UNIVERSIDAD POLITÉCNICA DE CARTAGENA

ESCUELA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Estudio de herramientas de desarrollo de *software* basado en modelos: MDA y Factorías de *Software*

AUTOR
Ramón García Soto

DIRECTOR
José Juan Sánchez Manzanares

Septiembre/2006

Autor	Ramón García Soto
E-mail del Autor	ramongar@eresmas.com
Director	José Juan Sánchez Manzanares
E-mail del Director	pepe.manzanares@si.upct.es
Codirector(es)	
Título del PFC	Estudio de herramientas de desarrollo de *software* basado en modelos: MDA y Factorías de *Software*
Descriptores	

Resumen. En este documento se plasma la realización de un estudio de mercado de herramientas de desarrollo *software* de última generación para la realización de aplicaciones *web*, orientadas a la gestión completa de todas las fases del ciclo de vida del producto.

En primer lugar, se introducen los modelos de ciclo de vida y metodologías más comunes para dar paso, a continuación, al estudio de los modelos de desarrollo más novedosos. Tras el repaso de las metodologías se realiza un estudio de mercado de herramientas, para finalmente ofrecer una solución que cubra una serie de criterios propuestos.

Con el conjunto de herramientas seleccionado, se ha desarrollado una aplicación a modo de ejemplo.

Titulación	Ingeniería Superior de Telecomunicación
Intensificación	
Departamento	
Fecha de Presentación	Septiembre - 2006

ÍNDICE

1. Introducción.

El producto *software*, como cualquier otro producto, pasa por una fase de incubación y desarrollo antes de llegar a ser un producto maduro y listo para su explotación. Son sus características tecnológicas y de desarrollo especiales las que lo diferencian de los proyectos ingenieriles clásicos. Al ser también la ingeniería del *software* una ciencia relativamente nueva, hasta hace pocos años no se disponían de unos pasos estandarizados para la creación de *software*. Aún hoy en día no se puede hablar de un estándar cerrado para la metodología de desarrollo *software* que compartan todas las empresas públicas y privadas.

También es interesante tener en cuenta que se ha pasado de unos pocos lenguajes de programación estructurados, bases de datos simples y comunicaciones entre procesos simples a muchos lenguajes de programación (estructurados, orientados a objetos, orientados a aspectos,...), a complejas bases de datos (relacionales, orientadas a objetos,...) y a arquitecturas distribuidas sofisticadas (Java-RMI, CORBA, DCOM, SOA...). Esta variedad de tecnologías y su posible unión para llegar a una solución en proyectos de gran magnitud complica en gran medida el desarrollo de aplicaciones.

Por otro lado, el tamaño incremental de los proyectos *software* ha requerido un aumento de los recursos necesarios para el desarrollo y así ha aumentado la dificultad y el número de los componentes de los equipos de trabajo. Esto requiere el uso de herramientas adicionales para el control del cumplimiento de los objetivos y que estos se cumplan con una cierta calidad. Hoy en día, no es posible hablar de proyectos *software* en los que intervienen cientos de personas sin unas herramientas de apoyo que ayuden a la gestión integral del desarrollo del *software* en todas sus etapas. Dichas herramientas velan por el cumplimiento de los requisitos impuestos al inicio del proyecto, además de contemplar sus futuras variaciones durante el desarrollo, disminuyen el tiempo de desarrollo del proyecto mediante una mejor coordinación y mejoran la calidad final del producto. Este último punto es quizás el más importante, ya que mejorando la calidad final del proyecto se disminuye el número de errores del producto final y minimiza el mantenimiento de la aplicación, que es la fase donde el coste de la eliminación de errores es mayor.

Por todo esto, se sigue apreciando la evolución de la ingeniería del *software* que comienza a pasar de su fase de desarrollo artesanal a la de industrialización. Ya no basta con contar con un editor de código y un compilador, y se comienza a perfilar el siguiente salto de calidad en este campo de la ingeniería.

2. Motivaciones y objetivos.

La motivación del desarrollo de este proyecto ha sido encontrar un entorno de desarrollo estable, duradero y ampliable para la implementación de aplicaciones *web* dentro del Servicio de Informática de la Universidad Politécnica de Cartagena.

El entorno de desarrollo debe permitir el uso de una tecnología de desarrollo duradera, ya que en los últimos proyectos desarrollados no se ha seguido una línea única. Véase el ejemplo de dos de los últimos proyectos desarrollados, como son EBUP o DUMBO. En EBUP se optó por una solución basada en *Java Server Pages* (JSP) y en el proyecto DUMBO se optó por tecnologías de última generación como son eX*tensible Stylesheet Language* (XSL), *eXtensible Markup Language* (XML) e Hibernate. Esta falta de homogeneidad en la línea de desarrollo conlleva un aprendizaje y adaptación del grupo de trabajo que interviene a nuevas tecnologías y, en algunos casos, a nuevos entornos de desarrollo.

Como se acaba de comentar, la línea tecnológica actual que se sigue se basa en la utilización de XSL, XML e Hibernate, que es la solución por la que se optó en DUMBO. Respecto al entorno de desarrollo, se ha estado trabajando con un entorno basado en la Plataforma Eclipse [1] y un servidor de versiones.

Esta elección tecnológica seleccionada para el proyecto DUMBO disminuye la dificultad del desarrollo de aplicaciones *web*, siguiendo así un modelo de aplicación en tres capas. Se separa la presentación de la lógica de negocio, y la lógica de negocio del modelo de persistencia de datos.

El principal objetivo de este proyecto será que el entorno de desarrollo finalmente elegido pueda cubrir eficientemente el ciclo de vida de los productos *software* a desarrollar y, que en la medida de lo posible, mejore la línea tecnológica seguida por el Servicio de Informática. Se pretende implantar un entorno que permita integrar un control de versiones, y que permita integrar alguna solución para la persistencia de objetos JAVA mediante bases de datos estructuradas (por ejemplo, Hibernate o *Enterprise Java Beans*).

Para poder conseguir el objetivo principal, se va a realizar en principio una breve introducción al ciclo de vida del producto *software* [2][3] y sus modelos más interesantes. Tras estudiar los modelos de ciclo de vida, se revisarán algunas de las metodologías más populares dentro del mundo del *software*, ya que aportan conceptos interesantes sobre el desarrollo de aplicaciones, formas de trabajo en equipo y otros aspectos menos técnicos y más a nivel organizativo, que no aparecen en los modelos de ciclo de vida (apartado 3).

También se revisarán en el documento los modelos de desarrollo *software* que más fuerza han tomado en los últimos años (MDA y Factorías de *software*), proponiéndose como el siguiente salto de calidad en el mundo del *software* (apartado 4).

Una vez conocidos los nuevos paradigmas, se debe plantear un modelo que será el que se intente cubrir mediante herramientas *software* integradas entre sí (apartado 5), y se realizará un estudio comparativo entre las herramientas más relevantes (apartado 6 y 7) para finalmente llegar a una solución que cumpla los objetivos propuestos (apartado 8).

Con la solución desprendida del estudio comparativo se desarrollará un proyecto a modo de ejemplo (apartado 9) y, finalmente, se presentarán las conclusiones desprendidas del documento (apartado 10).

3. El ciclo de desarrollo del *software*.

Las fases más importantes que aparecen en todos los modelos son: planificación y toma de requisitos, análisis y especificaciones, diseño, implementación, pruebas unitarias, pruebas de integración y mantenimiento. En la actualidad, existe un esfuerzo por todas las compañías de la industria del *software* para proveer de herramientas integrables para todas y cada una de las fases del ciclo de desarrollo. También se intentan crear lenguajes estándar para la definición de cada una de las fases, véase por ejemplo *Unified Modeling Language* (UML) para toma de requisitos, análisis y diseño, o *Tree and Tabular Combined Notation* (TTNC) para pruebas. Más adelante se verá que estas herramientas y lenguajes también nos ayudan a adecuar las fases de desarrollo mediante un modelado algo distinto al modelado clásico de proyectos.

En los siguientes apartados se abordará una visión general de las fases del ciclo de desarrollo del *software*, algunas de las distintas metodologías propuestas y herramientas para la gestión integral del desarrollo del *software* en todas sus fases.

3.1. Fases del ciclo de desarrollo.

Normalmente las fases dentro del ciclo de desarrollo del *software* están claras, lo que no está tan claro en muchos casos es el modelado que se realiza a partir de dichas fases. En una primera aproximación, el modelado propuesto podría ser el siguiente:

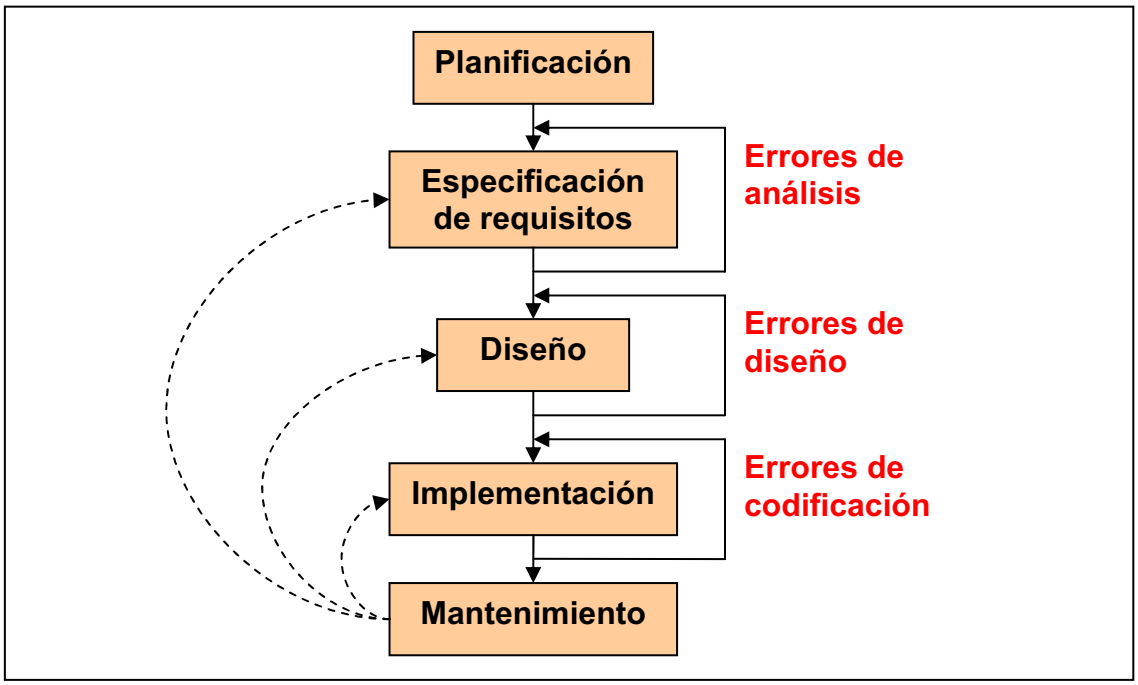


Figura 1. Modelo de desarrollo en cascada.

Es el llamado **modelo en cascada** (modelo clásico). En primer lugar se parte de una planificación del proyecto dentro de la estructura de la organización teniendo en cuenta los recursos disponibles, riesgos y beneficios económicos. Tras la aceptación de la primera fase, se pasa a una especificación de los requisitos en las que interviene la organización y el cliente para el que se desarrolla el producto, aunque en muchos casos el cliente es la propia organización. A partir de los requisitos se plantea un diseño y sobre dicho diseño se realiza una implementación, que da lugar a un producto sobre el cuál se realizará un mantenimiento. Si en cualquier fase se detecta un error se vuelve a la fase en la que se produjo y se subsana. El modelo propuesto parece simple y rápido, pero es poco realista. Puede servir para proyectos pequeños pero no para proyectos a gran escala.

Los errores no se suelen detectan durante las fases de diseño e implementación (lo que es el desarrollo del producto en sí), si no que tardan mucho más tiempo en detectarse. Esto es debido a que en medianos y grandes proyectos existen diferentes equipos de desarrolladores que implementan módulos de forma separada que se comunicarán en un futuro por interfaces definidos previamente. La mayoría de los errores provienen de la integración de dichos módulos y no se detectan hasta las fases de pruebas. Esto produce una repetición continua de dichas fases, por lo que dentro del ciclo de vida se llegan a repetir varias veces las distintas fases. Una aproximación más realista al modelo en cascada sería la siguiente:

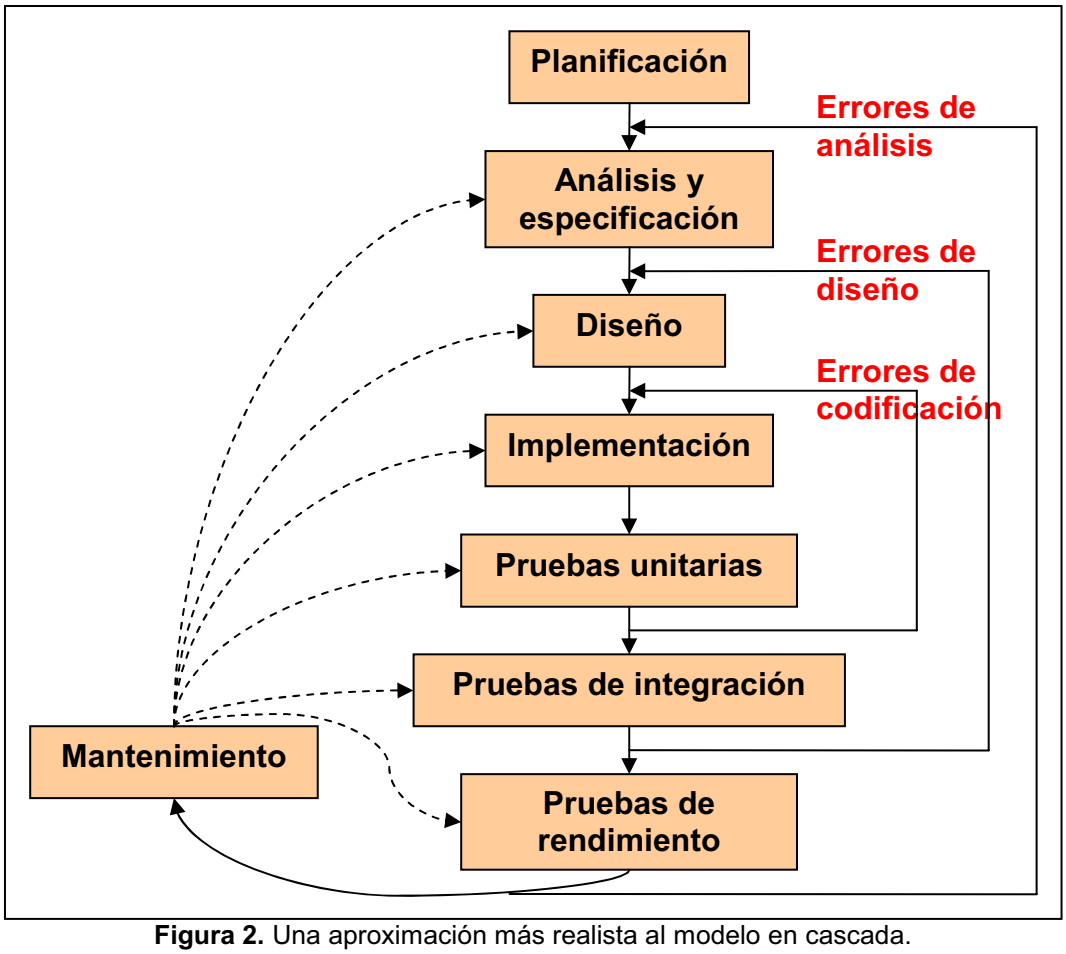


Figura 2. Una aproximación más realista al modelo en cascada.

El coste económico, temporal y de recursos aumenta según avanzamos en las fases del ciclo de vida. Este es uno de los motivos mencionados anteriormente que nos llevan a la utilización de herramientas que minimicen el riesgo del proyecto mediante la reducción sistemática del número de errores. El modelo en cascada presentado no es el más apropiado para el desarrollo *software*, aunque es el modelo de desarrollo más generalizado.

El modelo en cascada mejora el modelo clásico al ser más realista, pero introduce inconvenientes como el efecto "bola de nieve", por el que los errores se propagan a las etapas siguientes. Conforme avanza el proyecto, el modelo tiende a deformarse y el *software* tiende a deteriorarse y resulta cada vez más difícil de mantener.

Existen variaciones del modelo en cascada que ayudan a mitigar los efectos negativos recién comentados. Uno de ellos es el **modelo en cascada con prototipo desechable**.

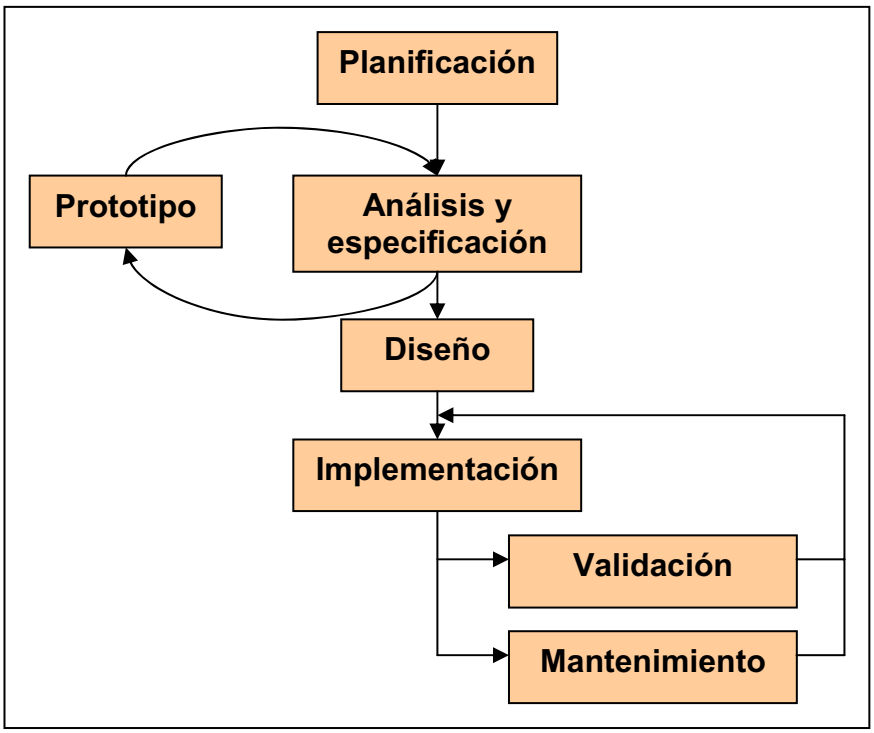


Figura 3. Modelo en casada con prototipo desechable

En esta variación del modelo en cascada, no se entra en la fase de diseño hasta que el prototipo ha sido completamente validado por medio de una serie de análisis y especificaciones. Una vez validado el prototipo, éste se desecha, se pasa a la fase de diseño y se sigue con un diseño en cascada.

El modelo introduce mejoras como la mitigación del efecto "bola de nieve", aunque se presentan inconvenientes como que el usuario tome demasiado en cuenta el prototipo (no se asume que no es una versión ni robusta ni completa) y la inversión necesaria para realizar el mismo.

Se debe de tener en cuenta que, a veces, la inversión en un prototipo puede no ser rentable, además que se pueden perder oportunidades de mercado debido al tiempo invertido en construirlo.

Una variación sobre el modelo anterior sería el modelo de **programación automática**. Pretende introducir un alto grado de automatización en el proceso de desarrollo del *software*.

La idea base de este modelo es la programación por transformaciones, que se basa en la construcción de una primera versión que expresa formalmente el comportamiento deseado. A partir de dicha versión, se realiza una transformación en una versión más eficiente, preservando la funcionalidad introducida de forma inicial. Lo que se pretende es unir totalmente la fase de especificación y de creación de prototipo, generando de forma automática el prototipo a partir de la especificación.

Para la especificación, se utilizan lenguajes de especificación formal. De esta forma, todas las acciones de validación y mantenimiento recaen directamente sobre la especificación, y el producto se obtiene a través de un proceso mecánico de transformación (a priori rápido, seguro y de bajo coste).

Como cabe esperar, para este modelo se requieren herramientas altamente especializadas para el desarrollo del producto por medio de transformaciones durante todo el ciclo de vida del producto. Veremos algunas de las herramientas aportadas para ello a lo largo del documento, analizando las facilidades y el grado de automatización que ofrecen.

En muchos casos interesa utilizar una combinación de un modelo de programación automática y un modelo manual, ya que, a veces, la complejidad del proyecto dificulta la posibilidad de crear una especificación completa (La dificultad de especificar es equivalente a la dificultad de programación del sistema). Para este tipo de modelos híbridos se debe tener en cuenta que los cambios en las distintas fases de implementación (modelado e implementación manual), no provoquen estados inconsistentes. La capacidad de que un cambio a un nivel no afecte al estado de otro nivel se llama consistencia incremental.

La filosofía de la programación automática queda plasmada claramente en las herramientas MDD/MDA, en las que se centrará el documento más adelante.

Para terminar, siguiendo con las variaciones del modelo en cascada, cabe destacar el **Modelo en V**. Desarrollado por el Ministerio de Defensa de Alemania, supone una evolución del modelo en cascada, incluyendo mejoras en la fase de pruebas.

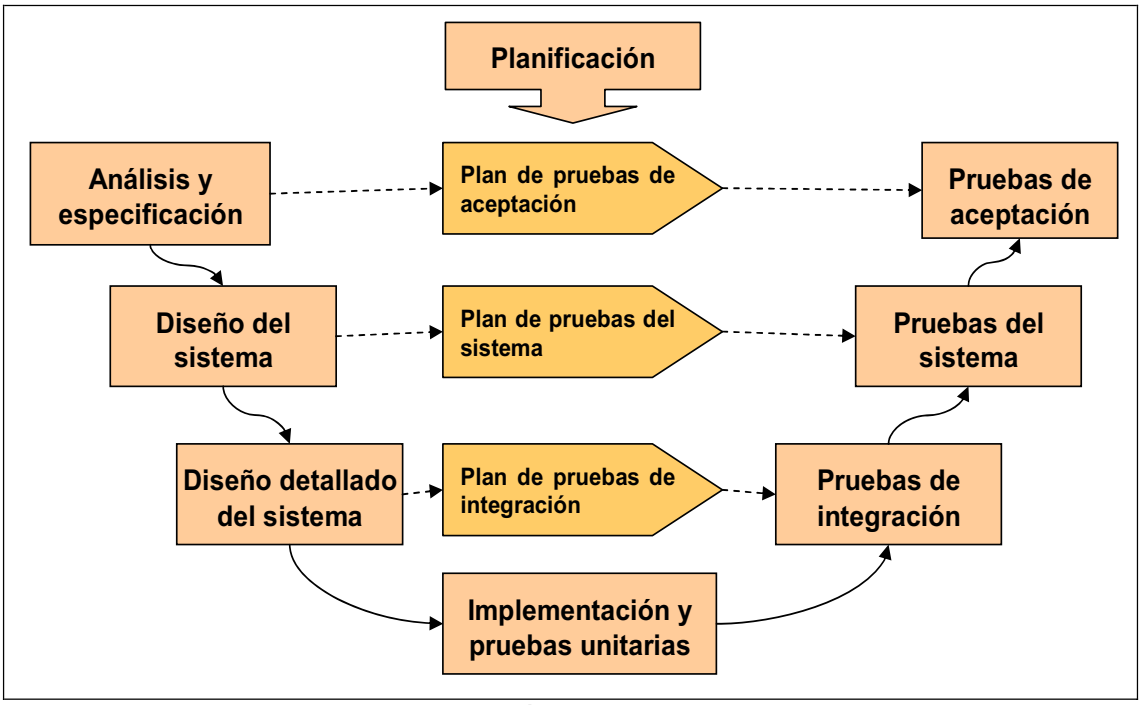


Figura 4. Modelo en V.

Puede notarse que su primera mitad es similar al Modelo en Cascada, y la otra mitad tiene como finalidad hacer pruebas asociadas a cada una de las etapas de la mitad anterior. Se puede identificar una ventaja principal con respecto al Modelo Cascada más simple, y se refiere a que este modelo introduce pruebas de cada una de las etapas del modelo de cascada.

Dentro de sus principales desventajas:

- Mayor riesgo que en otros modelos, ya que las pruebas se efectúan antes de haber terminado la implementación. Esto puede tener como consecuencia un *roll-back* de todo el proceso.
- El modelo no contempla la posibilidad de retorno a etapas anteriores, situación bastante probable.
- El problema se afronta por completo desde un principio, en vez de realizar un proceso iterativo incremental.

También existen otros modelos como el incremental, el modelo en espiral, el modelo de ensamblaje de componentes,... de los cuales no se va a realizar una explicación.

Antes de proponer un modelo, se va a pasar a realizar un breve análisis sobre metodologías *software*. Este nuevo análisis enriquecerá el del ciclo de vida del producto y aportará un modelo más eficiente.

3.2. Metodologías del software.

Las metodologías del *software* son importantes ya que ofrecen a las organizaciones un instrumento útil para la sistematización de las actividades que dan soporte al ciclo de vida del *software*. Además, sin un estudio previo de las metodologías no se podrían tener claros conceptos como la gestión de proyectos, gestión de la configuración, control de versiones o gestión de la calidad.

La **gestión de proyectos** tiene como finalidad principal la planificación, seguimiento y control de las actividades y de los recursos humanos y materiales que intervienen en el desarrollo de un producto.

La **gestión de la configuración** consiste en la aplicación de procedimientos administrativos y técnicos durante el desarrollo del producto y su posterior mantenimiento. Debe de identificar, definir, proporcionar información y controlar los cambios en la configuración del producto, así como las modificaciones y versiones de los mismos. Permite conocer el estado de cada uno de los productos que se hayan definido como elementos de la configuración, garantizando que no se realizan cambios incontrolados y que todos los participantes del sistema disponen de una versión correcta del mismo.

Por último, el **aseguramiento de la calidad** proporciona un marco común de referencia para la definición y puesta en marcha de planes específicos de aseguramiento de calidad aplicables a proyectos concretos. En este apartado también se encuadran los distintos de pruebas que se realizan sobre el producto para asegurar que cumple con los requisitos funcionales y no funcionales impuestos. Se aplican distintos tipos de pruebas como las funcionales, las de rendimiento, las de regresión o la búsqueda de defectos en la implementación.

Las metodologías *software* más extendidas en la actualidad son ***Rational Unified Process*** (RUP) [4] y ***Capability Maturity Model Integration*** (CMMI) [5]. Además, se debe de hablar de otras metodologías como pueden ser ***IT Infraestructure Library*** (ITIL) [6], para servicios TI, o **METRICA Versión 3** [7], para el caso de administraciones públicas españolas. Estas son las metodologías en las que se va a centrar este apartado.

3.2.1. Rational Unified Process (RUP).

El principal aporte de RUP es la visión que ofrece del ciclo de vida del producto. Aunque en principio divide el ciclo de vida en cuatro fases (Inicio, elaboración, construcción y transición), estas fases se encuentran bajo dos disciplinas que ejecutan dichas fases de forma iterativa. Esta repetición de las fases tiene como origen la corrección de errores mencionada al comentar el modelo en cascada (apartado 3.1).

La primera de dichas disciplinas es la llamada disciplina de desarrollo y consta de las fases de modelado de negocio, captura de requisitos, análisis y diseño, implementación, pruebas y despliegue.

Por otro lado, se tiene la disciplina de soporte, que tiene en cuenta gestión de la configuración y administración del cambio, gestión del proyecto y ambiente de desarrollo. O sea, se tiene menos en cuenta lo que es el desarrollo propiamente dicho y más en cuenta las "buenas prácticas" a la hora de desarrollar.

Otro aspecto que tiene en cuenta RUP es la percepción del riesgo asociado al desarrollo del producto en cada una de sus fases. Se tiene en cuenta que las dos primeras fases del ciclo de vida (Inicio y elaboración), son las fases que mayor riesgo implican, pudiendo ser decisivas en el éxito o fracaso del proyecto.

En la figura presentada a continuación se muestran las distintas fases y la importancia de cada una de las disciplinas en ellas.

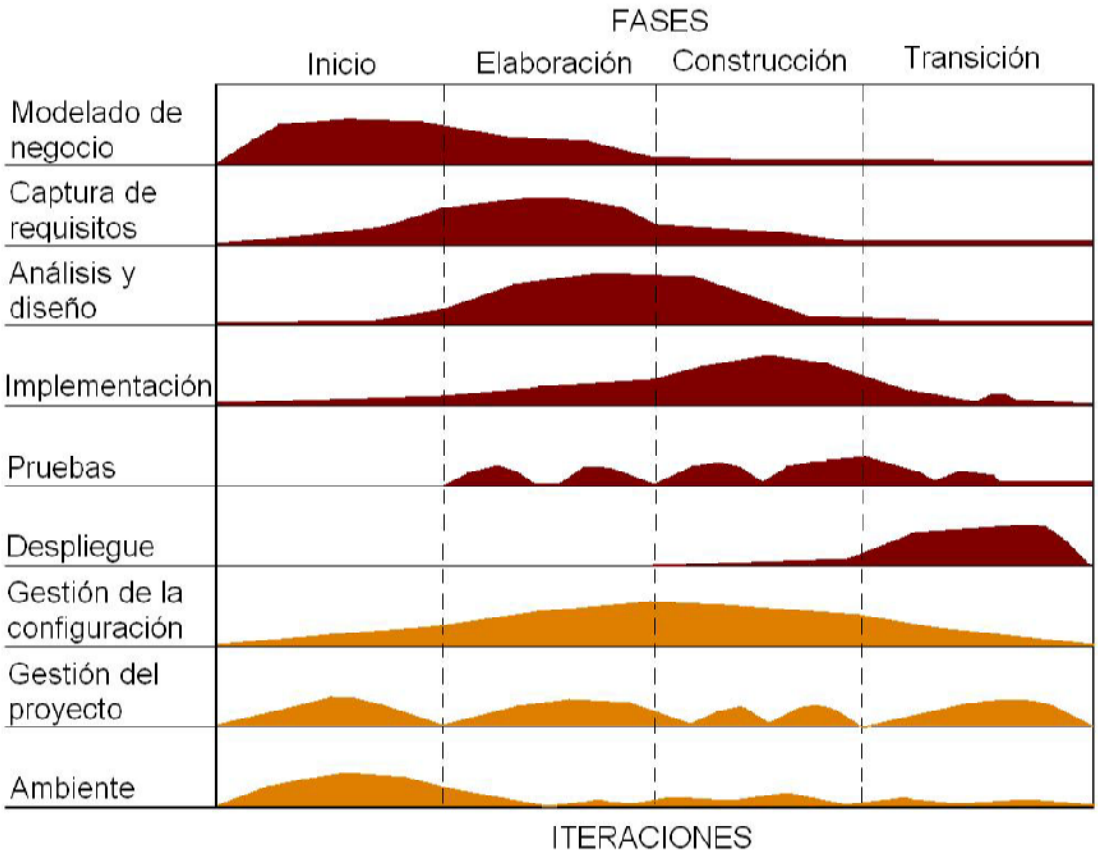


Figura 5. Fases *Rational Unified Process (RUP)*.

De la figura 5 se desprende un dato importante, y es la importancia de la disciplina de soporte en todas las fases del ciclo de vida, destacando la gestión de proyectos como un soporte fundamental en todas. Otro dato importante que se desprende del modelo propuesto es la presencia de las pruebas en todas las fases de implementación (elaboración y construcción), tal y como se propone en los modelos del apartado 3.1.

3.2.2. Capability Maturity Model Integration (CMMI).

CMMI es una metodología más orientada a la organización, y no a lo que es el desarrollo en sí. Propone un modelo para la mejora y evaluación de los procesos de desarrollo y mantenimiento de sistemas y productos *software*. Dicho modelo se subdivide en dos, un modelo orientado a *software* y un modelo orientado a ingeniería de sistemas.

El modelo **CMMI para *software*** (CMM-SW) establece 5 niveles de madurez para clasificar a las organizaciones, en función de qué áreas de procesos consiguen sus objetivos y se gestionan con principios de ingeniería. Es lo que se denomina un "modelo escalonado", o centrado en la madurez de la organización. Los 5 niveles de madurez propuestos son:

- **Inicial o Nivel 1.** Este es el nivel en donde están todas las empresas que no aplican procesos. Los presupuestos se disparan y no es posible entregar el proyecto en las fechas previstas. No hay un control sobre el estado del proyecto y el desarrollo de este es completamente opaco.
- **Repetible o Nivel 2.** El éxito de los resultados obtenidos se pueden repetir. La principal diferencia con el nivel anterior es que el proyecto es gestionado y controlado durante el desarrollo del mismo. El desarrollo es más transparente y es posible controlar el estado.
- **Definido o Nivel 3.** Para alcanzar este nivel se debe tener una definición clara de la forma de desarrollar proyectos (gestión e ingeniería). Se deben aplicar métricas durante el desarrollo del proyecto para obtener datos estadísticos que ayuden a controlar el estado, se debe realizar una documentación continua y realizar un control continuo sobre el proyecto.
- **Cuantitativamente gestionado o Nivel 4.** Los proyectos usan objetivos ponderables para alcanzar las necesidades de los clientes y de la organización. Las métricas se aplican tanto en el campo del desarrollo de proyectos como en la gestión de la organización.
- **Optimizado o Nivel 5.** Los procesos de los proyectos y de la organización están orientados a la mejora de las actividades. Mejoras incrementales e innovadoras de los procesos que mediante métricas son identificadas, evaluadas y puestas en práctica.

Una ampliación del modelo CMM-SW es el modelo **CMMI para la ingeniería de sistemas** (CMM-SE). En este modelo se establecen 6 niveles posibles de capacidad para cada una de las 18 áreas de proceso implicadas en la ingeniería de sistemas. No se agrupan los procesos en 5 tramos para definir el nivel de madurez de la organización, sino que directamente analiza la capacidad de cada proceso por separado. Es lo que se denomina un "modelo continuo". En el caso de la versión que integra desarrollo de *software* e ingeniería de sistemas, las áreas aumentan a 22, y en el caso que cubra también la integración de productos, aumentan a 25. Estas áreas son las mostradas en la siguiente tabla.

Área de proceso	Categoría
Análisis y resolución de problemas	Soporte
Gestión de la configuración	Sopote
Análisis y resolución de decisiones	Soporte
Gestión integral de proyecto	Gestión de proyectos
Gestión integral de proveedores	Gestión de proyectos
Medición y análisis	Soporte
Entorno organizativo para integración	Soporte
Innovación y desarrollo	Gestión de procesos
Definición de procesos	Gestión de procesos
Procesos orientados a la organización	Gestión de procesos
Rendimiento de los procesos de la organización	Gestión de procesos
Formación	Gestión de procesos
Integración de producto	Ingeniería
Monitorización y control de proyecto	Gestión de proyectos
Planificación de proyecto	Gestión de proyectos
Gestión calidad procesos y productos	Soporte
Gestión cuantitativa de proyectos	Gestión de proyectos
Desarrollo de requisitos	Ingeniería
Gestión de requisitos	Ingeniería
Gestión de riesgos	Gestión de proyectos
Gestión y acuerdo con proveedores	Gestión de proyectos
Solución técnica	Ingeniería
Validación	Ingeniería
Verificación	Ingeniería

Tabla 1. Áreas de proceso CMMI.

De entre esas 25 áreas, se destacan las áreas de gestión de la configuración, gestión integral de proyectos, integración de producto, desarrollo de requisitos, validación y verificación.

A mediados de este año 2006, se acaban de producir cambios en CMMI tras la publicación de CMMI versión 1.2, denominada "CMMI *for Development*". Aquí ha comenzado un proceso de diferenciación con una versión paralela denominada CMMI para servicios, que verá la luz en 2007. CMMI *for Development* engloba a CMMI-SE/SW, y es orientada al desarrollo de productos *software*.

3.2.3. IT *Infraestructure Library* (ITIL).

ITIL es un estándar de facto en el área de la gestión del servicio. La metodología ITIL propone pautas de "buenas prácticas" y describen más bien el "qué" que el "cómo", ayudando a cumplir con los requerimientos del negocio respetando costes acordados. Aunque estas pautas estén orientadas a la

gestión del servicio, como ya se ha comentado, se pueden trasladar al campo del desarrollo del *software* como lo que son, un conjunto de “buenas prácticas”.

Las propuestas de ITIL abarcan muchos campos dentro de la gestión del servicio, aunque se van a comentar los más interesantes desde el punto de vista del desarrollo *software*. Estas propuestas son las que versan sobre la gestión de configuraciones, gestión de cambios y gestión de versiones.

Según ITIL, el objetivo de la gestión de configuraciones es proporcionar información actualizada y segura de los elementos de inventario en uso, asegurando de esta manera un enlace directo con otras disciplinas de la gestión de servicios. Una correcta gestión de los elementos que componen la configuración permite un control más directo de los recursos y permite proporcionar servicios informáticos de alta calidad cumpliendo con los costes. Para ello se tienen en cuenta las siguientes tareas:

- Planificación de la gestión de configuraciones.
- Identificación de las configuraciones disponibles.
- Control de los elementos que componen la configuración.
- Comprobación periódica del estado de las configuraciones.

Otro aspecto en el que se centra ITIL es la gestión de cambios. Un cambio siempre conlleva problemas, que si se realiza de forma equivocada y luego se quiere rectificar, puede acarrear un coste superior al del propio cambio. El principal objetivo de la gestión de cambios es llevar a cabo cambios de forma segura y minimizando riesgos. Una mala gestión del cambio puede llevar un proyecto al fracaso, incluso en las fases finales.

Por último, queda comentar la gestión de versiones. El término versión hace referencia a uno o varios cambios autorizados. Las dependencias entre una versión particular de *software* y el *hardware* requerido para que funcione determinan la agrupación de cambios de *software* y *hardware* que, junto con otros requerimientos funcionales, constituyen una nueva versión.

La gestión de versiones tiene como objetivo la planificación exitosa y el control de las instalaciones de hardware y *software*, y se lleva a cabo basándose en la base de datos de gestión de configuraciones, para así asegurar que la infraestructura está actualizada.

Esta visión que ofrece ITIL de la gestión de versiones aporta un concepto nuevo y muy interesante, que es la relación *software-hardware*. No siempre se tiene en cuenta que un cambio en el *software* puede requerir también un cambio en el *hardware* para un buen funcionamiento.

3.2.4. MÉTRICA Versión 3.

La última de las teorías que se va a exponer es MÉTRICA Versión 3. Ya que el entorno de desarrollo se pretende desplegar en el seno de una administración pública, es interesante exponer la visión que da una metodología propuesta precisamente para este tipo de organizaciones.

La metodología MÉTRICA Versión 3 ofrece a las organizaciones un instrumento útil para la sistematización de las actividades que dan soporte al ciclo de vida del *software* dentro del marco que permite alcanzar los siguientes objetivos:

- Dotar a la Organización de productos *software* que satisfagan las necesidades de los usuarios dando una mayor importancia al análisis de requisitos.
- Mejorar la productividad, permitiendo una mayor capacidad de adaptación a los cambios y teniendo en cuenta la reutilización en la medida de lo posible.
- Facilitar la comunicación y entendimiento entre los distintos participantes en la producción de *software* a lo largo del ciclo de vida del proyecto, teniendo en cuenta su papel y responsabilidad, así como las necesidades de todos y cada uno de ellos.
- Facilitar la operación, mantenimiento y uso de los productos *software* obtenidos.

La metodología contempla el desarrollo de sistemas de información para las tecnologías actuales y los aspectos de gestión que aseguran el cumplimiento de los tres objetivos básicos de calidad, coste y plazo.

MÉTRICA Versión 3, también cubre los distintos tipos de desarrollo *software* (estructurado y orientado a objetos), facilitando a través de interfaces la realización de los procesos de apoyo u organizativos. Las más interesantes para este documento son la gestión de proyectos, gestión de configuración y aseguramiento de la calidad.

Las actividades de la Interfaz de Gestión de Proyectos en Métrica Versión 3 son de tres tipos:

- Actividades de Inicio del Proyecto (GPI), que permiten estimar el esfuerzo y establecer la planificación del proyecto.
- Actividades de Seguimiento y Control (GPS), supervisando la realización de las tareas por parte del equipo de proyecto y gestionando las incidencias y cambios en los requisitos que puedan presentarse y afectar a la planificación del proyecto.
- Actividades de Finalización del Proyecto, cierre y registro de la documentación de gestión.

La interfaz de Gestión de Métrica Versión 3 permite definir las necesidades de gestión de configuración para cada sistema de información, recogiéndolas en un “Plan de Gestión de Configuración”, en el que se especifican actividades de

identificación y registro de productos, que se realizan durante todas las actividades de METRICA Versión 3.

También permite controlar el sistema como producto global a lo largo de su creación, obtener informes sobre el estado de desarrollo en que se encuentra y reducir el número de errores durante el mismo. Esto se traduce en un incremento de la calidad del proceso de desarrollo, mejora la productividad y facilita el mantenimiento del sistema.

Para concluir, en Métrica versión 3 las actividades de calidad son realizadas por un grupo de Asesoramiento de la Calidad independiente de los responsables de la obtención de los productos. Dichas actividades no entran en contradicción con el Plan General de Garantía de Calidad (PGGC), siendo lo suficientemente abiertas como para soportar una nueva versión del PGGC en el futuro.

Las actividades contempladas en la interfaz son las siguientes:

- Reducir, eliminar y prevenir las deficiencias de calidad de los productos a obtener.
- Alcanzar una confianza en que las prestaciones y servicios esperados por el cliente o el usuario queden satisfechas.

Los interfaces comentados se obtienen del Modelo de Ciclo de Vida de Desarrollo propuesto en la norma ISO 12.207, junto a los siguientes procesos:

- Planificación
- Desarrollo
- Mantenimiento

En los procesos principales se especifica el contenido, forma y momento en que se obtienen los productos, así como la relación entre los productos obtenidos en cada tarea, su reutilización en tareas posteriores y el producto final de cada actividad o proceso.

Ya que no se va a profundizar en los procesos de desarrollo, se debe destacar que dentro del proceso de desarrollo se encuadran los Estudios de Viabilidad del Sistema (EVS). Es una parte importante para este estudio, ya que la finalidad del proyecto es obtener una herramienta que cubra unas ciertas necesidades y, antes de obtener dicha solución, es necesario un estudio de viabilidad.

EVS expone que en el estudio se deberán tener en cuenta el impacto en la organización de cada una de las soluciones, la inversión y los riesgos asociados, y se obtendrán productos relacionados con las mismas.

4. Modelos de desarrollo *software*.

Antes de proponer un modelo de desarrollo final, basado en los modelos de ciclo de vida y metodologías de desarrollo estudiadas previamente, se va a buscar el modelo de desarrollo *software* que se empleará para la implementación de las aplicaciones.

Entre los modelos posibles se puede optar por un modelo clásico de desarrollo, que tiene la ventaja de ser conocido y simple, o por modelos de última generación, que aportan una reducción de costes y tiempo a la vez que aumentan la calidad.

Al ser novedosos y poco conocidos, en los siguientes apartados se van a exponer sus arquitecturas, junto con un estudio comparativo para la elección del más apropiado.

Los modelos de desarrollo de última generación que más se están imponiendo en la actualidad son la propuesta *Model Driven Architecture* (MDA) del OMG [8], basado en el enfoque *Model Driven Development* (MDD), y las Factorías de *Software* [9], que se pasan a explicar a continuación.

4.1. Model Driven Architecture (MDA).

MDA es un *framework* establecido por el OMG en noviembre del año 2000. Se basa en un nuevo paradigma de creación de *software*, en el que todo el proceso de desarrollo es guiado por modelos (*Model Driven Development*, MDD). Su principal objetivo es separar la especificación de la funcionalidad del sistema bajo desarrollo, de manera que los desarrolladores *software* describan mediante modelos la lógica del sistema, y el código sea generado de forma automática.

Tal y como sucedió con los lenguajes de programación, el desarrollo dirigido por modelos pretende aumentar el nivel de abstracción y automatización, dando así, una vez más, un salto cualitativo en el campo de la ingeniería del *software*. Esto promete una mejora de la productividad y de la calidad del *software* producido, y una reducción de la dificultad en el transito entre el dominio del problema y el dominio de la solución.

La arquitectura dirigida por modelos se asienta en un conjunto de estándares como MOF (*Meta-Object Facility*), UML, JMI (*Java Metadata Interface*) o XMI, y aplica una arquitectura de cuatro capas. MOF es el lenguaje de metamodelado a partir del que se pueden definir lenguajes de modelado como UML. El proceso MDA se basa en transformar modelos independientes de detalles de implementación en otros que aportan los aspectos específicos de una plataforma concreta. Esta transformación se denomina transformación *model-to-model* y pasa de un modelo denominado PIM (*Platform Independent Model*) a otro denominado PSM (*Platform-Specific Model*).

Tras la transformación de modelo a modelo, se realiza la transformación *model-to-code*, en la que el código fuente es generado a partir del modelo PSM.

En los siguientes apartados se va a ahondar aún más en dicha arquitectura y en las fases de desarrollo MDA.

4.1.1. Fases y niveles MDA.

Como ya se comentó en la introducción, existe una primera transformación que traduce de modelo PIM a modelo PSM, y una segunda transformación que traduce de modelo PSM a código. Basándose en dichas transformaciones se establecen tres fases:

- Fase I. Creación de un Modelo Independiente de la Plataforma (PIM). Es el modelo de mayor nivel de abstracción del sistema, y describe la funcionalidad del sistema omitiendo los detalles de cómo y dónde será implementado.
- Fase II. Transformación del PIM a uno o varios Modelos Específicos de Plataforma (PSM). Es el modelo PIM traducido a una plataforma específica.
- Fase III. Generación del código a partir del PSM. Transforma el modelo PSM a la plataforma en la que está especificado. La traducción es directa, ya que PSM está muy ligado a dicha plataforma.

La fase que realmente aporta la independencia con la plataforma sobre la que se realizará el sistema es la fase II, ya que traduce del modelo genérico al modelo específico.

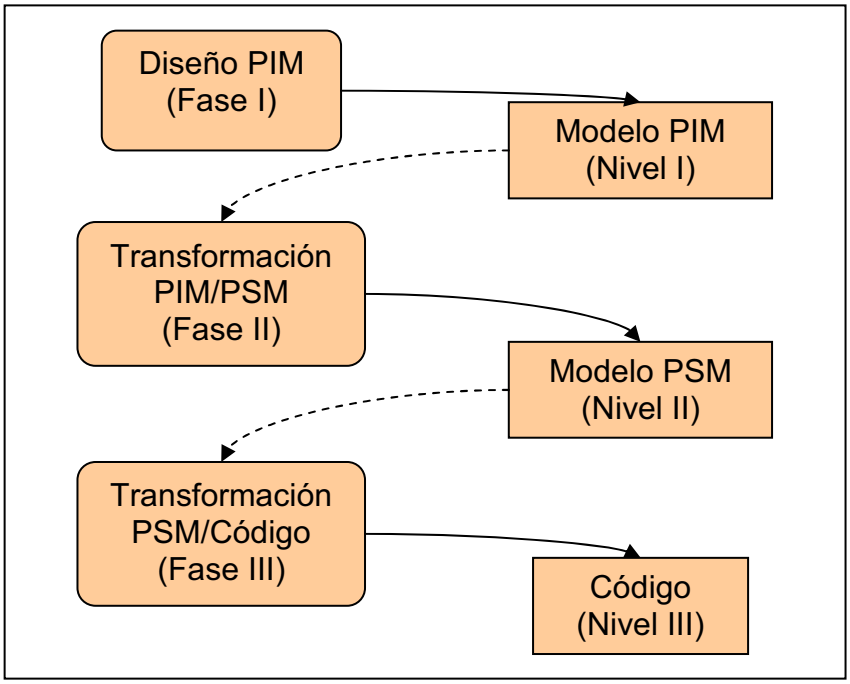


Figura 7. Fases y niveles MDA.

4.1.2. Arquitectura MDA.

Hasta el momento, UML era el principal mecanismo en el que se apoyaban las herramientas de diseño de hoy en día. Sin embargo, este nuevo enfoque ofrecido por MDD necesita de un apoyo superior al que las herramientas UML convencionales pueden aportar. Se requiere un lenguaje de descripción de lenguajes de modelado que pueda ser utilizado en el ámbito MDA. Este lenguaje es MOF (*Meta Object Facility*).

MOF es un estándar de modelado, desarrollado por el OMG, por medio del cual se puede definir formalmente la sintaxis abstracta de su conjunto de constructores de modelos, además de proporcionar una semántica informal por medio del lenguaje natural. Se basa en los diagramas de clases de UML para describir la sintaxis abstracta de los constructores de modelado para definir la sintaxis de un metamodelo. Se modela un constructor de modelado como una clase y las propiedades del constructor como los atributos de la clase, aplicando también relaciones entre constructores como asociaciones.

A partir de MOF, se define una arquitectura de cuatro niveles para el modelado:

- En el **nivel M3**, se definen los nuevos metamodelos mediante MOF. Es el nivel superior ya que MOF se puede definir así mismo, convirtiéndose en un metamodelo.
- Todos los metamodelos que son instancias de MOF se encuadran en el **nivel M2**. Dentro de estos metamodelos se encuentra UML.
- En el **nivel M1** se encuentran las instancias de los metamodelos.
- Por último, en el **nivel M0** se encuentran los objetos y datos que son instancia de modelos de nivel M1.

Una vez clarificado el desarrollo de modelos, se debe definir un estándar para el intercambio de éstos. Dicho estándar es XMI, que surgió a partir de la especificación de MOF y se basa en XML.

XML Metadata Interchange (XMI), permite representar modelos MOF en ficheros XML, simplificando así la comunicación entre aplicaciones que usen los modelos y potenciando la reutilización de objetos y componentes.

Como complemento a MOF, también se utiliza OCL (*Object Constraint Language*). Permite expresar restricciones semánticas a los metamodelos que MOF no puede añadir.

Por último, se puede aplicar también como complemento JMI, que es una especificación de Sun que define reglas de cómo representar metadatos mediante el uso de objetos Java. Se basa en la especificación de MOF y especifica como transformar la sintaxis abstracta de los metamodelos en APIs Java.

4.1.3. Transformaciones entre modelos.

Una transformación entre modelos dentro del contexto MDA consiste en la generación de un modelo destino a partir de un modelo origen, a través de un conjunto de reglas de transformación ejecutadas a través de una herramienta de transformación. Dicha transformación debe tener asociada un conjunto de características esenciales:

- Configuración. Es la más importante de las características. Consiste en adaptar una transformación a las necesidades requeridas antes de ejecutarla.
- Trazabilidad. Es la capacidad de poder seguir la traza desde un elemento del modelo destino hasta el elemento o elementos del modelo origen que lo generaron.
- Consistencia incremental. Capacidad de mantener los cambios manuales realizados en elementos del modelo destino aunque éste vuelva a ser regenerado con la transformación que lo originó.
- Bidireccionalidad. Consiste en poder obtener el modelo origen a partir del modelo destino a través de una transformación.

En la actualidad, no hay un lenguaje estándar que sirva para definir las transformaciones entre modelos y cada compañía desarrolla su propio mecanismo como JPython y AIM (*Atomistic Information Mapping*) para ArcStyler o TPL (*Template Pattern Language*) y *Technology Patterns* en OptimalJ. Esto es un problema que la OMG trata de solucionar en el ámbito de MOF a través de QVT (*Query/Views/Transformations*).

QVT fue una solicitud de propuestas lanzada en abril de 2002 para suplir la falta de un estándar de transformación en el ámbito de MOF. Las propuestas tenían que ser compatibles con UML, MOF, CWM (*Common Warehouse Metamodel*) y SPEM (*Software Process Engineering Metamodel*), de tal forma que el resultado cumpliera las siguientes premisas:

- Debía poder realizar transformaciones entre modelos de metamodelos definidos bajo MOF.
- Debía permitir la creación de la vista de un metamodelo.
- Debía permitir flexibilidad en su implementación.
- Debía ser declarativo.

Entre las propuestas más importantes cabe destacar:

- *QVT-Partners*. Propone como lenguaje de consultas y restricciones la extensión del lenguaje OCL 2.0 y que las vistas sean proyecciones de modelos padres creadas por una transformación. Se definen entonces dos tipos e transformaciones:
    - Relaciones: especificaciones de relaciones multidireccionales, que sirven para chequear la consistencia entre dos o más modelos relacionados.
    - Mapeos: Implementaciones de transformaciones, son unidireccionales y pueden devolver valores.

- *Interactive Objects Software GmbH & Project Technology*. Propone el uso de OCL 2.0 para seleccionar y filtrar la información del modelo. Las transformaciones se definen como modelos en si mismas para lo cual definen un metamodelo en MOF 2.0 para describirlas. Respecto a la vista de un metamodelo, se propone que sea la vista de un metamodelo derivado del original, como un caso especial de metamodelo. También permite mecanismos para utilizar las reglas de transformación mediante definición de plantillas.

- Otro propuesta se basa en realizar una pequeña extensión de MOF en lo que denomina XMOF, para poder realizar transformaciones entre metamodelos de forma análoga a como sería la transformación de nodos XML mediante XSLT. Para ello lo extiende añadiendo soporte para la definición de transformaciones y añade el concepto de Patrón. Un Patrón es una asociación del modelo y restricciones OCL. También cabe destacar el almacenamiento de los mapeos realizados entre distintos modelos para posteriores cambios incrementales.

4.2 Factorías de software.

Las Factorías de *Software* es la solución paralela a MDA que aporta Microsoft, con la que también se pretende dar un salto cualitativo en el desarrollo del *software*.

En lugar de adoptar un enfoque genérico que se adapte a todos los tamaños, la iniciativa *Software Factories* utiliza colecciones personalizadas de lenguajes DSL para ofrecer conjuntos de abstracciones personalizadas que satisfagan las necesidades de familias de sistemas específicas como aplicaciones de comercio electrónico o aplicaciones *web*.

Con las Factorías de *Sotfware*, los modelos no sólo se utilizan para el análisis y el diseño, sino también para admitir distintos tipos de cálculos en todo el ciclo de vida del *software* incluso en tiempo de ejecución. Este es el principio fundamental que se propone y también la iniciativa de sistemas dinámicos de Microsoft *Dynamic Systems Initiative* (DSI), que implementa y complementa la iniciativa *Software Factories*.

Las Factorías de *Software* definen una metodología adaptada a una familia concreta de productos que utilizan un gráfico de puntos de vista. Cada punto de vista define algún aspecto del ciclo de vida de los miembros de la familia, como la captura de requisitos, el diseño de bases de datos o la definición de contratos de servicios. Se asocian los activos reutilizables con cada punto de vista y los presenta en el contexto de dicho punto de vista ante los miembros de desarrollo del equipo de la familia de sistemas, lo que elimina la necesidad de buscar activos aplicables, permite la validación y admite la orientación automática y manual.

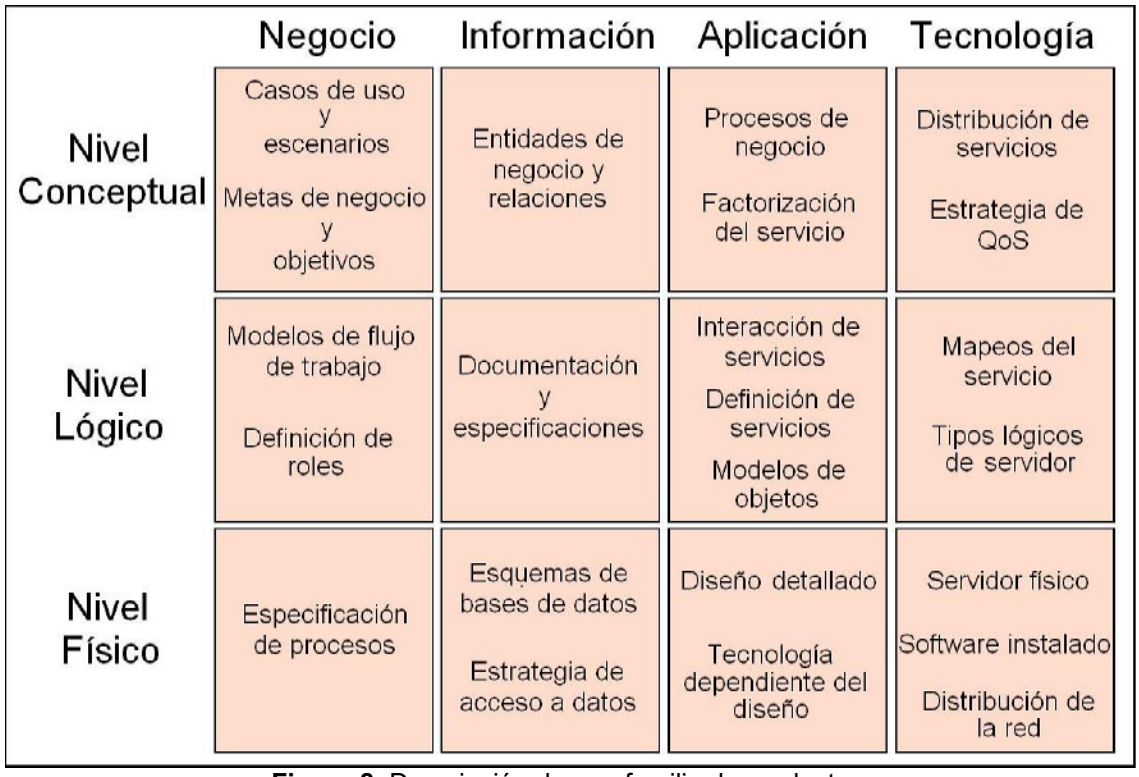


Figura 8. Descripción de una familia de productos.

Las tres ideas básicas en las que se asientan las Factorías de *Software* son un esquema de fabricación, una plantilla de factoría de *software* y un ambiente de desarrollo extensible.

El gráfico de puntos de vista, mencionado anteriormente, se denomina **Esquema de Factoría de *Software***, y relaciona el trabajo realizado en un nivel de abstracción, en una parte del sistema o en una fase del ciclo de vida, con el trabajo realizado en otros niveles o en otras partes o fases.

En un esquema se listan elementos como proyectos, código fuente, directorios, archivos SQL y archivos de configuración, y explica cómo deberían ser combinados para crear el producto perteneciente a una familia específica. Se especifica qué Lenguajes de Dominio Específico (DSL) pueden ser usados y describe cómo los modelos basados en estos DSLs pueden transformarse en código y otros artefactos, o en otros modelos. Describe la arquitectura de la línea de producto, y las relaciones clave entre componentes y *frameworks* que la componen.

Las **Plantillas de Factoría de *Software*** contienen los elementos presentados en el esquema. Provee los patrones, guías, plantillas, *frameworks*, ejemplos, herramientas personalizadas, *scripts*, hojas de estilos, y otros ingredientes para construir el producto.

Por último, se requiere de un **ambiente de desarrollo extensible**, como Visual Studio Team System, que se presenta como una fábrica de *software* para la familia de productos. Ya se ahondará más sobre este tema en el apartado de análisis de herramientas 6.2.

En definitiva, el modelo que define a las factorías de *software* es el siguiente:

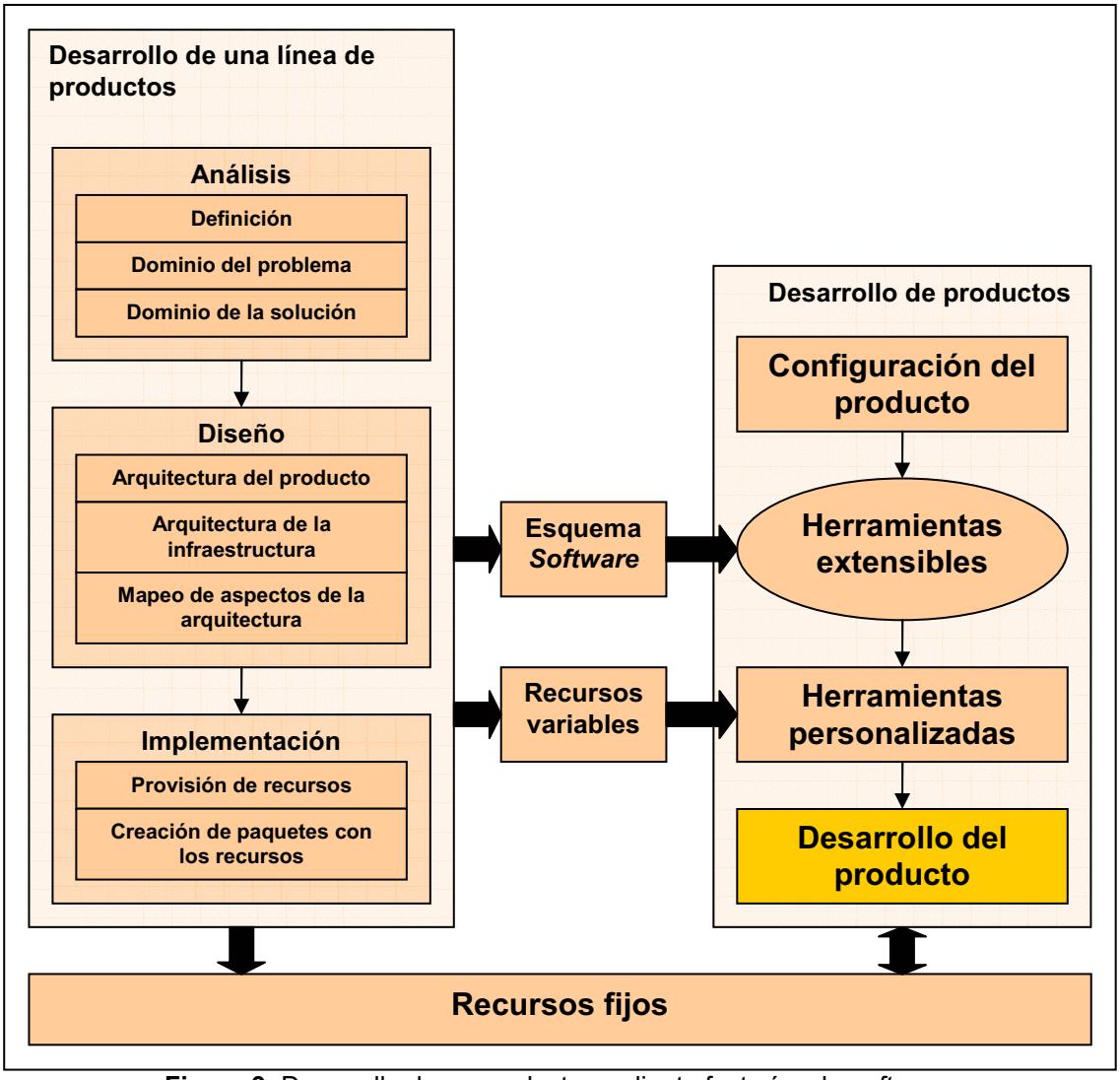


Figura 9. Desarrollo de un producto mediante factorías de *software*.

Claramente, el principio fundamental que siguen las factorías de *software* es la modularidad de componentes y la reutilización de los mismos, y convertir así el proceso de desarrollo en un proceso industrializado. Esto se puede apreciar en la figura 9.

4.3. MDA frente a las Factorías de Software.

Actualmente existe un intenso debate entre los defensores de las propuestas *Model Driven Architecture* (MDA) y las Factorías de *Software*. Artículos defendiendo y atacando cada propuesta se suceden en revistas y *blogs* personales en Internet. Habitualmente, los participantes en estos debates pertenecen a alguna de las compañías implicadas en la promoción de ambos enfoques, por lo que en ocasiones es difícil dilucidar si las argumentaciones responden a argumentos técnicos o a intereses comerciales.

En realidad dichas discusiones no tienen mucho fundamento, ya que ambos enfoques no son contrarios, sino complementarios. Las Factorías de *software* no es algo nuevo, ya que son una línea de productos *software* que configura herramientas extensibles y procesos para automatizar el desarrollo y mantenimiento de variantes de un producto arquetípico mediante la adaptación, ensamblaje y configuración de componentes basados en *frameworks*. Esta solución es compatible con MDA, e incluso en algunas herramientas MDA se cubren aspectos de Factorías de *Software* mediante la reutilización de módulos ofrecidos por el fabricante o realizados por desarrolladores independientes.

Aspectos que apuntan a MDA como solución más correcta a estudiar son que indica claramente las técnicas a utilizar (por ejemplo UML, MOF, QVT,...), está más asentada en el mercado que las Factorías de *Software* (primera versión MDA en 2001 y el primer libro sobre las Factorías de *Software* en 2004), existen más herramientas disponibles y está promovido por un consorcio de empresas (el OMG).

Otro motivo de peso para seleccionar MDA como solución es que promueve la multiplataforma. Es posible desarrollar modelos independientes del lenguaje final, para una vez desarrollado el modelo, elegir entre las distintas plataformas disponibles (J2EE, .NET, etc.). En cambio, las Factorías de *Software* protegen de forma clara el desarrollo de aplicaciones únicamente bajo la plataforma .NET.

En el caso que ocupa, e intentando seguir las líneas actuales presentadas en el aparatado 2, la solución de estudiar MDA sería la más conveniente, aunque no se va a descartar del estudio de herramientas propuestas por Microsoft orientadas a las Factorías de *Software*.

5. Modelo propuesto.

Tras el estudio de los distintos modelos existentes referentes al ciclo de vida de un producto, de las metodologías de desarrollo *software* y de las arquitecturas de desarrollo, se debe concluir en un modelo a seguir a lo largo del estudio de herramientas.

Del apartado 3.1, referente al ciclo de vida del producto, se obtienen los conocimientos para elegir un modelo apropiado y una división del ciclo en fases coherentes con el desarrollo actual de aplicaciones, que junto con el ciclo de vida propuesto con la metodología RUP, aportan las fases a cubrir. Un aspecto a eliminar es la repetición iterativa de fases para el refinamiento del producto, ya que dichas iteraciones son el resultado de la depuración de errores y cambios en la especificación.

Un punto de vista que no tienen en cuenta los modelos de ciclo de vida es el de la gestión. Por ello era necesario el estudio de las metodologías (apartado 3.2), ya que se cree que es fundamental llevar a cabo una gestión de proyectos y del cambio y de la configuración. Todas las metodologías presentan de una u otra forma la gestión de proyectos y de configuración como una parte esencial para llegar a la consecución de un proyecto (*software* o no) cumpliendo objetivos de coste y temporales.

Por todo esto, el modelo propuesto es el siguiente:

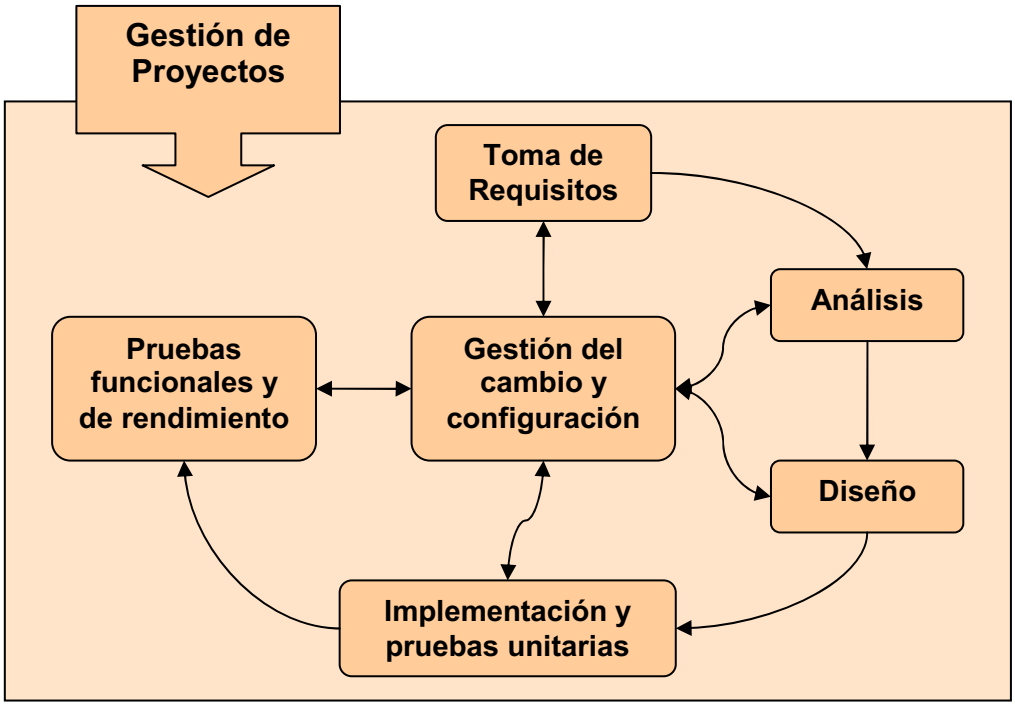


Figura 10. Modelo propuesto.

La gestión de proyectos ha de velar por el cumplimiento de los objetivos tanto técnicos como de coste y tiempo. Además de conseguir una buena planificación e integración dentro del marco de trabajo de la organización.

Se debe evitar el retorno a fases anteriores del ciclo de vida en la medida de lo posible, y en el caso de que se debiera realizar un cambio en una fase previa, este no debe afectar a los desarrollos ya realizados en las siguientes (consistencia incremental). A la misma vez, debe haber una comunicación bidireccional entre los elementos que gestionan la configuración y el cambio y los elementos que gestionan cada una de las fases del ciclo.

A partir de ahora, se debe realizar una búsqueda de herramientas que cubran de forma eficiente las fases de gestión de proyectos, toma de requisitos, análisis, diseño, implementación, pruebas y gestión del cambio y configuración. Se debe valorar la capacidad de integración de las herramientas entre ellas, de forma que el producto de una herramienta que cubra una fase sea la entrada de la herramienta que cubra la siguiente.

Todo esto se ha de realizar sin perder de vista los importantes consejos que ofrecen las metodologías comentadas anteriormente.

A continuación, se realizará un estudio de las herramientas de desarrollo candidatas a cubrir las distintas fases propuestas en el modelo de la figura 10, teniendo en cuenta los modelos de desarrollo presentados en el apartado 4.

6. Estudio de herramientas de desarrollo.

Después de la elección del modelo bajo estudio, se pasa a realizar un estudio de las herramientas provistas por las empresas más importantes dentro de la ingeniería del *software*, que intentan cubrir una o más fases del ciclo de vida del *software* y que tienen en cuenta ciertos puntos de las metodologías revisadas anteriormente.

Antes de entrar en detalle en las herramientas, el apartado se centrará en un estudio de la Plataforma Eclipse, en las facilidades que aporta y en algunas de las herramientas que integra.

Una vez claros los conceptos básicos de Eclipse, el estudio se centrará en el estudio de las herramientas propuestas por Microsoft, que se basan en las Factorías de *software*.

Como se observará, la única compañía que no converge hacia MDA es Microsoft, ya que Borland, IBM Rational, Telelogic, Compuware e Iteractive Objects, entre muchas otras compañías [10], comienzan a adaptar sus entornos de desarrollo al nuevo estándar propuesto por el OMG. Por este motivo, el estudio de herramientas se centrará principalmente en herramientas que cumplan en mayor o menor medida dicha propuesta. El esquema que se va a seguir en el estudio es el siguiente:

- Estudio de la Plataforma Eclipse y herramientas relacionadas.
- Herramientas provistas por Microsoft, entre las que destaca *Visual Studio Team*. Esta se basa en el enfoque de Factorías de *Software*.
- Herramientas provistas por Borland. Se proponen soluciones basadas en MDA.
- Herramientas provistas por IBM Rational. Se proponen soluciones basadas en MDA.
- Herramientas provistas por Telelogic (e I-Logix). Se proponen soluciones basadas en MDA.
- Estudio de la herramienta MDA propuesta por *Interactive Objects*: ArcStyler.
- Herramientas provistas por *Compuware*, centradas en torno a su herramienta de desarrollo MDA: OptimalJ.

6.1. Plataforma Eclipse.

Es interesante realizar un estudio de **Eclipse**, ya que su objetivo es desarrollar una plataforma industrial robusta, con todas las características y de calidad industrial para el desarrollo de herramientas altamente integradas. Es un proyecto que ha atraído a grandes compañías (IBM Rational, Borland, Telelogic, Compuware,…) y se perfila como un modelo a seguir, marcando una tendencia en el mundo del *software*.

Eclipse es un IDE (*Integrated Development Environment*) *open source* y extensible. El proyecto se lanzó originalmente en Noviembre de 2001, cuando IBM donó 40 millones de dólares del código fuente de Websphere Studio Workbench y formó el *Eclipse Consortium* para controlar el desarrollo continuado de la herramienta.

Al final, el *Eclipse Consortium* se había enfocado en tres proyectos principales:

- ***The Eclipse Project,*** que es responsable de desarrollar el banco de trabajo del IDE Eclipse (la plataforma que contiene las herramientas de Eclipse), JDT (Java Development Tools), y PDE (*Plug-in* Development Environment) utilizado para ampliar la plataforma.

- ***The Eclipse Tools Project*** se enfoca en la creación de herramientas para la plataforma Eclipse. Entre los subproyectos actuales se incluyen un IDE Cobol, un IDE C/C++, y una herramienta de modelado EMF (Eclipse Modeling Framework).

- ***The Eclipse Technology Project*** se enfoca en investigaciones tecnológicas, incubación y educación utilizando la plataforma Eclipse.

La plataforma Eclipse, cuando se combina con JDT, ofrece muchas de las características que cabría esperar de un IDE de calidad comercial: editor con sintaxis coloreada, compilación incremental, un depurador que tiene en cuenta los *threads* a nivel fuente, un navegador de clases, un controlador de ficheros/proyectos, e interfaces para control estándar de código fuente.

A pesar del gran número de características estándar, Eclipse se diferencia de los IDEs tradicionales en varias cosas fundamentales. Quizás lo más interesante de Eclipse es que es completamente neutral a la plataforma y al lenguaje. Además de la mezcla ecléctica de lenguajes soportados por el Eclipse Consortium (Java, C/C++, Cobol), también hay otros proyectos para añadir a Eclipse el soporte de lenguajes tan diversos como Python, Eiffel, PHP, Ruby y C#. Con respecto a las plataformas, Eclipse Consortium proporciona instalaciones binarias para Windows, Linux, Solaris, HP-UX, AIX, QNX, y Mac OS X.

La mayoría de los aspectos interesantes que posee Eclipse se centran en la arquitectura de *plug-ins* y el API proporcionado por el *Plug-in Development Environment* para ampliarlo. Añadir soporte para un nuevo tipo de editor, una vista, o un lenguaje de programación, es remarcadamente fácil, con el API y los bloques de construcción que proporciona Eclipse.

6.1.1. Herramientas y *plug-ins* de Eclipse para la gestión del ciclo de vida de un producto *software*.

Se debe diferenciar entre herramientas integradas y *plug-ins*. Existen algunas herramientas construidas directamente con Eclipse y que forma una parte indivisible con él. Los *plug-ins* son herramientas externas que se han desarrollado de forma que se puedan integrar con Eclipse mediante su interfaz para plug-ins.

En referencia a las herramientas, Eclipse incluye varias características únicas, como la refactorización de código, la actualización/instalación automática de código (mediante Update Manager), una lista de tareas, soporte para unidades de prueba con JUnit, e integración con la herramienta de construcción de Jakarta: ANT. Antes de avanzar, se va a explicar brevemente en que consisten JUnit y ANT.

JUnit es un *framework* que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de un valor de entrada se evalúa el valor de retorno esperado, si la clase cumple con la especificación, entonces devolverá que el método de la clase pasó exitosamente la prueba. En caso de que el valor esperado sea diferente al que devolvió el método durante la ejecución, devolverá un fallo en el método correspondiente.

JUnit también es un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificada y se desea ver que el nuevo código cumple con los requerimientos anteriores, y en definitiva no se ha modificado su funcionalidad después de la nueva modificación.

Por otro lado, **ANT** es un proyecto de código abierto de la *Apache Software Foundation*. Consiste en una herramienta construida en Java que permite agilizar la ejecución de programas mediante *scripts* basados en XML, pero en vez de basarse en ejecución de instrucciones por línea de comandos utiliza clases Java. Con la llamada a una clase ANT se permite la ejecución de varias tareas. Cada tarea se encapsula en el objeto que la implementa y ofrece un interfaz de tarea específico. Está orientada a la gestión de servidores de aplicaciones.

Ambas, son herramientas interesantes capaces de facilitar en gran medida el trabajo a la hora de realizar pruebas.

En referencia a los *plug-ins*, se ha hecho una breve recopilación de los que se han considerado, dentro del marco de este proyecto, los *plug-ins* más útiles para la gestión del ciclo de vida, y así también demostrar la heterogeneidad de las ampliaciones disponibles.

JRequire. Es un *plug-in* que cubre la fase de captura de requisitos. Facilita de forma eficiente el grado de cumplimiento de un código con los requisitos previstos, y permite al desarrollador obtener una vista rápida de los requisitos

que se han cumplido, los incumplidos y los que aún no se han cubierto. Así, el director de proyecto obtiene información de forma rápida del estado del proyecto y de las áreas potencialmente conflictivas.

Subclipse. Permite realizar un control de versiones de forma integrada en la plataforma Eclipse, accediendo de forma fácil al *Concurrent Versions System* (CVS) y pudiendo realizar operaciones sobre el mismo. Si se añade el *plug-in* **Boneclipse-cvsgrapher**, se permite obtener una vista gráfica del histórico del CVS e interactuar con dicha vista, pudiendo así realizar operaciones de *zoom* y comparación, entre otras. Además de Subclipse y Boneclipse, se puede integrar otra herramienta llamada **TortoiseSVN**, que es un p*lug-in* para realizar operaciones sobre el servidor de versiones.

Omondo EclipseUML. Es una herramienta para modelado UML. Permite trabajo en equipo, sincronización bidireccional, integración de CVS y generación de diagramas de clase y secuencia a partir de ingeniería inversa.

JNIUS Generator. Permite realizar un desarrollo basado en MDD (Model Driven Development). Los modelos de datos pueden ser:

- Ficheros UML provenientes de cualquier herramienta de diseño que cumpla con el estándar, como pueden ser Rational Rose, Together, ArgoUML...
- Un mapeo desde la base de datos mediante Hibernate (ficheros .hbm y .xml).

Además de todos los *plug-ins* comentados, Eclipse ofrece un total de más de 500 *plug-ins* que cubren los campos de análisis, diseño, implementación y pruebas (entre muchos otros). Algunos de estos plug-ins se repasarán en los siguientes estudios de herramientas, ya que IBM Rational, Borland y Telelogic, entre otras muchas empresas, se han lanzado de lleno al “mundo de Eclipse” lanzando plug-ins de sus mejores herramientas comerciales.

Teniendo en cuenta un factor importante como el coste, éste es inmejorable, ya que el IDE de Eclipse, y la mayoría de sus ampliaciones, es totalmente gratis. Este puede aumentar si se pretende integrar algún *plug-in* comercial para cubrir alguna etapa del ciclo de vida o dar soporte a algún proceso especial.

6.2. Microsoft Visual Studio Team y la Plataforma Avenade.

Las herramientas de desarrollo ofrecidas por Microsoft entre el año 2005 y 2006 se orientan sobre todo al nuevo enfoque propuesto por las Factorías de *Software*.

La herramienta integrada más importante, actualmente, de Microsoft para el desarrollo de proyectos *software* es Visual Studio Team (VST) [11]. Ayuda a reducir la complejidad al crear soluciones orientadas a servicios que estén diseñadas para operaciones y facilita la colaboración entre todos los miembros del equipo, acelerando los tiempos de desarrollo y asegurando la fiabilidad del proceso de desarrollo.

Una primera aproximación a lo que ofrece Microsoft para cubrir las fases de desarrollo es la mostrada en la siguiente figura:

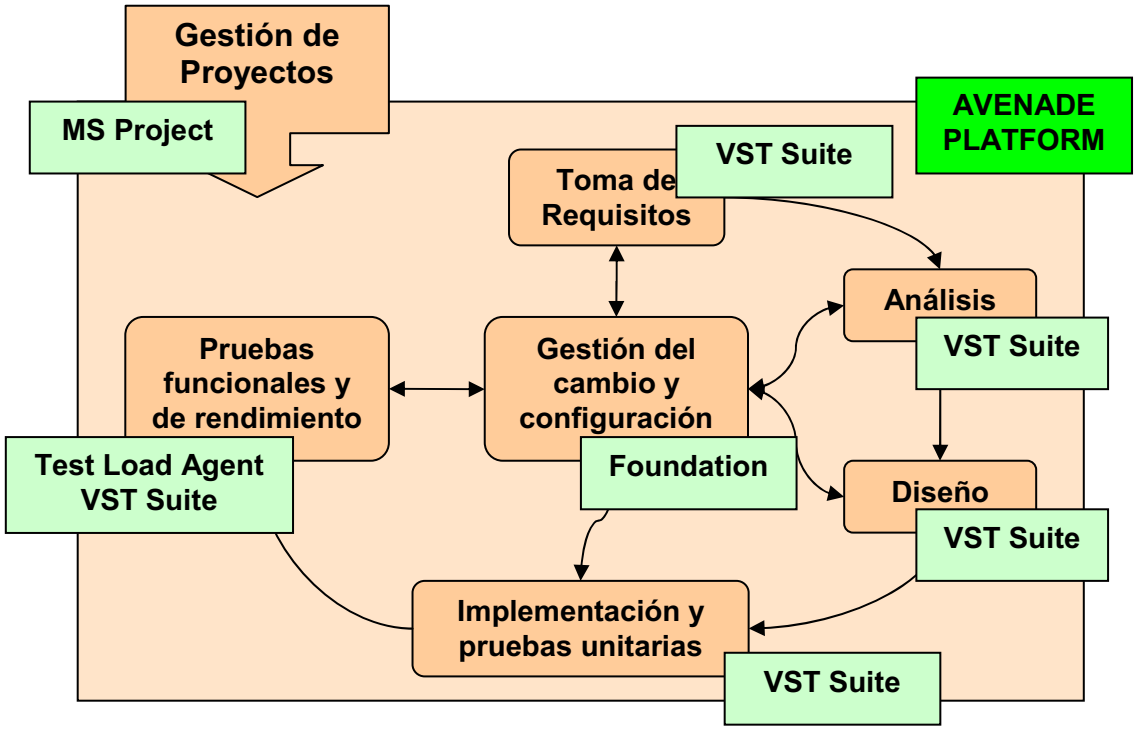


Figura 11. Modelo propuesto en apartado 5 cubierto por herramientas Microsoft.

Para la gestión de proyectos se integra **MS Project Server** (última versión del 2003), que es una herramienta bastante conocida y extendida que permite gestionar los recursos disponibles y realizar una planificación temporal de los proyectos, teniendo en cuenta dichos recursos.

VST Suite 2005 se compone de un conjunto de herramientas de gestión de ciclo de vida integradas y extensibles, donde también se diferencia entre los distintos roles de los participantes involucrados en el proyecto (VST para Arquitectos, Desarrolladores y Probadores).

VST Suite contiene:

- Herramientas integradas de modelado de servicios *web*.
- Herramientas para el control de calidad de código y rendimiento.
- Herramientas de pruebas de carga para cumplir rigurosas demandas de rendimiento.

En el caso de **VST 2005 para Arquitectos de *Software***, se provee de herramientas integradas de diseño de aplicaciones para el desarrollo orientado a servicios. Diseñadores visuales que permiten a los arquitectos, responsables de operaciones y desarrolladores el diseño de soluciones orientadas a servicios que pueden ser validadas contra sus entornos operacionales.

Las herramientas integradas permiten diseño de aplicaciones con servicios *web* para la visualización de arquitecturas orientadas a servicios, herramientas de diseño para mostrar la infraestructura lógica de la red y herramientas de diseño de implantación para validar que las aplicaciones funcionan correctamente en la infraestructura de red disponible.

En **VST 2005 para Desarrolladores de *Software***, se pueden encontrar herramientas de rendimiento y calidad de código. Permiten a los equipos de proyecto la creación de aplicaciones fiables y robustas. Se dispone de herramientas integradas para realizar pruebas unitarias y de cobertura de código, herramientas de perfilado de código y herramientas de diagnosis de errores críticos de seguridad y rendimiento. Todo esto sirve para asegurar la calidad de las aplicaciones antes de llegar a producción.

En el caso de **VST 2005 para Probadores de *Software***, se ofrecen herramientas integradas para pruebas de carga de servicios y aplicaciones *web*. Estas permiten pruebas de carga para verificar el rendimiento de las aplicaciones antes de implantarlas. Integra herramientas de autoría de pruebas para poder firmar rápidamente *scripts* de pruebas de carga, herramientas de pruebas de carga para crear y ejecutar pruebas distribuidas y herramientas para compartir resultados de pruebas de carga con otros miembros del equipo.

Una aplicación integrable con VST para pruebas es **Test Load Agent 2005**. Ofrece una carga suplementaria para pruebas para usar con VST Edition para probadores de *software* que permite a las organizaciones simular más usuarios y así probar de forma más fidedigna el rendimiento de los servidores y aplicaciones Web.

Test Load Agent, requiere una licencia por procesador y permite simular aproximadamente 1.000 usuarios por procesador.

Por último, para la gestión del cambio, se ofrece como solución **VST Foundation Server**. Consiste en un servidor de colaboración en equipo para asegurar una comunicación más eficiente entre sus miembros. Permite a todos los miembros del equipo gestionar y seguir el progreso y estado de los proyectos. Para acceder a Team Foundation Server, cada usuario debe disponer de una licencia de acceso de cliente CAL *(Client Access License*).

Permite:

- Gestión de activos y control de código fuente a nivel empresarial.
- Seguimiento e informes integrados de los puntos de trabajo para controlar el estado de los proyectos.
- Metodología de procesos integrada para que el desarrollo de *software* sea más eficiente y previsible.

Para que todas las herramientas propuestas puedan funcionar de forma conjunta para cubrir el ciclo de vida de una forma eficiente, Microsoft ofrece la plataforma **Avenade**. Es una plataforma, centrada en la tecnología .NET, que ayuda a la gestión integral del proyecto en todas las fases de su ciclo de vida.

Las ventajas que aporta *Avenade Software Lifecycle Platform* (SLP) son las siguientes:

- Un entorno de desarrollo que facilita la ejecución de proyectos en equipos geográficamente distribuidos.
- Integran la metodología de desarrollo con el entorno de desarrollo.
- Integran las herramientas de gestión de proyectos con el entorno de desarrollo.

Sobre la plataforma Avenade, **Visual Studio Team System** (VSTS) es un entorno para la ejecución de proyectos que puede ser adaptado a las necesidades del equipo de proyecto y para proyectos específicos. Incluye control de versiones, generación de informes, procesos guía y herramientas para construcción automática. En este punto es donde las Factorías de *Software* comentadas anteriormente aparecen, a esto es a lo que se refiere VSTS con construcción automática.

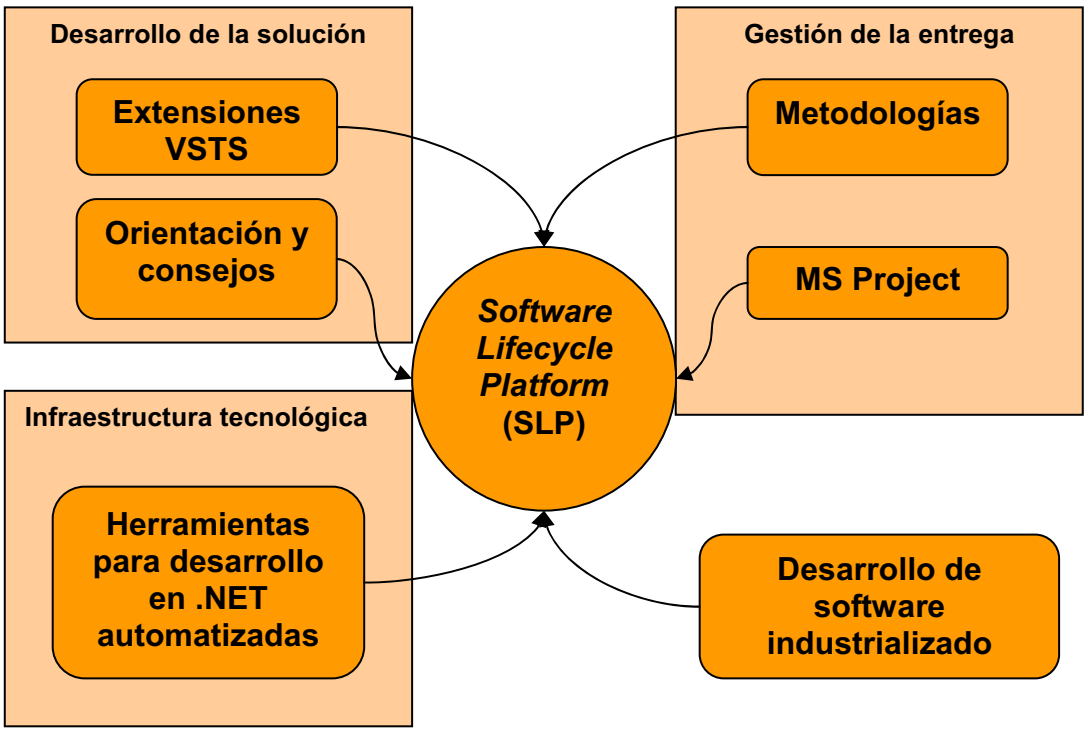


Figura 12. Plataforma para la gestión del ciclo de vida propuesta por Microsoft.

Un aspecto clave en el desarrollo de proyectos es la **provisión de un entorno de desarrollo común**. Todos los proyectos parten de dicho entorno y en la mayoría de los casos es una fase de larga duración y manual, que requiere una importante atención por parte de desarrolladores y los ingenieros de sistemas. En la mayoría de los casos se requiere un entorno de desarrollo similar al entorno de producción para eliminar inconsistencias en el producto debido a diferentes entornos de trabajo.

Una de las soluciones adoptadas recientemente ha sido crear un entorno de desarrollo virtual. Una imagen de desarrollo base puede ser creada y distribuida entre todos los desarrolladores. Esto permite que todos los desarrolladores partan del mismo entorno de desarrollo, evitando así inconsistencias y *bugs* relacionados con el sistema. Esta solución no es válida

por varios motivos. En primer lugar, no es práctico copiar a través de una red imágenes de varios gigabytes. En segundo lugar, aunque se parta de una misma imagen inicial, no se asegura que los desarrolladores realicen cambios en la misma durante el desarrollo del proyecto. En tercer lugar, el uso de VSTS implica la creación de varias configuraciones complejas (al menos se requerirían 3 imágenes para trabajar en un entorno distribuido de trabajo). Es necesario encontrar una forma de distribuir dichas imágenes de forma dinámica y segura.

Avenade ofrece una solución al problema de la distribución de las imágenes mediante *Avenade Automated Build Framework* (ABF), que permite la creación de *scripts* que realizan las actualizaciones de forma automática y a intervalos de tiempos configurables.

Una solución más avanzada que ABF consiste en el uso de la herramienta ACA *Dynamic Systems Framework* (DSF), que facilita el almacenamiento, despliegue y reutilización de recursos de Avenade. ACA DSF encapsula en su núcleo todas las facilidades de diseño, configuración, implementación y *testeo* de Avenade.

Otro aspecto importante de Avenade es que da **soporte a entornos de desarrollo distribuidos**, mediante su sistema de control de código fuente. También permite acceso web mediante http a Team Foundation Server (TFS). Se tienen en cuenta escenarios en los que exista una alta latencia en el acceso y un bajo ancho de banda, permitiendo así una gran flexibilidad en el acceso. También se tienen en cuenta escenarios en los que los límites geográficos impongan diferentes dominios corporativos.

Para la **integración de Metodologías de Desarrollo**, VSTS permite la integración de metodologías específicas de la organización y de procesos estándar mediante el uso de plantillas de procesos que provee la metodología mediante Visual Studio IDE. VSTS también contiene dos procesos de plantillas llamadas *MSF for Agile Software Development* y MSF para CMMI *Process Improvement*. También existe una plantilla para *Avenade Connected Methods* (ACM) para .NET, que es la metodología para la ejecución de proyectos .NET. La plantilla ACM para .NET incluye soporte para las diferentes fases:

- Previsión.
- Planificación.
- Ejecución.
- Estabilización.
- Despliegue.

Las plantillas se centran en cumplir los siguientes objetivos:

- Orientación a proceso. Describe la metodología en detalle. Incluye una visión general, una definición de roles, *work items* (más adelante se verán en detalle) y los flujos de trabajo y actividad.
- Grupos de seguridad y permisos. TFS provee de una seguridad basada en roles. Se definen grupos de seguridad y permisos para los diferentes

roles (Arquitecto, desarrollador, ingeniero de sistemas, probador y director de proyecto).
- Control de fuente y permisos. La plantilla del proceso también define la información de entrada de la que podrá disponer un nuevo equipo de proyecto y si el nuevo equipo requiere una verificación exclusiva. Además, la plantilla de del proceso define permisos por defecto para el control de la fuente.
- Work Item Types. Un *work item* es un registro de base de datos que VST utiliza para seguir la asignación y estado del trabajo. Unos ejemplos de *work item* para ACM podrían ser "Requerimiento", "Tema" y "Defecto".
- *Default Work Items*. Una plantilla de proceso crea *default work items* cuando se crea un nuevo grupo de proyecto. Por ejemplo, una platilla de proceso puede crear un grupo de tareas que deben ser completadas por los miembros del equipo al inicio del proyecto.
- Informes y documentación. Cuando se crea un nuevo grupo de proyecto mediante una plantilla de proceso ACM, también se crea conjunto de documentos e informes descriptivos del proyecto en el repositorio de documentos del proyecto.
- Áreas e iteraciones. Las plantillas de proceso definen áreas relacionadas con la partición del proyecto en diferentes aspectos. Las iteraciones determinan cuantas veces debe repetir el equipo un conjunto de actividades, tales como planificar, desarrollar y probar. ACM no divide el proyecto de esta forma. En su lugar, las actividades son agrupadas por fases de proyecto. Las áreas y las iteraciones se mapean en los correspondientes elementos dentro de ACM.

Para la **integración de la gestión de proyecto**, VSTS provee herramientas para la gestión de proyectos. Algunos de las facilidades que aporta son las siguientes:

- Base de datos para *work items*.
- Puede enlazar entradas de código, defectos y compilaciones con *work items* en la base de datos.
- Imponer criterios de restricción a tareas, esto es, no poder salir de una tarea en concreto hasta que se completen unas actividades específicas.
- Dar soporte a una metodología mediante plantillas de procesos (como se explicó anteriormente).
- Obtención de métricas basadas en datos obtenidos de la base de datos de *work items*.
- Creación de informes sobre calidad del código, progreso de la planificación y efectividad de pruebas.
- Permite integración con MS Project y con MS Excel.

Aunque VSTS no provee de forma directa gestión de proyectos, ofrece una plataforma donde dicha gestión se puede aplicar. Existen una gran cantidad de métricas que se pueden obtener de los datos de Team Foundation Server (TFS). Al utilizar SLP junto con VSTS podemos capturar esas métricas y redirigirlas hacia herramientas de gestión de proyectos (por ejemplo, hacia MS Project Server). Así se obtiene una vista completa de toda la información

disponible y se puede realizar un seguimiento del estado de uno o varios proyectos.

Con esta arquitectura, TFS es la fuente primaria de datos estadísticos y podemos sincronizar estos datos con Project Server, consiguiendo así realizar la gestión de proyectos con Project Server en vez de utilizar VSTS. En la siguiente figura se puede observar un posible despliegue que cubra la arquitectura propuesta.

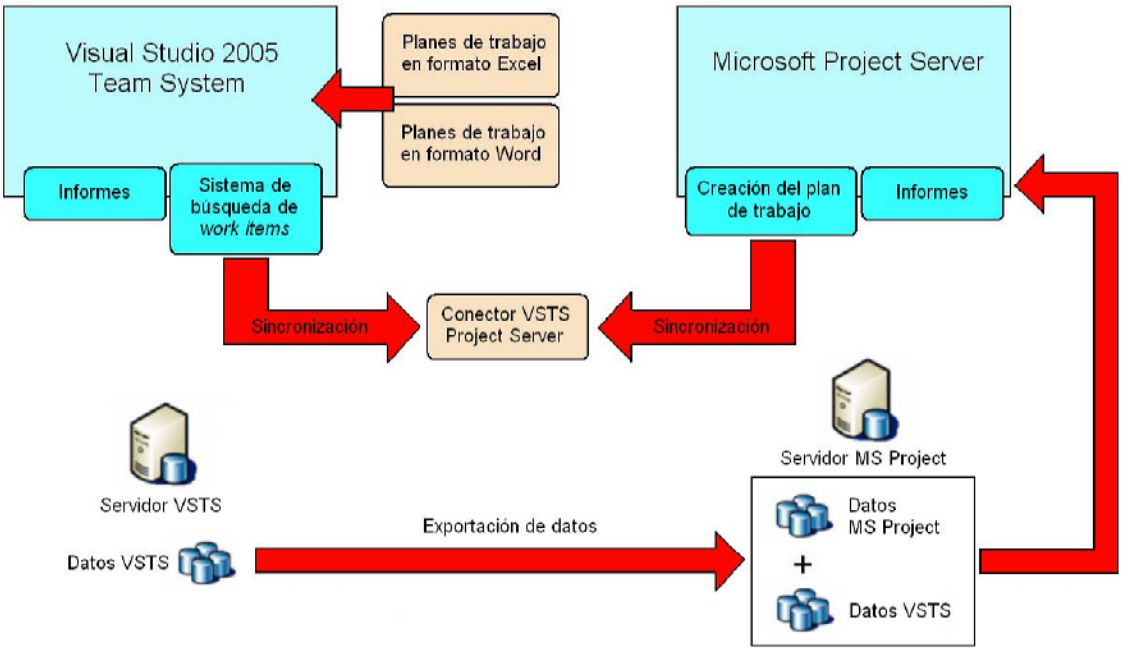


Figura 13. Integración de VSTS con MS Project.

Para terminar con este apartado, se va a realizar un repaso de las ventajas que ofrece la utilización de SLP.

Integración continuada. Elimina sesiones donde los desarrolladores emplean gran parte del tiempo en la búsqueda de *bugs* debidos a actualizaciones de otros desarrolladores (mala integración del código de distintos desarrolladores). Dichos *bugs* son muy difíciles de detectar debido a que el error no se encuentra en el trabajo de un desarrollador sino a la interacción del código de distintos desarrolladores. Mediante la integración continua se eliminan dichos errores ya que se pueden detectar en el mismo momento en que el código se introduce en el proyecto. Por ello, la integración se convierte en una tarea diaria y se reduce de forma considerable el esfuerzo.

Políticas de validación de cambios parametrizables. Team Foundation ofrece un control de versiones que nos permite crear nuestras propias reglas de validación del código entrante en la actualización. Se consideran dos interacciones en particular:

- Definición de política: el proceso por el cual un director o jefe técnico define los requerimientos que los desarrolladores deben satisfacer para actualizar el repositorio. Los usuarios definen una política a través de la

opción de configuración del equipo de proyecto, y la definición de la política se facilita a través del diálogo de propiedades de proyecto.

- Evaluación de la política: es el proceso de determinar en que grado se satisface la política y ocurre cuando un desarrollador intenta llevar a cabo una actualización.

El control de versiones de Team Foundation almacena las políticas que os desarrolladores definen en una localización específica donde cualquier miembro del proyecto puede acceder y obtenerlas. Las políticas también están asociadas a proyectos en particular, por lo que el control de versiones de Team Foundation tiene un almacén de esquemas que ofrece una vista rápida para encontrar las políticas asociadas a un proyecto en particular.

Ciclo diario de compilación automática. Si el sistema que esta bajo desarrollo no puede ser compilado de forma directa desde su código fuente y a la vez situado de forma directa en su ruta de instalación, entonces no es posible integrar muchos de los procesos requeridos para que su entrega tenga éxito. Para que una compilación pueda ser automática y repetible se deben cumplir los siguientes requerimientos:

- Mantener un único repositorio para todo el código fuente perteneciente al proyecto donde cualquier miembro del proyecto pueda obtener la última versión disponible (o cualquier versión anterior).

- Automatizar el proceso de compilación de tal forma que cualquier miembro del proyecto pueda construir de forma rápida y sencilla el sistema a partir del código fuente.

- Automatizar el proceso de pruebas de tal forma que se pueda aplicar un conjunto predefinido de pruebas al sistema en cualquier momento.

6.3. Borland Tempo, Caliber, Together y StarTeam.

Borland [12] provee herramientas para cada fase del producto, con una plataforma completamente integrada, facilitando la labor de cada uno de los integrantes del proyecto. Pone en práctica una filosofía basada en roles y procesos, e integra soluciones basadas en el enfoque MDA.

Las fases que componen el ciclo de vida del proyecto son definición, diseño, desarrollo y *test*, con sus correspondientes roles para los integrantes del proyecto (Analista, arquitecto, desarrollador y probador). El modelo propuesto se puede observar a continuación en la siguiente figura:

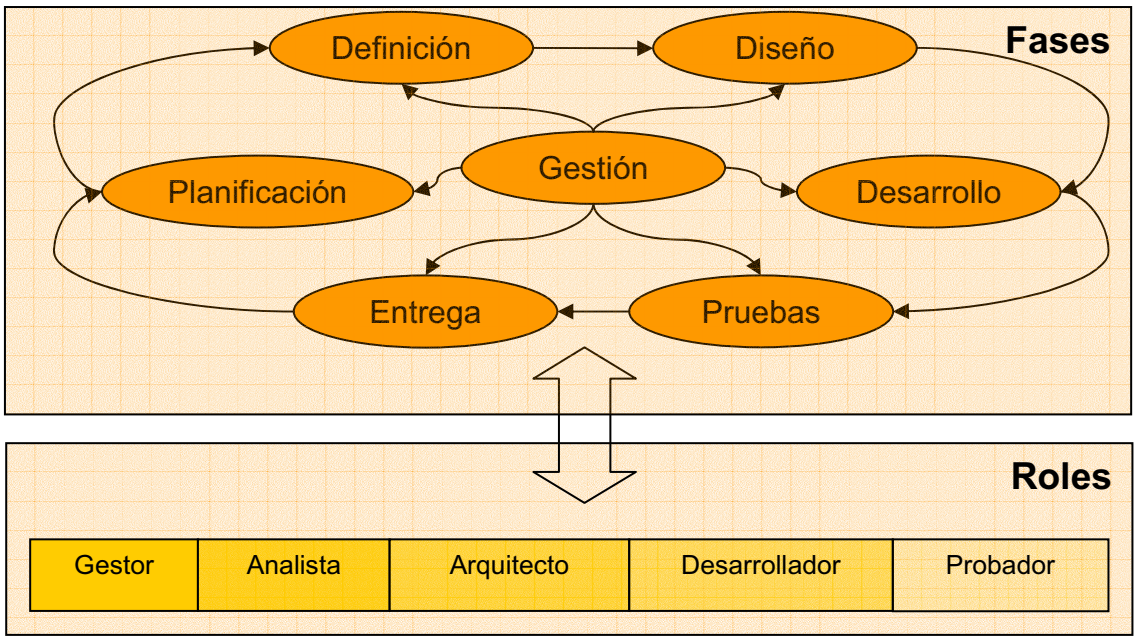


Figura 14. Modelo propuesto por Borland.

La plataforma propuesta por Borland permite a una organización automatizar el proceso de desarrollo de productos a la vez que provee de un repositorio centralizado. Esto permite compartir información entre todos los componentes del proyecto y durante todas las fases del ciclo de vida.

La función de cada uno de los roles está claramente definida y tiene un mayor énfasis según la fase en la que se encuentre el producto. Borland ofrece entornos adaptados a cada uno de los roles:

- Interfaz de Analista. Traduce las necesidades de negocio a requerimientos *software*, teniendo en cuenta el efecto de futuros cambios en los requisitos del sistema y así haciendo que se cumplan los objetivos previstos.
- Interfaz de Arquitecto. Ofrece una arquitectura con un entorno integrado que mantiene las especificaciones, modelos y código en sincronía durante todas las fases del ciclo de vida de la aplicación, incluso tras cambios en los requerimientos.
- Proceso de Desarrollador. Aplica herramientas de desarrollo, ofreciendo vistas sobre las especificaciones, cambio de requerimientos y pruebas, permitiendo un desarrollo eficiente.
- Interfaz de Probador. Permite realizar pruebas sobre el producto teniendo en cuenta los parámetros de calidad, rendimiento y fiabilidad.

Antes de analizar las herramientas ofrecidas por Borland, se muestra en la siguiente figura la utilidad de cada una de ellas en cada fase del ciclo de desarrollo *software* propuesto anteriormente.

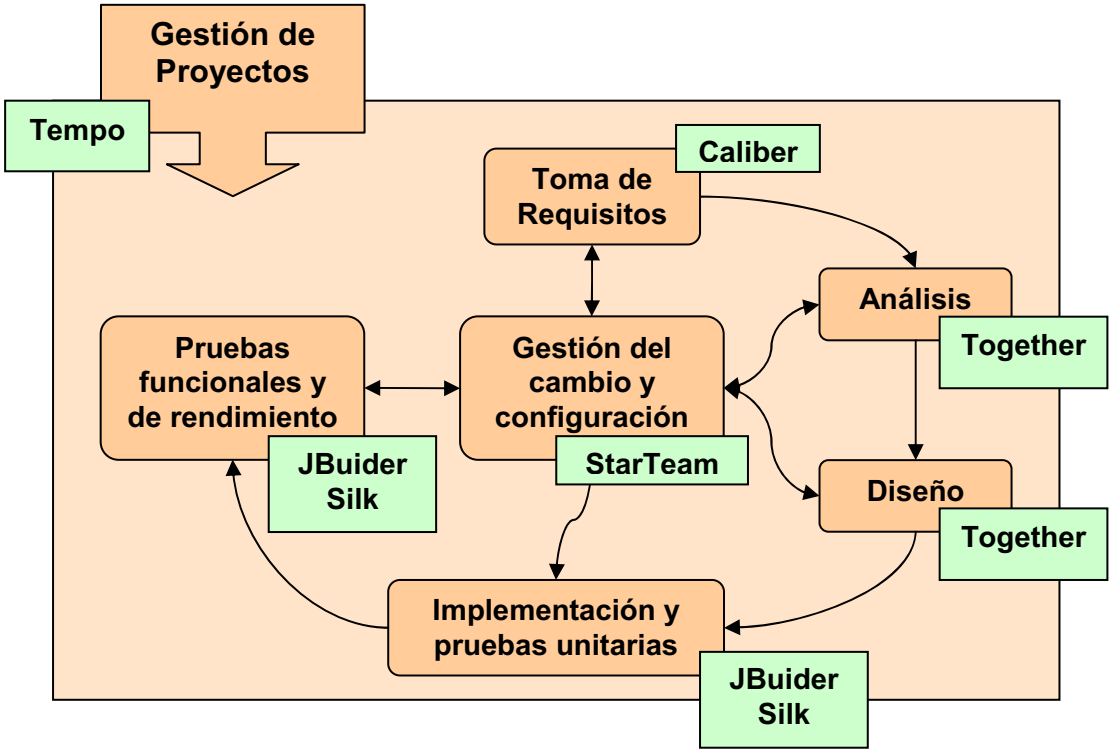


Figura 15. Modelo propuesto en apartado 5 cubierto por herramientas Borland.

Borland Tempo es una herramienta que permite realizar una gestión de proyectos eficiente teniendo en cuenta las tres variables principales implicadas en la realización de proyectos: Tiempo, Recursos (humanos y no humanos) y Financiamiento.

Provee seis módulos para gestionar los distintos aspectos que componen los proyectos:

- Gestión de la demanda. Donde se puede realizar una gestión de las solicitudes de proyectos, automatización de los procesos de desarrollo y recopilación de información referente a los proyectos en curso.
- Gestión de cartera de inversión. En este módulo se definen las métricas importantes para el seguimiento de los proyectos y el análisis de las carteras.
- Gestión de proyecto y programa. Incluye herramientas para la planificación de proyectos y herramientas de seguimiento del estado de los mismos.
- Gestión financiera. Módulo para la gestión y seguimiento de los recursos económicos asignados.
- Gestión de recursos humanos. Incluye facilidades para la gestión basada en las habilidades de los participantes en los proyectos y seguimiento de los equipos de trabajo a lo largo del tiempo.
- Gestión de recursos no humanos. Posibilita análisis de impacto, gestión del ciclo de vida y gestión del cambio y de la configuración, a nivel de recursos.

Se debe destacar que Borland Tempo permite implementar procesos personalizados para evaluar y seleccionar la mejor inversión tecnológica basada en costes, beneficios y riesgos. Incluye facilidades para planificar, gestionar la carga de trabajo del proyecto y visualizar el estado y el progreso de los proyectos en ejecución, monitorizando la eficiencia.

Borland Caliber 2006 ha sido diseñado para capturar y gestionar requerimientos de negocio, técnicos, funcionales y operacionales, además de permitir un seguimiento a través de todas las fases del ciclo de vida del producto. Ofrece dos herramientas para esto:

- Borland CaliberRM Datamart. Provee de inteligencia orientada a negocio monitorizando tendencias, descubriendo oportunidades y optimizando estrategias. Genera informes basados en la tecnología de BussinessObjects haciendo más fácil el análisis y comparando métricas clave entre múltiples proyectos, e incluso entre múltiples tecnologías como StarTeam y Mercury TestDirector. Incluye:
    - Datamart Extractor. Obtiene de projectos CaliberRM datos y los prepara para análisis y genera informes para su uso mediante herramientas de informes basadas en SQL como BussinessObjects, Cognos, Brio y Crystal Reports.
    - Datamart Synchronizer. Permite manejar información obtenida de BussinessObjects y actualiza los datos para reflejar cualquier cambio en algún campo y asegurando que siempre se utiliza información actualizada para el análisis.

- Borland CaliberRM Estimate Professional. Ofrecen ayuda a los directores de proyecto para planificar el alcance, el programa y los recursos durante el desarrollo del *software* a lo largo de su ciclo de vida.

Borland Together es la herramienta más importante de todas las que ofrece Borland, ya que es el entorno de análisis y desarrollo, al unirse con el IDE **JBuilder** (**C#/C++ Builder** en el caso de que no se desarrolle en Java), propiamente dicho. Existen diferentes versiones según el rol del trabajador que lo vaya a utilizar. A continuación se realiza un breve estudio sobre cada una de ellas.

Borland Together Designer 2006 es una solución multiplataforma de modelado UML para analistas empresariales y personas que desarrollan su actividad en un entorno donde los modelos visuales pueden optimizar la definición de requisitos y las especificaciones sobre la arquitectura y el código del *software*. Together Designer combina las capacidades de modelado que requieren los analistas y que son necesarias para una definición inequívoca de los requisitos del *software* junto con una generación de código rápida y eficaz que garantiza el cumplimiento de los criterios establecidos. Los arquitectos y programadores pueden utilizar los casos de uso, actividades y otros modelos de UML creados en Together Designer para diseñar y crear aplicaciones que reflejen de manera exacta los requisitos definidos.

Borland Together Architect 2006 es una solución de modelado UML exhaustiva y compatible con diversos lenguajes diseñada para los arquitectos de *software* que diseñan, programan y especifican arquitecturas de aplicaciones empresariales en colaboración con clientes de departamentos de programación y empresariales. Together Architect permite a los arquitectos optimizar las aportaciones de los analistas empresariales y de otras personas que definen y evalúan los requisitos de las aplicaciones empresariales. Posibilita una comunicación entre los arquitectos y los programadores y verificadores de *software* con descripciones inequívocas del código de la aplicación que deben crear y verificar en proyectos de programación complejos donde diversos equipos trabajan en paralelo. Together Architect permite a los arquitectos de *software* optimizar los procesos de programación en todo el ciclo de vida de la aplicación para limitar los riesgos, agilizar la implementación, reducir los costes y mejorar la calidad.

La última versión que provee Borland es la versión **Borland Together Developer 2006**, que ya es una versión orientada directamente al desarrollador.

En principio, Borland Together sólo permitía sincronizar modelos y código, pero en la última versión de Together se ha comenzado a añadir el enfoque MDD, que va un paso más allá. Esta última versión ya utiliza el lenguaje OCL y permite importar y exportar modelos en formato XMI, permitiendo también realizar transformaciones de modelo a código.

La tecnología de transformación de modelos a código utilizada por Together es la **tecnología LiveSource**. Dicha tecnología permite realizar transformaciones bidireccionales entre modelos específicos (PSM) y código. Pero el diseño no comienza en PSM, sino en la creación de modelos PIM (como dicta el estándar MDA). Por lo que Borland implementa el modelo en tres niveles propuesto por el OMG.

Además, se proveen versiones de Borland Together integradas dentro de otros entornos de desarrollo como las siguientes:

- Borland Together Edition para Microsoft Visual Studio .NET ofrece a los programadores en C# y Microsoft Visual Basic .NET un entorno de lenguaje de modelado unificado integrado y normalizado.

- Borland Together Edtion para Eclipse. Con su entorno de diseño integrado y ágil, permite a los equipos de trabajo agilizar la programación de aplicaciones de mayor calidad utilizando la plataforma Eclipse de código abierto e IBM WebSphere Studio.

- Borland Together Edition para JBuilder. Ofrece la potencia de la tecnología UML de Borland LiveSource a un sector más amplio de programadores Java. La funcionalidad de Together permite ver el código existente en diagramas de clase y de secuencia, detectar errores con herramientas de análisis estático, utilizar patrones de diseño y generar automáticamente documentación de proyectos.

La herramienta que se encarga de la gestión de la configuración es **Borland StarTeam 2005**, que consiste en una plataforma para coordinación y gestión de todo el proceso para la entrega del producto. Promueve la comunicación del equipo y la colaboración mediante un control centralizado de los recursos del proyecto. El acceso es seguro y flexible, lo que permite que los miembros del equipo puedan trabajar en cualquier localización física mediante herramientas de escritorio, *web* y clientes en modo consola.

Incluye gestión de requerimientos, gestión de cambio, búsqueda de defectos, control de versiones y gestión de proyecto y tareas.

Por último, para la fase de pruebas se utiliza **Borland Silk**. Es un potente conjunto de herramientas que cubre todo los tipos de pruebas posibles y ofrece un concepto importante: Gestión de Pruebas.

Con la Gestión de Pruebas, Borland permite un proceso dirigido hacia la planificación, documentación y gestión del proceso de pruebas. Las herramientas que se encargan de esto son SilkCentral TestManager y SilkCentral Issue Manager.

SilkCentral TestManager permite planificar y gestionar cada etapa del proceso de pruebas mejorando así la calidad y la productividad, mientras que **SilkCentral Issue Manager** es un proceso dirigido a optimizar la búsqueda de defectos y su resolución a lo largo de todas las fases de desarrollo.

Silk Test es la herramienta que cubre las pruebas funcionales, orientándose a cumplir las expectaciones del usuario final y los requisitos de negocio. Una característica interesante es **SilkTest International**, que orienta las pruebas a distintos aspectos lingüísticos, de plataforma y de *browser* que pueden afectar a la aplicación.

Por último, **SilkPerformer** cubre las pruebas de carga, estrés y ejecución, aplicando simulaciones en condiciones reales antes de su distribución.

6.4. IBM Rational.

Rational [13] ofrece una gestión completa del ciclo de vida del *software*, cubriendo por completo las fases de toma de requisitos, análisis, diseño, gestión del cambio y de la configuración y automatización de pruebas funcionales y rendimiento. Para su modelo de desarrollo comienza a integrar en sus últimas herramientas el enfoque MDA, aunque con algunas variaciones con respecto a la propuesta del OMG.

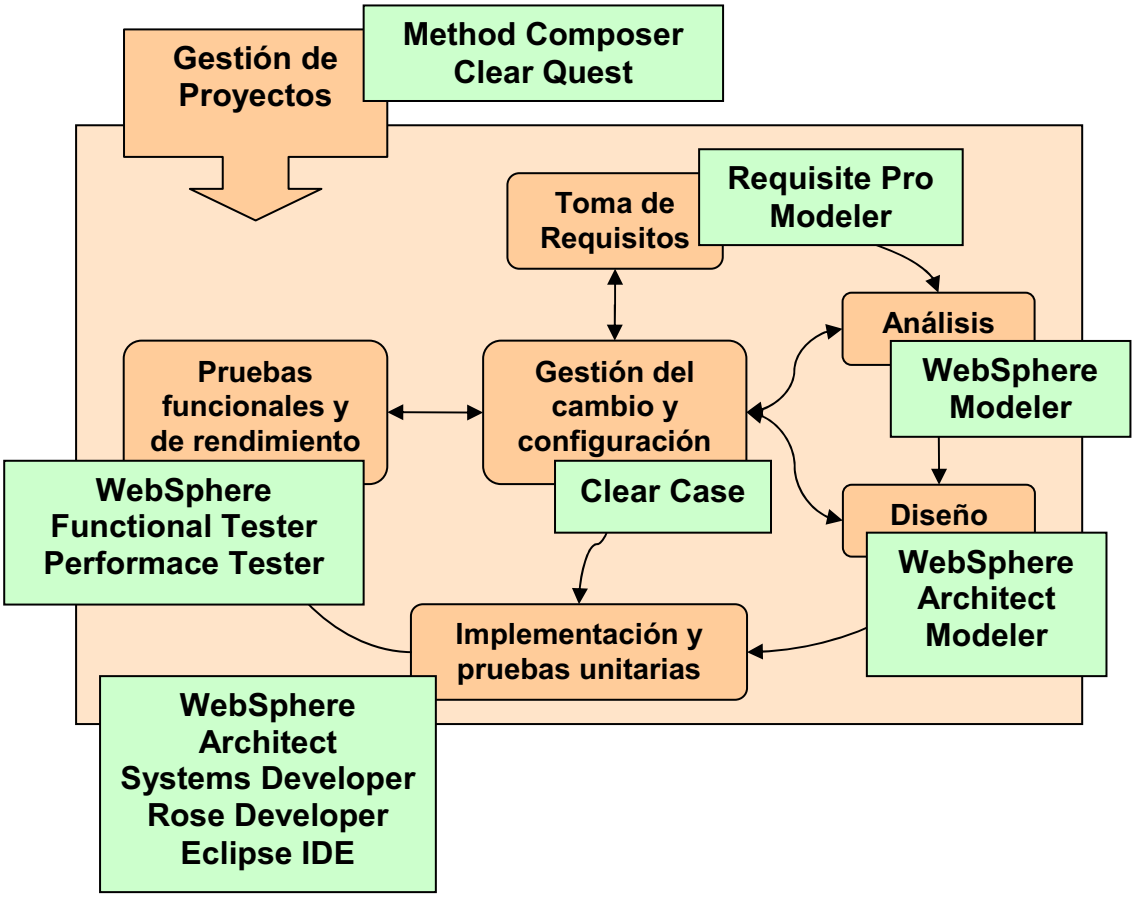


Figura 16. Modelo propuesto en apartado 5 cubierto por herramientas IBM Racional.

IBM Rational también plantea una división en roles muy parecida a la propuesta por Borland, distinguiendo entre analista, arquitecto, desarrollador, probador y gestor de despliegue. Así se pretende distinguir mejor la labor y las herramientas a utilizar por cada miembro del equipo de desarrollo.

Un aspecto a destacar de IBM Rational es que participa de forma activa en **Eclipse.org**. Apuesta por dicho entorno como un estándar de código abierto, permitiendo a los equipos trabajar de forma efectiva en distintos lenguajes y entornos.

Dentro de los productos de IBM Rational construidos directamente sobre la plataforma Eclipse, se destacan los comentados a continuación.

Para la gestión de proyectos se proponen las herramientas Rational Method Composer y Clear Quest. **IBM Rational Method Composer** contiene herramientas y procesos aplicables a todo el ciclo de vida del producto. Es la siguiente generación de *Rational Unified Process* (RUP), y parte de un *framework* basado en procesos de Eclipse.

IBM Rational ClearQuest es una herramienta y un *plug-in* para Eclipse que permite una mejor visión, capacidad de predicción y control del proceso de desarrollo de *software* entre entornos geográficamente distribuidos.

Para la fase de toma de requisitos ofrece la herramienta **IBM Rational RequisitePro**, que también se encuentra como *plug-in* para Eclipse. Producto integrado para requerimientos y manejo de casos de uso, que proporciona una mejor comunicación, mejora el trabajo en equipo y reduce el riesgo del proyecto.

IBM Rational *Software*. En su **versión *Modeler*** se presenta una herramienta basada en modelado UML visual para analistas de sistemas y diseñadores. Permite realizar un diseño claro de las especificaciones y de la arquitectura del sistema. En su **versión *Architect*** se ofrece una herramienta para la creación de arquitecturas *software* sobre Java/C++ que utiliza como apoyo UML. Permite unificar los aspectos de diseño e implementación. La versión Modeler está orientada al rol de analista y la versión Architect está orientada al rol de arquitecto.

IBM Rational Application/Web Developer (WebSphere). Es un IDE que permite a los desarrolladores realizar en un corto periodo de tiempo diseños, desarrollos, pruebas e integración de aplicaciones *web* basadas en Java. Por lo que cubre ampliamente la mayoría de las fases del ciclo de vida del modelo propuesto.

A parte de WebSphere, IBM Rational ofrece una amplia familia de productos directamente dirigidos a desarrolladores, y que aplican el enfoque MDD, como **Rational Rose Developer** o **IBM Rational Systems Developer.** Estas herramientas permiten realizar desarrollos orientados a modelos y transformaciones finales en código C/C++ y Java.

El motor de transformaciones propuesto por IBM Rational es ***IBM Model Transformation Framework* (MTF)**. Este motor implementa algunos de los conceptos de QVT y se basa en un metamodelo y un conjunto de mapeos predefinidos para la generación de modelos a partir del metamodelo inicial. Especificando dichos mapeos entre modelos se provee una forma declarativa para definir cual debe ser el resultado de una transformación. Por ejemplo, si un mapeo requiere la existencia de un objeto en el modelo destino, el motor de transformación lo crea para satisfacer el mapeo. Pero MTF no propone MOF como lenguaje de metamodelado. Propone EMF, ya que así sigue en la línea iniciada por la Plataforma Eclipse.

Otro concepto que propone MTF es el de “Reconciliación”. Es el mecanismo que provee para mantener consistencia incremental. La reconciliación intenta satisfacer las relaciones generadas por la creación de nuevos elementos y modificación o borrado de elementos existentes. En algunos casos no es necesaria, si el modelo sigue siendo consistente tras la modificación, y en otros casos no es posible. Por este motivo, el modelo de consistencia de MTF no es, por el momento, del todo aceptable.

Para la fase de pruebas se proponen IBM Rational Funcitional Tester e IBM Rational Performance Tester. **IBM Rational Functional Tester** es una herramienta de *test* funcional para desarrolladores GUI. Realiza pruebas sobre aplicaciones Java, VS.NET y aplicaciones Web. **IBM Rational Performance**

Tester es una herramienta de pruebas para validación de escalabilidad y fiabilidad de aplicaciones complejas.

Para el control de versiones existe una herramienta llamada **IBM Rational ClearCase**, que se integra con el resto de herramientas para poder mejorar la eficiencia de equipos de trabajo. También se encuentra disponible como *plug-in* para Eclipse.

Como se ha podido observar a lo largo del desarrollo de este apartado, la mayoría de herramientas de IBM Rational o están desarrolladas directamente sobre Eclipse o se pueden incluir a modo de plug-in. Por este motivo también se podría proponer en este apartado al Eclipse IDE como herramienta para cubrir la fase de implementación.

6.5. Telelogic DOORS, SYNERGY, TAU y Rhapsody.

Telelogic [14] ofrece, principalmente, tres herramientas para la gestión integral del ciclo de vida del proyecto *software*. Estas son **DOORS, SYNERGY y TAU**. Uniendo las tres herramientas se consiguen cubrir las etapas de toma de requisitos, gestión del cambio y de la configuración y diseño. De entre estas tres herramientas, TAU es la que ofrece una solución de desarrollo basada en MDA.

Junto a DOORS, SYNERGY y TAU se encuentran otras herramientas interesantes como Telelogic Focal Point, Telelogic Logiscope y Telelogic DocExpress.

Telelogic Focal Point es una herramienta de apoyo para toma decisiones de mercado. Basada en *web*, permite a la organización determinar, gestionar y monitorizar el mejor producto o proyecto en el que invertir, centrándose en objetivos de negocio, necesidades del consumidor, costes y recursos disponibles. También ofrece un análisis de las futuras consecuencias de una decisión mediante distintos escenarios (escenarios "*what if*").

Telelogic Logiscope permite evaluar de forma automática el código y detectar errores y *bugs* de programación. También detecta módulos conflictivos y ofrece recomendaciones para mejorar la calidad del código.

Telelogic DocExpress es un *software* de documentación que simplifica la gestión del proceso de documentación permitiendo a los usuarios recoger datos, formatearlo y publicar documentación técnica del proyecto en los formatos más habituales.

La distribución de las herramientas en ciclo de vida del producto se muestra en la figura que viene a continuación:

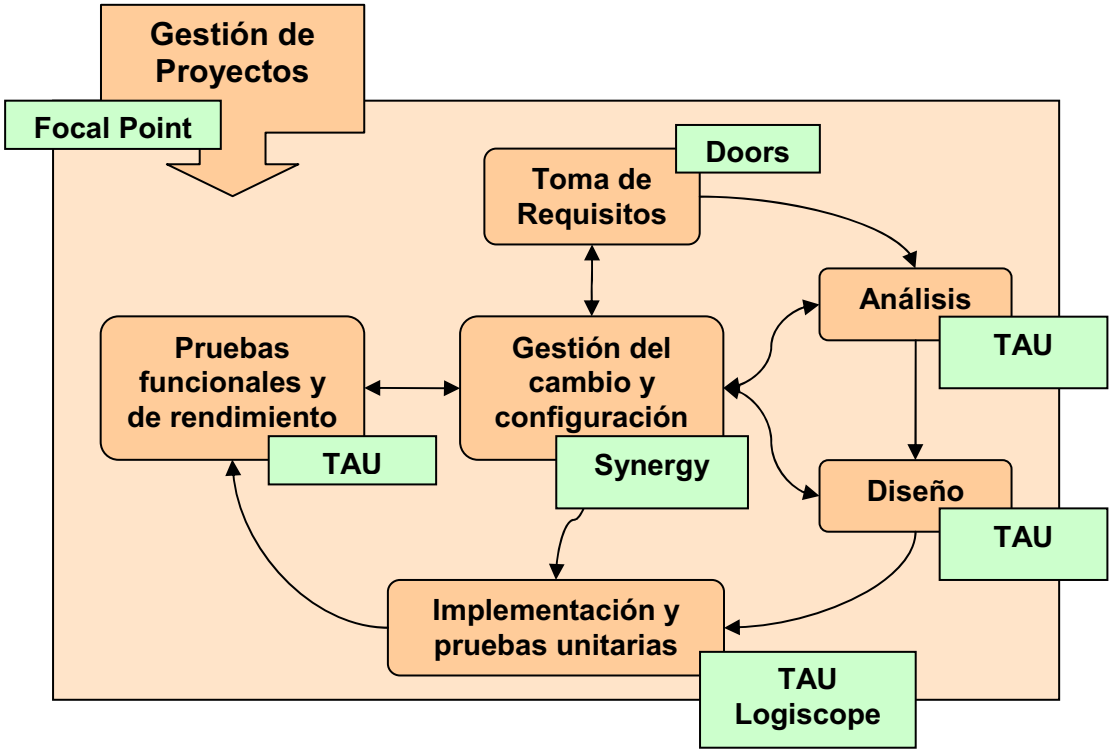


Figura 17. Modelo propuesto en apartado 5 cubierto por herramientas Telelogic.

Telelogic DOORS es la herramienta que ofrece Telelogic para cubrir la toma de requisitos. Mejora la calidad optimizando la comunicación, la colaboración y verificación. Ofrece:

- Interfaces intuitivas que facilitan la adopción de un sistema de gestión de requerimientos.
- Escalabilidad. DOORS es útil para cualquier proyecto, de forma independiente de su tamaño y número de usuarios.
- Ofrece una estructura útil para almacenar, estructura, administrar y analizar requerimientos y así realizar un seguimiento eficaz de los mismos.
- Permite una integración completa con el resto de herramientas de Telelogic y de terceros, para tener así una visibilidad clara de los requerimientos en cualquier fase del proyecto.

Para proyectos y empresas globales, **Telelogic DOORS XT** maximiza el valor de la optimización del proceso de negocio e incrementa la calidad de la ingeniería de sistemas mejorando el seguimiento de los requerimientos y la comunicación entre equipos geográficamente distribuidos. Incrementa la visibilidad de los requerimientos tales como los objetivos de negocio, necesidades del consumidor, las especificaciones técnicas y la regulación.

Mediante una gran capacidad para capturar, enlazar, analizar y gestionar cambios en los requerimientos y sus herramientas de seguimiento, esta multiplataforma se asegura de que el proyecto siga los requerimientos y cumpla con la regulación y los estándares.

Para analistas de *software*, **Telelogic DOORS/Analyst** ofrece un interfaz fácil de aprender para crear modelos de requisitos basados en UML. Permite al analista complementar los documentos de requerimientos textuales con modelos de requerimientos y mantener un seguimiento de los modelos y de los propios requisitos.

Como complemento de Telelogic DOORS, también se puede utilizar **DOORSnet** para facilitar el acceso a los requisitos a usuarios remotos. Dicho acceso remoto da la facilidad de editar los requerimientos creados mediante Telelogic DOORS de forma remota mediante acceso Web.

Los usuarios que accedan mediante DOORSnet podrán:

- Acceder a la base de datos de DOORS a través de una interfaz que presenta los datos de forma jerárquica.
- Listar, buscar, ordenar y editar los requerimientos de forma remota.

Con **Telelogic SYNERGY** se puede llevar a cabo la gestión del cambio y la gestión de configuración. Es una herramienta de control válida para todo el ciclo de vida del producto y para la gestión de los recursos digitales. Permite valorar y realizar cambios en los recursos existentes, tanto desde fuentes internas como externas. Permite:

- Conseguir una mejora considerable en el desarrollo, mejorando la calidad del mismo y pudiendo conseguir una reducción de los costes.
- Acelera los ciclos de lanzamiento de productos.
- Maximiza la productividad, de forma independiente de la localización geográfica de los equipos de proyecto.
- Minimiza la carga de trabajo automatizando las tareas que se repiten a lo largo del ciclo de vida.

Para una gestión remota, **SYNERGY/Change** es una herramienta de acceso *web*, completamente integrada, que permite realizar un seguimiento sobre las peticiones de cambio y crear informes sobre dicho seguimiento. Incrementa la calidad y recude el riesgo de la implementación de cambios no autorizados. Como complemento, **Telelogic DASHBOARD**, permite a los directores de proyecto obtener una vista rápida sobre el riesgo del proyecto, su estado y obtiene de forma automática medidas de SYNERGY/Change para su análisis y creación de informes.

Otra facilidad integrada con Telelogic SYNERGY es **SYNERGY/CM**, que consiste en una herramienta para la gestión de la configuración basada en tareas, que ayuda a los equipos de desarrollo a trabajar de forma más sencilla y rápida, mejorando la comunicación y la colaboración. Agiliza la gestión y construye procesos de gestión, maximiza la eficiencia cuando existen recursos limitados y facilita el desarrollo cuando existen equipos distribuidos. También cuenta con un repositorio de información distribuido.

Telelogic TAU se aplica en el campo de la ingeniería de sistemas, desarrollo de *software* y automatización de pruebas. Facilita la concepción de proyectos y ofrece una visualización de los requerimientos.

Se basa en modelado visual mediante diagramas donde los usuarios pueden especificar todos los aspectos del diseño del sistema bajo desarrollo, simular y verificar su comportamiento, comprobar que el diseño sigue el camino correcto (incluso en las etapas más tempranas del proyecto), generar código directamente desde el modelo validado y definir, generar y ejecutar pruebas sobre el mismo.

TAU se cumple con los estándares para modelado visual UML (Unified Modeling Language) 2.0, SysML (System Modeling Language) y SDL (Specification and Description Language). También cumple con los lenguajes estándar para definición de casos de *test* TTCN-2/3.

Para simulación y generación de Tests Telelogic permite la integración de **TAU G2**. Es un entorno de desarrollo dirigido por modelos (MDD) orientado principalmente a ingeniería de sistemas y desarrollo de *software*. Es compatible con UML 2.0, y SysML. También permite simulación de modelos dinámicos y generación de código.

Otra herramienta de pruebas interesante es **TAU/Tester**. Basado en TTCN-3, realiza pruebas sobre sistemas y pruebas de integración del *software* en las distintas fases del ciclo de vida del producto. Las pruebas pueden ser automáticas y manuales, y pueden ser almacenadas para su posterior repetición.

Telelogic **TAU TTCN Suite**, es el estándar de facto para la realización de pruebas de sistemas de comunicación. Es utilizado tanto para desarrollo de chips de comunicación como para el diseño de grandes conmutadores y servicios de red inteligente. El lenguaje de *script* de TAU TTCN es TTCN-2, que está ampliamente difundido y es estándar de la ETSI e ISO.

TAU TTCN está totalmente integrado en **TAU SDL Suite**, que es un entorno de programación visual para especificación y diseño de *software* de comunicación, basado en el estándar SDL.

Pero esto no es todo lo que hoy en día puede ofrecer Telelogic. Como bien se comentó al principio del apartado, Telelogic ofrece principalmente tres herramientas, pero a principios del año 2006 la compañía I-Logix [15] fue adquirida por Telelogic. De esta forma se suma una cuarta herramienta a las tres mencionadas anteriormente. Esta herramienta es **Rhapsody** (versión 6.2), y sobre la que centrará la atención del estudio.

La filosofía de Rhapsody siempre ha sido la generación de modelos independientes de la plataforma, incluso mucho antes de que el OMG lanzara la iniciativa del MDA.

Rhapsody genera aplicaciones PIM que son ejecutadas sobre el framework de Rhapsody y los adaptadores de sistema operativo. En la siguiente figura se muestra el procedimiento que se sigue en el desarrollo de aplicaciones con Rhapsody, y en las pruebas de las aplicaciones generadas o en desarrollo.

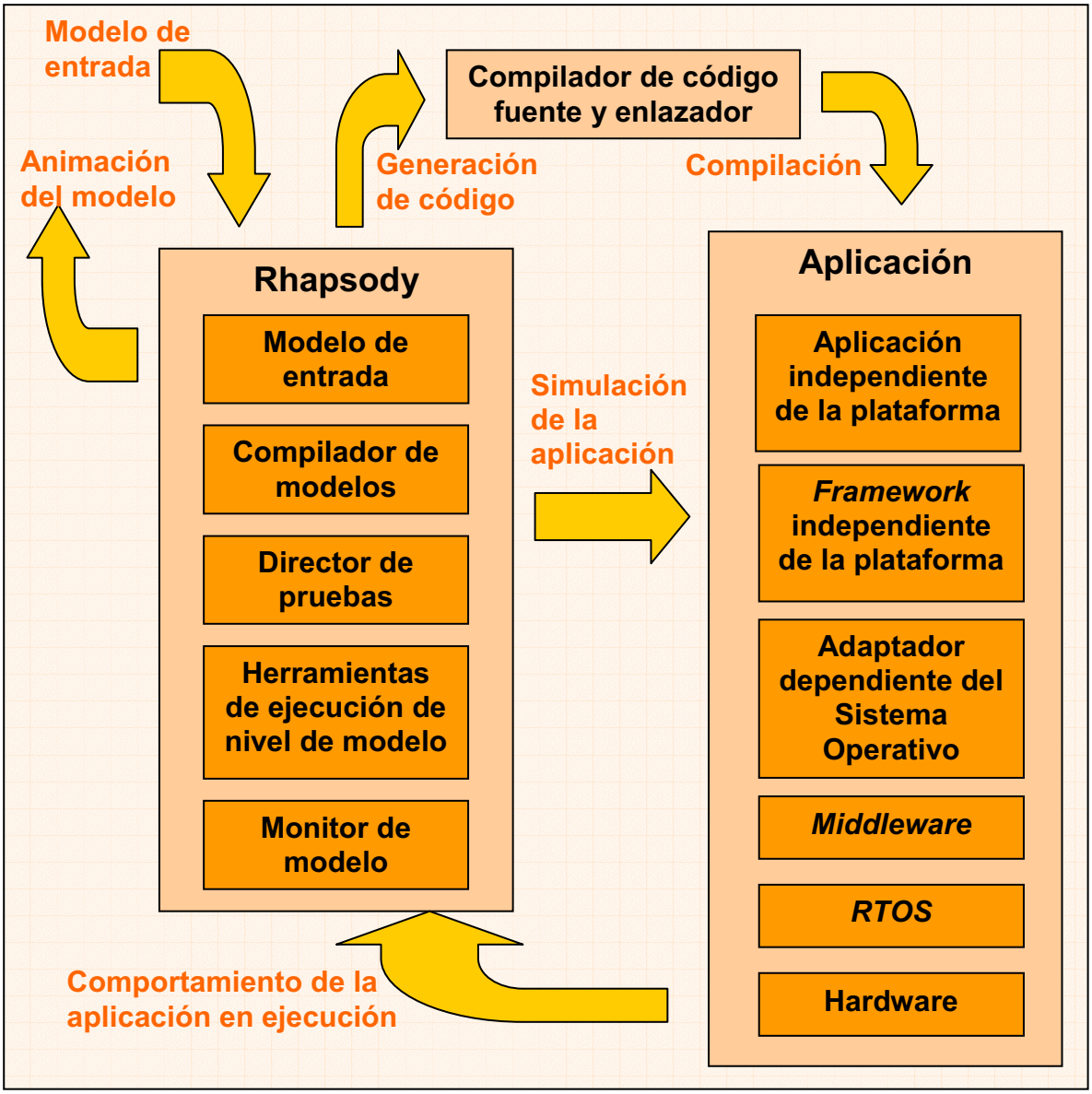


Figura 18. Modelo de desarrollo y pruebas de Rhapsody.

Rhapsody destaca principalmente por las siguientes características:

- Sistema de modelo de entrada. El desarrollador introduce un modelo PIM, mediante diagramas UML estándar.
- Compilador de modelo. El desarrollador genera código fuente para un lenguaje dado y lo compila.
- Testeador de modelos. Permite al probador simular y monitorizar la ejecución de la aplicación generada mediante el PIM introducido, en un *host* y para una plataforma dada.

- *Framework*. Provee un *framework* PIM en tiempo real que se ejecuta por debajo del código PIM.
- Adaptador dependiente del sistema operativo. Consiste en una capa de adaptación ligera que controla la interacción con el RTOS (*Real-Time OS*) subyacente.

El modelo UML comportamental se basa en la tecnología de diagramas de estado de I-Logix. Es una tecnología que ofrece un completo soporte de generación de código para máquinas de estados. El compilador de modelos UML de I-Logix es muy avanazado, y produce código legible y eficiente para los lenguajes y compiladores más habituales en proyectos modulares y de tiempo real. Se puede cambiar de un entorno (Nucleus, VxWorks, OSE, QNX o Windows) a otro con un solo clic.

El punto fuerte de I-Logix es su entorno de ejecución y pruebas. Permite realizar verificaciones de modelos y de código. Además, no sólo permite generar código para una gran variedad de plataformas, sino que también permite realizar pruebas directamente sobre dichos entornos.

Se pueden ejecutar pruebas directamente sobre los diagramas de estado, de secuencia y de objetos, incluso cuando la aplicación está integrada como subsistema en un sistema superior. El desarrollador puede capturar y ejecutar vectores de *test* y ejecutar *test* de regresión en varios entornos.

Es muy interesante la visión propuesta por Rhapsody, ya que sigue la filosofía presentada en el apartado 3.1 de la programación automática, uniendo totalmente la fase de especificación y de creación de prototipo. Además, permite realizar las pruebas sobre los prototipos generados directamente desde la aplicación mediante modelos de simulación muy avanzados.

Otra ventaja que ofrece Rhapsody frente a sus competidores es que da soporte a tecnología *middleware*. Permite utilizar objetos CORBA o COM de forma sencilla, y generar interfaces IDL de forma automática.

6.6. Interactive Objects ArcStyler.

La solución que ofrece Interactive Objects [16] se basa en **ArcStyler** (versión 5.5). Es una de las herramientas MDA comerciales más extendidas. Puede generar código a partir de modelos para cualquier plataforma como .NET o J2EE, siendo una herramienta genérica que nos permite transformaciones de modelo a código sin restricciones. Incluso permite generar ficheros relacionados con las pruebas de la aplicación, de despliegue y ficheros de proyecto para herramientas como JBuilder o Eclipse.

Permite realizar diseños en UML 1.4, aunque es independiente de cualquier versión de UML, ya que se apoya en otra herramienta que le proporciona dicha funcionalidad (MagicDraw). Utiliza diagramas de clases para describir el comportamiento estático del sistema y los diagramas de estados para capturar el comportamiento dinámico de la vista y control del sistema. Se apoya en MOF

para definir sus propios modelos y en XMI para almacenarlos, lo que le permite exportar e importar los distintos modelos utilizados. También contiene un repositorio de modelos accesible a través de JMI.

Las transformaciones en ArcStyler están ligadas a los llamados Cartuchos-MDA (*MDA-Cartridges*). El Cartucho-MDA es un de los elementos que distinguen ArcStyler del resto de de herramientas MDA, y el elemento principal de su arquitectura. Existen cartuchos-MDA para la generación de código J2EE o .NET (entre otros), y funcionan a modo de *plug-in*, por lo que pueden ser integrados otros cartuchos fácilmente.

Los Cartuchos-MDA funcionan como una caja negra que tiene como entrada un PIM y genera a su salida un modelo de código. Sin embargo, un cartucho debe contener más información. Para esto, ArcStyler proporciona un conjunto de marcas que tienen que ser añadidas a los elementos del PIM para dotarlos de dicha información, fundamental para la transformación. Cada marca proporciona la siguiente información:

- Tipo de dato asociado.
- Elementos del modelo a los que se le aplica la marca.
- Valor por defecto.

En resumen, el procedimiento sería el siguiente:

- Como entrada al cartucho-MDA introducimos un PIM.
- Se aplican las Marcas MDA introducidas por el usuario y se modifica el PIM según dichas marcas.
- El cartucho-MDA realiza un mapeo del PIM marcado y genera una salida.

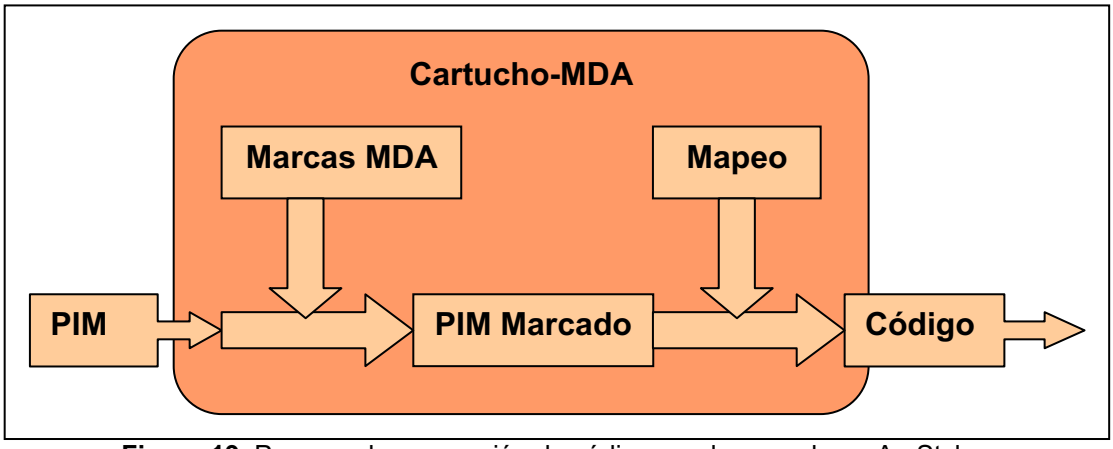


Figura 19. Proceso de generación de código en el que se basa ArcStyler.

Los conjuntos de marcas MDA se encuentran recogidos en lo que ArcStyler denomina perfiles MDA, que son una extensión de los perfiles UML, y que por lo tanto recogen a través de sus marcas los aspectos específicos de la plataforma para la que se genera el código.

Los cartuchos-MDA también incluyen un conjunto de reglas para poder realizar las transformaciones de modelos y, opcionalmente, incorporan verificadores de modelos que aseguran que los modelos de entrada sean válidos para realizar la transformación implementada.

El soporte de creación de cartuchos es complejo y permite incluso herencia. Esto permite extender, redefinir y cambiar las reglas de transformación que nos interesa en cada momento, a partir de un cartucho padre dado.

Una vez claro el concepto de cartucho, se va a centrar la atención en cómo se desarrollan las aplicaciones con ArcStyler.

ArcStyler separa el modelado de negocio de la interfaz con la que interactúa el cliente de la aplicación. Por este motivo se puede decir que provee de dos modelos principalmente.

Para el **desarrollo de interfaces de usuario** existen los componentes de tipo *Accessor*, que son los que modelan dicha interfaz. El modelado de este tipo de componentes se realiza mediante diagramas UML, y dispone de asistentes de diseño, hojas de propiedades y generadores de modelos.

Además de esto, ArcStyler provee de un *framework* de clases disponible para ASP .NET y para JSP. Para cada uno de estos lenguajes existe un cartucho MDA, que permiten el desarrollo de interfaces mediante los siguientes componentes:

- Componente *Accessor*. Representa al controlador dentro del paradigma Modelo-Vista-Control (MVC). Describe el comportamiento dinámico y el control de flujo de la interfaz externa. El modelado del comportamiento se realiza mediante diagramas de actividad UML.

- Componente *Representer*. Representa a la vista dentro del paradigma MVC. Representa tanto páginas JSP como ASP.

- Componente *Web Application*. Representa la unidad de despliegue de la aplicación *web*. En JSP corresponde al fichero WAR que empaqueta componentes *Accessor* y *Representer*. En ASP corresponde al directorio raíz de la aplicación y todos los ficheros necesarios de configuración (no existe un único fichero).

Para la visión estática de la página, el modelado se realiza mediante diagramas de clases que representan las relaciones entre los componentes *Representer* y los componentes *Accessor*.

Para la visión dinámica de la página, se refinan los componentes Accessor mediante diagramas de transición de estados, indicando así las transiciones entre páginas y las acciones a realizar en cada transición.

Para el **desarrollo del modelo de negocio** se proponen diagramas de clases. El diseño de los diagramas es sencillo, aunque el refinamiento de cada uno de

los componentes es lo más complejo, ya que a parte de tener en cuenta conceptos de bases de datos, se debe ir refinando los cartuchos MDA seleccionados. Esto es un proceso bastante laborioso y que fácilmente puede producir errores no intencionados por un mal refinamiento.

La principal desventaja de ArcStyler es su escasa trazabilidad. No es posible saber el destino de un elemento del modelo de entrada en los ficheros de código generados. Sin embargo, para un elemento del modelo si es posible mostrar todos os diagramas que lo contienen.

Centrando la atención en las fases del ciclo de vida que ArcStyler cubre, se puede concluir en que ArcStyler da soporte a todas las fases menos a la gestión de proyectos y captura y seguimiento de los requisitos de la aplicación. Esto se debería de conseguir mediante herramientas externas.

6.7. Compuware OptimalTrace y OptimalJ.

Compuware [17] ofrece, principalmente, tres productos para el desarrollo de aplicaciones y la realización de pruebas, estos son **Optimal Trace** y **OptimalJ** y **QACenter**.

OptimalJ (versión 4.1) es la herramienta que mejor adapta la visión MDA ofrecida por la OMG. Se pueden encontrar bien diferenciados los niveles de Modelo PIM, Modelo PSM y Modelo de código. Permite desarrollar aplicaciones J2EE completas a partir del modelo PIM. Esto es un inconveniente, ya que lo que se pretendía inicialmente era realizar modelos de forma independiente de la plataforma.

Todos los metamodelos de OptimalJ se encuentran definidos bajo MOF, sin embargo amplían la implementación de MOF añadiéndole nuevas operaciones a los elementos de MOF para proporcionar un mejor soporte a la funcionalidad de OptimalJ.

OptimalJ define los tres modelos mencionados anteriormente como Modelo del dominio, modelo de la aplicación y modelo de código, que pasamos a comentar a continuación.

- Modelo de dominio (Domain Model). Es el modelo de más alto nivel y no contiene detalle alguno sobre la implementación. Es el equivalente al modelo PIM de MDA en OptimalJ, y de cara al usuario permite definir diagramas UML 1.3, además de modelos de los servicios que puede ofrecer la aplicación web. La función principal de este dominio es capturar el comportamiento estático de la aplicación. Se subdivide en dos submodelos:
    - Modelo de clases (Domain Class Model). Se basa en diagramas de clases UML, que definen la estructura de la información del dominio. Se puede considerar que este es realmente el modelo que representa el PIM.
    - Modelo de servicios (Domain Service Model). Permite definir vistas sobre las clases definidas en el modelo de clases. Limita el

acceso a las acciones crear, leer, modificar o eliminar instancias, y ocultar la aparición de determinados atributos en la interfaz de la aplicación generada.

- Modelo de aplicación (Application Model). Es el modelo del sistema desde el punto de vista de la tecnología J2EE. Es el equivalente al modelo PSM de MDA en OptimalJ. Se basa en cinco diagramas de clases orientados a la plataforma J2EE, que se generan de forma automática a partir del Modelo de Dominio.
    - Modelo de presentación. Contiene un diagrama de clases especializado en elementos web, que ayudan a la creación de la capa web de la aplicación.
    - Modelo de negocio. Contiene la información relacionada con la manipulación y comportamiento de los datos. Se basa en un modelo de componentes de Enterprise Java Beans (EJB).
    - Modelo de base de datos. Lo componen un esquema relacional que servirá para generar *scripts* en lenguaje SQL, que dará lugar a la generación y acceso de la base de datos de la aplicación.
    - Modelo de fachada de negocio. Es lo que se denomina "modelo puente". Se encarga de comunicar los modelos de negocio y de presentación de datos.
    - Modelos comunes. Contiene estructuras compartidas por el resto de modelos.

- Modelo de código (Code Model). Se genera a partir del modelo de aplicación. Todo el código es generado de forma automática por OptimalJ y es situado en bloques protegidos que no pueden ser modificados por los desarrolladores, aunque también existen bloques libres que pueden ser modificados para añadir funcionalidad.

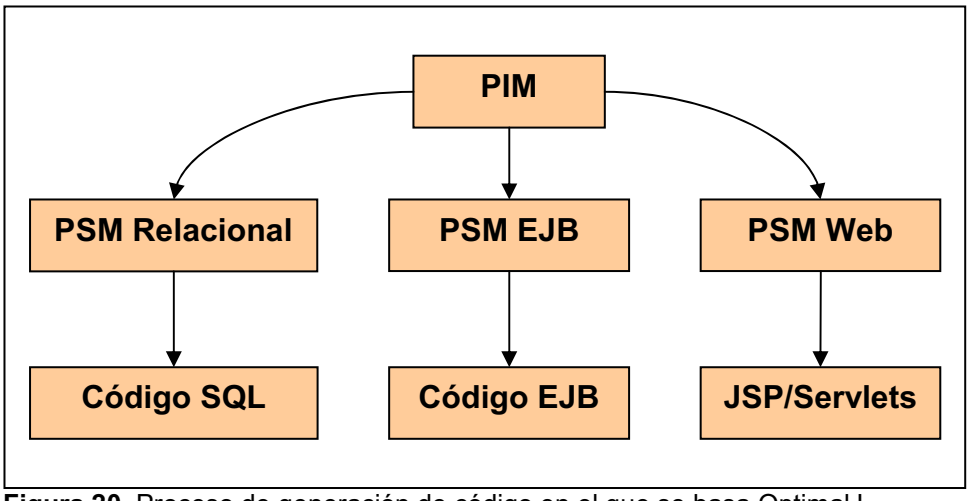


Figura 20. Proceso de generación de código en el que se basa OptimalJ.

Una vez definida la estructura básica, esta se puede modificar para terminar de definir la funcionalidad del sistema. Además, se provee un conjunto de patrones que nos sirven tanto para modificar los modelos de un nivel a otro como para modificarlos internamente. Estos patrones son los siguientes:

- Patrones de transformación entre modelos. Permiten realizar transformaciones entre modelos y son de dos tipos:
    - Patrones de tecnología. Transforman de modelo de dominio a modelo de aplicación.
    - Patrones de implementación. Transforman de modelo de aplicación a modelo de código. Utilizan un lenguaje propio llamado TPL (*Template Pattern Language*). Con TPL se pueden definir distintas formas de generar código JAVA.

- Patrones funcionales. Permiten realizar transformaciones dentro de un mismo modelo, reduciendo el número de errores y acelerando el desarrollo de la aplicación. Se subdivide en tres tipos de patrones, según se esté trabajando a nivel de dominio, de aplicación o de código.
    - Patrones de dominio. Básicamente es un modelo de dominio reutilizable, de manera que se pueda construir rápidamente un nuevo modelo a partir de uno ya existente.
    - Patrones de aplicación. Sigue la misma filosofía que los patrones de dominio.
    - Patrones de dominio. Patrones de bajo nivel aplicados al código.

Como se ha podido observar, OptimalJ se encuentra muy cerca del modelo de tres niveles de MDA, aunque propone su propio modelo para definir las transformaciones.

Por último comentar que fases del ciclo de vida de un producto cubre OptimalJ. La filosofía general de ArcStyler y OptimalJ es tan parecida que cubren las mismas fases, con la salvedad de que en este caso se dispone de una herramienta directamente integrable con OptimalJ que ayuda a cubrir la gestión de proyectos y la captura y seguimiento de los requisitos de la aplicación. Esta aplicación es Optimal Trace.

Optimal Trace (versión 4.1) es una herramienta de captura de requisitos que permite a todas las partes del proyecto colaborar en la definición, comunicación y gestión de los requisitos de un producto *software* a lo largo de todo el ciclo de vida. Esto permite unir y coordinar la parte tecnológica con la de negocio de forma clara y sencilla.

La toma de requisitos es una parte importante a tener en cuenta ya que la mayoría de fracasos en proyectos se deben a un mal análisis de los requisitos. La principal razón es el vacío de coordinación existente entre requisitos de negocio y lo que finalmente se entiende a la hora de plasmar dichos requisitos mediante soluciones tecnológicas.

Optimal Trace ofrece un interfaz gráfico intuitivo que guía al usuario en la escritura de los requisitos en varios pasos y los almacena en un repositorio central, saltando el vacío existente entre las necesidades del usuario y la funcionalidad del sistema, de forma que se plasma la intención del usuario y la respuesta del sistema para cada uno de dichos pasos. También se provee de una visión funcional y una visión no funcional de los requisitos.

Existen dos versiones de Optimal Trace, según las necesidades de la empresa.

La versión *Professional* es la más simple de las dos y sirve para trabajar de forma individual.

La versión *Enterprise* está diseñada para equipos de desarrollo y contiene un conjunto de herramientas de colaboración para la captura y gestión de requisitos. Permite la gestión de varios proyectos desde un único repositorio y un mecanismo de concurrencia que permite a varios miembros del equipo de desarrollo acceder y editar de forma simultánea el proyecto.

Dentro de la versión *Enterprise* se dispone de una versión cliente y otra versión servidor. El cliente puede trabajar de forma similar a como se haría en la versión Professional, sin necesidad de un servidor. De esta forma, se trabaja con ficheros locales y se elimina la funcionalidad multiusuario. Por otro lado, el servidor permite la funcionalidad multiusuario mediante bases de datos MySQL, MS SQL Server 2000/2005 y Oracle 9i.

Sin embargo, el aspecto más novedoso a tener en cuenta no tiene nada que ver con la funcionalidad que ofrece en la gestión de requisitos, sino que es completamente integrable dentro de un entorno de desarrollo MDA y herramientas de *test*. Optimal Trace permite ser integrado dentro de las siguientes herramientas:

- Mercury Quality Center.
- Compuware OptimalJ.
- Compuware QACenter.
- Borland Together.
- IBM Rational Rose.

De este conjunto de herramientas para pruebas hay que resaltar **QACenter**, que también es la herramienta que ofrece Compuware para realizar pruebas sobre las aplicaciones realizadas. La integración con Optimal Trace se realiza mediante lo que se denominan “casos de *test*”, que son casos de uso orientados a pruebas.

QACenter Enterprise Edition integra cinco productos para cubrir todos los tipos de pruebas posibles:

- Para automatizar pruebas funcionales y de regresión, incluye la herramienta QARun/TestPartner.

- Para optimizar la gestión de las pruebas, integra la herramienta QADirector.

- También integra una herramienta de gestión de requisitos llamada Reconcile. Aunque para este caso ya se optó por la solución Optimal Trace (recién adquirida por Compuware).

- Para la búsqueda de defectos en el código integra la herramienta TrackRecord.

- Por último, y de forma opcional, se puede encontrar una herramienta para la gestión de los datos resultantes de las pruebas File-AID/CS. Es más una herramienta para ofrecer información que para realizar pruebas.

6.8. Soluciones de código abierto.

También se han empezado a incorporar al mundo de MDA herramientas de desarrollo de código libre, aunque la mayoría más que herramientas son utilidades o accesorios para otras herramientas. Son, en su mayoría, librerías que se añaden a herramientas de diseño UML para realizar transformaciones directas de UML a código, o transformaciones de modelos XMI a código. En el siguiente listado se presentan algunas de las más comunes junto con sus principales características:

- *UML Model Transformation Tool* (UMT). ES una herramienta que permite la transformación de modelos y la generación de código basada en especificaciones UML/XMI.

- *ATLAS Transformation Language* (ATL). Lenguaje de transformación desarrollado por el equipo de INRIA Atlas.

- *MTL Engine*. Implementación de QVT desarrollada por INRIA Triskel para Netbeans MDR y Eclipse EMF, basada en el lenguaje MTL.

- *Generative Model Transformer* (GMT). Es un proyecto basado en la Plataforma Eclipse, que proporciona tecnología para la transformación de modelos bajo.

- *OpenMDX*. Es un entorno MDA que genera código para las plataformas J2EE y .Net.

- *AndroMDA*. Herramienta basada en plantillas para la generación de código J2EE desde modelos UML/XMI. Se basa en un lenguaje script llamado *Velocity Template Engine* (VTL) y utiliza como modelos NetBeans MDR.

- *Xdoclet*. Herramienta para la generación de código J2EE, basada en atributos. Aunque no está basada en modelos, puede ser combinada con otras herramientas, como UMT, para lograr resultados basados en modelos.

- Middlegen. Es un una herramienta para la generación de código dirigido por bases de datos. Por si misma no tiene gran utilidad, pero se puede convinar con VTL, *Xdoclet* y ANT para darles un valor añadido.

- OMELET. Es un proyecto basado en la plataforma Eclipse que trata de proporcionar un *framework* de carácter general que integre modelos, metamodelos y transformaciones.

- Openmodel, es un *framework* basado en MOF/JMI para herramientas MDA.

Como se puede apreciar del listado, no existen herramientas completas que permitan realizar un análisis, diseño e implementación, como cabe esperar de una herramienta MDA. Son simples utilidades y *frameworks* que en cierto modo se podrían utilizar para realizar aproximaciones al desarrollo MDA.

Como conclusión para este breve apartado, si MDA está en periodo de maduración, las herramientas *open source* MDA aún no han llegado a ese punto, aunque convendría seguir la evolución de dichas herramientas, ya que son muchas, de código abierto y están en continuo desarrollo.

7. Elección de herramienta según modelo de desarrollo.

De este apartado se desprenderá finalmente la herramienta que cubrirá la parte más laboriosa del proyecto, que es el desarrollo de la aplicación. Por las características de las herramientas de desarrollo MDA, la herramienta resultante cubrirá las fases de análisis, diseño e implementación, por lo que serán necesarias herramientas complementarias para la gestión de requisitos y realización de pruebas.

Para la elección de la herramienta se tendrán en cuenta unos ciertos criterios, de entre los que destaca el Soporte del ciclo de vida del desarrollo, ya que una vez elegida la herramienta esta debe dar soporte a la mayor cantidad de fases del ciclo de vida posibles, y en caso de que no de soporte a todas, debe permitir su fácil integración con otras que lo complementen.

El modelo de desarrollo elegido ha sido, como ya se ha venido adelantando, el enfoque MDA. Por este motivo, todas las herramientas bajo a estudio deben cumplir, en la medida de lo posible, el estándar propuesto por el OMG.

De todas las propuestas, se han elegido tres herramientas para su estudio. Estas han sido Rhapsody, ArcStyler y OptimalJ, que ya fueron introducidas en los apartados 6.5, 6.6 y 6.7. Los motivos para la elección de estas herramientas han sido los siguientes.

Rhapsody ofrece dos conceptos interesantes que no ofrecen el resto de herramientas. Estos son el concepto de multiplataforma enfocado al sistema operativo y al lenguaje final de la implementación, y los modelos de simulación integrados. Además, también es interesante el enfoque MDA no orientado al desarrollo de aplicaciones *web*.

Se ha seleccionado ArcStyler por el mecanismo refinamiento de cartuchos MDA. Esta es una solución alternativa a la ofrecida por el OMG para lograr una aplicación multiplataforma a partir de un modelo de entrada independiente.

La selección de OptimalJ viene motivada por ser la herramienta que mejor adapta la visión del OMG sobre MDA.

En el primer apartado, se analizarán los criterios de evaluación de herramientas para, posteriormente, dar paso al desarrollo de cada uno de dichos criterios aplicándolos a las tres herramientas en cuestión.

7.1. Criterios para el estudio.

Es importante saber elegir los criterios de comparación que se usarán, ya que serán los que ayuden a elegir una solución u otra. Unos serán más relevantes que otros dependiendo de que implementación final se quiera y la filosofía de trabajo que se quiera seguir. A continuación, se van a definir los criterios que se entienden como más representativos:

- **Similitudes con la arquitectura MDA**. Es un factor de relevancia que el entorno de desarrollo siga las directrices propuestas por el OMG en lo que a la arquitectura se refiere. Se tendrá en cuenta que se puedan diferenciar los niveles propuestos en MDA.

- **Multiplataforma**. Como ya se ha esbozado con anterioridad, puedo ser o no un factor crítico. Depende de la línea de trabajo que se vaya a seguir en el futuro. Si el entorno final va a ser siempre único, puede que no sea si quiera un factor positivo. En cambio, si las aplicaciones a implementar pueden variar de entorno, o se desea crear una aplicación para varios entornos, será un factor altamente decisivo. También se ha de tener en cuenta que si soporta multiplataforma, la herramienta se aproxima más a las directrices propuestas por el OMG.

- **Compatibilidad con la plataforma Eclipse**. Ya que en la línea actual de desarrollo del Servicio de Informática se cuenta con el entorno de desarrollo Eclipse, sería importante analizar el grado de compatibilidad que permiten las herramientas bajo estudio.

- **Desarrollo de aplicaciones *web***. El entorno seleccionado tras el estudio debe ser capaz de desarrollar aplicaciones web, ya que la finalidad que se persigue es el desarrollo de este tipo de aplicaciones.

- **Definición de transformaciones**. En este apartado se tendrá en cuenta la capacidad de extensión y personalización de las transformaciones entre modelos. Se medirá el grado de flexibilidad y complejidad.

- **Verificación de modelos**. Medirá la capacidad de las herramientas de comprobar si un modelo se ajusta a las reglas de diseño o no.

- **Facilidad de comprensión de los modelos**. Se atenderá a la presentación visual de la jerarquía de los modelos, claridad a la hora de presentar los diagramas y sus relaciones, facilidad de navegación entre modelos,...

- **Herramientas de soporte integradas**. Además del entorno de programación en sí, se deben tener en cuenta las herramientas que ya vienen integradas con el entorno de desarrollo, así como la facilidad de uso y configuración y la posibilidad de integrar herramientas nuevas. Se tendrán en cuenta editores de código, capacidad de integrar servidores *web* y de base de datos de pruebas, generación de clientes y servidores de prueba,...

- **Soporte para consistencia incremental**. Es una característica muy importante para el desarrollador. Tiene en cuenta la capacidad de conservación de variaciones en el código final tras la realización de nuevas transformaciones, lo que vulgarmente se podría denominar como “no machacar código”.

- **Soporte del ciclo de vida de desarrollo**. Capacidad de integración dentro de un conjunto de herramientas que en conjunto cubran todas las fases del ciclo de vida de un producto *software*, donde el producto de una herramienta es directamente la entrada para otra.

- **Trazabilidad**. Capacidad de saber que componente de un modelo inicial dio como resultado un componente de un modelo resultante tras una transformación.

- **Control y refinamiento de transformaciones**. Mide el grado de control y personalización de las transformaciones entre modelo.

7.2. Desarrollo del estudio.

Aplicando cada uno de los criterios anteriormente comentados sobre las herramientas OptimalJ (versión 4.1), ArcStyler (versión 5.5) y Rhapsody (versión 6.2), se obtienen las siguientes conclusiones:

- **Similitudes con la arquitectura MDA.**

De las tres herramientas estudiadas, la única herramienta que cumple con un desarrollo en tres niveles (modelo independiente de plataforma, modelo dependiente de plataforma y modelo de código) es OptimalJ. Como ya se conoce del estudio de la herramienta, OptimalJ cuenta con un nivel de modelo (equivalente a PIM), un nivel de aplicación (equivalente a PSM) y un modelo de código. Lógicamente, también cuenta con transformaciones PIM-PSM, PSM-Código.

Tanto ArcStyler como Rhapsody cuenta sólo con un nivel PIM, un nivel de código y una transformación PIM-Código, rompiendo con el esquema propuesto por el OMG.

- **Multiplataforma.**

A priori, ya se descarta OptimalJ de este análisis, ya que se sabe de antemano que es una herramienta de plataforma única (J2EE). Por este motivo, el análisis se debe centrar en Rhapsody y ArcStyler.

Con la versión estándar de ArcStyler se dispone de cartuchos para realizar transformaciones de modelos PIM a J2EE y .NET. Se debe recordar que existen transformaciones para el modelo de negocio y para interfaces de usuario (*Accessors*). A la hora de realizar una aplicación *web* completa ambas transformaciones son necesarias.

En el caso de Rhapsody, se dispone de transformaciones de nivel de modelo a codigo C/C++, ADA y Java. En esta herramienta, el concepto multiplataforma no se refiere a distintos lenguajes de programación, sino, a distintos entornos de ejecución (Rhapsody permite generar aplicaciones para distintos sistemas operativos a partir de un mismo modelo). Es uno de los problemas que provoca

la falta de definición del estándar MDA, que no ofrece una definición clara del concepto multiplataforma.

- **Compatibilidad con Eclipse.**

La herramienta más aventajada en este apartado, sin duda alguna OptimalJ. OptimalJ es totalmente compatible con Eclipse ya que está desarrollado sobre dicha plataforma, por lo que no existiría ningún problema de compatibilidad.

El caso de ArcStyler es distinto. En principio, ArcStyler no es compatible con Eclipse, aunque existe una implementación como *plug-in* para IBM *Rational Software Modeler* (RSM).

IBM RSM, que ya se introdujo en el apartado 6.4, es una herramienta de modelado UML construido sobre la plataforma Eclipse (igual que OptimalJ), y al añadir el *plug-in* de ArcStyler se convierte en una especie de herramienta MDA, aunque se pierden muchas de las facilidades que ofrece ArcStyler en su versión original y hace muy engorroso el desarrollo.

En el caso de Rhapsody v.6.2, existe un *plug-in* para Eclipse, pero que genera los mismo problemas que el *plug-in* de ArcStyler para RSM (pérdida de utilidades y mayor dificultad de desarrollo).

En definitiva, la mejor solución para la compatibilidad con Eclipse es contar con una herramienta construida directamente sobre dicha plataforma, y utilizar los *plug-ins* como complementos.

- **Desarrollo de aplicaciones *web*.**

De las tres herramientas, la única herramienta no orientada a desarrollo *web* es Rhapsody. Los productos desarrollados por Rhapsody están orientados principalmente al desarrollo de aplicaciones Real-Time (RT) multiplataforma.

En contraposición, la función principal de OptimalJ y de ArcStyler es generar aplicaciones *web*, por lo que cumplen perfectamente con este requisito.

- **Definición de transformaciones.**

Por transformaciones no sólo se entiende generación de código para una u otra plataforma. Una transformación es la generación de unos objetos destino a partir de unos objetos origen, y esto es directamente aplicable a las transformaciones entre modelos.

OptimalJ no permite generar código para distintas plataformas, pero si transformar modelos. Se pueden transformar un modelo de clases a uno de servicio (transformaciones dentro del modelo de dominio) o transformar los submodelos del modelo de dominio a modelos de aplicación. En la futura versión 4.2 de OptimalJ, se incluirá una herramienta para la edición de transformaciones, de forma que se puedan personalizar las transformaciones existen y crear nuevas transformaciones.

Ya se sabe que ArcStyler utiliza los cartuchos MDA para realizar sus transformaciones. Esto está relacionado de forma intrínseca con la capacidad multiplataforma. Para poder realizar transformaciones a distintos lenguajes es necesario disponer de un cartucho MDA para este propósito. Con la herramienta ya se dispone de dos cartuchos (para .NET y J2EE), aunque es posible descargar otros cartuchos o desarrollarlos de forma personalizada.

En el caso de Rhapsody, sólo se ha podido observar que a la hora de realizar la instalación, se pregunta que paquetes de transformaciones se desea instalar. Tras realizar pruebas y leer diversa documentación anexa a la herramienta, no se ha podido encontrar una forma de extensión de las transformaciones, dejando como únicas posibilidades las transformaciones a C/C++, JAVA y ADA, que no son pocas.

En cuanto a los refinamientos necesarios para realizar las transformaciones, ArcStyler dispone del marcado del PIM y su posterior mapeo. Conforme se va desarrollando la aplicación, se van realizando refinamientos específicos de cada uno de los componentes de la aplicación. Estos refinamientos se pueden hacer de forma simultánea para todos los cartuchos disponibles, y finalmente realizar las transformaciones que se deseen.

El caso de Rhapsody es distinto. El lenguaje final se elige a la hora de crear el proyecto, aunque como se mencionó en el análisis del criterio de multiplataforma, Rhapsody entiende por plataforma el sistema operativo final en el que se va a ejecutar. En el caso de que se desee generar para una u otra plataforma, si se puede variar en cualquier momento de forma sencilla.

- **Verificación de modelos.**

Las tres herramientas analizadas disponen de herramientas para la verificación de los modelos desarrollados, aunque en el caso de ArcStyler y Rhapsody se limita a verificar los metamodelos en el nivel más alto de abstracción, ya que no dispone de un nivel intermedio entre PIM y código.

Se debe destacar que la verificación de modelos en OptimalJ es más compleja, ya que su modelo de aplicación se subdivide en cinco modelos (modelo de presentación, modelo de negocio, modelo de base de datos, modelo de fachada de negocio y modelos comunes), aunque normalmente no es necesario realizar verificaciones si no se realizan modificaciones en a nivel de modelo de aplicación.

- **Facilidad de comprensión de los modelos.**

Una vez más Rhapsody y ArcStyler son las herramientas que más se parecen, en este caso respecto a la presentación de los modelos en desarrollo. Rhapsody permite implementar todos los diagramas contemplados en UML, incluyendo diagramas de secuencia, clases, objetos, actividad, casos de uso,... En el navegador de modelos existen unos paquetes por defecto donde se intuye que tipo de componentes se encontrarán en su interior, pudiendo crear nuevos paquetes para una mejor organización de los modelos. Una de las

principales diferencias con todas las herramientas estudiadas es la animación de los modelos dinámicos, que permite obtener una demostración dinámica de como se comporta el modelo implementado.

La apariencia de ArcStyler es muy parecida a la de Rhapsody, aunque se centra principalmente en dos tipos de diagramas UML, con los que se puede modelar toda la aplicación. Los diagramas fundamentales son el diagrama de clases y los diagramas de actividad. Los diagramas de clases tienen dos propósitos fundamentales según se este implementando el modelo de negocio o la interfaz de usuario:

- En el modelo de negocio, la finalidad de los diagramas de clases es obtener una estructura de datos relacional, que ayuda a implementar y comprender la base de datos final.
- En el desarrollo de interfaz de usuario ayuda a obtener una visión rápida de la relación entre los componentes que intervendrán en la presentación final de la aplicación, dejando la visión dinámica a otro tipo de diagramas.

Los diagramas de actividad son muy útiles para representar las transiciones entre elementos, siendo el principal motor del comportamiento dinámico de las interfaces de usuario.

Es importante destacar que en Rhapsody y ArcStyler no se contempla una arquitectura en tres capas, luego toda la implementación se hace directamente sobre el modelo PIM. Por este motivo es muy importante el orden a la hora de crear los diagramas y agruparlos. En algunas aplicaciones el número de diagramas en la fase de modelado puede ser elevado, y una mala ordenación puede dar lugar a la completa incomprensión del modelo.

En OptimalJ todo se reduce a diagramas de clases y vistas sobre las clases. De esto se encargan las dos submodelos del modelo de dominio (modelo de clases y modelo de servicios), teniendo en cuenta el modelo de proceso cuando se desean modelar actividades dentro de la aplicación. Además, la arquitectura en tres capas ayuda a una mejor comprensión y seguimiento de las implementaciones. Como resultado de esto, es realmente fácil comprender las implementaciones de alto nivel.

- **Herramientas de soporte integradas.**

En el caso de Rhapsody, destacan sobre todo los editores UML disponibles, uno por cada tipo de diagrama UML permitido:

- Diagramas de objetos.
- Diagramas estructurales (clases, objetos, bloques, puertos y dependencias).
- Diagramas de actividad.
- Diagramas de estado (*statecharts*).
- Diagramas de secuencia.
- Diagramas de colaboración.

- Diagramas de casos de uso.
- Diagramas de colaboración.
- Diagramas de componentes.
- Diagramas de despliegue.

Se contempla el uso de más diagramas ya que en este caso no se modelan aplicaciones *web*, si no aplicaciones en tiempo real. Por esto son necesarios más diagramas para modelar el comportamiento de la aplicación, como por ejemplo los diagramas de secuencia.

La herramienta a destacar es Animator. Permite realizar animaciones de los modelos dinámicos para analizar su comportamiento. Es una gran herramienta para la realización de pruebas y detección de errores de diseño.

Dispone de herramientas para realizar ingeniería inversa y editores de código. También permite enlazar el código a editar con Eclipse IDE, si se encuentra disponible.

También dispone de generador de código XMI y da soporte a aplicaciones distribuidas en CORBA y COM.

Por último, contiene interfaces para la conexión con otras aplicaciones como telelogic DOORS y Rational Rose.

ArcStyler también permite el uso de un gran número de herramientas, aunque en su mayoría son herramientas externas que requieren de una costosa configuración. Integra servidores *web*, como Tomcat, y *scripts* ANT para permitir un entorno de pruebas más flexible.

En ArcStyler es bastante rápido desplegar un entorno de pruebas de aplicaciones *web* en las que no intervengan bases de datos. En el caso de que se requiera desplegar una base de datos de pruebas, el proceso no es tan rápido.

El gran fallo de ArcStyler es que no dispone de un editor de código integrado. Además, no da ninguna opción a integrar uno dentro de la propia aplicación, por lo que el proceso de edición de código y personalización es bastante engorroso.

OptimalJ es la herramienta más aventajada en este aspecto, ya que dispone de todo tipo de herramientas integradas. Además, OptimalJ está construido sobre la plataforma Eclipse, que ofrece una interfaz de *plugins* altamente extensible. Dispone de editores de código, editores gráficos de diagramas UML de clases y casos de uso, de servidor *web* completamente integrado, clientes de bases de datos con generadores automáticos de *scripts* para bases de datos y de una base de datos de pruebas que se instala con la aplicación (SOLID). Todas las herramientas trabajan perfectamente de forma conjunta y facilitan el trabajo del diseñador, desarrollador y probador.

- **Soporte para consistencia incremental.**

Tanto OptimalJ como ArcStyler diferencian entre bloques libres y bloques protegidos. Una vez realizado un cambio en el modelo, sólo se modifican los bloques protegidos al regenerar el código a partir del modelo, por lo que la personalización del código queda protegida.

En Rhapsody no se ha observado esta filosofía, aunque a partir de sus herramientas de ingeniería inversa se pueden modificar los modelos a partir del código fuente.

Por medio de ambas soluciones se puede conseguir consistencia entre el código final y el modelo de partida.

- **Soporte del ciclo de vida del desarrollo.**

Las tres herramientas aplicaciones intentan dar soporte a todas las fases del ciclo de vida del producto, aunque la herramienta más perjudicada en este aspecto es ArcStyler, ya que no integra todas las herramientas necesarias para la edición de código y despliegue de un prototipo.

Las herramienta más recomendadas para esta característica son OptimalJ y Rhapsody, ya que integra todas las herramientas necesarias para análisis, diseño, implementación y pruebas, saliendo aventajada OptimalJ, debido a todas las herramientas integradas para pruebas y su alta capacidad de extensión mediante *plugins* para Eclipse.

El único problema detectado en OptimalJ y Rhapsody es la falta de una herramienta para la captura y seguimiento de requisitos, aunque al menos la captura se puede conseguir mediante diagramas de casos de uso. Otra solución a este problema, en el caso de OptimalJ, sería extender la herramienta mediante algún *plug-in* para Eclipse, por ejemplo JRequire, u optar por alguna solución comercial como Optimal Trace (analizada en el apartado 6.7). También se puede mejorar la fase de pruebas en OptimalJ si se integra con QACenter.

- **Trazabilidad.**

En esta característica, la herramienta dominante es OptimalJ, ya que es la única herramienta que trabaja a tres niveles y realiza todas las transformaciones mediante patrones de transformación. Gracias a este registro, se puede conocer qué elemento del PIM corresponde a un determinado elemento del PSM y su destino en el código final.

- **Control y refinamiento de las transformaciones.**

La única herramienta de las tres estudiadas que, por el momento, permite realizar un refinamiento completo es ArcStyler. Es la gran ventaja de la utilización de cartuchos MDA, que de verdad aportan una solución

multiplataforma. El problema de los cartuchos MDA es que alarga el proceso de implementación del modelo y su extensión es bastante compleja y laboriosa.

La tabla que se muestra a continuación presenta un resumen de los resultados de la comparativa, indicando el grado de cumplimiento del criterio con las calificaciones alta, media, baja.

Criterio	ArcStyler	OptimalJ	Rhapsody
Similitudes con MDA.	Baja	Alta	Baja
Multiplataforma.	Alta	Baja	Alta
Compatibilidad Eclipse.	Media	Alta	Media
Desarrollo aplicaciones *web*.	Alta	Alta	Baja
Definición de transformaciones.	Alta	Alta	Media
Verificación de modelos.	Alta	Alta	Alta
Facilidad de comprensión modelos.	Media	Alta	Media
Herramientas de soporte integradas.	Media	Alta	Media
Soporte para consistencia incremental	Alta	Alta	Alta
Soporte ciclo de vida	Baja	Alta	Alta
Trazabilidad	Baja	Alta	Baja
Control y refinamiento de transformaciones	Alta	Media	Baja

Tabla 2. Comparativa de herramientas MDA.

En la comparativa se aprecia que dos puntos importantes como las similitudes con MDA y la trazabilidad son asignaturas pendientes para ArcStyler y Rhapsody. El motivo por el que no cumplen estas características se encuentra relacionado con el no cumplimiento del modelo de desarrollo en tres niveles propuesto por el OMG. Al saltar directamente del modelo independiente de la plataforma al código no se puede saber que componentes del modelo generaron ciertas partes de código.

El único problema de OptimalJ es que no permite generar código para distintas plataformas, aunque esto no es un problema si se tiene claro que la solución será en código Java.

Un punto que no se comprende es por qué ArcStyler no integra una herramienta para la edición de código. Esto afecta a la comodidad del desarrollador, ya que debe tener por un lado ArcStyler y por otro un entorno de desarrollo para editar el código generado.

Hay que destacar que las herramientas tienden a dar soporte a la consistencia incremental mediante la distinción entre bloques libres y bloques protegidos. Esta es la solución que aportan ArcStyler y OptimalJ, y es muy eficiente.

En términos generales, fuera del estudio comparativo de estas tres herramientas, se aprecian unos ciertos inconvenientes en como aplican las compañías el enfoque MDA. Unas tienden a cumplir unas de las propuestas del OMG y desatender otras. En el caso de ArcStyler y OptimalJ se utiliza como lenguaje de metamodelado MOF, tal y como propone el OMG. En cambio IBM Rational propone como lenguaje de metamodelado EMF, divergiendo del resto de compañías.

El caso de las transformaciones entre modelos es otro tema de discusión. Cada compañía está desarrollando tecnologías propias para la implementación de transformaciones. Véase el caso de ArcStyler y OptimalJ. ArcStyler ha introducido OCL para el filtrado de la información en sus transformaciones mientras que OptimalJ ha implementado un nuevo lenguaje llamado XMOF, que, aunque también contempla OCL como lenguaje de restricción, es una solución unilateral de Compuware.

Un tercer problema, es el de la arquitectura en tres niveles. No todas las compañías la están integrando y algunas siguen con la filosofía de transformar directamente el modelo de análisis en el código de la implementación, cuando la idea de una plataforma independiente del modelo y otra dependiente es una gran solución. Mientras que Borland y Compuware siguen la filosofía en tres capas, Interactive Objects o Telelogic apuestan por una transformación directa.

Todas estas divergencias pueden desencadenar en un estancamiento, o el abandono, de la propuesta del OMG y el triunfo de otros enfoques como el ofrecido por Microsoft mediante sus Factorías de *Software*. Si esto sucediera, no sería un paso hacia delante, sino un estancamiento, ya que el enfoque de las Factorías de *Software* no es algo nuevo. Si MDA siguiera adelante, y las compañías consiguieran converger hacia una solución, seguramente, Microsoft acabaría por integrar MDA con Factorías de *Software* y se realizaría finalmente el verdadero salto cualitativo en el desarrollo del *software* llegando finalmente a la industrialización del mismo.

8. Solución propuesta.

Atendiendo a la comparativa realizada en el apartado anterior, se aprecia claramente que la herramienta que cumple mayoritariamente los criterios es OptimalJ, y está será la solución propuesta a partir de la cual se seleccionan el resto de herramientas complementarias para cubrir el resto de fases.

Además de las características mencionadas en el estudio comparativo, hay otro factor importante a destacar, aunque no haya sido decisivo para la elección de la herramienta. Este factor es que en la próxima edición de OptimalJ (versión 4.2) se incluirá la persistencia de objetos Java mediante Hibernate, además de la ya disponible con EJB. Es un buen motivo también para elegir OptimalJ, ya que ayudaría a no alejar la solución de la línea de trabajo actual.

Siguiendo en la línea del documento, se presenta el modelo de ciclo de vida propuesto, junto con las herramientas propuestas para cubrir cada fase:

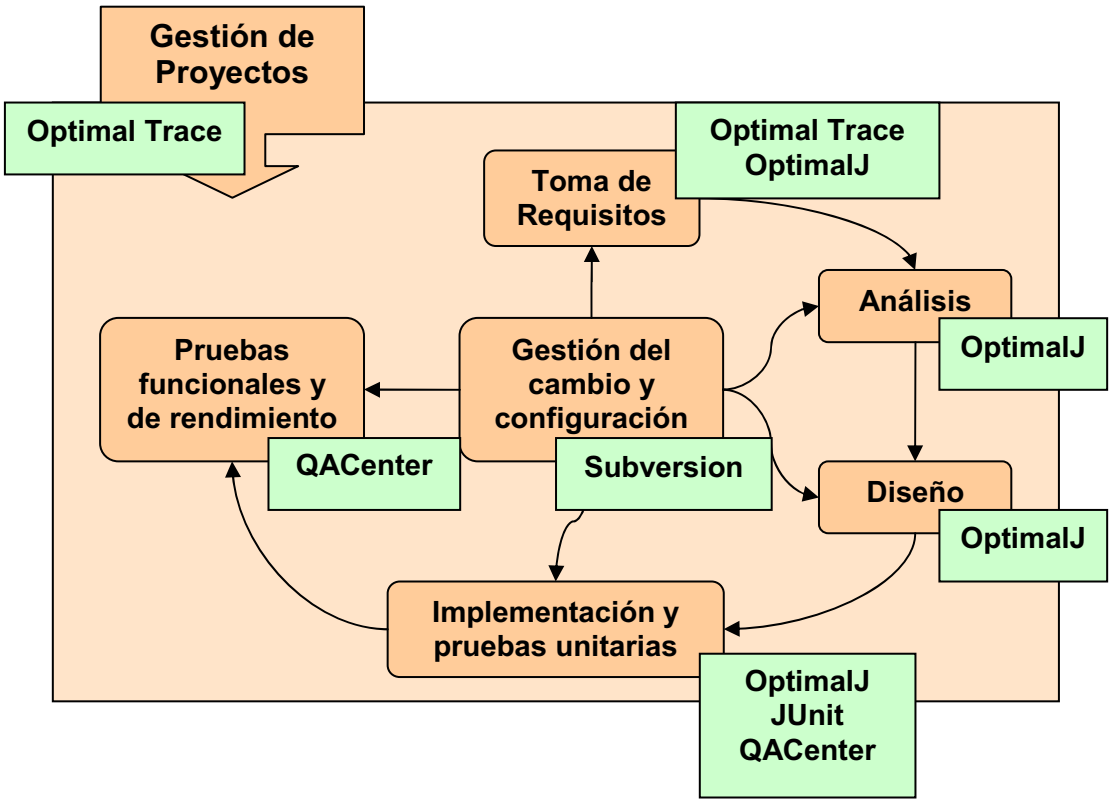


Figura 21. Modelo propuesto en apartado 5 cubierto por herramientas propuestas en la solución final.

Las únicas herramientas comerciales, que supondrían un desembolso económico, son Optimal Trace, OptimalJ y QACenter. Subversion, y JUnit son herramientas de libre distribución que no supondrían coste alguno.

El proceso de desarrollo sería, generar en primer lugar los requisitos a partir de Optimal Trace, que permite realizar un seguimiento de los mismos a lo largo de

todo el ciclo de vida. Una vez generados, se pueden exportar a un fichero XMI para posteriormente ser utilizados por OptimalJ.

Una vez importados los requisitos en OptimalJ, ya es posible comenzar con el resto de fases, aunque se puede considerar que el proceso de análisis, diseño e implementación son uno mismo, ya que esa es la filosofía MDA.

Realizada la implementación, en el modelo de código se pueden añadir las librerías de JUnit y crear una batería de pruebas unitarias para aplicación implementada, aunque la opción más correcta sería seguir en la línea de productos de Compuware e implementar las pruebas utilizando QACenter.

El repositorio de versiones seleccionado ha sido Subversión, ya que se tiene experiencia con este repositorio y combina eficiencia con sencillez. Se debería configurar de tal forma que gestionara de forma correcta los ficheros de OptimalJ. Esto se explica detenidamente en el apartado 9.2.7.

En el siguiente apartado, se va a realizar un pequeño proyecto utilizando las herramientas que componen la espina dorsal del entorno de desarrollo propuesto. Estas son Optimal Trace y OptimalJ. No se ha incluido dentro de las pruebas QACenter, ya que no se ofrece ningún *trial* de esta herramienta.

9. Desarrollo de una aplicación mediante Optimal Trace y OptimalJ.

Antes de comenzar con el desarrollo de la aplicación se deben realizar un par de aclaraciones. La primera de ellas es que no se van a utilizar las versiones comerciales de Optimal Trace y OptimalJ, ya que no se dispone de ellas. En su defecto se utilizarán para el desarrollo dos versiones *Trial* de OptimalJ 4.1 y Optimal Trace 4.1.

Por otro lado, es importante recordar que son dos herramientas muy jóvenes y que aún se encuentran en periodo de crecimiento. En el caso de Optimal Trace, es una herramienta recientemente adquirida por Compuware y está en periodo de integración con el resto de herramientas, por lo que su funcionalidad es aún algo limitada. En el caso de OptimalJ, se encuentra a las puertas del lanzamiento de una muy prometedora nueva versión. Entre las ventajas de dicha nueva versión se encuentra el motor de desarrollo de patrones de transformación, la inclusión de esquemas de desarrollo para nuevas arquitecturas de aplicaciones y la posibilidad de permitir persistencia de objetos mediante tecnologías distintas a EJB (por ejemplo, mediante Hibernate).

Teniendo estas limitaciones en mente, se pasa a la explicación del proyecto que se va a realizar. Este se basa en el proyecto DumboV7 desarrollado por el Servicio de Informática de la Universidad Politécnica de Cartagena. Será una versión muy reducida de dicha aplicación, donde se podrán crear una estructura que permita lo siguiente:

- Crear sistemas Dumbo, y poder realizar algunas operaciones sobre dichos sistemas.
- Crear secciones que dependan de un sistema Dumbo, y poder realizar operaciones sobre las secciones creadas.
- Crear aplicaciones que dependan de una sección, y poder realizar operaciones sobre las aplicaciones.

A la misma vez que se crea la aplicación, se pretende ofrecer información sobre las características más importantes de las dos herramientas seleccionadas.

9.1. Captura de requisitos mediante Optimal Trace.

Como ya se comentó en la introducción del apartado 9, Optimal Trace es una herramienta que recién adquirida por Compuware, por lo que su funcionalidad está en periodo de ampliación. En principio se pretende que, además de capturar, gestionar y presentar requisitos, se pueda hacer un seguimiento de estos a lo largo de todo el periodo de desarrollo de la aplicación. En un futuro también promete realizar un control de los recursos asignados para el cumplimiento de cada uno de los requisitos y la creación del modelo inicial de desarrollo de OptimalJ. Actualmente, también existe Optimal Trace Server, que permite mantener todos los análisis de requisitos de todos los proyectos en un mismo servidor, lo que facilita la gestión de los proyectos.

Una buena primera aproximación a la herramienta es observar su aspecto, que es lo que se muestra en la siguiente figura:

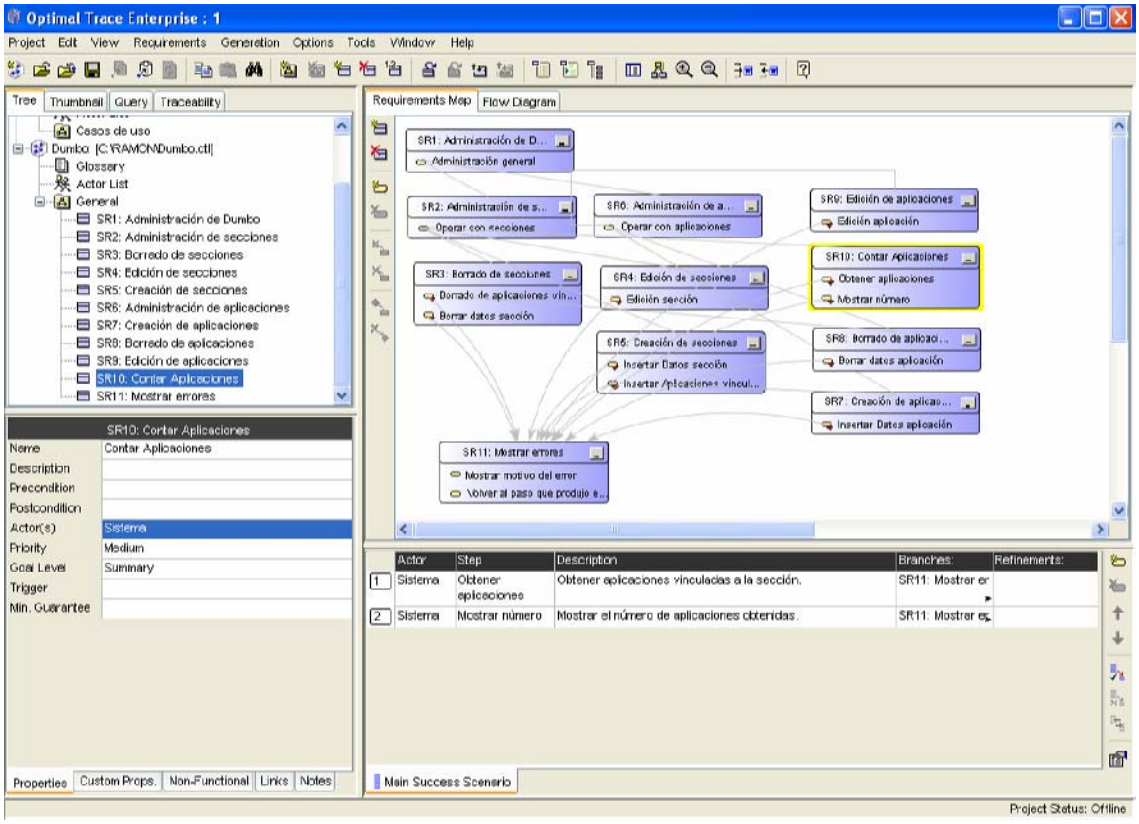


Figura 22. Entorno Optimal Trace.

La herramienta se divide en cuatro sectores. En la parte superior izquierda se encuentra el navegador del proyecto que ofrece información general sobre este. En la parte superior derecha se encuentra la zona de trabajo, donde se generan los diagramas que dan forma al conjunto de los requisitos y permite mostrar las relaciones entre estos. En la parte inferior izquierda se muestran las características propias de cada uno de los requisitos, donde estos pueden ser personalizados. Por último, en la parte inferior izquierda se encuentra el editor de pasos (*steps*) de los requisitos.

Un requisito se compone de un nombre, unas propiedades funcionales, unas propiedades no funcionales y un número de pasos a seguir para su cumplimiento (entre otras características menos relevantes). Además, los requisitos se pueden relacionan entre sí, aunque puede haber requisitos independientes que no dependan de ningún otro. Debido a estas relaciones, en algunos casos, para poder cumplir con éxito un requisito se deben cumplir otros antes. Por esto, existe el concepto de trazabilidad, que expresa todos los requisitos que se deben cumplir antes de que otro pueda darse por finalizado.

Esta relación entre requisitos se vuelve más compleja cuando se introducen bifurcaciones (*branches*) y escenarios alternativos. Una bifurcación puede alterar el orden de cumplimiento de los pasos si se produce un evento en particular. A la misma vez, un requisito puede contener más de un escenario,

esto es, un mismo requisito puede realizar pasos distintos según el escenario en el que se encuentre. Además, las bifurcaciones dentro de un requisito pueden llevar a un cambio de escenario. Por este motivo la trazabilidad de los requisitos que contengan bifurcaciones y escenarios es más compleja. Como es un tema complejo, también se muestra el siguiente ejemplo de requisito con dos escenarios y dos bifurcaciones:

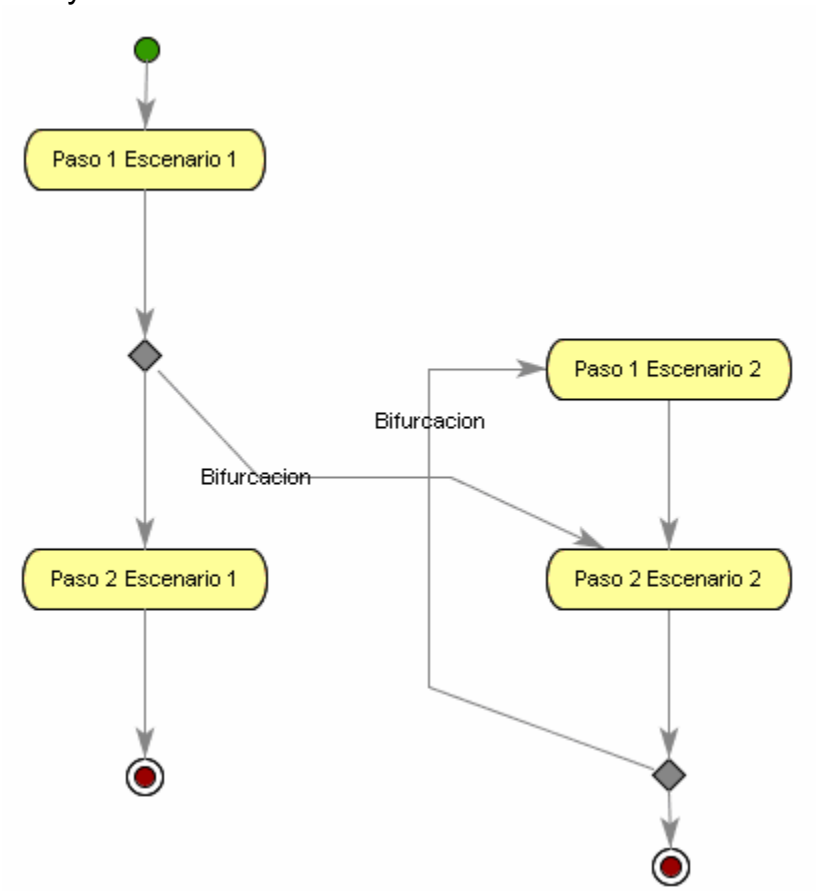


Figura 23. Ejemplo de diagrama de flujo en Optimal Trace.

Antes de seguir adelante y presentar los requisitos de la aplicación Dumbo en versión reducida, se va a dar un breve repaso a la interfaz que ofrece Optimal Trace.

9.1.1. Entorno de Optimal Trace.

De las cuatro zonas comentadas en el apartado anterior, la más importante es la del navegador, ya que es la que ofrece una visión general del proyecto. El navegador se subdivide en cuatro apartados: árbol, reseña gráfica, consulta y trazabilidad. Se muestran las cuatro vistas juntas en la siguiente figura:

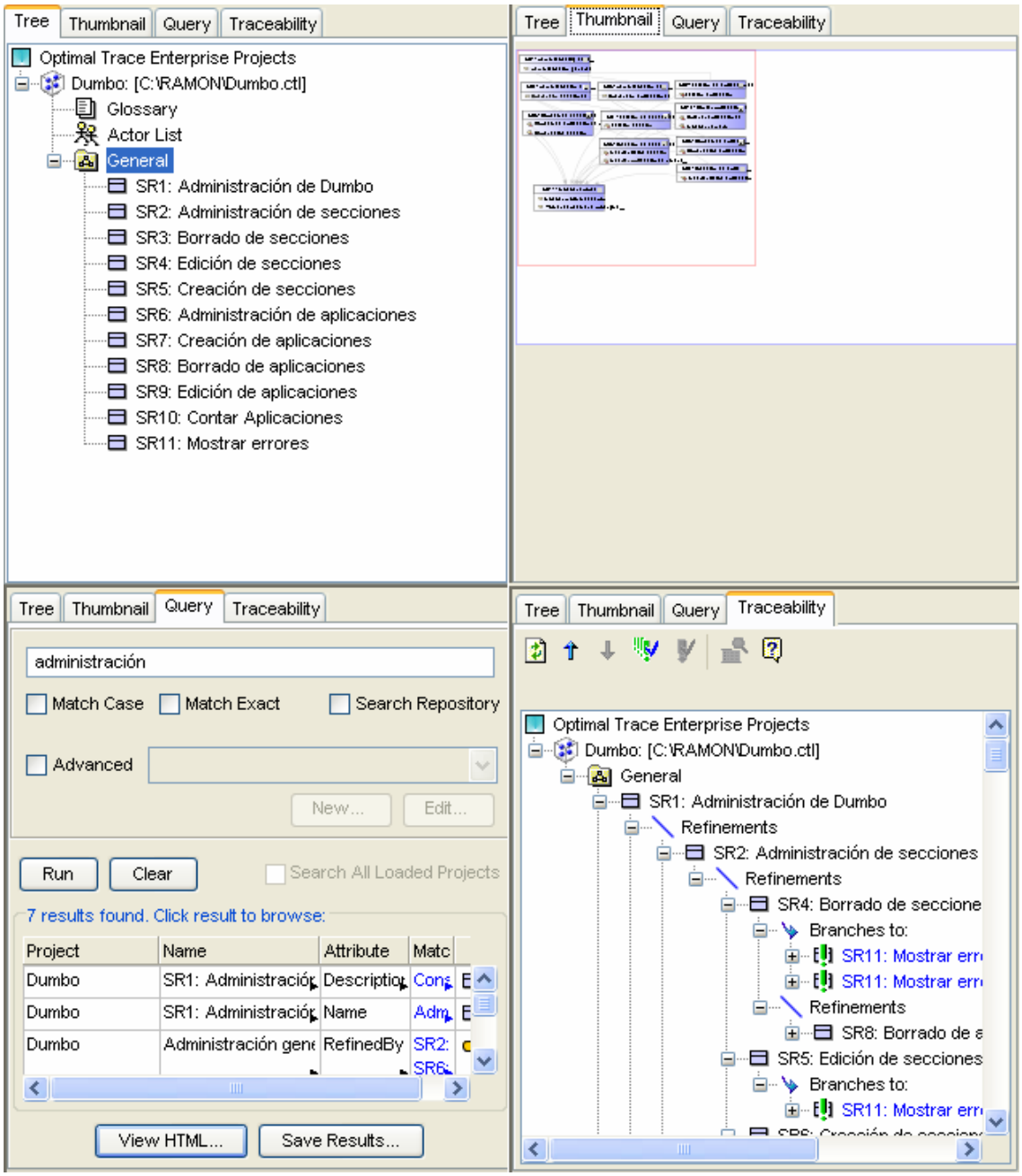


Figura 24. Aspecto de las distintas opciones del navegador de Optimal Trace.

En la vista árbol (*tree*), se presentan los proyectos activos y los componentes que integran cada proyecto. Cada proyecto dispone de un glosario de términos, una lista de los actores que intervienen y un paquete general que contiene a los requisitos. Además, el paquete general se puede subdividir en varios subpaquetes para una mejor clasificación de los requisitos.

En la reseña gráfica (*thumbnail*), se presenta el mapa de requisitos en miniatura para tener una mejor visión de los mismos.

La vista de consulta (*query*), permite realizar búsquedas dentro del mapa de requisitos. En la figura se presenta el resultado de realizar la búsqueda del término administración, y se muestran 7 coincidencias junto con información

adjunta (nombre, tipo de atributo, etc.). Ya que la ventana de resultados no es muy grande, se recomienda ver el resultado en versión HTML, mediante la opción *View HTML*. Esta opción presenta los resultados en una página HTML, formateada de una forma clara y limpia.

La última vista que se presenta puede llegar a ser la más importante. La vista de trazabilidad (*Traceability*) muestra todos los requisitos ordenados en forma de árbol según sus dependencias, y mostrando bifurcaciones y escenarios. En proyectos que intervengan muchos requisitos es una herramienta fundamental.

Una vez revisado el navegador, se pasa a explicar la herramienta de diseño de diagramas. Esta se divide en dos herramientas, una para el diseño del mapa de requisitos y otra para el diseño del diagrama de flujo de requisitos.

A continuación, se muestra una vista en detalle de la herramienta de diseño del mapa de requisitos, en la que también se muestran los requisitos planteados para la implementación de la versión reducida de Dumbo.

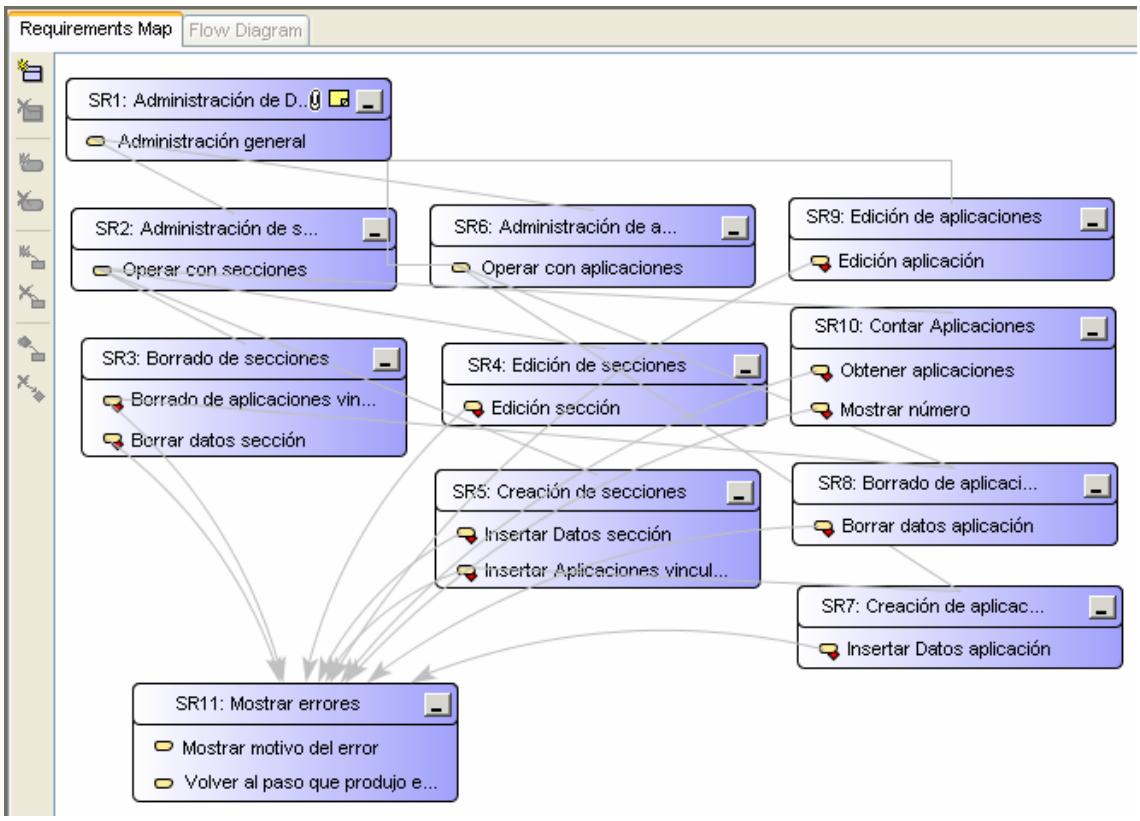


Figura 25. Herramienta de diseño de mapas de requisitos.

Para realizar la captura se han tenido que reagrupar todos los requisitos y por este motivo no se puede apreciar claramente la relación entre ellos. De todas formas, el fin de la figura 25 es mostrar el aspecto de la herramienta y distinguir los distintos requisitos, y sus pasos, que se han planteado.

Por otro lado, se tiene la herramienta de diseño de diagramas de flujo. Esta vista se activa al seleccionar un requisito en concreto. Una vez seleccionado el requisito, se puede hacer clic en la pestaña *Flow Diagram* y acceder al diagrama de flujo del requisito en cuestión. A modo de ejemplo, se presenta el diagrama de flujo del requisito SR4: edición de secciones.

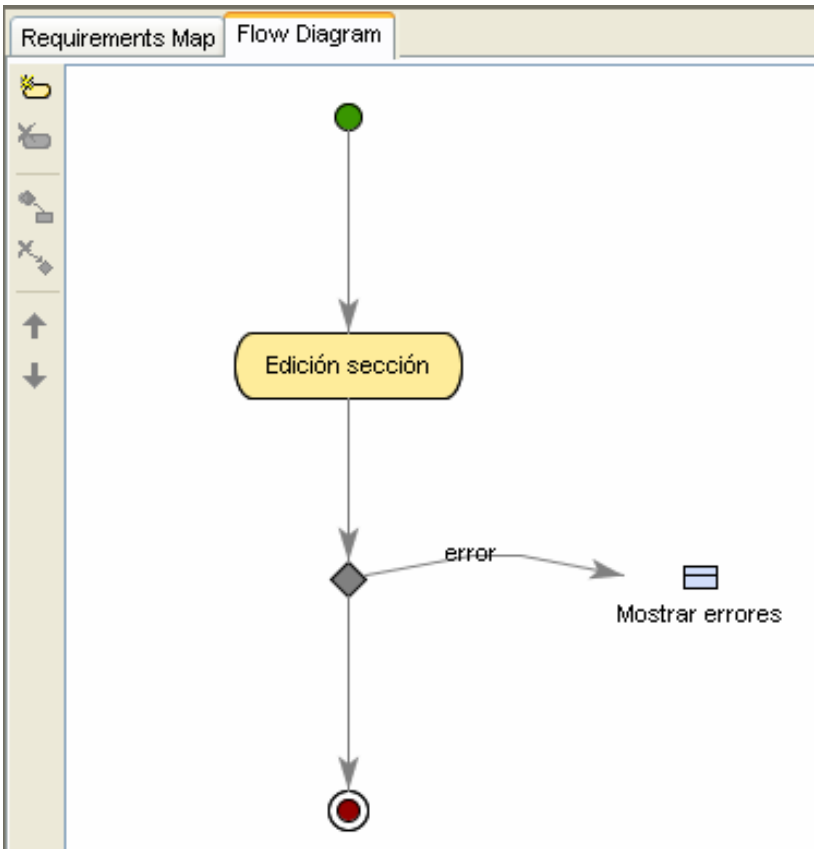


Figura 26. Diagrama de flujo del requisito SR4: Edición de secciones.

Es un diagrama de flujo bastante simple, donde sólo existe un paso que puede tener una bifurcación hacia el requisito SR11: Mostrar errores. Todos los requisitos del proyecto son muy similares.

El diagrama de flujo está íntimamente relacionado con la vista del detalle de los pasos que componen un requisito, situada en la parte inferior derecha de la ventana. El aspecto de esta vista es el que se muestra en la siguiente figura:

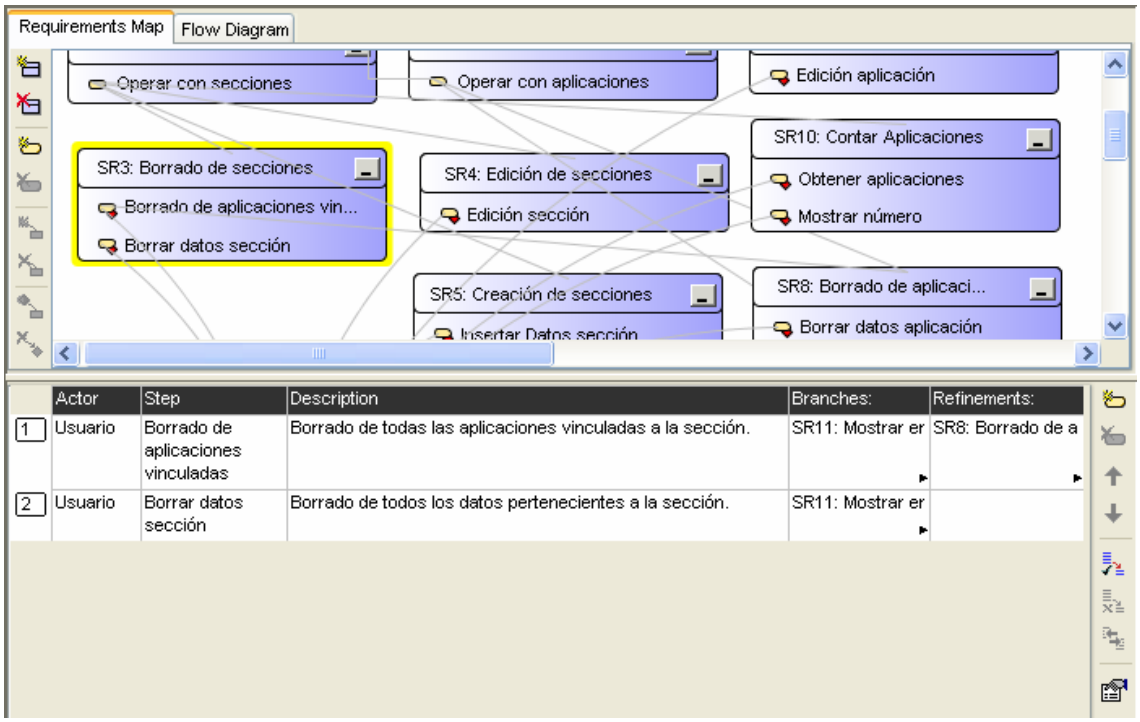


Figura 27. Detalle de los pasos del requisito SR3: Borrado de secciones.

El detalle muestra la misma información que el diagrama de flujo, pero en forma de tabla. Aquí es donde se puede encontrar la opción de crear un nuevo escenario y permite facilitar operaciones como cambiar el orden de los pasos del requisito. Esta vista sólo aparece cuando se tiene un requisito seleccionado, al igual que el diagrama de flujo.

Por último, queda por revisar la vista de propiedades de los requisitos. Aquí es donde se personaliza la información propia de cada uno de los requisitos. No aporta valor a la estructura, pero sí información sobre los elementos que la componen. Las distintas personalizaciones que se pueden realizar sobre un requisito son: modificar sus propiedades generales, crear propiedades personalizadas, crear requisitos no funcionales, crear enlaces relacionados y añadir notas. Las propiedades generales son fijas y dan información sobre el nombre del requisito, su utilidad, el actor principal, su prioridad, etc.

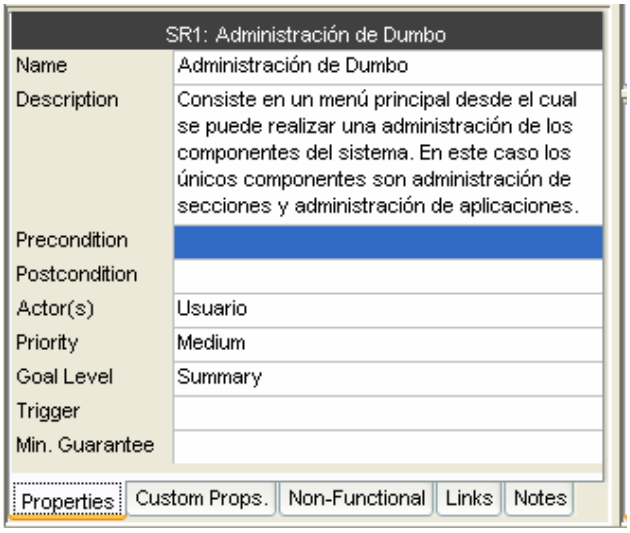


Figura 28. Detalle de las propiedades del requisito SR1.

En algunos casos no basta con esta descripción y se ha de extender un poco más. Para esto están las propiedades personalizadas.

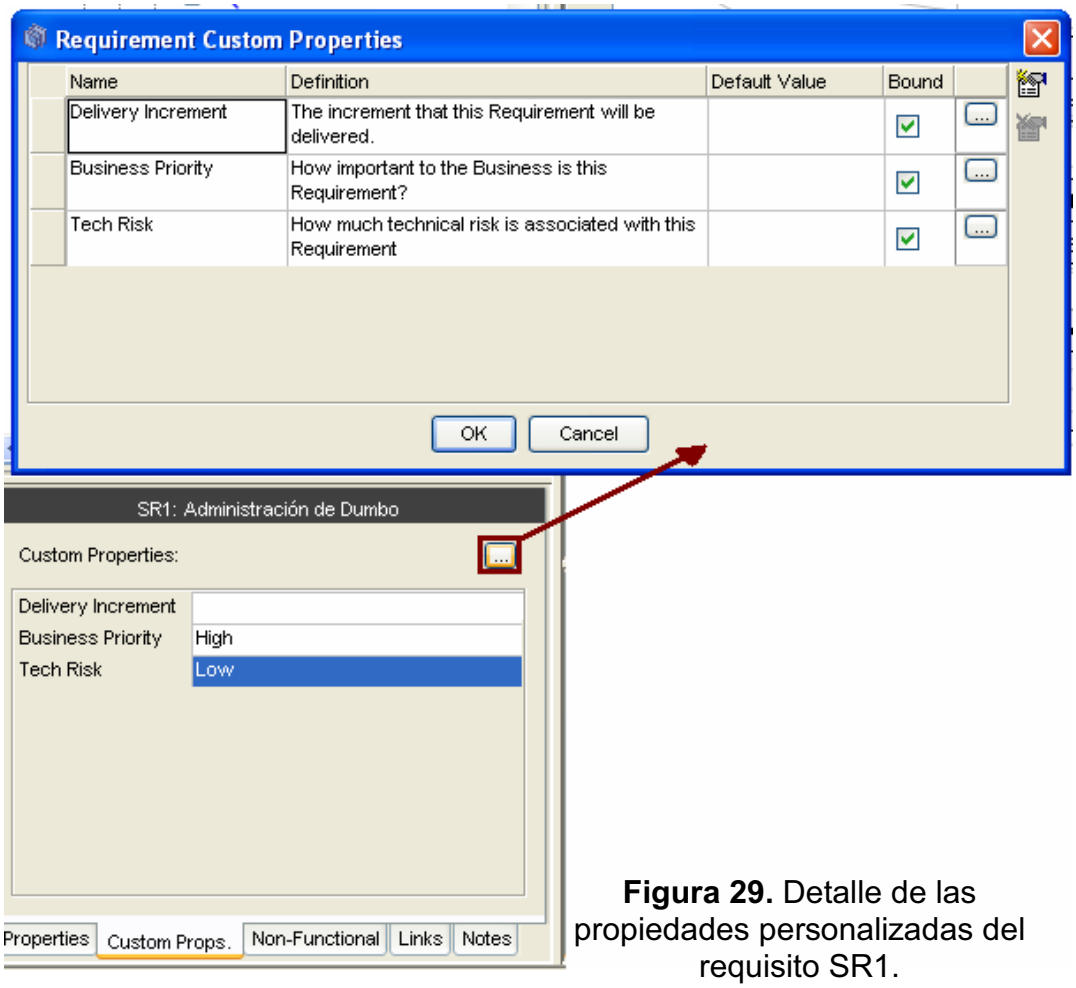


Figura 29. Detalle de las propiedades personalizadas del requisito SR1.

Otro aspecto a tener en cuenta son los requisitos no funcionales del sistema, que pueden ser igual de importantes que los requisitos funcionales. Es posible añadirlos en la opción *Non-Functional*. Un ejemplo de requisitos no funcionales podrían ser los idiomas soportados por la aplicación y la plataforma sobre la que se va a desarrollar la implementación.

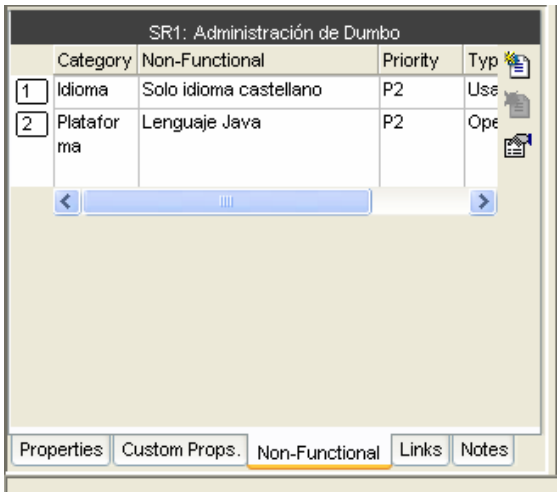


Figura 30. Detalle de las propiedades no funcionales del requisito SR1.

Para el final se han dejado los menos importantes, como son los enlaces y las notas. Son herramientas útiles para la documentación generada pero no tanto para la implementación del producto.

9.1.2. Diseño y explicación del mapa de requisitos.

Como ya se introdujo al comienzo del apartado 9, la versión reducida de la aplicación Dumbo se va a basar en tres componentes: Sistemas Dumbo, Secciones y Aplicaciones. También se desea poder realizar sobre cada uno de los componentes. Las operaciones que se han propuesto son:

- Creación de los componentes.
- Edición de los componentes.
- Borrado de los componentes.

Además, los componentes están relacionados entre sí de forma que un sistema contiene secciones y las secciones contienen aplicaciones. Por lo que desde un sistema se deben de poder crear secciones y desde una sección se han de poder crear aplicaciones. Por otro lado, realizar una operación de forma incorrecta debe de generar un aviso de error. Estas relaciones se pueden apreciar en los refinamientos de los pasos de los requisitos involucrados.

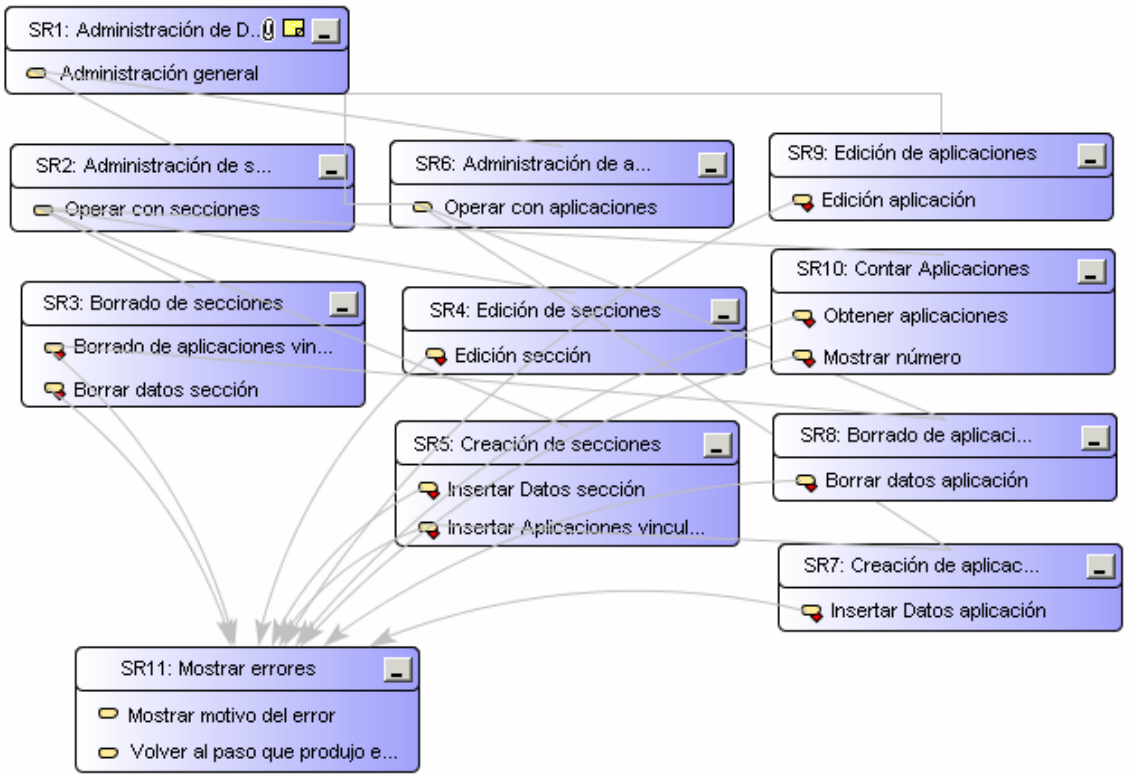


Figura 31. Mapa completo de requisitos de la aplicación.

Ya diseñado el mapa de requisitos y la implementación del diagrama de flujo de cada uno de ellos, se puede pasar a la generación de informes y la exportación del diseño a un formato que OptimalJ pueda entender. Este formato es XMI, y la exportación del mapa de requisitos a este formato dará lugar al diagrama de casos de uso de la aplicación.

9.1.3. Generación de informes en Optimal Trace.

Mediante la generación de informes se puede obtener una visión nítida de todos los aspectos del proyecto plasmados en el mapa de requisitos. Existen varios tipos de informes que se complementan entre sí. Se pueden acceder a ellos mediante la opción *report generation*.

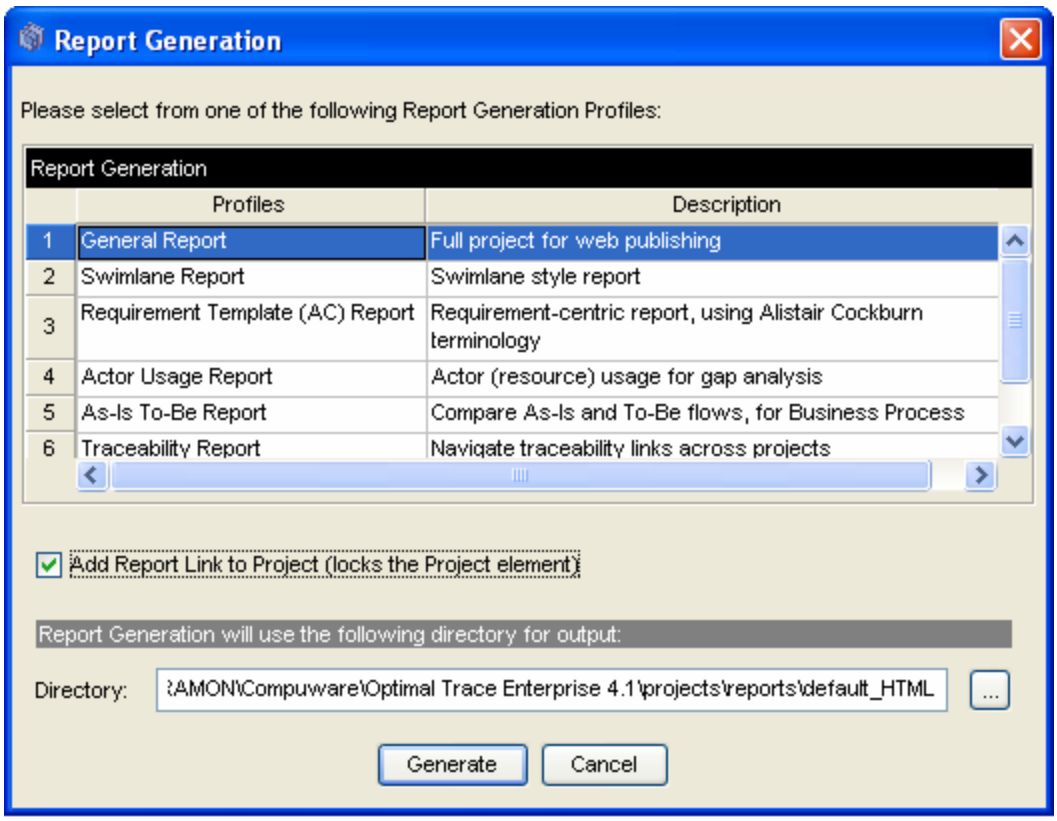


Figura 32. Ventana para la generación de informes en Optimal Trace.

El primer informe que aparece en el listado es *General Report*. Es el informe más general de todos y ofrece una visión global del proyecto, permitiendo navegar por los requisitos, ver el mapa de requisitos por completo e interactuar con él, etc. Se podría decir que es una visión desde la perspectiva del proyecto hacia los requisitos.

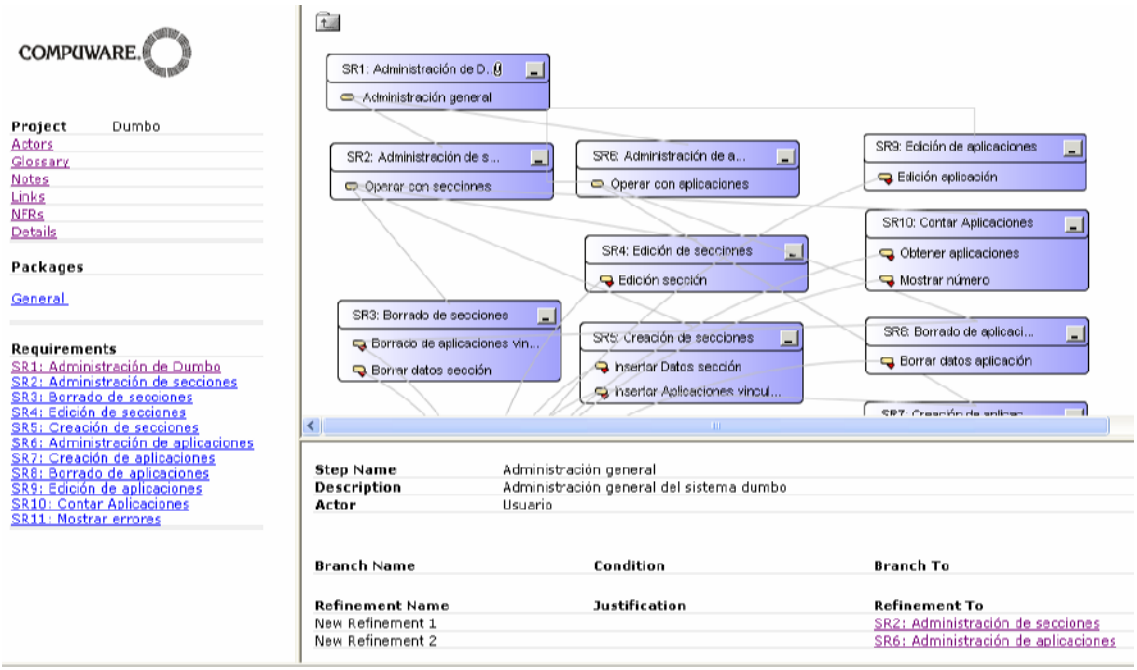


Figura 33. Informe *General Report*.

El siguiente tipo de informe que aparece en el listado es *Swimlane Report*. Es justamente la visión contraria a la de *General Report*. En este caso se podría decir que es una visión desde un requisito en concreto hacia la perspectiva general del proyecto.

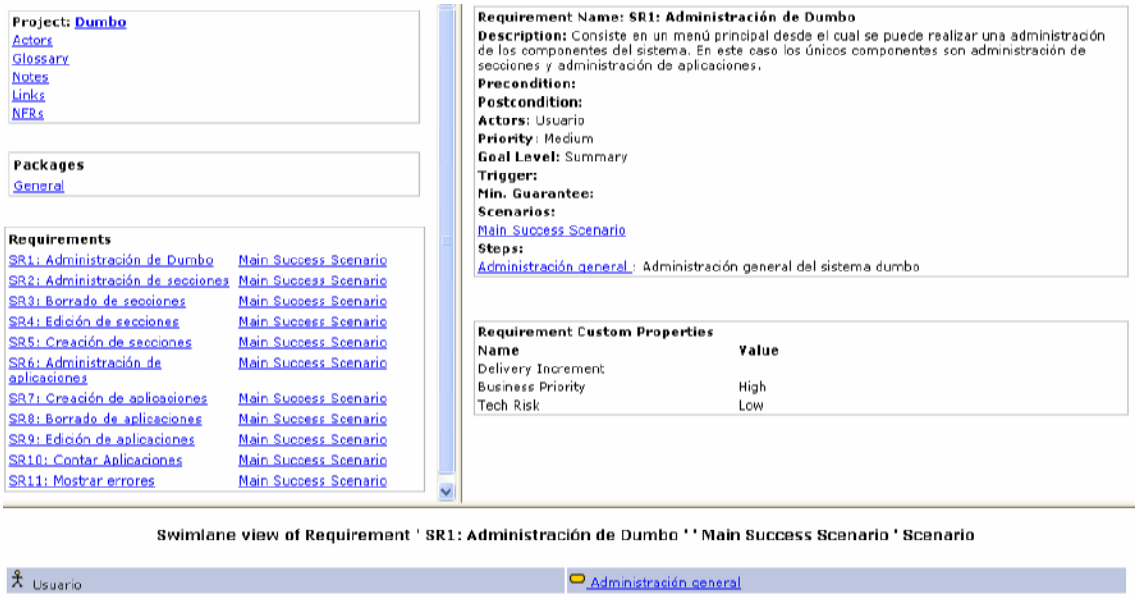


Figura 34. Informe *Swimlane Report*.

El informe de tipo *Requirement Template Report* es una mezcla entre *General Report* y *Swimlane Report*. Se puede ir navegando por una visión general de los requisitos y haciendo clic en cada uno de ellos para ver el detalle, pero desde una perspectiva parecida a la de *Swimlane Report*.

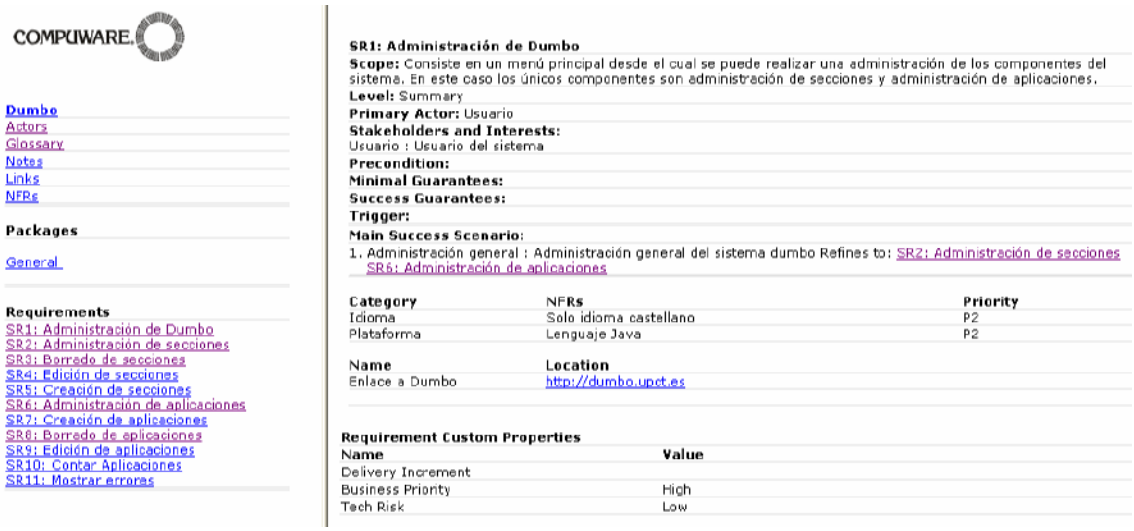


Figura 35. Informe *Requirement Template Report*.

El siguiente informe, *Actor Usage Report*, se centra en la figura de los actores para realizar el informe. Muestra los requisitos, los escenarios y los pasos agrupados por actor. También muestra los requisitos que se encuentran sin un actor asociado, algo que puede ser realmente útil antes de generar los casos de uso.

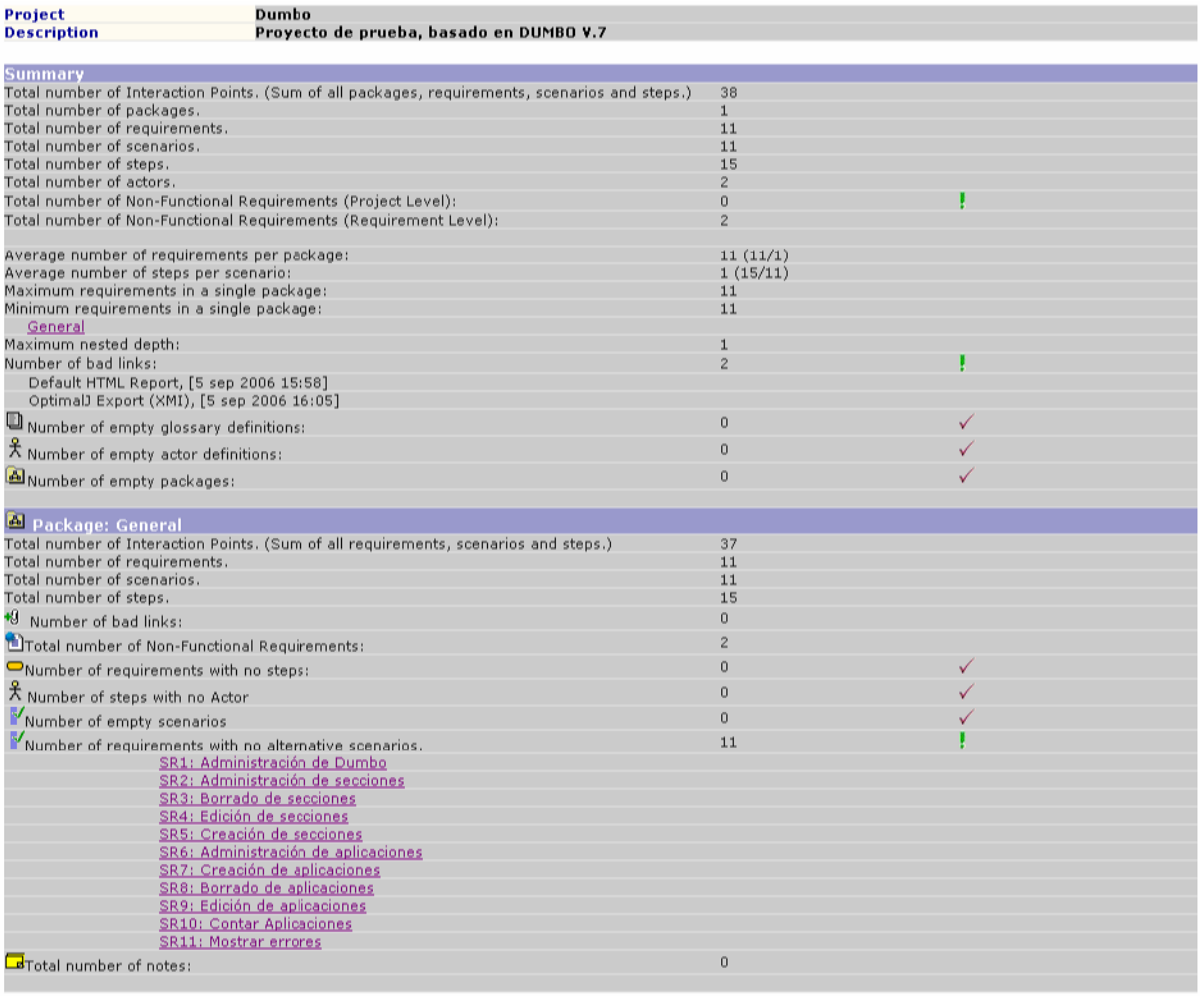
Figura 36. Informe *Actor Usage Report.*

El siguiente informe es el más importante en grandes proyectos, ya que es el informe de trazabilidad (*traceability report*). Muestra las relaciones y las dependencias entre todos los requisitos que componen el mapa.

Package	Requirement	Links	Branches	Refinements
General	SR1: Administración de Dumbo	Enlace a Dumbo		SR2: Administración de secciones SR6: Administración de aplicaciones
	SR2: Administración de secciones			SR3: Borrado de secciones SR4: Edición de secciones SR5: Creación de secciones SR10: Contar Aplicaciones
	SR3: Borrado de secciones		SR11: Mostrar errores SR11: Mostrar errores	SR8: Borrado de aplicaciones
	SR4: Edición de secciones		SR11: Mostrar errores	
	SR5: Creación de secciones		SR11: Mostrar errores SR11: Mostrar errores	SR7: Creación de aplicaciones
	SR6: Administración de aplicaciones			SR7: Creación de aplicaciones SR8: Borrado de aplicaciones SR9: Edición de aplicaciones
	SR7: Creación de aplicaciones		SR11: Mostrar errores	
	SR8: Borrado de aplicaciones		SR11: Mostrar errores	
	SR9: Edición de aplicaciones		SR11: Mostrar errores	
	SR10: Contar Aplicaciones		SR11: Mostrar errores SR11: Mostrar errores	
	SR11: Mostrar errores			

Figura 37. Informe *traceability report*.

El último tipo de informes es más bien informativo. Es el llamado informe de complejidad (*Complexity and Completeness Report*). Realiza un recuento general de todos los componentes que intervienen en el análisis. Aquí también se puede apreciar si algo se dejó a medias, ya que aparecen datos sobre los componentes creados y que aún no han sido definidos por completo. El análisis se realiza de forma general y centrado en los paquetes que componen el proyecto.



Project	Dumbo	
Description	Proyecto de prueba, basado en DUMBO V.7	
Summary		
Total number of Interaction Points. (Sum of all packages, requirements, scenarios and steps.)	38	
Total number of packages.	1	
Total number of requirements.	11	
Total number of scenarios.	11	
Total number of steps.	15	
Total number of actors.	2	
Total number of Non-Functional Requirements (Project Level):	0	!
Total number of Non-Functional Requirements (Requirement Level):	2	
Average number of requirements per package:	11 (11/1)	
Average number of steps per scenario:	1 (15/11)	
Maximum requirements in a single package:	11	
Minimum requirements in a single package:	11	
General		
Maximum nested depth:	1	
Number of bad links:	2	!
Default HTML Report, [5 sep 2006 15:58]		
OptimalJ Export (XMI), [5 sep 2006 16:05]		
Number of empty glossary definitions:	0	✓
Number of empty actor definitions:	0	✓
Number of empty packages:	0	✓
Package: General		
Total number of Interaction Points. (Sum of all requirements, scenarios and steps.)	37	
Total number of requirements.	11	
Total number of scenarios.	11	
Total number of steps.	15	
Number of bad links:	0	
Total number of Non-Functional Requirements:	2	
Number of requirements with no steps:	0	✓
Number of steps with no Actor	0	✓
Number of empty scenarios	0	✓
Number of requirements with no alternative scenarios.	11	!
SR1: Administración de Dumbo		
SR2: Administración de secciones		
SR3: Borrado de secciones		
SR4: Edición de secciones		
SR5: Creación de secciones		
SR6: Administración de aplicaciones		
SR7: Creación de aplicaciones		
SR8: Borrado de aplicaciones		
SR9: Edición de aplicaciones		
SR10: Contar Aplicaciones		
SR11: Mostrar errores		
Total number of notes:	0	

Figura 38. Informe *Complexity and completeness report*.

Todos los informes que han sido comentados son en versión HTML, por lo que se pueden publicar en un servidor *web* para que todos los miembros de un grupo de trabajo tengan acceso a ellos. Otra opción que también ofrece Optimal Trace es la de generar una versión para MS Word de los informes, aunque para ello no basta con configurar Optimal Trace, sino que también hay que configurar MS Word para que Optimal Trace pueda hacer su labor.

9.2. Análisis, Diseño e Implementación mediante OptimalJ.

Este es el momento de comprobar la potencia que ofrecen las herramientas MDA, en el caso que ocupa mediante OptimalJ, cuando llega el momento de crear una aplicación. También se va a comprobar la capacidad de integración de OptimalJ con Optimal Trace, aunque como se comentó en el apartado 9, se debe tener en cuenta que son unas de las primeras versiones de la línea de productos y también son versiones *trial*. Esto limita la capacidad de desarrollo e integración pero permite una primera aproximación a ellas.

El desarrollo del aparatado será el siguiente. Primero se importarán los requisitos del fichero exportado desde Optimal Trace, para poder generar los casos de uso del proyecto a partir del estudio de los requisitos. Una vez claros los casos de uso y los requisitos, se pasará al diseño del modelo de dominio de la aplicación (modelo PIM), para transformarlo en el modelo de aplicación (PSM) y luego generar el código de la aplicación.

En este caso no se profundizará tanto en la herramienta como en Optimal Trace y se comentarán sus características más interesantes a la vez que se desarrolla la aplicación.

Antes de comenzar a utilizar OptimalJ, se recuerda que la arquitectura que sigue es la misma que la propuesta por el OMG. Esta consta de tres capas donde la primera es el modelo de dominio (PIM), la segunda es el modelo de aplicación (PSM) y la última es el modelo de código (*Code Model*). El paso de un modelo a otro se denomina transformación, y el paso de un modelo a código se denomina generación.

9.2.1. Creación del proyecto e importación de requisitos.

El proyecto que se va a crear es un proyecto de aplicación web en tres capas, siguiendo la línea de desarrollo del Servicio de Informática. Es una de las arquitecturas soportadas en esta versión de OptimalJ, por lo que no hay problema en seguir este planteamiento. A continuación, se muestra la opción a seleccionar para crear el proyecto.

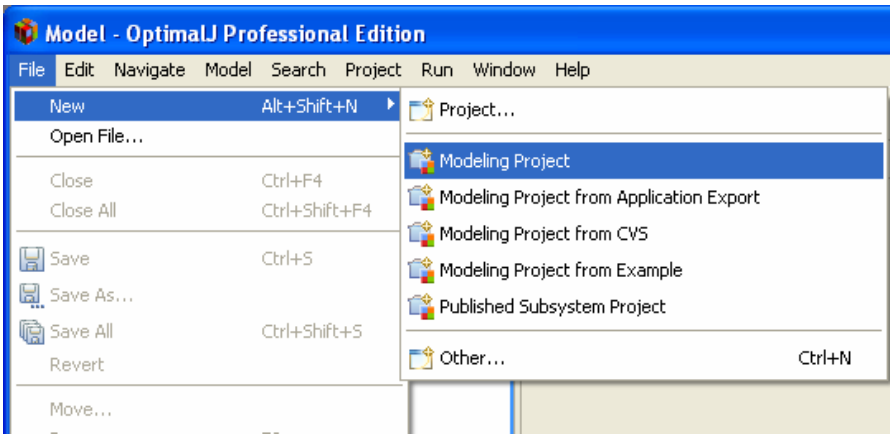


Figura 39. Creación de un proyecto en OptimakJ.

Una vez creado el proyecto, es el momento de importar los requisitos que se exportaron de Optimal Trace. La opción para importar los requisitos desde un fichero en formato XMI es la siguiente:

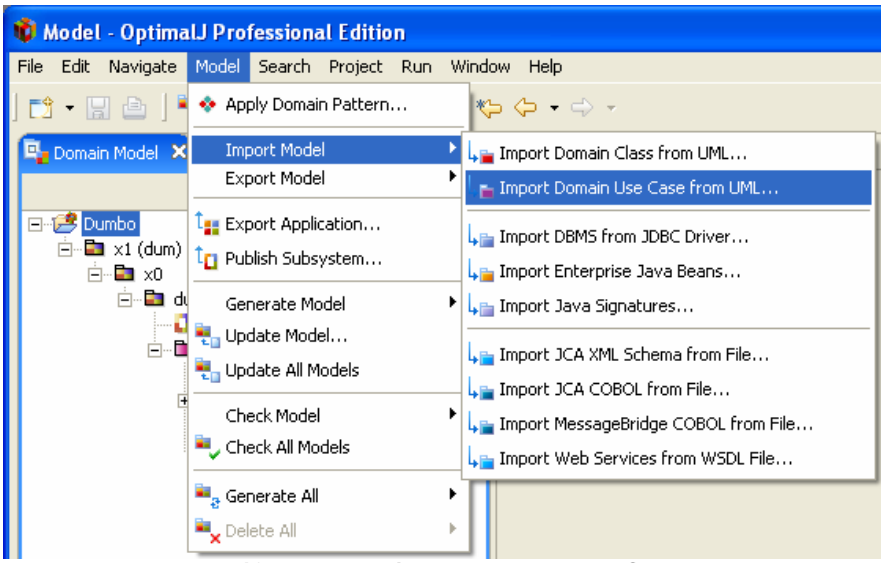


Figura 40. Importación de requisitos en OptimalJ.

Una vez importados los casos de uso desde el fichero, se genera el modelo de casos de uso dentro de modelo de dominio de OptimaJ. Si el archivo XMI se generó a partir de la estructura de requisitos propuesta en el apartado 9.1 (sin paquetes que agrupen los requisitos), el diagrama de casos de uso generado es el siguiente:

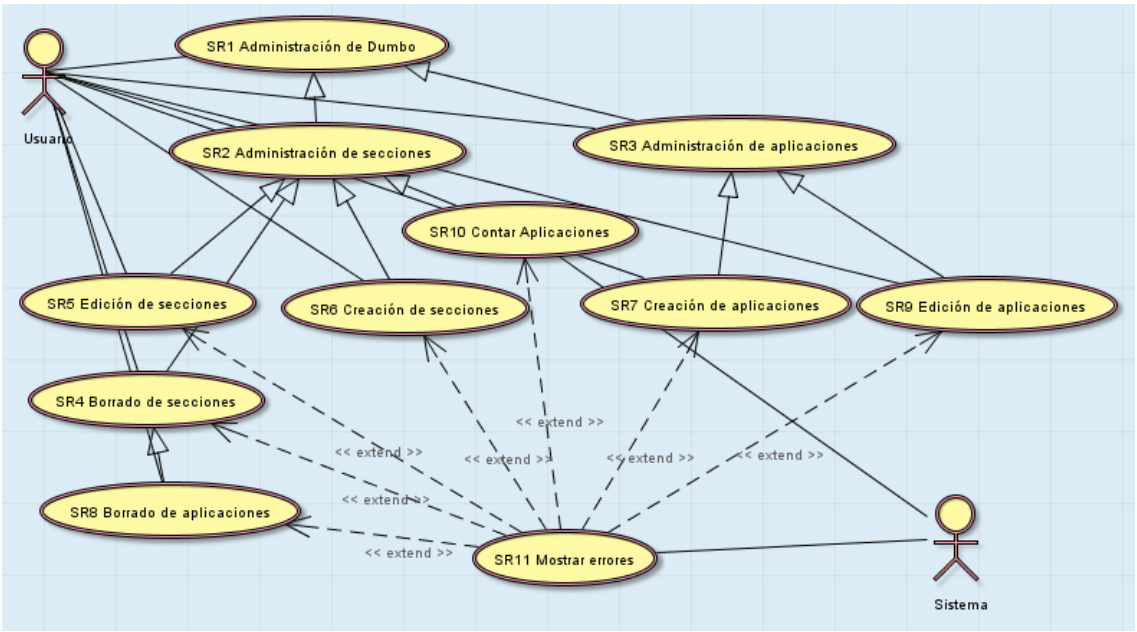


Figura 41. Casos de uso desagrupados en OptimalJ

Al no estar agrupados los casos de uso, el diagrama es bastante complejo y farragoso. Por este motivo se propone la siguiente estructura de requisitos en Optimal Trace para que el diagrama de casos de uso se más fácil de comprender.

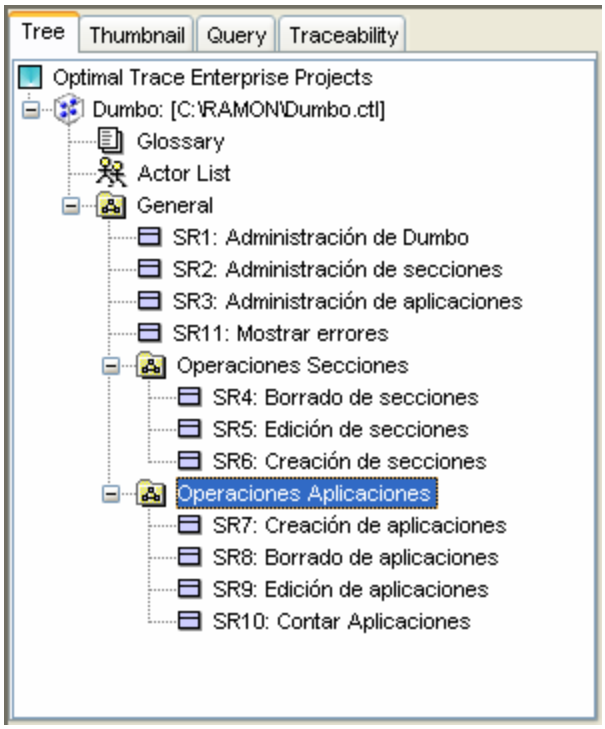


Figura 42. Requisitos agrupados en Optimal Trace.

El paquete *General*, se ha subdividido en dos paquetes que contienen las operaciones con secciones y las operaciones con aplicaciones.

De esta forma, la organización se plasmará también en el diagrama de casos de uso resultante en OptimalJ.

En la siguiente figura se presenta el diagrama de casos de uso generado por OptimalJ a partir del diseño en Optimal Trace.

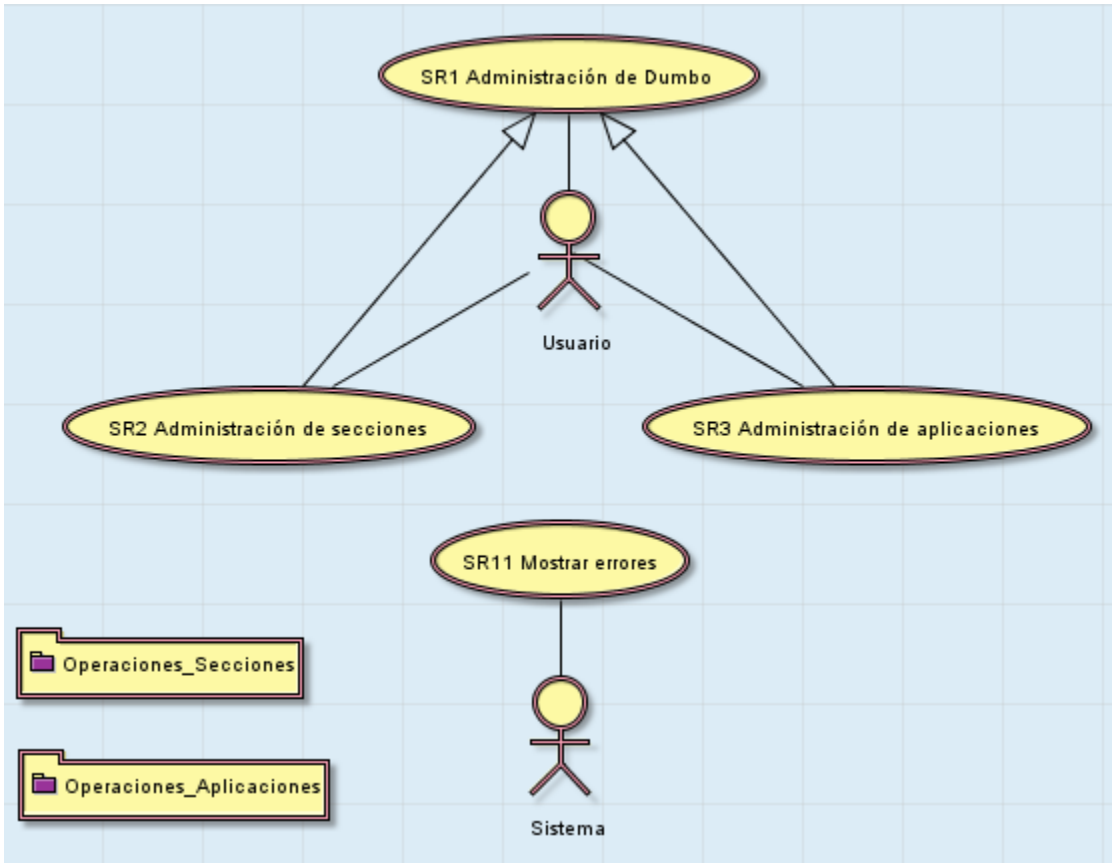


Figura 42. Casos de uso agrupados en OptimalJ. Visión general.

El diseño es mucho más claro, aunque hay que resaltar que se ha perdido información en el diagrama. Esta información es la referente a la extensión de las operaciones de secciones y aplicaciones del requisito Mostrar errores. Si se accede al contenido de ambos paquetes se puede observar que ahí tampoco se hace referencia a dicha relación.

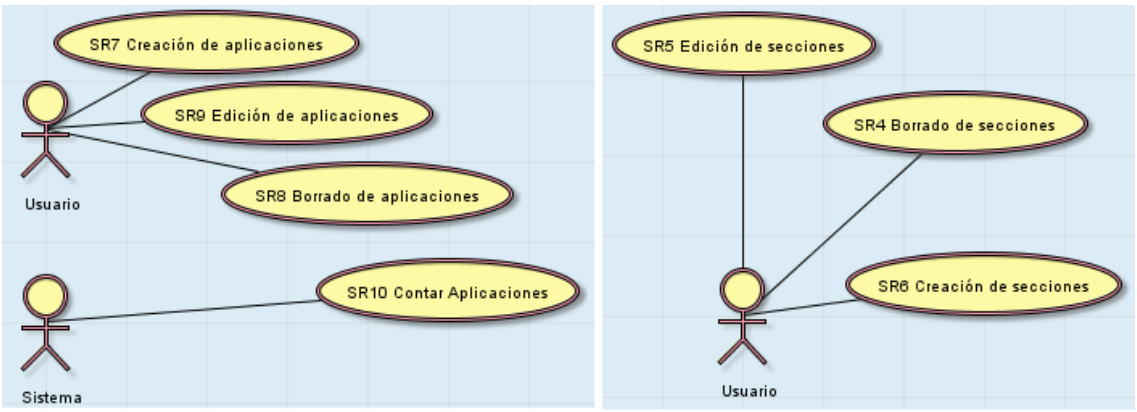


Figura 43. Detalle de los casos de uso de operaciones_secciones y operaciones_aplicaciones.

Este es el único problema detectado al importar casos de uso desde Optimal Trace.

El siguiente paso a realizar, es comenzar con el diseño del modelo de dominio de la aplicación.

9.2.2. Creación del modelo de dominio.

Recordando lo aprendido en el apartado 6.7, el modelo de dominio es el modelo de más alto nivel, y equivale al modelo independiente de la plataforma (PIM) en la arquitectura MDA. Este modelo captura el comportamiento estático de la aplicación y se subdivide en modelo de clases y modelo de servicios. El modelo de clases define la estructura de la información y el de servicios define acciones sobre dicha estructura. En la versión *trial* que se está utilizando, se han limitado las posibilidades de personalización, por lo que ciertas acciones no estarán disponibles.

En esta primera aproximación, sólo se va a crear la estructura de la aplicación, por lo que sólo se va a realizar un modelo de clases. El modelo de clases definido es el siguiente:

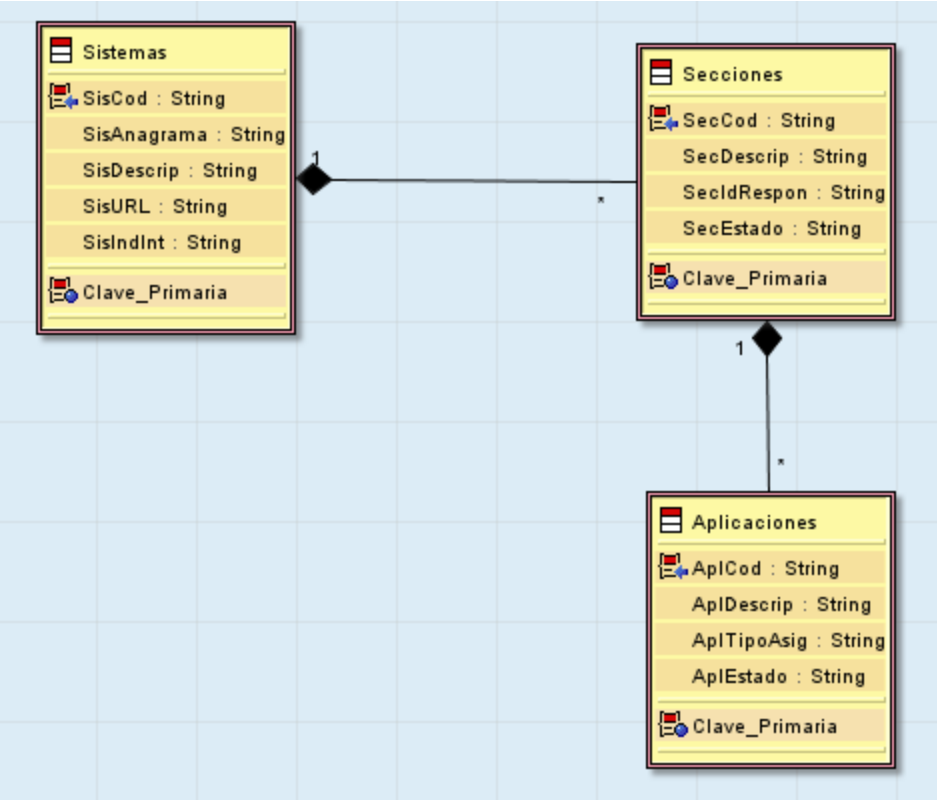


Figura 44. Modelo de clases dentro del modelo de dominio de la aplicación.

El modelo propuesto se basa en las clases Sistemas, Secciones y Aplicaciones. Un Sistema puede componerse de muchas aplicaciones, mientras que una sección sólo puede depender de un sistema (relación de composición 1 – 0...*). Por otro lado, una sección puede componerse de muchas aplicaciones, mientras que una aplicación sólo puede depender de una sección (relación de composición 1 – 0...*).

Cada una de las clases se compone unos atributos y una restricción de tipo *primary*. Esta restricción dará lugar más adelante a la restricción de clave primaria. La relación de composición sumada a una restricción *primary* puede dar lugar a una restricción de tipo *foreign key*, aunque esto se puede parametrizar. Por este motivo, no se ha añadido un campo dentro de secciones que haga referencia al código del sistema, que compondría la verdadera clave primaria de secciones.

Una vez creado el diseño, se pueden refinar los componentes y los atributos de los componentes. Esto se hace mediante la opción *properties*, disponible en la parte inferior de la ventana de OptimalJ. Si se selecciona una clase, aparece en la ventana *properties* las propiedades de la clase, y si selecciona un atributo, aparece en la ventana *properties* las propiedades del atributo.

Si se selecciona la clase secciones, aparecen las siguientes propiedades en la ventana *properties*.

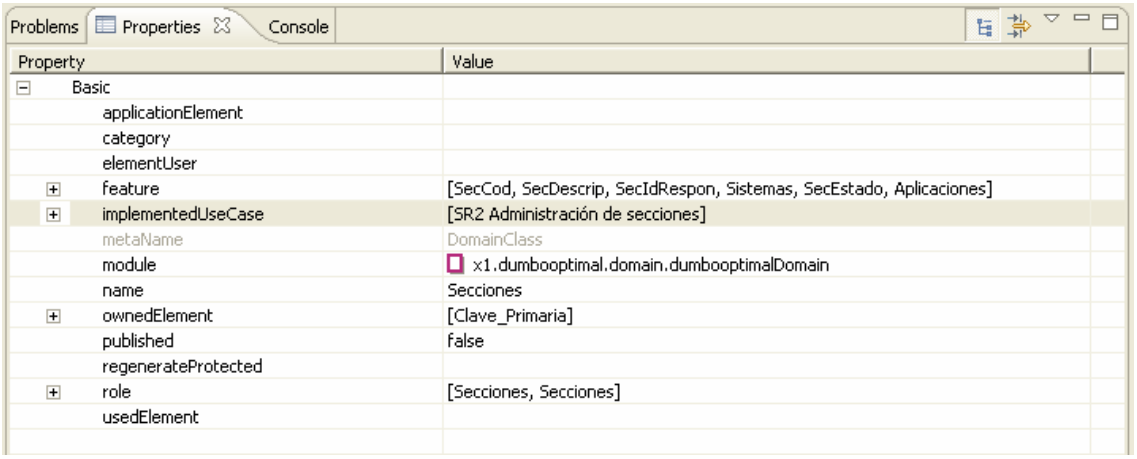


Figura 45. Detalle de las propiedades de la clase Secciones.

Destacan propiedades como *implementedUseCase*, que permite identificar una clase con un caso de uso; *published*, que es un atributo que permite publicar clases para su reutilización en otros proyectos y para que los objetos que se creen de este tipo puedan ser accedidos de forma remota; o *feature*, que muestra los elementos de los que se compone (incluyendo clases relacionadas por composición).

Si en vez de seleccionar una clase se selecciona un atributo, cambian algunas de las propiedades respecto a la selección de una clase completa. A continuación se muestra la ventana *properties* tras seleccionar el atributo SisCod de la clase Sistema.

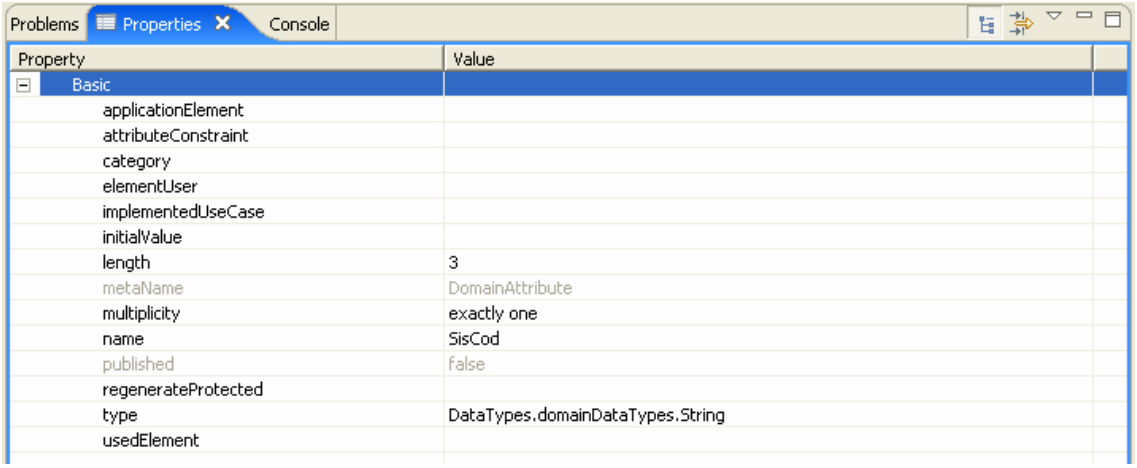


Figura 46. Propiedades del atributo SisCod de la clase Sistemas.

La propiedad más importante es *attributeConstraint*. Permite añadir restricciones a un atributo en particular. Las restricciones pueden ser un formato específico (por ejemplo, formato tipo URL) o reducir los valores permitidos a un grupo. Estas restricciones pueden dar lugar finalmente a restricciones en base de datos o restricciones en la interfaz de usuario. En la versión *trial* no se han añadido las restricciones, por lo que no se puede mostrar un ejemplo de ellas.

Una vez terminado el diseño de la estructura de la aplicación, se puede seleccionar la opción *Update All Models*, para generar el modelo de servicios básicos y el modelo de aplicación, junto con sus submodelos.

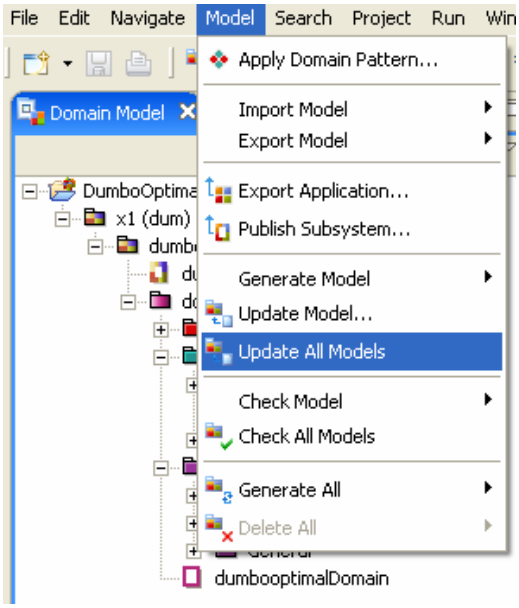


Figura 47. Actualización de todos los modelos.

9.2.3. Revisión del modelo de aplicación.

Tras seleccionar la opción de actualizar todos los modelos, se pasa a la opción *Application Model*, para comprobar que se ha generado el modelo de aplicación. El modelo de aplicación es el modelo dependiente de la plataforma (PSM), aunque en este caso la plataforma sólo es J2EE. También varían las transformaciones según la estructura final que se desee, en el caso que ocupa la arquitectura es una aplicación *web* en tres capas.

Como también se introdujo en el apartado 6.7, las transformaciones se realizan mediante patrones de transformación. OptimalJ provee un gran número de ellos, y permite la creación de patrones de transformación propios. Existen patrones de modelo de clases a modelo de negocio, patrones de modelo de servicio a modelo de negocio, patrones de modelo clases a modelo de servicio, etc. En el caso de la versión *trial*, no se permite la posibilidad de crear nuevos patrones.

Recordando conceptos introducidos anteriormente, el modelo de aplicación se compone de varios submodelos. Estos son el modelo de presentación, el modelo de negocio, el modelo de base de datos, el modelo de fachada de negocio y los modelos comunes.

El modelo de presentación contiene un diagrama de clases especializado en elementos *web*, del que parte la interfaz *web* de la aplicación. En la versión comercial de OptimalJ, se incluye una herramienta que permite visualizar el aspecto de cada uno de los componentes generados, modificar dicho aspecto o crear interfaces *web* nuevas. También incluye plantillas para los formatos de

listados y formularios más comunes (por ejemplo, plantillas para vistas maestro/detalle). Al estar trabajando con la versión *trial* no se puede ofrecer una visión sobre dicha herramienta.

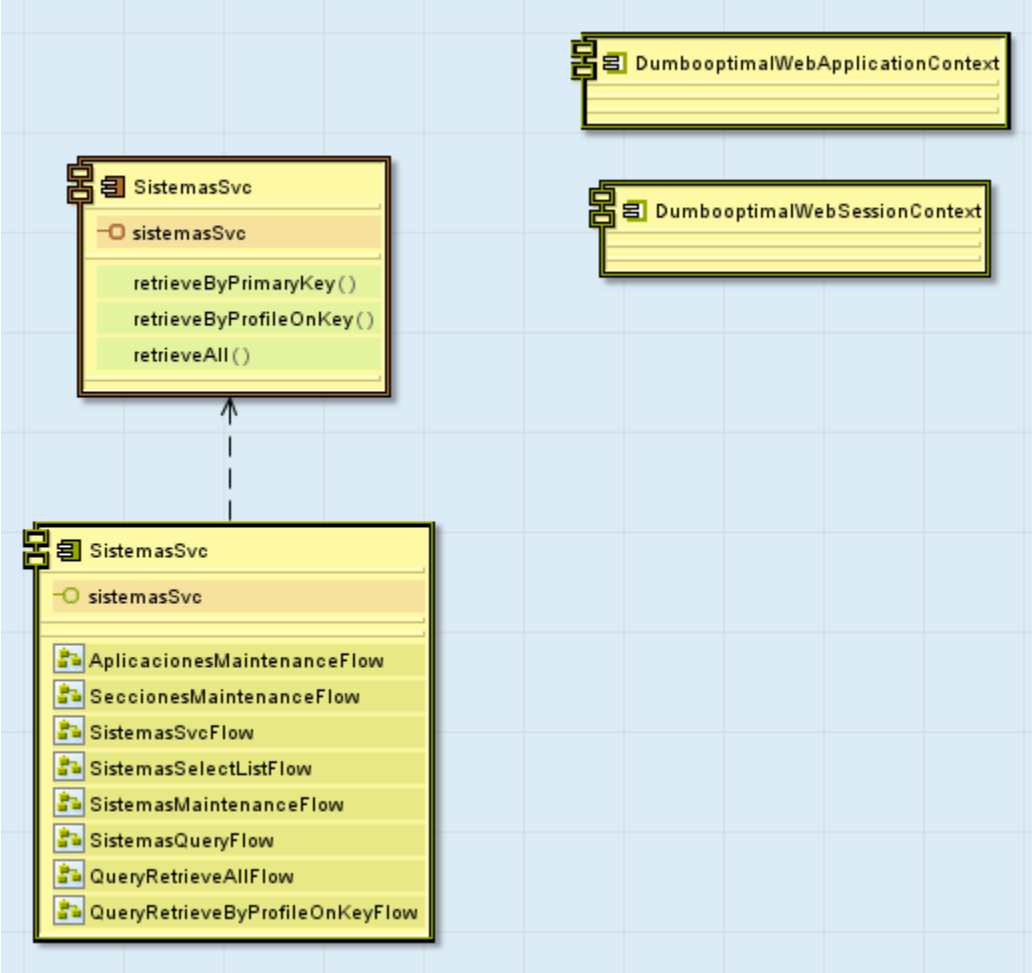


Figura 48. Modelo de presentación dentro del modelo de aplicación.

El modelo de lógica de negocio contiene la información relacionada con la manipulación y comportamiento de los datos. Se pueden añadir nuevas consultas para que posteriormente sean utilizadas en otros modelos, o desde llamadas implementadas en los bloques libres del código.

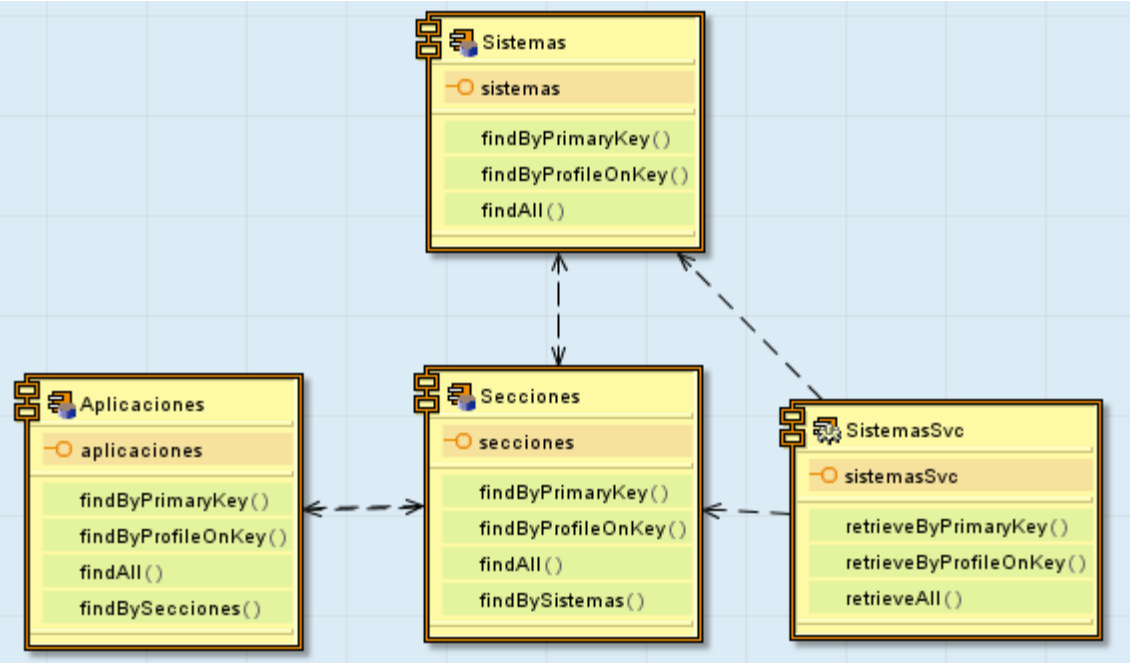
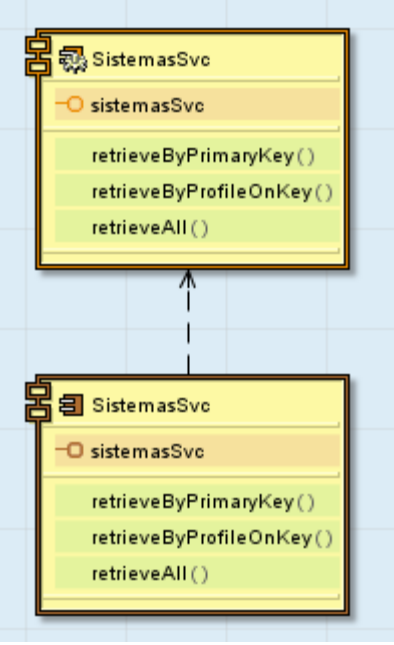


Figura 49. Modelo de lógica de negocio dentro del modelo de aplicación.

En la figura 49, se pueden observar algunas de las operaciones generadas por defecto, como es obtener el sistema a través de su clave primaria (*findByPrimaryKey*) o obtener todos los sistemas (*findAll*).

El siguiente modelo a comentar es el de fachada de negocio, que comunica el modelo de lógica de negocio con el modelo de presentación. Es un modelo puente. Sólo hay que modificarlo en caso de que se realice una modificación en alguno de los dos modelos mencionados. Si se quisiera realizar una consulta no contemplada anteriormente y mostrar el resultado en la interfaz *web*, se debería realizar una modificación en este diagrama. El diagrama resultante tras la actualización de modelos se muestra a continuación:



Si se observa detenidamente el diagrama, aparecen operaciones que también se muestran en el modelo de presentación y el modelo de lógica de negocio, como por ejemplo *retriveAll*. Esto es porque en la interfaz web aparecerá una opción para obtener todos los sistemas, la lógica de negocio realizará la consulta para obtenerlos y se intercambiarán a través de la fachada de negocio.

Figura 50. Modelo de fachada de negocio dentro del modelo de aplicación.

El último modelo a comentar es el de base de datos, ya que los modelos comunes no presentan ningún tipo de diagrama y sólo ofrecen información relativa sobre las estructuras compartidas por los modelos.

El modelo de base de datos está compuesto por un esquema relacional, parecido a un diagrama de entidad-relación, que servirá para generar los *scripts* SQL para la creación de la base de datos sobre la que se implementará la persistencia de datos.

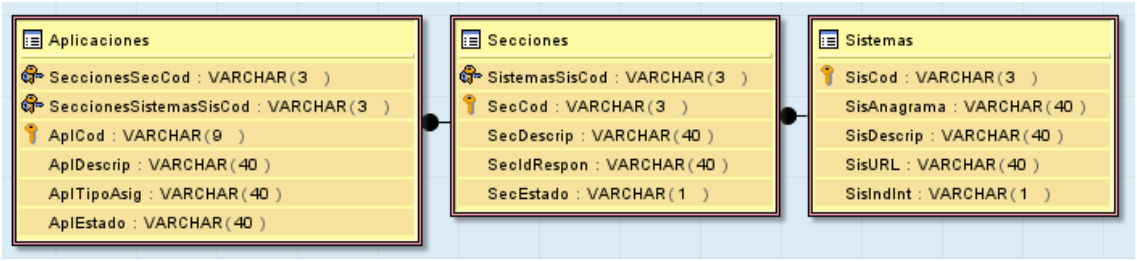


Figura 51. Modelo de base de datos dentro del modelo de aplicación.

Se distinguen dos tipos de atributos. Los atributos que tienen a su izquierda una llave dorada son parte de la clave primaria, y los que tiene dos llaves cruzadas son parte de la clave primaria y, además, son clave ajena. Como ya se comentó anteriormente, algunos de estos atributos se generan por las relaciones de composición y restricción *primary*, como es el caso de *SeccionesSecCod* o *sistemasSisCod*.

A partir de este modelo de aplicación, ya se puede generar el modelo de código y comenzar a realizar pruebas sobre este. Esto será lo que se explique en el siguiente apartado.

9.2.4. Generación del modelo de código y pruebas.

Una vez analizado el modelo de aplicación, sólo queda generar el código de la implementación de la aplicación. Para ello se debe seleccionar la opción de generar código y luego la de generar los archivos de despliegue. Estas opciones se muestran en la siguiente figura.

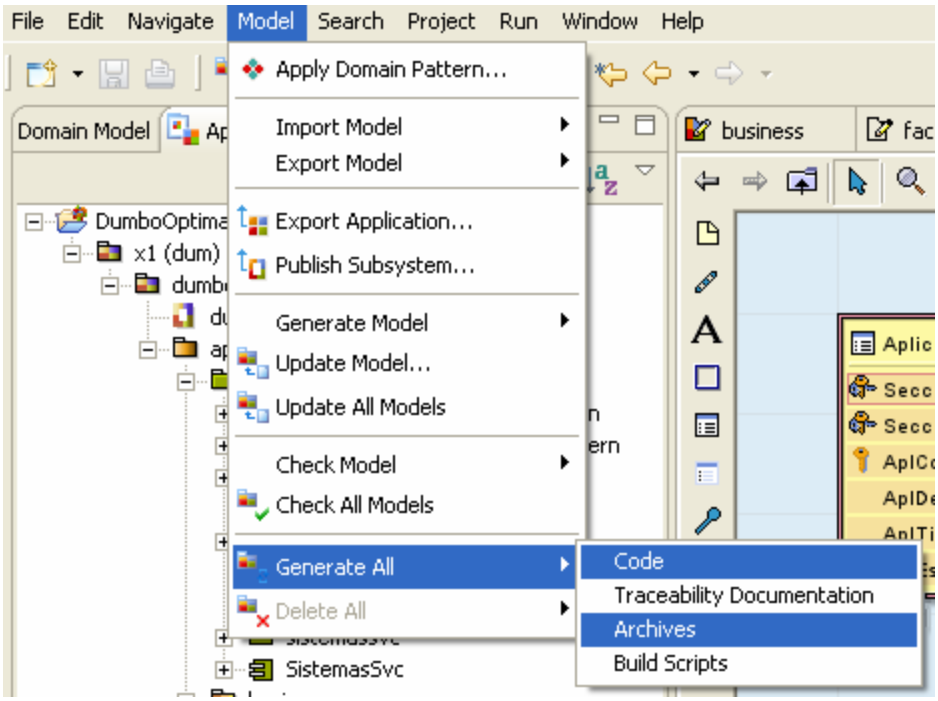


Figura 52. Generación de código y archivos en OptimalJ.

Al terminar el proceso de generación, se debe de cambiar la vista de OptimalJ a *Code Model*, donde aparecen listados en el navegador todos los ficheros que se han generado. Los ficheros están ordenados por carpeta según su función y modelo de procedencia.

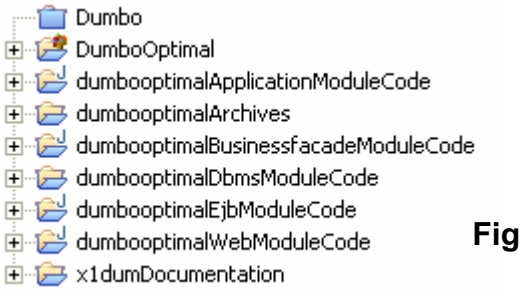


Figura 53. Vista de la estructura de archivos generada en el modelo de código.

Las carpetas más interesantes son la que contienen los ficheros de la interfaz web (*WebModule*), la que contiene la estructura de persistencia de objetos (*EJBModule*) y la que contiene los scripts para la creación de la base de datos (*DbmsModule*). Estas carpetas contienen la estructura básica de una aplicación *web* en tres capas.

También existe una carpeta de documentación. En todos los modelos se permite una opción desde el navegador de clases que es *generate documentation*, el resultado de dichas generaciones se almacena en la carpeta *Documentation* siguiendo una estructura basada en HTML.

Antes de probar la aplicación, se debe crear la estructura en una base de datos. OptimalJ integra dos herramientas muy útiles para la creación de una base de datos de pruebas. La primera de ellas es una base de datos, muy

simple, llamada SOLID. Esta se instala junto con OptimalJ y permite arrancarla haciendo clic sobre el icono  que se instala en el escritorio. Una vez arrancada la base de datos se puede llamar a la herramienta DBMS WorkBench desde la carpeta *DbmsModule*. Se debe hacer doble clic sobre el fichero indicado en la siguiente figura:

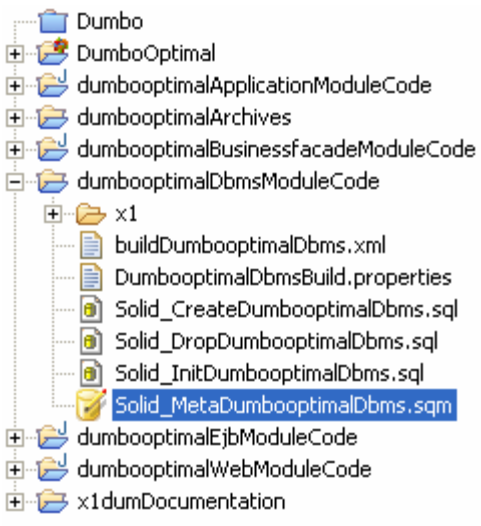


Figura 54. Detalle de la Vista de la estructura de archivos centrada en el módulo *Dbms*.

Una vez ejecutado el banco de pruebas, aparece la pantalla del cliente de acceso a la base de datos SOLID, donde se deben introducir los datos de acceso. Normalmente estos datos vienen previamente configurados y simplemente hay que continuar. El aspecto de la pantalla de acceso es el siguiente:

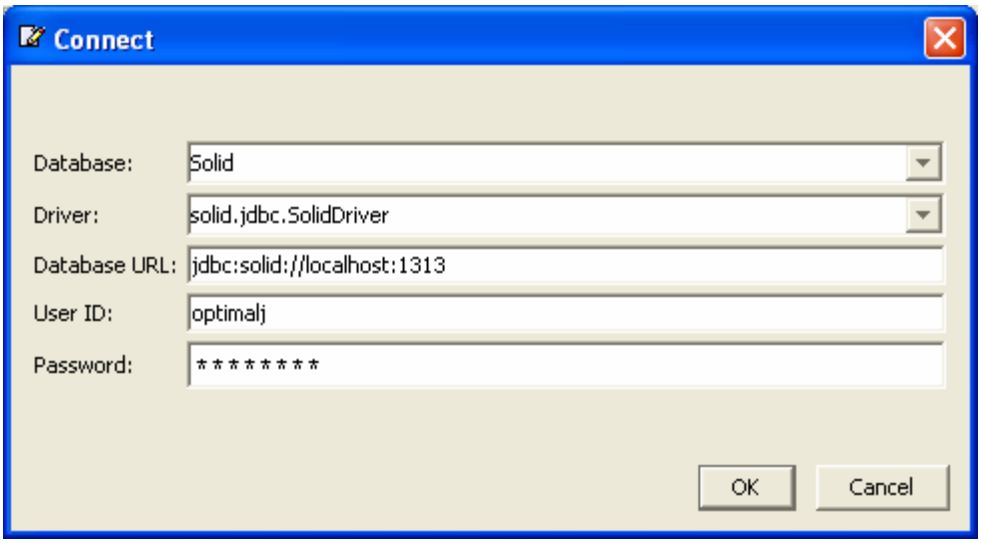


Figura 55. Detalle de la pantalla de acceso del cliente de base de datos.

Una vez se haya accedido a la base de datos, se muestra una nueva pantalla sobre la que se pueden ejecutar *scripts* de creación y borrado de tablas. Los *scripts* se generan con el código de la aplicación a partir del modelo de base de datos del modelo de aplicación, por lo que para crear la estructura de base de datos sólo se hay que pulsar el botón *generate* y luego el botón *exec batch*.

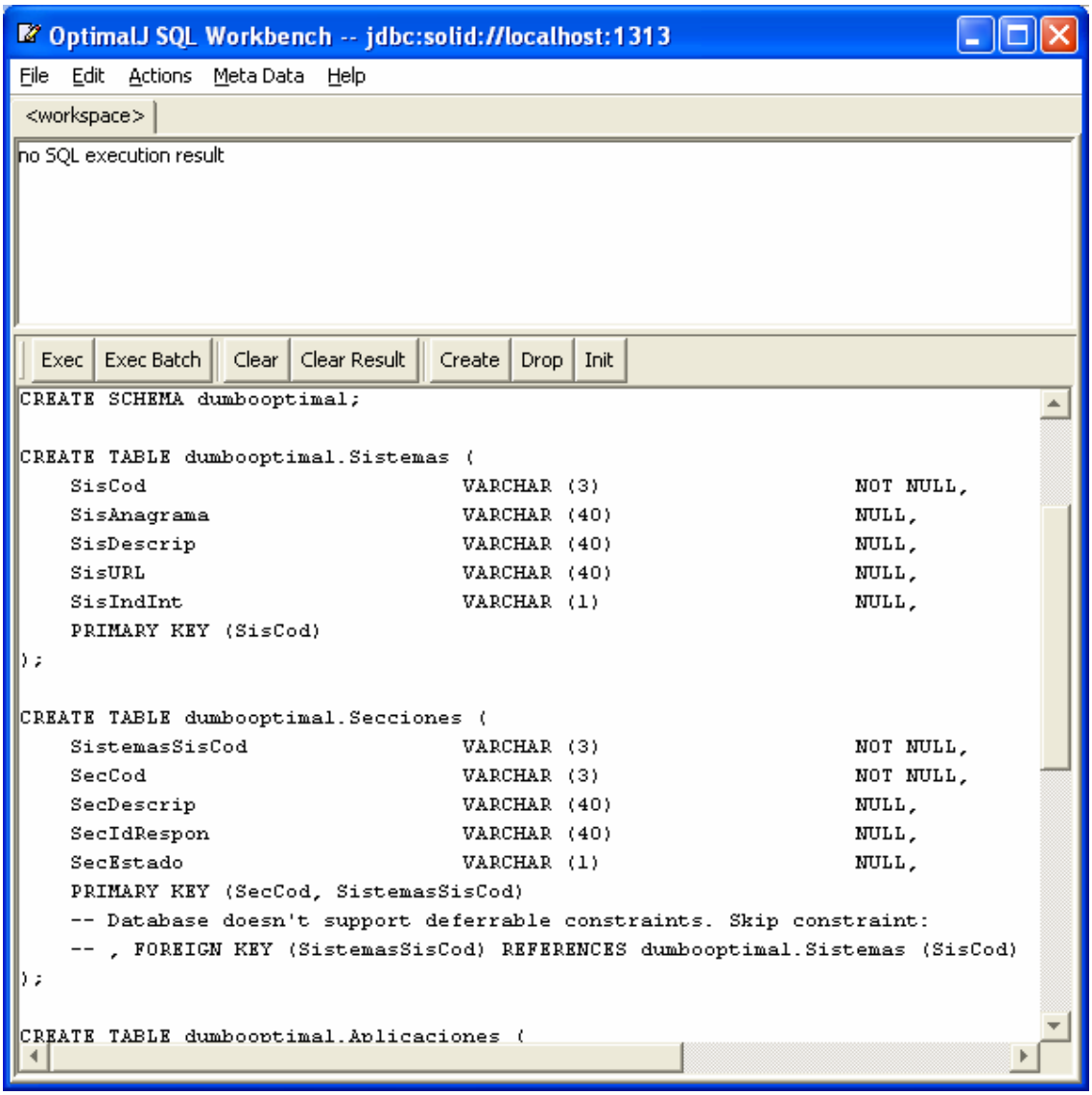


Figura 56. Creación de tablas en base de datos de pruebas.

En las tablas generadas, existe una fila comentada. Esta fila es la que crearía la referencia de clave ajena, pero este tipo de referencias no están permitidas en SOLID.

Una vez creadas las tablas en la base de datos, se puede desplegar la aplicación sobre el servidor de aplicaciones que lleva integrado OptimalJ. Para esto sólo hay que pulsar sobre la opción *start application server*, que se encuentra en el lugar indicado por la siguiente figura.

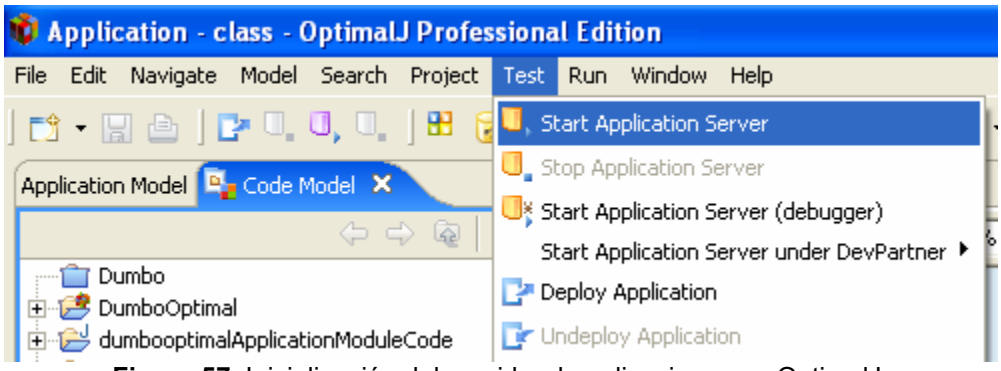


Figura 57. Inicialización del servidor de aplicaciones en OptimalJ.

Tras arrancar el servidor de aplicaciones y desplegar la aplicación en él, se abrirá un explorador *web* que integra OptimalJ para que se puedan realizar pruebas sobre la aplicación creada.

9.2.5. Aspecto de la aplicación generada.

Ya desplegada la aplicación en el servidor, es el momento de realizar algunas pruebas de usuario sobre ella. El aspecto de la aplicación se puede variar mediante el editor de interfaces web disponible sólo en la versión comercial de la aplicación (de la que no se dispone), por lo que el resultado no será muy vistoso. De todas formas, la idea que tiene que quedar es que el desarrollo la capa de negocio de la aplicación y su estructura de base de datos se desarrolla rápidamente mediante OptimalJ, y finalmente sólo queda depurar la presentación de la aplicación.

Una vez arrancada la aplicación, se puede acceder a ella mediante un navegador cualquiera, o el que integra OptimaJ. La primera pantalla que se encuentra el usuario es la de la administración de sistemas, desde la que se pueden realizar consultas y crear sistemas.

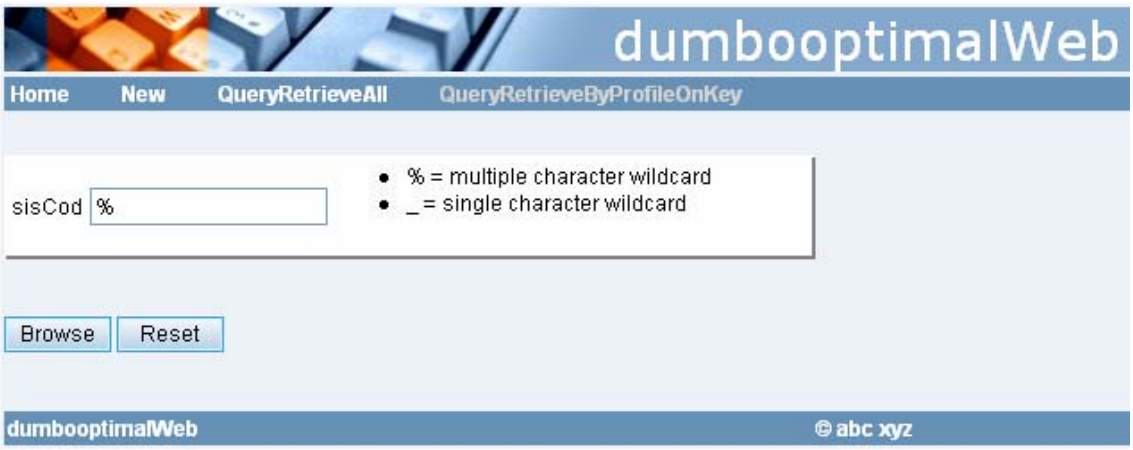


Figura 58. Aplicación de prueba. Inicio de la administración de sistemas.

Si se pulsa sobre la opción *new*, el navegador dirigirá al usuario hacia la página de creación de sistemas. En la página de creación de sistemas, además se

pueden añadir secciones al sistema a crear. El aspecto de la página es el que se muestra a continuación.

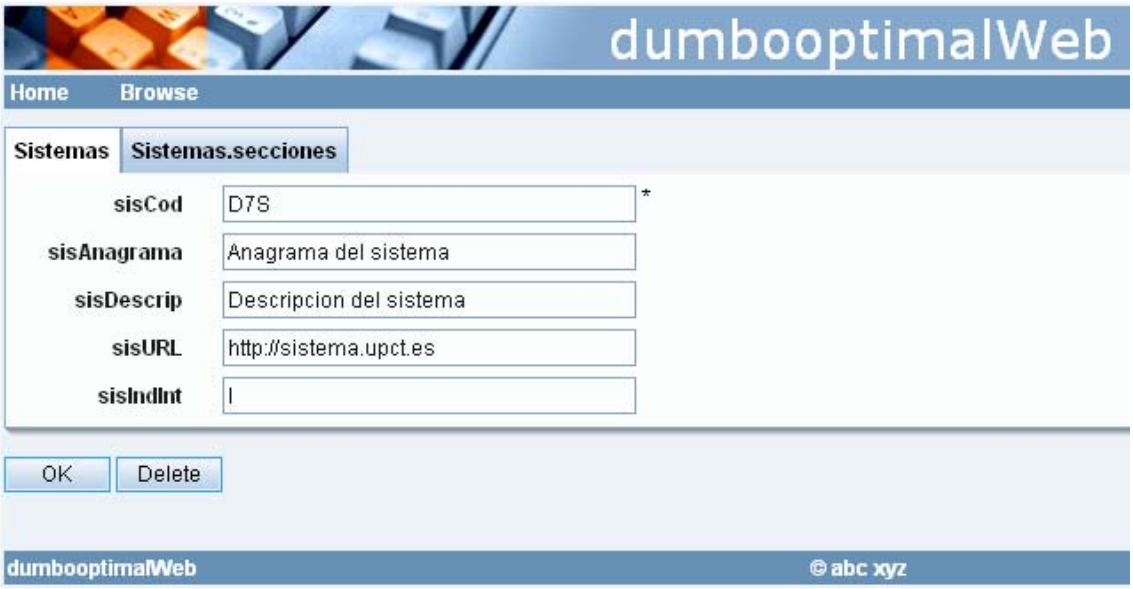


Figura 59. Aplicación de prueba. Creación de sistemas.

Una vez creados los sistemas, se podrá realizar un listado de los mismos, y también se podrán realizar modificaciones. Un listado de un conjunto de sistemas se presenta de la siguiente forma:

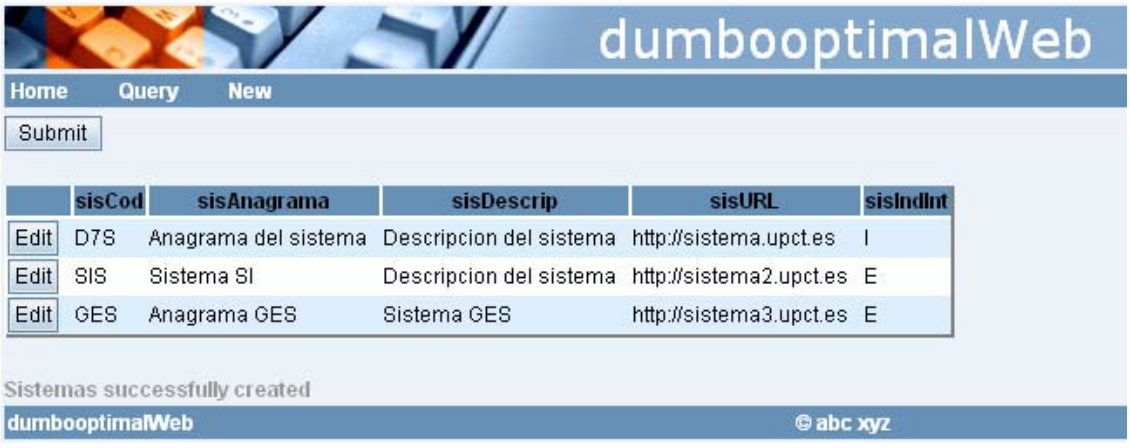


Figura 60. Aplicación de prueba. Listado de sistemas.

Si se pulsa en *edit*, se abrirá la página de modificación para un sistema en concreto, donde se podrán variar sus campos y añadir nuevas secciones.

Si en la página de creación de sistemas o modificación sistemas se pulsa sobre la opción *Sistemas.secciones*, se abrirá la página para crear nuevas secciones vinculadas al sistema activo en ese momento. El aspecto de dicha página es muy parecido al de crear sistemas, y permite de igual forma abrir ir al formulario de creación de aplicaciones para esa sección en concreto.

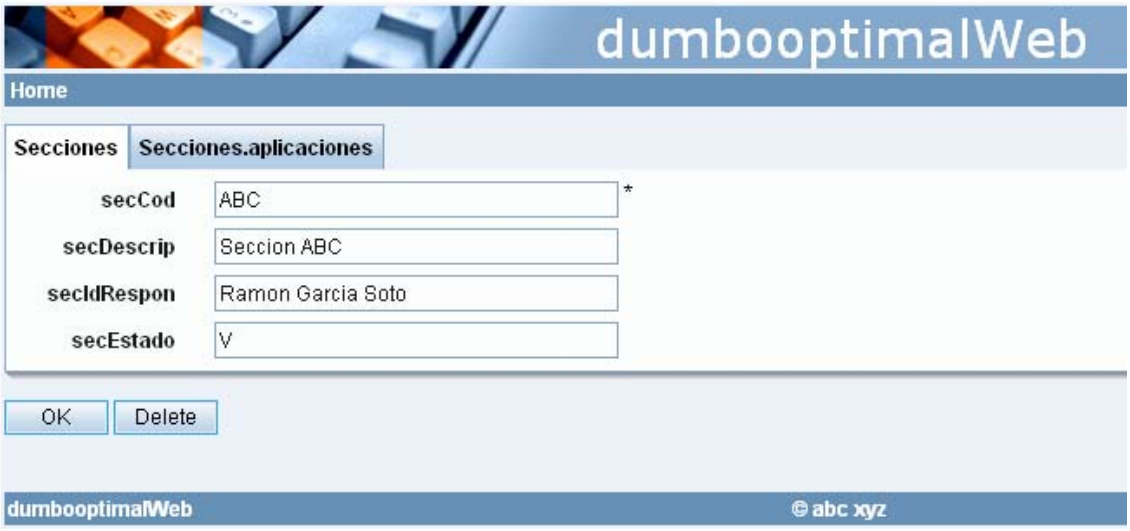


Figura 61. Aplicación de prueba. Creación de secciones.

Tras crear varias secciones, el aspecto del listado final de secciones para un sistema es el siguiente:

Figura 62. Aplicación de prueba. Listado de secciones.

Por último, quedan por analizar las operaciones con aplicaciones. La vista de los listados y formularios de creación y modificación son también muy parecidos a los sistemas y secciones. A continuación se muestra un listado de aplicaciones.

Figura 63. Aplicación de prueba. Listado de aplicación.

Un aspecto que hasta el momento no se ha tenido en cuenta ha sido el de los servicios en el dominio de modelo. En el siguiente apartado se presenta un ejemplo de como crear un servicio con OptimalJ.

9.2.6. Creación de un servicio con OptimalJ.

Como recordatorio, el modelo de servicios permite definir vistas sobre clases definidas en el modelo de clases, permitiendo ofrecer así acciones más complejas y personalizadas. Para la explicación de esta modelo, servirá el requisito SR10: Contar Aplicaciones.

Se pretende crear una interfaz a través de la cual el usuario pueda acceder al número de aplicaciones totales que poseen todos los sistemas. También se va a crear una interfaz un poco más compleja que permita obtener solamente las aplicaciones vinculadas a una sección. Hay que recordar también que una sección sólo puede pertenecer a un sistema.

Para crear el servicio hay que volver al modelo de dominio y crear un nuevo servicio de dominio (*DomainService*). De tal forma que el modelo de servicios queda tal y como se presenta a continuación:

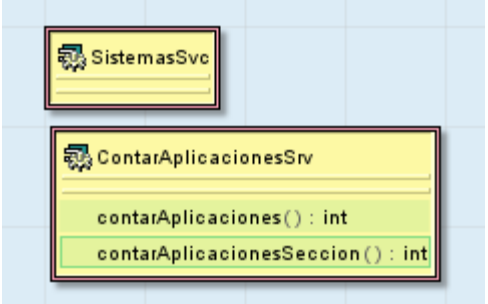


Figura 64. Modificación del modelo de servicio dentro del modelo de dominio.

El nuevo servicio se llama *ContarAplicacionesSrv* y contiene dos operaciones. La operación *contarAplicaciones* devuelve el número total de aplicaciones y *contarAplicacionesSeccion* devuelve el número de aplicaciones de una sección.

Para obtener el número total de aplicaciones de todos los sistemas no es necesario ningún parámetro de entrada, pero para contar las aplicaciones pertenecientes a una sección se requieren dos parámetros de entrada: el sistema y la sección (se debe indicar en *contarAplicacionesSeccion*).

Otro tema a tener en cuenta es que todas las clases del modelo de clases que vayan a intervenir en el servicio deben tener activa la propiedad *published*. Si no es así, no se generarán los objetos necesarios para hacer llamadas a métodos de objetos de forma remota.

Una vez modificado el modelo de servicios, se deben actualizar todos los modelos y se debe de volver a generar el código. Tras regenerar el código, se cambia la vista al modelo de aplicación y se selecciona el componente de la lógica de negocio *ContarAplicacionesSrv*, que se ha creado, después de la modificación, a través del patrón de transformación *EJBFromServicePattern*. Al seleccionar este componente, se habilitará a la derecha de la ventana una nueva opción llamada *Model2Code*, donde se muestran los bloques libres del componente seleccionado.

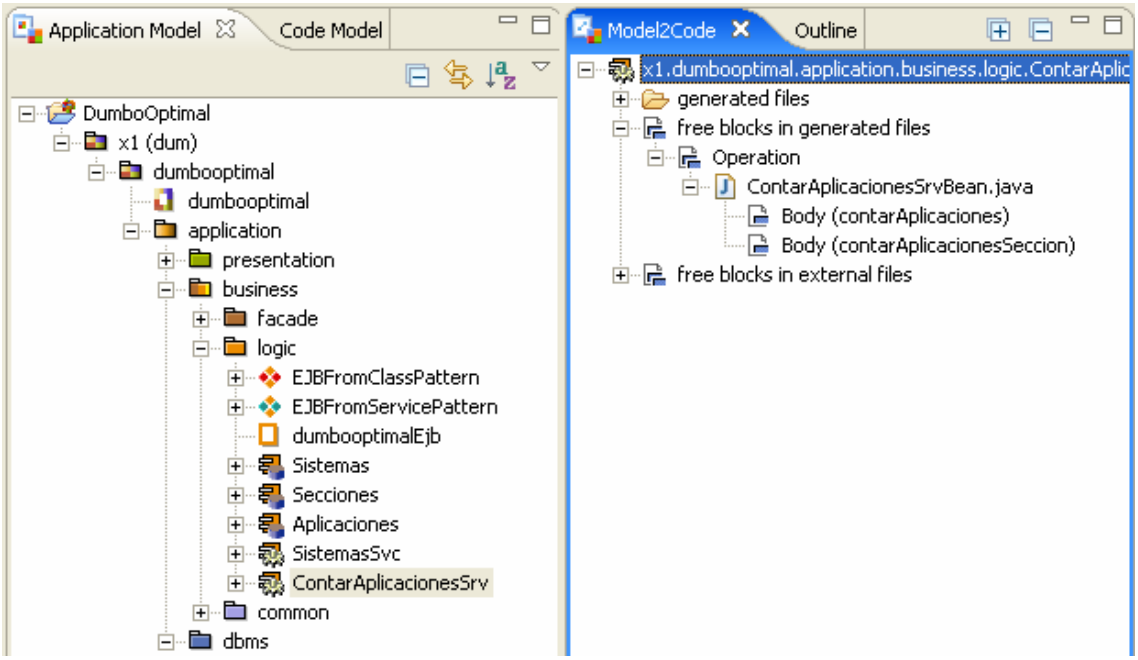


Figura 65. Detalle del navegador del modelo de lógica de negocio y de la ventana *Model2Code*.

En los bloques libres de *ContarAplicacionesSrv*, se encuentran las dos operaciones del servicio. Para implementar *contarAplicaciones*, se hace doble clic y entonces se abrirá el editor de código de OptimalJ. A continuación, se muestra el código implementado (marcado en azul), junto con el código perteneciente a los bloques protegidos (marcados en color rojo).

```
public int contarAplicaciones()throws AlturaPreConditionException,
AlturaPostConditionException
{
      int  returnValue = 0;
      try
      {
            AplicacionesHome aplicacionesHome = (AplicacionesHome)
serviceLocator.getRemoteHome("Aplicaciones",AplicacionesHome.class);
            Collection aplicacionesCollection =
aplicacionesHome.findAll();
            Iterator aplicacionesIterator =
aplicacionesCollection.iterator();
            while(aplicacionesIterator.hasNext())
            {
                  returnValue++;
                  aplicacionesIterator.next();
            }
       }
       catch(Exception e)
       {
            System.out.println("¡Error!");
       }
       return returnValue;
}
```

De igual forma, se muestra el código del servicio *contarAplicacionesSeccion*.

```
public int contarAplicacionesSeccion(String seccion, String sistema)
        throws AlturaPreConditionException,
AlturaPostConditionException
{
      int  returnValue = 0;
      try
      {
            AplicacionesHome aplicacionesHome =(AplicacionesHome)
serviceLocator.getRemoteHome("Secciones",AplicacionesHome.class);
            Collection aplicacionesCollection =
aplicacionesHome.findBySecciones(seccion,sistema);
            Iterator aplicacionesIterator =
aplicacionesCollection.iterator();
            while(aplicacionesIterator.hasNext())
            {
                  returnValue++;
                  aplicacionesIterator.next();
            }
        }
        catch(Exception e)
        {
            System.out.println("¡Error!");
        }
        return returnValue;
}
```

En ambos casos se obtiene un *stub*, para posteriormente invocar el método remoto necesario para realizar la consulta. En el primer caso se realiza la consulta mediante el método *findAll*, que devuelve todoa las aplicaciones, y en el segundo caso se utiliza *findBySecciones*, que devuelve las aplicaciones

vinculadas a la sección y al sistema que se pasan como argumentos. Ambos métodos están definidos en el modelo de lógica de negocio, dentro del modelo de aplicación.

Una vez personalizado el código de los servicios, se deben regenerar los archivos y ya se puede volver a lanzar el servidor de aplicaciones. En las siguientes figuras se muestra el resultado de esta última modificación.

Figura 66. Aplicación de prueba. Pantalla de acceso a los servicios implementados.

Si se selecciona la opción de menú *contarAplicaciones*, se redirigirá al usuario a una nueva pantalla donde se muestra el resultado de la operación.

Figura 67. Aplicación de prueba. Pantalla de resultados tras la ejecución del servicio contarAplicaciones.

En cambio, si se selecciona la opción de menú contarAplicacionesSeccion, se redirigirá al usuario a una pantalla donde podrá introducir el código de sección y el código de sistema con los que se ejecutará la consulta. Una vez realizada la consulta, se mostrará una pantalla con el resultado.

Figura 68. Aplicación de prueba. Formulario de introducción de datos para la ejecución del servicio contarAplicacionesSecciones.

Figura 67. Aplicación de prueba. Pantalla de resultados tras la ejecución del servicio contarAplicacionesSecciones.

Llegados a este punto, todos los requisitos impuestos inicialmente se han cumplido. La aplicación se puede dar por completada y sería el momento de comenzar las pruebas con QACenter, pero dado que no existe una versión de prueba de este producto, no se podrá continuar con esta última parte.

Antes de acabar con el apartado y dar paso a las conclusiones, convendría comentar unos aspectos importantes sobre el control de versiones para trabajo en equipo y OptimalJ.

9.2.7. Desarrollo en equipo en el entorno de OptimalJ

Una ventaja que ofrece OptimalJ, es la posibilidad de llevar a cabo proyectos de gran magnitud desarrollados mediante equipos de trabajo por medio de repositorios.

Antes de que un equipo de desarrollo comience a utilizar OptimalJ para producir aplicaciones mediante ciclos iterativos de desarrollo, el dominio de clases y las principales restricciones de los subsistemas deben ser capturados en el dominio de modelo. Este modelo es el resultado del análisis de negocio combinado con prototipos de la arquitectura realizados mediante casos de uso. Este modelo, expresado como un modelo UML de clases, tiene dos propósitos:

1. El mismo modelo está disponible para analistas y desarrolladores.
2. El modelo es la base para generar y mantener los modelos de aplicación y, como consecuencia de esto, del código generado finalmente.

En la fase de análisis, los analistas determinan el nivel de modularización. La finalidad de esto es crear un modelo de dominio los más completo y estable que sea posible para cada uno de los subsistemas. Los métodos RAD para prototipado de casos de uso ayudan a definir las clases necesarias, junto con las clases y dependencias necesarias entre los componentes de la aplicación. OptimalJ ofrece herramientas para el diseño interactivo de prototipos RAD.

En esta fase, los analistas comparten todos los subsistemas que son dependientes unos de otros, con la intención de definir la arquitectura con la mínima dependencia y la máxima cohesión.

Cuando se trabaja con OptimalJ y un repositorio de modelos, solo es necesario mantener un control de versiones de los ficheros modelo (*.xcm). El código generado para los prototipos puede ser ignorado, ya que este se genera a partir de los modelos.

La finalidad de la fase de análisis es ofrecer un producto medianamente estable, desde el cual se pueda arrancar y comenzar a desarrollar en profundidad cada uno de los subsistemas. En esta fase se tiene en cuenta, sobre todo, la interacción entre subsistemas. De forma más concreta, los objetivos a conseguir son los siguientes:

1. Obtener un modelo de dominio de clases casi completo para cada subsistema.
2. Un modelo de servicio que contenga los servicios de los casos de uso. Normalmente se define un servicio para cada caso de uso.
3. Obtener un modelo de dominio de aplicación casi completo, generado a partir del modelo de dominio.

El siguiente paso a realizar es generar código que se pueda instalar en un servidor de aplicaciones para realizar pruebas de la versión inicial de la aplicación. Si existen dependencias entre los subsistemas, las pruebas se realizaran sobre los subsistemas relacionados al mismo tiempo.

Los subsistemas ya están listos para compartirlos entre los miembros del equipo de desarrollo.

9.2.7.1. Puesta en marcha del entorno de gestión de configuración.

Los procesos y herramientas requeridas deben dar soporte a los siguientes mecanismos:

- La versión inicial y las versiones actualizadas de los subsistemas deben poder ser distribuidas a todos los miembros del equipo desde un punto central.
- Los cambios y las nuevas implementaciones de los miembros del equipo deben poder actualizar las versiones anteriores y estar disponibles desde un punto central.

Se supone que debe existir un repositorio con control de versiones inicializado, y debe estar configurado para ignorar ciertos ficheros de OptimalJ. De esto se hablará más adelante, ya que la finalidad de este apartado es dar una visión teórica del entorno de gestión de configuración. Cada uno de los miembros del equipo de desarrollo debe disponer solamente de una copia del cliente utilizado para acceder al repositorio.

El siguiente paso es importar todos los ficheros pertenecientes al subsistema a un nuevo módulo CVS dentro del repositorio de control de versiones. Todos los miembros deben poder acceder al nuevo módulo desde su entorno local. Tras

la comprobación, los desarrolladores pueden montar las carpetas del repositorio en su entorno local, para obtener el modelo completo y el código generado. De aquí en adelante:

- Los desarrolladores pueden comprobar los cambios del código generado en el repositorio por si mismos.
- Los desarrolladores pueden aplicar y comprobar cambios limitados en el modelo del repositorio por si mismos.

9.2.7.2. Sincronización de cambios.

OptimalJ almacena sus modelos en ficheros de texto con formato XMI, como ya se sabe de apartados anteriores. Cuando se realizan cambios paralelos en un fichero de texto, el CVS intenta realizar la mejor mezcla entre las modificaciones.

Si las modificaciones se solapan, entonces el CVS aplica marcas de conflicto entre versiones y asigna el estatus al fichero de "Conflicto sin resolver" ("*Unresolved Conflict*"). Esto genera un problema, ya que dichas marcas no permiten a OptimalJ leer los ficheros, y los editores visuales de modelo no son capaces de resolver este tipo de conflictos. Además, en las últimas versiones de OptimalJ, los ficheros dentro del repositorio pueden tener dependencias entre ellos. Por estas razones, las actualizaciones concurrentes de ficheros de modelo deben estar prohibidas.

Los cambios se pueden categorizar de la siguiente forma:
- Cambios en el dominio de modelo.
- Cambios en el dominio de aplicación.
- Cambios de código fuente.

Los **cambios en el modelo de dominio** pueden provocar un gran impacto en todos los subsistemas. Estos cambios afectan normalmente a muchos componentes dentro del modelo de aplicación, y consecuentemente afecta a muchos ficheros del código fuente. Los cambios de modelo suelen proceder de cambios en los requisitos de la aplicación, nuevas vistas en el modelo de negocio o cambios arquitectónicos en la aplicación. Los cambios son aplicados por los analistas y, ya que afectan de forma potencial a muchos componentes del sistema, deben ser introducidos de forma coordinada. Para realizar cambios en el modelo de domino se debe seguir el siguiente procedimiento:

1. Todos los desarrolladores deben comprobar si su modelo de aplicación o de código tienen cambios sin actualizar.
2. Los analistas comprueban los cambios aplicados por los otros miembros del equipo. Esto implica a todos los ficheros del modelo de aplicación y todos los ficheros del modelo de código generados por los ficheros del modelo de dominio que van a ser modificados.
3. El analista aplica cambios al dominio de modelo.
4. El analista regenera todo el código de modelo de aplicación y el código fuente resultante del modelo de aplicación.

5. El analista comprueba si todos los ficheros fueron actualizados tras el paso anterior.
6. Los desarrolladores obtienen el nuevo modelo de dominio, el nuevo modelo de aplicación y el nuevo código fuente generado.

Se debe de volver a recordar que los cambios concurrentes dentro del dominio de modelo deben estar prohibidos.

Los cambios en el modelo de aplicación, son cambios que afectan a un componente dentro del modelo de aplicación. Los cambios dentro de un componente del modelo de aplicación afectan al código fuente del modelo de código. Estos cambios afectan a todos los miembros que trabajan con dicho componente del modelo de aplicación. Estos cambios pueden ser realizados por el desarrollador, pero siempre con el analista como medio de sincronización. Los pasos para realizar un cambio de este tipo son los siguientes:

1. El desarrollador aplica cambios en su modelo de aplicación, regenera el código y realiza pruebas sobre él.
2. El desarrollador se asegura de actualizar su entorno con los cambios aplicados por otros miembros del equipo. Esto implica que todos los ficheros del modelo de aplicación y todos los ficheros de código fuente generados por el modelo de aplicación son los correctos.
3. Si se detecta una nueva versión de los ficheros del modelo de aplicación en el paso anterior, el desarrollador deberá volver a regenerar el código y volver a realizar sus pruebas locales.
4. El desarrollador pone a disposición del resto de miembros del equipo los cambios realizados.

Los cambios en el código fuente son los menos problemáticos, ya que se realizan sobre los bloques libres que se generaron tras la transformación del modelo de aplicación a modelo de código. Se permiten cambios concurrentes ya que se trata de código fuente normal. De todas formas, los cambios en el código fuente afectan a otros miembros del equipo de trabajo que realizan llamadas desde su código a los métodos modificados, y un módulo no actualizado correctamente puede llevar a la aplicación a un mal funcionamiento. Los pasos a seguir al actualizar cambios en el modelo de código son los siguientes:

1. El desarrollador aplica cambios locales y prueba el código.
2. El desarrollador se asegura de que su entorno local está correctamente actualizado con los cambios realizados por otros miembros del equipo. Esta parte implica también a todos los elementos del modelo de aplicación. También puede llevar al desarrollador a resolver conflictos existentes con otras versiones de otros desarrolladores.
3. Si la nueva versión del modelo de aplicación no era la correcta en el paso anterior, se deberá regenerar el código y volver a realizar las pruebas locales.
4. El desarrollador actualiza en el repositorio sus cambios para que el resto de miembros del equipo puedan actualizar los cambios.

9.2.7.3. El modelo de repositorio de OptimalJ.

OptimalJ hace uso de una avanzada tecnología para almacenar sus modelos, a ésta se le llama modelo de repositorio. El modelo de repositorio se basa en un sistema de ficheros XML que usa el formato XMI. Para desarrollo en equipo, es muy útil entender como se almacenan estos ficheros y que información contiene cada uno de ellos.

Los ficheros del repositorio tienen la extensión ".xcm", y cada nombre consta de tres partes:

- Nombre del elemento raíz.
- Tipo del nodo raíz.
- Nombre del submodelo (opcional).

Hay que tener en cuenta que algunos elementos hijos del modelo pueden ser elementos raíz para otros. También se debe tener en cuenta que al borrar o crear un elemento hijo que sirve de raíz a otro fichero del repositorio, implica la modificación del fichero que contiene el padre de ese hijo.

Los ficheros que intervienen en un proyecto OptimalJ son los mismos que intervienen en un proyecto normal junto con otros específicos. A continuación se presenta un listado de dichos ficheros:

Extensión del fichero	Texto o binario	Control de versiones o local	Descripción
*.xcm	Texto	Control de versiones	Fichero de repositorio de OptimalJ, en formato XMI.
*.mcr	Binario	Local	Mantiene la integridad entre elementos de modelo y ficheros de código fuente generados. Permite navegar por los ficheros generados a través de elementos del modelo.
*.xenon	Texto	Local	Contiene información para los editores de diagramas de OptimalJ en formato XMI.
*.sql	Texto	Control de versiones	Contiene DDL o DML en sintaxis SQL.
*.sqm	Texto	Control de versiones	Metainformación requerida por el cliente SQL para acceder a la base de datos.
*.java	Texto	Control de versiones	Código fuente Java.
*.xml	Texto	Control de versiones	Ficheros en formato XML.
*.jsp	Texto	Control de versiones	Fichero Java Server Page
*.properties	Texto	Control de versiones	Ficheros de propiedades para Struts.

Extensión del fichero	Texto o binario	Control de versiones o local	Descripción
*.mf	Texto	Control de versiones	Fichero de manifiesto
*.tld	Texto	Control de versiones	Fichero para definir librerias de etiquetas en Struts.
*.html	Texto	Control de versiones	Ficheros Hypertext markup Language.
*.css	Texto	Control de versiones	Ficheros de estilos.
*.class	Binario	Local	Ficheros compilados Java.
*.eardef	Texto	Local	Ficheros de definición de aplicación en formato XML.
*.ear	Binario	Local	Fichero Enterprise Archive Resource.
*.jar	Binario	Local	Ficheros JAR.
*.war	Binario	Local	Fichero Web Application Resource.
*.dar	Binario	Local	Fichero Database Application Resource.
*.idx	Binario	Local	Índice para repositorio de OptimalJ.
.nbattrs	Texto	Local	Fichero de atributos. Existe en cada subdirectorio del sistema de ficheros montado en el modelo de código. Describe el contenido de cada subdirectorio.
.~		Local	Copias de seguridad.

Tabla 3. Ficheros que intervienen en un proyecto con OptimalJ.

Cabe destacar los siguientes comentarios:

- Los ficheros *.mcr contienen información relacionada con el modelo y el código generado en local, y no debe estar en ningún momento en bajo control de versiones.
- Los ficheros *.eardef contienen información especifica del sistema local, como por ejemplo la localización el disco del fichero ejb.jar y web.war. Esta información varía según el desarrollador y, por este motivo, no debe estar bajo control de versiones.
- Los ficheros *.jar generados por OptimalJ, como por ejemplo ejb.jar y web.war, pueden ser regenerados a partir de las clases compiladas Java, por lo que, normalmente, no están bajo control de versiones.
- Los ficheros .nbattrs contienen información del contenido de los directorios locales y no deben estar bajo control de versiones.

10. Conclusiones.

Se ha desarrollado una aplicación siguiendo el nuevo enfoque de desarrollo dirigido por modelos. El tiempo en el que se consigue una versión estable de la aplicación es mucho menor que el tiempo que se emplearía en desarrollar la misma de la forma tradicional, y con la seguridad de que la implementación sigue al pie de la letra el análisis y el diseño planteados. La calidad de las aplicaciones y la productividad de las empresas se ven mejoradas. Las aplicaciones implementadas son más robustas a la hora de realizar cambios, por lo que el mantenimiento se agiliza. Si se requiriera realizar un cambio en el modelo inicial, este no afectaría a la implementación debido al alto grado de consistencia incremental que ofrece la herramienta.

El entorno de desarrollo elegido consigue cubrir todas las fases del ciclo de vida del producto bajo desarrollo, ya que OptimalJ cubre las fases de análisis, diseño e implementación; Optimal Trace cubre las fases de gestión e implementación de requisitos y QACenter cubriría todas las fases de pruebas.

Se ha conseguido mejorar las líneas de trabajo expuestas en el apartado 2, sin alejarnos de ellas. OptimalJ permite la utilización de un repositorio con el que puede trabajar un equipo de desarrollo y, además, permite crear de forma rápida y eficiente aplicaciones *web* en tres capas con persistencia de datos mediante Enterprise Java Beans. En la próxima versión 4.2, se permitirá persistencia mediante Hibernate, que es la herramienta que se ha utilizado en el Servicio de Informática en su último proyecto (DUMBO V.7).

Hay que tener en cuenta que la tecnología empleada es nueva, donde aún no hay un estándar cerrado ni todas las compañías están siguiendo exactamente la misma filosofía. Esto se puede observar de forma clara en el caso de, por ejemplo, ArcStyler, OptimalJ y Rhapsody, donde ciertas criterios que deberían converger no lo hacen.

Se debe empezar a promover tanto en el mundo de la docencia como en el mundo del desarrollo comercial el uso de herramientas que implementen este nuevo enfoque, para que así comience una transición hacia mejores herramientas de desarrollo y, poco a poco, estas se integren en el mundo de la ingeniería del *software* y converjan hacia un estándar.

Este estándar podría incluir tanto el enfoque de desarrollo MDA como el enfoque de las Factorías de *Software*, que no son contrarios, sino complementarios. El desarrollo de herramientas que integraran ambas soluciones lograría un desarrollo más fácil, cómodo y rápido, y permitirían finalmente la entrada de la ingeniería del *software* en la fase industrial.

Glosario

Automated Build Framework (ABF). Herramienta de Microsoft para la creación de *scripts* que permiten realizar actualizaciones automáticas.

Atomistic Information Mapping (AIM). Respuesta de Interactive Objects a la propuesta QVT planteada por el OMG. Se presenta como el motor de transformación que utiliza ArcStyler.

Borland LiveSource. Motor de transformación de modelos propuesto por Borland.

Capability Maturity Model Integration (CMMI). Modelo para la mejora o evaluación de los procesos de desarrollo y mantenimiento de sistemas y productos de *software*.

Common Warehouse Metamodel (CWM). Especificación que contiene directivas necesarias para poder almacenar la meta-información de cualquier modelo mediante un formato estándar y fácilmente intercambiable.

Dynamic Systems Framework (DSF). Herramienta de Microsoft que facilita el almacenamiento, despliegue y reutilización de recursos.

Dynamic Systems Initiative (DSI). Iniciativa de Microsoft para simplificar y automatizar la forma en que las empresas diseñan, implementan y operan los sistemas de TI.

Domain Specific Language (DSL). Lenguaje de programación diseñado para ser útil para un grupo específico de tareas. Es un concepto totalmente contrario a UML.

Enterprise Java Beans (EJB). Son una de las API que forman parte del estándar de construcción de aplicaciones empresariales J2EE. Su especificación detalla cómo los servidores de aplicaciones proveen objetos desde el lado del servidor.

Eclipse Modeling Framework (EMF). *Framework* de modelado y para la generación de código, con características similares a MOF.

Factorías de *Software*. Enfoque, propuesto por Microsoft, para el desarrollo de aplicaciones donde el proceso se basa en el desarrollo de familias de productos y la reutilización de componentes.

Hibernate. Herramienta para la plataforma Java que facilita el mapeo de atributos entre una base de datos relacional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) que permiten establecer estas relaciones.

IT Infraestructure Library (ITIL). Marco de trabajo empleado en los departamentos de Tecnologías de información (TIC) que utiliza un enfoque de mejores prácticas para aumentar la calidad en los servicios que ofrecen.

Java Develpment Tools (JDT). Proyecto integrado en la Plataforma Eclipse que soporte al desarrollo de cualquier aplicación Java, incluyendo *plug-ins* para Eclipse.

Java Metadata Interface (JMI). Especificación para Java, basada en MOF, para implementar infraestructuras dinámicas e independientes de plataforma alguna.

Java Server Pages (JSP). Tecnología Java que permite a los programadores generar contenido dinámico para publicaciones *web*, en forma de documentos HTML, XML o de otro tipo.

Model Driven Architecture (MDA). Especificación del paradigma MDD propuesta por el OMG.

Model Driven Development (MDD). Enfoque para el desarrollo de aplicaciones donde todo el proceso viene dirigido por modelos abstractos de alto nivel.

METRICA Versión 3. Metodología que ayuda a las organizaciones públicas a sistematizar las actividades que dan soporte al ciclo de vida del *software*.

Meta-Object Facility (MOF). Lenguaje para el diseño de Metamodelos, a partir del que se pueden definir lenguajes de modelado.

Model Transformation Framework (MTF). Motor de transformación de modelos propuesto por IBM Rational. Parte de EMF.

Object Constraint Language (OCL). Lenguaje adoptado por el OMG como parte de UML 2.0 para la descripción formal de expresiones en los modelos UML.

Object Management Group (OMG). Consorcio dedicado al cuidado y al establecimiento de diversos estándares de tecnologías orientadas a objetos.

Plug-in Develpment Environment (PDE). Entorno de desarrollo de *plug-ins* para Eclipse.

Platform Independent Model (PIM). Modelo propuesto por el OMG para el diseño de modelos independientes de plataforma alguna.

Platform-Specific Model (PSM). Modelo propuesto por el OMG para el diseño de modelos dependientes de una plataforma específica.

Query/View/Transformations (QVT). Solicitud de propuestas lanzada por el OMG para suplir la falta de un estándar de transformación en el ámbito de MOF.

Rational Unified Process (RUP). Metodología de desarrollo *software* producto de IBM Rational. Se caracteriza por ser iterativo e incremental, estar centrada en la arquitectura y guiado por casos de uso.

Specification and Description Language (SDL). Lenguaje diseñado para la especificación de sistemas complejos, interactivos, orientados a eventos, de tiempo real o que presentan un comportamiento paralelo, y donde módulos o entidades independientes se comuniquen por medio de señales para efectuar su función.

Software Lifecycle Platform (SLP). Plataforma para la gestión del ciclo de vida de aplicaciones *software* propuesta por Microsoft.

Software Process Engineering Metamodel (SPEM). Metamodelo empleado en la definición de modelos de procesos concretos.

System Modeling Language (SysML). Lenguaje de dominio específico para el modelado de aplicaciones de ingeniería de sistemas.

Template Pattern Language (TPL). Respuesta de Compuware y Sun Microsystems a la propuesta QVT planteada por el OMG. Se presenta como el motor de transformación que utiliza OptimalJ.

Tree & Tabular Combined Notation (TTNC). Lenguaje estándar para definición de casos de pruebas.

Unified Modeling Language (UML). Lenguaje de modelado de sistemas de *software*.

XML Metadata Interchange (XMI). Especificación, basada en XML, para el intercambio de diagramas.

eXtensible Markup Language (XML). Metalenguaje extensible de etiquetas que permite definir la gramática de lenguajes específicos. No es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades.

eXtensible Stylesheet Language (XSL). Familia de lenguajes basados en el estándar XML, que permite describir cómo la información contenida en un documento XML cualquiera debe ser transformada o formateada para su presentación en un medio específico.

Bibliografía

[1]. *Web* oficial de la Plataforma Eclipse.
http://www.eclipse.org .

[2]. "Ingeniería del Software", Ian Sommerville, 2005. Addison Wesley.

[3]. "Ingeniería del Software", Roger S. Pressman, 2001. MacGraw Hill.

[4]. "Proceso Unificado de Desarrollo de Software", Jacobson, Booch y Rambaugh, 1999. Addisson Wesley.

[5]. "Una Guía del CMM: Comprender el modelo de Madurez de Capacidad del Software", Kenneth M. Dymond, 1998. Process Transition International.

[6]. "Introducción a la metodología ITIL". Abast Group.

[7]. "Metodología de Planificación, Desarrollo y Mantenimiento de sistemas de información".
http://www.csi.map.es/csi/metrica3/ .

[8]. *Web* oficial del Objects Management Group.
http://www.omg.org/ .

[9]. "Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools", Jack Greenfield y Keith Short. John Wiley & Sons.

[10]. Referencia a compañías comprometidas con MDA.
http://www.omg.org/mda/committed-products.htm .

[11]. Web de Visual Studio Team y Visual Studio Team Systems.
http://www.microsoft.com/latam/vstudio .

[12]. *Web* oficial de Borland.
http://www.borland.com .

[13]. *Web* oficial de IBM Rational Software.
http://www-306.ibm.com/software/rational/ .

[14]. *Web* oficial de Telelogic.
http://www.telelogic.com .

[15]. *Web* oficial de I-Logix.
http://www.ilogix.com .

[16]. *Web* oficial de Interactive Objects.
http://www.interactive-objects.com .

[17]. *Web* oficial de Compuware.
http://www.compuware.com .