FRAMEWORK DE SOPORTE AL DESARROLLO DE APLICACIONES DE NEGOCIOS

SEMINARIO FINAL

VERSIÓN FINAL

AUTOR:
CASAS, GONZALO [INF383]
LICENCIATURA EN INFORMATICA

TUTOR:
ING. ENZO DANIEL GARCIA / ING. GUSTAVO PEREZ ARES

INSTITUCIÓN:
UNIVERSIDAD EMPRESARIAL SIGLO 21

Noviembre 2006

ÍNDICE DE CONTENIDOS

ÍNDICE DE GRÁFICAS

FRAMEWORK DE SOPORTE AL DESARROLLO DE APLICACIONES DE NEGOCIOS

UNIVERSIDAD EMPRESARIAL SIGLO 21

INTRODUCCIÓN

El siguiente proyecto de desarrollo de software se constituye del diseño e implementación de una librería de clases o *framework* para dar soporte a las tareas de desarrollo de aplicaciones de negocios, abstrayendo al equipo de diseñadores y programadores de software de las labores de persistencia de datos en distintos sistemas de almacenamiento, y de las de gestión y control de interfaces, con especial énfasis en permitir la creación de software que sea agnóstico de la tecnología de presentación subyacente.

El primer punto tiene como finalidad primordial permitir que aquellos sistemas desarrollados utilizando las funcionalidades de éste proyecto puedan ser independientes del motor de base de datos seleccionado, y permite la migración entre sistemas sin modificar una sola línea de código.

Mientras que el segundo punto permite a un sistema desarrollarse independizado de la forma de acceso y presentación del mismo, y ejecutarse en distintas modalidades, ya sea vía Web o Desktop, sin modificaciones al código.

El proyecto se denomina *Independence Framework*, ya que éste refleja las intenciones primarias de su autor, esto es, promover y proveer mecanismos de independencia tecnológica entre aplicaciones de negocios y sus sistemas de soporte.

ANTECEDENTES

El problema de la dependencia en el desarrollo de sistemas existe desde el mismo momento en que se inició la construcción del primer sistema de negocios complejo, y ha sido atacado por múltiples organizaciones y personas en distintos momentos del tiempo, y con diferentes grados de éxito. Sin embargo, el problema permanece sin resolver.

Un relevamiento inicial de la situación actual en la comunidad de desarrollo destacó la carencia de sistemas que unifiquen éstos dos conceptos de independencia en un único producto. Existen, sin embargo, múltiples variedades de sistemas que proveen una o la otra funcionalidad separada, especialmente en el área de *Object/Relational Mapping*.

Se han logrado numerosos avances en éste campo, y, en menor cantidad, existen algunos desarrollos orientados a lograr independencia de la tecnología de presentación, pero invariablemente las implementaciones atacan sólo uno de las áreas de problemática. Así mismo, los desarrollos tienen curvas de aprendizaje muy elevadas para comenzar a utilizarlos.

DESCRIPCIÓN GENERAL DEL AREA PROBLEMÁTICA

Una de las constantes en el desarrollo de sistemas orientados a los negocios es ***el cambio***. El cambio es permanente, inevitable y hasta deseable para prevenir la obsolescencia del sistema. Sin embargo, la tasa de cambio en los sistemas ha crecido exponencialmente en los últimos 20 años, y para poder mantenerse a la par de éstos, debe existir un alto grado de desacoplamiento y componentización en los desarrollos, que permita evolucionar rápidamente, y detectar y controlar la propagación de un cambio estructural tempranamente.

La problemática fundamental a la cual apunta el presenta proyecto es reducir la dependencia o acoplamiento entre una aplicación de negocios y las tecnologías de soporte que ésta utiliza, es decir, motores de almacenamiento de datos y librerías gráficas de presentación de interfaces de usuario.

La misma afecta severamente numerosas áreas durante el desarrollo y mantenimiento de sistemas y aplicaciones de negocios de complejidad media y alta. El impacto más importante se da en la etapa posterior al desarrollo inicial, durante el período de mantenimiento, donde un cambio en alguna de las capas tecnológicas subyacentes dispara un efecto de cascada, propagando el cambio en el resto del sistema que muchas veces es difícil de ponderar y controlar.

Un caso típico de ésta problemática se da cuando es necesario alterar el nombre de algún campo o tabla o modificar la estructura de la misma en un sistema que tenga un acoplamiento fuerte con el motor de base de datos. En estas situaciones, las consecuencias pueden ser tan amplias y complejas de contrarrestar que se decide violar las reglas de diseño y modelado de base de datos para encontrar una solución alternativa o atajo que no implique cambios en el sistema actual. Esto trae como consecuencia evidente una degradación progresiva de la calidad técnica de la aplicación en cuestión.

Por otro lado, la dependencia generada con la tecnología de interfase gráfica, a veces es tan alta que la migración a otra tecnología es inviable, tanto en términos económicos como técnicos.

JUSTIFICACIÓN

En base a la problemática planteada es que surge la motivación de desarrollar el presente trabajo para cubrir la ausencia de un producto o herramienta que satisfaga las necesidades insatisfechas en cuanto a independencia y desacoplamiento de tecnologías de soporte, así como asistir en el proceso de generación de modelos de negocios válidos y simples de controlar.

Algunas de estas necesidades existen aún porque no hay productos que unifiquen de una forma simple y clara los dos ámbitos de la problemática planteada. En otros casos, no hay ninguna implementación que las cubra ni total ni parcialmente, como es el caso de la detección temprana de un cambio propagado desde la base de datos, o la consulta de datos en una modalidad totalmente alineada con el paradigma de orientación a objetos.

El impacto potencial que tiene la aplicación de un framework como éste en el proceso de desarrollo de aplicaciones en una organización es muy alto, en términos de incremento de la productividad promedio y la reducción drástica del tiempo requerido para colocar un producto en el mercado (tradicionalmente referido con el término en inglés *Time-to-market*).

OBJETIVOS

Analizar, diseñar e implementar un Framework o librería de clases utilizado el paradigma de orientación a objetos, que propicie y dé soporte al desarrollo de aplicaciones de negocios.

Proveer al desarrollador facilidades para crear modelos de negocios independientes de los almacenes de datos, y sistemas independientes de las tecnologías de presentación, aislando las complejidades y detalles de implementación técnica de cada uno, mejorando así la productividad del desarrollador al permitirle concentrarse únicamente en la definición de la *lógica del negocio*.

Alentar, a través del uso del Framework, el diseño de aplicaciones con separación entre capas, mediante la abstracción de conceptos y la aplicación de patrones de diseño.

Implementar un caso tipo de uso del mismo, y así poder estimar mediante métricas, el incremento de productividad introducido por el Framework en comparación con un enfoque tradicional.

LÍMITES & ALCANCES

El alcance del presente proyecto está expresando en los siguientes ítems:

- Proveer 100% de independencia y transparencia del almacén de datos.
    - Realizar el mapeo transparente de los objetos a sus almacenes de datos, y que el mismo pueda ser controlado programáticamente (desde código) o descriptivamente (desde archivos de configuración).
    - Generar dinámicamente las consultas al almacén de datos.
    - Permitir crear entidades persistentes que puedan almacenarse en distintos almacenes de datos sin necesidad de ser modificados.
    - Permitir consultar el modelo de datos con una sintaxis y semántica orientada a objetos fuertemente tipificada a través de objetos que representen expresiones de búsqueda y de ordenamiento.
    - Permitir el control manual o automático de los límites transaccionales.
- Proveer independencia de la tecnología de presentación (Web / Desktop).
    - Permitir la definición de interfaces genéricas en un lenguaje descriptivo neutral a la tecnología de presentación, expresado como un vocabulario XML.
    - Permitir la ejecución de una misma aplicación a través de una interfaz Web o una interfaz Desktop sin realizar cambios en la codificación.
- Propiciar la creación de modelos de negocios auto-validados incorporando mecanismos de validación en los métodos de acceso a los atributos de cada entidad.

Quedan excluidas del presente desarrollo, las siguientes características:

- El framework no proveerá capacidades de transacciones en dos fases por el momento.
- El framework no proveerá herramientas para generación de código, aunque propiciará la creación de las mismas por terceros.
- El framework no permitirá más de una estrategia de mapeo Objeto / Relacional.

GLOSARIO

ABREVIATURAS, DEFINICIONES & ACRÓNIMOS

A continuación se describen las abreviaturas, definiciones, y acrónimos utilizados a lo largo del presente documento.

FRAMEWORK

En desarrollo de software, es una estructura de soporte definida sobre la cual pueden organizarse y desarrollarse otros proyectos. Típicamente, un Framework puede incluir programas de soporte, librerías de código y un lenguaje de scripting entre otros elementos de software para asistir al desarrollo y combinar los diferentes componentes de un proyecto.

OBJECT/RELATIONAL MAPPING

O/R Mapping es el proceso de transformación entre los enfoques relacionales y los enfoques orientados a objetos y entre los sistemas que soportan dichos enfoques. Las diferencias entre ambos enfoques es referida comúnmente con el término en inglés “Impedance Mismatch”, que resulta de las diferencias estructurales entre un diseño normalizado de bases de datos relacionales y una jerarquía de clases orientada a objetos.

CAPAS

La definición de “capa” en el contexto del presente documento, se condice con la de *layer* utilizada por *Martin Fowler* en el libro *Patterns of Enterprise Application Architecture*, quien distingue entre los conceptos *tier* y *layer*, el primero refiriéndose a capas de separación física, mientras que el segundo se relaciona con la separación lógica de capas de un componente de software.

SCM

Administración de la Configuración del Software, por sus siglas en inglés (Software Configuration Management). Según la definición de Roger Pressman (2002), es un conjunto de actividades diseñadas para controlar el cambio al identificar los productos del trabajo (work products) que son probables de cambiar, estableciendo relaciones entre ellos, definiendo mecanismos para la administración de diferentes versiones de éstos, controlar los cambios impuestos, auditar y reportar los datos realizados.

En otras palabras, SCM es una metodología para controlar y manejar un proyecto de desarrollo de software.

Debido a la gran cantidad de terminología propia de la disciplina de SCM, se incluye a continuación una breve definición de aquellas que son utilizadas a lo largo de éste documento.

a. ÍTEMS DE CONFIGURACIÓN

Un objeto que es tratado como una unidad auto-contenida para los propósitos de identificación y control de cambios. Un ítem de configuración es una unidad de configuración que puede ser administrada y versionada individualmente.

b. LÍNEA BASE O BASELINE

Una especificación o producto que ha sido formalmente revisado y acordado, y a partir de ese momento, sirve como base para el desarrollo subsiguiente, y que solo pueden ser modificados a través del procedimiento formal de control de cambios.

Una línea base provee una instantánea de los componentes de un sistema en un punto dado del tiempo. Son utilizados comúnmente para establecer la integridad del sistema al final de cada fase del ciclo de vida del software.

c. CHECK-IN

Un check-in (también llamado *install*, *submit*, o *commit*), ocurre cuando una copia de los cambios realizada en la copia local de un ítem de configuración, es enviada al repositorio central de control de cambios.

d. RAMIFICACIÓN O BRANCHING

Existen varias interpretaciones del término *branching*. En su definición general, es un mecanismo utilizado por muchos sistemas de control de versiones para soportar el desarrollo de software concurrente, que crea una "línea de desarrollo paralela" sobre la cual puede trabajar un equipo de desarrollo.

En éste documento, se utiliza éste término para referirse a lo que comúnmente se denomina *ramificación orientada a proyectos*, donde se generan y organizan *branches*, (en español, "ramas") en el contexto de un producto o sistema completo. Este tipo de *ramificación* impone una estructura de árbol bastante uniforme para todos los ítems de configuración del sistema. En lugar de enfatizar las modificaciones por archivos individuales, la *ramificación orientada a proyectos* se enfoca principalmente en el flujo de cambios lógicos sobre un sistema entero.

e. RECONCILIACIÓN O MERGING

Es la acción de reconciliar múltiples cambios hechos a diferentes copias del mismo ítem de configuración. Generalmente es necesario cuando un ítem de configuración es modificado por dos o más personas en diferentes equipos al mismo tiempo. Más adelante, éstos cambios son combinados, resultando un único ítem de configuración conteniendo todos los cambios.

ANSI

El Instituto Nacional Estadounidense de Estándares (ANSI, por sus siglas en inglés: American National Standards Institute) es una organización sin ánimo de lucro que supervisa el desarrollo de estándares para productos, servicios, procesos y sistemas en los Estados Unidos. ANSI es miembro de la Organización Internacional para la Estandarización (ISO) y de la Comisión Electrotécnica Internacional (International Electrotechnical Commission, IEC). La organización también coordina estándares del país estadounidense con estándares internacionales, de tal modo que los productos de dicho país puedan usarse en todo el mundo. Por ejemplo, los estándares aseguran que la fabricación de objetos cotidianos, como pueden ser las cámaras fotográficas, se realice de tal forma que dichos objetos puedan usar complementos fabricados en cualquier parte del mundo por empresas ajenas al fabricante original. De éste modo, y siguiendo con el ejemplo de la cámara fotográfica, la gente puede comprar carretes para la misma independientemente del páis donde se encuentre y el proveedor del mismo.

UML

Lenguaje Unificado de Modelado (UML, por sus siglas en inglés, *Unified Modeling Language*) es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; aún cuando todavía no es un estándar oficial, está apoyado en gran manera por el

OMG (*Object Management Group*). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema de software. UML ofrece un estándar para describir un "plano" del sistema (modelo), incluyendo aspectos conceptuales tales como procesos de negocios y funciones del sistema, y aspectos concretos como expresiones de lenguajes de programación, esquemas de bases de datos y componentes de software reutilizables.

TRAZABILIDAD

La trazabilidad es un conjunto de medidas, acciones y procedimientos que permiten registrar e identificar cada producto desde su origen hasta su destino final.

Consiste en la capacidad para reconstruir la historia, recorrido o aplicación de un determinado producto, identificando:

- Origen de sus componentes.
- Historia de los procesos aplicados al producto.
- Distribución y localización después de su entrega.

CASE

Las Herramientas CASE (**C**omputer **A**ided **S**oftware **E**ngineering, *Ingeniería de Software Asistida por Ordenador*) son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software reduciendo el costo de las mismas en términos de tiempo y de dinero.

SGML

En ingles "Standard Generalized Markup Language" o "Lenguaje de Marcación Generalizado". Consiste en un sistema para la organización y etiquetado de documentos. SGML ha sido normalizado por la Organización Internacional de Estándares (ISO) en 1986.

El lenguaje SGML sirve para especificar las reglas de etiquetado de documentos y no impone en sí ningún conjunto de etiquetas en especial.

XML

El *Extensible Markup Language* (XML) es un formato de texto simple, y altamente flexible derivado de SGML (ISO 8879). Originalmente fue diseñado para la publicación electrónica a gran escala. XML juega un rol importante en el intercambio de datos en la Web y fuera de ella.

XML se ha consolidado definitivamente como un lenguaje estándar de información, tanto como medio de persistencia como medio de intercambio de información.

XML SCHEMAS

Las XML Schemas expresan vocabularios compartidos y permiten mecanismos para validar automáticamente reglas generadas por humanos. Proveen medios para definir la estructura, el contenido y la semántica de documentos XML. La definición de un documento XML Schema se realiza en lenguaje XML.

.NET FRAMEWORK

El Framework .NET creado por Microsoft, es una plataforma de desarrollo de software enfocada en el desarrollo rápido de aplicaciones, independencia de plataforma y transparencia de red. .NET el la iniciativa estratégica de Microsoft para la próxima década.

.NET brinda nuevas funcionalidades y herramientas a la API (Application programming interface). Estas innovaciones permiten a los programadores desarrollar apicaciones tanto para Windows como para la Web, asi como componentes y servicios. .NET utiliza el paradigma de orientación a objetos y está diseñado para se suficientemente genérico como para soporte más de un lenguaje para su sintaxis. En muchos aspectos, el Framework .NET es una evolución de la tecnología de Java Virtual Machine de Sun Microsystems.

CLR

Common Language Runtime, es la denominación de la maquina virtual de ejecución de la plataforma .NET de Microsoft®. Es la implementación de Microsoft® del estándar *Common Language Infrastructure (CLI)*. El CLR ejecuta un tipo de bytecode denominado *CIL (Common Intermediate Language)*.

POJO / POCO

Las expresiones POJO (Plain Old Java Object) y POCO (Plain Old CLR Object) se refieren a objetos simples que siguen la filosofía de “mientras más simple, mejor”. El término POJO (después aplicado a la plataforma .NET como “POCO”) fue acuñado por Martin Fowler, Rebecca Parsons y Josh MacKenzie en septiembre de 2000. En general, la denominación POJO también aplica a lo que se conoce como JavaBeans, con la excepción de que éstos últimos adhieren a una estricta convención de nombres.

MARCO TEÓRICO

Existen numerosos estudios teóricos sobre los conceptos vertidos anteriormente, algunos de ellos expresados en forma de patrones de diseño de software. En las secciones subsiguientes, se analizan aquellos que brindan un marco de soporte para la comprensión del presente documento.

DESCRIPCIÓN Y FUNDAMENTACIÓN TEORICA GENERAL

> *"El propósito de un framework es hacer más fácil la construcción de aplicaciones." (Johnson R.E., 1998)*

Un framework es un componente fundacional en el desarrollo de software. Provee los cimientos sobre los que se apoya un sistema. Básicamente, es un conjunto de abstracciones reusables diseñadas para solucionar problemas recurrentes y / o simplificar tareas comunes a un número de aplicaciones.

La reusabilidad de un framework es directamente proporcional al nivel de éxito que el mismo puede alcanzar, sin embargo, diseñar componentes reusables no es un tarea simple y, generalmente, requiere de múltiples iteraciones. La reusabilidad del software es considerada una precondición técnica crucial para mejorar la calidad global del mismo y reducir los costos de producción y mantenimiento (Pree W., 1994).

Para alcanzar la reusabilidad, un framework debe ser abstracto y potente para permitir ajustarse a diferentes situaciones, debe ser extensible y debe ser simple de comprender. La simplicidad es un factor a veces desestimado, sin embargo, es uno de los factores clave de éxito en un proyecto de éste tipo. El razonamiento detrás de ésta afirmación es evidente: *Una pieza de software compleja de comprender será utilizada por pocas o ninguna persona.*

A menudo, los frameworks basan su desarrollo en patrones de diseño bien conocidos y probados que aportan una solución general y repetible a problemas de común ocurrencia en el software. Sin embargo, es importante destacar una diferencia: los patrones de diseño no son soluciones finalizadas que pueden ser transformadas directamente al código, sino que representan descripciones o "plantillas" para solucionar un problema que puede ser usadas en muchas situaciones. Por el contrario, un framework es una implementación concreta que apunta a solucionar o simplificar un problema o dominio definidos.

Salvando las diferencias, se podría establecer una analogía entre los framework de desarrollo de software y los modelos matemáticos, ya que ambos operan sobre la realidad, generando abstracciones más simples de comprender.

CATEGORIZACIÓN DE FRAMEWORKS

Los frameworks de software orientados a objetos (OO) utilizan principios de diseño que en la mayoría de los casos son independientes del dominio al cual éstos son aplicados. Sin embargo, es útil establecer algunas clasificaciones:

1. **Por su alcance:**

   - *System infrastructure frameworks (Infraestructura de sistemas):* Orientados a simplificar el desarrollo de infraestructuras de sistemas portables y eficientes. Por ejemplo frameworks de comunicación, frameworks para interfaces de usuario y herramientas de procesamiento de lenguajes.

   - *Middleware integration frameworks* (Integración): Utilizados comúnmente para integrar aplicaciones y componentes distribuidos. Están diseñados para simplificar la modularización y facilitar la operación en entornos

distribuidos. Entre los ejemplos más comunes de ésta categoría encontramos los ORB (Object-Request Broker) frameworks, middleware orientado a mensajería, etc.

- *Enterprise application frameworks (Aplicaciones empresariales):* Éstos frameworks abarcan amplios dominios de aplicación, como ser telecomunicaciones, aviación, etc. y son la piedra fundacional de las actividades empresariales.

Las primeras dos se enfocan principalmente en los detalles internos del desarrollo de software, mientras que la tercera está claramente enfocado en los procesos de negocios.

2. **Por las técnicas utilizadas:**

- *Frameworks de caja blanca:* Este tipo de desarrollos se apoya ampliamente en las características propias de los lenguajes orientados a objetos, como la herencia y la vinculación dinámica (*dynamic binding*), para alcanzar la extensibilidad. La funcionalidad existente es reutilizada y extendida heredando clases bases del framework y sobrescribiendo métodos de enganche predefinidos
- *Frameworks de caja negra:* En éste caso, la extensibilidad es soportada por la definición de interfaces o contratos para componentes, que son conectados al framework a través de la composición de objetos. La funcionalidad existente es extendida al definir componentes que se ajustan a una interface determinada y se integran en el framework utilizando patrones de diseño como *Estrategia* (*Strategy).*

Ésta última categorización no es estricta: no existen frameworks totalmente de caja blanca, ni totalmente de caja negra, sino un espectro gradual de variantes intermedias.

ELEMENTOS DE UN FRAMWORK

Los frameworks de software consisten en bloques constructivos semi-finalizados y listos para usar (Pree W., 1994). Estos están compuestos principalmente por dos tipos de elementos claramente diferenciables:

- *Puntos congelados (Frozen spots):* Son aquellos elementos que conforman la estructura fundamental del framework. Definen la arquitectura general del mismo, así como los componentes básicos y las relaciones entre ellos. Su comportamiento está prefijado, y permanecen idénticos en cada instancia del framework.
- *Puntos calientes (Hot spots):* Por el otro lado, los *puntos calientes*, están diseñados para ser adaptados a las necesidades de la aplicación que se desarrolla. Proveen flexibilidad y extensibilidad al diseño, y representan una métrica concreta de la calidad del framework en términos de reusabilidad. En el paradigma de orientación a objetos, un *punto caliente* está dado por la existencia de *métodos de enganche (hook methods)*, es decir, métodos declarados abstractos, cuyo comportamiento es modificado gracias a las características de herencia y polimorfismo.

TIPIFICACIÓN DE USUARIOS

Un framework es un componente de software diseñado específicamente para ser utilizado por desarrolladores de software, que apunta a simplificar la tarea de la construcción de aplicaciones. Por ende, el tipo de cliente de un framework es radicalmente distinto a un usuario final de un sistema de negocios.

El usuario final de un framework tiene conocimientos técnicos amplios sobre las tecnologías subyacentes y sobre la problemática que éste soluciona. Por éstas razones, tiene objetivos claramente definidos en cuanto a las características que busca en un framework y, generalmente, tiene una opinión formada sobre cual es la mejor forma de solucionar la problemática en cuestión. Esto normalmente deriva en una polarización subjetiva que genera "corrientes" o "grupos de preferencias" en cuanto al uso y desarrollo de frameworks.

OBJETOS DE ESTUDIO ESPECÍFICOS

El presente documento se enfoca en el estudio y desarrollo de un framework de soporte al desarrollo de aplicaciones de negocios basándose en dos objetos de estudio específicos, y comunes al grueso de sistemas empresariales:

- La problemática de la persistencia, principalmente los almacenes de datos relacionales, y la forma en que éstos interactúan y se coordinan con el paradigma de orientación a objetos.
- La definición de interfaces de usuarios portables, simples y efectivas; y las implicancias del acoplamiento entre las aplicaciones de negocios y las tecnologías de presentación.

LA PROBLEMÁTICA DE LA PERSISTENCIA

La mayoría de las aplicaciones de negocios modernas se estructuran alrededor de dos ejes principales:

- Una plataforma de programación orientada a objetos, como J2EE o .NET.
- Un motor de base de datos relacional.

Esto no implica que no existan otras opciones, como bases de datos de objetos o de XML, lenguajes procedurales como COBOL, etc. Sin embargo, la norma de facto en cuanto a desarrollo de aplicaciones empresariales se refiere, es, por lejos este conjunto de elementos mencionados.

Sin embargo, existen diferencias fundamentales entre el enfoque relacional y el paradigma de orientación a objetos. El paradigma OO está basado en principios de ingeniería de software, mientras que el enfoque relacional se basa en principios matemáticos. Estas diferencias son comúnmente referidas en ingles con el término "*impedance mismatch*", (en castellano, *diferencial de impedancia*).

En electrónica, se utiliza éste término cuando no podemos conectar dos componentes con diferente impedancia. Lo mismo ocurre cuando se intenta conectar y conciliar los conceptos utilizados en una aplicación orientada a objetos, con un motor de base de datos relacional.

Las diferencias más notables entre ambos modelos son las siguientes:

- **Foco:** Evidentemente, hay una diferencia de enfoque esencial entre los dos modelos. El relacional está claramente orientado a los datos, mientras que el de objetos incorpora el comportamiento de negocios fuertemente acoplado a los datos.
- **Navegación:** Una de las diferencias más importantes entre los dos enfoques es la manera de representar las relaciones y la forma de navegación entre elementos relacionados. Mientras que en el mundo de objetos, las relaciones se representan con referencias en memoria que apuntan al objeto relacionado, en el modelo relacional las relaciones se establecen mediante un identificador, y para acceder al elemento relacionado es necesario realizar una nueva consulta al motor de base de datos.

- **Relaciones muchos a muchos:** Otra diferencia notable es la ausencia de relaciones muchos-a-muchos en el modelo relacional, principalmente debido a una limitación impuesta por las formas normales. Para compensar esta faltante, el modelo relacional introduce las tablas de asociación, que permiten modelar una relación muchos-a-muchos.
- **Herencia:** La herencia es un concepto que no tiene equivalente en el mundo relacional. Es una de las características más notables del paradigma de orientación a objetos, que permite crear complejas jerarquías de relaciones que comparten atributos y comportamiento con sus ancestros. Asociado estrechamente a ésta característica está el polimorfismo, que, evidentemente, tampoco está soportado por el modelo relacional.

Existen dos posibles métodos para reconciliar estos dos mundos: el primero, y menos recomendable, es implementar la persistencia manualmente incorporando los comandos SQL en el código; el segundo consiste en utilizar algún tipo de framework de persistencia que gestione las tareas relativas a la interacción con el motor de base de datos.

A pesar de que la elección entre ambos debería ser lógica, ya que el primer método es negativo en términos de mantenibilidad de la aplicación, no es raro ver aplicaciones que lo utilizan, principalmente por motivos de performance.

La incorporación de comandos SQL en el código fuente presenta numerosas complicaciones a la hora de mantener y modificar un sistema. Además, se genera una fuerte dependencia y acoplamiento entre el mecanismo físico de persistencia (el motor relacional) y el sistema. Generalmente, esto se traduce en la imposibilidad práctica de portar un sistema a otro motor relacional, ya que los comandos SQL dispersos a lo largo del código pueden y suelen no ser 100% compatibles con el estándar ANSI.

Por otro lado, los frameworks de persistencia también plantean numerosos retos. Una de las ventajas principales de utilizar éste tipo de método, es la automatización de tareas repetitivas de persistencia, y por lo tanto la simplificación del código fuente. Sin embargo, si bien los frameworks de persistencia solucionan algunos de los inconvenientes que presenta la inclusión de SQL directamente en el código, también traen aparejados nuevas problemáticas.

a. CATEGORIZACIÓN DE MÉTODOS DE PERSISTENCIA

Entonces, a la hora de decidir que método de persistencia es conveniente, podemos empezar por plantear la siguiente categorización:

- **Persistencia manual:** SQL en el código
- **Persistencia automática:** Frameworks de persistencia
  - *O/R Mappers*
  - *Generadores de código*

Como ya discutimos en el punto anterior, la persistencia manual en realidad no es un método recomendable, ya que opera en detrimento de la calidad y estabilidad del software, y solo debería utilizarse en casos críticos donde un framework de persistencia no provee la flexibilidad y/o performance necesarias. Este punto, sin embargo, es ampliamente discutible, ya que los beneficios en términos de mantenibilidad y claridad que provee un framework de persistencia, en general, superseden ampliamente otros factores. La decisión en éste ámbito habitualmente se reduce a un análisis de costo-beneficio que debe realizarse por proyecto individual.

Por el otro lado, la opción de utilizar un método de persistencia automático –un framework de persistencia –puede sub-catalogarse en dos grandes grupos: los *O/R Mappers*, y los *generadores de código*. Los del primer grupo operan principalmente en tiempo de ejecución, utilizando archivos de mapeo (normalmente en formatos XML) y las capacidades de introspección y reflexión de las plataformas orientadas a objetos; mientras que los segundos en

realidad son herramientas de generación, que construyen las clases de persistencia necesarias para el mapeo en tiempo de construcción. Entre medio, existe un amplio espectro de opciones que se ajustan, en mayor o menor grado a éstas categorías, y utilizan aspectos de ambas modalidades.

Si bien es cierto que la denominación "*O/R Mapper*", en realidad es un tanto incorrecta, y debería aplicarse por igual a cualquiera de los grupos nombrados anteriormente ya que ambos encaran la tarea del mapeo entre el modelo de objetos y el relacional, existe cierto consenso en la comunidad sobre el uso de éstos términos.

El objeto de estudio del presente proyecto se centra principalmente en el grupo de *O/R mappers*, aunque se utilizan también algunos conceptos de *generadores de código* en aquellos casos donde se necesita balancear el impacto de performance causado por la reflexión de objetos. Con todo esto, se intenta obtener un producto que, si bien provee un mapeo sumamente transparente y aísla totalmente al desarrollador del motor de base de datos subyacente, puede establecer algunos compromisos en pos de un nivel aceptable de performance.

b. DESARROLLOS EXISTENTES

Actualmente existe en el mercado una amplia variedad de productos, tanto comerciales como Open-Source, que proporcionan distintas implementaciones de persistencia automática, desde el mapeo totalmente transparente de objetos planos (POJO, Plain-Old-Java-Object y POCO, Plain-Old-CLR-Object) hasta los generadores de código que construyen entidades de negocio completas a partir de una definición de mapeo o directamente desde una base de datos.

El siguiente es un relevamiento de las opciones más relevantes, que si bien no es exthausiva en cuanto a la cantidad, es representativo de la calidad y variedad de opciones.

Hibernate

Hibernate es un producto *Open Source* muy popular para persistencia y mapeo de objeto a relacional para Java. Existe también una versión portada a la plataforma .NET. Ambas versiones son sumamente estables y tienen varios años de madurez. Hibernate provee persistencia a objetos POJO / POCO, depende fuertemente de las capacidades de reflexión de la plataforma y define el mapeo en archivos descriptivos en formato XML. Opcionalmente permite la definición del mapeo a través de atributos en código.

Para la consulta de datos utiliza un lenguaje propietario (HQL) no-tipado definido como "una extensión portable de SQL". A su vez, existe una modalidad de consulta utilizando criterios, que incorpora un nivel de type-safety al declarar el tipo de objetos que se espera, pero utiliza strings para la configuración de los atributos comparados. Las versiones más nuevas del producto permiten también consultas mediante ejemplo.

Hibernate, utiliza un modelo en el cual la ejecución de sentencias SQL no es totalmente deterministica, sino que es diferida hasta que se cumplan una serie de condiciones que actúan como disparador. También provee una serie de métodos para disparar la ejecución.

Disponible en: http://www.hibernate.org/

LLBLGen Pro

Es una herramienta de generación de código comercial. Tiene varios años de madurez, y en la actualidad cuenta con varias características importantes, como una IDE de generación bastante avanzada, un mecanismo de consultas fuertemente tipado, soporte integrado para numerosas bases de datos, etc.

Disponible en: http://www.llblgen.com/

Apache Object/Relational Bridge - OJB

Es un proyecto Open Source de O/R mapping para la plataforma Java. Actualmente es parte del proyecto Jakarta de la Apache Fundation. OJB es una herramienta de mapeo de objetos a relacional que permite persistencia transparente de objetos Java en motores de base de datos relacionales. OJB provee distintas APIs para el acceso a la funcionalidad de persistencia, un ajustada al estándar ODMG y otra al JDO. OJB es publicado bajo la licencia GNU LGPL.

Disponible en: http://db.apache.org/ojb/

iBatis

iBatis es un proyecto de Apache para el mapeo de objeto a relacional. También ofrecido bajo la modalidad *Open Source*, éste proyecto realiza el mapeo entre objetos y relacional mediante un descriptor en formato XML, al igual que la mayoría de los productos de éste tipo, sin embargo en éste caso el mapeo se realiza entre un objeto y una sentencia SQL determinada, en lugar de realizarse contra una estructura de tablas en un motor de base de datos. Existen versiones para Java y para .NET.

Disponible en: http://ibatis.apache.org/

OpenQuasar

OpenQuasar es un proyecto de O/R Mapping Open-Source desarrollado en Alemania, que cuenta con varias características interesantes. Quizás la limitación más importante de éste producto es que toda la documentación, a pesar de que es muy extensa y detallada, está en idioma alemán.

Disponible en: http://www.openquasar.de/

Microsoft DLinq

El proyecto *Microsoft® DLinq* es parte la infraestructura de *O/R Mapping* previamente denominada *ObjectSpaces*. Originalmente, estaba previsto el lanzamiento de *ObjectSpaces* para la versión 2.0 del Microsoft .NET Framework, sin embargo, el proyecto sufrió numerosos atrasos y modificaciones y aún no ha salido de Beta. Hace poco fue publicado el proyecto *Linq*, del cual forma parte *DLinq*: un lenguaje de consulta fuertemente tipado orientado a objetos con una orientación funcional. La sintaxis de DLinq es similar a la de SQL, pero a diferencia de éste, está íntimamente ligado a la plataforma orientada a objetos y por lo tanto cuenta con todas las características de la misma: type-safety, herencia, polimorfismo, etc.

El proyecto DLinq se apoya en numerosas características funcionales que fueron incorporadas a la plataforma .NET, como ser Lambda expressions, dynamic type-checking, etc. *DLinq* se presenta como una de las opciones con mejores perspectivas para brindar una solución definitiva y de raíz a la problemática de la persistencia y el diferencial de impedancia, pero aún es muy prematuro, y recién está prevista su inclusión a partir del .NET Framework 3.0.

Disponible en: http://msdn.microsoft.com/data/ref/linq/

DEFINICIÓN DE INTERFACES DE USUARIO

Entendemos por interfaces de usuario (*UI*, en ingles, *User Interface*) a una representación, en general, gráfica, que permite al usuario de un sistema interactuar con éste de una manera comprensible, clara y eficiente, despojada de las complejidades técnicas propias de la codificación de un sistema. Es el vínculo entre el usuario y el software, generalmente compuesta por un conjunto de comandos o elementos de menú a través de los cuales se establece la interacción.

Las interfaces de usuario se presentan hoy en día en una numerosa variedad de plataformas. Cada plataforma cuenta con un conjunto de singularidades, lo cual implica una enorme brecha de incompatibilidad entre distintas plataformas. A las diferencias tecnológicas, se suman restricciones de formato, de objetivo y muchas más: existen plataformas móviles con interfaces que deben caber en una pantalla de menos de 5 cm de ancho, existen plataformas que utilizan interfaces sonoras, como los dispositivos IVR (centrales de atención telefónica), etc.

En las próximas páginas, se intenta describir la problemática relacionada a la descripción y definición de interfaces de usuario, así como una breve reseña histórica de la evolución de las mismas.

a. TIPOS DE INTERFACES DE USUARIO

Las interfaces de usuario pueden ser catalogadas de acuerdo a una multitud de criterios: son su nivel de interacción, según el método de interacción, según su apariencia, etc. Sin embargo, vamos a centrarnos en la clasificación de interfaces de acuerdo a su estilo de presentación. Entre los tipos más notables dentro de ésta categorización encontramos las siguientes:

- **Interfaces batch:** Las interfaces batch no son interactivas, el usuario especifica todos los detalles del trabajo que desea realizar por anticipado, y recibe la salida cuando acaba el procesamiento. Una vez que se ha iniciado el procesamiento, la computadora no vuelve a solicitar interacción con el usuario hasta que finaliza.

- **Interfaces textuales:** Las interfaces de texto, o interfaces de línea de comando utilizan el teclado como método de interacción con el usuario. Los comandos son ingresados como líneas de texto, y el resultado del mismo se muestra por pantalla también en formato de texto. Este tipo de interfaces se originaron en la década del 50, cuando se conectaron máquinas de teletipo a las computadoras. Este tipo de interfaces son aún muy utilizadas para tareas de administración de sistemas.

- **Interfaces gráficas:** Las interfaces gráficas de usuario (GUI, por sus siglas en inglés, *Graphical User Interface*) representan la evolución más palpable en el campo de la interacción del usuario y el software. El paradigma de éste tipo de interfaces se denomina WIMP (por sus siglas en inglés: *Windows, Icons, Menu, Pointing device*), y se aplica a casi todas las interfaces gráficas comercialmente disponibles hoy en día. En las próximas páginas exploraremos la evolución de éste paradigma hasta la actualidad.

- **Interfaces auditivas:** Las interfaces auditivas se utilizan comúnmente en las aplicaciones telefónicas, y utilizan como método de interacción el teclado telefónico o la voz del usuario en algunos casos. Los resultados de cada operación están limitados a una representación auditiva.

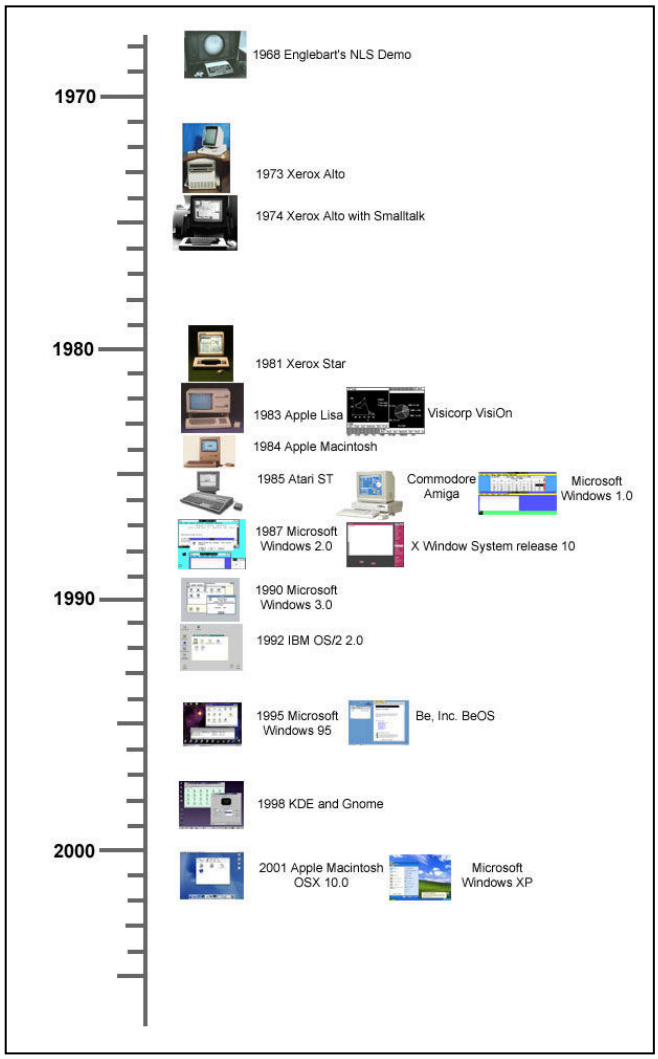
El presente documento se centra en el análisis y estudio de las interfaces gráficas de usuario (GUI), ya que las mismas representan la gran mayoría de las aplicaciones de negocios. Fundamentalmente, el foco se orienta a las interfaces gráficas de usuario montadas sobre plataformas Desktop y plataformas Web.

b. HISTORIA DE LAS INTERFACES DE USUARIO

Los diseñadores de software que no comprenden la historia que les precede, usualmente se encuentras a si mismos condenados a repetirla. Por ello, antes de atacar el tópico de las interfaces de usuario, se incluye una breve reseña de su evolución hasta la actualidad.

La creación de las interfaces de usuario es un área del desarrollo de software que ha evolucionado dramáticamente a partir de la década de los setentas.

Como muchos desarrollos en la historia de la computación, algunas de las ideas de las interfaces gráficas de usuario fueron concebidas mucho antes de que la tecnología para realizaras estuviera disponible. Una de las primeras personas en expresar éstas ideas fue Vannevar Bush. En los comienzos de la década del 30, Bush escribió sobre un dispositivo que bautizó “Memex”, un aparato similar a un escritorio con dos pantallas gráficas táctiles, un teclado y un scanner, que permitiría a un usuario acceder a todo el conocimiento humano utilizando conexiones muy similares a lo que hoy conocemos como hyperlinks.



En 1945, Bush reeditó y renovó sus ideas en un articulo titulado “As We May Think”, publicado en *Atlantic Monthly*. Fue éste ensayo el que inspiró a un joven de nombre Douglas Englebart.

Douglas Englebart era un ingeniero eléctrico del Stanford Research Institute, que, habiendo trabajado como operador de radar durante la segunda guerra mundial, tuvo la visión, fuertemente inspirada por Bush, de crear un sistema de pantalla hecho por un tubo de rayos catódicos que presentara modelos de información gráficamente y permitiera aumentar el intelecto humano.

Esto significaba un enorme avance ideológico allá por 1962. Las únicas computadoras que existían eran mainframes gigantescos, que tenían como único medio de interacción una interface batch basada en tarjetas perforadas. Incluso la idea de una interface de texto donde el usuario ingresara comandos en tiempo real era considerada radical en aquel momento.

En 1968, Englebart realizó una demostración de sus ideas, materializadas en un sistema que denominó NLS (oN-Line System). El sistema contaba con tres dispositivos de interacción: un teclado estándar, un mouse de tres botones y un “teclado” de cinco teclas que se utilizaba para realizar combinaciones.

El sistema utilizaba como un puntero de mouse, una flecha recta apuntando hacia arriba, en lugar de la flecha en diagonal que conocemos y utilizamos hoy en día. Durante la presentación se observaron numerosos avances, como hipervínculos, edición de documentos en pantalla completa, ayuda contextual, etc. Sin embargo, había limitaciones técnicas en el sistema gráfico que dificultaban la comprensión. Por ejemplo, el sistema NLS soportaba múltiples ventanas, pero no tenía divisiones discernibles, como bordes, barra de título, etc.

El trabajo de Englebart fue continuado por el instituto de investigación de Xerox, PARC (Palo Alto Research Center). El primer resultado de su investigación fue la computadora experimental Xerox Alto, que utilizaba una pantalla con orientación vertical, como una hoja de papel, y por primera vez, incorporaba el puntero diagonal del mouse, tal como lo conocemos hoy en día. Sin embargo, la primera versión comercialmente disponible de una interface gráfica de usuario fue la Xerox Star 8010, en 1981, una versión reducida de la Xerox Alto.

El equipo de Xerox PARC fue el pionero en la implementación del paradigma WIMP, por sus siglas en inglés, *Windows, Icons, Menu, Pointing device*, que denota el estilo de interacción utilizando éstos elementos.

Otro de los pioneros de las interfaces gráficas de usuario fue Steve Jobs, en la recién constituida Apple Computer, que, tomando muchas de las ideas planteadas por la Xerox PARC, construyó una computadora personal con una interface de usuario muy similar a las que conocemos en la actualidad: la LISA, que incorporaba los conceptos de drag & drop, la papelera de reciclaje, y la idea de marcar en gris las opciones deshabilitadas de un menú.

A ésta altura, las GUI ya estaban disponibles comercialmente al público masivo, especialmente desde la introducción de la Apple Macintosh en 1984. Junto con Apple, otras compañías estaban desarrollando entornos de interacción gráfica: VisiCorp con el sistema ViSion, Microsoft con Windows 1.0 y Commodore Amiga con el entorno Amiga Workbench.

Para finales de 1987, Microsoft lanzó Windows 2.0, que ya incorporaba una interface bastante más familiar, sobre todo, el modelo de ventanas superpuestas, en contraste con el modelo de ventanas en mosaico de años anteriores.

En 1988, Steve Jobs, en su primer gran proyecto después de abandonar Apple, lanzó el sistema operativo NeXTSTEP, que contaba con un sistema GUI con un estilo de relieve en todos los componentes de la interface, e incluía por primera vez, el símbolo de la “X” para cerrar ventanas.

Justo antes del final de la década de 1980, AT&T, Sun, DEC y HP comenzaron a incluir interfaces gráficas de usuario basadas en la plataforma *X Window System*. Esta arquitectura sería la base de las interfaces gráficas en Linux.

En la década del 90, las interfaces gráficas comenzaron un proceso de evolución y homogeneización a partir de la amplia difusión y aceptación de la interface gráfica de Windows, sobre todo desde 1995, cuando Windows 95 introdujo el concepto del menú de Inicio. En el mundo de Apple, Mac OS X introdujo un sistema de GUI denominado Aqua, que representó una evolución en términos técnicos (double-buffered windows en memoria, etc.) y estéticos.

Junto a la evolución de las interfaces gráficas en la plataforma de escritorio, la década del 90 fue testigo de la violenta incursión de Internet en el mercado comercial y con ello, la introducción de la plataforma Web como un nuevo actor en el paisaje de las interfaces gráficas. Esta plantea nuevos desafíos técnicos y tecnológicos que deberán ser resueltos en los años por venir.

El futuro promete nuevas formas y tecnologías de interacción con los sistemas informáticos: uso masivo de 3D en la capa de presentación, zooming interfaces, interfaces inmersivas, realidad virtual, y muchas otras experiencias que aún no somos capaces de imaginar.

c. ARQUITECTURAS DE INTERFACES GRÁFICAS

A medida que las interfaces gráficas han ido evolucionado, se han diseñado diferentes arquitecturas para organizar la capa de presentación en una aplicación.

El siguiente es un recuento de las arquitecturas más notables que se han presentado a la largo de la historia.

Formularios y controles

Esta es una de las arquitecturas más simples y familiares. En realidad, no tiene un nombre definido, por lo cual, utilizaremos la denominación de Martin Fowler (2006). Este tipo de arquitecturas se popularizó durante la década del 90 con herramientas como Visual Basic, Delphi y PowerBuilder.

Esta arquitectura, tal como lo indica su nombre, está formada por dos elementos bien diferenciados: un formulario y una serie de controles. Los formularios son específicos de la aplicación que diseñamos, sin embargo, los controles son genéricos. A pesar de que en la mayoría de los entornos de éste tipo es posible crear nuevos controles, éstos, en general, son construidos para ser reusables en otros proyectos.

Los formularios contienen dos responsabilidades:

- La disposición de la pantalla: Definir el ordenamiento de los controles en la pantalla.
- La lógica del formulario: Ya que el comportamiento no puede ser incorporado fácilmente dentro de los controles, la lógica se encuentra contenida en el formulario.

En la mayoría de las situaciones existen tres copias diferentes de los cada uno de los datos presentados por un formulario:

- Una copia está en la base de datos propiamente dicha. Es lo que se denomina **Record State**.
- Otra copia se encuentra en memoria, en algún formato del tipo RecordSet, dentro de la aplicación. Nos referimos a ésta con el nombre de **Session State**.
- Por último, existe una copia del dato en el control propiamente dicho, que se denomina **Screen State**.

Una de las tareas más importantes de las interfaces es mantener sincronizados el *Session state* y el *Screen state*. Una herramienta que facilita esto es el *Data Binding*. La idea es que cualquier cambio, ya sea en los datos de un control o en el RecordSet se propagara al otro. Esto es más complicado de lo que parece, ya que es necesario evitar un ciclo infinito de actualización entre el control y el RecordSet.

Sin embargo, el *Data Binding* no soluciona todas las necesidades. Para los casos donde es necesario mayor control, el formulario debe ser notificado cuando un dato es modificado en alguno de los controles. La forma más tradicional de lograr éste efecto es a través de **eventos**. Esencialmente, es una versión del patrón de diseño **Observer**, donde el formulario está observando al control.

Model View Controller (MVC)

Es probablemente el patrón de organización de UI más mencionado de todos. Los orígenes de MVC están atados a los de Smalltalk-80. En el corazón de MVC se sitúa una de las ideas más influyentes en frameworks posteriores, *Presentación Separada*. La idea detrás de éste patrón es trazar una clara división entre los objetos del dominio que modelan nuestra percepción del mundo real, y los objetos de presentación que son los elementos GUI que vemos en pantalla. Los objetos del dominio deben ser completamente auto-contenidos y funcionar sin referencias a la capa de presentación.

En MVC, los objetos del dominio son referidos como el *Modelo*. La porción de presentación de MVC son los dos elementos restantes: *View* y *Controller*. La tarea del controlador es tomar el input del usuario y decidir que hacer con éste.

Es importante notar que no existe una única vista y un único controlador por pantalla, sino que existe un par vista-controlador por cada elemento de la pantalla, cada uno de los controles y la pantalla como un todo.

Por lo tanto, no hay un objeto general observando los controles, como en el patrón anterior, sino que los controles observan al modelo. Una de las consecuencias de éste patrón es que el controlador puede permanecer ajeno a los cambios que otros controles necesitan. Sin embargo, la desventaja que presenta es que éste tipo de patrón de *observación*, hace muy complicado entender el código a partir de una lectura del mismo, y en general, solo es discernible desde una sesión de *debug*.

VisualWorks Application Model

Smalltalk fue producido originalmente por el centro de investigaciones de Palo Alto de Xerox (PARC). Pero, más adelante, se separó para formar una nueva compañía, ParcPlace, que desarrolló VisualWorks, una versión de Smalltalk que funcionaba en diferentes plataformas.

VisualWorks tenía un concepto similar al MVC, sin embargo, introdujo una construcción denominada *Application Model*. Un elemento clave en éste modelo fue la idea de convertir propiedades en objetos. En la notación común de objetos, es común pensar en un objeto `Persona`, que contiene una propiedad `Nombre`. El modelo de VisualWorks incorpora *Property Objects*, por lo tanto, para acceder a nombre de usa persona, utilizaríamos la siguiente sintaxis:

Ejemplo:

```
person.Name.Value = "Jose"
```

Este tipo de objetos hacen el mapeo entre los controles y el modelo un poco más simple.

La diferencia fundamental entre el MVC clásico y el modelo de VisualWorks es que ahora tenemos una clase intermedia entre el modelo y el control, esta es la clase del *Application Model*.

Model View Presenter (MVP)

MVP es una arquitectura que apareció por primera vez en IBM y ganó visibilidad en Taligent en los 90s. MVP significa un paso hacia la unificación de dos líneas de pensamiento completamente diferentes en cuanto a arquitecturas GUI: por un lado, la arquitectura Formularios y Controles, que provee un diseño simple y fácil de comprender , y brinda una buena separación entre código específico de la aplicación y componentes reusables; y por el otro MVC y sus derivativos, que tiene como características principales la fuerte separación entre la capa de presentación y la del dominio de negocios.

El primer elemento de éste modelo es tratar a la vista como una estructura de controles, sin embargo, no contiene ninguna pista sobre como reaccionar a la interacción del usuario.

La reacción activa vive en un objeto de presentación diferente, el *Presenter*. Los manejadores de las acciones principales del usuario aún existen en los controles, pero éstos simplemente delegan el mando al *Presenter*, quien decide como reaccionar ante un evento.

Como el *Presenter* actualiza el modelo, la vista es actualizada con el mismo patrón *Observer* que en MVC.

Para aclarar el panorama, contrastamos el enfoque MVP contra las arquitecturas mencionadas anteriormente:

- *Formularios y controles:* MVP tiene un modelo que el *Presenter* manipula y actualiza. La vista es actualizada con el patrón *Observer*. Se permite el acceso directo a los controles desde el *Presenter*, pero no como primera opción.
- *MVC:* Los controles no están separados en *View* y *Controller*. El *Presenter* actúa a nivel de formulario, ésta es una de las diferencias más notables con la arquitectura MVC.
- *VisualWorks Application Model*: El *Presenter* no actúa igual que el *Application Model*. Si bien existen similitudes entre los dos enfoques, el *Presenter* tiene acceso directo a los controles para comportamientos que no pueden modelarse fácilmente con el patrón *Observer*.

Humble View

En los últimos años, ha habido una fuerte tendencia a la construcción de código testeable, sobre todo con la notoriedad adquirida por los denominados xUnit Frameworks, que facilitan el Unit Testing de código, y la popularización del paradigma TDD (Test-Driven Development) o desarrollo conducido por tests.

Pero cuando es preciso testear las interfaces de usuario, rápidamente surgen los problemas. Esto es debido a que las UI están fuertemente atadas al entorno gráfico y es muy difícil dividirlas para testearlas individualmente.

Como resultado, existe una predilección a diseñar interfaces de usuario de tal forma que se minimice el comportamiento en los objetos que son difícilmente testeables. Este movimiento se denomina *Humble View*, o *Vista Humilde*, y consiste en crear vistas carentes de comportamiento propio de forma tal que minimizamos las posibilidades errores indetectables.

Este enfoque utiliza un *Presenter*, pero de una forma mucho más intensiva que el modelo MVP tradicional. No solo decide como reaccionar a la interacción con el usuario, sino que también controla la población inicial de datos en los controles. Como resultado, los controles ya no tienen, ni necesitan, visibilidad con el modelo; conforman una vista pasiva que es manipulada por el *Presenter*.

d. NECESIDAD DE UN LENGUAJE UNIFICADO

Habiendo expuesto el estado actual del desarrollo en cuanto a interfaces gráficas se refiere, es que se plantea la necesidad de un nuevo componente: un lenguaje de definición de interfaces que pueda aplicarse a cualquier plataforma gráfica, tanto Web como Desktop, y, pueda ser extendido en el futuro a nuevas plataforma a medida que emerjan.

Durante el desarrollo del presente proyecto, se especifica un lenguaje expresado en un vocabulario XML, orientado a la definición de interfaces gráficas de usuario, que soporte tanto

las plataformas Web como Desktop, y genere un mecanismo común y simplificado de construcción de GUIs totalmente abstraídas de los aspectos técnicos de la codificación.

Se promueve así la separación de responsabilidades, devolviendo la tarea de diseño de interfaces al ambiente del diseño gráfico, en lugar del entorno de desarrollo de software. Así mismo, da lugar a la creación de herramientas de diseño que emitan código formateado en éste lenguaje, acortando así los tiempos de construcción.

e. DESARROLLOS EXISTENTES

Existen en la actualidad distintos tipos de soluciones y desarrollos existentes en el mercado de las interfaces gráficas de usuarios. Cada uno resuelve en mayor o menor medida el problema general de la representación gráfica. Sin embargo, casi sin excepciones, todos apuntan a una plataforma especifica y no a una homogeneización del desarrollo de interfaces.

BARRACUDA FRAMEWORK

Barracuda es un framework de capa de presentación orientada al desarrollo Web, que abstrae al programador de la implementación tecnológica. La versión existente es únicamente aplicable a la plataforma de desarrollo J2EE, y su núcleo funcional está compuesto por una herramienta denominada XMLC, que compila código XML, HTML y XHTML en una clase Java que puede ser accedida y alterada programáticamente.

MOZILLA APPLICATION FRAMEWORK

Si bien no apunta exactamente a la misma problemática, el Mozilla Application Framework contiene el *Gecko Rendering Engine*, basado en XUL, que es utilizado para abstraer a los desarrolladores de Mozilla de las librerías gráficas propias de cada plataforma y sistema operativo.

UIML

UIML, por la sigla ***U**ser **I**nterface **M**arkup **L**anguage*, es una aplicación de XML para describir interfaces de usuario. El objetivo es crear un estándar abierto para descripción de interfaces en XML que pueda ser implementado libremente por cualquier persona. Hoy en día, UIML está siendo estandarizado por OASIS, sin embargo, parece no tener suficiente apoyo de la comunidad, y por lo tanto, está un poco estancado su desarrollo. UIML apunta a unificar la descripción de todos los tipos posibles de interfaces, no restringiéndose a interfaces visuales únicamente.

XAML

Pronuciado “Zammel”, XAML es el lenguaje de definición de interfaces de usuarios para la próxima versión de Microsoft .NET Framework. Forma parte de la infraestructura gráfica denominada *Windows Presentation Foundation*. Es un lenguaje declarativo basado en XML, optimizado para describir interfaces visuales ricas. XAML fue creado por Microsoft® y su sigla viene de (e***X**tensible **A**pplication **M**arkup **L**anguage*). XAML promete ser una buena plataforma de desarrollo de interfaces gráficas, sin embargo, se encuentra aún en un estado muy prematuro de desarrollo.

Aún no es claro cual será la arquitectura final de *Windows Presentation Foundation*, sin embargo, hasta el momento, está previsto que las aplicaciones de éste tipo requieran la presencia del .NET Framework 3.0 instalado en la máquina cliente, lo que compromete seriamente el objetivo de las aplicaciones Web, que pueden correr en cualquier PC que tenga instalado un navegador. Existe sin embargo, una iniciativa denominada *WPF/E* (*Windows*

Presentation Foundation/Everywhere) que apunta a crear plug-ins para distintas plataformas, incluida la Web, que permitan la ejecución de éste tipo de aplicaciones.

XUL

XUL (acrónimo de **X**ML-based **U**ser-interface **L**anguage, lenguaje basado en XML para la interfaz de usuario) es la aplicación de XML a la descripción de la interfaz de usuario en el navegador *Mozilla*.

La principal ventaja de XUL es que aporta una definición de interfaces GUI simple y portable. Esto reduce el esfuerzo empleado en el desarrollo de software.

Otras aplicaciones aparte de Mozilla usan este lenguaje para sus interfaces de usuario. Algunas de ellas usan JavaScript para su lógica. Las aplicaciones XUL tienen la ventaja de poder correr en distintos sistemas operativos.

GESTIÓN DE PROYECTO

Habiendo definido los objetivos, límite y alcance del proyecto, se desarrolló un plan de gestión de proyecto describiendo la forma en que se administra el avance del proyecto y estipula un cronograma de actividades a llevar a cabo.

ESTRUCTURA DE DIVISIÓN DEL TRABAJO

Para poder gestionar los tiempos adecuada y eficientemente, se dividió el grueso de trabajo necesario para la consecución de los objetivos del proyecto, en tareas más pequeñas y manejables. Éstas conforman la base del cronograma de avance y control del proyecto presentado en las páginas siguientes.

- Etapa inicial
  - Preparación del documento de proyecto de TFG
  - Aprobación del documento de proyecto de TFG
- Etapa de desarrollo del TFG
  - Formulación del plan de proyecto
  - Definición del GANNT de proyecto
  - Definición de las estrategias globales
  - Recolección de requerimientos / Análisis
    - Requerimientos de persistencia y acceso a datos
    - Requerimientos relacionados a las interfaces de usuarios
    - Requerimientos derivados de los objetivos de modelos de negocios auto-validados
    - Desarrollo y modelado de requerimientos a través de casos de uso UML
    - Construcción de la matriz de trazabilidad (Requerimientos-Casos de Uso)
  - Diseño
    - Diseño arquitectónico
    - Diseño estático – Diagramas UML estructurales
      - Desarrollo del diseño estático de persistencia
      - Desarrollo del diseño estático de los componentes de abstracción de interfaces de usuario
      - Desarrollo del diseño estático de modelos de negocios auto-validados
    - Diseño dinámico – Diagramas UML de comportamiento
      - Desarrollo del diseño dinámico de persistencia
      - Desarrollo del diseño dinámico de los componentes de abstracción de interfaces de usuario
      - Desarrollo del diseño dinámico de modelos de negocios auto-validados
    - Construcción de la matriz de trazabilidad (Casos de Uso-Componentes)
  - Codificación
    - Construcción de los componentes de persistencia
    - Construcción de los componentes de abstracción de interfaces de usuario
    - Construcción de los componentes de modelos de negocios auto-validados
  - Evaluación
    - Pruebas del subsistema de persistencia
    - Pruebas del subsistema de interfaces de usuario
    - Pruebas del subsistema de modelos de negocios auto-validados
    - Pruebas de integración de componentes
  - Integración de la documentación en el Documento de TFG
  - Desarrollo de la presentación del proyecto
- Etapa de Examen Oral
  - Presentación oral del proyecto finalizado

ENTREGABLES

Durante la vida del proyecto, el desarrollo del mismo se generan una serie de elementos o "productos", que denotan el cumplimiento y finalización de una o más etapas del proceso.

Los elementos entregables identificados, agrupados según las distintas etapas, son los siguientes:

- *Análisis*
  - Especificación de Requerimientos de Software
  - Modelo de Casos de Uso
  - Matriz de Trazabilidad (Requerimientos-Casos de Uso)
- *Diseño*
  - Diagramas estructurales UML
    - Diagramas de clases
    - Diagramas de componentes
    - Diagramas de paquetes
  - Diagramas de comportamiento
    - Diagramas de secuencia
    - Diagramas de actividad
    - Diagramas de estado
  - Matriz de Trazabilidad (Casos de Uso-Componentes)
- *Construcción*
  - Código fuente del proyecto
  - Ejecutables y librerías compiladas generadas
  - Instaladores del proyecto

MODELO DE GESTIÓN DE PROYECTO

Para controlar los riesgos del proyecto, el mismo se desarrolló siguiendo un modelo iterativo e incremental conocido como Modelo Espiral, que cuenta con 4 fases (Definición de objetivos, Análisis de riesgos, Ingeniería, y Evaluación) que se repiten en un número de iteraciones hasta alcanzar los objetivos definidos.

Se seleccionó el Modelo Espiral porque presenta una metodología formal para minimizar los riesgos, y cuenta a su vez con los beneficios de los modelos iterativos que permiten desarrollar un software incrementalmente, dándole a los desarrolladores la oportunidad de tomar ventaja de las lecciones aprendidas en etapas anteriores del proyecto.

Se planteó para éste proyecto la ejecución de 3 iteraciones del Modelo Espiral, atacando respectivamente los objetivos relacionados a Persistencia, Interfaces de Usuario y Modelos de negocios auto-validados en cada una de ellas.

Para la representación y modelado del proyecto se utilizó el lenguaje de modelado unificado UML 2.0, para esto, se emplearon extensivamente las capacidades de la herramienta CASE / UML *Enterprise Architect* de Sparx Systems© (http://www.sparxsystems.com.au).

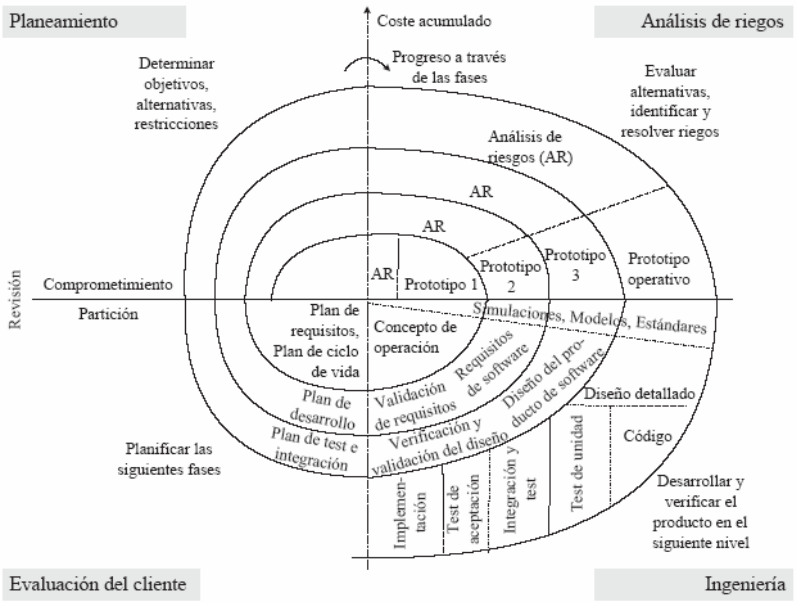


Fig 1. Modelo de desarrollo en espiral (BORGES DE BARROS PEREIRA, H., 2002)

CRONOGRAMA

Para alcanzar los objetivos del proyecto debe planificarse correctamente en el tiempo los pasos que deben cumplirse. A continuación se detalla la distribución temporal de las tareas, lo que conforma el cronograma del proyecto. La misma se deriva de la estructura de división del trabajo (Ver *Estructura de división del trabajo*) y del ciclo de vida seleccionado para el proyecto.

El cronograma está dividido en las tres etapas definidas por el reglamento de Seminario Final, y dentro de la etapa de desarrollo, está reflejado el ciclo de vida seleccionado, en tres iteraciones distintas, cada una con cuatro fases.

En términos generales, el cronograma de proyecto se descompone de la siguiente forma:

- Etapa inicial
- Etapa de desarrollo del TFG
  - Definiciones globales
  - Primera iteración: Abstracción de la persistencia y el acceso a datos
    - Primera fase: Objetivos
    - Segunda fase: Análisis de riesgos
    - Tercera fase: Ingeniería
    - Cuarta fase: Evaluación
  - Segunda iteración: Abstracción de las interfaces de usuario
    - Primera fase: Objetivos
    - Segunda fase: Análisis de riesgos
    - Tercera fase: Ingeniería
    - Cuarta fase: Evaluación
  - Tercera iteración: Modelos de negocios auto-validados
    - Primera fase: Objetivos
    - Segunda fase: Análisis de riesgos
    - Tercera fase: Ingeniería
    - Cuarta fase: Evaluación
  - Entrega final del Documento de TFG
- Etapa de Examen Oral

Fig 2. Diagrama GANNT de proyecto

ENFOQUE TÉCNICO

El siguiente proyecto se basa en la construcción de un framework que dé soporte al desarrollo de sistemas de negocios orientados a objetos. Para el desarrollo del mismo, se utilizó la plataforma .NET Framework 2.0 provista por Microsoft©, siguiendo el paradigma de programación orientada a objetos. Se aplicaron en casos puntuales, algunos conceptos del paradigma de orientación a aspectos, que se conjuga con el de objetos para brindar una mayor y mejor separación de tareas entre los componentes.

Durante el desarrollo se utilizó el meta-lenguaje XML para la definición de vocabularios específicos al dominio del problema.

Para las etapas de ingeniería se hizo uso extensivo de herramientas CASE / UML para el modelado de los componentes.

GESTIÓN DE LA CONFIGURACIÓN DE SOFTWARE

La gestión de la configuración de software (SCM por sus siglas en inglés) es una disciplina para el control de la evolución de sistemas de software. Si bien el tamaño del presente proyecto no es significativamente grande, se decidió emplear ésta técnica por dos motivos:

1. Ejercitar y experimentar la aplicación de ésta práctica ampliamente extendida en la industria del desarrollo de software profesional
2. Formalizar las instancias de cambio y permitir un seguimiento claro de la evolución del proyecto.

En las siguientes secciones se detallan las tareas de SCM desarrolladas para el presente proyecto.

CONFIGURACIÓN AMBIENTAL

a. HERRAMIENTAS DE SOFTWARE

Se utilizó el sistema de SCM provisto por *Perforce* (http://www.perforce.com), debido a que ofrece las características necesarias para el nivel de SCM del proyecto, y además, utiliza un modelo de licenciamiento que permite el uso ilimitado del mismo hasta un máximo de dos usuarios. El sistema de SCM de *Perforce* está compuesto por dos herramientas: *Perforce P4 Server* y *Perforce P4 Cliente*.

b. RECURSOS DE HARDWARE

Los siguientes recursos de hardware fueron afectados la tareas de SCM durante la vida del proyecto:

- Un equipo servidor para la ejecución del sistema *Perforce P4 Server*, que contiene una copia completa y actualizada de los ítems de configuración identificados, junto con el historial de modificación de los mismos.
- Un equipo de desarrollo con la aplicación *Perforce P4 Cliente*.

IDENTIFICACIÓN

a. IDENTIFICACIÓN DE LOS ÍTEMS DE CONFIGURACIÓN

Los siguientes tipos de ítems fueron puestos bajo control de configuración de software:

- Documentación
  - Documento de TFG
  - Documentación normativa de la materia TFG
  - Documentación de diseño generada por herramientas CASE / UML
  - Documentación bibliográfica disponible en medios electrónicos
  - Manual de usuario / Referencia para el programador
- Fuentes
  - Archivos de código fuente del Framework
  - Archivos de configuración por defecto
  - Archivos de código fuente de los bancos de prueba del Framework
- Productos de software
  - Productos de la compilación del Framework
  - Instaladores del Framework

La localización de los mismos en la herramienta de SCM es la siguiente:

- **Raíz del proyecto**
  - *Documentación*
    - Diseño
    - Información adicional
      - Bibliografía
    - Normativa
    - Reportes
    - Manuales
  - *Fuentes*
    - Indy
    - Referencias
    - TestApplication
  - *Instaladores*

b. LÍNEAS BASE

Para el presente proyecto, se utilizaron líneas bases (*Baselines*) por etapa terminada. Por lo tanto, existe una línea base para la documentación al momento de finalizar la etapa inicial de cursado de la materia Seminario Final, una línea base al finalizar cada una de las iteraciones, y una última línea base al finalizar la etapa de desarrollo del TFG. La etapa final, de examinación oral no genera entregables, por lo tanto, no se define ninguna línea base al concluir la misma.

La siguiente es la nomenclatura y contenido de las líneas bases definidas:

- **TFG-Etapa.inicial**
  - Esta línea base se define en el momento de la aprobación de la materia Seminario Final.
  - Incluye el documento de proyecto de TFG, así como la documentación bibliográfica inicial y la documentación normativa del TFG.
- **TFG-Primera.iteración**
  - Esta línea base se define al finalizar la primera iteración de la etapa de desarrollo.
  - Incluye toda la documentación y código fuente generada hasta el final de la primera iteración, así como la documentación bibliográfica recolectada hasta este punto.
- **TFG-Segunda.iteración**
  - Esta línea base se define al finalizar la segunda iteración de la etapa de desarrollo.
  - Al igual que el anterior, incluye toda la documentación y código fuente generados hasta el final de la segunda iteración.
- **TFG-Tercera.iteración**

  - Esta línea base se define al finalizar la tercera iteración de la etapa de desarrollo.
  - Siguiendo el patrón establecido, ésta línea base incluye toda la documentación y código fuente generados hasta el final de la tercera iteración.
- **TFG-Final**
  - La última de las líneas bases es definida al darse por concluida la etapa de desarrollo del proyecto, una vez finalizados los coloquios.
  - Incluye toda la documentación, código fuente y productos de compilación generados por el proyecto, así como toda la información relevante la finalización del desarrollo.

CONTROL DE CAMBIOS

Los pedidos de cambio recibidos durante el desarrollo del proyecto son administrados y aplicados mediante las utilidades que provee la herramienta de SCM para éste fin. Por lo tanto, se genera un registro por cada pedido de cambio, y cada check-in en el repositorio deberá estar acompañado por una referencia a éste registro, con lo cual se logra la identificación del origen de todos los cambios ingresados en el repositorio.

Debido a la naturaleza del presente proyecto, no se establecen políticas de *branching*, ya que el mismo no será requerido. Por ende, tampoco existirán situaciones de *merge*.

GESTIÓN DE RIESGOS

La gestión de riesgos ayuda a minimizar y contener el impacto de los riesgos del proyecto, y proporciona lineamientos de acción contingente en el caso de la ocurrencia de los mismos. A su vez, una correcta gestión de riesgos reduce los niveles de incertidumbre del proyecto.

FACTORES DE RIESGO	TIPO	PROB. DE OCURRENCIA	NIVEL DE IMPACTO	ACCIONES
Errores en las estimaciones	Proyecto	Media	Crítica	*Estrategia de mitigación:* Debido a la poca experiencia en estimaciones, las mismas pueden no ajustarse totalmente a la realidad, lo que extendería los plazos del proyecto. Para mitigar éste riesgo, se desarrollará un cronograma de actividades detallado, y se controlará periódicamente el cumplimiento del mismo para detectar desviaciones a tiempo.
Inviabilidad técnica	Técnico	Baja	Catastrófica	*Estrategia de aceptación:* En cada iteración se evaluarán las opciones técnicas disponibles para solucionar cada uno de los problemas que se planteen.
Pérdida de información	Proyecto	Baja	Catastrófica	*Estrategia de mitigación:* Para evitar la pérdida de información durante el desarrollo del proyecto, ya sea por error o por accidente, se utilizará una herramienta de SCM, que a su vez, mantendrá una copia del historial de modificación de cada ítem de configuración identificado.
Complejidad demasiado elevada	Técnico	Baja	Crítica	*Estrategia de mitigación:* Para mantener en niveles aceptables la complejidad del desarrollo, se diseñará una arquitectura en capas, en donde la

				complejidad relativa de cada una sea baja. Así mismo, se utilizará un lenguaje de alto nivel para reducir la complejidad inherente del lenguaje.
Performance inaceptable	Técnico	Baja	Crítica	*Estrategia de mitigación:* La performance del producto final es crítica para el éxito del proyecto. Para asegurar una performance adecuada se realizarán pruebas de performance en la fase de evaluación de cada iteración del proyecto.
Código poco comprensible y baja mantenibilidad	Técnico	Baja	Marginal	*Estrategia de mitigación:* Se desarrollará un estándar de codificación para éste proyecto, con lo cual se intenta homogeneizar el proceso de construcción. Adicionalmente, se utilizarán patrones de diseño comunes, para hacer el código más comprensible.
El lenguaje seleccionado no provee las capacidades requeridas por el desarrollo	Técnico	Baja	Despreciable	-

DESARROLLO DE PROYECTO

La ejecución del proyecto está estructurada en cuatro grandes áreas: *Análisis*, *Diseño*, *Construcción*, y *Evaluación*. Las primeras dos, de naturaleza estrictamente documental, y las últimas dos de naturaleza constructiva.

En las siguientes secciones de éste documento están compilados los resultados de la ejecución de las etapas de *Análisis* y *Diseño*. De acuerdo a lo estipulado en la sección Entregables del plan de proyecto, los resultados de cada etapa están distribuidos de la siguiente forma:

- *Análisis*
  - Especificación de Requerimientos de Software
  - Modelo de Casos de Uso
  - Matriz de Trazabilidad (Requerimientos-Casos de Uso)
- *Diseño*
  - Diagramas estructurales UML
    - Diagramas de clases
    - Diagramas de componentes
    - Diagramas de paquetes
  - Diagramas de comportamiento
    - Diagramas de secuencia
    - Diagramas de actividad
    - Diagramas de estado
  - Matriz de Trazabilidad (Casos de Uso-Componentes)

ESPECIFICACIÓN DE REQUERIMIENTOS DE SOFTWARE

Los requerimientos detectados para el desarrollo del proyecto se encuentran divididos en categorías. En primer lugar, los requerimientos funcionales, representan las reglas básicas del dominio del problema y son un claro indicador de los objetivos que debe alcanzar el desarrollo en términos de funcionalidad ofrecida. Por otro lado, los requerimientos no funcionales describen criterios de performance, seguridad, y otros parámetros operacionales, son en general consideraciones o restricciones asociadas a un servicio del sistema. Cada una de éstas categorías ha sido subdividida en otras más específicas para gestionar la complejidad de cada una de manera individual.

El modelo de requerimientos que se presenta en las próximas secciones es uno de los entregables definidos en el plan de proyecto.

REQUERIMIENTOS FUNCIONALES

a. REQUERIMIENTOS DE INTERFACES DE USUARIO

NOMBRE	DETALLES	NOTAS
RQ01 - Abstraer la tecnología de presentación.	**Dificultad:** Alta	La tecnología de presentación debe permanecer totalmente transparente al cliente del framework, abstrayendo las diferencias de implementación entre las plataformas Web y Desktop, en términos de interfaces de usuario.
RQ02 - Permitir la definición de interfaces de usuario a través de un lenguaje declarativo de tipo XML.	**Dificultad:** Alta	El framework debe permitir definir interfaces gráficas de usuario en un lenguaje declarativo XML que no esté atado a una plataforma determinada, sino que sea genérico para distintas tecnologías de presentación.

NOMBRE	DETALLES	NOTAS
RQ03 - Permitir la construcción de interfaces basadas en el concepto de anidación de componentes.	**Dificultad:** Media	El lenguaje de definición de interfaces gráficas debe permitir crear estructuras de componentes anidados, donde cada uno puede contener cero, uno o más componentes que a su vez pueden estar formados por otros más.
RQ04 - Permitir la manipulación programática de los controles de interface de usuario.	**Dificultad:** Alta	Una vez definidas las interfaces de usuarios de forma declarativa, el framework debe proveer una API para control y manipulación programática de los mismos.
RQ05 - Separar la definición de estilo y de estructura de interfaces de usuario.	**Dificultad:** Baja	La definición de estilo y de estructura de las interfaces gráficas de usuario debe estar dividida en recursos (archivos) diferentes.

El siguiente diagrama describe las relaciones de agregación y dependencia entre los requerimientos funcionales de interfaces de usuario.

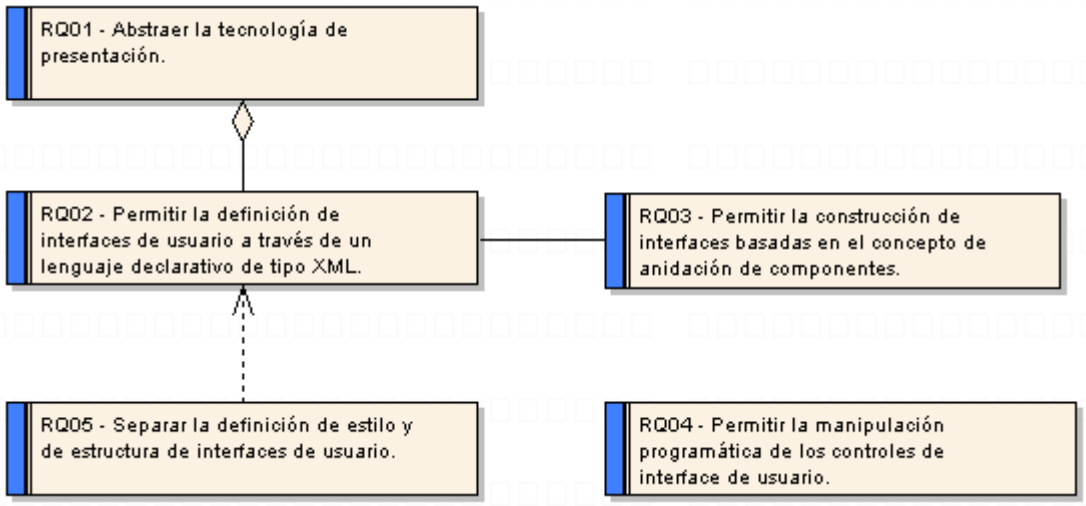


Fig 3. Diagrama de requerimientos funcionales de interfaces de usuario

b. REQUERIMIENTOS DE AUTO-VALIDACIÓN DE MODELOS

NOMBRE	DETALLES	NOTAS
RQ06 - Permitir la creación de modelos de negocio que realicen validaciones automáticamente.	**Dificultad:** Media	El framework debe facilitar la creación de objetos de negocio, parte integral de un modelo de negocio, que incluyan reglas de validación sin necesidad de codificación adicional.
RQ07 - Validaciones de longitud máxima de un atributo.	**Dificultad:** Baja	El framework debe permitir la definición de la longitud máxima permitida para un atributo determinado mediante los validadores predefinidos.
RQ08 - Validaciones de longitud mínima de un atributo.	**Dificultad:** Baja	El framework debe permitir la definición de la longitud mínima permitida para un atributo determinado mediante los validadores predefinidos.
RQ09 - Validaciones de rangos de valores válidos.	**Dificultad:** Baja	El framework debe permitir la definición un rango de valores permitidos para un atributo determinado mediante los validadores predefinidos.

NOMBRE	DETALLES	NOTAS
RQ10 - Validación de atributos requeridos.	**Dificultad:** Baja	El framework debe permitir cuales atributos son requeridos y cuales no lo son mediante los validadores predefinidos.
RQ11 - Máscaras de validación de un atributo.	**Dificultad:** Media	El framework debe permitir la definición máscaras de validación expresadas como expresiones regulares para un atributo determinado mediante los validadores predefinidos.
RQ12 - Permitir la creación de validadores personalizados.	**Dificultad:** Media	Debe proveerse una interface programática que permita la creación de validadores personalizados.

El siguiente diagrama describe las relaciones de agregación y dependencia entre los requerimientos funcionales de modelos auto-validados.

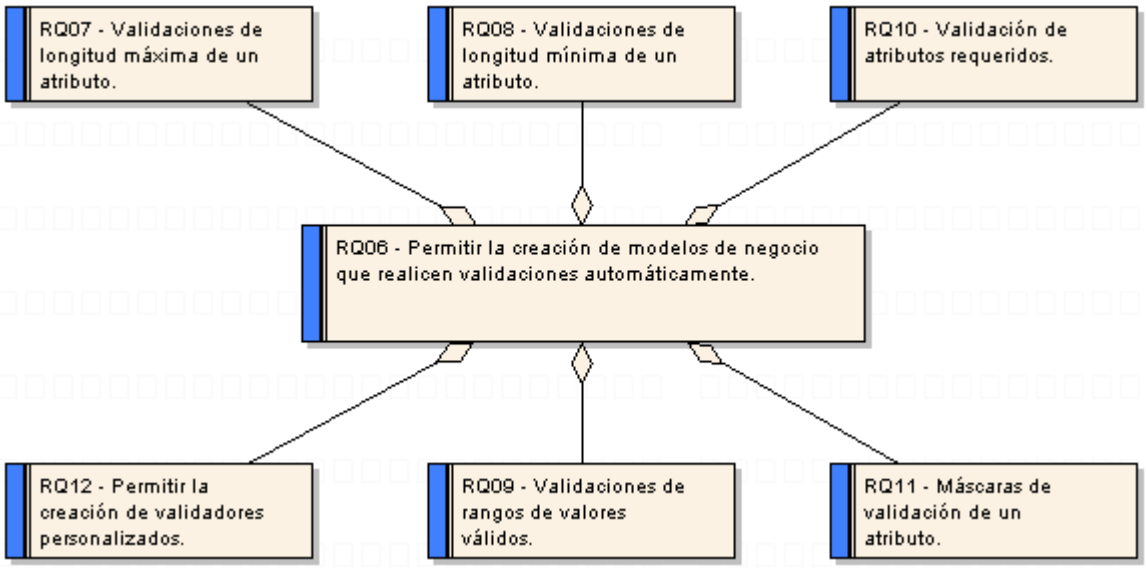


Fig 4. Diagrama de requerimientos funcionales de modelos auto-validados

c. REQUERIMIENTOS DE PERSISTENCIA

NOMBRE	DETALLES	NOTAS
RQ13 - El almacén de datos debe permanecer totalmente transparente para el cliente del framework.	**Dificultad:** Alta	El requerimiento principal de persistencia consiste en abstraer totalmente la existencia del almacén de datos y generar una interface de programación amigable y genérica para cualquier mecanismo de persistencia subyacente.
RQ14 - Generar consultas dinámicamente, eliminando la utilización del lenguaje propietario del almacén de datos.	**Dificultad:** Alta	Para cumplir con la abstracción del almacén de datos, debe abstraerse fundamentalmente el lenguaje de consulta (generalmente SQL) utilizado para extraer datos del mismo.
RQ15 - Generar consultas al almacén de datos con una sintaxis orientada a objetos fuertemente tipadas.	**Dificultad:** Alta	En el ambiente altamente cambiante de la actualidad, es fundamental que las consultas al almacén de datos sean fuertemente tipadas (type-safety), porque permite la detección rápida de errores y facilita el análisis y contención del impacto de un cambio. El framework debe proveer una

NOMBRE	DETALLES	NOTAS
		sintaxis orientada a objetos derivada del modelo de negocios que define el cliente del framework.
RQ16 - Generar consultas polimórficas.	**Dificultad:** Alta	Debido a la posibilidad de mapear estructuras de herencia (RQ24), las consultas deben tener capacidades polimórficas. Por ejemplo, si se mapean las clases Persona, Profesor y Alumno, una consulta de personas debería dar como resultado una lista de alumnos y profesores.
RQ17 - Mapear relaciones entre objetos.	**Dificultad:** Alta	Debe proveerse un mecanismo de mapeo de las relaciones entre objetos, para que puedan ser persistidas en el mecanismo de persistencia.
RQ18 - Mapear relaciones muchos-a-muchos.	**Dificultad:** Alta	El mecanismo de mapeo de relaciones debe permitir mapear relaciones del tipo muchos-a-muchos. Este tipo de relación no está soportada por los motores de datos relacionales, por lo que se utilizan tablas de asociación para la representación de las mismas.
RQ19 - Mapear relaciones 1-a-muchos.	**Dificultad:** Media	El mecanismo de mapeo de relaciones debe permitir mapear relaciones del tipo uno-a-muchos de una forma que sea semánticamente natural para el dialecto orientado a objetos.
RQ20 - Mapear relaciones 1-a-1.	**Dificultad:** Baja	El mecanismo de mapeo de relaciones debe permitir mapear relaciones del tipo uno-a-uno.
RQ21 - Mapear relaciones 1-a-muchos recursivas.	**Dificultad:** Alta	Un tipo especial de relación uno-a-muchos que debe soportar el Framework es la relación recursiva del tipo padre-hijo, que permite la creación de estructuras jerárquicas. Casi ningún motor de datos relacional permite generar consultas de relaciones recursivas jerarquizadas.
RQ22 - Mapear clases a un almacén de datos para que sus datos y estado sean persistentes en el tiempo.	**Dificultad:** Alta	Una de las funciones principales de persistencia consiste en mapear clases del paradigma de orientación a objetos al soporte provisto por el mecanismo de persistencia subyacente.
RQ23 - Mapear una clase a una tabla del almacén de datos.	**Dificultad:** Alta	El mecanismo primario de mapeo debe permitir mapear una clase a una tabla del almacén de datos.
RQ24 - Mapear estructuras de herencia a una estructura de datos.	**Dificultad:** Alta	Establecer un mecanismo de mapeo que permita mapear una estructura de herencia de clases a un mecanismo de persistencia. Los motores de base de datos relacionales no brindan soporte para el concepto de "herencia".
RQ25 - Permitir el mapeo horizontal de jerarquías de herencia.	**Dificultad:** Alta	Debe permitirse mapear cada clase concreta en una estructura de herencia a una tabla diferente del almacen de datos. (Mapeo de Tabla Múltiple o Mapeo Horizontal).
RQ26 - Permitir el mapeo filtrado de jerarquías de herencia.	**Dificultad:** Media	Debe permitirse mapear una estructura de herencia completa a una única tabla en el almacén de datos (Mapeo de Tabla Simple o Mapeo Filtrado).
RQ27 - Mapeo automático de clases.	**Dificultad:** Media	Proveer un mecanismo de mapeo automático de clases para proyectos simples, que utilice una convención de nombres bien definida para establecer la relación entre la clase y el almacén de datos.
RQ28 - Mapeo programático desde código.	**Dificultad:** Media	Proveer un mecanismo para el control programático del mapeo desde el código de la aplicación.

NOMBRE	DETALLES	NOTAS
RQ29 - Mapeo descriptivo en archivos de configuración.	**Dificultad:** Media	Proveer un mecanismo descriptivo para el definir mapeos en archivos o recursos de configuración.
RQ30 - Proveer control transaccional.	**Dificultad:** Alta	Debe proveerse un control de los límites transaccionales de una manera genérica, no orientada a la transacción del almacén de datos sino a la transacción de negocios.

El siguiente diagrama describe las relaciones de agregación y dependencia entre los requerimientos funcionales de persistencia.

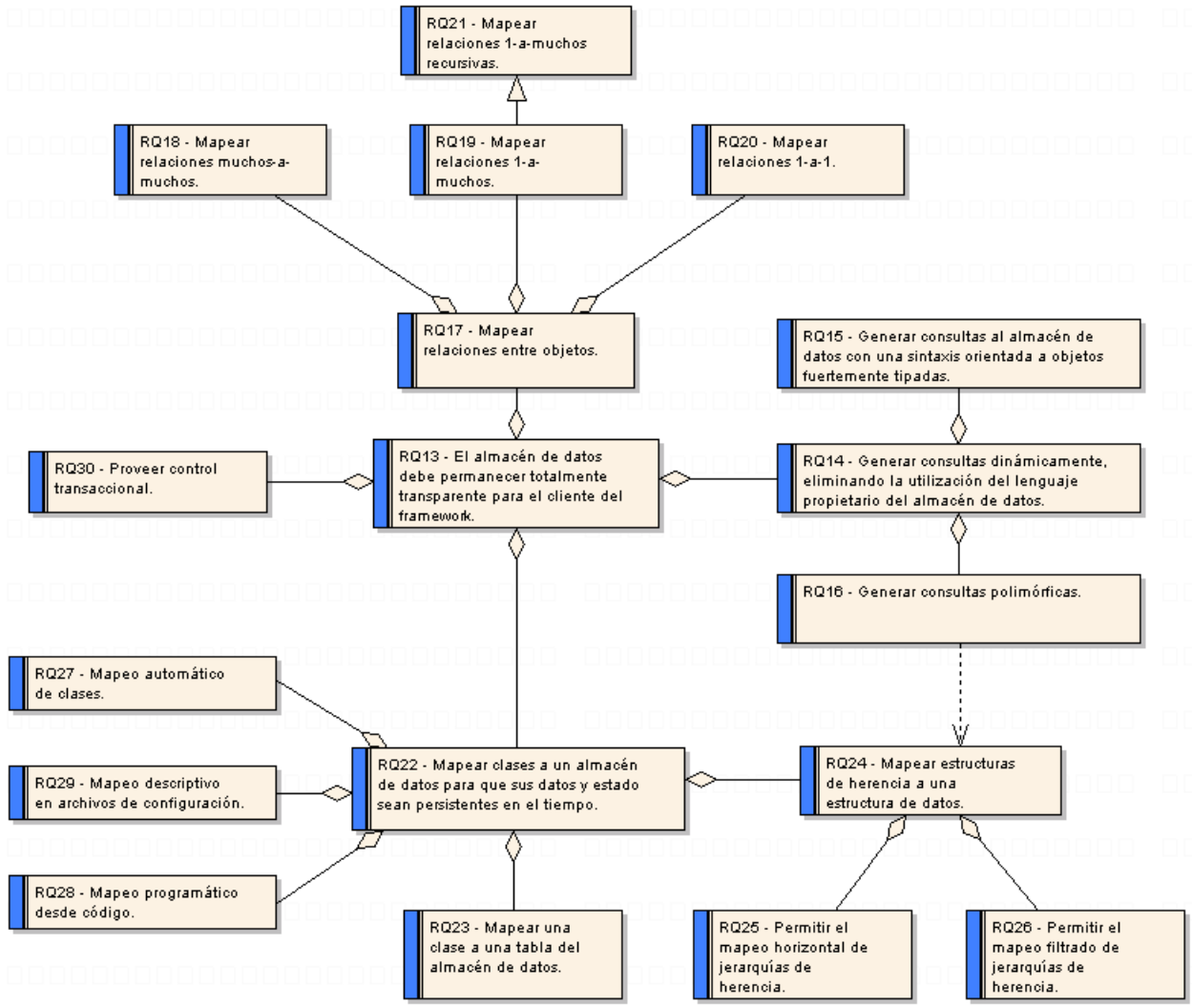


Fig 5. Diagrama de requerimientos funcionales de persistencia

REQUERIMIENTOS NO FUNCIONALES

d. REQUERIMIENTOS DE PERFORMANCE

NOMBRE	DETALLES	NOTAS
RQ31 - Implementar un subsistema de cache	**Dificultad:** Alta	El framework debe proveer un subsistema de cache para mantener niveles de performance aceptable.

NOMBRE	DETALLES	NOTAS
integrado con a las operaciones de persistencia		Entre los tipos de cache a implementar, deberá existir un cache de primer nivel en el contexto del hilo de ejecución, y un cache de nivel secundario en el contexto del dominio de la aplicación y opcionalmente, un cache distribuido.

MODELO DE CASOS DE USO

A partir de los requerimientos funcionales relevados, se desarrollaron y modelaron los casos de uso del sistema, que forman parte de los entregables definidos en el plan de proyecto.

El modelado de *Casos de Uso* es una de las técnicas más efectivas y a la vez más simples para modelar los requerimientos del sistema desde la perspectiva del usuario. Los casos de uso se utilizan para modelar la forma en que un sistema o negocio funciona. Su objetivo no es generar una aproximación a la orientación a objetos; sino una asistir en el modelado del negocio. Es, sin embargo, una muy buena manera de dirigirse hacia el análisis de sistemas orientado a objetos. Los casos de uso son generalmente el punto de partida del análisis orientado a objetos con UML.

El modelo de casos de uso consiste en actores y casos de uso. Los actores representan usuarios y / o sistemas que interaccionan con el sistema. En realidad, los actores constituyen tipos o roles de usuario, no una instancia concreta de usuario. Los casos de uso describen el comportamiento del sistema, los escenarios que el sistema atraviesa en respuesta a un estímulo desde un actor.

Las siguientes secciones describen en detalle los casos de uso encontrados, así como sus elementos dependientes. Cada caso de uso está documentado por una descripción del escenario diagramada en una tabla con un formato paso a paso.

ACTORES

Para el modelado de los casos de uso del sistema, se detectaron los siguientes actores:

ACTOR	DESCRIPCIÓN
Cliente de software	El cliente de software es el usuario directo del framework. Es cualquier pieza de software que utilice los servicios provistos por la API del Framework.
Desarrollador de software	Los desarrolladores de software que utilicen el Framework como soporte para sus aplicaciones, también son usuarios del mismo. Este tipo de actor incluye a todos los miembros de un equipo de desarrollo y codificación de cada proyecto con base en el Independence Framework.

CASOS DE USO DE OPERACIONES DE PERSISTENCIA

Los casos de uso de ésta sección abarcan las operaciones de consulta y actualización de datos a través de los mecanismos de persistencia provistos por el Framework. El siguiente es un diagrama general de la distribución de casos de uso.

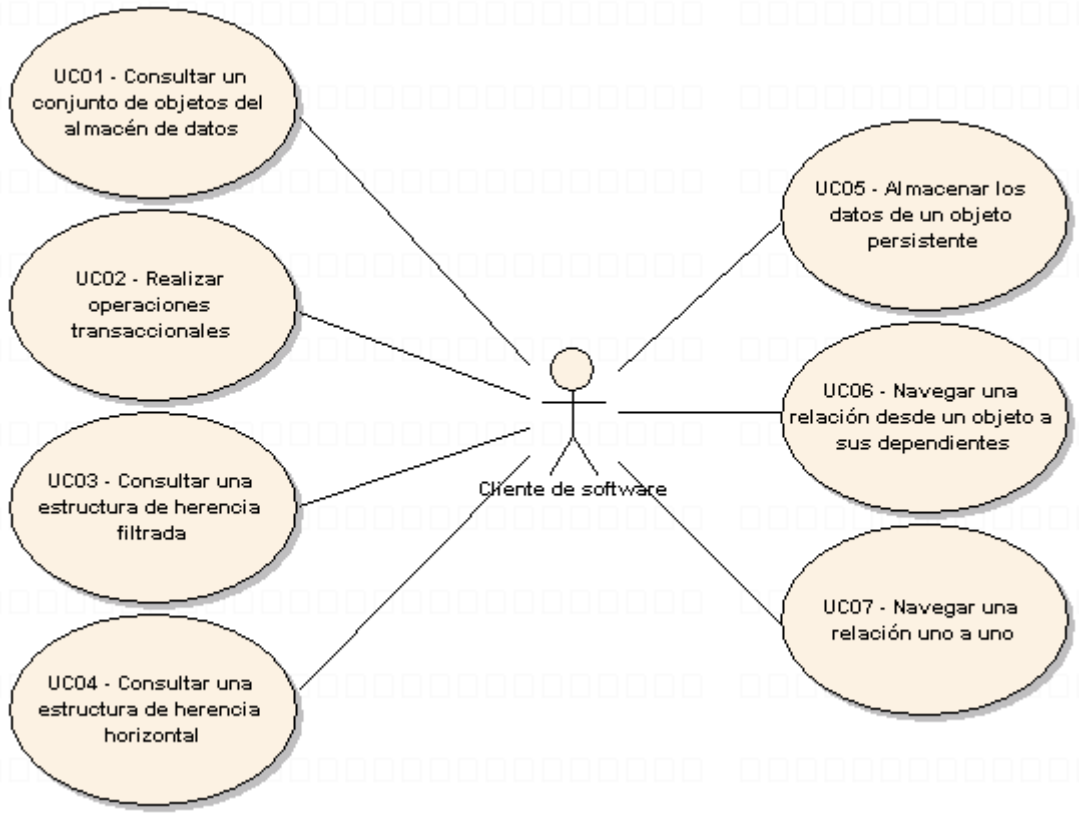


Fig 6. Diagrama general de casos de uso de operaciones de persistencia

DETALLE DE CASOS DE USO:	
Nombre:	UC01 - Consultar un conjunto de objetos del almacén de datos
Objetivo:	Permitir la recuperación de instancias de objetos persistentes de una manera transparente e independiente al almacén de datos.

Requerimientos relacionados:

- RQ14 - Generar consultas dinámicamente, eliminando la utilización del lenguaje propietario del almacén de datos.
- RQ15 - Generar consultas al almacén de datos con una sintaxis orientada a objetos fuertemente tipadas.
- RQ22 - Mapear clases a un almacén de datos para que sus datos y estado sean persistentes en el tiempo.
- RQ23 - Mapear una clase a una tabla del almacén de datos.

Escenarios:

Escenario Normal	Se devuelve una lista con los objetos consultados	1. El cliente expresa una consulta sobre el objeto mapeado utilizando sintaxis orientada a objetos. 2. El framework utiliza la meta-información de mapeo del objeto para generar una consulta para el almacén de datos. 3. El framework utiliza el proveedor de dialecto configurado para convertir la expresión de consulta al dialecto SQL particular del motor de base de datos. 4. El framework instancia y reconstruye los objetos consultados. 5. El cliente de software recibe los objetos instanciados correctamente.
Escenario Alternativo	Se encuentra un error durante el mapeo	4.a. El framework encuentra un error durante el mapeo y / o la instanciación y emite una excepción.

Condiciones:

Mapeo existente	Pre-condición	Existe el mapeo de una clase y una tabla del almacén de datos.
Exito	Post-condición	Se reconstruyen los objetos consultados.
Fracaso 1	Post-condición	Se encuentra un error durante la consulta y emite una excepción.

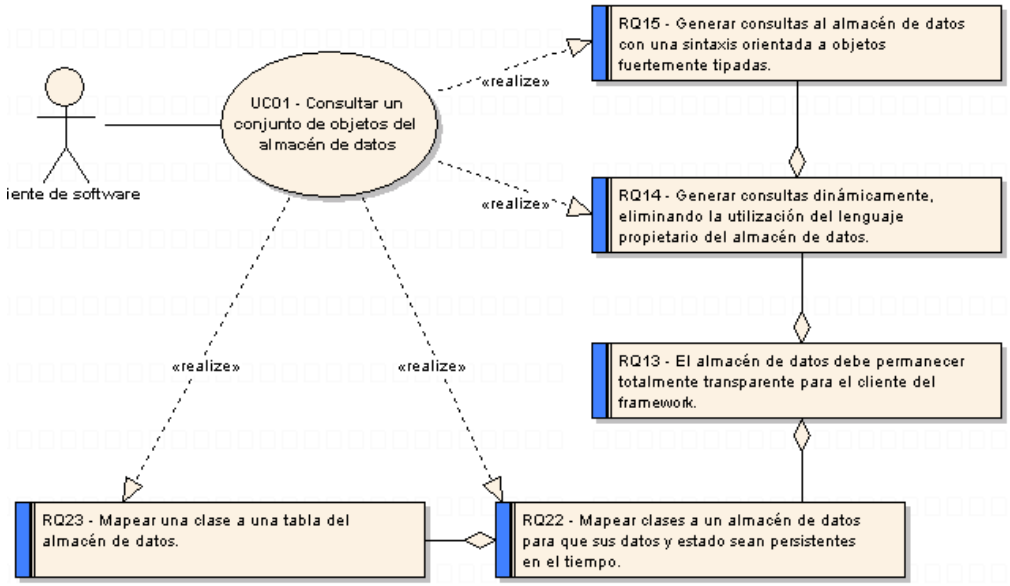


Fig 7. UC01 - Consultar un conjunto de objetos del almacén de datos

DETALLE DE CASOS DE USO:		
Nombre:	UC02 - Realizar operaciones transaccionales	
Objetivo:	Asistir y permitir mantener la integridad de datos de la aplicación.	
Requerimientos relacionados:		
▪ RQ30 - Proveer control transaccional.		
Escenarios:		
Escenario Normal	La transacción se confirma correctamente	1. El cliente de software indica el principio de una transacción. 2. El framework asigna el contexto transaccional a cada objeto que se instancie dentro del límite de la transacción. 3. El cliente de software vota el éxito de la transacción. 4. Si no hubo votos de cancelación, la transacción se confirma y se graban los datos en la base de datos.
Escenario Alternativo	La transacción se cancela	4.a. Si hubo votos de cancelación, la transacción se aborta y se vuelven atrás los cambios en la base de datos.
Condiciones:		
Exito	Post-condición	La transacción se confirma y se guardan los datos.
Fracaso 1	Post-condición	Se cancela la transacción y se vuelven atrás los cambios realizados.

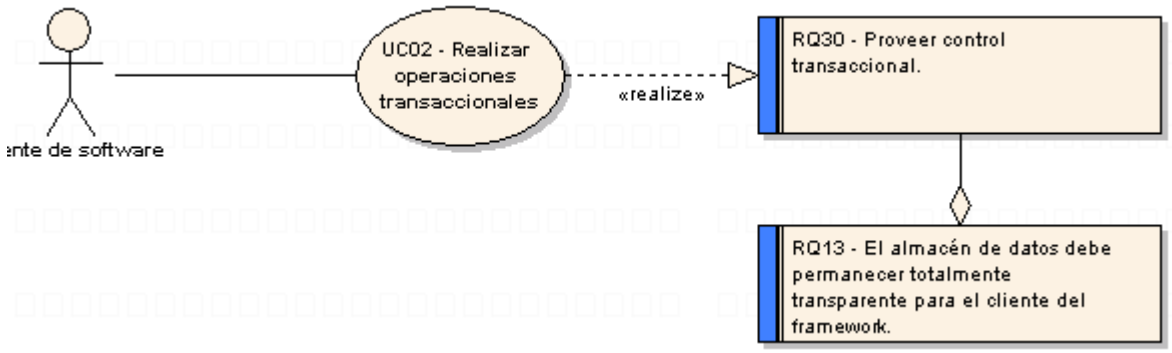


Fig 8. UC02 - Realizar operaciones transaccionales

DETALLE DE CASOS DE USO:	
Nombre:	UC03 - Consultar una estructura de herencia filtrada
Autor:	Gonzalo Casas
Objetivo:	Simplificar la recuperación de clases y subclases de una misma jerarquía.

Requerimientos relacionados:

- RQ16 - Generar consultas polimórficas.
- RQ24 - Mapear estructuras de herencia a una estructura de datos.
- RQ26 - Permitir el mapeo filtrado de jerarquías de herencia.

Escenarios:

Escenario Normal	Se devuelve una lista de objetos de una estructura de herencia	1. El cliente de software expresa una consulta sobre un objeto de una estructura de herencia utilizando sintaxis orientada a objetos. 2. El framework utiliza la meta-información de mapeo del objeto y la cadena de herencia implicada para generar una consulta para el almacén de datos. 3. El framework utiliza el proveedor de dialecto configurado para convertir la expresión de consulta al dialecto SQL particular del motor de base de datos. 4. El framework ejecuta una consulta sobre la tabla filtrada, y a partir de éstos datos, instancia y reconstruye los objetos encontrados. 5. El cliente de software recibe una lista polimórfica de objetos instanciados correctamente.
Escenario Alternativo	Se encuentra un error durante el mapeo.	4.a. El framework encuentra una condición de error durante el mapeo o ejecución de la consulta y devuelve una excepción.

Condiciones:

Mapeo existente	Pre-condición	Existe el mapeo filtrado de una estructura de herencia a una tabla del almacén de datos.
Exito	Post-condición	Se devuelve una lista de objetos de una jerarquía de herencia.
Fracaso 1	Post-condición	Se detecta una condición de error durante el mapeo y devuelve una excepción.

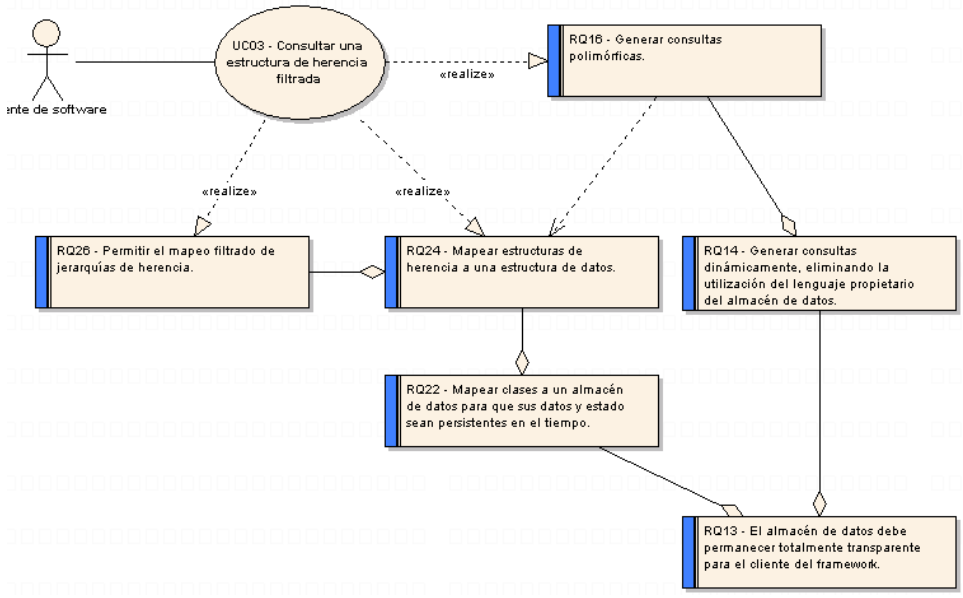


Fig 9. UC03 - Consultar una estructura de herencia filtrada

DETALLE DE CASOS DE USO:	
Nombre:	UC04 - Consultar una estructura de herencia horizontal
Objetivo:	Simplificar la recuperación de clases y subclases de una misma jerarquía.

Requerimientos relacionados:

- RQ16 - Generar consultas polimórficas.
- RQ24 - Mapear estructuras de herencia a una estructura de datos.
- RQ25 - Permitir el mapeo horizontal de jerarquías de herencia.

Escenarios:

Escenario Normal	Se devuelve una lista de objetos de una estructura de herencia	1. El cliente de software expresa una consulta sobre un objeto de una estructura de herencia horizontal utilizando sintaxis orientada a objetos. 2. El framework utiliza la meta-información de mapeo del objeto y la cadena de herencia implicada para generar una consulta para el almacén de datos. 3. El framework utiliza el proveedor de dialecto configurado para convertir la expresión de consulta al dialecto SQL particular del motor de base de datos. 4. El framework ejecuta N consultas sobre las tablas necesarias, y a partir de éstos datos, instancia y reconstruye los objetos encontrados. 5. El cliente de software recibe una lista polimórfica de objetos instanciados correctamente.
Escenario Alternativo	Se encuentra un error durante el mapeo	4.a. El framework encuentra una condición de error durante el mapeo o ejecución de la consulta y devuelve una excepción.

Condiciones:

Mapeo existente	Pre-condición	Existe el mapeo filtrado de una estructura de herencia horizontal en varias tablas del almacén de datos.
Exito	Post-condición	Se devuelve una lista de objetos de una jerarquía de herencia.
Fracaso 1	Post-condición	Se detecta una condición de error durante el mapeo y devuelve una excepción.

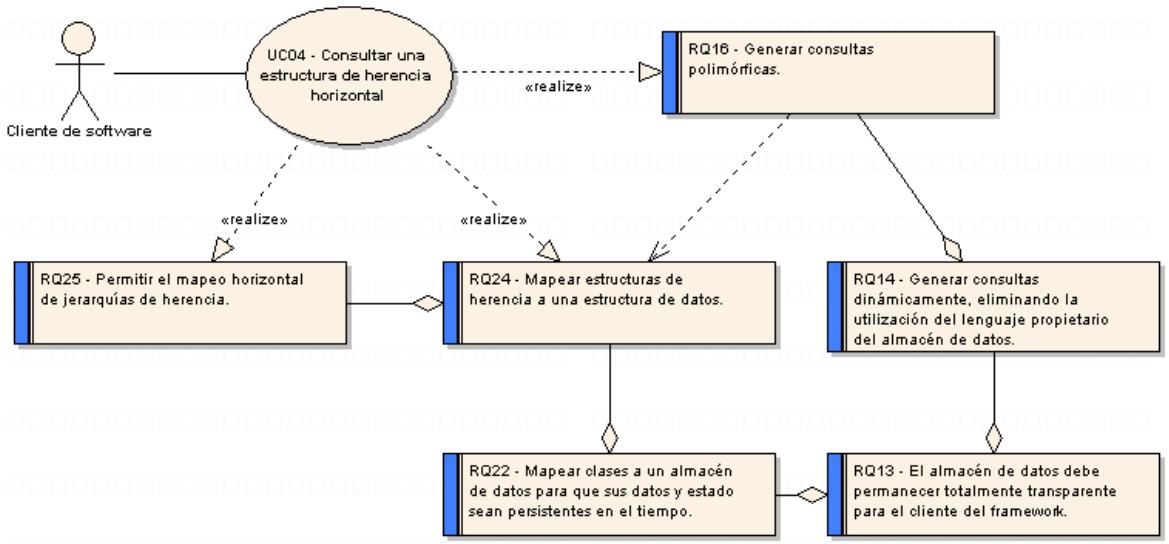


Fig 10. UC04 - Consultar una estructura de herencia horizontal

DETALLE DE CASOS DE USO:		
Nombre:	UC05 - Almacenar los datos de un objeto persistente	
Objetivo:	Hacer persistentes en el tiempo los cambios realizados a un objeto en memoria.	
Requerimientos relacionados:		
▪ RQ22 - Mapear clases a un almacén de datos para que sus datos y estado sean persistentes en el tiempo. ▪ RQ23 - Mapear una clase a una tabla del almacén de datos.		
Escenarios:		
Escenario Normal	Los datos se guardan correctamente en la base de datos	1. El cliente de software recupera una instancia de un objeto. 2. El cliente de software realiza cambios en los datos del objeto. 3. El cliente de software solicita que se persistan los cambios. 4. El framework almacena los cambios y devuelve el control al cliente de software.
Escenario Alternativo	Los datos no se guardan	4.a. El framework encuentra un error durante el almacenamiento y devuelve una excepción.
Condiciones:		
Mapeo existente	Pre-condición	Existe la definición del mapeo entre una clase y el mecanismo de persistencia.
Exito	Post-condición	Los datos se almacenan correctamente en la base de datos
Fracaso 1	Post-condición	Los datos no se guardan en la base de datos

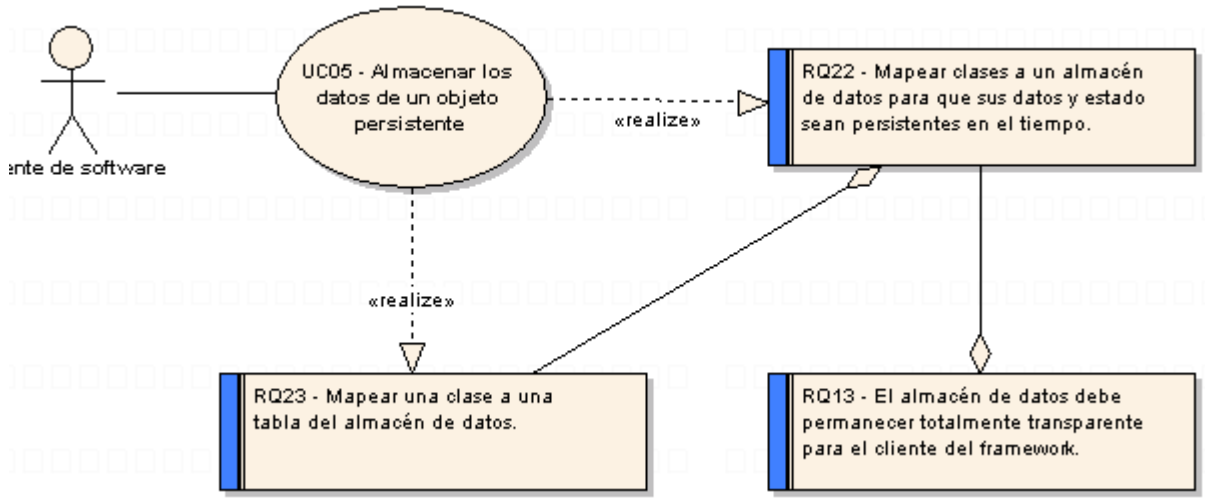


Fig 11. UC05 - Almacenar los datos de un objeto persistente

DETALLE DE CASOS DE USO:	
Nombre:	UC06 - Navegar una relación desde un objeto a sus dependientes
Objetivo:	Proveer un método fácil y simple de navegar las relaciones entre objetos.

Requerimientos relacionados:

- RQ18 - Mapear relaciones muchos-a-muchos.
- RQ19 - Mapear relaciones 1-a-muchos.

Escenarios:

Escenario Normal	Se accede a la colección de objetos definida por la relación.	1. El cliente de software instancia un objeto que tiene definida una relación de contención de objetos, del tipo Libro -> Hojas. 2. El cliente navega desde el objeto padre hasta la colección de objetos relacionados con la notación de punto de la sintáxis orientada a objetos. 3. El framework instancia los objetos de la relación y devuelve una colección con éstos.
Escenario Alternativo	Se encuentra un error al establecer o recuperar la relación.	3.a. El framework encuentra una condición de error estableciendo la relación y devuelve una excepción.

Condiciones:

Mapeo existente	Pre-condición	Debe existir el mapeo de la clase y de las relaciones.
Exito	Post-condición	Se accede a la colección de objetos definida por la relación.
Fracaso 1	Post-condición	Se encuentra un error al establecer o recuperar la relación.

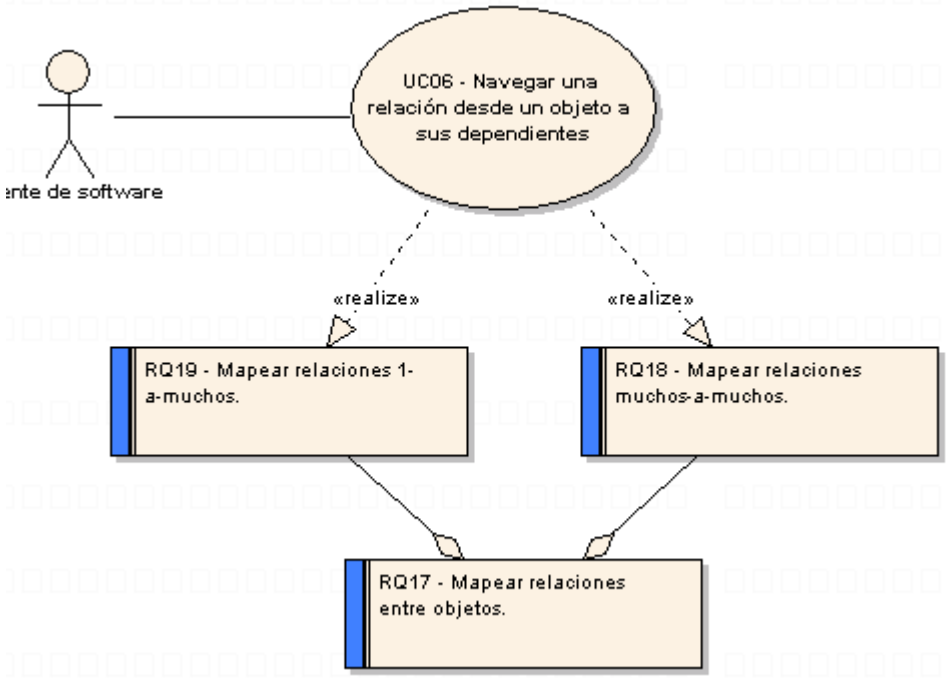


Fig 12. UC06 - Navegar una relación desde un objeto a sus dependientes

DETALLE DE CASOS DE USO:		
Nombre:	UC07 - Navegar una relación uno a uno	
Objetivo:	Proveer un método fácil y simple de navegar las relaciones entre objetos.	
Requerimientos relacionados:		
▪ RQ20 - Mapear relaciones 1-a-1.		
Escenarios:		
Escenario Normal	Se accede a la instancia del objeto relacionado	1. El cliente de software instancia un objeto con una o más relaciones uno-a-uno, por ejemplo: Usuario -> Dirección. 2. El cliente de software accede al objeto relacionado utilizando la notación de punto del paradigma de orientación a objetos. 3. El framework instancia el objeto relacionado y devuelve el control.
Escenario Alternativo	Se detecta una condición de error durante el mapeo	3.a. El framework detecta una condición de error durante el mapeo y devuelve una excepción.
Condiciones:		
Mapeo existente	Pre-condición	Debe existir el mapeo de las clases y de la relación uno-a-uno.
Exito	Post-condición	Se accede a la instancia del objeto relacionado
Fracaso 1	Post-condición	Se detecta una condición de error durante el mapeo

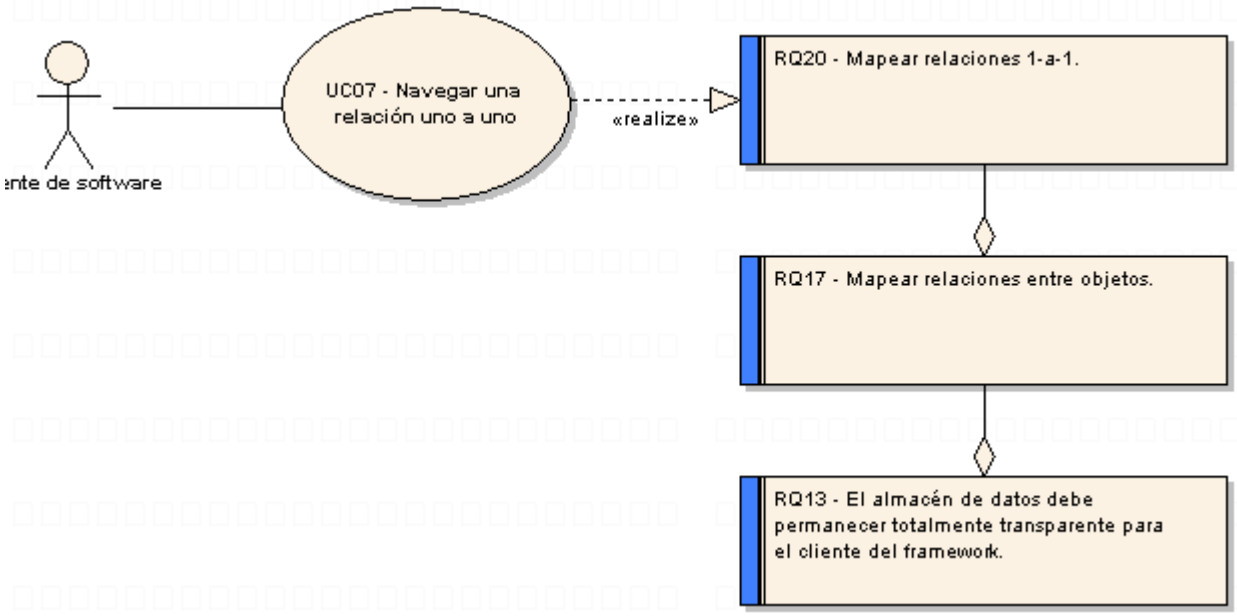


Fig 13. UC07 - Navegar una relación uno a uno

CASOS DE USO DE MAPEOS DE PERSISTENCIA

Los casos de uso de ésta sección representan las situaciones de configuración, en las cuales el actor principal del caso de uso no es el software cliente del Framework, sino el desarrollador propiamente dicho, que utiliza la API de configuración que brinda el Framework para personalizarlo y adaptarlo a sus necesidades. El siguiente diagrama general es un vistazo global a los casos de uso de ésta sección que más adelante serán descriptos en detalle.

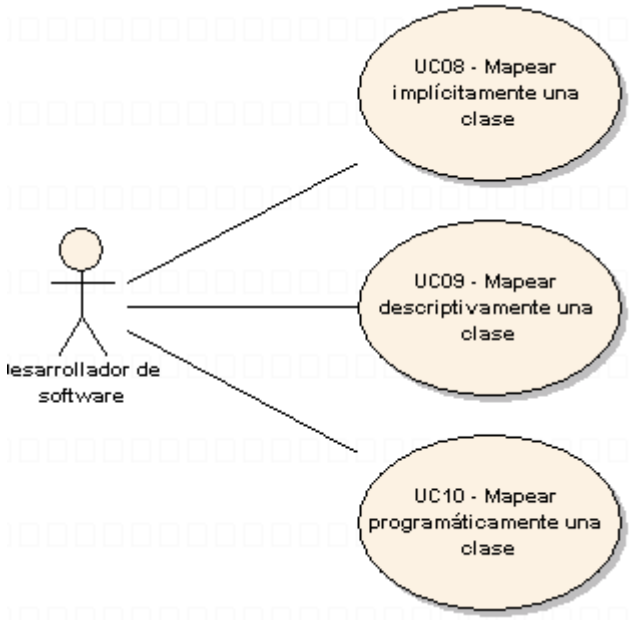


Fig 14. Diagrama general de casos de uso relacionados al mapeo de persistencia

DETALLE DE CASOS DE USO:		
Nombre:	UC08 - Mapear implícitamente una clase	
Objetivo:	Permitir mapeos rápidos y eficientes, especialmente para proyectos pequeños que no tienen requerimientos complejos en cuanto a la estructura del dominio de negocios.	
Requerimientos relacionados:		
▪ RQ27 - Mapeo automático de clases.		
Escenarios:		
Escenario Normal	Se reconstruye el objeto	1. El desarrollador construye una clase de negocios persistente utilizando la convención de nombres para mapeo implícito. 2. Al momento de recuperar una instancia, el framework detecta la ausencia de un mapeo explícito y realiza el mapeo implícito. 3. El framework reconstruye el objeto correctamente.
Escenario Alternativo	Se produce un error de mapeo	3.a. El framework detecta un error de mapeo y devuelve una excepción.
Condiciones:		
Exito	Post-condición	El mapeo implícito es correcto y se reconstruye el objeto.
Fracaso 1	Post-condición	Se detecta un error de mapeo

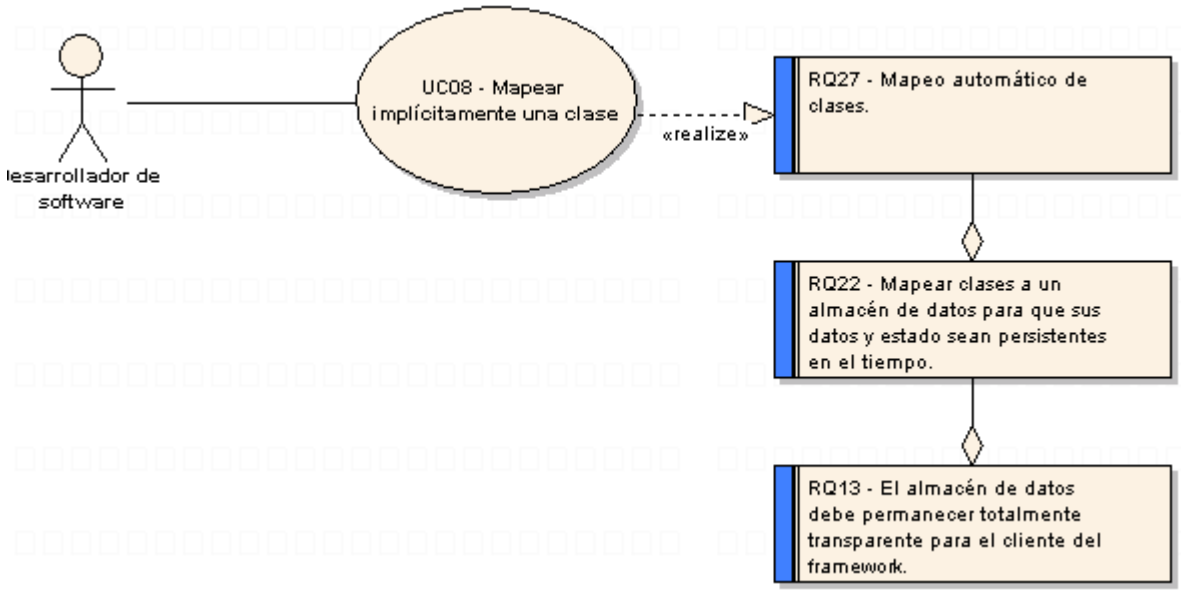


Fig 15. UC08 - Mapear implícitamente una clase

DETALLE DE CASOS DE USO:		
Nombre:	UC09 - Mapear descriptivamente una clase	
Objetivo:	Permitir un mapeo flexible, que pueda ser alterado sin necesidad de recompilar la aplicación.	
Requerimientos relacionados:		
▪ RQ29 - Mapeo descriptivo en archivos de configuración.		
Escenarios:		
Escenario Normal	Se reconstruye el objeto	1. El desarrollador construye una clase de negocios persistente. 2. El desarrollador construye el archivo descriptivo de mapeo indicando las conexiones entre la clase de negocios y el almacén de datos. 3. Al momento de recuperar una instancia, el framework detecta la configuración de mapeos. 4. El framework reconstruye el objeto correctamente.
Escenario Alternativo	Se produce un error de mapeo	4.a. El framework detecta un error de mapeo y devuelve una excepción.
Condiciones:		
Exito	Post-condición	El mapeo es correcto y se reconstruye el objeto.
Fracaso 1	Post-condición	Se detecta un error de mapeo

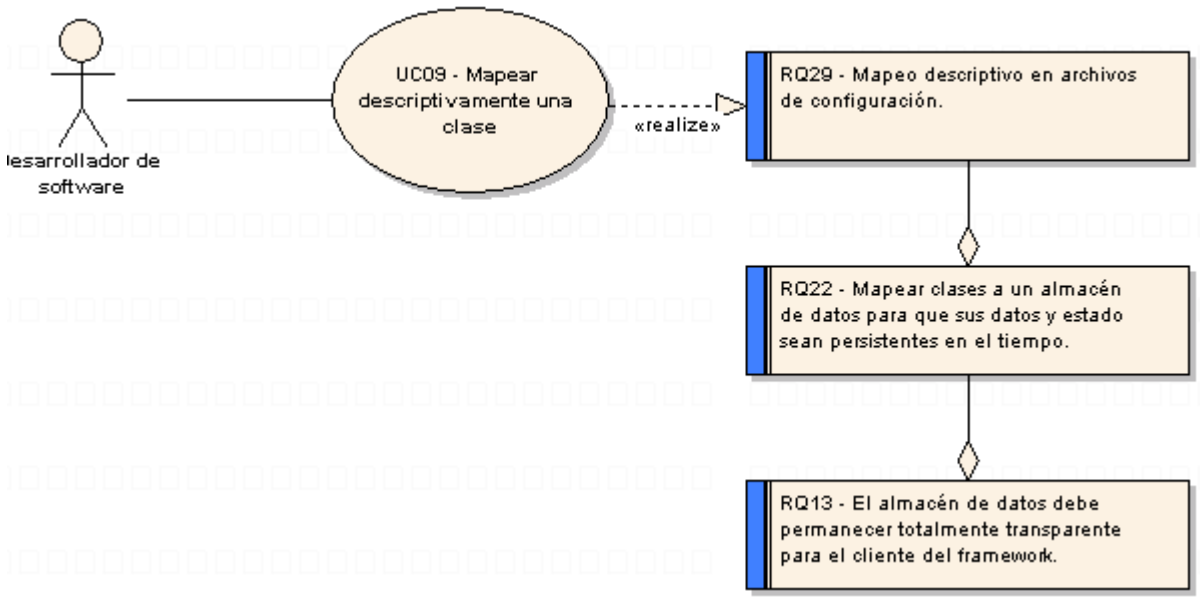


Fig 16. UC09 - Mapear descriptivamente una clase

DETALLE DE CASOS DE USO:		
Nombre:	UC10 - Mapear programáticamente una clase	
Objetivo:	Permitir un mapeo eficiente y reducir la cantidad de recursos (archivos) que es necesario mantener.	
Requerimientos relacionados:		
▪ RQ28 - Mapeo programático desde código.		
Escenarios:		
Escenario Normal	Se reconstruye el objeto	1. El desarrollador construye una clase de negocios persistente. 2. El desarrollador genera las instrucciones de mapeo programáticamente, indicando las conexiones entre la clase de negocios y el almacén de datos. 3. Al momento de recuperar una instancia, el framework detecta la configuración de mapeos. 4. El framework reconstruye el objeto correctamente.
Escenario Alternativo	Se produce un error de mapeo	4.a. El framework detecta un error de mapeo y devuelve una excepción.
Condiciones:		
Exito	Post-condición	El mapeo es correcto y se reconstruye el objeto.
Fracaso 1	Post-condición	Se detecta un error de mapeo

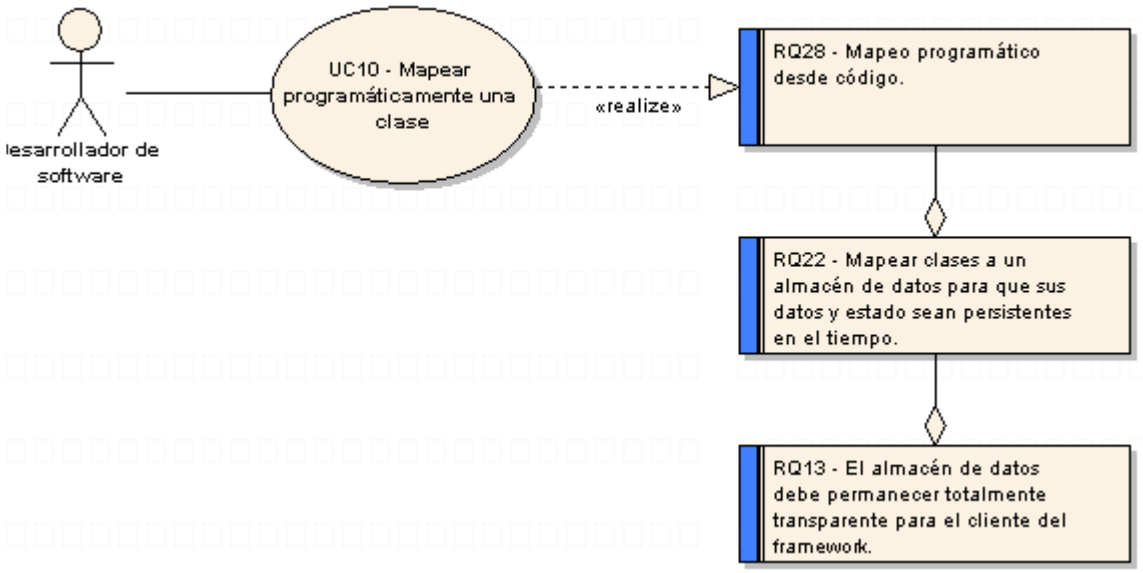


Fig 17. UC10 - Mapear programáticamente una clase

CASOS DE USO DE AUTO-VALIDACIÓN DE MODELOS

En ésta sección se detallan los casos de uso derivados de los requerimientos de auto-validación de modelos, que tienen como actor principal al software cliente del Framework, que interactúa con éste a través de la API provista.

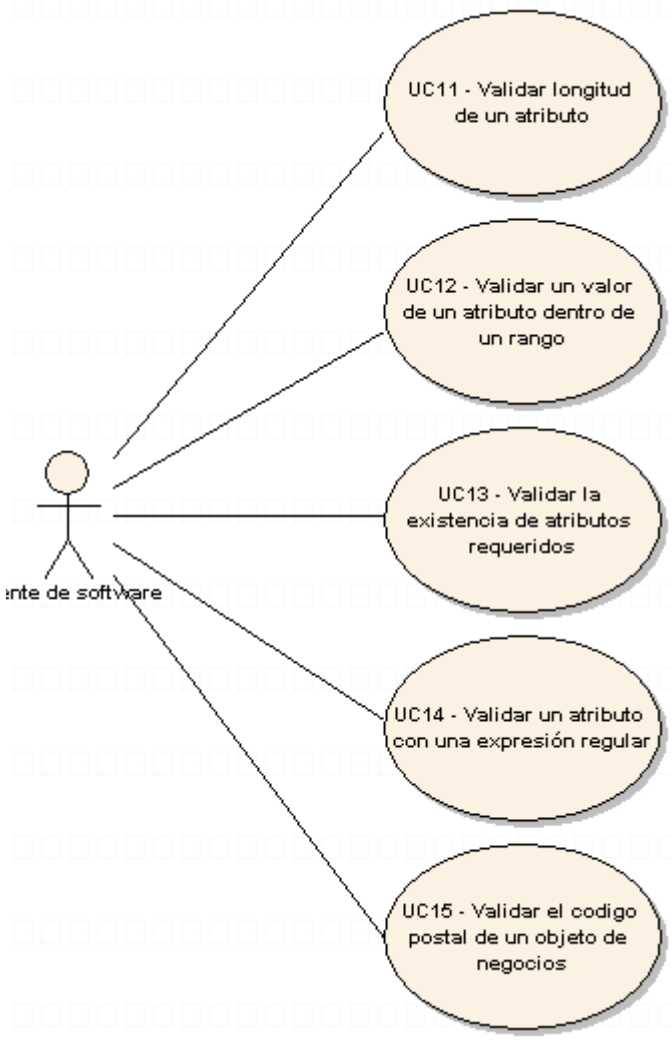


Fig 18. Diagrama general de casos de uso de auto-validación de modelos

DETALLE DE CASOS DE USO:	
Nombre:	UC11 - Validar longitud de un atributo
Objetivo:	Impedir que se asignen valores de longitud inválida para los atributos de negocio, y con esto, mantener la integridad de los datos.

Requerimientos relacionados:

- RQ07 - Validaciones de longitud máxima de un atributo.
- RQ08 - Validaciones de longitud mínima de un atributo.

Escenarios:

Escenario Normal	El atributo es válido	1. El cliente de software instancia un objeto del modelo de negocios. 2. El cliente de software asigna un valor a un atributo del objeto instanciado. 3. El framework valida automáticamente la longitud del dato asignado. 4. El framework comprueba que el valor es válido para el atributo.
Escenario Alternativo	La longitud es mayor a la permitida	4.a. El framework detecta que la longitud del valor asignado sobrepasa la máxima definida como válida para el atributo, y emite una excepción.
Escenario Alternativo	La longitud es menor a la permitida	4.a. El framework detecta que la longitud del valor asignado es menor a la mínima definida como válida para el atributo, y emite una excepción.

Condiciones:

Validadores definidos	Pre-condición	Deben existir los validadores de longitud para un atributo de una de las clases del modelo de negocios.
Exito	Post-condición	El atributo es válido.
Fracaso 1	Post-condición	La longitud es mayor a la permitida, el atributo es inválido.
Fracaso 2	Post-condición	La longitud es menor a la permitida, el atributo es inválido.

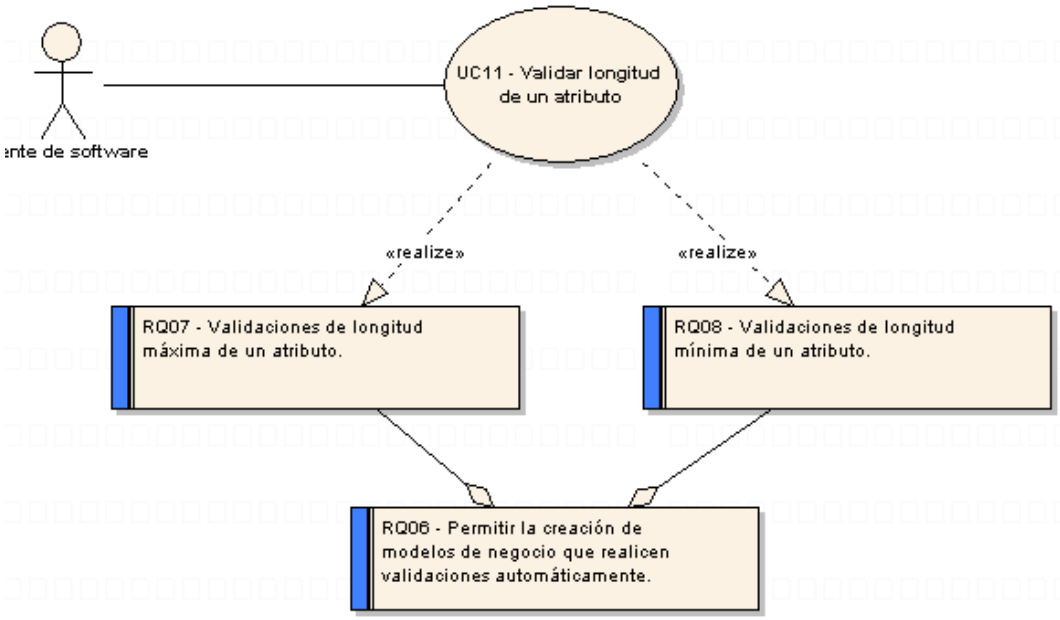


Fig 19. UC11 - Validar longitud de un atributo

DETALLE DE CASOS DE USO:		
Nombre:	UC12 - Validar un valor de un atributo dentro de un rango	
Objetivo:	Impedir que se asignen valores fuera de los rangos válidos para los atributos de negocio, y con esto, mantener la integridad de los datos.	
Requerimientos relacionados:		
▪ RQ09 - Validaciones de rangos de valores válidos.		
Escenarios:		
Escenario Normal	El atributo es válido	1. El cliente de software instancia un objeto del modelo de negocios. 2. El cliente de software asigna un valor a un atributo del objeto instanciado. 3. El framework valida automáticamente que el dato asignado esté dentro de un rango permitido. 4. El framework comprueba que el valor es válido para el atributo.
Escenario Alternativo	El valor está fuera del rango permitido	4.a. El framework detecta que el valor asignado está fuera del rango definido como válido para el atributo, y emite una excepción.
Condiciones:		
Validadores definidos	Pre-condición	Debe existir el validador de rango para un atributo de una de las clases del modelo de negocios.
Exito	Post-condición	El atributo es válido.
Fracaso 1	Post-condición	El valor del atributo está fuera del rango permitido, el atributo es inválido.

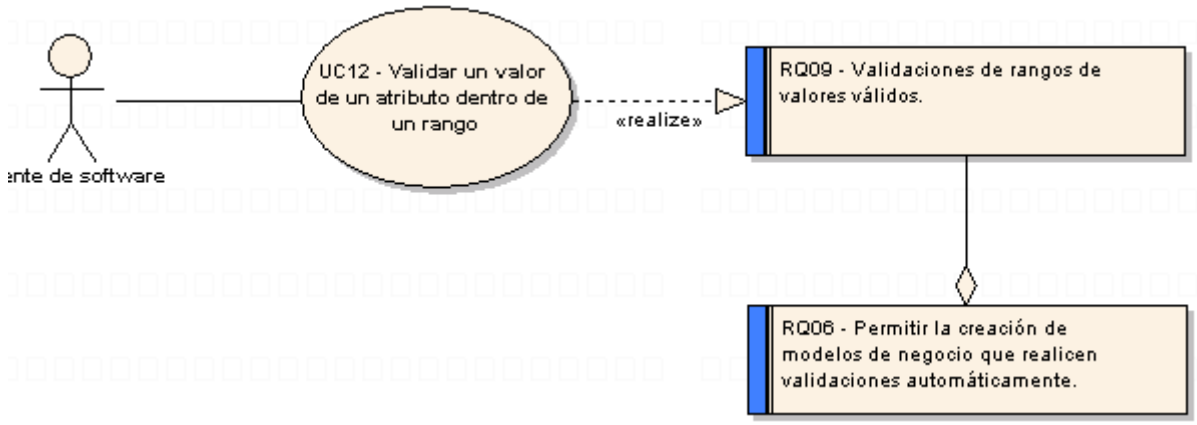


Fig 20. UC12 - Validar un valor de un atributo dentro de un rango

DETALLE DE CASOS DE USO:		
Nombre:	UC13 - Validar la existencia de atributos requeridos	
Objetivo:	Impedir que se guarden objetos de negocio incompletos o con valores inválidos, y con ésto, mantener la integridad de los datos.	
Requerimientos relacionados:		
▪ RQ10 - Validación de atributos requeridos.		
Escenarios:		
Escenario Normal	La clase es válida	1. El cliente de software instancia un objeto del modelo de negocios. 2. El cliente de software asigna uno o más valores a un atributo del objeto instanciado. 3. El framework comprueba que todos los atributos requeridos estén presentes.
Escenario Alternativo	La clase es inválida, hay atributos no asignados	3.a. El framework detecta que uno o más valores de atributos no han sido asignados, y emite una excepción de validación.
Condiciones:		
Validadores definidos	Pre-condición	Deben existir los validadores de existencia requerida para uno o más atributos de una de las clases del modelo de negocios.
Exito	Post-condición	La clase es válida.
Fracaso 1	Post-condición	Uno o más valores de atributos no están definidos, la clase es inválida.

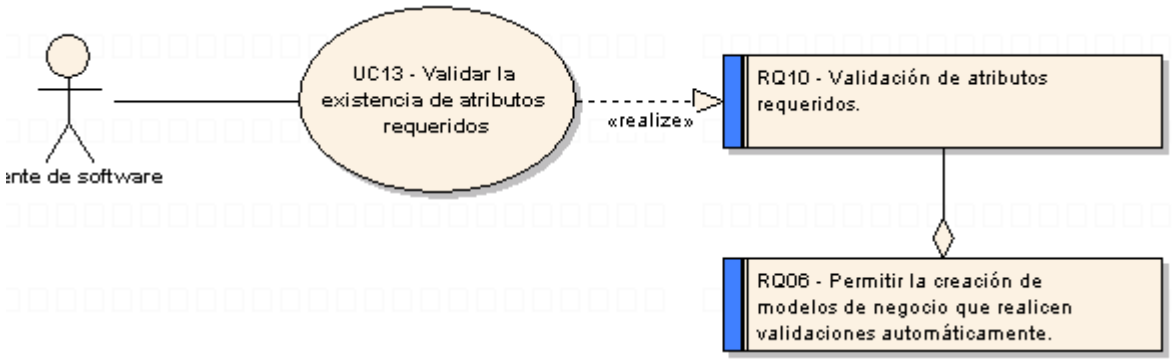


Fig 21. UC13 - Validar la existencia de atributos requeridos

DETALLE DE CASOS DE USO:		
Nombre:	UC14 - Validar un atributo con una expresión regular	
Objetivo:	Impedir que se asignen valores inválidos para los atributos de negocio, y con ésto, mantener la integridad de los datos.	
Requerimientos relacionados:		
▪ RQ11 - Máscaras de validación de un atributo.		
Escenarios:		
Escenario Normal	El valor del atributo es válido	1. El cliente de software instancia un objeto del modelo de negocios. 2. El cliente de software asigna un valor a un atributo del objeto instanciado. 3. El framework valida automáticamente que el dato asignado esté validado contra la expresión regular definida. 4. El framework comprueba que el atributo es válido.
Escenario Alternativo	El valor del atributo es inválido	4.a. El framework detecta que el valor del atributo no es válido, y emite una excepción de validación.
Condiciones:		
Validadores definidos	Pre-condición	Debe existir al menos un validador de expresión regular para un atributo de una de las clases del modelo de negocios.
Exito	Post-condición	El valor del atributo es válido.
Fracaso 1	Post-condición	El valor del atributo es inválido.

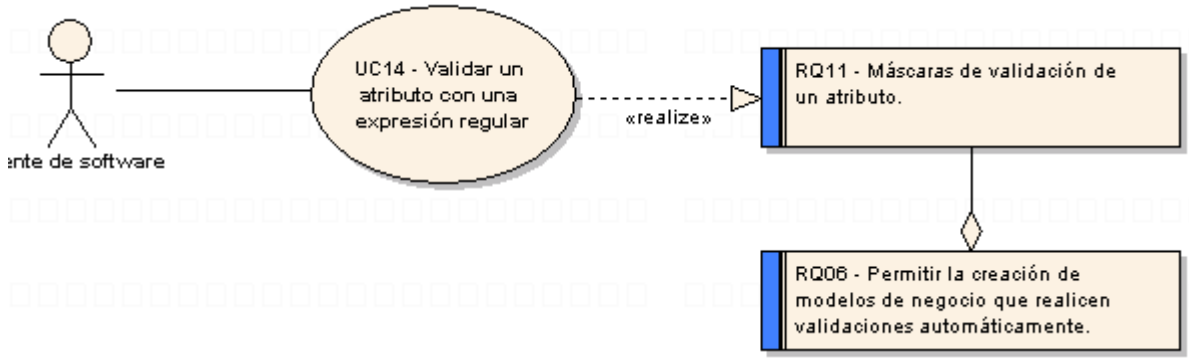


Fig 22. UC14 - Validar un atributo con una expresión regular

DETALLE DE CASOS DE USO:		
Nombre:	UC15 - Validar el código postal de un objeto de negocios	
Objetivo:	Permitir las validaciones personalizadas de los atributos de negocio, y con ésto, mantener la integridad de los datos.	
Requerimientos relacionados:		
▪ RQ12 - Permitir la creación de validadores personalizados.		
Escenarios:		
Escenario Normal	El valor del atributo es válido	1. El cliente de software instancia un objeto del modelo de negocios. 2. El cliente de software asigna un valor al atributo "Código Postal" del objeto instanciado. 3. El framework valida automáticamente el código postal asignado. 4. El framework comprueba que el código postal es válido.
Escenario Alternativo	El valor del atributo es inválido	4.a. El framework detecta que el código postal no es válido, y emite una excepción de validación.
Condiciones:		
Validadores definidos	Pre-condición	Debe existir un validador personalizado para códigos postales asignado a un atributo de una de las clases del modelo de negocios.
Exito	Post-condición	El valor del atributo es válido.
Fracaso 1	Post-condición	El valor del atributo es inválido.

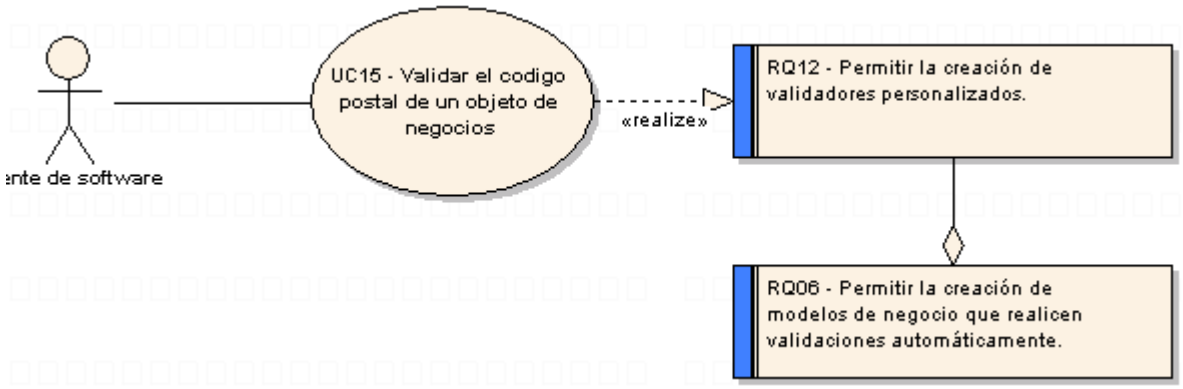


Fig 23. UC15 - Validar el código postal de un objeto de negocios

CASOS DE USO DE INTERFACES DE USUARIO

Los casos de uso de interfaces de usuario describen las interacciones entre los actores del sistema y el mecanismo de abstracción de tecnología de interfaces de usuario provisto por el Framework.

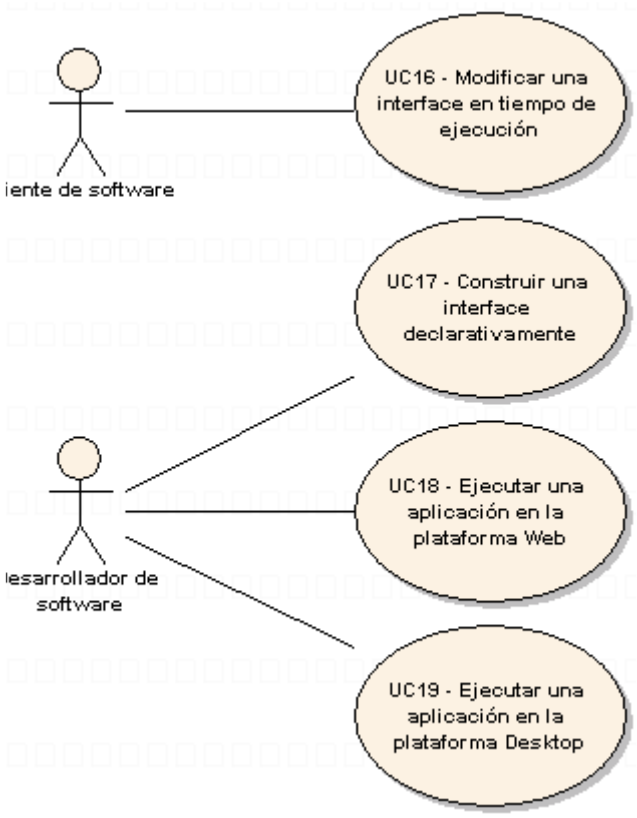


Fig 24. Diagrama general de casos de usos para interfaces de usuario

DETALLE DE CASOS DE USO:		
Nombre:	UC16 - Modificar una interface en tiempo de ejecución	
Objetivo:	Permitir alterar una interface de usuario en tiempo de ejecución.	
Requerimientos relacionados:		
▪ RQ03 - Permitir la construcción de interfaces basadas en el concepto de anidación de componentes. ▪ RQ04 - Permitir la manipulación programática de los controles de interface de usuario.		
Escenarios:		
Escenario Normal	La interface se actualiza con las modificaciones	1. El cliente de software selecciona un control de la interface de usuario 2. Se construye programáticamente un nuevo control 3. El cliente de software asigna el control recién creado, como un sub-componente del control seleccionado en el primer paso. 4. El framework redibuja la interface para reflejar el cambio.
Condiciones:		
Interface definida	Pre-condición	Existe una interface de usuario definida que contiene al menos un control asignado.
Exito	Post-condición	La interface se actualiza con las modificaciones

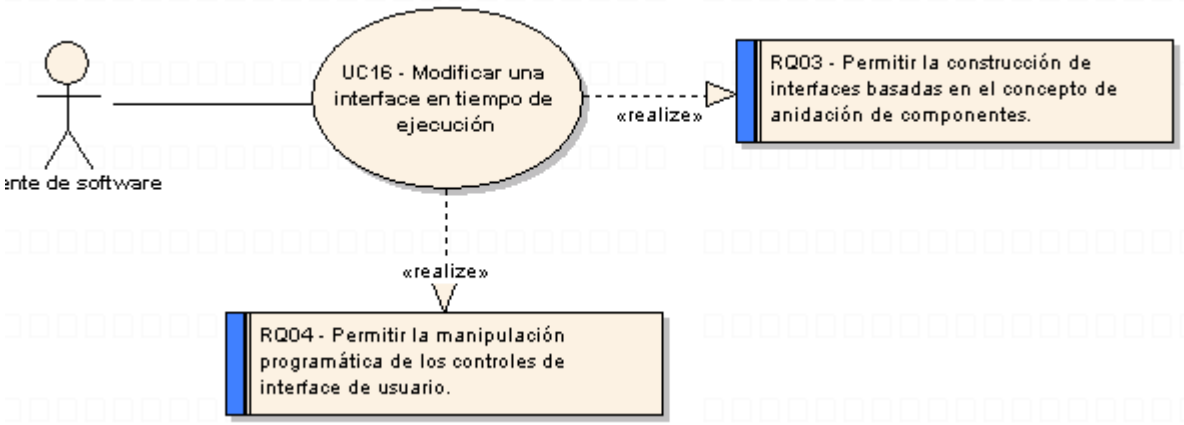


Fig 25. UC16 - Modificar una interface en tiempo de ejecución

DETALLE DE CASOS DE USO:		
Nombre:	UC17 - Construir una interface declarativamente	
Objetivo:	Propiciar la creación de interfaces declarativas con separación de estilo y estructura.	
Requerimientos relacionados:		
▪ RQ02 - Permitir la definición de interfaces de usuario a través de un lenguaje declarativo de tipo XML. ▪ RQ05 - Separar la definición de estilo y de estructura de interfaces de usuario.		
Escenarios:		
Escenario Normal	La interface queda definida declarativamente	1. El desarrollador de software define un archivo para la declaración de estructura de la interface de usuario. 2. El desarrollador de software define un archivo de hoja de estilos para la interface de usuario. 3. La interface queda definida declarativamente para ser utilizada por el Framework.
Condiciones:		
Exito	Post-condición	La interface queda definida declarativamente

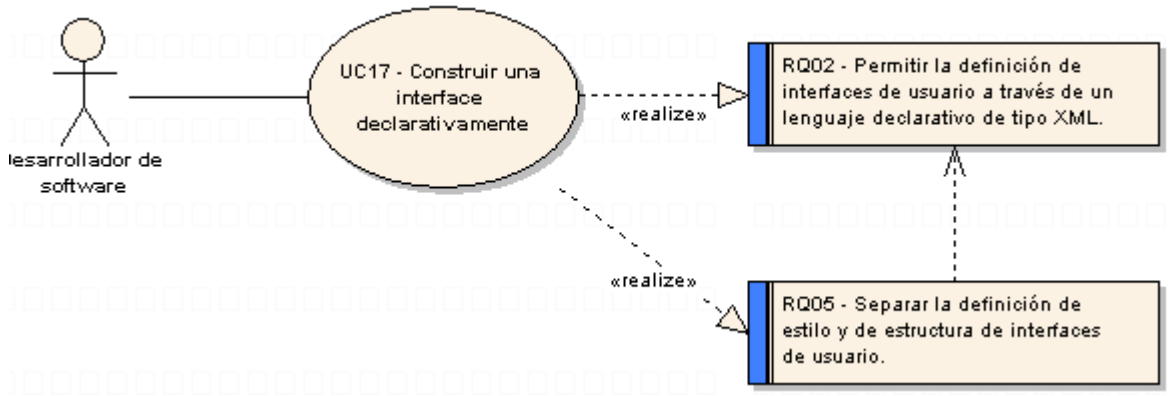


Fig 26. UC17 - Construir una interface declarativamente

DETALLE DE CASOS DE USO:		
Nombre:	UC18 - Ejecutar una aplicación en la plataforma Web	
Objetivo:	Comprobar la independencia de plataforma de presentación provista por el framework.	
Requerimientos relacionados:		
▪ RQ01 - Abstraer la tecnología de presentación.		
Escenarios:		
Escenario Normal	Se ejecuta la aplicación bajo la plataforma Web	1. El desarrollador de software accede al punto de entrada Web de la aplicación. 2. El framework genera en tiempo de ejecución la interface de usuario basada en la definición declarativa de la misma. 3. La aplicación se ejecuta correctamente.
Escenario Alternativo	Se produce un error de presentación en tiempo de ejecución	2.a. El framework encuentra un error durante la generación de la interface gráfica y devuelve una excepción.
Condiciones:		
Exito	Post-condición	Se ejecuta la aplicación bajo la plataforma Web.
Fracaso 1	Post-condición	Se produce un error de presentación en tiempo de ejecución.

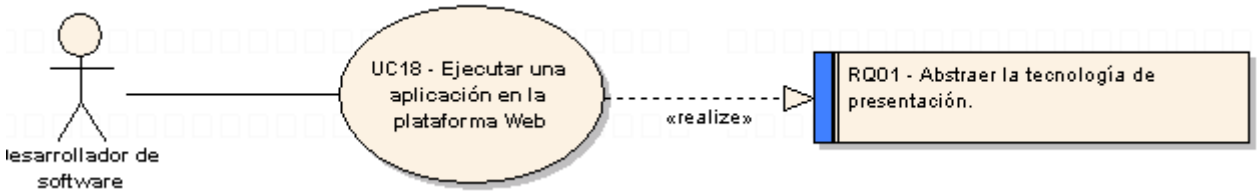


Fig 27. UC18 - Ejecutar una aplicación en la plataforma Web

DETALLE DE CASOS DE USO:		
Nombre:	UC19 - Ejecutar una aplicación en la plataforma Desktop	
Objetivo:	Comprobar la independencia de plataforma de presentación provista por el framework.	
Requerimientos relacionados:		
▪ RQ01 - Abstraer la tecnología de presentación.		
Escenarios:		
Escenario Normal	Se ejecuta la aplicación bajo la plataforma Desktop	1. El usuario ejecuta el archivo principal de entrada de la aplicación. 2. El framework genera en tiempo de ejecución la interface de usuario basada en la definición declarativa de la misma. 3. La aplicación se ejecuta correctamente.
Escenario Alternativo	Se produce un error de presentación en tiempo de ejecución	2.a. El framework encuentra un error durante la generación de la interface gráfica y devuelve una excepción.
Condiciones:		
Exito	Post-condición	Se ejecuta la aplicación bajo la plataforma Desktop.
Fracaso 1	Post-condición	Se produce un error de presentación en tiempo de ejecución.

Fig 28. UC19 - Ejecutar una aplicación en la plataforma Desktop

MATRIZ DE TRAZABILIDAD : CASOS DE USO A REQUERIMIENTOS

La matriz de trazabilidad es una herramienta que permite la gestión e identificación del origen de cada uno de los casos de uso del sistema. La misma forma parte de los entregables definidos en el plan de proyecto.

	UC01 - Consultar un conjunto de objetos del almacén de datos	UC02 - Realizar operaciones transaccionales	UC03 - Consultar una estructura de herencia filtrada	UC04 - Consultar una estructura de herencia horizontal	UC05 - Almacenar los datos de un objeto persistente	UC06 - Navegar una relación desde un objeto a sus dependientes	UC07 - Navegar una relación uno a uno	UC08 - Mapear implícitamente una clase	UC09 - Mapear descriptivamente una clase	UC10 - Mapear programáticamente una clase	UC11 - Validar longitud de un atributo	UC12 - Validar un valor de un atributo dentro de un rango	UC13 - Validar la existencia de atributos requeridos	UC14 - Validar un atributo con una expresión regular	UC15 - Validar el código postal de un objeto de negocios	UC16 - Modificar una interface en tiempo de ejecución	UC17 - Construir una interface declarativamente	UC18 - Ejecutar una aplicación en la plataforma Web	UC19 - Ejecutar una aplicación en la plataforma Desktop
RQ01 - Abstraer la tecnología de presentación.																			
RQ02 - Permitir la definición de interfaces de usuario a través de un lenguaje declarativo de tipo XML.																			
RQ03 - Permitir la construcción de interfaces basadas en el concepto de anidación de componentes.																			
RQ04 - Permitir la manipulación programática de los controles de interface de usuario.																			
RQ05 - Separar la definición de estilo y de estructura de interfaces de usuario.																			
RQ06 - Permitir la creación de modelos de negocio que realicen validaciones automáticamente.																			
RQ07 - Validaciones de longitud máxima de un atributo.																			
RQ08 - Validaciones de longitud mínima de un atributo.																			
RQ09 - Validaciones de rangos de valores válidos.																			
RQ10 - Validación de atributos requeridos.																			
RQ11 - Máscaras de validación de un atributo.																			
RQ12 - Permitir la creación de validadores personalizados.																			
RQ13 - El almacén de datos debe permanecer totalmente transparente para el cliente del framework.																			

RQ14 - Generar consultas dinámicamente, eliminando la utilización del lenguaje propietario del almacén de datos.	■																		
RQ15 - Generar consultas al almacén de datos con una sintaxis orientada a objetos fuertemente tipadas.	■																		
RQ16 - Generar consultas polimórficas.			■	■															
RQ17 - Mapear relaciones entre objetos.																			
RQ18 - Mapear relaciones muchos-a-muchos.						■													
RQ19 - Mapear relaciones 1-a-muchos.						■													
RQ20 - Mapear relaciones 1-a-1.							■												
RQ21 - Mapear relaciones 1-a-muchos recursivas.																			
RQ22 - Mapear clases a un almacén de datos para que sus datos y estado sean persistentes en el tiempo.	■				■														
RQ23 - Mapear una clase a una tabla del almacén de datos.	■				■														
RQ24 - Mapear estructuras de herencia a una estructura de datos.			■	■															
RQ25 - Permitir el mapeo horizontal de jerarquías de herencia.				■															
RQ26 - Permitir el mapeo filtrado de jerarquías de herencia.			■																
RQ27 - Mapeo automático de clases.								■											
RQ28 - Mapeo programático desde código.										■									
RQ29 - Mapeo descriptivo en archivos de configuración.									■										
RQ30 - Proveer control transaccional.		■																	

DISEÑO DE ARQUITECTURA

El presente trabajo, a nivel arquitectónico se estructura en dos grandes estructuras funcionales, que tienen como objetivo abstraer las tecnologías subyacentes; En el primer caso, la tecnología de almacenamiento de datos (sistemas de bases de datos relacionales) y en el segundo, la tecnología de presentación o de interfaces de usuario.

A su vez, para proveer un mayor nivel de consistencia al desarrollo, el componente de abstracción de la tecnología de almacenamiento está subdivido en dos componentes: la *capa de acceso a datos*, que tiene como objetivo proveer una interface común a múltiples sistemas de almacenamiento; y la *capa de persistencia*, que provee las funcionalidades de abstracción de las técnicas y metodologías de almacenamiento tradicionales, permitiendo manipular objetos persistentes de forma transparente.

Para proveer estas funcionalidades, los componentes que conforman el presente proyecto se apoyan en un conjunto de librarías de soporte funcional, que simplifican tareas rutinarias. Entre los elementos principales de las librerías de soporte están:

- **Configuración:**
  - Infraestructura extensible para gestión de parámetros de configuración. Este componente también puede ser aprovechado por las aplicaciones cliente ya que la funcionalidad que provee es genérica y extensible.
- **Logging:**
  - Actualmente, la librería de *logging*, o registro, está basada en el componente open-source LOG4NET (http://logging.apache.org/log4net).
- **Propiedades:**
  - El concepto de propiedades como una clase instanciable, en lugar de la concepción tradicional de variables miembro y métodos de acceso es una de las principales ventajas provistas por el presente proyecto. Esta idea, nacida originalmente en el lenguaje Smalltalk, brida una flexibilidad difícil de igualar con otras técnicas, y una semántica de OO muy clara.
- **Expresiones:**
  - La librería de generación de expresiones está a cargo de la construcción de estructuras de datos que representan consultas orientadas a objeto.

El siguiente es un diagrama de alto nivel que representa estas divisiones arquitectónicas, y las maneras en que cada componente interactúa con los demás.

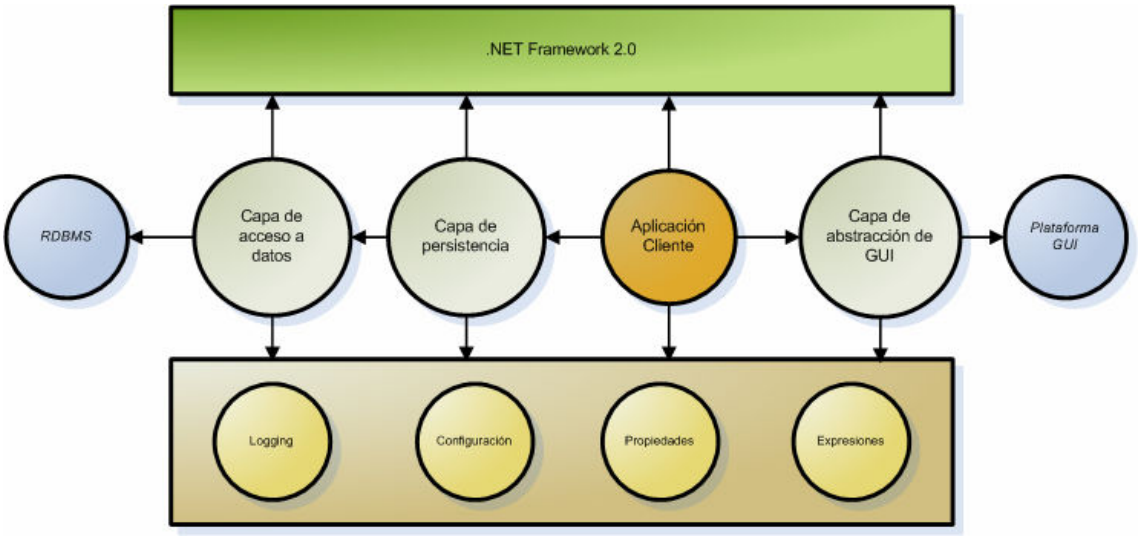


Fig 29. Diagrama de arquitectura general

MODELO DE COMPONENTES

Para la formalización del diseño de arquitectura, se desarrolló el modelo de componentes de UML 2.0 del proyecto. El diagrama a continuación proporciona una visión global y formalizada de la arquitectura descripta en la sección anterior.

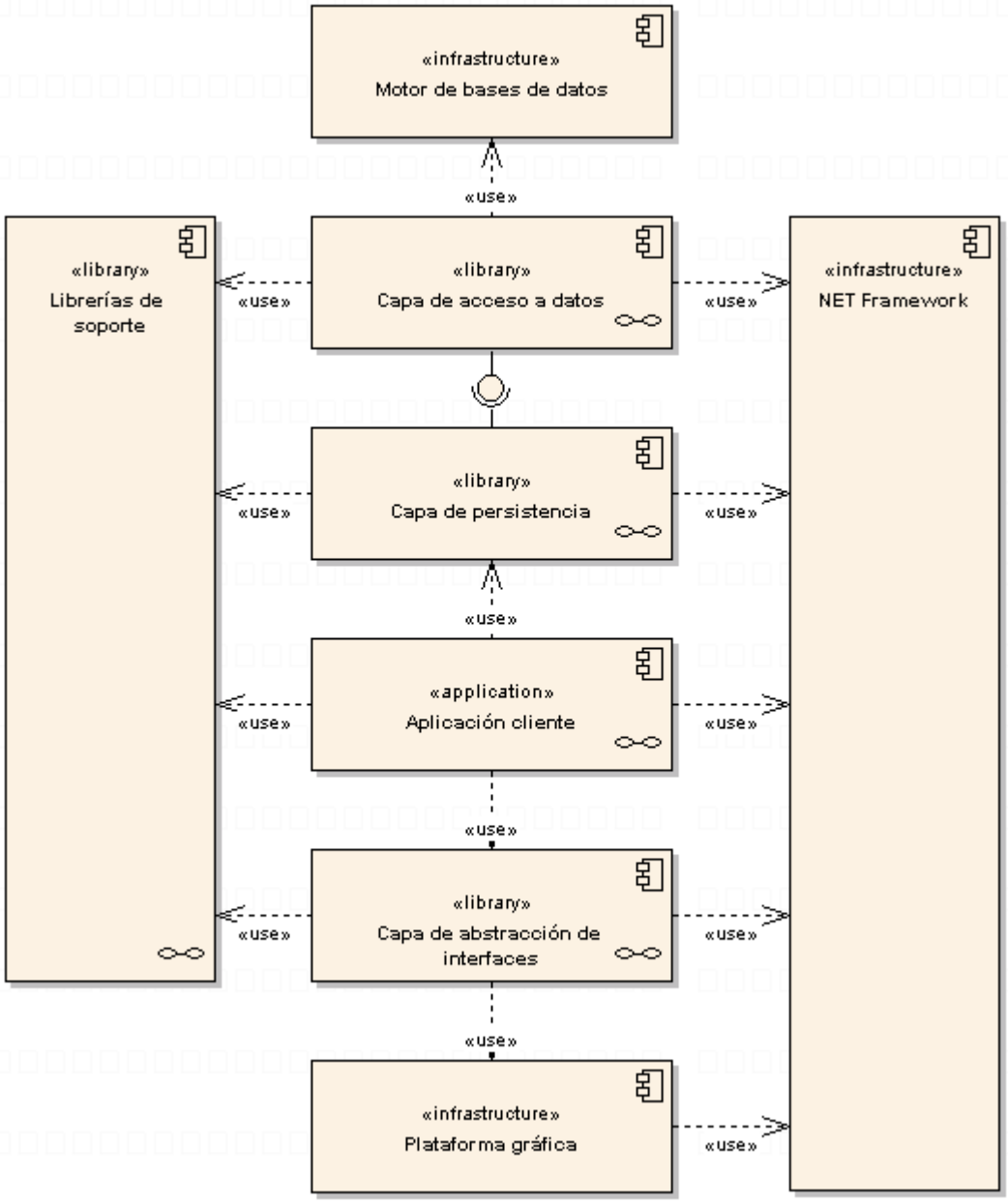


Fig 30. Diagrama de componentes: vista global

Capa de acceso a datos

La capa de acceso a datos tiene como función principal, abstraer la implementación concreta de los distintos controladores de conexión a cada uno de los motores de almacenamiento soportados por el Framework. Para ello, se basa en dos interfaces, una para la implementación de controladores de conexión y la otra para implementación de proveedores de dialecto. Para mantener un bajo nivel de acoplamiento entre los controladores específicos y la capa de acceso a datos, la instanciación de los mismo se realiza dinámicamente, a través de las características de reflexión de la plataforma .NET.

Adicionalmente, la capa de acceso a datos tiene la responsabilidad de proveer control de los límites transaccionales. Para esto se proveen dos métodos posibles, uno explícito, utilizando los métodos de control transaccional provistos por las interfaces de controladores de conexión (IDatastore), y otro implícito, utilizando los servicios de la clase de contextos transaccionales diagramada en el siguiente esquema.

Este componente de control implícito de transacciones deberá basarse en los contextos de cada hilo de ejecución para poder establecer una relación implícita de transacción entre instancias de conexión múltiples.

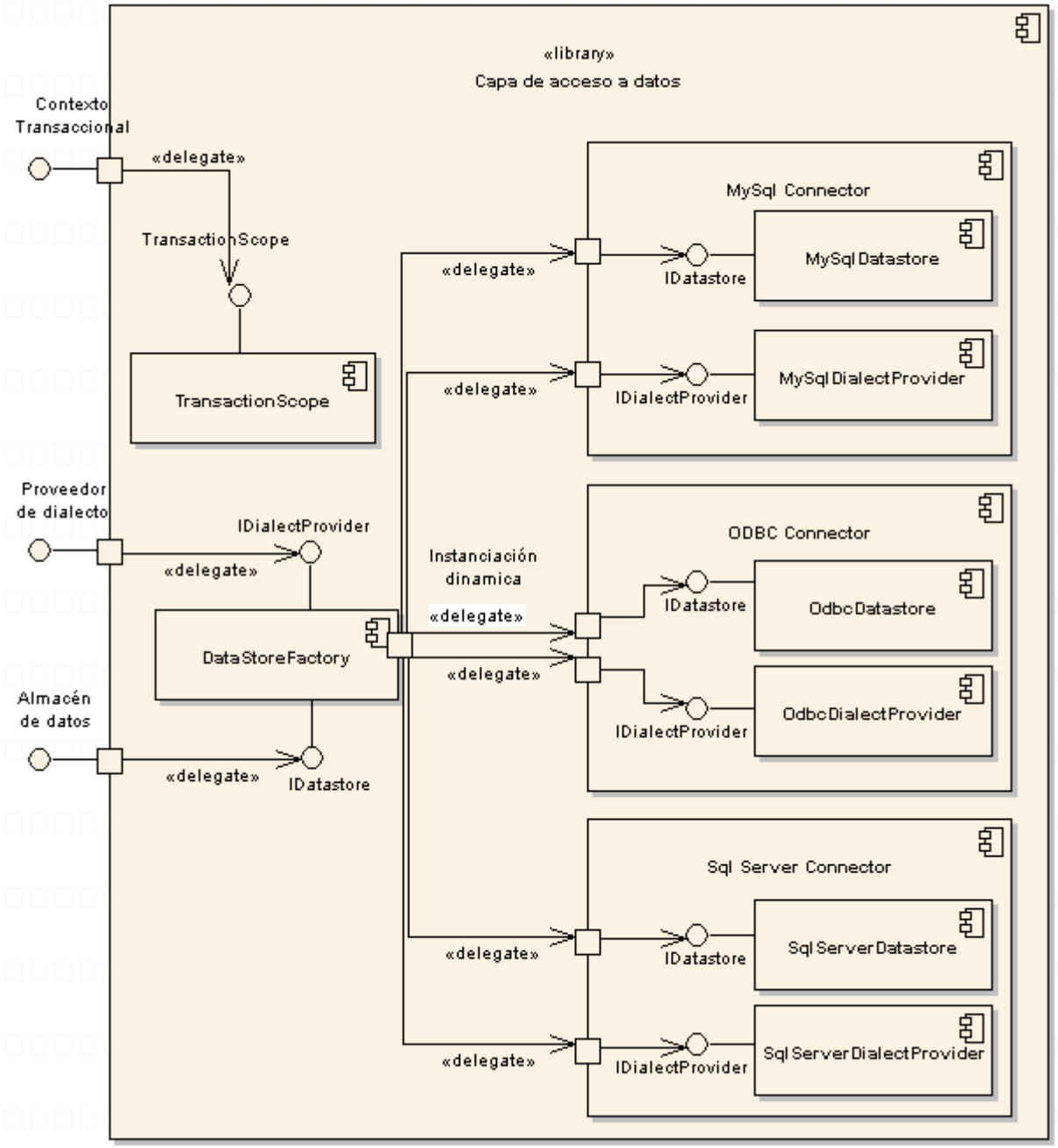


Fig 31. Diagrama de componentes de la capa de acceso a datos

Capa de persistencia

La capa es uno de los componentes fundamentales del presente proyecto. Su funcionalidad básica consiste en proveer servicios de persistencia a objetos de negocios, es decir, permitir a la aplicación cliente la recuperación, eliminación, actualización e inserción (conocido comúnmente como operaciones CRUD) de objetos sin necesidad de conocer el almacén de datos subyacente, ni los lenguajes de consulta y modificación del mismo.

Esta capa se apoya en los servicios provistos por la capa de acceso a datos, y publica una interfase estática para la manipulación de los servicios de persistencia. La misma está basada ampliamente en la capacidad de *Genéricos*, provista por el lenguaje C# 2.0.

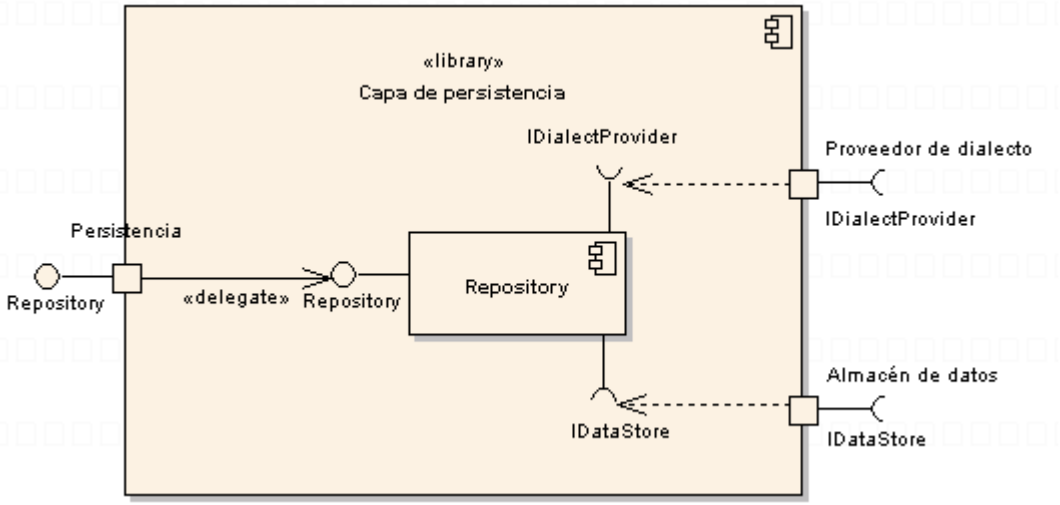


Fig 32. Diagrama de componentes de la capa de persistencia

Capa de abstracción de interfaces

La capa de abstracción de interfaces de usuario es el segundo componente vital del presente proyecto. Su objetivo es permitir la construcción y manipulación simple de interfaces de usuario en múltiples plataformas.

Para el presente proyecto, éste componente se centra en la abstracción de las plataformas Web y Desktop, con lo cual apunta a permitir crear una aplicación que pueda ser ejecutada en cualquiera de éstas plataformas indistintamente, y sin modificaciones de código.

Para alcanzar éste objetivo se vale de archivos de definición de interfaces en un formato de marcado derivado de XML, la construcción dinámica de instancias de motores de visualización, y numerosas técnicas avanzadas como AJAX, para emular un entorno de *cliente grueso* sobre una plataforma Web (*de cliente delgado*).

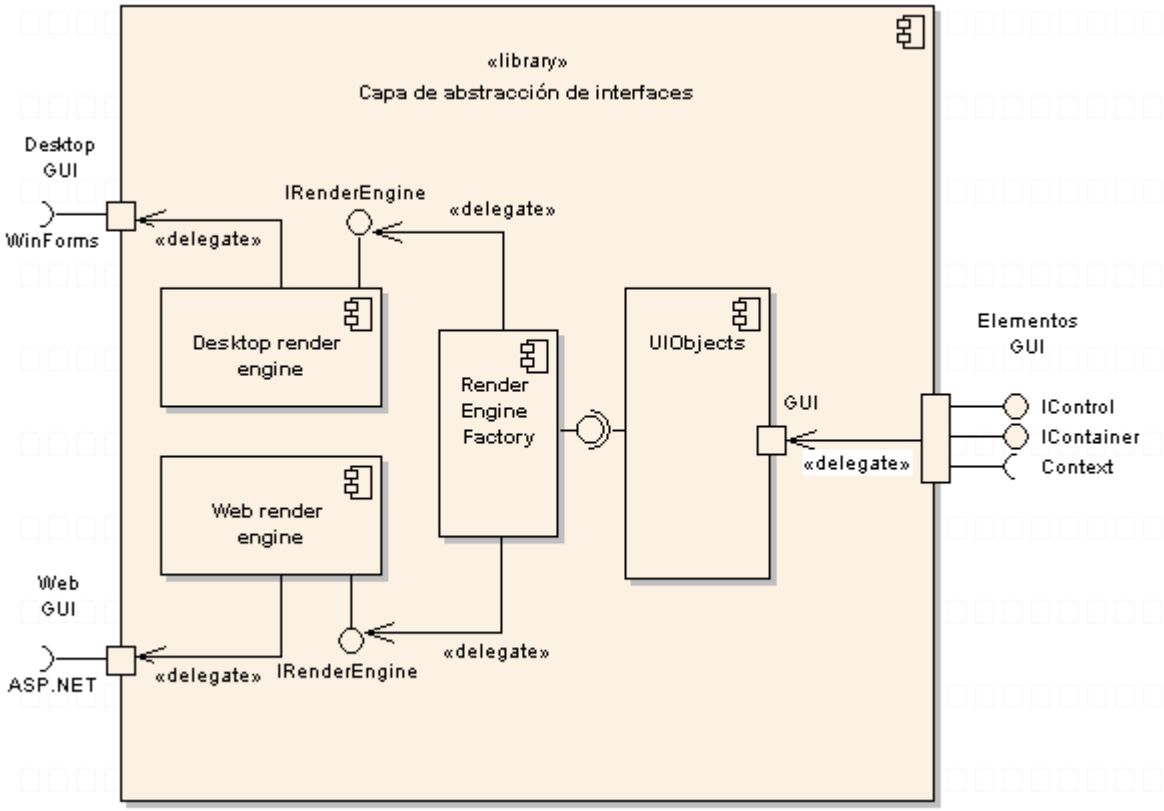


Fig 33. Diagrama de componentes de la capa de abstracción de interfaces

Librerías de soporte

Las librerías de soporte del presente proyecto son también un componente sumamente importante en la arquitectura del desarrollo.

Entre las funcionalidades principales que brindan se encuentran la de los objetos de propiedades. Este concepto, que como se mencionó anteriormente, se origina en el lenguaje de programación Smalltalk, es un elemento muy poderoso que permite generar construcciones semánticas que serían imposibles usando los elementos nativos del lenguaje actual.

Básicamente, lo que implican los “objetos de propiedad” es reemplazar el método de definición de propiedades general, en donde se define una variable miembro privada y se agregan métodos de acceso públicos, por una instancia de objeto de propiedad, que, utilizando las características de *genéricos* del lenguaje, pueden actuar como contenedores de cualquier tipo de datos.

En importante notar que sin el uso de genéricos, la performance del sistema se vería seriamente afectada por la forma en que se opera con los tipos de referencia y los tipos de valor en el Framework .NET. En pocas palabras, la conversión de un tipo de valor (o primitivos) a un tipo de referencia implica una operación de “empaquetado” (*boxing*) y “desempaquetado” (*unboxing*) en donde el objeto se transforma en una referencia a su valor y viceversa, que es muy costoso en términos de performance.

El subsistema de expresiones está fuertemente basado en el concepto de los objetos de propiedad, a partir del cual puede generar una estructura de datos que contiene una expresión no evaluada, que podrá convertirse a otros lenguajes de evaluación en capas inferiores.

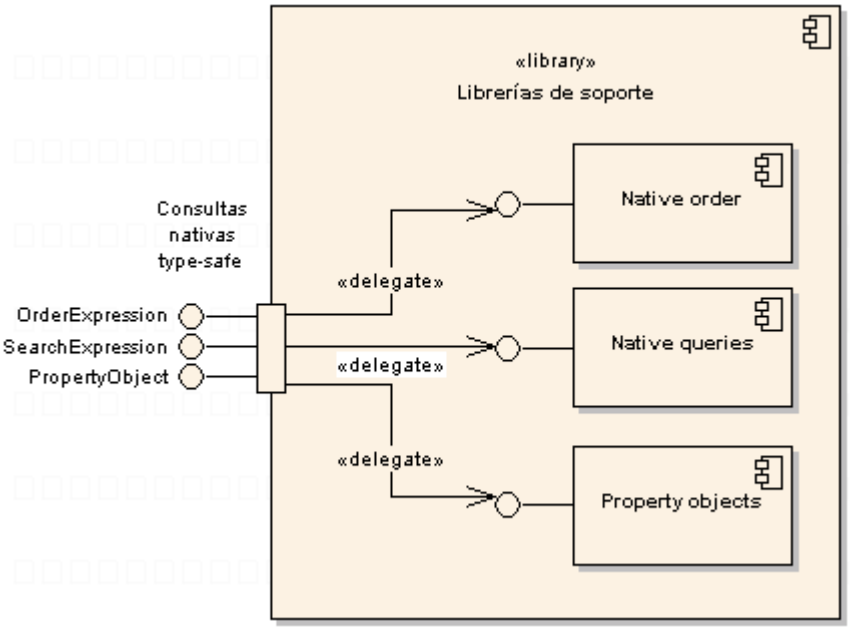


Fig 34. Diagrama de componentes de las librerías de soporte

Aplicación cliente

La aplicación cliente está fuera del alcance del presente proyecto, sin embargo, se incluye el siguiente diagrama a fines aclarar la forma en que ésta se vincula con los servicios provistos por el framework.

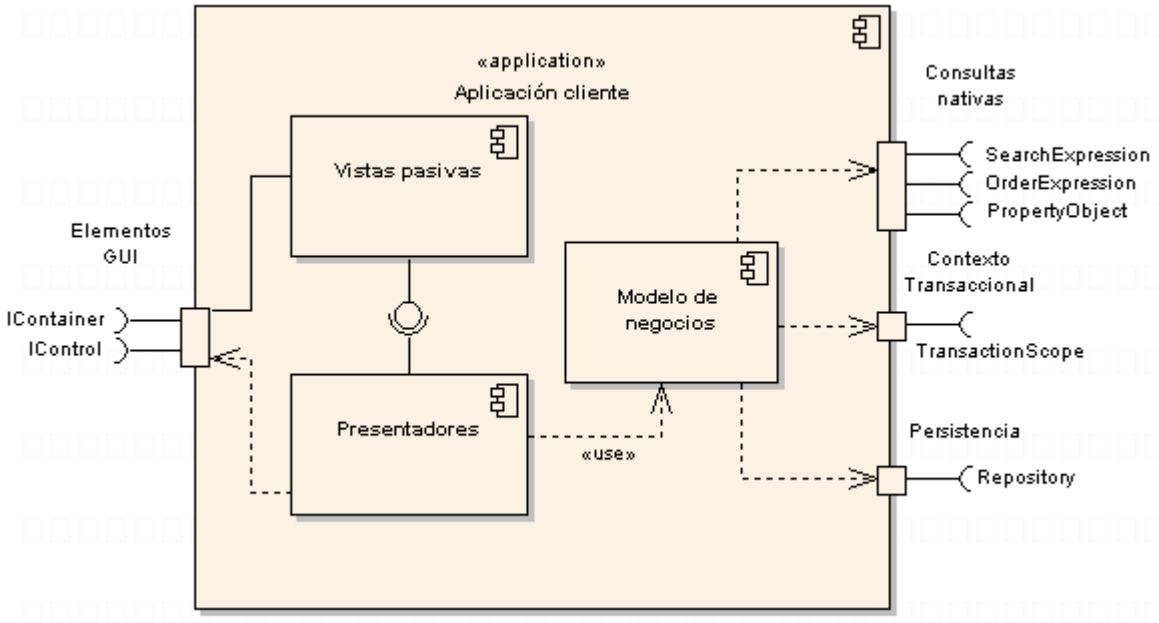


Fig 35. Diagrama de componentes ejemplificando una aplicación cliente

MODELO DE DESPLIEGUE

El modelo de despliegue es una descripción formal de la disposición de los componentes en una instalación normal del producto. De acuerdo a la arquitectura planteada para el proyecto, existen dos escenarios de despliegue primarios: el escenario de despliegue Desktop, y el escenario de despliegue Web. En cualquiera de los escenarios puede utilizarse uno o más almacenes de datos. La disposición de los almacenes de datos no afecta la estructura de despliegue del framework.

Escenario de despliegue Desktop

El escenario de despliegue Desktop es aquel en el que una aplicación desarrollada con el framework se distribuye en los equipos clientes para ejecución local. Casi la totalidad del entorno de ejecución es el equipo del cliente. El almacén de datos puede ser local o remoto.

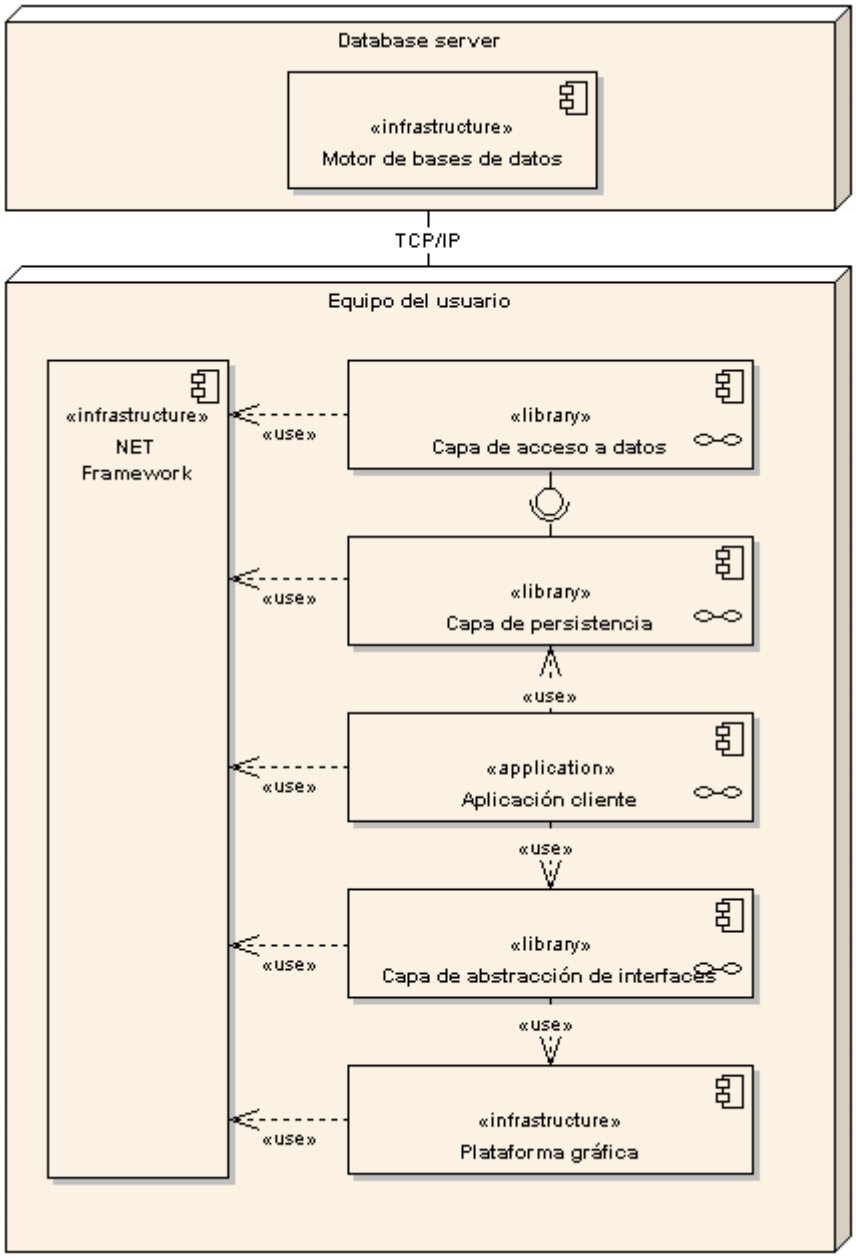


Fig 36. Diagrama de despliegue Desktop

Escenario de despliegue Web

El escenario de despliegue Web es aquel en el que se utilizan las características de la capa de abstracción de interfaces para publicar en la Web una aplicación desarrollada con el framework.

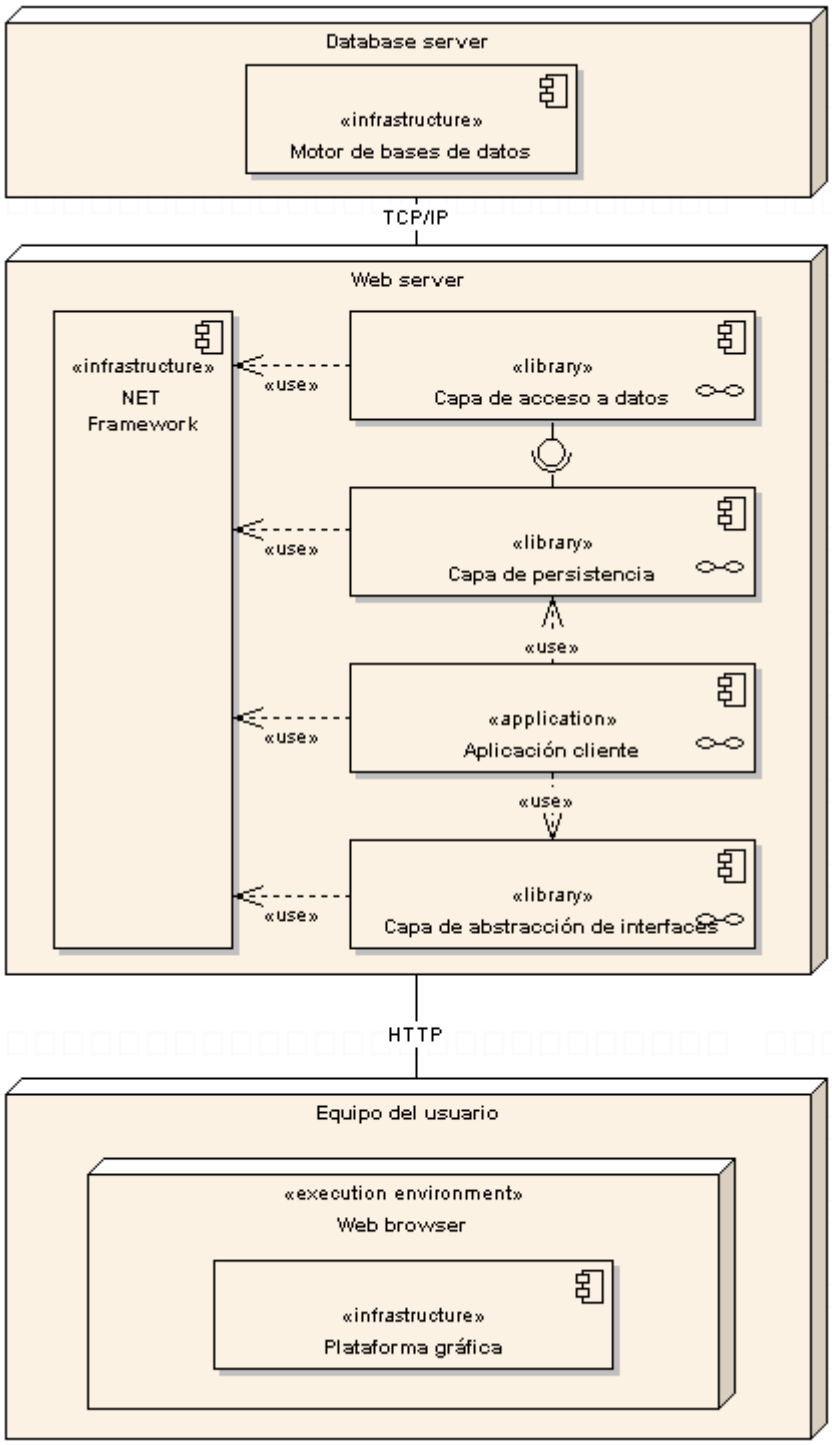


Fig 37. Diagrama de despliegue Web

Este escenario es el que se conoce tradicionalmente como desarrollo en 3-capas, en donde el entorno de ejecución es un cliente liviano (navegador de Internet), la lógica de negocios se ejecuta en un servidor de aplicación, y por último, el almacén de datos se encuentra separado físicamente.

La descripción de los beneficios de éste tipo de instalaciones está fuera del alcance del presente documento, sin embargo, es importante notar que esta última provee un mecanismo más simple a la hora de actualizar la aplicación, y un mayor nivel de control sobre el entorno de seguridad de los almacenes de datos.

Pero la elección entre los distintos modos de despliegue dependerá de las necesidades y circunstancias puntuales de cada proyecto.

Justamente, el objetivo del presente proyecto es proveer la posibilidad de elegir libremente en entre estas alternativas, sin requerir la alteración de la arquitectura de la aplicación.

MATRIZ DE TRAZABILIDAD : COMPONENTES A CASOS DE USO

La presente matriz de trazabilidad complementa el diseño de arquitectura del sistema y forma parte de los entregables definidos en el plan de proyecto.

Su función es permitir rastrear el origen de los cambios y evaluar el impacto de los mismos.

	UC01 - Consultar un conjunto de objetos del almacén de datos	UC02 - Realizar operaciones transaccionales	UC03 - Consultar una estructura de herencia filtrada	UC04 - Consultar una estructura de herencia horizontal	UC05 - Almacenar los datos de un objeto persistente	UC06 - Navegar una relación desde un objeto a sus dependientes	UC07 - Navegar una relación uno a uno	UC08 - Mapear implícitamente una clase	UC09 - Mapear descriptivamente una clase	UC10 - Mapear programáticamente una clase	UC11 - Validar longitud de un atributo	UC12 - Validar un valor de un atributo dentro de un rango	UC13 - Validar la existencia de atributos requeridos	UC14 - Validar un atributo con una expresión regular	UC15 - Validar el código postal de un objeto de negocios	UC16 - Modificar una interface en tiempo de ejecución	UC17 - Construir una interface declarativamente	UC18 - Ejecutar una aplicación en la plataforma Web	UC19 - Ejecutar una aplicación en la plataforma Desktop
Capa de abstracción de interfaces																■	■	■	■
Capa de acceso a datos		■																	
Capa de persistencia	■		■	■	■	■	■	■	■	■									
Desktop render engine																			■
Librerías de soporte	■										■	■	■	■	■				
Native order	■																		
Native query	■																		
Property objects											■	■	■	■	■				
Repository	■		■	■	■														
UIObjects																■			
Web render engine																		■	

ANÁLISIS DE IMPACTO

Uno de los objetivos del presente proyecto es simplificar el proceso de desarrollo al proveer una serie de funcionalidades comunes en la mayoría de las aplicaciones de negocios, principalmente en el área de la persistencia y la gestión de las interfaces de usuario.

Por lo tanto, es factible realizar un análisis sobre el impacto producido por la utilización del Framework en un proyecto de desarrollo de aplicaciones de negocios.

Se deben analizar al menos dos dimensiones de impacto: la dimensión económica y la dimensión técnica.

IMPACTO ECONÓMICO

El impacto económico es difícil de medir con precisión para todos los proyectos en un nivel genérico, ya que los valores exactos serán variables según el tipo y tamaño de proyecto, el equipo de desarrollo y las decisiones arquitectónicas que se tomen durante las etapas de diseño.

Las razones principales que afectan el análisis de impacto económico son las siguientes:

- **Reducción sustancial del tiempo de desarrollo**

  El framework reduce el tiempo de desarrollo al proveer una infraestructura de soporte que proporciona un gran número de funcionalidades. Esta infraestructura minimiza la cantidad de líneas de código que deben ser codificadas manualmente por el equipo de desarrollo.

  Dependiendo de la arquitectura seleccionada para el proyecto y la naturaleza del mismo, la reducción en la cantidad de líneas de código requeridas oscila entre un 20% y un 50%. Esta reducción se traduce directamente en una disminución de los costos de producción.

  En algunos casos, sobre todo en proyectos simples orientados al registro y recuperación de objetos simples, la reducción puede alcanzar valores porcentuales más altos, ya que estas funcionalidades están provistas “de fábrica” por los mecanismos propios del framework.

  Los proyectos más beneficiados por la reducción de código requerido son:

  - Aquellos que utilizan objetos de acceso a datos codificados manualmente, ya que los mismos dejan de ser necesarios. Esta funcionalidad es provista por la capa de persistencia y las características de consultas orientadas a objetos.
  - Aquellos que definen validaciones de negocios, y que deben replicar las validaciones a través de múltiples capas de la aplicación (datos, negocios e interface de usuario).
  - Aquellos que utilizan definiciones de interfaces de usuario codificadas manualmente, ya sea en lenguajes como HTML o a través de la interface de programación provista por la plataforma .NET.

  Para ejemplificar los niveles de reducción en tiempo de desarrollo, podemos asumir que un objeto contiene en promedio entre 175 y 250 líneas de código (Mah, M. & Putnam L., 1998). Una reducción promedio del 35% de código requerido para una aplicación típica compuesta por 50 objetos (~10,000 líneas de código), significa ~3,700 líneas de código menos.

Acorde al informe estadístico producido por META Group (2003) sobre las tendencias y la performance en el sector de tecnología a nivel mundial, la productividad promedio durante el ciclo de un desarrollo nuevo es de ~35,000 líneas de código al año por profesional. Por ende, en el ejemplo planteado, la codificación tomaría 0,30 año o ~79 días, asumiendo un año de 260 días laborales; mientras que el mismo desarrollo utilizando el framework propuesto tendría una duración de 0.20 año o ~51 días.

Es importante notar que la métrica "líneas de código" no es recomendable como medida de productividad en términos generales, solo es utilizada aquí como mecanismo para realizar una evaluación temprana del tamaño y el esfuerzo aproximados de proyectos de desarrollo de software genéricos.

- **Minimiza la necesidad de recursos altamente especializados**

  Al proveer cimientos sólidos sobre los cuales construir aplicaciones, se minimizan los requerimientos de recursos humanos altamente especializados, ya que muchas de las tareas críticas del sistema son controladas por el framework.

  Tradicionalmente, los desarrolladores de software de negocios tienen que estar familiarizados con dos lenguajes como mínimo: un lenguaje imperativo para la construcción del software (Java, C#, etc.), y un lenguaje funcional para la consulta del almacén de datos subyacente (SQL).

  La introducción del framework propuesto elimina la necesidad de conocer el lenguaje de consulta del almacén de datos, de lo cual se derivan las siguientes ventajas:

  - El proyecto puede desarrollarse con personal menos calificado, ya que las habilidades / conocimientos técnicos requeridos son menos. En general, esto se traduce en la posibilidad de reemplazar desarrolladores *Senior*, por desarrolladores *Junior* sin riesgo de reducir la calidad del producto final.
  - Los desarrolladores no tienen que conocer dos paradigmas distintos de programación y pueden concentrarse únicamente en la solución del problema de negocios.
  - Al liberar al equipo de las tareas de conexión y consulta a bases de datos, se reduce o elimina la necesidad de contar con un DBA (Administrador de bases de datos) durante toda la etapa del desarrollo.
  - Los mecanismos de acceso a datos y de control transaccional permiten prescindir de expertos en ADO.NET, COM+ y MS DTC (Microsoft© Distributed Transaction Coordinator).

- **Reduce el tiempo de evaluación y pruebas**

  Existe una consecuencia lateral al uso del framework, que tiene un fuerte impacto positivo en el tiempo total de ejecución de un proyecto: *el tiempo de evaluación y pruebas se reduce notablemente.*

  Las razones son evidentes, al minimizar la cantidad de código requerida para el desarrollo, se obtiene una reducción proporcional en el tiempo requerido para pruebas.

- **Reduce los costos de mantenimiento**

  Un factor importante en la gestión de proyectos de desarrollo de software es el costo de mantenimiento de sistemas. A medida que un sistema incrementa su complejidad, los costos de mantenimiento asociados cada vez son mayores.

  La funcionalidad provista por el framework permite que la complejidad relativa del software se reduzca, ya que el mismo no debe lidiar con la problemática asociada a las particularidades de implementación técnica de las tecnologías subyacentes.

  A nivel de métricas, esta reducción puede observarse en la *complejidad ciclomática* del software, que mide el número de caminos linealmente independientes a través del código fuente. La complejidad ciclomática se computa a través de un grafo que describe el flujo de un programa.

- **Incrementa la productividad de los desarrolladores**

  Debido a las características mencionadas anteriormente, el uso del framework incrementa la productividad de los desarrolladores al permitirles enfocarse en la solución al problema de negocios en lugar de lidiar con detalles de implementación.

IMPACTO TÉCNICO

- **Produce código mejor diseñado**

  El framework provee a los desarrolladores una arquitectura diseñada en capas, y promueve buenas prácticas de programación en múltiples niveles:

  - Implementa las prácticas recomendadas por Microsoft© para el acceso a datos.
  - Implementa un manejo óptimo de transacciones y fomenta el uso adecuado de las mismas a través de una interface simple.
  - Impulsa el diseño en capas a través de los mecanismos de definición de objetos de negocios.
  - Determina la separación de lógica de negocios y de presentación a través de la capa de abstracción de interfaces.

- **Produce código de mayor calidad**

  Al reducir la cantidad de código que los programadores deben escribir manualmente, se minimiza la probabilidad de introducir defectos en la etapa de construcción. Esto resulta en un código que, en promedio, produce mejores métricas de calidad, ya que contiene una menor cantidad de errores.

- **Simplifica el desarrollo**

  Como ya se detalló en puntos anteriores, el uso del framework simplifica el desarrollo al proveer funcionalidad común en las capas de persistencia y de gestión de interfaces de usuario. Adicionalmente, las librerías de soporte proveen los cimientos requeridos por la mayoría de las aplicaciones empresariales.

- **Independencia tecnológica**

  Uno de los impactos técnicos que no es fácilmente cuantificable es la independencia tecnológica que provee la utilización del framework.

  Los dos niveles de independencia más importantes son:

  - *Independencia de almacén de datos:*

    Desarrollar una aplicación que funcione con múltiples bases de datos indistintamente no es una tarea trivial. Requiere de una rigurosa disciplina y recursos humanos altamente capacitados en distintos motores de bases de datos para asegurar la compatibilidad en todos los niveles.

    El framework provee las herramientas necesarias para desarrollar aplicaciones que operen transparentemente con múltiples almacenes de datos, sin requerir esfuerzos adicionales para alcanzar la compatibilidad.

  - *Independencia de plataforma gráfica:*

    La independencia de plataformas gráficas, puntualmente la plataforma web y la plataforma Desktop, es sumamente costosa de alcanzar en un proyecto de desarrollo, ya que implica construir una interface totalmente separada para cada una de las plataformas.

    Lograr emular las capacidades de un cliente grueso (Desktop), en un cliente delgado (navegador de Internet) es un proceso complejo que requiere de expertos en la materia y la utilización de técnicas y tecnologías avanzadas, para homogeneizar los paradigmas de cada una de las plataformas.

- **Propagación controlada de cambios**

  Tal como fuera descrito en la introducción del presente documento, los cambios en un sistema son inevitables y hasta deseables. Sin embargo, el impacto de un cambio puede ser difícil de estimar y controlar dependiendo de la arquitectura de la aplicación. Existen numerosas herramientas para estimar los impactos de un cambio, como por ejemplo, *la matriz de trazabilidad bi-direccional*. Sin embargo, hay pocas herramientas que ayuden a controlar la propagación de un cambio determinado, siendo la más comúnmente utilizada la estructuración en capas de abstracción.

  El framework provee los elementos para construir una aplicación en capas, que controle la propagación de los cambios, al proporcionar un nivel de abstracción completo entre los almacenes de datos y las capas de negocio.

LICENCIAMIENTO

ESTRATEGIA DE LICENCIAMIENTO

La estrategia de licenciamiento prevista para la distribución del producto *Independence Framework* comprende tres objetivos específicos:

- **Promover el uso y la capacitación de la comunidad de desarrolladores**

  Es vital para un producto de éste tipo, lograr una aceptación en la comunidad y generar una base de usuarios capacitados en la utilización del mismo. En socio-dinámica, se denomina *masa crítica* al valor límite en la cantidad de personas necesaria para activar un fenómeno. En éste caso, se requiere una masa crítica de desarrolladores capacitados en el uso del producto para lograr la inserción de éste en los desarrollos empresariales.

- **Apoyar el desarrollo en las comunidades académica y Open Source**

  Sin el apoyo y la capacitación brindadas por las comunidades académica y Open Source, el presente proyecto no hubiera sido posible. Por ende, y a modo de retroalimentación, uno de los objetivos personales del autor es *devolver un poco* a ambas comunidades.

- **Soportar el desarrollo y la evolución del producto**

  Para evitar la obsolescencia del producto es vital mantener activo el desarrollo del mismo, incorporando nuevas técnicas y tecnologías, y optimizando la performance a medida que el sistema madura.

MODELO DE LICENCIAMIENTO

Para la distribución del producto *Independence Framework* se desarrolló un **modelo de licenciamiento dual**. Este tipo de modelo permite cumplimentar con los objetivos estratégicos de licenciamiento planteados, ya que provee un nivel de libertad en cuanto a la disponibilidad y distribución, y a la vez define un mecanismo de sustentación económica del proyecto a largo plazo.

El modelo dual está compuesto por dos licencias diferentes:

1. Una licencia GNU LGPL 2.1[1] (*GNU Lesser General Public License*) destinada a proyectos Open Source y proyectos académicos / educativos. Permite el uso libre del Framework sin imponer condiciones estrictas de distribución sobre el producto derivado.
2. Una licencia del tipo comercial por equipo para el uso en proyectos empresariales con procesos de comercialización de código cerrado.

Los usuarios del framework son libres de elegir el tipo de licencia que utilizarán de acuerdo a las circunstancias propias de cada proyecto.

[1] Las condiciones específicas de la licencia GNU LGPL 2.1 pueden consultarse vía web en http://www.gnu.org/licenses/lgpl.txt o en el archivo de licencia adjunto a la librería.

COSTOS DE LICENCIAMIENTO

Si el usuario decide licenciar el producto bajo GNU LGPL 2.1, entonces, no hay costos de licenciamiento involucrados, siempre y cuando se respeten las condiciones previstas en la licencia.

Por el contrario, si el tipo de proyecto requiere el uso de la licencia comercial, entonces deberá seleccionar la configuración más adecuada a su estructura. El modelo de licenciamiento comercial se basa en la cantidad de equipos en donde será instalado el producto.

Debe notarse la diferencia entre *equipo*, *microprocesador* y *núcleo*. Los grandes servidores utilizan múltiples procesadores en un único equipo, y a su vez, los procesadores más modernos utilizan tecnología de doble y cuádruple núcleo. Sin embargo, el costo por licencia está estipulado a nivel de equipo, por lo que éstos factores no afectan en la elección de la licencia.

Producto	Cantidad de equipos	Precio final
Independence Framework 2007 Professional Edition	1	***$ 599.-***
Independence Framework 2007 Enterprise Edition	Ilimitada	***$ 2999.-***

Los costos están expresados en pesos (ARG), e incluyen el IVA.

Todas las licencias comerciales de *Independence Framework* contemplan un servicio de soporte técnico provisto a través de e-mail, y actualizaciones gratuitas durante el período de un año a partir de la fecha de compra.

La edición profesional está diseñada para desarrolladores independientes que proveen sistemas de negocios para uso profesional, que serán utilizados en computadoras personales de escritorio, generalmente por un único usuario.

En cambio, la edición empresarial apunta a empresas que desean utilizar el producto en instalaciones multi-sitio sin restricciones sobre el número de equipos instalados.

ANÁLISIS DE COSTOS

Basándose en el análisis de impacto presentado en las páginas anteriores, es posible elaborar un análisis de costos sobre un desarrollo promedio y determinar así las ventajas económicas de utilizar el framework en comparación con un desarrollo similar que no lo utilice.

Es importante notar que existen ventajas no económicas del uso del framework, que fueron detalladas en la sección de Impacto técnico.

Para el análisis de costos, se utilizaron los siguientes parámetros:

Parámetro	Valor	Detalle
Cantidad de objetos en el sistema	50	Es un valor estimado para una aplicación de negocios pequeña / mediana.
LOC promedio por objeto[2]	212.5	Líneas de código en promedio por objeto del sistema.
LOC promedio por año por profesional[3]	35,000	Líneas de código por año por profesional en nuevos desarrollos.
Días laborables por año	260	Cantidad aproximada de días laborables por año.
Reducción promedio	35%	La reducción de código promedio provista por el framework oscila entre 20% y 50%.
Honorarios en promedio de un administrador de base de datos[4]	$ 2,000	Los horarios de un DBA oscilan entre $1,500.- y $2,500.-
Honorarios en promedio de un analista Senior	$ 2,000	Los horarios de un analista Senior oscilan entre $1,500.- y $2,500.-
Honorarios en promedio de un analista Junior	$ 1,500	Los horarios de un analista Junior oscilan entre $1,000.- y $2,000.-

Se estima que para completar el desarrollo descrito sin utilizar el framework, se requerirán los servicios de un analista Senior y un administrador de base de datos durante toda la etapa de codificación. Mientras que, utilizando el framework, se pueden obtener los mismos resultados con un equipo de dos analistas Junior. Cabe destacar que estos estimados se aplican únicamente a la etapa de codificación.

	LOC del sistema	Esfuerzo en años	Esfuerzo en días	Recursos asignados	Costo mensual por recurso	Costos totales
Sin Framework	10625	0.30	78.93	2	$ 2,000.-	**$ 14,571.43.-**
Con Framework	6906	0.20	51.30	2	$ 1,500.-	**$ 7,103.57.-**
Ahorro						**$ 7,467.86.-**

El beneficio económico total durante la fase de construcción / codificación de un proyecto de desarrollo como el planteado es de **$ 7,467.86.-**, lo que significa un **ahorro de más del 50%** del presupuesto destinado a los recursos profesionales.

[2] Los objetos desarrollados en C++, Java y similares contienen en promedio entre 175 y 250 líneas de código (Mah, M. & Putnam L., 1998).

[3] De acuerdo al *Worldwide IT Benchmark Report*, producido por META Group, Inc. (2003), a nivel mundial un desarrollador produce un promedio de 35,000 líneas de código por año en nuevos desarrollos.

[4] Los valores de honorarios fueron obtenidos de las *Referencias de Honorarios* publicadas en la página Web del Consejo Profesional de Ciencias Informáticas de la Provincia de Córdoba, http://www.cpcipc.org/honorarios.asp, Octubre de 2006.

En fases subsiguientes existe también un beneficio potencial, que varía de acuerdo a la estrategia de verificación y pruebas (Testing) que utilice la empresa.

COMPARATIVA EN LÍNEAS DE CÓDIGO

El siguiente gráfico describe la reducción promedio en líneas de código, lo cual se traduce en un sistema más fácil de mantener, y un beneficio económico concreto durante la etapa de desarrollo, según lo demostrado en el punto anterior.

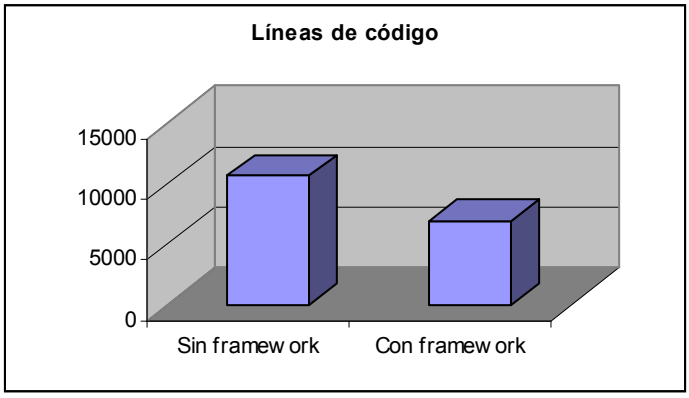


Fig 38. Gráfico comparativo de LOC

COMPARATIVA DEL ESFUERZO EN DÍAS

El diagrama a continuación detalla la reducción del esfuerzo requerido en tiempo de codificación.

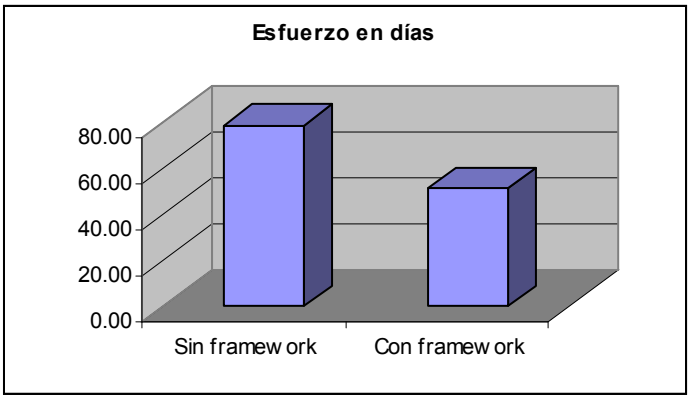


Fig 39. Gráfico comparativo de esfuerzo en días

COMPARATIVA DE COSTOS TOTALES

En base a la información planteada en el análisis de costos, se deriva el siguiente gráfico comparativo de costos entre un proyecto utilizando el framework propuesto y otro no utilizándolo.

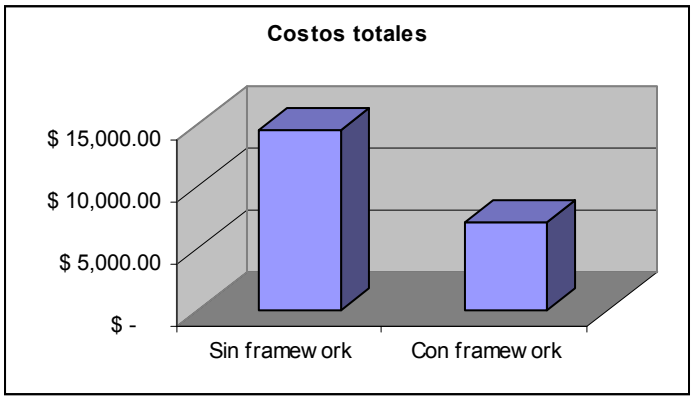


Fig 40. Gráfico comparativo de costos totales

RECUPERACIÓN DE LA INVERSIÓN

Utilizando la información de análisis de costos y los costos de licenciamiento en el modelo de licencias comerciales para empresas se calcula el período de tiempo requerido para recuperar la inversión económica inicial.

Uno de los elementos considerados a la hora de determinar el precio final de las licencias de éste proyecto fue alcanzar un valor que permitiera a una empresa recuperar el costo de licenciamiento con la ejecución de un solo proyecto mediano, y aún así mantener un margen de ganancia.

El análisis de recuperación de la inversión es calculado en base a la licencia empresarial, que tiene un valor final de **$ 2,999.00.-**, y el ejemplo de análisis de costos, donde el costo total de la etapa de construcción asciende a **$ 7,103.57.-.**

Suponiendo que el proyecto ejemplificado es el primero en utilizar el framework dentro de la empresa, debemos incorporar el costo de licencia a los costos totales del desarrollo, con lo cual éste asciende a **$ 10,102.57.-**. Aún así, el costo total sigue estando por debajo del costo esperado para un proyecto similar sin la utilización del framework.

El ahorro provisto para el primer proyecto de la inversión es de **$ 4,468.86.-**. Con lo cual se establece que el framework tiene un tiempo de repago menor al tiempo de ejecución previsto para un proyecto tipo, y aún así, provee un ahorro del 31% sobre la opción de desarrollo tradicional.

	Licencia	Costo total sin framework	Costo total con framework	Ahorro total	% de Ahorro
Primer proyecto	$ 2,999.-	$ 14,571.43.-	$ 10,102.57.-	**$ 4,468.86.-**	**31%**
Proyectos subsiguientes	$ 0.00.-	$ 14,571.43.-	$ 7,103.57.-	**$ 7,467.86.-**	**51%**

CONSIDERACIONES FINALES

El desarrollo del presente Trabajo Final de Grado fue, sin lugar a dudas, un desafío importante y una experiencia de aprendizaje sumamente valiosa.

A nivel profesional, la investigación llevada a cabo para poder desarrollar las fundamentaciones teóricas, construir el proyecto propuesto y elaborar una defensa extensa sobre el mismo, me permitió extender mis conocimientos sobre arquitecturas de sistemas de información, patrones de diseño y algoritmos para optimización de performance, así como familiarizarme con nuevas técnicas y tecnologías de análisis, diseño y programación.

A su vez, pude poner en práctica el uso de metodologías de modelado UML y aplicar los conceptos de gestión de proyectos y costos estudiados durante el cursado de la carrera.

A nivel interpersonal, me permitió explorar y mejorar mis habilidades para exponer y defender mis ideas, y me otorgó una visión más clara de los procesos de la comunicación involucrados en el campo de la informática.

En un nivel pragmático, la conclusión de éste proyecto me permitió concretar numerosas ideas sueltas que había concebido durante los últimos años, así como explorar nuevos campos tecnológicos.

Con una mirada hacia el futuro, creo que las experiencias vividas a lo largo de éste proyecto ha moldeado mi carrera, así como mis intereses y mis objetivos profesionales.

Agradezco a todos los que me acompañaron durante éste recorrido, familiares, amigos y docentes...

REFERENCIAS BIBLIOGRÁFICAS

- AMBLER, S., 2003. *Agile Database Techniques*. New Jersey, USA: Wiley. ISBN: 0471202835.
- APPLETON, B. et at, 1998. *Streamed Lines: Branching Patterns for Parallel Software Development* [online]. Disponible en: http://www.cmcrossroads.com/bradapp/acme/branching/patterns.html. En: APPLETON, B. et at, 2002. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration.* Boston, USA: Addison-Wesley Professional. ISBN: 0201741172.
- BORGES DE BARROS PEREIRA, H., 2002. *Análisis experimental de los criterios de evaluación de usabilidad de aplicaciones multimedia en entornos de educación y formación a distancia*. Tesis doctoral. Universitat Politècnica de Catalunya
- BOUNDS, N.M. & DART S.A., 1993. *Configuration Management Plans: The Beginning of Your CM Solution* [online]. Pittsburgh, USA, SEI Carnegie-Mellon University. Disponible en: ftp://ftp.sei.cmu.edu/pub/case-env/config_mgt/papers/cm_plans.pdf
- CARR, M.J. et al, 1993. *Taxonomy-Based Risk Identification* [online]. Pittsburgh, USA, SEI Carnegie-Mellon University. Disponible en: http://www.sei.cmu.edu/pub/documents/93.reports/pdf/tr06.93.pdf
- CONSTRUX SOFTWARE BUILDERS, INC., 2002. *CxOne Standard: Source Code and Construction*. Washington, USA: Construx Software Builders, Inc.
- CORCOS D., 2000. El modelo espiral. *Reportes Técnicos en Ingeniería de Software* [online], 1 (2). Disponible en: http://www.itba.edu.ar/capis/rtis/articulosdeloscuadernosetapaprevia/CORCOS-ESPIRAL.pdf
- POPKIN SOFTWARE AND SYSTEMS, [n.d]. *Modelado de Sistemas con UML* [online]. Irvine, USA: Telelogic North America, Inc. Disponible en: http://es.tldp.org/Tutoriales/doc-modelado-sistemas-UML/single-html/doc-modelado-sistemas-uml.pdf
- CHAMBERS & ASSOCIATES, 1997. *C&A Glossary* [online]. Australia: Chambers & Associates Pty Ltd. Disponible en: http://www.chambers.com.au/glossary/glossary.htm
- DART S.A., 2004. *Concepts in Configuration Management Systems* [online]. Pittsburgh, USA, SEI Carnegie-Mellon University. Disponible en: ftp://ftp.sei.cmu.edu/pub/case-env/config_mgt/papers/cm_concepts.pdf
- ESTEVES J. et al, 2005. Implementación y Mejora del Método de Gestión de Riesgos del SEI en un proyecto universitario de desarrollo de software. *Revista IEEE America Latina* [online], 3 (1). Disponible en: http://www.ewh.ieee.org/reg/9/etrans/Marzo2005/paper119.pdf
- FAYAD M. & SCHMIDT D.C, 1997. *Special Issue on Object-Oriented Application Frameworks.* ACM Communications [online], 40 (10). Disponible en: http://www.cs.wustl.edu/~schmidt/CACM-frameworks.html
- FOWLER, M., 1997. *Analysis Patterns: Reusable Object Models.* Boston, USA: Addison-Wesley Professional. ISBN: 0201895420.
- FOWLER, M., 2002. *Patterns of Enterprise Application Architecture*. Boston, USA: Addison-Wesley Professional. ISBN: 0321127420.
- FOWLER M., 2006. *GUI Architectures* [online]. Chicago, USA: ThoughtWorks, Inc. Disponible en: http://www.martinfowler.com/eaaDev/uiArchs.html
- HOLLAND M., 2004. *How to: Cite References* [online]. Poole, UK, Bournemouth University. Disponible en: http://www.bournemouth.ac.uk/academic_services/documents/Library/Citing_References.pdf
- JOHNSON R.E., 1998. *How to design Frameworks* [online]. Software Architecture Group University of Illinois. Disponible en: http://st-www.cs.uiuc.edu/users/johnson/cs497/notes98/day18.pdf
- MAH, M. & PUTNAM L., 1998. Software by the numbers: An aerial view of the software metrics landscape. *American Programmer* [online], 10 (11). Disponible en: http://www.qsm.com/aerialview.html

- McCONNELL, S., 1993. *Code Complete*. Redmond, USA: Microsoft Press. ISBN: 1556154844.
- McCONNELL, S., 2004. *Code Complete 2*. Redmond, USA: Microsoft Press. ISBN: 0735619670.
- META GROUP, 2003. *Worldwide IT Benchmark Report*. Connecticut, USA: Meta Group, Inc.
- PREE, W., 1994. *Meta Patterns: A Means For Capturing the Essentials of Reusable Object-Oriented Design* [online]. Austria, University Linz. Disponible en: http://www.exciton.cs.rice.edu/comp410/frameworks/Pree/C010.pdf
- PRESSMAN, R.S., 2002. *Ingeniería de Software: Un Enfoque Práctico*. 5ta ed. McGraw-Hill. ISBN: 8448132149.
- REIMER, J., 2005. *A History of the GUI* [online]. Canada: Ars Technica, LLC. Disponible en: http://arstechnica.com/articles/paedia/gui.ars/1

REFERENCIAS A WIKIPEDIA

Para las referencias bibliográficas a *Wikipedia, La enciclopedia libre*, se utiliza el estilo ISO de citas bibliográficas. Debido a la controversia relativa a la confiabilidad de ésta fuente, solo fue utilizada para obtener referencias y definiciones concretas que no planteen polémica sobre su veracidad.

- Colaboradores de Wikipedia. *Lenguaje Unificado de Modelado* [online]. Wikipedia, La enciclopedia libre, 2006 [fecha de consulta: 11 de agosto del 2006]. Disponible en: http://es.wikipedia.org/w/index.php?title=Lenguaje_Unificado_de_Modelado&oldid=4244755
- Colaboradores de Wikipedia. *Desarrollo en espiral* [online]. Wikipedia, La enciclopedia libre, 2006 [fecha de consulta: 11 de julio del 2006]. Disponible en http://es.wikipedia.org/w/index.php?title=Desarrollo_en_espiral&oldid=3861020
- Colaboradores de Wikipedia. *Framework* [online]. Wikipedia, La enciclopedia libre, 2006 [fecha de consulta: 8 de agosto del 2006]. Disponible en: http://es.wikipedia.org/w/index.php?title=Framework&oldid=4206255
- Colaboradores de Wikipedia. *Plain Old Java Object* [online]. Wikipedia, La enciclopedia libre, 2006 [fecha de consulta: 15 de agosto del 2006]. Disponible en: http://en.wikipedia.org/w/index.php?title=Plain_Old_Java_Object&oldid=67433235
- Colaboradores de Wikipedia. *Common Language Runtime* [online]. Wikipedia, La enciclopedia libre, 2006 [fecha de consulta: 15 de agosto del 2006]. Disponible en: http://en.wikipedia.org/w/index.php?title=Common_Language_Runtime&oldid=66581118
- Colaboradores de Wikipedia. *Herramienta CASE* [online]. Wikipedia, La enciclopedia libre, 2006 [fecha de consulta: 20 de agosto del 2006]. Disponible en: http://es.wikipedia.org/w/index.php?title=Herramienta_CASE&oldid=4298953
- Colaboradores de Wikipedia. *Trazabilidad* [online]. Wikipedia, La enciclopedia libre, 2006 [fecha de consulta: 20 de agosto del 2006]. Disponible en: http://es.wikipedia.org/w/index.php?title=Trazabilidad&oldid=3666766
- Colaboradores de Wikipedia. *WIMP (computing)* [online]. Wikipedia, La enciclopedia libre, 2006 [fecha de consulta: 21 de agosto del 2006]. Disponible en: http://en.wikipedia.org/w/index.php?title=WIMP_%28computing%29&oldid=65636891
- Colaboradores de Wikipedia. *Cyclomatic complexity* [online]. Wikipedia, La enciclopedia libre, 2006 [fecha de consulta: 20 de octubre del 2006]. Disponible en: http://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=82551676

ANEXOS

ANEXO I - ESTÁNDAR DE CODIFICACIÓN (C#)

INTRODUCCIÓN

El presente anexo, denominado Estándar de codificación (C#) contiene un conjunto de reglas de construcción y codificación que son aplicables al programar en lenguaje C# de la plataforma .NET, aunque también aplican, en cierto grado, a otros lenguajes y / o tecnologías. Tanto como sea posible, todas las reglas y lineamientos deben ser utilizados en todos los lenguajes que intervienen en el desarrollo de una solución. Obviamente, esto no siempre es posible, y a veces las reglas específicas del lenguaje sobrescribirán explícitamente las reglas generales de éste estándar.

Los contenidos de éste están basados en el *CxOne Standard Source Code And Construction*. Muchas de las reglas en éste anexo, así como en el *CxOne Standard* fueron tomadas del libro *Code Complete* de Steve McConnell (1993).

DEFINICIONES

Esta sección lista las abreviaciones, términos y conceptos utilizados dentro de este anexo.

- **Pascal Casing:** Es la práctica de capitalizar la primera letra de cada palabra en un identificador largo formado por varias palabras.
  Ejemplo: `ThisIsPascalCasing`.

- **Camel Casing:** Es la práctica de capitalizar la primera letra de cada palabra en un identificador largo formado por varias palabras, con excepción de la primera letra del mismo, que permanece en minúscula.
  Ejemplo: `thisIsCamelCasing`.

ESTÁNDAR DE CODIFICACIÓN

CONVENCIÓN DE NOMBRES

- Todos los nombres deben ser claros y descriptivos. Esto incluye variables, archivos, funciones, clases, directorios, etc. No limitar artificialmente la longitud de nombres salvo que la tecnología lo demande.
- Utilice Pascal Casing cuando incluya siglas de tres caracteres o más en nombres de funciones, variables, etc. Si la sigla es de dos caracteres, utilice todas mayúsculas.
  Ejemplo: `XmlDocument`, `UIControl`.

CLASES & FUNCIONES

- Utilice el método verbo-sustantivo para nombrar funciones que realizan operaciones sobre un objeto.
  Ejemplo: `CalcularDistancia()`.
- Utilice Pascal Casing para nombres de clases y funciones.
  Ejemplo: `GetDocument()`.

- Anteceda la declaración de las clases y funciones con un comentario indicando su propósito, para esto utilice las características de documentación XML de C#, indicando como mínimo, el sumario de la misma.
- Todas las sobrecargas de una función deberían realizar tareas similares.
- En C# y otros lenguajes orientados a objetos, es redundante incluir el nombre de la clase en el nombre de las propiedades, como `Libro.TituloDelLibro`. En su lugar utilice: `Libro.Titulo`.

VARIABLES

- Utilice nombres de variables descriptivos. No utilice abreviaturas. Un nombre de variable debe ser legible, preciso, y describir su propósito sin ambigüedades. No use variables de un carácter salvo iteradores en ciclos cortos `for`.
- Utilice Camel Casing para nombre de variables.
  Ejemplo: `string mensajeSaliente`.
- Nunca utilice números mágicos dentro del código, aún cuando se utiliza una sola vez, siempre declare un nombre simbólico para una constante numérica.

COMENTARIOS

- Comente mientras codifica, porque seguramente no tendrá tiempo para hacerlo después.
- Utilice la característica de documentación XML en C#.
- Cuando modifique código, siempre actualice los comentarios existentes.
- Al comienzo de cada función, es útil proveer una breve introducción que explique porque existe y que puede hacer.
- Evite colocar comentarios de final de línea, este tipo de comentarios hacen mas difícil la lectura. Sin embargo, los comentarios de fin de línea son apropiados para comentar la declaración de variable, en cuyo caso, deberán estar todos alineados en el mismo nivel.
- Utilice frases completas en los comentarios, y mantenga la puntuación y capitalización de las palabras. No abrevie, ni resuma la información más de lo comprensible.
- Utilice comentarios para describir la intención del programador, o proveer un resumen de una sección de código.
- No use cometarios para escribir que hace el código, sino para explicar porqué lo hace.
- Indente los comentarios al mismo nivel de indentación que el código al cual se refieren.
- Utilice el marcador estándar `TODO:` (Del inglés, *to do:* por hacer) para indicar que una sección de código está incompleta y debe ser revisada.

FORMATO

- Si utiliza código de terceros y no existe un estilo discernible en el código existente, formatee el código de acuerdo al estándar de construcción.
- Si existe un estilo discernible pero difiere del estándar, considere reformatear el código de acuerdo al estándar.
- En lo posible, mantenga la longitud de las líneas por debajo de los 80 caracteres.
- Siempre indente el código siguiendo las construcciones lógicas.
- Para la indentación utilice TABS, no espacios. En la mayoría de los entornos de desarrollo el indentado con TAB puede configurarse para mostrarse de mayor o menor tamaño. Configure éste valor en 4.
- Use líneas libremente para separar y ordenar el código. El código debe ser armónico para su lectura. Mantenga un balance visual que permita interpretar rápidamente el código.

- En lo posible, inserte 1 línea blanca dentro de funciones, 2 líneas blancas para separar dos funciones juntas, y 3 o 4 líneas blancas para separar secciones dentro del archivo.
- Coloque un espacio antes y después de cada operador (+,-,==, etc.)
  Ejemplo: `if ( 1 == 3 )`
- Coloque un espacio después de cada coma separando parámetros en la definición e invocación de una función.
  Ejemplo: `funcionUno ( int param1, string param2 )`
- Mantenga las llaves de apertura en la misma línea, y las de cierre alineadas con la línea de apertura:
```
for (i = 0; i < 100; i++) {
   …
}
```
- Utilice espacios en blanco para organizar el código. Esto crea párrafos de código, que ayudan al lector a comprender la segmentación lógica del mismo.

ESTRUCTURAS DE CONTROL

- Utilice nombres descriptivos para índices de loops en lo posible.
  Ejemplo: `for ( int cliente = 0; cliente < array.length; cliente++ )`
- No utilice instrucciones `goto`.

ARCHIVOS Y DIRECTORIOS

- Si utiliza herramientas para generar código que será extendido o mantenido, formatee el código de acuerdo a éste estándar de construcción.
- Los nombres de archivos y directorios deben contener únicamente caracteres alfanuméricos.
- Utilice Pascal Casing para nombre de archivos y directorios: la primer letra de cada palabra capitalizada.
  Ejemplo: `AssemblyInfo.cs`

PLANTILLA DE DESARROLLO EN C#

La siguiente es una plantilla para el desarrollo de nuevas clases en lenguaje C#, donde se ejemplifican algunas de las reglas de éste estándar.

```
using System;

namespace Organizacion.Sistema.Modulo {

        /// <summary>
        /// Descripcion del propósito de la clase.
        /// </summary>
        public class NombreDeLaClase {

                /// <summary>
                /// Descripción del constructor.
                /// </summary>
                public NombreDeLaClase( ) {
                }

                /// <summary>
                /// Descripción del propósito del método.
                /// </summary>
                /// <param name="param1">Descripcion del parametro.</param>
                /// <returns>Descripcion del valor de retorno</returns>
                public string MetodoUno( int param1 ) {
                }

        }
}
```