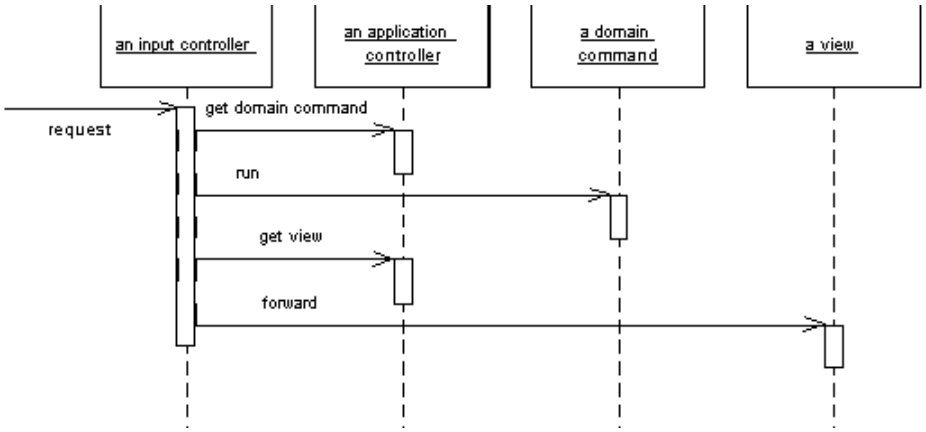
Es un controlador que maneja a todos los **request** de un sitio web.



En un sitio web complejo existe mucho comportamiento similar que posiblemente se necesite centralizar al momento de procesar un request. Estas cosas incluyen cuestiones como seguridad, internacionalización y vistas.

Application Controller

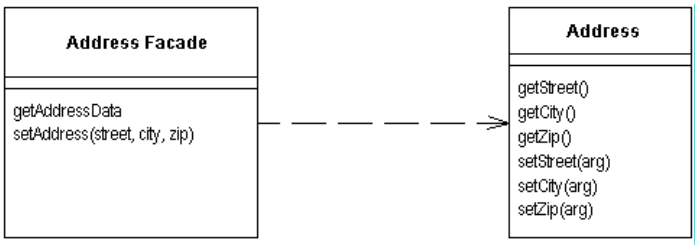
Centralización del manejo de la navegación y flujo de la aplicación.



Patrones de distribucion

Remote Facade

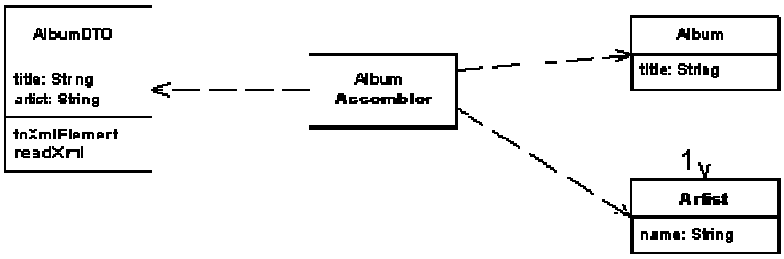
Provee una fachada de grano grueso para interactuar con un conjunto de objetos de interfase de grano fino para mejorar la eficiencia en el uso de red.



En un modelo orientado a objetos, en donde los objetos poseen una interfase pequeña para un gran número de objetos, se logra control del comportamiento, un mayor entendimiento de la lógica y abstracción.
Una de las consecuencias de este modelo es la gran interacción que existe entre el modelo y por lo tanto numerosas invocaciones, que cuando son remotas provocan una gran utilización de red.
Es necesario entonces generar una interfase que minimice el numero de llamadas entre objetos para realizar una actividad.

Data Transfer Object

Un objeto que lleva información entre procesos con el objetivo de reducir el numero de las invocaciones a metodos.



Cuando se trabaja en esquemas de interfases remotas, cada llamada es cara.
Como resultado de esto es necesario reducir el numero de llamadas remotas, tratando de aumentar la cantidad de información que se acarrea en cada solicitud remota.

Una forma de lograr esto es utilizar numerosos parámetros, pero esto no siempre es valido en lenguajes de programación que retornan un solo valor.
La soluicion a esto es crear un objeto de transferetncia DTO que pueda contener toda la información necesaria para la accion especifica que disparo la llamada remota.

Patrones de manejo de sesion

Client Session State

Guarda la el estado de la sesión del cliente.

Server Session State

Guarda el estado de sesión en el servidor persistiéndolo de una manera serializada.

Durante el caso de estudio, algunos de estos patrones serán utilizados para definir el marco de arquitectura.

Los patrones se combinan con el estilo seleccionado generando arquitecturas "conocidas". Por ejemplo si hablamos de una arquitectura con estilo layered que implementa el patrón Remote Facade (Application service) hablamos de una SOA.

PARTE IV: METODOLOGÍA PARA DISPONER DE UNA ARQUITECTURA DE APLICACIÓN

El resultado de la arquitectura, como describimos en la introducción, debe generar el ***soporte*** necesario para la construcción de los casos de uso, que ayude a garantizar el cumplimiento de los requerimientos no funcionales, asegure la calidad y maximice la productividad.

Equipo de trabajo

La experiencia que hemos adquirido demuestra que para aplicar correctamente la metodología, es necesario constituir un equipo de arquitectura que tenga a su cargo las siguientes responsabilidades:

- ✓ Definir el marco teórico de arquitectura
- ✓ Definir las tecnologías que se utilizarán en cada uno de los elementos del marco de arquitectura definidos
- ✓ Actualización de las tecnologías y frameworks utilizados.
- ✓ Evaluación de frameworks existentes en la comunidad que puedan ser de utilidad en el proyecto.
- ✓ Evaluación de herramientas
- ✓ Diseño y Desarrollo de Componentes y frameworks que resuelvan los problemas comunes de las aplicaciones Enterprise (componentes estructurales).
- ✓ Diseño y Desarrollo de componentes y frameworks que abstraen conceptos comunes del negocio (componentes funcionales).
- ✓ Soporte al equipo de desarrollo
    - o Soporte técnico a los desarrolladores
    - o Capacitación al equipo de desarrollo
    - o Resolución de problemas de performance.

Metodología propuesta

La metodología propuesta, tiene por objetivo organizar las actividades del equipo de arquitectura describiendo el alcance de cada una.

La metodología se organiza en etapas.

Cada etapa tiene por objetivo generar distintos artefactos. Estos artefactos serán insumos de actividades posteriores (en la misma o en subsiguientes etapas) para la generación de nuevos artefactos. La construcción de estos artefactos es incremental e iterativa. Es decir, un artefacto puede ser tomado para realizar una actividad aún sin estar terminado.

Cada etapa concluye con un artefacto de más alto nivel que los artefactos de las actividades.

Se reconocen tres grandes etapas metodológicas

Etapa 1: Marco de arquitectura. El artefacto generado en esta etapa es la arquitectura estructural.

Etapa 2: Generalización funcional. El artefacto generado en esta etapa es la arquitectura funcional básica.

Etapa 3: Extensión, corrección y mantenimiento de la arquitectura. El artefacto generado es la arquitectura terminada.

Etapa 1- Marco de arquitectura.

El artefacto generado en esta etapa es la arquitectura estructural.

Objetivos de la etapa:

Conseguir la primera versión de la arquitectura estructural que brinde el ***soporte*** necesario para comenzar con la construcción de los casos de uso. Este marco debe dejar claro al equipo de construcción como un caso de uso debe desarrollarse sobre la arquitectura.

¿En qué momento debe estar esta primera versión?

Es una tarea que debe estar terminada, en su primera versión, antes de comenzar el desarrollo. Si se comenzara a desarrollar sin la definición del marco

de arquitectura, será muy difícil reeducar a los desarrolladores y será impracticable una futura migración de lo ya desarrollado a la nueva arquitectura.

¿Cuáles son las cuestiones a las que debe dar respuesta el marco de arquitectura?

A continuación se describen los puntos a los que debe dar solución un marco de arquitectura. Tener resueltos estos puntos, previo a comenzar el desarrollo, **es un factor fundamental para estandarizar el desarrollo, para simplificar el futuro mantenimiento, para mejorar la productividad del equipo de implementación y para garantizar la calidad del producto generado**.

- ¿Como reusar componentes gráficas? Lo consideramos como una de las claves para mejorar la productividad del desarrollo. Según estadísticas y nuestra experiencia, el desarrollo de la GUI se lleva más del 50%. La construcción de componentes que se puedan reusar en distintas pantallas reduce notablemente este tiempo de construcción de GUIs.
- Como transferir objetos del servidor al cliente y viceversa.
- Como generar reportes
- Como particionar el sistema en varios subsistemas. Esta partición favorece el desacoplamiento entre los mismos, evitando los efectos colaterales en el mantenimiento del sistema. Un subsistema ve a otro como una caja negra a través de una interfaz bien definida sin preocuparse por su implementación interna.
- Como distribuir los elementos que componen un caso de uso sobre la arquitectura.
- Como manejar la seguridad
- Como realizar la persistencia
- Como desarrollar test de unidad, test de integración y test funcionales.
- Como realizar la automatización de test de regresión.
- Como interactuar con otros sistemas
- Como usar la tecnología y los frameworks.
- Como manejar errores.
- Como generar y consultar el log operacional.

- Como generar y visualizar la historia.
- Como generar información de auditoría.

¿Cuáles son los artefactos detallados generados en esta etapa?

1.1 Marco teórico de arquitectura

1.2 Primera implementación de la arquitectura

1.3 Manual de arquitectura

1.4 POC: Implementación de referencia

1.5 Prueba de carga

Analicemos en detalle cada uno de los artefactos que se deben generar:

1.1 Marco teórico de arquitectura

Ver capítulo 3.

1.2 Primera implementación de la arquitectura

Una vez definido el marco teórico de la arquitectura, es necesario proveer una implementación que sea la base sobre la cual se van a desarrollar los casos de uso.

Sobre la base de los estilos y los patrones de arquitectura como medios de reuso, existen plataformas que implementan algunos estilos y patrones. Estas plataformas serán la base de la implementación completa de la arquitectura.

Dependiendo de la magnitud del proyecto, será necesario combinar la plataforma elegida con otros frameworks, tecnologías, y seguramente, **desarrollos propios**.

Como ejemplo de plataformas Java podemos mencionar *JEE* (Java Enterprise Edition) y Spring. Como implementación de un patrón de arquitectura implementado a través de un framework aparece Hibernate que implementa el

patrón "*Metadata Mapping*". En el caso de estudio abordaremos con más detalle las plataformas y frameworks más utilizados en la industria.

Para obtener esta primera implementación se desarrollan las siguientes actividades

1.2.1- Definir las tecnologías y frameworks existentes en la comunidad que se utilizarán en cada uno de los elementos del marco de arquitectura

En esta actividad, es necesario elegir adecuadamente las tecnologías y frameworks que se utilizarán de manera de poder implementar la aplicación siguiendo los lineamientos definidos en el marco teórico.

Esta elección incluye desde el lenguaje de programación hasta frameworks de la comunidad que puedan ser de utilidad para el proyecto.

Como toda investigación, el primer paso comienza conociendo todo lo que existe para el tema que se va a estudiar.

Cuando se presenta un abanico de alternativas, la tarea requiere de pruebas de concepto de los distintos productos para justificar de manera fundada la decisión que se adopte.

Como ejemplo tomemos una arquitectura con tecnología Java que presentara sus GUIs a través de una capa web. Actualmente existen más de 50 frameworks para desarrollar aplicaciones web usando tecnología Java. Si bien la comunidad aporta mucho en foros y artículos, la elección no es una tarea fácil.

Una vez definidos los frameworks, es imprescindible definir las guías que expongan detalladamente la utilización de cada uno de ellos, como así también la interacción entre los mismos.

Estas guías deben ser rigurosamente respetadas por el equipo de implementación para aplicar adecuadamente el marco de arquitectura definido. Este apego a las guías definidas garantiza la estandarización del desarrollo y facilita el futuro mantenimiento.

Las pautas definidas en estas guías deben considerar la flexibilidad necesaria para cubrir los casos que se presenten.

1.2.2-Evaluación de herramientas

Existen herramientas que simplifican el uso de las tecnologías y frameworks. Esas herramientas son:

- Ambiente de desarrollo. Como ejemplo Eclipse.
- Herramientas de generación de código
- Herramientas para simplificar la escritura de archivos de configuración
- Herramientas de compilación
- Herramientas para realizar la ejecución automática de test de unidad, de integración y funcionales
- Herramientas para la administración de la configuración
- Herramientas para deploy de las aplicaciones

En ciertos casos, el hecho de disponer una herramienta para simplificar el desarrollo con un framework puede ser la clave de una decisión para el uso del framework. Por ejemplo, al momento de elegir un framework para desarrollo web, uno de los puntos importantes a considerar es si existen herramientas WYSIWYG (What You See Is What You Get) para desarrollar las pantallas a través de una paleta de componentes gráficas y drag&drop sobre la pantalla.

1.2.3-Diseño y Desarrollo de las primeras versiones de componentes y frameworks propios (componentes estructurales)

Como mencionamos en el punto anterior, suelen presentarse distintos frameworks y tecnologías para la implementación de los distintos elementos de las arquitecturas. **Sin embargo, en aplicaciones Enterprise de gran tamaño es frecuente que aparecezca la necesidad de proveer soluciones genéricas ad-hoc.**

Estas soluciones genéricas podrán basarse en frameworks existentes y, en tal caso, serán *wrappers* de las mismas.

En la implementación de la arquitectura, un desarrollo ad-hoc cubrirá algunos de los elementos definidos en el marco teórico de la arquitectura. Para esto, y como se expresó en capítulos anteriores, se deberán conocer los atributos de calidad que incumben a ese desarrollo y

posiblemente requerimientos funcionales que sea pertinentes considerar para el desarrollo de la arquitectura.

Consideramos que una buena práctica en la implementación de la arquitectura, es encapsular el comportamiento en componentes y frameworks.

Algunos ejemplos de componentes y frameworks, que describiremos en detalle en el capítulo del caso de estudio, son:

- Framework para llevar objetos al cliente remoto
- Framework que abstraiga el mapeo objeto-relacional
- Framework para maximizar la productividad en el desarrollo de GUIs
- Framework para desarrollar reportes
- Framework para realizar búsquedas de manera genérica
- Framework para la ejecución de reglas de precondición.

De la misma manera que se definen pautas de cómo usar los frameworks de la comunidad, es muy importante definir guías de uso de las componentes ad-hoc y hacer control de calidad para verificar que los usuarios de los frameworks, respeten los lineamientos de uso.

1.3 Documentación

Las decisiones que se tomen deben quedar escritas en el documento del marco de arquitectura de la aplicación. Esta documentación debe contener al menos tres partes

- Arquitectura de la aplicación: Dirigida a todo integrante del proyecto(analistas, testers, desarrolladores, diseñadores, entre otros). Explica los rasgos generales de la arquitectura. Debe formar parte de la capacitación inicial.
- Guía de uso: Define el conjunto de pautas que los diseñadores y desarrolladores deberán tener en cuenta durante la implementación de los casos de uso. Es buena práctica utilizar esta guía para capacitar al equipo de trabajo previo al comienzo del desarrollo. Se obtienen mejores resultados

cuando la capacitación incluye práctica. Estas prácticas promueven debates interesantes.

- Guía de solución: Es importante dejar documentadas tanto las decisiones tomadas como así también las opciones descartadas. Estas últimas sirven como base de conocimiento al equipo de mantenimiento de la arquitectura, que no siempre es el mismo que la definió.

1.4 POC: Implementación de referencia

Es necesario acompañar las decisiones arquitectónicas que se toman con POCs(Proof of concepts) para validar la correctitud de la solución completa.

Muchas veces ocurre que decisiones que en teoría funcionan muy bien, son descartadas tempranamente gracias a la detección de problemas en estos POCs.

Es necesario desarrollar estos POCs usando casos que permitan probar los distintos aspectos de la arquitectura y la combinación de frameworks propios y de la comunidad utilizados para la implementación.

1.5 Prueba de carga

La prueba de carga es imprescindible para detectar futuros problemas de performance. Si bien los POCs permiten determinar que la arquitectura sirve como base para el desarrollo del producto, las pruebas de carga permiten exigir la arquitectura para probarla en casos extremos. Estos casos extremos simulan gran cantidad de usuarios concurrentes, accediendo a grandes volúmenes de datos y ejecutando distintas operaciones.

Estas pruebas de carga tienen que ser automatizadas para correrlas junto con el resto de los test de la arquitectura, para detectar efectos inesperados en la performance, como consecuencia de la extensión y mantenimiento de la arquitectura.

Etapa 2: Generalización funcional.

El artefacto generado en esta etapa es la arquitectura funcional básica.

En esta etapa un equipo de "frameowrks", de existir en un proyecto, tomaría participación activa. Si bien es algo interesante para debatir, nuestra propuesta es

que esta generalización quede en el marco del equipo de arquitectura. De todas maneras, se puede distinguir como un subequipo que llamaremos "arquitectura funcional".

En esta etapa, a través de relevamiento con analistas y usuarios, se deben detectar las cuestiones funcionales generales. A partir de ellas se diseñan y desarrollan frameworks que serán usados por el equipo de diseño y construcción para desarrollar los casos de uso. El objetivo de estos frameworks será simplificar el desarrollo, maximizar la productividad del equipo de desarrollo, generar un producto de alta calidad y simplificar el futuro mantenimiento.

Etapa 3: Extensión, corrección y mantenimiento de la arquitectura.

El artefacto generado es la arquitectura terminada.

La implementación de la arquitectura debe ser vista como cualquier otro producto de software. Es por esto, que tendrá una etapa de mantenimiento y extensión.

Los requerimientos seguirán llegando de los distintos equipos de trabajo: Análisis, Diseño y Desarrollo. Siendo que los clientes son muchos, es importante seguir metodologías para el manejo de cambios centralizando los pedidos y utilizando herramientas que faciliten la tarea. Además, es importante organizar entregas de versiones fixes que agrupen varios requerimientos.

PARTE V: CASO DE ESTUDIO

e-Sidif

Antecedentes y descripción del dominio

Sancionada la Ley Nº 24.156 de Administración Financiera y de los Sistemas de Control del Sector Público Nacional el 30 de septiembre de 1992 se desarrolla un sistema integrado de información financiera (SIDIF) cuyo principal propósito es la **formulación del presupuesto nacional y registro de la ejecución presupuestaria**.

El SIDIF fue concebido como un sistema integrado, compuesto por diversos subsistemas y módulos dentro de una visión funcional, que contempla la distribución de la base de datos lógica, en una base de datos central y tantas bases institucionales cuantos fueren los SAF (Servicios de Administración Financiera). Esta estructura física propuesta para el SIDIF está soportada por arquitecturas abiertas y redes locales trabajando en la modalidad "cliente/servidor".

Bajo este esquema operativo, la Secretaría de Hacienda se comunica, a través del SIDIF Central con los sistemas periféricos instalados en los organismos (sistemas locales), con el sistema de gestión para la Unidades Ejecutoras de Préstamos Externos y demás aplicaciones que interactúan con la base de datos central. A través de esta comunicación la base central concentra el registro de la ejecución presupuestaria. La información de gestión permanece en las bases locales.

Dentro del SIDIF convivían una variedad de sistemas locales en las distintas entidades con características diferentes. Esta diversidad de aplicativos incrementó el costo de mantenimiento y de replicación de las adecuaciones en los sistemas locales, lo que implicaba en algunas oportunidades demoras para su disponibilidad.

Ante esta situación se da comienzo a una etapa de transformación que consiste en un proceso de homogeneización de los sistemas locales mediante la implementación de un nuevo sistema SIDIF Local Unificado.

Este proceso, que aún se está llevando a cabo, implica reemplazar los sistemas locales existentes en distintos organismos por una versión unificada con mejoras en cuanto a la provisión de información confiable para la alta gerencia de las entidades, el ajuste de procedimientos de compras, presupuesto, contabilidad y

tesorería y la reducción de aplicativos complementarios requeridos para soportar la gestión.

Habiendo cumplido esta primera etapa de transformación que ha introducido importantes mejoras a través del SLU, se plantea un nuevo desafío, la denominada segunda transformación que se materializa con la extensión del alcance y renovación tecnológica con el desarrollo del sistema **e-Sidif**.

Es este sistema el que busca mejorar la calidad del SIDIF incorporando las innovaciones y posibilidades que en materia de tecnología de comunicaciones se han producido en la última década, aprovechando la flexibilidad y reducción de costos que posibilita la modernización, introduciendo cambios que sirvan como base para una mayor orientación de la gestión pública a resultados y una ampliación del alcance y vinculación del sistema.

Este capítulo utiliza al eSidif como un caso de estudio de la construcción de una arquitectura de aplicación Enterprise. **Las características del mencionado proyecto permiten aplicar la metodología propuesta en esta tesis.**

Como parte del caso de estudio se consideraron los siguientes desafíos que permiten validar la metodología y definición de arquitectura:

Desafíos tecnológicos

- Reemplazo gradual de funcionalidad de sistemas legados
- Gran tamaño del sistema (14000 UCP)
- Despliegue gradual de funcionalidad
- Sistema adaptable a diferentes entornos y tamaños de organizaciones
- Repositorio único central de toda la información de gestión y de registro.
- Facilidad de administración de versiones de aplicación y software de base
- Arquitectura multicapa

- Bajo costo de hardware y software

Desafíos metodológicos

- Alta interacción de equipos de trabajo
- Alta productividad
- Bajo costo de mantenimiento
- Bajo costo de adaptación a los cambios
- Utilización de herramientas y metodologías modernas para el diseño

¿Por qué este caso de estudio?

Por su complejidad, por ser un sistema Enterprise, por el desafío tecnológico y arquitectónico que este proyecto presenta. La metodología y la definición de la arquitectura detallada requerida en un sistema de esta envergadura presenta situaciones complejas para ser analizadas y tratadas como un caso de estudio.

Arquitectura e-Sidif

Definición del Marco de Arquitectura para E-Sidif

Marco teórico

Documento de Visión

Siguiendo con la metodología propuesta, el primer paso para definir el marco teórico de la arquitectura, es definir los requerimientos de alto nivel y mencionar los atributos de calidad necesarios para este sistema proporcionando el porqué de

estas necesidades. Para este propósito se necesita de un documento de visión del sistema que detallaremos en un anexo y solo presentaremos los puntos más sobresalientes.
Anexo Documento de Visión.doc
Se destacan los siguientes puntos representativos del documento de visión que hacen a la definición de la arquitectura:

Perspectiva del sistema

El sistema surge como una tercera etapa de actualización de un software de la administración financiera del estado. Esta etapa ocurre como una transición en donde el nuevo sistema deberá convivir y relacionarse con otros sistemas existentes a los cuales debe proveer de una interfaz clara y definida y que necesita que se modifiquen interfaces de algunos sistemas con los cuales se relacionan.

Características funcionales

A diferencia del sistema que reemplaza, el e-Sidif deberá utilizar una Base de Datos única que centralice toda la información para una gestión unificada, y además permitir una descentralización operativa.

Restricciones

Debido al contexto donde este sistema es utilizado, debe contemplar que los usuarios puedan realizar la operatoria de la gestión por fuera del sistema.

Características Técnicas

El diseño del sistema e-Sidif será incremental y orientado a objetos, permitiendo una conceptualización del dominio mas natural y rica.
Tanto la metodología como la tecnología se ajustan a los requerimientos técnicos que una aplicación de tipo Enterprise requiere.

Características Generales de la Arquitectura

Los entes gubernamentales se encuentran en el medio de una transformación que está impulsando el uso de estándares para intercomunicación y la automatización del procesamiento de las transacciones posibilitando reducir costos y tiempos redundando en mayores beneficios para dichas instituciones y para sus contribuyentes.
Asimismo la imposición de regulaciones globales y locales han llevado a la adopción de nuevas tecnologías para actualizar e integrar sus sistemas, muchas veces resultado de un procesamiento manual. El mercado está tendiendo hacia el desarrollo de soluciones abiertas que integren y hagan uso de las inversiones existentes abriendo un sinnúmero de posibilidades para aplicaciones centrales al funcionamiento como e-Sidif.

Como fue mencionado la aplicación tendrá un alto enfoque transaccional y de integración, que estará basado sobre estándares ampliamente difundidos en la industria. Uno de los conceptos mencionados es el de SOA.

El enfoque orientado a servicios y su aplicación en instituciones gubernamentales, el cual es similar en algunos aspectos al enfoque orientado a objetos que adoptó la industria a mitad de los 90's, está siendo aplicado especialmente por las instituciones de mayor poder económico y generalmente early adopters de la tecnología.

SOA está basado sobre dos principios de aquella era, modularidad – subdividiendo el sistema en partes más pequeñas – y encapsulamiento – utilizando una interfaz claramente definida para aislar las características internas del exterior. Estos principios permiten que las aplicaciones SOA sean de fácil desarrollo y mantención pues se puede alcanzar una clara separación de roles y alcanzar el desarrollo de componentes que luego puedan ser reutilizados. Al ser independiente de la tecnología permite que mundos diferentes legados o no puedan intercomunicarse.

Asimismo la evolución de eSidif desde una arquitectura client-server con la filosofía de fat-client a una arquitectura orientada a servicios guiada por los objetivos de negocio incorporando :

Modelo de N-Capas permitiendo mayor modularidad y flexibilidad.

Independencia o portabilidad de sus componentes de infraestructura de software y hardware.

Capacidades más sencillas de deployment.

Interfaz rica con alcance a cualquier tipo de cliente desktop (Windows / Linux).
permitirá a e-Sidif :

- Administrar la complejidad inherente a un sistema de su tamaño.
- Ofrecer una flexibilidad que permita "componer" funcionalidades a partir de servicios existentes dando una respuesta más rápida a requerimientos de negocio.
- Facilitar la reutilización a través de la continua disponibilidad de esos servicios de negocio de forma bajamente acoplada, ahorrando así recursos de manera significativa.

Estructura global del sistema

El objetivo de esta sección es definir la estructura global del sistema sobre la base de la información descripta en el documento de visión.

Como se describió en el capítulo anterior, existen estilos arquitectónicos que permiten reusar estructuras globales de los sistemas que funcionaron correctamente en aplicaciones de características similares.

El estilo seleccionado nos sirve de punto de partida aunque será necesario adecuarlo a las características propias del caso de estudio.

El estilo arquitectónico que guía la estructura global en el caso de estudio, es el *N-layered*. Esto significa, como se describió en el capítulo anterior, organizar el sistema en *layers* que se comunican para poder completar la funcionalidad requerida. Cada layer tiene una funcionalidad y un alcance bien definidos que detallaremos a continuación.

La arquitectura de layers se puede describir usando distintos niveles de detalle. Se comenzará describiendo la estructura global del sistema usando layers de alto nivel. Luego, los layers se irán explotando para mostrar su organización interna (posiblemente también a través de layers).

Cada vez que explotemos un elemento de la arquitectura en layers, describiremos:

- Cuáles son los layers?
- Un gráfico de interacción entre los layers
- Descripción de la responsabilidad de cada layer.

Nuestra estructura de más alto nivel divide el sistema en 4 grandes layers:

- Layer de presentación
- Layer de lógica de negocio
- Layer de acceso a datos
- Layer de fuente de datos e información.

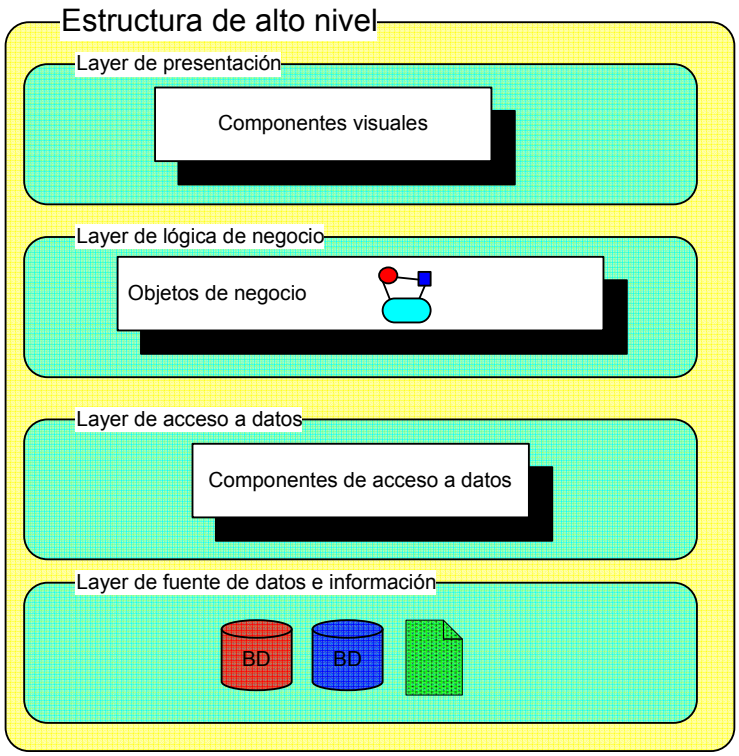


Ilustración 7 - Gráfico de Layers

Layer de presentación

La lógica de presentación conforma parte de la "vista" o interfaz gráfica [GUI] de la aplicación y de cómo ésta organiza la interfaz de usuario conectándose con el layer de la lógica de negocios. El hecho de tener la presentación como un layer separado de la lógica de negocios permite desacoplar la lógica de negocios de la presentación permitiendo alterar la interfaz de la aplicación o agregar un nuevo canal sin necesidad de alterar la lógica de negocios. Esta independencia del modelo de servicios permite acoplar rápidamente nuevas tecnologías de presentación preservando la inversión a realizar.

Existirán dos tipos de clientes diferentes. Un cliente rico que se ejecute como aplicación en la máquina cliente y un cliente web que pueda ser ejecutado desde un browser web.

Layer de lógica de negocios

En este layer se ejecutarán las reglas específicas del negocio. Recibirá peticiones desde el layer de presentación que deberá resolver aplicando las reglas que correspondan.

Llevar a cabo la lógica implicará comunicarse con la capa de acceso a datos con los siguientes propósitos:

- Recuperar los objetos involucrados en las reglas que haya que aplicar
- Persistir los cambios realizados.

Como regla general se utilizará el paradigma orientado a objetos para el diseño e implementación tanto de las reglas de negocio como de las precondiciones y postcondicones de estas reglas.

Puede aparecer el uso de otro paradigma en casos donde la performance sea crítica y el tiempo de respuesta esperado no se pueda lograr usando el paradigma orientado a objetos.

Este layer debe garantizar la ejecución de reglas controlando los permisos definidos como así también el registro de la ejecución para una posterior auditoría. Este log sirve además de base para encontrar causas de problemas en la ejecución de la aplicación.

Se utilizará programación orientada a aspectos como mecanismo para la implementación de aquellas características que son comunes a todos los casos de uso (por ejemplo seguridad y logging) y que no deberían mezclarse con las reglas de negocio. El uso de aspectos facilita, por un lado, el mantenimiento de la implementación de los aspectos y por otro lado el desacoplamiento entre las reglas de negocio y aspectos que no deben preocupar a quien diseña y desarrolla las reglas.

Este layer brindará acceso a las reglas de negocio no solo al layer de presentación sino también a otros sistemas autorizados que así lo requieran. Para esto, utilizará *web services*.

Layer de acceso a datos

Este layer es el responsable de

- Proveer acceso a los datos y objetos que necesita el layer de lógica de negocio para poder llevar a cabo su cometido

- Llevar a cabo la persistencia de los cambios realizados por el layer de la lógica de dominio.

Para llevar a cabo estas operaciones deberá comunicarse con BD del sistema, con sistemas legados y/o con otros sistemas que provean datos.

Siendo que el medio de persistencia de la aplicación será una BD relacional, deberá encargarse del mapeo objeto-relacional para poder convertir los objetos en filas de tablas y viceversa.

Layer de datos

Esta capa será la responsable de almacenar los datos de la aplicación.

El medio de persistencia principal de la aplicación será una base de datos relacional. Además existirán otras fuentes/destinos de datos, entre las cuales se encuentra el sistema legado.

ESTRUCTURA GLOBAL DETALLADA

A continuación explotaremos cada una de los 4 layers mencionados para describir su organización interna.

Layer de lógica de negocios

Dividimos el layer de lógica de negocios en tres partes:

- ✓ Modelo de dominio: Tiene la lógica de dominio de la aplicación. Usa el paradigma orientado a objetos para implementar las reglas del negocio.
- ✓ Layer de servicios: Fachada sobre el modelo de dominio. Coordina una operación interactuando con la capa de acceso a datos y delegando en el modelo de dominio. Define las precondiciones que deben darse para que el servicio se ejecute.
- ✓ Layer de servicios remotos: mantienen toda la lógica de aplicación. En este caso la lógica de aplicación incluye exposición de los servicios para poder ser ejecutados remotamente, manejo de transaccionalidad, seguridad, logging (detallaremos cada uno de ellos en la sección de servicios remotos)

Modelo de dominio

Como mencionamos anteriormente, el paradigma orientado a objetos guiará el modelado de la lógica del negocio. La *herencia*, el *polimorfismo* y el *binding dinámico* son tres mecanismos que provee este paradigma que, bien utilizadas, simplifican el modelado y el futuro mantenimiento.

Si bien el uso del paradigma orientado a objetos para expresar la lógica de dominio, es una práctica muy utilizada y conocida, Martin Fowler describe esta

práctica a través del patrón de arquitectura *DomainLayer*. Este será uno de los patrones que usaremos en este layer.

En casos donde el paradigma de objetos no puede brindar una solución que cumpla con ciertos atributos de calidad como pueden ser la performance o el consumo de memoria , se podrá utilizar el patrón *transactionScript*.

Layer de servicios locales

Con el objetivo de organizar la funcionalidad expuesta por el modelo de domino, se propone anteponer un layer de servicios (que funciona como *facade* [Gamma]). Cada servicio representa una operación que brinda el sistema (caso de uso por ejemplo) y su responsabilidad es coordinar esta operación: Recupera los objetos que tienen que participar en la operación y delega la operación en ellos que son los que realizan la lógica propiamente dicha.

Cabe recalcar que un servicio coordina la operación pero no tiene la lógica pues, si la tuviera, nuestra arquitectura estaría respondiendo a un *transactionScript* y no a un *DomainLayer*. Este facade funcional sobre el domainModel es conocido como el patrón *ServiceLayer*.

Hasta el momento tenemos, entonces, un layer de servicios y un layer de dominio donde viven los objetos que usan las características del paradigma orientado a objetos.

Precondiciones de los servicios

Cada servicio del sistema definirá precondiciones de negocio que se tienen que cumplir para poder ser ejecutado. Estas precondiciones se basan en los valores ingresados por el usuario y en el estado del modelo de dominio.

Se busca desacoplar las precondiciones de los servicios propiamente dicho para cumplir dos objetivos:

- Poder reusar las precondiciones
- Acotar el desarrollo del servicio a la coordinación de lógica

Se debe considerar también que una precondición se asocia a un servicio de distintas maneras:

- Informativa: Si la precondición no se cumple, el servicio se ejecuta igual y se informará un warning al usuario final.
- Restrictiva: Si la precondición no se cumple, no se podrá ejecutar el servicio.

Para la ejecución de estas precondiciones, existirá un *aspecto* de "reglas de validación" que se encargará, al momento de ejecutar un servicio, de buscar las precondiciones asociadas, ejecutarlas y dependiendo del resultado y de su configuración (restrictiva o informativa) actuar en consecuencia.

Es necesario aclarar que al desarrollar un servicio, se deberán indicar las precondiciones que se quieren asociar y el tipo de asociación que se quiere para que el aspecto de "reglas de validación" actúe correctamente. La implementación de la arquitectura deberá brindar herramientas para poder realizar esta asociación.

Considerar también que el repositorio de precondiciones podrá ser extendido por el equipo de desarrollo. La implementación de la arquitectura deberá, también, brindar herramientas para poder realizar esta implementación.

Hasta este punto tenemos la siguiente arquitectura:

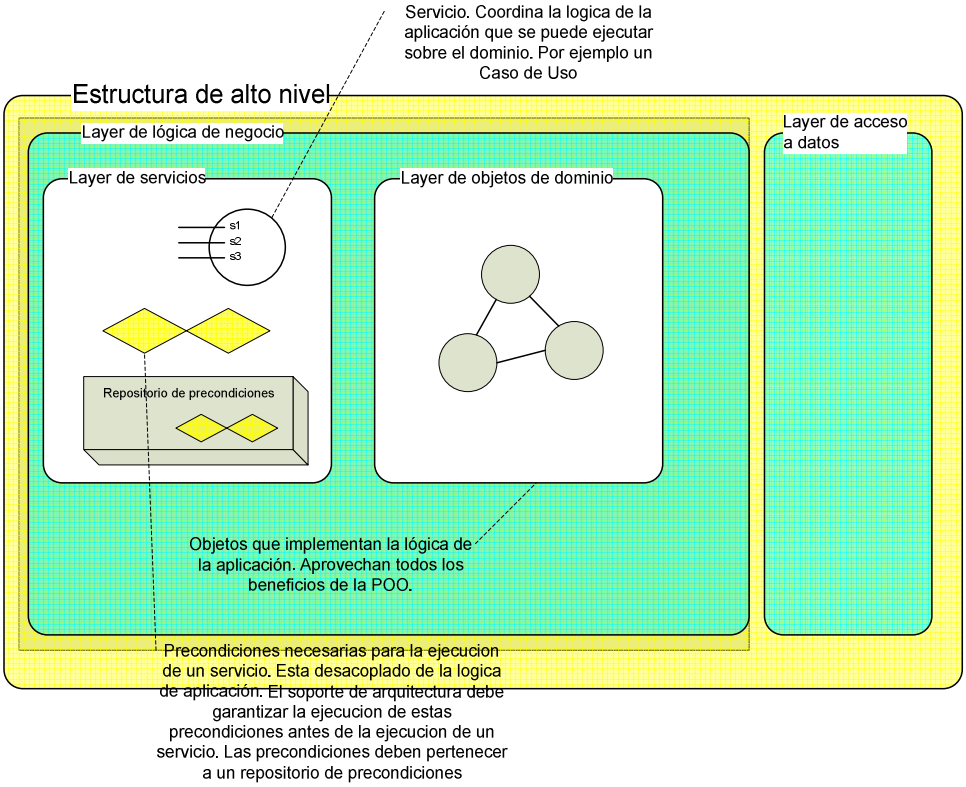


Ilustración 8 - Arquitectura eSidif con layer de negocio explotado

Capa de servicios remotos (Lógica de aplicación)

La lógica de aplicación abarca todos los aspectos que no están relacionadas con la lógica de dominio sino con la lógica de aplicación. En esta aplicación, estos aspectos son: transaccionalidad, ejecución remota, seguridad, logging, captura de excepciones de servidor escapadas.

Transaccionalidad: Aspecto que asegura las propiedades *ACID* para la ejecución del servicio. Este aspecto se encarga de abrir la transacción previo a ejecutar cualquier operación en la capa de acceso a datos y es el encargado de pedir a la capa de acceso a datos el commit o rollback una vez concluida la operación.

Ejecución remota: Aspecto que se encarga de exponer el servicio local a través de distintos protocolos y formatos. En este caso los servicios se exponen a

través del protocolo HTTP. Además se encarga de la serialización y deserialización de objetos.

Seguridad: Se encarga de la autenticación y autorización. La primera es para verificar que el usuario es quien dice ser. La segunda es para verificar que el usuario tiene permisos para poder ejecutar el servicio que está pidiendo ejecutar.

Logging: Este aspecto lleva registro de los servicios ejecutados con información adicional como fecha, hora, tiempo de respuesta, usuario, entre otros.

Captura de excepciones de servidor escapadas: Aspecto que asegura que ninguna excepción de servidor viaje al cliente. Es el último punto del servidor, antes de devolver la respuesta. Si le llega una excepción de servidor, la convierte a alguna que esté esperando el cliente.

Ciertos autores consideran que los servicios son responsables no solo de coordinar la lógica de dominio sino también de proveer la lógica de aplicación (transaccionalidad, ejecución remota, seguridad y logging). Esta solución tiene como problema que cuando un servicio es invocado localmente (por ejemplo un servicio invoca a otro), hay ciertos aspectos de la lógica de aplicación que posiblemente no sea necesario ejecutar:

- Seguridad: En el caso de estudio, alcanza con chequear el acceso al primer punto de entrada al servidor (servicio remoto).
- La transaccionalidad, si la invocación es local, tampoco es necesaria pues la transacción la abrió el primer servicio invocado remotamente (primer punto de entrada al servidor).
- Mecanismo de invocación remota, tampoco es necesario cuando la invocación es local, porque se puede hacer una invocación entre objetos dentro de la misma máquina virtual. Si la hiciéramos remota, estaríamos serializando y deserializando objetos sin necesidad.
- Captura de excepciones de servidor escapadas: No es necesario porque ya está el aspecto de captura de excepciones en servicio remoto. Será éste, el último punto en el servidor antes de devolver la respuesta y no el aspecto del servicio local.

Por esta razón, todos estos aspectos de la lógica de aplicación (transaccionalidad, seguridad, autorización de ejecución de servicios, ejecución remota) deben estar despegados de la lógica de dominio. Se propone entonces, una capa que se encargue exclusivamente de la lógica de aplicación. Llamaremos a esta capa **"Capa de servicios remota"**

Esta capa estará compuesta de:

- Servicios remotos: En general actúan como proxies de los servicios locales agregando solo la lógica de la aplicación. De todas maneras en esta capa pueden aparecer servicios que agrupen otros servicios locales para resolver algún problema de performance. Por ejemplo, se podría tener los servicios S1, S2 y S3 en la capa de servicios locales que no tengan relación entre sí. A nivel negocio, no corresponde un servicio que agrupe a estos 3 servicios. Supongamos que se tiene una GUI que muestra los resultados de los tres servicios S1,S2 y S3. En este caso, no tiene sentido definir en la capa de servicios locales un servicio S123 que invoque a los tres servicios y devuelva un gran objeto con los tres resultados. La GUI tendrá que hacer entonces 3 invocaciones al servidor, para conseguir los resultados de los 3 servicios. En este caso, para evitar hacer las 3 invocaciones al servidor, se podría agregar un servicio en la capa remota que invoque a los 3 servicios de manera que la GUI resuelva todo lo que necesita en un solo viaje, ejecutando el servicio de la capa remota. No es algo frecuente.

- Aspectos que implementan los distintos servicios de aplicación:

    - Aspecto de seguridad
    - Aspecto de logging
    - Aspecto de transaccionalidad
    - Aspecto que permite la invocación remota de los servicios.
    - Aspecto que maneja las excepciones, asegurando que no se enviarán al cliente excepciones que no corresponden.

    Estos aspectos se agregan a cada uno de los servicios remotos.

Incluyendo estos 3 layers, la arquitectura queda como muestra la siguiente figura:

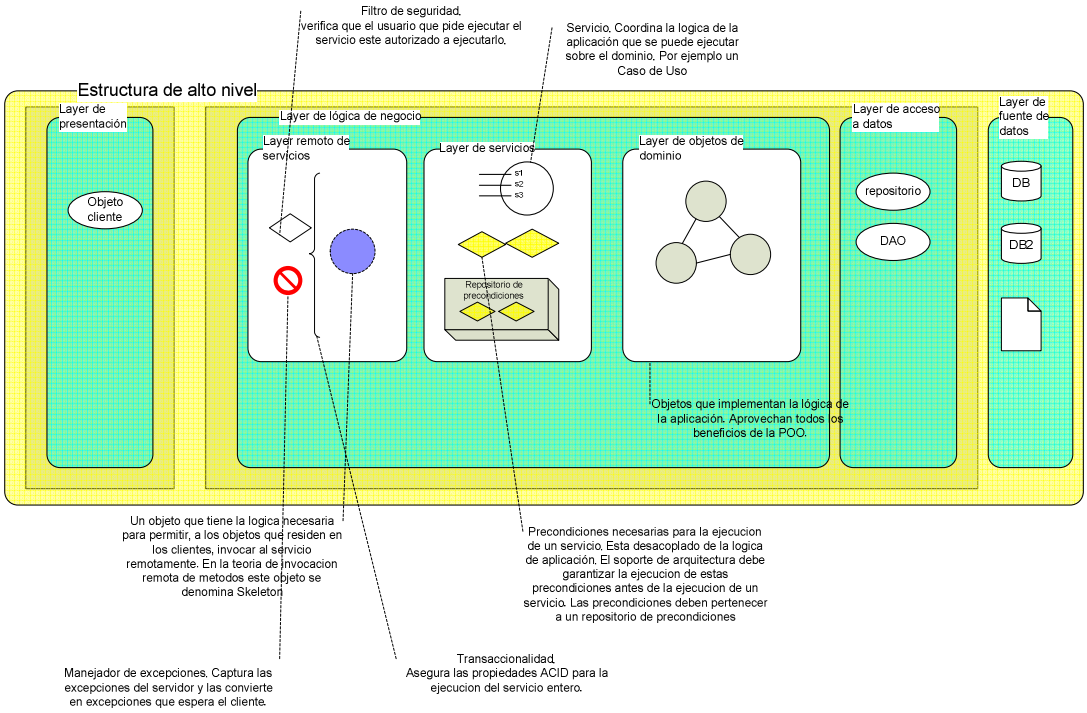


Ilustración 9 - Arquitectura eSidif con layer de negocio detallada

Layer de presentación

El layer de presentación es el encargado de presentar la información al usuario final y es el medio con el cual interactúa este usuario final.

Este layer debe interactuar con el layer de lógica de negocio para poder obtener la información que va a mostrar y para delegar muchas de las acciones que pide realizar el usuario final. Algunas de las acciones las puede resolver el mismo layer de presentación, pero muchas otras las delega en el layer de lógica de la aplicación.

Existen varios problemas que se presentan en el layer de presentación que describiremos a continuación, pero antes revisaremos la arquitectura interna que proponemos para este layer.

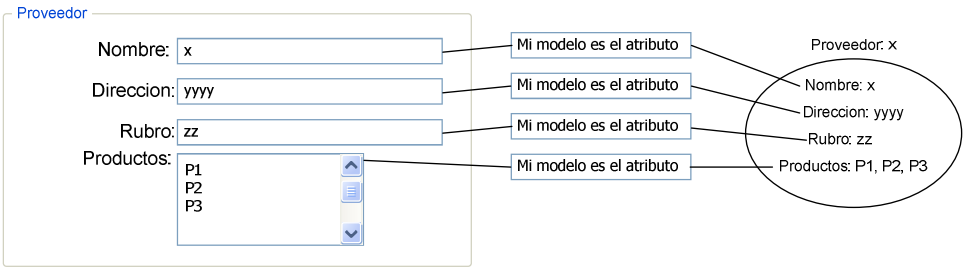
La arquitectura que proponemos utiliza el patrón MVC. La vista (V de MVC) en el layer de presentación está representada por la interfaz gráfica (compuesta por widgets). Esta vista utiliza al "modelo de cliente" (M de MVC) como fuente de la información a mostrar. Utiliza también el modelo para resolver validaciones y lógica de cliente y registrar los cambios realizados por el usuario.

El controlador (C del MVC) es el que permite mantener sincronizados el modelo y la vista evitando el alto acoplamiento entre ambos.

Miremos un poco más en detalle como se relacionan los widgets de la vista con los objetos del "modelo de cliente".

El caso más simple de interacción es aquel donde el modelo del cliente es un objeto determinado con una serie de atributos primitivos. Cada widget en la vista referencia a un atributo de ese modelo. El widget mostrará el valor del atributo y cambios en el widget deben modificar el atributo del objeto.

A continuación se muestra como sería la implementación de una componente gráfica reusable que permite visualizar y actualizar datos de un proveedor:



Como se muestra en el gráfico, cada widget tiene su propio modelo. Esta es la filosofía que proponemos para la arquitectura del layer de cliente, cuál? cada widget tiene su propio modelo. Otra vez? Sí, es importante remarcarlo.

En este ejemplo simple se muestra como un widget tiene como modelo un atributo del objeto "root". Llamamos objeto root al objeto principal de la GUI.

Este mismo esquema se puede extender a casos donde el modelo tiene mayor profundidad y los widgets tienen, como modelo, atributos de los objetos relacionados con el objeto root. Un ejemplo sería aquel donde la GUI necesita mostrar el nombre del proveedor pero la GUI tiene como modelo root una instancia de la clase Factura que es la que tiene el objeto proveedor.

Hasta el momento vimos dos ejemplos donde las vistas tienen widgets que apuntan a un atributo de tipo *primitivo* en el modelo (sea del objeto root o de algún objeto de su grafo).

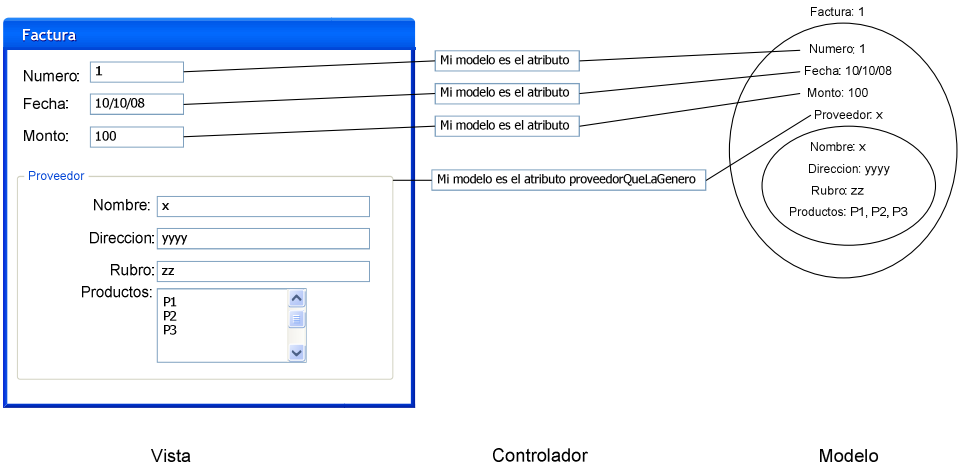
Nuestra propuesta es llevar esta misma idea (aquella en la que una view tiene un modelo y los widgets de la view apunten a atributos de ese modelo) para el desarrollo de componentes visuales que puedan ser reusadas. **Según nuestra experiencia este reuso de componentes gráficas mejora notablemente la productividad en el desarrollo de las GUIs, actividad que se lleva gran parte del desarrollo de los casos de uso.**

Cómo es esto de que una componente tenga su propio modelo?

Claro, cada componente gráfica reusable se desarrolla asumiendo que tiene como modelo un objeto de un tipo determinado, con tales atributos y tales relaciones. Los widgets internos de esa componente, se asociarán con atributos de ese modelo asumido.

Pero dónde está el reuso? Una vez desarrolladas estas componentes, el desarrollo de las GUIs será componer las componentes gráficas viendo a cada una de ellas como una caja negra. Solo será necesario indicar como conseguir el "modelo" que está esperando esa componente gráfica, que seguramente será algún objeto relacionado con el objeto root de la GUI.

Por ejemplo, la componente gráfica de proveedores presentada puede ser usada por la GUI de una "Factura", por la GUI de una "solicitud De Orden De Compra", dentro de un registro de proveedores deudores y en la GUI de operaciones de ABM de proveedores. En todos los casos se usará la componente y se indicará como conseguir el objeto proveedor sobre el cual trabajar. Continuando con el ejemplo, en el caso de la factura se indicará que el modelo de la componente gráfica del proveedor se consigue a través del atributo *proveedorQueLaGenero* del objeto factura como muestra la siguiente figura, que completa la GUI presentada anteriormente para la visualización y edición de una factura:



Además de mejorar la productividad, esta arquitectura del layer del cliente permite mantener el encapsulamiento de cada componente gráfica reusable. **Si se agregara un dato al proveedor, solo se modifica la componente gráfica proveedor y automáticamente se logra que las GUIs que la usaban se vean modificadas.**

Esta solución deberá ser complementada con una herramienta WYSIWYG (What You See Is What You Get) para poder construir las pantallas y las componentes gráficas aún más rápido. Estas herramientas deben permitir construir las GUIs haciendo Drag&Drop desde una paleta de componentes (de la tecnología que se use y las propias de la aplicación) hacia la GUI. Además, se deben proveer herramientas que permitan configurar cada uno de los widgets arrojados sobre la GUI. Cabe recalcar que estos widgets podrán ser tanto componentes de fábrica como propios.

Además de todas las configuraciones gráficas, se deberá poder setear el atributo del modelo de la componente que funciona como "modelo del widget".

Modelo del cliente

Para esta aplicación se decidió usar el patrón DTO (Data Transfer Object) que propone no enviar el modelo de dominio del servidor para que funcione como

modelo de cliente, sino que se mande la representación del modelo de dominio necesario para poder construir las GUIs. El uso de DTOs implica:

- Crear objetos DTO a partir de objetos de dominio

- Sincronizar el modelo del cliente - considerando las modificaciones realizadas por el usuario - con el modelo de dominio teniendo en cuenta los problemas de concurrencia que puedan surgir si otros usuarios tocaron el mismo objeto en otro cliente.

- Permitir traer relaciones de objetos bajo demanda. La decisión de cuándo una relación de un DTO viaja al cliente de manera lazy y cuándo no, debe quedar en manos del desarrollador del caso de uso. Esta decisión estará alineada con la interacción del usuario con el caso de uso. Si por ejemplo, el proveedor de una factura se ve en un diálogo que se abre luego de que el usuario presiona un botón sobre la factura, esa relación podría ser lazy.

- Debido a que las colecciones en las GUIs se muestran paginadas, el modelo de cliente debe soportar esta característica

Estas tareas son tediosas de repetir en todos los casos de uso, es por esto que la implementación de la arquitectura estructural deberá proveer un framework que simplifique la creación y la interacción con el modelo del cliente.

Cómo es la comunicación con el servidor?

Además de la actualización del modelo a partir de los widgets, el usuario final dispondrá de elementos visuales que pueden generar eventos a partir de la interacción del usuario. Un ejemplo de esto es perder el foco de un widget. Estos eventos disparan la ejecución de acciones. Estas acciones en ciertos casos ejecutan operaciones en el modelo del cliente, en otros ejecutan un servicio de los provistos por el layer de lógica de negocio. Estos servicios pueden devolver información que se utiliza para actualizar el "modelo de cliente" y como consecuencia de ello la vista.

Los servicios pueden devolver excepciones que también deben ser manejadas por este mecanismo de invocación de servicios al servidor. Se debe distinguir las excepciones de negocio que deben ser informadas al usuario – por ejemplo, una

precondición del servicio que falla - de las no esperadas, que se producen por un error de desarrollo o de conexión.

Tipos de clientes

El layer de presentación será implementado como una aplicación standalone. Se denomina así una aplicación que se instala en las máquinas cliente. Esta puede ser real o virtual. Será real cuando la máquina dispone de recursos o virtual cuando se utiliza un emulador gráfico.

La comunicación con el servidor se hará a través del protocolo HTTP.

Ciertos módulos especiales de la aplicación - solo disponibles para usuarios autorizados - se ejecutarán a través de una aplicación web – vía browser.

La tecnología y frameworks utilizados en la aplicación standalone y en la aplicación web serán diferentes. Ambos deberán respetar todo lo presentado sobre la arquitectura del layer de presentación.

La aplicación web será RIA (Rich Internet Application). Estos tipos de aplicación están alineados con lo expuesto hasta el momento pues, a diferencia de las aplicaciones web tradicionales, en el browser no "vive" solamente la parte visual sino que existe un modelo que le da vida a esa vista.

Layer de acceso a datos

En esta sección analizaremos la organización interna del layer de acceso a datos.
Como mencionamos anteriormente tendremos distintos tipos de fuente de datos. Si bien este layer debe brindar acceso a cada una de las fuentes de datos (Acceso a la base de datos principal para persistencia y consulta de datos, acceso a sistemas legados, accesos a sistemas externos) será objeto de estudio de esta tesis solo el acceso a la base de datos principal.
Solo haremos una mención especial para el acceso a sistemas legados. Es importante no desestimar el tiempo que puede llevar la sincronización con el sistema legado.
Con respecto a la comunicación con otros sistemas se deben usar estándares.

Acceso a la base de datos principal

Cuando usamos paradigma orientado a objetos para modelar e implementar la lógica de dominio, tal como propusimos y detallamos en la sección de "layer de dominio", lo que tenemos que persistir son los objetos que vamos creando y modificando.
Lo que necesitamos persistir es el estado del objeto. La manera más natural de pensar en la persistencia de objetos es disponer de una base de datos que almacene y permita recuperar objetos, mejor conocidas como base de datos orientada a objetos.
Existen varios productos en el mercado, como JPOX, que proveen una API para persistir y recuperar objetos y realizar consultas sobre los mismos. Además suelen proveer herramientas graficas para visualizar el estado de la base de datos orientada a objetos.

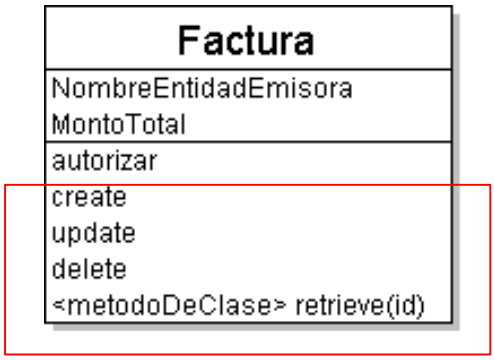
Según nuestra opinión no existe una base de datos orientada a objetos que se adapte a las necesidades y atributos de calidad que exigen las aplicaciones Enterprise. Los productos que existen no tienen la madurez necesaria para manejar el volumen de datos que maneja una aplicación Enterprise ni los tiempos de respuesta que este tipo de aplicaciones necesita.
Cabe aclarar asimismo que la entidad cliente del caso de estudio, utiliza una base relacional para sus aplicaciones actuales - Oracle. Estas aplicaciones convivirán con el nuevo sistema y el desarrollo de esta convivencia se verá facilitado utilizando el mismo manejador de BD en ambos casos.
Es por lo expuesto que se decidió utilizar una BD relacional. Como consecuencia de esta decisión es necesario que la capa de acceso a datos pueda trabajar con una base de datos relacional como medio de persistencia.
El problema que aparece entonces es que estamos tratando con dos mundos distintos. **En el layer de dominio tratamos con el mundo de los objetos. En el layer de datos tratamos con el mundo relacional**. En aquél hablamos de objetos, atributos y relaciones; mientras que en éste tratamos con tablas, columnas y claves foráneas. Aquí es donde aparece la función del layer de acceso a datos para interactuar con una base de datos relacional como medio de persistencia principal.
Este layer se ocupará de dar solución al problema conocido como "mapeo objeto-relacional". Este problema incluye:

- ✓ Cómo representar en la bd relacional los objetos con sus atributos usando tablas y columnas. El mapeo más simple que se puede pensar es cada clase a una tabla y cada atributo de la clase a una columna de esa tabla.

- ✓ Cómo representar las relaciones entre los objetos usando claves foráneas.
    - Relaciones uno a uno se mapean usando una clave foránea en la tabla del objeto padre
    - Relaciones uno a muchos se logran usando una clave foránea en la tabla del objeto hijo.
- ✓ Como mapear las operaciones CRUD (create, retrieve, update y delete) en las correspondientes sentencias SQL que se deben ejecutar en la base de datos relacional.
    - Create: Se debe ejecutar el insert en la tabla correspondiente. Debido a que un objeto está compuesto por un grafo de objetos, para cada uno de los objetos relacionados se deberá hacer el INSERT o UPDATE correspondiente y representar las relaciones usando las claves foráneas que correspondan
    - Retrieve: Se debe realizar el select correspondiente que devuelva una sola fila. En general se utiliza un object_id para llevar a cabo esta operación. Con los datos recuperados, se crea la instancia correspondiente. Debe soportar *lazy loading* para evitar levantar el grafo completo de objetos y que los objetos de adentro del grafo se vayan cargando a medida que se necesiten.
    - Update: Se deben aplicar los cambios que se produjeron sobre los atributos y objetos relacionados a través de sentencias UPDATE e INSERT.
    - Delete: Se debe poder borrar las tuplas correspondientes de la base de datos para que ese objeto deje de ser parte del sistema.

Existen varios patrones para resolver este problema. Algunos patrones no son una buena alternativa si trabajamos con un *domainModel* y por eso quedaron afuera del alcance de este análisis.

Una de las primeras alternativas, etiquetada bajo el patrón *ActiveRecord,* es que las clases de dominio persistentes además de proveer mensajes con el comportamiento propio de la clase, implementen las operaciones CRUD (create, retrieve, udpate, dalete)



Como muestra la figura, el método autorizar es propio del comportamiento del objeto, mientras que los mensajes dentro del rectángulo rojo son necesarios para hacer el mapeo objeto-relacional.

- ✓ create: En el modelo de dominio se crea una instancia, se trabaja con la misma y se pide a la clase que la cree a través de este mensaje create.

- ✓ retrieve: Es un mensaje que se pide a la clase. Instancia la clase, ejecuta el select y usa el resultado para "completar" la instancia creada.

- ✓ Update: Luego de modificados los datos de un objeto, se ejecuta su update que traduce los cambios realizados en el objeto al SQL UPDATE correspondiente.

- ✓ Delete: A la instancia cargada se le pedirá delete. Esta instancia será la encargada de ejecutar la sentencia SQL DELETE utilizando su object_id.

Este patrón funciona bien cuando tenemos un modelo de dominio muy simple y la función que nos lleva de las clases de dominio a las tablas es prácticamente una clase a una tabla. **Descartamos entonces esta solución, por la complejidad que presenta nuestro caso de estudio.**
Cuando el modelo de dominio es más complejo (como en el caso de estudio), aparecen varias relaciones entre objetos de dominio, colecciones de objetos para modelar relaciones de una clase a muchas instancias de otra, herencia, composición. Estas relaciones y mecanismos complican el mapeo objeto-relacional y por ende las operaciones CRUD resultan más difíciles de implementar. Algunos ejemplos:

- ✓ Relaciones entre objetos

    - Al dar de alta un objeto, no solo debe hacer insert del objeto root sino también de los objetos relacionados. Se deberá verificar si los objetos relacionados están o no en la BD para determinar si corresponde un UPDATE o un INSERT. Las relaciones se deben traducir a claves foráneas.

    - Cuanto más profundos son los objetos, más se complican las consultas SQL que hay que ejecutar para conseguir los datos necesarios que permitan reconstruir los objetos. Además, existen muchas maneras de ejecutar estas consultas. Algunas serán mejores en cierto caso pero otras serán la solución óptima en otros. Será cuidadosa decisión del desarrollador de la operación cómo implementar las consultas considerando la profundidad, la frecuencia de consulta, entre otras cosas.

    - Cuando una clase está relacionada con muchas instancias de otra clase, son más complicadas aún las consultas. Además, como las colecciones son demasiado grandes, se debe proveer paginado para evitar el uso excesivo de memoria.

- ✓ Herencia

    Hay distintas estrategias para representar una herencia de objetos en tablas. Cualquier estrategia hará más complejas las operaciones CRUD. Las estrategias son:

    - Una tabla por clase concreta: En esta tabla aparecen los campos que representan los atributos de la clase concreta y los campos que representen los atributos heredados.

    - Una tabla por clase: Cada clase (sin importar si es abstracta o concreta) tiene su tabla asociada. La desventaja de esto es que siempre voy a tener que trabajar con varias tablas cuando quiera tratar con un objeto, aún sin tener relaciones.

    - Una tabla por jerarquía.

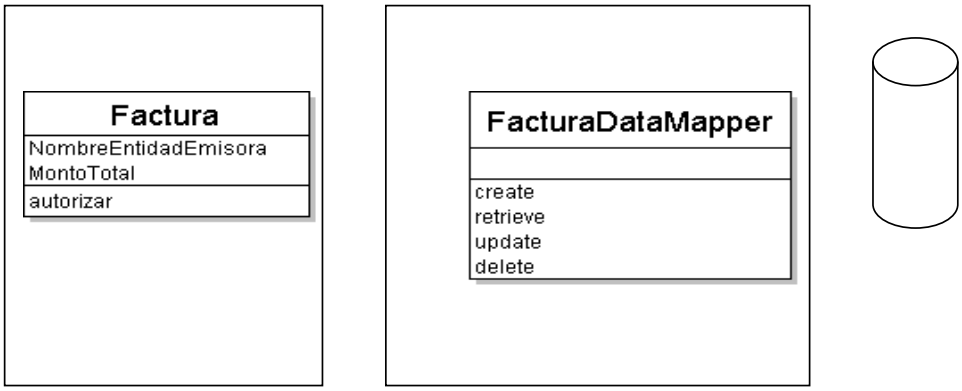
- ✓ Lazy loading

  Cuando se levanta un objeto de dominio del medio de persistencia, no siempre todos los objetos de su grafo serán utilizados en ese momento. Sería contraproducente -considerando la memoria ocupada y el tiempo de respuesta - levantar todos los objetos del grafo si no se van a utilizar. Incluso, en sistemas grandes con muchas relaciones entre los objetos, podría resultar imposible cargar todo el grafo de un objeto en la memoria del servidor.
  Es necesario entonces, proveer la posibilidad de carga de objetos relacionados bajo demanda.
  El mecanismo de mapeo debe permitir al desarrollador indicar qué atributos serán recuperados bajo demanda y qué atributos no, para determinar como hacer la consulta y como generar el grafo de objetos.

- ✓ Concurrencia

  Dos usuarios pueden estar trabajando con objetos iguales pero no idénticos. Recordemos que la igualdad de dos objetos es una igualdad definida a nivel funcional. En general, si se trabaja con object_ids, entonces la igualdad de estos objetos estará dada por al igualdad de este atributo. Por otro lado dos objetos son idénticos si son la misma instancia.
  Cuando dos usuarios piden el mismo objeto a través del *retrieve(object_id)* están generando dos instancias distintas pero iguales por tener el mismo object_id.
  Será necesario proveer esquemas de lockeo optimista - aquél donde todos llevan pero graba el primero - y pesimista - aquél donde lleva solo el primero - para garantizar la consistencia de los objetos en la BD.

Considerando la complejidad que presentan los mensajes CRUD en los objetos de dominio, aparece un nuevo patrón para implementar el mapeo objeto relacional. Este patrón tiene como objetivo desacoplar del objeto de dominio estos complicados mensajes. Para cada clase persistente existirá una clase DataMapper que implementa los mensajes CRUD de esa clase. Esto permite "despegar" los objetos de dominio de la BD.



Para implementar las operaciones CRUD de los dataMapper se deben tener las mismas consideraciones que las presentadas para activeRecord.

Como el estado de los objetos de dominio está en las instancias de las clases de dominio, el DataMapper es *stateless*. Es por esto que los DataMapper pueden ser implementados como Singletons o usando mensajes de la clase DataMapper.

Cada DataMapper debe disponer de la información de mapeo:

- Clases a tablas
- Atributos a columnas
- Jerarquías a tablas
- Relaciones a uno
- Relaciones a muchos

Considerando que el comportamiento de los dataMappers es el mismo y que lo que varía es la información de mapeo, se puede desacoplar la información de mapeo en archivos de metedata y concentrar la lógica de mapeo objeto relacional y la solución a todos los problemas antes mencionados en un solo lugar.
Llamaremos a partir de ahora Repositorio, al elemento de la arquitectura que permite realizar las operaciones CRUD de las entidades del sistema proveyendo soluciones a los problemas de relaciones entre objetos, lazy loading, herencia, concurrencia.
Este repositorio se nutrirá de la información de metadata para poder realizar las operaciones CRUD.
En la sección de componente detallaremos la implementación de este elemento.

De esta manera, cuando un objeto necesite hacer una operación CRUD sobre un objeto de dominio, deberá pedirlo al Repositorio.

El caso más general será aquel donde:

- ✓ Un servicio de la capa local recupera, a través del repositorio, los objetos "root" que necesita para llevar a cabo la operación.

- ✓ El servicio "entra al modelo de dominio" para que opere

- ✓ El modelo de dominio ejecuta la lógica de dominio. Gracias al lazy loading que provee el repositorio, será transparente la carga de los objetos relacionados.

- ✓ Se interactúa con el Repositorio para dar de alta objetos y borrar objetos.

- ✓ El repositorio registra todas las modificaciones sobre los objetos de dominio involucrados, en una unidad de trabajo.

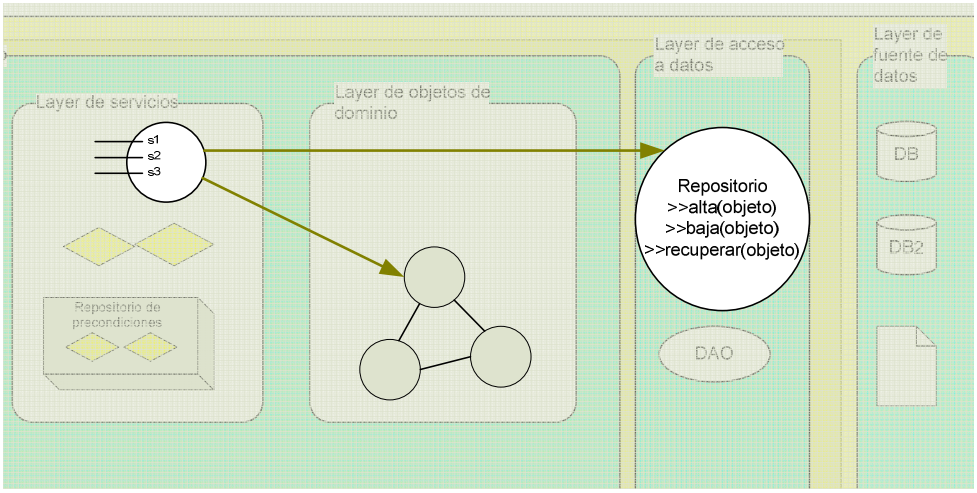
- ✓ Cuando termina el servicio, el repositorio tienen la unidad de trabajo completa con las altas y bajas que le pidieron explícitamente y las modificaciones que fue registrando. Es en este momento que convierte esta unidad de trabajo en los INSERT, UPDATE y DELETE correspondientes.

Es importante mencionar que para poder realizar todas estas operaciones con el Repositorio, es necesario estar dentro de un contexto transaccional.
Es el aspecto transaccional agregado al servicio remoto, que funciona como punto de entrada al servidor, el que asegura que todo el servicio se ejecuta dentro de ese contexto transaccional. Para éste, el aspecto interactúa con la capa de acceso a datos para crear una transacción previo a comenzar la ejecución del servicio y para hacer commit o rollback luego de ejecutado el servicio.

En general quienes interactúan con el Repositorio son los servicios de la capa local pero, en ciertos casos, podrá ser accedido por objetos de dominio.

Hasta el momento, la API que provee el Repositorio incluye
>>retrieve(objectID)
>>alta(objeto)
>>baja(objeto)

Como comentamos anteriormente la modificación no es parte de la API porque el mismo repositorio es quien se encarga de registrar cada uno de los cambios sobre los objetos involucrados.



Performance en mapeos objeto-relacional:

Si bien el Repositorio nos abstrae de la base de datos relacional subyacente, la performance no nos deja olvidar completamente de esta base de datos relacional. El volumen de datos de las aplicaciones Enterprise, nos exige manejar con cuidado las consultas que se generan a partir de las operaciones CRUD que realizamos contra el Repositorio.

Todo lo expresado en los archivos de mapeo será utilizado por el Repositorio para poder generar las sentencias SQL correspondientes. El lenguaje de mapeo deberá ser entonces lo suficientemente flexible ofreciendo facilidades para optimizar las consultas SQL generadas.

Performance en la recuperación de objetos:

Hasta el momento, la carga de objetos se consigue de dos maneras:

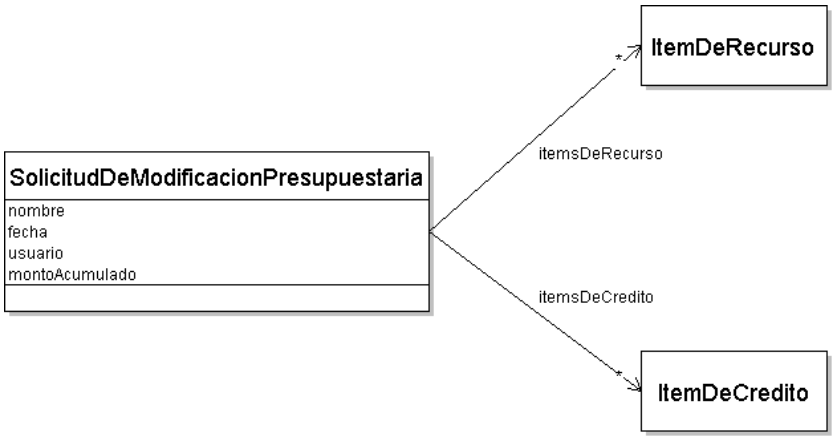
- Retrieve(oid): Recupera un objeto a partir de su id
- Navegar un objeto ya cargado a través de sus relaciones

Ya mencionamos la necesidad de trabajar con relaciones lazy. Esta flexibilidad permite al desarrollador mejorar sus tiempos de respuesta diferenciando las relaciones que necesita levantar junto al objeto root y las que dejará para levantar bajo demanda.

Si bien esta decisión del desarrollador impacta directamente en la consulta generada, la metadata debe proveer configuración directamente relacionada con la consulta como puede ser:

- La relación del objeto root con un objeto que se quiere levantar no lazy, se resuelve como dos SELECT o como un JOIN?

Con los dos mecanismos de recuperación de objetos mencionados (retrieve y navegación a través de sus relaciones), nos basta para llegar a cualquier objeto que necesitemos. Pero aquí aparece nuevamente el problema de la performance.
Si bien lo más natural y simple para resolver consultas sería disponer de todo el modelo de objetos en memoria y escribir las consultas usando las buenas prácticas y mecanismos del paradigma orientado a objetos, existen casos donde ésta no es una buena solución porque traerá problemas de performance.
Por ejemplo, filtrar en memoria los objetos que cumplen una condición, siendo que solo 40 de 100000 son los que la cumplen, no parece ser una buena solución a nivel performance comparada con realizar una consulta que solamente levante de la BD los 40 objetos que cumplen la condición.
Tomemos un ejemplo del caso de estudio.



Un dato importante del problema es que la SolicitudDeModificacionesPrespuestaria puede tener hasta 300.000 elementos.

Para conseguir todos los ítems de recursos cuyo monto sea mayor a 5000, la manera más expresiva de hacerlo sería utilizar objetos y mensajes como muestra el siguiente pseudocódigo:

```
SolicitudDeModificacionPresupuestaria
>>ColeccionDeItems dameLaColeccionDeItemsConMontoMayorA(int limiteInferior)

    solicitud=Repositorio.retrieve("SolicitudDeModificacionPresupuestaria", 1);

    itemsAltos = new Coleccion();

    for  each item in solicitud.getItemsCredito()
        item.agregateSiSosAlto(itemsAltos, limiteInferior);

    return itemsAltos;
```

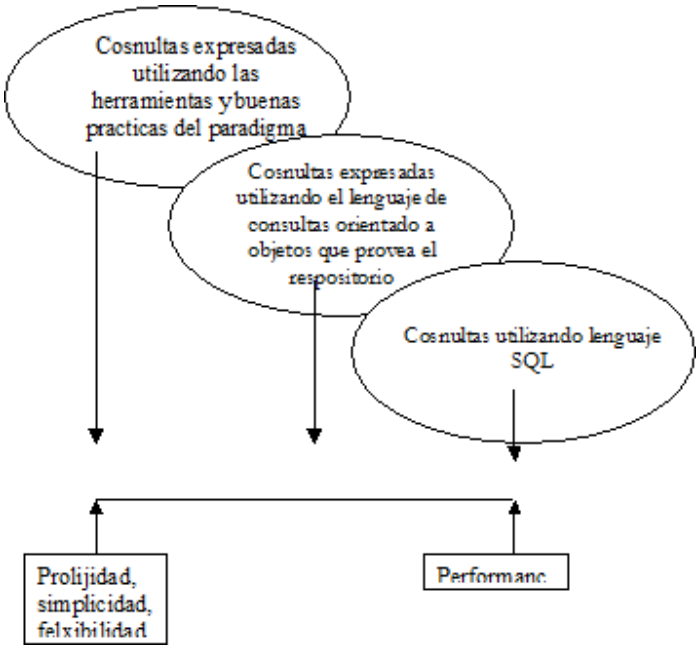
```
ItemDeCredito
>>agregateSiSosAlto(Colección itemsAltos, int limiteInferior)

     if this.getMonto() > limiteInferior
       itemsAltos.add(self)
```

Si bien esta solución es bien expresiva, no es buena a nivel performance pues implica instanciar 300.000 instancias de ItemDeRecurso para luego quedarnos solamente con los pocos que cumplen con la condición.

Por lo expuesto en los ejemplos, la solución planteada debe ser complementada con un lenguaje de consultas orientado a objetos. Este último, si bien no alcanza la misma expresividad que el trabajo puro sobre objetos y mensajes, permite escribir consultas referidas a los objetos del modelo de dominio y no a las tablas de persistencia.
Sin este lenguaje, el desarrollador sería el responsable de escribir las sentencias SQL para especificar las consultas, procesar los resultSet, instanciar los objetos y poblarlos con los datos del resultSet.

Cabe utilizar nuevamente la balanza de performance vs. expresividad.



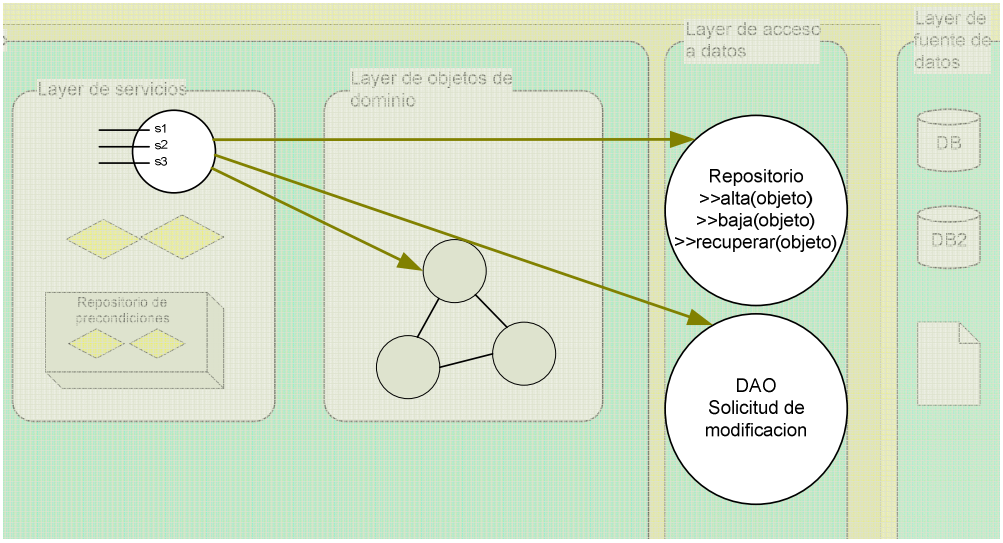
Como muestra la figura, la simplicidad, la expresividad y prolijidad del extremo izquierdo se consiguen resolviendo las consultas usando el paradigma orientado a objetos. En el otro extremo, el de la performance, aparecen las consultas SQL.
Como solución intermedia aparece el mencionado lenguaje de consultas orientado a objetos.
Depende del caso de uso que se esté desarrollando, el desarrollador deberá ubicarse donde corresponda.
El repositorio proveerá un mensaje para poder ejecutar consultas con un lenguaje orientado a objetos.

Performance en la actualización de objetos:

El lenguaje de consultas presentado en la sección de performance en la recuperación de objetos, debe proveer también sentencias para hacer actualizaciones masivas. Si bien se puede lograr de manera muy expresiva una actualización masiva usando el paradigma orientado a objetos, por cuestiones de performance en ciertos casos es necesario no levantar todos los objetos a memoria.

Por una cuestión de prolijidad y mantenimiento, proponemos que todas las consultas queden concentradas en la capa de acceso a datos y no estén diseminadas por toda la arquitectura. Para esto se propone utilizar el patrón DAO – Data Access Object-. Serán entonces los DAOs quienes utilicen el mensaje ejecutar(consulta) del Repositorio.

De esta manera la capa de acceso a datos queda compuesta por el Repositorio y los DAOs. Los servicios locales interactuarán con el Repositorio para pedir altas, bajas y recuperaciones a través de su oid, y con los DAOs para ejecutar otro tipo de consultas.



Implementación de la arquitectura E-Sidif

Tecnologías y frameworks de la comunidad

Se presentan las tecnologías que utiliza el e-Sidif de la misma manera que se presentaron las distintas secciones de la estructura global del sistema.

RICH CLIENT PLATFORM (RCP)

RCP es una plataforma de desarrollo de aplicaciones desktop que permite utilizar un grupo de artefactos que facilitan el desarrollo de software.

RCP está compuesto por las siguientes partes:

- Un CORE que contiene la base de la plataforma
- Un Framework estándar de desarrollo
- Un toolkit de widgets portables
- Editores de texto y manejadores de archivos y texto
- Un ambiente de ejecución (Workbench) que permite trabajar con editores, vistas, perspectivas y wizards
- Una herramienta de actualización automática

La construcción de los Casos de Uso sustentada en esta plataforma de funcionalidad base permite un desarrollo mas rápido de la aplicación y una mejor integración. Permite ganar productividad en el proyecto.

Otro beneficio asociado a esta plataforma es que los desarrollos realizados sobre ella son portables a distintos sistemas operativos.

STANDARD WIDGET TOOLKIT (SWT)

SWT es una librería de elementos visuales que se utiliza sobre la plataforma Java. Fue desarrollado por IBM y actualmente es actualizado y mantenido por el grupo de fundación Eclipse. SWT es una alternativa dentro de las librerías visuales existentes tales como AWT y Swing provistas por Sun Microsystems y que son parte de la plataforma estándar de Java (JSE).

SWT se utiliza para la visualización de elementos de interfaz gráfica. Para esto, se accede a las librerías nativas del sistema operativo utilizando JNI (Interfaz Nativa

de Java). Las aplicaciones que utilizan SWT son portables, sin embargo, si bien son herramientas desarrolladas en Java, su implementación es específica para cada plataforma. Este set de herramientas posee licencia pública de Eclipse y abierta.

SWT trabaja como un *wrapper* de los objetos nativos. Es por esta razón que usualmente a sus widgets se los considera como "pesados". En los casos en donde la plataforma nativa no soporta la funcionalidad que pretende proveer SWT, éste implementa su propio comportamiento tal como lo hace Swing. En esencia, podría decirse que SWT es una implementación que se compromete a tener la performance de los componentes nativos y respetar el *look and feel* de la plataforma, como así también simplificar la utilización de Swing. SWT posee además un diseño limpio inspirado en los patrones de diseños de Erich Gamma[10]

Los atributos de SWT lo destacan de otras librerías al momento de seleccionarlo como tecnología de visualización ya que en el caso de estudio el cliente utilizado para el sistema e-Sidif podría llegar a correr en distintas maquinas con distintos sistemas operativos.

Ventajas

- Diseño limpio
- Performante por estar cerca de las librerías nativas
- Portable
- *Look and Feel* similar a la plataforma nativa
- Estable - No requiere muchas actualizaciones ya que las plataformas no suelen cambiar su API
- Permite extensiones

Desventajas

- Su implementación base no utiliza MVC

[10] Gamma/Helm/Johnson/Vlissides, **DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE**

- Requiere liberación manual de la memoria (no utiliza el garbage collector) – posibles memory Leaks

TECNOLOGÍAS DEL LAYER DE ACCESO A DATOS

HIBERNATE

Hibernate es una herramienta de Mapeo objeto-relacional para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) que permiten establecer estas relaciones.
Hibernate es software libre, distribuido bajo los términos de la licencia GNU LGPL.

La utilización de Hibernate soluciona el problema de la diferencia entre los dos modelos usados para organizar y manipular datos: El mundo de los objetos que residen en memoria (orientación a objetos) y el mundo de datos y relaciones de las bases de datos (modelo relacional). Para lograr esto el desarrollador puede detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen mediante una configuración declarativa. Con esta información Hibernate le permite a la aplicación manipular los datos de la base operando sobre objetos, con todas las características de la *POO*. Hibernate convertirá los datos entre los tipos utilizados por Java y los definidos por SQL, genera las sentencias SQL y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias.

Hibernate ofrece también un lenguaje portable de consulta de datos llamado HQL (Hibernate Query Language), al mismo tiempo que una API para construir las consultas orientadas a objeto (conocida como "criteria"), además de permitir la ejecución de consultas en SQL nativo.

Hibernate es seleccionada como la herramienta de mapeo objeto-relacional por el caso de estudio en primera instancia por la restricción antes mencionada de convivencia entre el sistema legado y el nuevo sistema y además por brindar una persistencia transparente. Algunos de los atributos que permiten la elección de esta tecnología por sobre otras son los siguientes:

- Modelo natural de programación – Hibernate se adapta de manera natural a la programación orientada a objetos, herencia polimorfismo y los frameworks de colecciones de Java

- Permite el mapeo de objetos de alta granularidad con gran variedad de colecciones y objetos dependientes

- No requiere de una compilación agregada - No necesita una generación de código y de una compilación extra además de la necesaria para el proceso de construcción del código

- Escalabilidad extrema – Hibernate es performante ya que permite una doble capa de caché con la capacidad adicional de funcionar en un cluster

- Consultas – Permite obtener los objetos de la base de datos mediante consultas, además de acelerar el acceso a éstos cuando son solicitados nuevamente

- Distintos tipos de "conversaciones" con la base datos – Hibernate permite la persistencia en contextos de persistencia de larga vida como así también los procesos de *detach/reatach* a los objetos en la sesión. Permite tener control de concurrencia a través de lockeo optimista automáticamente.

- Implementa la Java Persistence Management API y los distintos mapeos de objetos a relacional.

TECNOLOGÍAS DEL LAYER DE LÓGICA DE DOMINIO

Modelo de Negocio

SPRING

El Spring es una estructura de soporte para desarrollar aplicaciones sobre la plataforma Java.

Fue escrito inicialmente por Rod Johnson en el a˜no 2000. Para que las aplicaciones que trabajan con distintas partes de la plataforma Java pueden ser mas simples y consistentes para los desarrolladores.

El Framework Spring adopta un conjunto de características que mencionamos a continuación:

- Spring pone foco en brindar facilidades para el manejo de los objetos de negocio.

- Es modular. Como Spring está estructurado en layers, los que pueden utilizarse por separado sin perder consistencia. Por ejemplo, se puede utilizar el Framework para simplificar los accesos de JDBC solamente o bien se puede utilizar para manejar todos los objetos de negocio.

- Provee facilidades mediante IOC para construir código fácil de testear.

- Posee una muy buena integración con distintas tecnologías.

Beneficios Arquitecturales

- Permite una organización de los objetos del *middle tier*, y de la interconexión de distintos frameworks

- El Core Container o Contenedor de Inversión de Control (Inversion of Control, IoC) es el núcleo del sistema. Responsable de la creación y configuración de los objetos.

- Aspect-Oriented Programming Framework, que trabaja con soluciones que son utilizadas en numerosos lugares de una aplicación, lo que se conoce como asuntos transversales (cross-cutting concerns).

- Se elimina la proliferación de Singletons. Esto facilita la testeabilidad y orienta hacia una forma de trabajo a objetos más pura.

- Permite reducir la utilización de un gran número de formatos de archivos de propiedades, trabajando de una manera consistente estas configuraciones.

- Permite realizar buenas prácticas de programación reduciendo la cantidad de interfaces

- No genera dependencias entre los objetos de negocios y las APIs de Spring.

- Facilita la construcción de tests de unidad

- Provee las facilidades de EJB sin requerir su uso. Por ejemplo, para el manejo de la transacción de manera declarativa usa AOP sin recurrir a la utilización de un container. Tampoco requiere una implementación de JTA, si solo se necesita trabajar con una base de datos.

- Provee un Framework de acceso a datos ya sea utilizando JDBC y/o productos de mapeo objeto relacional como Toplink, Hibernate o implementaciones de JDO.

Esencialmente Spring es una tecnología que permite construir aplicaciones utilizando POJOs que ayuda, en el caso de estudio, tanto al modelado estructural como al de comportamiento de las complejas reglas de negocio a las que se enfrenta en el caso de estudio.

Modelo de Dominio

El modelo de dominio se representa mediante clases planas y simples que no dependen de ningún Framework en especial, las cuales posteriormente mediante una herramienta de mapeo objetos/relacional son persistidos en una base de datos. Contienen además la lógica de negocio de la aplicación.

TECNOLOGÍAS DEL LAYER DE DATOS

ORACLE

Oracle es un sistema de gestión de base de datos relacional (o RDBMS por el acrónimo en inglés de Relational Data Base Management System), fabricado por Oracle Corporation.

Se considera a Oracle como uno de los sistemas de bases de datos más completos, destacando su:

- Soporte de transacciones.

- Estabilidad.
- Escalabilidad.
- Es multiplataforma.

Frameworks propios

Como presentamos en los capítulos precedentes, consideramos que el trabajo de un equipo de arquitectura no solo alcanza a la definición de un marco teórico de la arquitectura sino también a la implementación de la misma como medio para asegurar que se cumpla el marco teórico definido y para mejorar la productividad del equipo.

Esa implementación usará componentes y frameworks existentes en la comunidad que se alineen con las pautas definidas en el marco teórico de la aplicación. Existen otros elementos de la arquitectura para los cuales la comunidad no provee un framework adecuado; en estos casos es necesario desarrollar un framework propio. Esto ocurre cuando:

- En la comunidad no existe un framework para cubrir esa parte de la arquitectura definida
- El framework no tiene las características suficientes para cubrir las necesidades de la arquitectura
- La licencia no es gratuita y no puede cubrirse con los fondos disponibles
- El elemento de la arquitectura tiene una componente funcional importante que hace que una solución propia sea la mejor alterativa.
- Es necesario wrappear el framework existente para agregar funcionalidad propia.

A continuación se presenta un resumen de las componentes propias que se proponen, complementando los frameworks y plataformas ya mencionadas (Java, Spring, Hibernate, RCP, SWT)

REPOSITORIO

Como se presentó en la sección de "Layer de acceso a datos", existe un elemento de la arquitectura que denominamos repositorio y será quien nos brinde el acceso a los datos.
Si bien a nivel marco teórico de la arquitectura hablamos a nivel abstracto de un elemento "Repositorio", a nivel de implementación de la arquitectura debemos decidir como será la implementación de este elemento que brindaremos a los desarrolladores de la aplicación. Este elemento deberá cumplir con todos los requisitos descriptos en la mencionada sección.
Recordemos que la API contenía los siguientes métodos:

>>retrieve(objectID)
>>select(OOQuery)
>>alta(objeto)
>>baja(objeto)

Este elemento debe hacer los mapeos del mundo de objetos al mundo de tablas proveyendo facilidades como lazy loading, paginado de colecciones, manejo de concurrencia, lenguaje de consultas que evite potenciales problemas de performance.

La primera alternativa sería hacer una implementación del repositorio desde cero. Esta no sería una buena decisión porque estaríamos "reinventando la rueda". Existe muchos frameworks que ya proveen la funcionalidad implementada. Recordamos que Hibernate fue la alternativa elegida para este caso de estudio.

Teniendo la decisión de usar Hibernate y viajando al otro extremo de "reinventar la rueda", la solución más fácil que se podría proponer desde el equipo de arquitectura sería definir la vaga pauta: "se usa Hibernate como mapeador objeto relacional". El hecho de definir el elemento repositorio con esta vaga pauta, y no proveer pautas de uso y un frameowrk que wrapee a Hibernate, no era una buena decisión para el caso de estudio porque:

- La implementación del producto quedaría muy atada a la tecnología
- Luego de un análisis exhaustivo, se encontró cierta funcionalidad común que sería bueno proveer desde el repositorio que no provee Hibernate.
- Al no definir pautas de uso de un framework como Hibernate, el mantenimiento será muy complicado y probablemente existirán muchos malos usos de la herramienta.

Es por esto que se decidió hacer un framework **Repositorio.**

Gran parte de la API provista se delega en Hibernate, por ejemplo el save y el retireve. Se agrega funcionalidad extra para recuperar objetos que chequeen conurrencia y la principal componente agregada por este framework es la clase Search, que representa una búsqueda.
Si bien el corazón de esta clase Search, es la clase *Criteria* de Hibernate que permite especificar una consulta utilizando "objetos", se agrega cierta funcionalidad, entre las cuales podemos mencionar:

- ✓ Poder configurar a una condición para ser ejecutada en memoria o en base de datos. De esta manera, cuando se pide a repositorio ejecutar un Search se colectarán las condiciones de base de datos a partir de las cuales se armará el criteria correspondiente. Una vez ejecutado el criteria, se aplicarán cada una de las condiciones de memoria al resultado de haber ejecutado el criteria a través de Hibernate. Esto es muy útil para los casos de uso del caso de estudio pues existen condiciones que son mucho más simples de expresar en memoria y que no son las que hacen el filtro grueso, razón por la cual deberían ser aplicadas en BD para no perjudicar la performance.

- ✓ Poder realizar un post-procesamiento del resultado obtenido para, por ejemplo, crear un tipo de objeto particular que está esperando el "cliente" de este framework.

- ✓ Poder integrarse fácilmente con el framework de **"formularios de búsqueda"** que se describirá a continuación

Este framework de Repositorio podrá ser ejecutado desde la capa de objetos de dominio o desde la capa de servicios.
Además, se proveerá un servicio remoto que permita ejecutar parte de la API desde el cliente. Es decir, que desde el cliente y de manera transparente para el desarrollador se pueda:

- o Recuperar un objeto a partir de su id para llevar al cliente. Este objeto deberá ser serializable para poder viajar por la red.
- o Aplicar en la BD los cambios realizados sobre el objeto en el cliente.
- o Dar de alta un nuevo objeto a partir de datos ingresados por el usuario final en el cliente
- o Ejecutar un Search.

Más adelante veremos que además de este servicio que permite ejecutar parte de la API de Repositorio desde el cliente, existe otro servicio que permite hacerlo pero obteniendo DTOs (ver sección de patrones) en lugar de los objetos de

dominio. En ciertos casos será mejor trabajar en el cliente con objetos de dominio y en otros con DTOs, es por esto que consideramos que ambos servicios son necesarios.
Siendo que proponemos un framework para automatizar las tediosas y repetitivas tareas que trae consigo el uso de DTOs, dejaremos la descripción de este servicio para la sección que describa este frameowrk (*DyTO*)

Para maximizar el reuso, uno de los principales objetivos para mejorar la productividad, se incentiva el uso del patrón Factory para la creación de los searches. Estos factories podrán, en algunos casos, ser utilizados también por la aplicación cliente.

DYTO

Al describir el framework de Repositorio, mencionamos que en ciertos casos era una mejor alternativa utilizar DTOs en el cliente en lugar de utilizar los objetos de dominio. Es una discusión abierta en la comunidad si hacer una arquitectura que utilice o no DTOs pero que no será objeto de estudio en esta tesis. En el caso de estudio se detectaron casos en los cuales el uso de DTOs era una mejor alternativa. Por esta razón, y para evitar cargar al desarrollador con todas las tareas tediosas que trae consigo el uso de DTOs, se provee este framework desde la arquitectura.
Algunas de las tareas tediosas que trae consigo trabajar con DTOs:

- ✓ Crear las clases DTOs
- ✓ Instanciar un DTO a partir de un objeto de dominio
- ✓ Aplicar los cambios realizados sobre el DTO (en el cliente) en el objeto de dominio correspondiente, realizando los chequeos de concurrencia correspondientes.

El framework además de cubrir las necesidades básicas para simplificar el desarrollo de los DTOs, debería cubrir otros requerimientos que el uso de objetos (Hibernate) serializados ya provee:

- ✓ Propiedades lazy: Hasta que una propiedad no se accede, no se lleva al cliente.
- ✓ Colecciones paginadas

Agregamos a la componente un requerimiento que permita hacer Undo de las operaciones realizadas sobre un DTO desde un checkpoint definido explícitamente.

Si bien no entraremos en los detalles del framework, mencionaremos los puntos más importantes de su uso y las decisiones más importantes de diseño que permiten cubrir todos los requerimientos mencionadas.

Uso:

- ✓ El usuario del framework (desarrollador) deberá especificar los campos que quiere que formen parte del DTO

- ✓ Con anotaciones sobre la interfaz podrá configurar el DTO. Por ejemplo podrá especificar si una propiedad es o no lazy y podrá indicar que quiere que una colección quiere que sea paginada.

- ✓ Como se mencionó en la sección de Repositorio, existirá un servicio con parte de la API de repositorio que en vez de trabajar con los objetos, trabaja con DTOs. De esta manera, a través de este servicio el desarrollador podrá
    - o Recuperar un Dyto a partir de su id para llevar al cliente.
    - o Aplicar los cambios realizados sobre el dyto en la BD.
    - o Dar de alta un nuevo objeto a partir de un Dyto.
    - o Ejecutar un Search que retorne un conjunto de Dytos.

Decisiones de diseño:

- ✓ El corazón del Dyto será un STO (State transfer Object) que no es más que un mapa (con el nombre de la propiedad y el valor) más el identificador del objeto y la versión.

- ✓ El STO se wrapea con una clase Dyto que agrega la funcionalidad necesaria. Esta clase registrará cada una de las operaciones realizadas sobre el Dyto en una unidad de trabajo. Cuando se pida grabar el Dyto, se enviará esta unidad de trabajo, el id del objeto y la versión. Con toda esa información, y ya del lado del servidor, el framework podrá recuperar el objeto correspondiente, chequear la versión para verificar que no haya problemas de concurrencia y aplicar cada uno de los comandos de la unidad de trabajo.

- ✓ La instancia de la clase Dyto, a través de un mecanismo que permite crear proxies en forma dinámica, implementará la interfaz definida por el usuario. De esta manera el desarrollador no tendrá que trabajar con un mapa, sino que trabajará con los getters y setters de la interfaz que usó para describir el Dyto.

SERVICIOS

Como describimos en la sección de la arquitectura, cualquier operación estará representada por un servicio. Existen distintos tipos de servicios:

- CRUD de objetos: Alta, baja, modificación, consulta de objetos
- Servicios con lógica propia
- Reportes
- Transiciones: Ciertos objetos de la aplicación tienen un workflow de negocio que deben respetar. Un workflow es un grafo dirigido (los nodos son estados y las conexiones con las transiciones).

Para desarrollar un servicio un programador deberá desarrollar los elementos Java (y beans de Spring) correspondientes pero además deberá declarar el servicio y configurarlo.
Analizaremos, para cada tipo de servicio, cuales son los elementos Java que deben declarar y como declarar y configurar servicios utilizando una herramienta desarrollada para tal fin.

ELEMENTOS JAVA QUE SE DEBEN DECLARAR PARA DEFINIR UN SERVICIO

Servicios CRUD

En el caso de los servicios CRUD, existe el repositorio que permite ejecutar operaciones CRUD de manera genérica sobre cualquier clase de dominio. Con lo cual, el desarrollador solo deberá crear la clase de dominio y el mapeo correspondiente. NO necesitará hacer la clase CRUD específica para la clase de dominio. En caso de querer agregar funcionalidad antes o después, deberá crear una clase con dicha lógica y respetando una convención de nombres.

En la clase de dominio se debe implementar además un método predefinido, que chequea la integridad del objeto.

Servicios con lógica propia
En el caso de los servicios que tienen lógica propia, el desarrollador deberá implementar la interfaz del servicio y la clase con la implementación correspondiente.

Servicios de transición
En el caso de las transiciones deberán instanciar clases de un framework de "Diagrama de transición de estados" provisto desde la arquitectura y que describiremos más adelante.

Servicios de respores
En el caso de los reportes deberán instanciar clases de un framework de reportes provisto desde la arquitectura y que describiremos más adelante.

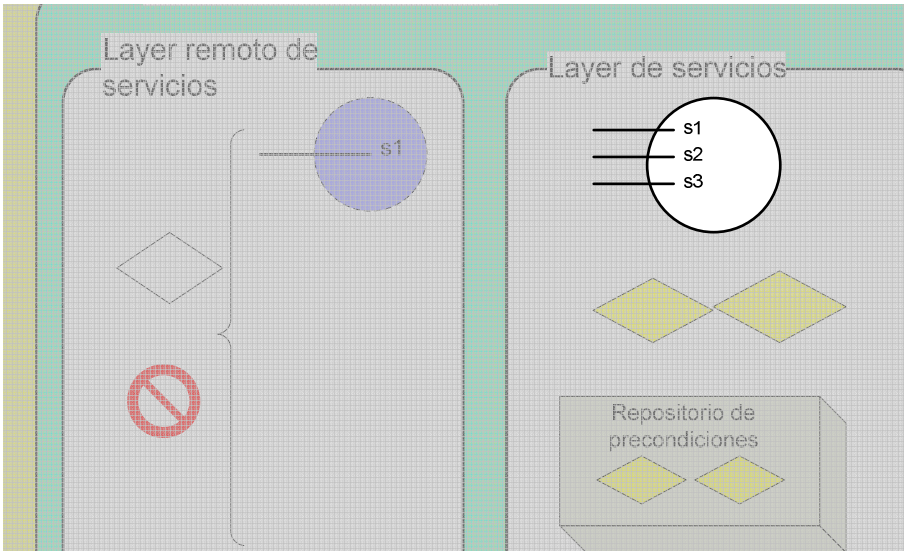
El framework de servicios provee un conjunto de clases y beans de Spring para heredar fácilmente funcionalidad que todo servicio debe tener:

- ✓ Para los servicios de la capa local: proveen
  - La ejecución de las precondiciones que se tienen que dar para poder ejecutar el servicio, teniendo en cuenta que algunas deben ser consideradas solo warnings (si fallan, igual el servicio debe ser ejecutado)
  - Log
  - Manejo de errores

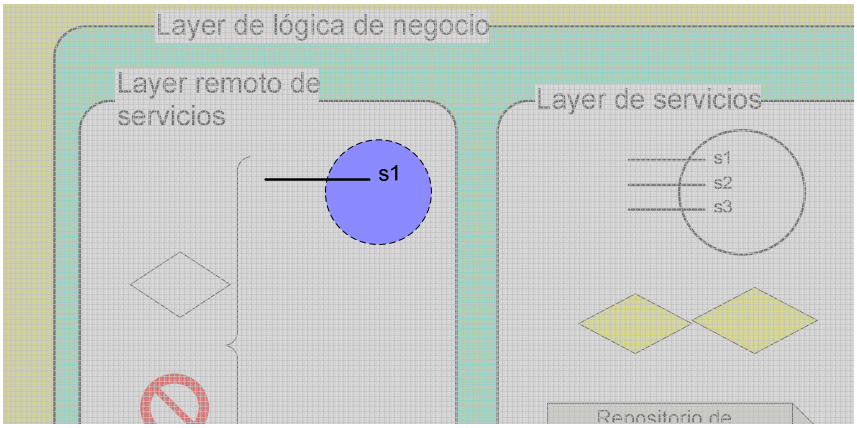
- ✓ Para los servicios de la capa remota, que en general no agregan implementación sino que delegan en el servicio de la capa local, proveen:
  - Seguridad: Autenticar al usuario y verificar que el usuario está autorizado a ejecutar el servicio.
  - Manejo de excepciones. Deben garantizar que al cliente no llegará una excepción de servidor.

DECLARACIÓN Y CONFIGURACIÓN DE SERVICIOS CON CO.S.E. (CONFIGURADOR DE SERVICIOS)

Aunque no todo servicio implique realizar una implementación en código Java, el servicio deberá ser siempre declarado y configurado en la capa de servicios locales. Esta declaración agrega la entrada en la "capa de servicios local".



Cuando un servicio quiera ser ejecutado remotamente, en general desde el cliente rico, deberá existir el servicio en la capa remota. En general, aunque no siempre, la capa de servicios remotos será uno a uno con la capa de servicios locales. Es decir, cada servicio local que se quiera ejecutar de manera remota deberá ser declarado y configurado para estar en la capa remota.



Cualquier servicio de la capa local podrá agregar precondiciones que se tienen que dar para poder ejecutar el servicio, considerando que algunas deben ser consideradas solo warnings (aunque fallen, el servicio debe ser ejecutado). Estas precondiciones estarán implementadas como clases Java. Para usuarios con permisos especiales, se debe mostrar una opción de menú que permite vulnerar los controles. Esto es, si un control restrictivo falla igual podrá ejecutar el servicio (registrando el log necesario).

Cuando el servicio se declare en la capa remota, se deberá generar el permiso que necesitarán tener los usuarios que intenten ejecutar la operación.

Teniendo en mente la mejora de la productividad y considerando que existirá una gran cantidad de servicios en el sistema y que será una actividad muy frecuente en el día a día de los desarrolladores, se decidió desarrollar una herramienta gráfica que simplifique la declaración y configuración de los servicios tanto de la capa local como de la capa remota. Esta herramienta es denominada CoSE: Configurador de servicios.

El Co.S.E. es un componente que queda instalado como plugin en el eclipse del desarrollador.

El CoSE permitirá:

- ✓ Declarar un nuevo servicio de cualquier tipo. El tipo determinará los parámetros que deberá completar el usuario de CoSE (desarrollador)

	Parametro1	Parametro2
CRUD	Clase de dominio	Operación (alta, baja, modificación,consulta)
NO CRUD	Interfaz Java	Método
TRANSICION	Clase de dominio	Estado+transición del workflow
Reportes	Clase del reporte	

- ✓ Agregar precondiciones a un servicio (restrictivas o warnings)

- ✓ Indicar si se va a ejecutar o no remotamente. Si así fuera, se generará no solo el servicio de la capa local sino también el de la capa remota y el permiso correspondiente

- ✓ Consultar descriptores de servicios declarados previamente

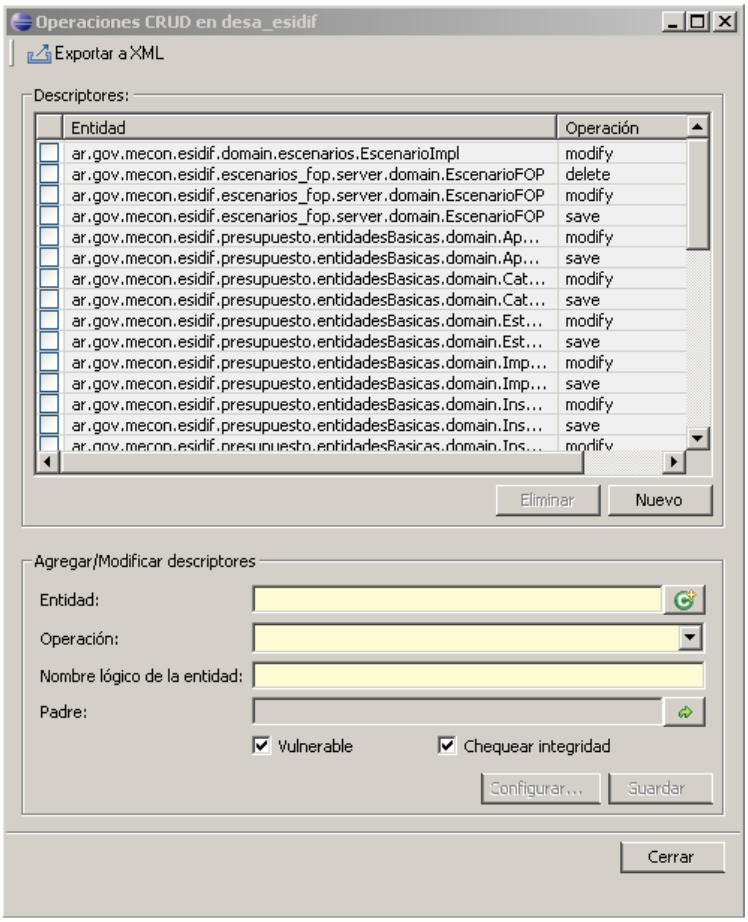
Revisemos la herramienta.

COSE: Declaración de Operaciones CRUD

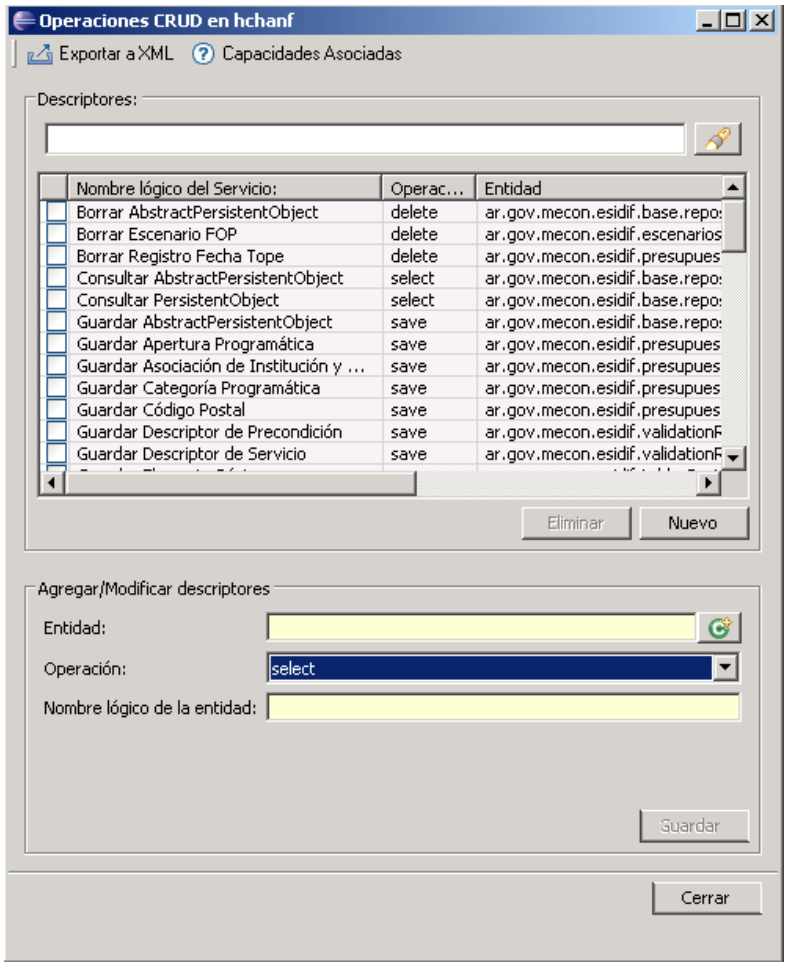
Cuando se va a declarar un servicio CRUD se debe indicar:

- ✓ La entidad para la cual se desea realizar la operación,
- ✓ La operación que se está realizando (save, modify, delete o select).
- ✓ Un nombre lógico para la entidad en cuestión y
- ✓ Opcionalmente se puede seleccionar un descriptor padre, del cual se heredará sus precondiciones.
- ✓ Si la operación sobre la entidad es create, modify o delete se debe determinar si el descriptor va a ser vulnerable y si requiere el chequeo de integridad de la entidad antes de ejecutar el servicio.

En la siguientes figuras se muestran un snapshots de la herramienta:



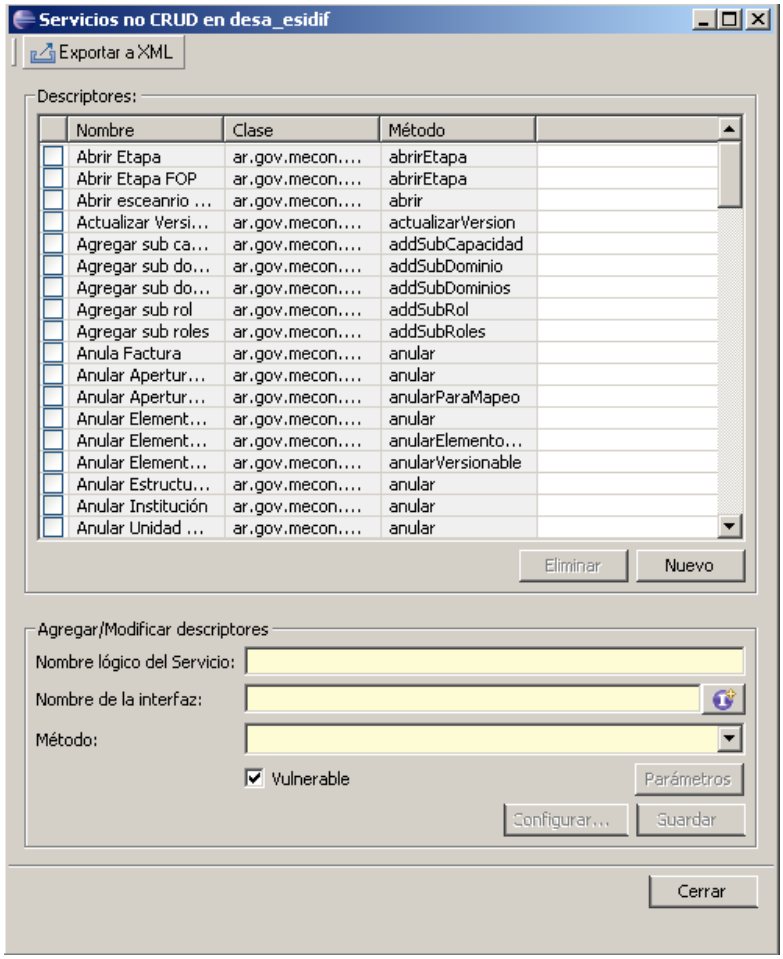
Si la operación es select no se setea ninguna de esas propiedades ni se le asocian precondiciones.



Para poder seleccionar una entidad que se encuentre en el workspace, se puede presionar Ctrl + Space, y se abrirá el buscador de clases del Eclipse. Si se quiere crear una nueva clase se puede presionar el botón correspondiente.
Se puede seleccionar un descriptor padre presionando el botón correspondiente.

En caso de elegir la opción de "REMOTO", se deberán especificar los datos del "Permiso de invocación a servicio". Este permiso es el que debe ser asignado a los usuarios para que puedan ejecutar este servicio.

Se debe indicar un nombre lógico para el servicio, el nombre de la interfaz y el método correspondiente al servicio.
También se determina si el descriptor va a ser vulnerable.

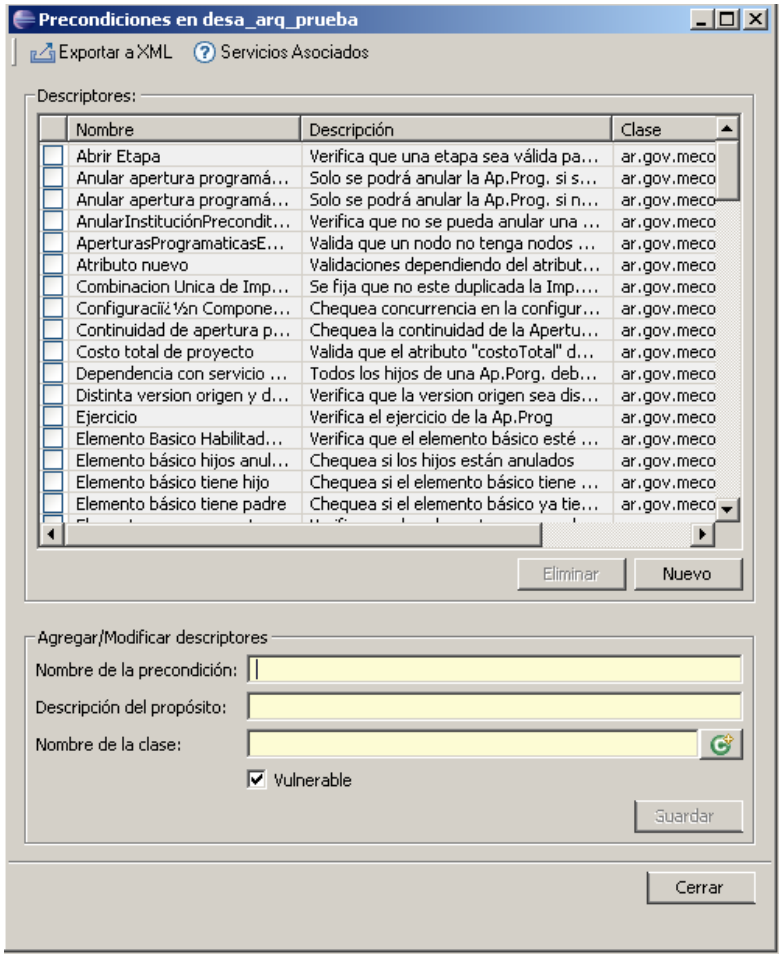


Para poder seleccionar una interfaz que se encuentre en el workspace, se puede presionar Ctrl + Space, y se abrirá el buscador de interfaces del Eclipse. Si se quiere crear una nueva interfaz se puede presionar el botón correspondiente. También estas opciones aparecen presionando el botón derecho del mouse.
Presionando en el botón Parámetros se deben definir los parámetros del método si es que corresponde, debiendo setear el nombre y tipo, opcionalmente una descripción y si se desea chequear integridad de los mismos.

En caso de elegir la opción de "REMOTO", se deberán especificar los datos del "Permiso de invocación a servicio". Este permiso es el que debe ser asignado a los usuarios para que puedan ejecutar este servicio.

CoSE: Precondiciones

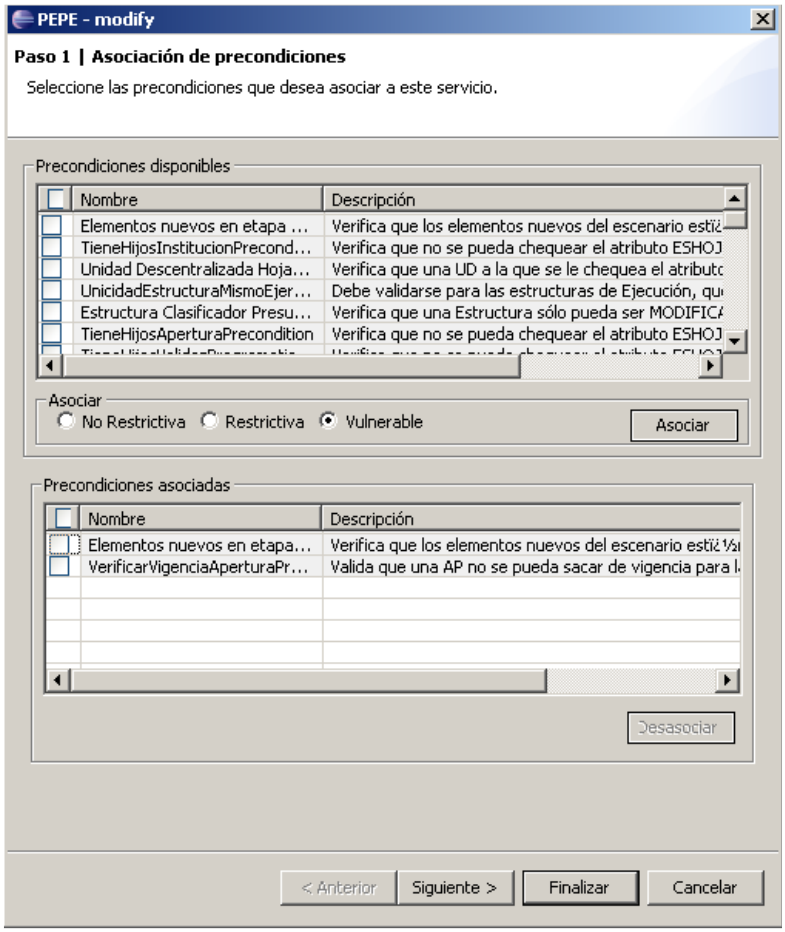
Se debe indicar el nombre de la precondición, describir el propósito de la misma y seleccionar el nombre de la clase que la implementa. Opcionalmente se puede marcar como vulnerable.



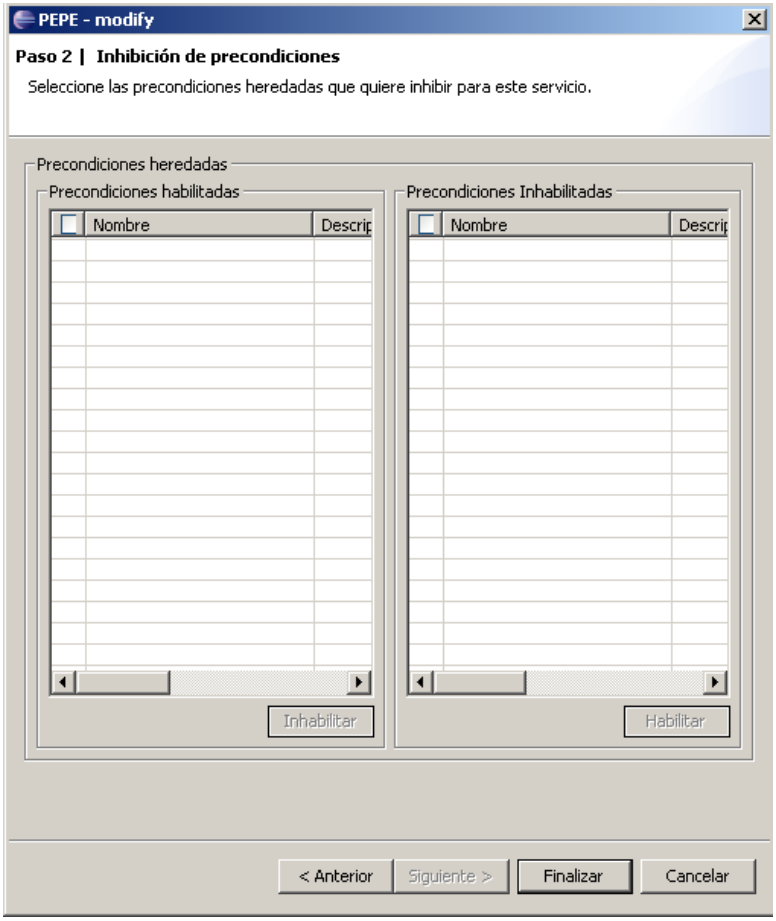
Para poder seleccionar una clase que se encuentre en el workspace, se puede presionar Ctrl + Space, y se abrirá el buscador de clases del Eclipse. Si se quiere crear una nueva clase se puede presionar el botón correspondiente.
También estas opciones aparecen presionando el botón derecho del mouse.
Presionando en el botón "Servicios Asociados" se pueden ver todos los descriptores de servicio que tienen la precondición seleccionada asociada.

Asociar precondiciones a un descriptor de servicio

Cuando se define un descriptor de servicio (cualquiera de los tres posibles), se le pueden configurar precondiciones de tipo no restrictivas, restrictivas y vulnerables. Para el último caso, se debe tener en cuenta que tanto el servicio como la precondición tienen que estar marcados como vulnerables.



Si el descriptor hereda de otro descriptor, se le pueden inhibir las precondiciones del padre que no se quiere que sean heredadas.



FORMULARIOS DE BÚSQUEDA

El framework para creación de "formularios de búsqueda" permite desarrollar componentes gráficas de búsqueda (aquellas que representan condiciones de búsqueda) reusables y pantallas de búsqueda en base a esas componentes reusables. **Como consecuencia del uso del framework y de la iniciativa de reuso que este propone, se consiguió una gran mejora de productividad y una estandarización en el desarrollo de formularios de búsqueda.**
El framework provee una clase SearchViewBuilder que recibe como parámetro un panel de búsqueda que se puede "dibujar" usando la herramienta WYSIWYG SWTDesinger ya mencionada. Esta clase se encarga de crear el modelo de este panel de búsqueda que será el Search con una condición para cada componente de búsqueda presente en el panel.

REPORTES

Si bien existen frameworks en la comunidad que permiten generar reportes en Java, se decidió wrappear a Jasper, para conseguir un framework que simplifique el desarrollo de reportes esidif.

Luego de un primer relevamiento detectamos que existen dos tipos de reportes: los reportes fijos y los reportes variables. El framework permite desarrollar ambos.

Los reportes variables son reportes que tienen formato de grilla, es decir de filas y columnas. En estos reportes es el usuario final el que decide que campos quiere mostrar. El desarrollador le brinda un super conjunto de columnas (posibles columnas que puede elegir el usuario final para su reporte). El usuario final, mediante un wizzard que provee el framework, elige no solo las columnas que quiere agregar a su reporte sino también los campos por los que quiere cortes de control, los filtros de búsqueda, la dirección de la hoja, si quiere o no mostrar los totales generales, si quiere o no mostrar la cantidad de elementos, la cantidad de decimales que quiere usar, si quiere o no descripciones de los campos entre otras cosas.

Para implementar un nuevo reporte variable, el desarrollador simplemente deberá instanciar una clase del framework (que representa un reporte variable) y configurarla. Esta configuración incluye:

- Clase de dominio sobre la cual se hará el reporte
- El super conjunto de columnas
- Los filtros de búsqueda
- Datos de configuración

El framework, a partir de esta instancia, se encargará de llevar adelante el reporte. Esto incluye mostrar al usuario final el wizzard correspondiente, tomar los datos de configuración que el usuario haya ingresado, enviarlos al servidor, generar el reporte, enviarlo al cliente y mostrar una vista que permite generar distintas salidas a partir de ese reporte.

En los reportes fijos, en vez de tener formato predefinido (grilla) como los variables, es el desarrollador el que tiene que dar el diseño del reporte. En Jasper esto es generar un archivo .jasper programándolo o a través de alguna herramienta como iReport. Además de crear el diseño Jasper, el desarrollador deberá instanciar una clase provista por el framework y configurarla.

El framework, a partir de esta clase hará todo el resto del trabajo: Arma el wizard para que el usuario final pueda ingresar la configuración del reporte.

CLIENTE RICO (REUSO DE COMPONENTES GRÁFICOS)

Como mencionamos durante la descripción de la arquitectura, el cliente de la aplicación es un cliente desktop. Para el desarrollo de las pantallas se utilizó SWT (Standard widget toolkit)

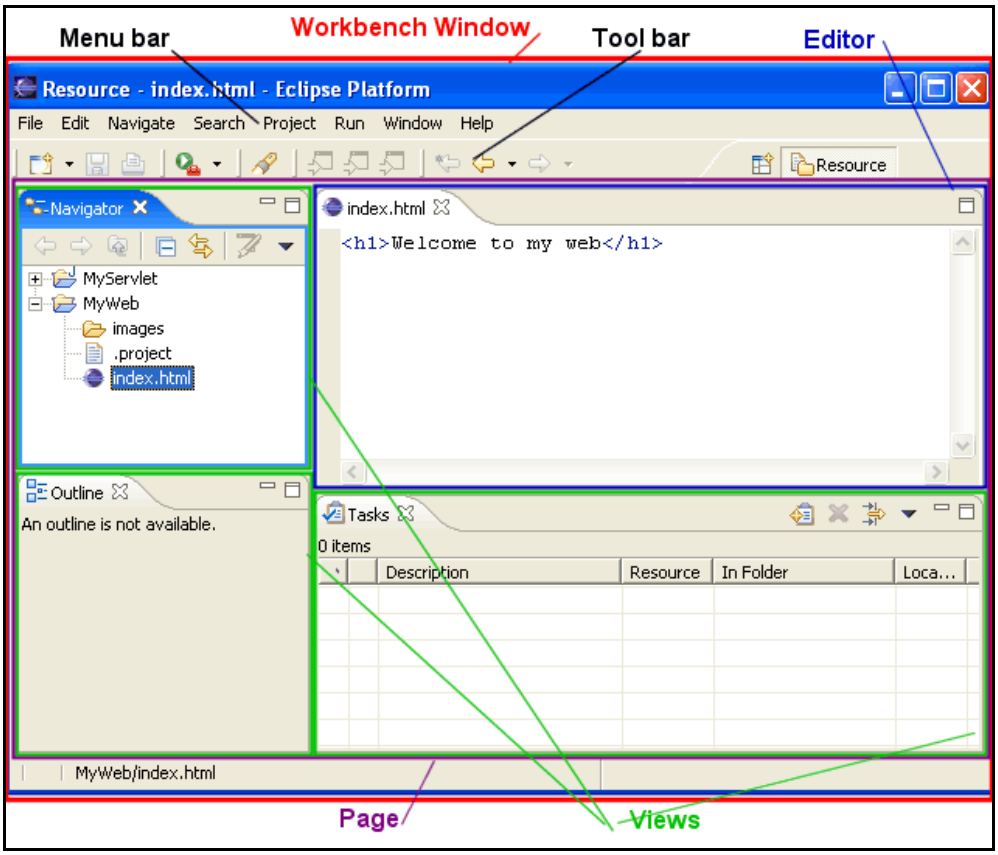
Para conseguir un diseño MVC y proveer un mecanismo que favorezca **el reuso de componentes gráficas que creemos es una de las claves de la productividad**, se desarrolló un framework que wrapea a SWT.

Esta componente wrapea muchas de las componentes de SWT agregando, además del diseño MVC, otras características que las componentes SWT no tienen.
Es importante hacer hincapié en la componente grilla que agrega mucha funcionalidad común a todas las grillas del sistema. Las más importantes son:

- Autofiltro
- Orden
- Paginado
- Romper paginado
- Impresión
- Columnas de imágenes
- Sincronización con vistas

ESCRITORIO

El espacio del trabajo definido por el área de usabilidad fue inspirado en Eclipse. Es por esta razón que se decidió utilizar RCP como tecnología base para armar dicho espacio de trabajo.
A continuación se muestra una imagen que presenta los elementos principales que provee RCP para armar el ambiente de trabajo de la aplicación:



Desde la arquitectura se proveyó un framework denominado "escritorio" que wrapea a RCP para agregar funcionalidad que el framework base no provee. Además, el framework "escritorio" automatiza muchas tareas relacionadas con el "ambiente de trabajo" que son iguales para todos los casos de uso del caso de estudio. De esta manera se simplifica la tarea del desarrollador y se mejora la tan buscada productividad.

A continuación se describen los puntos más importantes agregados por el framework de escritorio:

- Manejo de acciones:
  - ✓ Provee una clase Action que debe ser instanciada y agregada a algún contenedor (editor, vista, diálogo, wizzard). El framework asegura que cuando gana foco ese contenedor, se activan sus acciones. Cuando se instancia la acción se deberá indicar que es lo que debe ejecutar (servicio o alguna tarea en el cliente), condiciones funcionales para que se active y el lugar donde se quiere dibujar (menú bar, menú, menú contextual)

  - ✓ Integración con seguridad. Cuando la acción ejecuta un servicio, el framework se encarga de "consultar" al framework de seguridad si el usuario logueado tiene permiso para ejecutarla. Si no tuviera permiso, no prende la acción.

- Superclase para los editores esidif:
  - ✓ Brinda soporte para trabajar con modelos que sean DyTOS.

  - ✓ Agrega la opción para ver las propiedades del objeto que está detrás del editor

  - ✓ Agrega la opción para visualizar los datos de auditoría del objeto que está detrás del editor

  - ✓ Se encarga de activar las acciones relacionadas con el objeto que está detrás del editor.

- Vistas
  - ✓ Vista de entidades relacionadas: Es una vista que permite mostrar las vistas relacionadas con el objeto detrás del editor activo.

  - ✓ Vista de módulos: Permite dar una organización funcional a las acciones que el usuario puede ejecutar.

  - ✓ Supervista sincronizada: Clase base para permite crear vistas que se sincronicen con un editor o con una fila de una grilla.

- Estandarización de la visualización de errores
  - ✓ En caso de recibir una excepción del servidor en respuesta a la ejecución de un servicio, interpreta el contenido de la excepción (descripción de los errores y warnings) y presenta una vista distinguiendo ambos tipos de errores.

  - ✓ En caso de recibir un error inesperado lo muestra en un diálogo especial

SEGURIDAD

Permite manejar la autenticación y la autorización.
La autenticación verifica que el usuario es quien dice ser. Para esto, previo a arrancar la aplicación se presenta un diálogo de login y password. Esta información es chequeada contra un LDAP que mantiene los usernames y passwords de los usuarios registrados.
La autorización verifica que el usuario puede ejecutar el servicio que está intentando ejecutar. Seguridad provee un aspecto que puede agregarse fácilmente a los servicios que permite conseguir fácilmente y de manera desacoplada el chequeo de seguridad.

Generalización funcional

No se abordará en detalle como llevar a cabo la construcción de los frameworks funcionales, simplemente haremos una breve descripción de los frameworks desarrollados por el equipo de **arquitectura funcional.**

STE

Framework genérico para representar un workflow. El usuario del framework debe definir, para cada clase a la que quiera agregar un workflow (Clase que herede de StateHolder), los estados y transiciones usando programación Java. Luego, debe declarar cada transición como un servicio, usando la herramienta CoSE. Desde esta herramienta podrá asociar las precondiciones correspondientes.

Cuando en el cliente, el usuario final activa el editor de un StateHolder, el framework se encarga de activar solo las acciones correspondientes a las transiciones que "salen" del estado actual que tiene ese StateHolder.

Comprobantes

El framework abstrae la funcionalidad de todo comprobante. Provee la integración con el framework STE, haciendo que todo comprobante sea un StateHolder.

Provee la superclase de todo comprobante con la funcionalidad común y la superclase del editor de todo comprobante que tiene todo lo común a nivel cliente.

Cadena de firmas

Es un framework que permite agregar a ciertos estados del comprobante, una cadena de firmas. Esto es, poder especificar los roles que deben firmar para pasar el comprobante al próximo estado.

PARTE VI: CONCLUSIONES

Los desarrollos de software de gran escala y las arquitecturas de software han concentrado la mayor atención en la última década de las grandes compañías y pequeñas firmas de software en el mundo. Ha crecido el reconocimiento de la importancia del diseño y organización del sistema previo a la construcción, como así también de principios de diseño, de desarrollo y patrones reusables.

Si bien se desarrollan herramientas para facilitar el diseño, frameworks para potenciar el reuso, y se utilizan patrones de diseño para la construcción de sistemas, desafortunadamente el número de proyectos que fracasan es alto porque no se sostiene una metodología de trabajo a lo largo del proyecto que garantice el reuso y la productividad

Para sostener esa metodología es imprescindible entre otras consideraciones lo siguiente:

- Disponer en la etapa inicial del proyecto – antes de comenzar la implementación o construcción del sistema - de un equipo que defina la arquitectura del sistema para cubrir los requerimientos no funcionales y dar soporte a los funcionales.
- Durante el desarrollo mantener ese equipo para que brinde una implementación de la arquitectura y haga evolucionar la misma, de manera tal que se logre:
    - Estandarizar el desarrollo
    - Simplificar el futuro mantenimiento
    - Mejorar la productividad del equipo de implementación
    - Mejorar la calidad del producto generado.

La definición que se dio en esta tesis de arquitectura de aplicación fue "***soporte* necesario para la construcción de los casos de uso, que garantice el cumplimiento de los atributos de calidad, asegure la calidad y maximice la productividad."**
La misma por ser una **actividad** continua durante el proceso de desarrollo de un proyecto de software necesita, durante todo este proceso, de una metodología que le permita desarrollar estas actividades

Uno de los aportes de la presente tesis, es detallar las actividades que el equipo de arquitectura debe llevar a cabo para poder conseguir los objetivos antes mencionados. Para organizar las actividades se define y describe una metodología estructurada en etapas.

Las actividades incluyen:

- ✓ Definir la estructura global del sistema y las pautas de desarrollo.
- ✓ Definir las tecnologías que se utilizarán en cada uno de los elementos del marco de arquitectura definidos
- ✓ Actualización de las tecnologías y frameworks utilizados.
- ✓ Evaluación de frameworks existentes en la comunidad que puedan ser de utilidad en el proyecto.
- ✓ Evaluación de herramientas
- ✓ Definir pautas de uso de las tecnologías y frameworks elegidos.
- ✓ Diseño y Desarrollo de Componentes y frameworks que resuelvan los problemas comunes de las aplicaciones Enterprise (componentes estructurales).
- ✓ Diseño y Desarrollo de componentes y frameworks que abstraen conceptos comunes del negocio (componentes funcionales).
- ✓ Soporte al equipo de desarrollo
    - o Soporte técnico a los desarrolladores
    - o Capacitación al equipo de desarrollo
    - o Resolución de problemas de performance.

Las etapas propuestas por la metodología son:
Etapa 1: Marco de arquitectura.

Requiere como insumo la *línea base de requerimientos* (LBR) compuesta por los requerimientos significativos para la arquitectura más el conjunto de atributos de calidad.
El artefacto generado en esta etapa es la ***arquitectura estructural***.
En esta etapa distinguimos 5 pasos necesarios para lograr este artefacto que son:

- ✓ Definición del marco teórico de arquitectura,
- ✓ Primera implementación de la arquitectura,
- ✓ Documentación,
- ✓ Implementación de referencia,

- ✓ Prueba de carga.

Etapa 2: Generalización funcional.

Requiere como insumo la arquitectura estructural obtenida en la etapa anterior y el modelado de los casos de uso más significativos de toda la aplicación.
Se requiere realizar los siguientes pasos para la generación del artefacto

- ✓ Interpretación de los casos de uso
- ✓ Detección de reuso
- ✓ Desarrollo de funcionalidad general
- ✓ Prueba de conceptos

El artefacto generado en esta etapa es la ***arquitectura funcional básica***.

Etapa 3: Extensión, corrección y mantenimiento de la arquitectura.

Durante la etapa de extensión y mantenimiento, los insumos estarán dados por las necesidades del equipo de desarrollo y las cuestiones genéricas relevadas junto al equipo de análisis. La planificación de estos requerimientos estará alineada con las prioridades definidas por la gerencia del proyecto.
Como pasos recurrentes en esta etapa se detectan:

- ✓ Captura de requerimientos funcionales específicos
- ✓ Refactor constante de funcionalidad genérica
- ✓ Priorización de correcciones

El artefacto generado es la ***arquitectura terminada***.

El caso de estudio elegido es representativo de las aplicaciones Enterprise. La metodología propuesta fue aplicada exitosamente en el caso de estudio lográndose como resultado una arquitectura que responde a los requerimientos planteados. Las decisiones técnicas adoptadas pueden ser utilizadas en otros proyectos de la misma envergadura.

REFERENCIAS

[Cle96a] Paul Clements. "A Survey of Architecture Description Languages". Proceedings of the International Workshop on Software Specification and Design, Alemania, 1996.

[CN96] Paul Clements y Linda Northrop. "Software architecture: An executive overview". Technical Report, CMU/SEI-96-TR-003, ESC-TR-96-003. Febrero de 1996.

[Dij68a] Edsger Dijkstra. "The Structure of the THE Multiprogramming system." Communications of the ACM, 26(1), pp. 49-52, Enero de 1983.

[Wir71] Niklaus Wirth. The programming language Pascal. Acta Informatica, 1:35-63, 1971.

[DK76] Frank DeRemer y Hans Kron, "Programming-in-the-large versus programming-in-the-small". IEEE Transaction in Software Engineering, 2, pp. 80-86, 1976. [Shaw84] Shaw84 M. Shaw. "Abstraction techniques in modern programming languages." IEEE Software, Oct 84, 10-26 (SR).

[Shaw89]. M. Shaw. "Larger-scale systems require higher-level abstractions." Proceedings of the fifth International workshop on software specification and design, May 89, 143-146. Software Structures: comparison of typical architectures

[PW92]; PERRY, D.E.,.WOLF, A.L, Foundations for the Study of Software Architecture, ACM SIGSOFT, Vol17, No 4, Oct 1992, p40-52.

[WWI99] WWISA. "Philosophy". Worldwide Institute of Software Architects, http://www.wwisa.org, 1999.

[BR01] Jason Baragry y Karl Reed. "Why We Need a Different View of Software Architecture". The Working IEEE/IFIP Conference on Software Architecture (WICSA), Amsterdam, 2001.

[Cle96b] Paul Clements. "Coming attractions in Software Architecture". Technical Report, CMU/SEI-96-TR-008, ESC-TR-96-008, Enero de 1996.

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Design Patterns: Elements of reusable object- oriented software. Reading, Addison-Wesley, 1995.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad y Michael Stal. Pattern-oriented software architecture – A system of patterns. John Wiley & Sons, 1996.

[Ale77] Christopher Alexander. A pattern language. Oxford University Press, 1977.

[KRUTCHEN] Architectural blueprints – The "4 + 1" view model of Software architectuere

[FOWLER1] Fowler, Martin Patterns od Enterprise Application Architecture, Addison-Wesley 2002

[Kauf1] - Snodgrass, Developing Time-oriented Database Applications in SQL, Morgan-Kaufmann, 1999

BIBLIOGRAFIA

S.H. Kaisler, F. Armour, M. Valivullah, "Enterprise Architecting: Critical Problems", IEEE Proceedings 38va Conferencia Internacional en Hawaii en Ciencias de Sistemas, 2005

Thomas, Erl Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall, 2005

Bieberstein ,Norbert Service Oriented Architecture Compass, Pearson, 2006

Thomas, Erl Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall, 2005

Helland, Pat Thoughts on Data in Service Oriented Architecture, 2004

Kauffman, Morgan Transaction Processing -Concepts and Techniques, 1993

Lynch, Merrit, Weihl, Fekete. Atomic Transactions. Morgan-Kaufmann Publishers, 1994.

Molina, Garcia-Molina, Ullman, Widom. Database Systems: The Complete Book. Prentice Hall, 2002.

Weikum, Vossen. Transactional Information Systems. Morgan-Kaufmann. 2002.