Universidad de Valladolid

E. T. S. de Ingeniería Informática

Ingeniero en Informática

Revisión e Instanciación de un Framework basado en Reutilización para Herramientas de Desarrollo

---

Alumno: Julián Andújar Puertas

Tutores: Francisco Javier Pérez García
Yania Crespo González-Carvajal

La alegría está en la lucha, en el esfuerzo,
en el sufrimiento que supone la lucha,
y no en la victoria misma.
(Mahatma Gandhi)

Agradecimientos

a mi madre,
este proyecto es tuyo,
hemos trabajado duro, y tu todavía sigues.
No sé como agradecerte lo que haces por nosotros.
Eres mi estrella, la mejor madre.
Gracias por lo que nos ofreces
y por quien eres, te quiero mucho.

a mis hermanos,
mis amigos, sois mi experiencia, mis consejos
y mi ayuda, cada uno en su línea. Os quiero

por supuesto a ti, Alfonso,
has visto como si que te he metido!.
Eres como mi hermano, aunque me aguantas
más que ellos ;). Eres alguien especial y me
siento afortunado por conocerte. Te quiero.

a mis Tutores,
gracias por vuestra ayuda
e incluso vuestro apoyo en ciertos
momentos, en especial a ti
Javi, que has aguantado lo pesado
que puedo ser.

Índice

Capítulo 1

Introducción

1.1 Introducción

Hasta tiempos muy recientes el proceso de fabricación de software ha adolecido de un conjunto de características descritas por Pressman y Fairley: una planificación y estimación de costes imprecisa, poca productividad, elevadas cargas de mantenimiento, demandas cada vez más desfasadas con las ofertas, baja calidad de los productos y dependencia de los realizadores.

Actualmente nadie duda sobre el provecho de la reutilización, es una de las piezas clave para conseguir mejoras significativas en cuanto a productividad y calidad en el proceso de desarrollo del software. Es una solución para dar respuesta a un mercado que exige productos y servicios cada vez más fiables, baratos y entregados a tiempo.

Un paradigma muy extendido y que favorece de manera especial la reutilización, son los lenguajes Orientados a Objetos, ya que podemos empezar a analizar utilizando desde pequeñas piezas reutilizables a grandes elementos, creados en otras etapas o procesos, según el grado y forma del proyecto que manejemos.

Podemos ver la reutilización desde dos disciplinas, una que es el desarrollo para la reutilización y otra el desarrollo con reutilización. Ambas se complementan, y tienen puntos de vista diversos que siguen evolucionando e investigando como pueden ser la métrica y la reutilización.

La introducción de la reutilización sistemática de software en los procesos de desarrollo requiere, en primer lugar, lo que podemos llamar un modelo reutilización en el que son imprescindibles la definición de un modelo de componente reutilizable y de un proceso de desarrollo basado en reutilización.

Por otra parte, se requiere contar con definiciones y técnicas de certificación de calidad, cambios organizacionales, formación en técnicas específicas para el desarrollo basado en reutilización y un soporte operativo.

Otro enfoque que gana importancia dentro del ámbito de la reutilización es el desarrollo basado en ***frameworks***. Los ***frameworks*** son las "plantillas", que definen e implementan la arquitectura y funcionalidades comunes de un dominio específico de aplicaciones. Los *frameworks* permiten generar familias de aplicaciones pertenecientes a un mismo dominio. Por lo tanto, debe haber puntos de flexibilidad que se puedan modificar los requisitos particulares para ajustarse a la aplicación.

Los puntos de extensión de un *framework* se llaman **puntos calientes (hot-spots).** Los puntos calientes o Hot-spots son clases y métodos donde el framework se modifica, extiende, etc. para obtener una aplicación. A la aplicación obtenida a partir del framework se le denomina instancia del framework.

En este trabajo se revisa la definición, diseño e implementación de un framework de entornos distribuidos para el desarrollo de software basado en reutilización, así como un primer paso a la instanciación de un caso práctico. Por la propia definición de framework se permitirá la inserción e integración de herramientas existentes así como de nuevas.

Se ha utilizado una arquitectura genérica de colaboración entre entidades distribuidas capaces de trabajar según el modelo de reutilización.

Esta arquitectura viene definida por:

- Tipo de entidades (nodos) que pueden participar en la colaboración.
- Protocolo de comunicación, que permite a dichas entidades solicitar servicios e intercambiar objetos pertenecientes al modelo de componente reutilizable.

- Servicios básicos, que pueden ofrecer los nodos
- Un diseño basado en frameworks que permite la reutilización de mecanismos y servicios a la vez que la independencia de la implementación y la tecnología particular del nodo.

Por último, se debe mencionar dos importantes hechos; el modelo de componente reutilizable que se ha seleccionado, es el modelo de reutilización de MECANO, del grupo GIRO de la Universidad de Valladolid [Gar00], por ser el modelo para el que se tiene acceso a un buen número de herramientas y bases teóricas; los pilares del diseño del framework que se ha instanciado, se basan en otro proyecto fin de carrera <<*Entorno de Desarrollo Distribuido Basado en Reutilización para el Modelo de MECANO*>>, este proyecto consiguió fijar una base teórica sobre un entorno y el diseño funcional de cada una de las partes que lo formaban. Este proyecto es una evolución del primero, reproduciendo un entorno de explotación parte del fundamente teórico, redefiniendo y perfeccionando métodos ya existentes y diseñando e implementado otros nuevos.

1.2 Objetivos del proyecto

El objetivo del proyecto es el del desarrollo de un framework, partiendo de una serie de definiciones teóricas marcadas en el proyecto *[Per03]*. Con este framework se creará un entorno de explotación, cuyos principales servicios son los definidos en el proyecto *[NCAP02]*.

Con la instanciación del framework se consigue afianzar y mejorar características, creando nuevas y/o redefiniendo y utilizando los ya existentes. Esta evolución también es un objetivo de este proyecto, pues se marcarán los métodos y partes del framework que han sido alteradas, construyendo una documentación actualizada y fiable para el estudio de futuros usuarios.

Los objetivos se han dividido en principales o secundarios.

Después de la definición de los objetivos, éstos se han revisado con el framework que hemos obtenido y se han marcado como:

- *Abierto*: Se ha tocado parte de la funcionalidad y se han dejado funcionalidades definidas de forma teórica para posibles y futuras evoluciones.
- *Conseguido*: Cuando el objetivo se ha conseguido. Indicar que aunque el objetivo se haya dado por finalizado, por propia definición del framework es algo evolutivo y modificable.

1.2.1 Objetivos principales

Se han identificado los siguientes objetivos principales:

- Revisar la arquitectura del framework existente para que consigamos enviar, recibir y procesar mensajes. (*Conseguido*)
- Implementar la parte de la arquitectura que permite que al instanciar un nodo cargue las peculiaridades propias de su tipo así como aquellas que deseemos añadir, permitiendo incrementar sus propiedades. (*Conseguido*)
- Poner en práctica la funcionalidad básica que permita recibir un mensaje X-MIP, y transformar los datos que contiene en una representación que sea fácilmente accesible al resto de las funciones de un nodo. (*Conseguido*)
- Perfeccionar y codificar la funcionalidad que permite construir y enviar mensajes X-MIP. Un mensaje se formará inicialmente a petición del usuario y mediante alguno de los servicios o funciones de un nodo e irá dirigido a otro nodo que responderá con otro mensaje después de procesar y valorar el primero. (*Conseguido para los casos prácticos*)
- Utilizar la arquitectura multihilo y especializar sus métodos para que permita:
    - Resolver las peticiones de los usuarios locales convirtiéndolas en un tipo de mensaje apropiado (esto sólo se utilizará en el caso de los nodos Tool). (*Conseguido*)
    - Resolver las peticiones recibidas desde las diferentes funciones en ejecución en un nodo. (*Conseguido*)
    - Recibir varias peticiones desde el exterior. (*Abierto*)
- Se debe poder codificar los mensajes que enviamos de un nodo a otro, comprimiéndolos y/o encriptándolos, consiguiendo un grado de seguridad en el protocolo y, según los parámetros con los que se desarrolle, un tamaño de mensaje menor. Asimismo se debe poder incorporar nuevos métodos de codificación a los nodos. (*Conseguido*)
- Desarrollar mecanismos que permitan al usuario configurar un nodo mediante fichero de texto, de manera que al ejecutarse el nodo tomará los parámetros definidos por el usuario. (*Conseguido*)
- Se debe poder utilizar otros modelos de reutilización, además del de MECANO. El empleo de otros modelos debe ser sencilla. (*Abierto*).

- Se debe implementar algún servicio de comunicación entre un nodo y un Broker, para tener un ejemplo práctico de este tipo de comunicación. (*Conseguido*)
- Se debe poder enviar un mensaje con objetos y asociaciones coherentes, y que sean almacenadas en el repositorio localizado en GIRO. (*Conseguido*)
- Inserción de elementos Assets, desde la herramienta Together 6.2 al repositorio de GIRO, quedando abierta la inserción de elementos MECANO. (*Conseguido*)
- Que la documentación sea lo más aproximada al proyecto [*Per03*] así como clara, plasmando la evolución del proyecto y primera revisión de la documentación inicial.

1.2.2 Objetivos secundarios

Se han identificado los siguientes objetivos secundarios, que se presentan por orden de prioridad:

- Realizar adaptaciones de las herramientas disponibles, para que puedan integrarse en un entorno de desarrollo de pruebas, como instancias del framework. (*Conseguido*)
- Realizar pruebas de los nodos en un entorno de trabajo real. (*Conseguido*)
- Poner en práctica el mecanismo LOG, que nos permite almacenar en un fichero de texto, los mensajes que se van produciendo en la ejecución de un nodo. (*Conseguido*)
- Desarrollar los servicios y las funcionalidades que se han definido como opcionales y que se pueden ver en el Apéndice C localizado en el CD del proyecto. Por ejemplo:
  - Intercambio automático de información de enrutamiento entre los nodos broker. (*Abierto*)
  - ...

1.3 Organización de la memoria

Los contenidos del trabajo se organizan estructurando en los siguientes capítulos y partes:

Planificación, en este tema se mostrará las pautas temporales, riesgos y seguimiento de este proyecto.

Parte I Definición del entorno de desarrollo. Se definen y se describen brevemente los diferentes elementos que constituyen el entorno de desarrollo.

Parte II Plan de desarrollo del proyecto y desarrollo de un framework para el entorno distribuido.

Parte III Instanciación del framework, desarrollo de los nodos Tool (en **Together 6.2**), y FileBroker.

Conclusiones y líneas futuras que nos indican. Se recogen las conclusiones del proyecto, se plantean las tareas pendientes y las líneas de trabajo abiertas.

Planificación

El plan de proyecto señala cada una de las tareas que se han desarrollado para la obtención de este proyecto. Estas tareas van desde el estudio de definición del entorno de desarrollo del framework, realizado en el proyecto fin de carrera [*Per03*], hasta la elaboración de la documentación, pasando por características de evolución, implementación e instanciación del framework así como elaboración de servicios basados en otro proyecto fin de carrera [*NCAP02*].

En el Capítulo 2 reflejamos los recursos del proyectos, el análisis de riesgos, estimaciones temporales, seguimiento del proyecto y el plan de pruebas.

Primera Parte

En el Capítulo 2 se presenta la arquitectura propuesta, la definición de los tipos de entidades (nodos) que participan y los servicios que estas pueden ofrecer. No se ha querido alargar este tema pues hacemos referencia al proyecto base. Comentar que esta parte será introducida en el Apéndice C de ésta documentación, pues la documentación inicial fue revisa con los cambios realizados.

Este capítulo también presenta la definición del sistema a instanciar.

Segunda Parte

Compuesto por 5 Capítulos:

Capítulo 4: que refleja el rediseño, y ampliación de la Iteración 1 del proyecto inicial.

Capítulo 5, Iteración2: evolución del paquete de comunicación, es la parte que menos cambios ha sufrido.

Capítulo 6, Iteración 3: implementación de los modelos que utilizamos, modelo de reutilización (MECANO) y modelo del Protocolo.

Capítulo 7, Iteración 4: codificación del soporte para los mensajes que utilizaremos.

Capítulo 8, Iteración 5: Organización de la arquitectura del nodo, partes y servicios que lo forman.

En todos los capítulos se han marcado una descripción de lo que se espera, junto a esta se han dedicado unas pequeñas líneas a los cambios que ha sufrido, si es que lo ha hecho. Además se señalan los diagramas utilizados, y el plan de pruebas unitario.

Tercera Parte

Según la definición del entorno propuesto que veremos en el Capítulo 3, se han desarrollado dos nuevas partes que analizan, diseñan e implementan servicios específicos para los nodos:

1.3 Organización de la memoria

En el Capítulo 9 se detallan los pasos seguidos para instanciar los elementos Broker y FileBroker.

En el Capítulo 10 se recogen los detalles que han hecho posible insertar el framework en la herramienta Together 6.2.

Estos capítulos poseen la misma estructuración que los capítulos de la parte anterior.

Conclusiones y Líneas Futuras

En el Capítulo 11 se recogen las conclusiones del proyecto y se plantean las líneas de trabajo, inmediatas y a largo plazo, abiertas por este.

Apéndices

Se hace alusión a esta parte puesto que no formará parte física de esta memoria. Los distintos apéndices se localizarán en el CD de la documentación. Se ha optado realizarlo así debido al gran volumen de documentación que se ha producido, originado por la naturaleza del proyecto.

Capítulo 2

Plan de proyecto

2.1 Ámbito del proyecto

El presente trabajo se ha desarrollado para la asignatura Sistemas Informáticos de la titulación de Ingeniero en Informática. Comprende al desarrollo de un proyecto completo que servirá como síntesis de las materias de dicha titulación. Ha sido desarrollado dentro del grupo GIRO. Y principalmente une dos proyectos fin de carrera, [*Per03]* y [*NCAP02*] definiendo con el primero una arquitectura, herramientas y características de un framework y con el segundo un entorno de explotación.

Identificación del proyecto

El presente proyecto software, comprende la evolución del entorno de desarrollo definido en [*Per03*], marcando los cambios y nuevas estructuraciones. En el Apéndice C, localizado en el CD adjunto a esta documentación, podemos encontrar la documentación actualizada de la arquitectura del framework obtenido. Al mismo tiempo, se ha desarrollado un entorno de explotación sobre el que hemos instanciado el framework, haciendo uso de sus principales funcionalidades y características. Se espera que en revisiones sucesivas se depure aun más la definición, se optimice la implementación y se extiendan o eliminen funcionalidades según sea necesario, como lo que se ha conseguido con este proyecto.

Visión del proyecto

El objetivo principal de este trabajo es el de crear una infraestructura distribuida utilizando herramientas, nuevas o ya existentes, que nos ayudan en el desarrollo de un entorno real basado en reutilización. Una de las herramientas ya existentes que utilizamos, es el modelo reutilizable MECANO, sobe el que se basan los dos proyectos iniciales. La infraestructura creada es flexible, permitiendo ser integrada en otros entornos, como pueden ser herramientas de Cuarta Generación, o programas de que nos facilitan los procesos de análisis, diseño, implementación, etc. , como puede ser Together 6.2. Al igual que la arquitectura puede ser instanciada en entornos, ésta puede utilizar otras nuevas herramientas para adaptar su funcionamiento a uno nuevo, un ejemplo de esta cualidad es el de poder cambiar el modelo reutilizable y utilizar otro diferente, siempre y cuando se complemente el paquete que trate este nuevo modelo.

2.2 Recursos del proyecto

Se describen a continuación los recursos de los que se dispone para la realización del proyecto.

2.2.1 Recursos humanos

El desarrollo del proyecto software se realizará en solitario, asumiendo en una sola persona todos los roles participantes del mismo. El proyecto principalmente estaba establecido para la elaboración de algo más de un semestre pero finalmente la duración "neta" del proyecto ha sido prácticamente un año. Esta extensión no ha sido por una planificación errónea, sino por la oportunidad de trabajo que obtuve, intentando compaginar la finalización del proyecto con mi periodo laboral.

El horario empleado ha sido de 9:00 a 14:30 y de 16:30 a 21:45 laborables y 9:30 a 13:30 los sábados, desde Agosto del 2004 a Marzo del 2005 mes en el que me tuve que trasladar (Fiestas y Domingos libres). Después de algo mas de un mes de adaptación proseguí con la finalización del proyecto marcándome un horario flexible e inicialmente de 3-4 horas diarias dependiendo de mi jornada laboral. Bajo la herramienta Microsoft Proyect se ha establecido un calendario lo más real posible, marcando los días laborables, no laborables y festivos.

2.2.2 Recursos técnicos: hardware y software

Los recursos hardware con los que se cuenta son los siguientes:

Recursos hardware pertenecientes al alumno:

- Ordenador portátil Intel PentiumIV a 2.8GHz, con 512MB de memoria RAM, disco duro de 40 GB, lector de CDROM y grabadora de CD´s integrados. Sistema operativo instalados: Windows XP profesional. Se utilizará como el equipo principal de trabajo.
- Diversos periféricos de apoyo como: pendrive de 512 Kb con interfaz USB 2.0, para poder trasladar los documentos y ficheros de trabajo entre los diferentes equipos dónde se realizará el proyecto.

Recursos hardware pertenecientes al grupo GIRO:

- Ordenador personal en el laboratorio del grupo GIRO en la Escuela Técnica Superior de Ingeniería Informática de la Universidad de Valladolid, con las siguientes características: Intel PentiumII a 300MHz, 196MB de memoria RAM, disco duro de 4 GB. Sistema operativo instalado: Microsoft Windows 2000. En el que se instalará alguno de los nodos del entorno de explotación.
- Conexión a internet en el laboratorio del grupo GIRO a través de la red local del mismo.

Los recursos software de los que se dispone son:

- Together 6.2. Licencia 15 días de prueba, accediendo a la página oficial http://www.borland.com/downloads/download_together.html. Con la que hemos generado el análisis, diseño e implementación. Así como la instanciación del proyecto.
- J2SE, J2SDK 1.4.2, kit de desarrollo Java de Sun.
- Jdsl 2.1 (Java data structure library), biblioteca de estructuras de datos en Java, libre y de uso gratuito para fines académicos [jds].
- Word 2000, para la generación de la documentación y pdfMachine para la generación de documentación en PDFs .
- Acrobat Reader 5.0 para leer documentos en formato pdf.
- Paint de Windows XP para la elaboración y modificación de gráficos.
- Winrar y Winzip para la generación de ficheros comprimidos.
- Mircrosoft Proyect para la elaboración del plan de proyecto.

2.3 Análisis y gestión de riesgos

2.3.1 Identificación de riesgos

Se recogen en *Tabla 5.1* los riesgos que se han identificado para el desarrollo del proyecto software:

Riesgo	Descripción
Modificación de arquitectura	Debido a que la definición del entorno era bastante teórica, la parte que se ha implementado e instanciado ha sufrido algún cambio, respecto a lo original. Se han dejado partes abiertas y definidas, por ello existe la posibilidad de que al seguir realizando el desarrollo del framework vuelva a surgir la necesidad de volver a modificar la arquitectura del entorno.

Conflictos de funcionalidad	Puede que existan funcionalidades que al introducirlas en la implementación, no puedan coexistir sin ocasionar conflictos o que requieran de alguna solución compleja para poder ser implementadas con garantías.
Funcionalidad ausente	En la etapa de implementación podemos encontrar funcionalidades o características que no hayan sido tenidas en cuenta en la definición del entorno.
Funcionalidad Inviable	Puede que funcionalidades planteadas en la definición, se consideren inviables. Bien porque requieran de una implementación compleja que necesite unos recursos de tiempo de los que no se dispone, bien porque requieran de un consumo de recursos excesivos a la hora de su ejecución en un nodo o bien porque se encuentre en conflicto con otra funcionalidad, como se plantea en el riesgo “Conflictos de funcionalidad”.
Modificación del formato de los mensajes	Existe la posibilidad de que durante el desarrollo se modifique el formato de los mensajes. Puede que sea necesario incluir soporte sintáctico para intercambiar más información necesaria para el procesamiento de los mensajes, o que se modifique la estructura para facilitar su procesamiento.
Tiempo insuficiente para cumplir con los objetivos principales	Debido a la magnitud y evolución del framework así como la obtención de objetivos principales, pues estos últimos obligan a tener planteados, estructuras o funciones, de magnitud considerable, en las que se basan.
Tiempo insuficiente para cumplir con los objetivos secundarios	Debido a la magnitud y evolución del framework así como la obtención de objetivos secundarios, pues estos últimos obligan a tener planteados, estructuras o funciones, de magnitud considerable, en las que se basan.
Acotación del sistema a desarrollar	Debido a la magnitud de este trabajo es costoso establecer una serie de funciones y servicios básicos que se implementen. Se puede entrar en un círculo que haga el desarrollo del framework demasiado extenso. De ahí que se dejen partes abiertas.
Estudio y adaptación de Together 6.2	El proyecto [NCAP03] se desarrollo para la herramienta Together 5.2, puede que alguna de las funciones necesarias para la instalación de nuevos módulos haya cambiado, por lo que habría que profundizar en esta modificación y adaptar aún mas los servicios desarrollados en este proyecto.
Posibilidad de Trabajo	Debido a que únicamente se esta desarrollando el proyecto fin de carrera barajo la opción de empezar a trabajar.

Tabla 5.1: Riesgos identificados en el proyecto software

2.3.2 Incidencia sobre el proyecto

Para calcular la incidencia de los riesgos identificados sobre el proyecto, se han establecido niveles cuantitativos para evaluar cada uno de ellos. Se recogen estos baremos en la siguiente tabla:

Probabilidad	Intervalo	Impacto	Intervalo
Muy baja	[0.0 - 0.2)	Despreciable	[0 - 2)
Baja	[0.2 - 0.4)	Marginal	[2 - 4)
Media	[0.4 - 0.6)	Crítico	[4 - 8)

Alta	[0.6 - 0.8)	Catastrófico	[8 - 10]
Muy alta	[0.8 - 1.0]	-----	-----

Con los baremos de la tabla anterior se calcula la tabla de incidencias de los riesgos:

Impacto / Probabilidad	Despreciable	Marginal	Crítico	Catastrófico
Muy baja	Baja	Baja	Baja	Moderada
Baja	Baja	Baja	Baja	Moderada
Media	Baja	Baja	Moderada	Alta
Alta	Moderada	Moderada	Alta	Alta
Muy alta	Moderada	Moderada	Alta	Alta

Una vez que se tienen los baremos y la tabla de incidencia se puede elaborar la estimación de incidencia de los riesgos identificados en el proyecto, se recoge esta información en la siguiente tabla:

Riesgo	Probabilidad	Impacto	Incidencia
Modificación de la arquitectura	Muy baja (0.1)	Catastrófico (8)	Moderada (0.8)
Conflictos de funcionalidad	Baja (0.3)	Marginal (3)	Baja (0.9)
Funcionalidad ausente	Alta (0.6)	Crítico (6)	Moderada (3.6)
Funcionalidad inviable	Media (0.4)	Marginal (2)	Baja (0.8)
Modificación del formato de los mensajes	Muy Alta (0.9)	Marginal (3)	Moderada (3.6)
Tiempo insuficiente para los objetivos principales	Media (0.5)	Catastrófico (8)	Alta (4.0)
Tiempo insuficiente para los objetivos secundarios	Alta (0.6)	Marginal(3)	Moderada (2.4)
Estudio y adaptación de Together 6.2	Alta (0.7)	Crítico(6)	Moderada(4.2)
Posibilidad de Trabajo	Alta (0.7)	Marginal (2)	Moderada(1.4)

Se plantean a continuación las medidas a tomar para reducir la incidencia de los riesgos. El tiempo ha sido acotado a un año, por lo que este recurso nos dará mucho juego para poder tomar las medidas necesaria y conseguir reducir la incidencia de los riesgos en los casos en que esta es alta o moderada.

2.3.3 Medidas para la reducción y gestión de riesgos

Riesgo	Estrategias para reducir la **probabilidad** de aparición
Modificación de la arquitectura	Partimos de una arquitectura teórica propuesta en el proyecto anterior, en el se ha considerado que ésta es bastante estable, por eso la probabilidad de aparición de este riesgo es tan baja. No se pueden tomar medidas concretas para reducir esta probabilidad.
Conflictos de funcionalidad	Comprobar las funcionalidades definidas, dividirlas en subfuncionalidades y clasificarlas según éstas, localizando así las incompatibilidades.
Funcionalidad ausente	Establecer un proceso de desarrollo iterativo y revisar al comienzo de cada iteración la definición del entorno. Se debe prestar especial atención a las funcionalidades a implementar en dicha iteración y a las interacciones de estas con las ya implementadas. Esto debe conducir a localizar la funcionalidad ausente y a incluirla en la definición antes de que se produzca el riesgo durante la implementación.
Funcionalidad inviable	Revisar la definición del entorno para descartar la funcionalidad inviable antes de comenzar el desarrollo.
Modificación del formato de los mensajes	Declarar como 'cerrado' el formato de los mensajes, y adaptar el desarrollo al formato.
Tiempo insuficiente para los objetivos principales	Centrar el desarrollo exclusivamente en los objetivos principales y no pasar a desarrollar los secundarios hasta haber completado estos.
Tiempo insuficiente para los objetivos secundarios	Aumentar el número de horas de trabajo asignadas al desarrollo de los objetivos secundarios o reorganizar las tareas en las etapas finales del desarrollo.
Estudio y adaptación de Together 6.2	Al igual que este framework, las herramientas evolucionan alterando su funcionalidad y formas de ejecución. No se pueden tomar medidas más concretas para reducir esta probabilidad.
Posibilidad de Trabajo	No se pueden tomar medidas más concretas para reducir esta probabilidad.

Riesgo	Estrategias para reducir el nivel de **impacto**
Modificación de la arquitectura	Conservar las funcionalidades desarrolladas y reorganizarlas de forma que puedan ser empleadas en la nueva arquitectura. Reutilizar las decisiones y pautas tomadas para la elaboración de las diferentes partes del entorno durante la definición del mismo.
Conflictos de funcionalidad	Establecer prioridades entre las diferentes funcionalidades. En caso de conflicto se deberán descartar funcionalidades hasta que este desaparezca, utilizando como mecanismo de decisión las prioridades adjudicadas a cada una.

Funcionalidad ausente	Elaborar un diseño del framework en el que la arquitectura sea estable y constituya el núcleo del nodo. Las nuevas funcionalidades necesarias que surjan podrán ser incluidas con relativa facilidad al framework.
Funcionalidad inviable	En el caso de localizar funcionalidad inviable, se descartará en el desarrollo actual indicando los motivos. No se deshecha la opción de que pueda retomarse en revisiones posteriores del framework.
Modificación del formato de los mensajes	En las partes que realizan el procesamiento de los mensajes, realizar un diseño centrado en las estructuras sintácticas más estables de los mensajes. En todo caso, realizar una representación de los datos transportados por los mensajes, independiente de este formato.
Tiempo insuficiente para los objetivos principales	Dar prioridad en el desarrollo del framework a las funcionalidades más fundamentales y a la arquitectura y al diseño de los nodos. Elaborar un diseño en el que la arquitectura sea estable y constituya el núcleo del nodo. Las funciones se considerarán extensiones accesibles de formas comunes a través del núcleo del nodo. Elaborar plantillas y guías para el desarrollo e instanciación de nueva funcionalidad.
Tiempo insuficiente para los objetivos secundarios	Las medidas a tomar son las mismas que en el caso de los objetivos principales.
Estudio y adaptación de Together 6.2	Se instalará el proyecto [NCAP03] y ver si funciona, si no es así, habrá que estudiar los cambios que ha sufrido y la nueva manera de incorporar nuevos módulos, mediante algún tutorial o manual.
Posibilidad de Trabajo	Se alargará el desarrollo del proyecto, intentando compaginar ambas actividades.

2.4 Estimaciones temporales

Se recoge a continuación, la descomposición del proyecto en tareas y la distribución de las mismas a lo largo del tiempo disponible.

2.4.1 Fechas de comienzo y finalización del proyecto

La fecha límite de entrega del proyecto es el 9 de Septiembre, así que se dejan esos cinco primeros días de Septiembre para la preparación de la documentación, impresión y encuadernación de la misma, así como para la elaboración del material a entregar en formato digital. Se reservarán los últimos días de Agosto para revisar y finalizar la documentación a presentar, por lo que se considerará como fecha límite para la finalización del proyecto, el día 28 de Agosto. Se considerará como fecha de comienzo del proyecto, el 15 de Agosto de 2004, aunque los proyectos fueron entregados unos mes antes. Por lo que se dispone de casi doce meses para el desarrollo del proyecto software, aunque inicialmente se pretende conseguir en un semestre debido algún riesgo, este se podría alargar.

2.4.2 Lista de tareas

Se describen a continuación detalladas las diferentes tareas y subtareas que constituirán el desarrollo:

Planificación

Tarea	Descripción
Ámbito, objetivos y recursos	Elaboración de los documentos de ámbito, objetivos y recursos como parte de la documentación que constituye el plan de proyecto. Y que nos sirven de acotación en el diseño e implementación.
Plan de riesgos	Identificar y analizar los riesgos del proyecto software y elaborar planes de prevención y de contingencia.
Estimaciones temporales	Realizar la distribución de los recursos disponibles para las diferentes tareas que componen el proyecto y elaborar las estimaciones temporales para la consecución de las tareas.

Estudio de la arquitectura

Tarea	Descripción
Análisis de requisitos de la arquitectura	Analizar los requisitos del proyecto relacionados con el diseño de la arquitectura del framework.
Diseño de la arquitectura	Realizar un primer acercamiento a la arquitectura general del framework.

Estudio de los módulos

Tarea	Descripción
Análisis de requisitos de los módulos del framework	Analizar los requisitos que sirvan para definir la parte del framework que se desarrolló en cada una de las iteraciones.
Diseño de los módulos del framework	Examinar el diseño de la parte del framework de la iteración en la que nos encontremos desde el punto de vista estático y dinámico.

Ampliación del framework y Diseño del entorno real

Tarea	Descripción
Análisis de los requisitos de los nuevos módulos	Recoger y analizar los requisitos que sirvan para definir la parte del framework que se desarrollará durante esa iteración.
Diseño de los nuevos módulos	Realizar el diseño de la parte del framework que se esté desarrollando en la iteración en la que nos encontremos, desde el punto de vista está tico y dinámico.

2.4 Estimaciones temporales

Análisis de la integración con proyecto [*NCAP02*]	Recoger y analizar los requisitos que sirvan para definir la integración con la herramienta Together 6.2 y la posibilidad de inserción en el repositorio de GIRO.
Diseño de la integración con [*NCAP02*]	Realizar el diseño de la parte que se integrará con Together 6.2 y el repositorio de GIRO, desarrollando en la iteración actual y desde el punto de vista estático y dinámico.
Revisión del diseño para integración	Revisar el diseño y realizar las modificaciones que sean necesarias para integrar las partes desarrolladas en una iteración con las desarrolladas en las iteraciones anteriores.

Codificación

Tarea	Descripción
Implementación de los módulos internos	Realizar la codificación de la parte que se esté desarrollando durante la iteración actual.
Implementación del entorno de explotación	Realizar la codificación, según su análisis y diseño, de las herramientas que nos permitan integrar el framework junto con Together 6.2 y el repositorio de GIRO.

Pruebas

Tarea	Descripción
Pruebas de unidades	Diseñar, implementar y ejecutar las pruebas unitarias de las partes que se hayan desarrollado en la iteración actual.
Pruebas de integración	Diseñar, implementar y ejecutar las pruebas de integración en las que intervengan, todas las partes desarrolladas hasta la iteración actual.

Instalación de los componentes

Tarea	Descripción
Manual de instalación	Fácil y rápido manual con los pasos que han de seguirse para instanciar el framework.
Instalación física en los nodos	Instalación de todos los componentes desarrollados para crear la estructura real con la que veremos el framework instanciado y funcionando.

Documentación framework

Tarea	Descripción
Guía de instanciación	Elaborar una guía básica de instanciación para documentar cómo debe ser utilizado el framework.
Modelo de características	Plasmado en el diccionario de Clases con el que tenemos una especificación de los atributos y métodos de cada clase.

2.4.3 Planificación temporal

Lo primero que haremos es distribuir el tiempo del que inicialmente disponibles entre cada una de las tareas principales que hemos localizado. A continuación se señala esta distribución de tiempo según las tareas y el porcentaje de tiempo empleado.

Distribución del tiempo

Para la asignación de esfuerzos a las diferentes tareas, se ha dividido el tiempo del proyecto de la siguiente forma:

Porcentaje	Tarea
23%	Planificación y estudio
41%	Codificación y pruebas.
17%	Análisis y diseño de los nuevos módulos
19%	documentación del framework

La distribución del tiempo desglosada, quedaría de la siguiente manera

Porcentaje	Tarea	Días
2%	Planificación	3
9%	Estudio de la arquitectura	12
12%	Estudio de los módulos	17
17%	Ampliación del framework y Diseño del entorno real	24
32%	Codificación	45
9%	Pruebas	13
3%	Instalación de los componentes	4
16%	Documentación framework	22

Merece la pena comentar algunos valores:

- El porcentaje de Planificación es tan pequeño debido a que se conoce con exactitud que se requiere de este proyecto, otro tema es el cómo, del que se encarga las siguientes tareas.
- No podemos desarrollar si no conocemos lo existente, debemos entender cuales son las características, herramientas, servicios, limitaciones, etc., que posee, de ahí que haya un tiempo considerable en el estudio del framework, en total un 21% del tiempo del que disponemos.
- Una vez que tenemos claro lo que el proyecto tiene que hacer, y las herramientas de las que disponemos, así como funcionalidades ya establecidas, el siguiente paso es la creación y adaptación a lo ya existente de nuevos servicios, ampliando la funcionalidad del framework. Este nuevo framework se pondrá bajo prueba, en un entorno de explotación

real. Estos dos hechos es el grueso del proyecto de ahí que obtengamos el 58% del tiempo, desglosado sería: 17%+32%+9%, Análisis y Diseño, Codificación y Pruebas.

- El último paso, es dejar instalado y documentado todo el estudio que hemos realizado, señalando la evolución del framework y la instalación del entorno de explotación obtenido sobre el que se han realizado pruebas. Es importante para futuras ampliaciones que estas tareas sean realizadas con atención, de ahí su tiempo invertido.

Planificación de iteraciones según las partes del proyecto

Se planteará el proyecto software como el desarrollo sucesivo de las diferentes partes que componen el framework. El desarrollo de cada una de estas partes constituirá una iteración, incluyendo cada una de las tareas de análisis, requisitos, diseño, codificación y pruebas unitarias. En las iteraciones que corresponda, se incluirán también tareas de pruebas de integración.

Iteración	Parte a desarrollar	Descripción	Días
0	Arquitectura	Se diseñará la arquitectura general del framework, definiendo en forma de paquetes las partes principales que lo integrarán.	0
1	Utilidades	Se diseñarán una serie de componentes que al modo de utilidades serán utilizados en otras partes del framework. Se incluyen en un paquete aparte bien por tratarse de componentes que serán utilizados en prácticamente todas las demás partes del framework, o bien por tratarse de componentes que aunque sólo se empleen en una parte, no estén estrictamente relacionados con las funciones de esa parte. Otra razón para su inclusión en un paquete de utilidades es que esto favorezca su reutilización en otros desarrollos que no tengan nada que ver con este trabajo.	14
2	Comunicación	Se desarrollarán los módulos responsables de los canales de comunicación de los nodos con el exterior a través de la red. Los módulos ofrecerán las funcionalidades necesarias para poder crear canales de comunicación entrantes, por los que recibir las peticiones de otros nodos, y para poder crear canales de comunicación salientes, por los que poder enviar mensajes a otros nodos.	10

	Modelos	Modelos Se desarrollarán las estructuras de datos necesarias para poder representar los datos de tipo 'modelo' como grafos. Estas tienen que ofrecer un interfaz para crear y acceder a los modelos como si se tratase de grafos. Los modelos y los elementos pertenecientes a los modelos también deben 'entender' y generar su representación XML. Se desarrollarán las estructuras de forma que se soporten los dos tipos de modelo que define el entorno de desarrollo: el modelo de datos de gestión de protocolo y el modelo de componentes reutilizables. El modelo de componentes reutilizables debe ser desarrollado de forma que se permita utilizar diferentes modelos concretos, aunque en este caso se utilice el modelo de MECANO.	14
4	Datos de mensajes	Se desarrollarán las estructuras de datos necesarias para poder representar los elementos que constituyen la estructura de los mensajes. Estas estructuras deben poder ser creadas y manipuladas con facilidad, y deben 'entender' y generar su representación XML.	16
5	Núcleo de los nodos	Se desarrollará el núcleo de los nodos, como un conjunto de clases de control que se encargan de establecer y gestionar las diferentes fases que constituyen el procesamiento de los mensajes.	22
6	Servicios	Se desarrollarán las partes genéricas (no dependientes de una herramienta concreta), del conjunto de servicios obligatorios que se han definido para los diferentes tipos de nodos, además de aquellos que se han definido como comunes.	17
7	Adaptación en Together 6.2	Se desarrollaran las partes específicas, para la herramienta Together 6.2, del conjunto de servicios obligatorios y que nos servirán para interactuar con el usuario eligiendo servicios propios de la arquitectura framework.	12

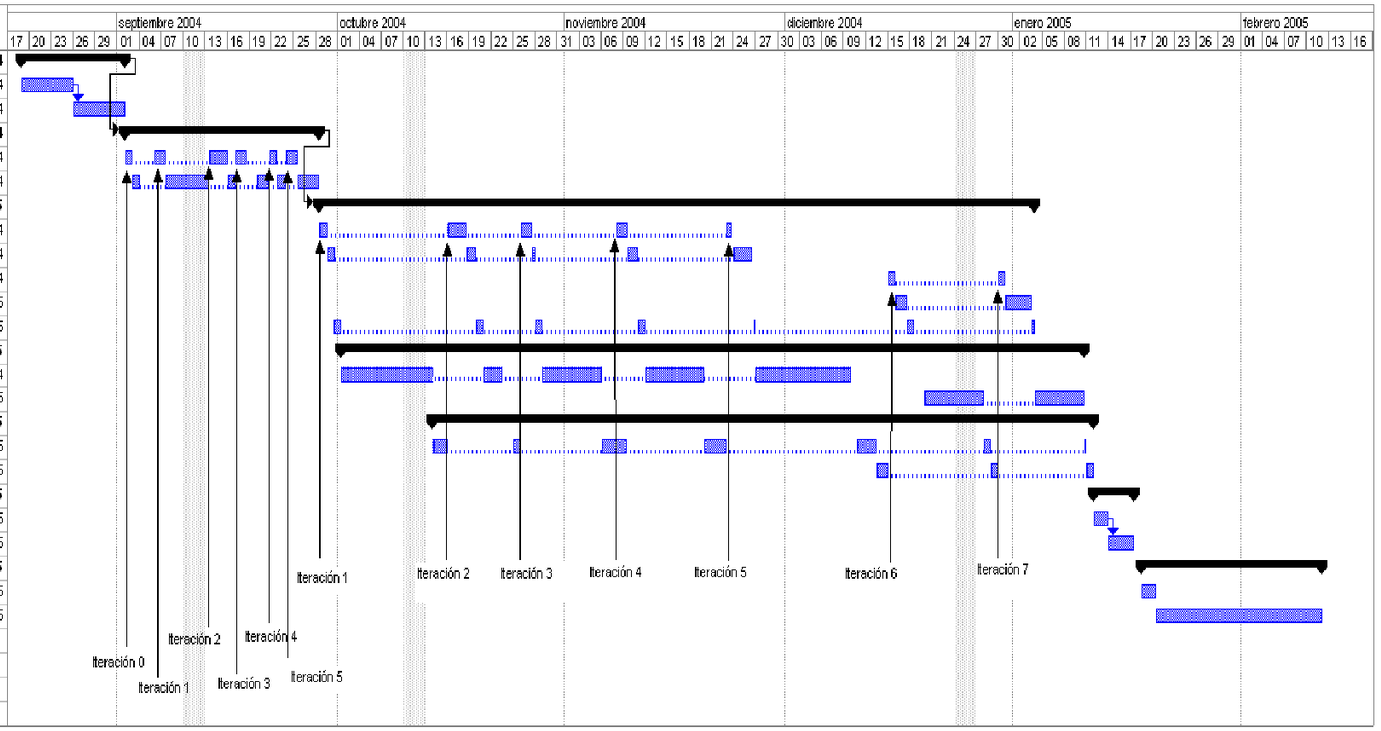
(En este ajuste de tiempos están incluidos las fiestas y domingos)

Planificación temporal de las tareas

Se muestran a continuación las estimaciones temporales para las diferentes tareas. La descomposición de tareas se ha realizado identificando aquellas que se repiten en las iteraciones del desarrollo de cada una de las partes del framework. Debido a esto, la tabla de estimaciones de tiempo por tareas, no aporta información acerca de la duración y fechas de comienzo y finalización de las diferentes iteraciones. Para consultar estos datos, será más apropiado utilizar el diagrama de Gantt.

2.4 Estimaciones temporales

Id	Nombre de tarea	Comienzo	Fin
5	**Estudio de la arquitectura**	**mié 18/08/04**	**mié 01/09/04**
6	Análisis de requisitos de la arquitectura	mié 18/08/04	mié 25/08/04
7	Diseño de la arquitectura	mié 25/08/04	mié 01/09/04
8	**Estudio de los módulos**	**jue 02/09/04**	**mar 28/09/04**
9	Análisis de requisitos de los módulos del framework	jue 02/09/04	sáb 25/09/04
10	Diseño de los módulos del framework	vie 03/09/04	mar 28/09/04
11	**Ampliación del framework y Diseño del entorno real**	**mar 28/09/04**	**lun 03/01/05**
12	Análisis de los requisitos de los nuevos módulos	mar 28/09/04	mar 23/11/04
13	Diseño de los nuevos módulos	mié 29/09/04	vie 26/11/04
14	Análisis de la integración con proyecto [NCAP02]	mié 15/12/04	jue 30/12/04
15	Diseño de la integración con [NCAP02]	jue 16/12/04	lun 03/01/05
16	Revisión del diseño para integración	jue 30/09/04	lun 03/01/05
17	**Codificación**	**vie 01/10/04**	**lun 10/01/05**
18	Implementación de los módulos internos	vie 01/10/04	jue 09/12/04
19	Implementación del entorno de explotación	lun 20/12/04	lun 10/01/05
20	**Pruebas**	**jue 14/10/04**	**mar 11/01/05**
21	Pruebas de unidades	jue 14/10/04	lun 10/01/05
22	Pruebas de integración	lun 13/12/04	mar 11/01/05
23	**Instalación de los componentes**	**mié 12/01/05**	**lun 17/01/05**
24	Manual de instalación	mié 12/01/05	jue 13/01/05
25	Instalación física en los nodos	jue 13/01/05	lun 17/01/05
26	**Documentación del framework**	**mar 18/01/05**	**vie 11/02/05**
27	Guía de instanciación	mar 18/01/05	jue 20/01/05
28	Modelo de características	jue 20/01/05	vie 11/02/05



2.4 Estimaciones temporales

	Nombre de tarea	Comienzo	Fin
5	**Estudio de la arquitectura**	**mié 18/08/04**	**mié 01/09/04**
6	Análisis de requisitos de la arquitectura	mié 18/08/04	mié 25/08/04
7	Diseño de la arquitectura	mié 25/08/04	mié 01/09/04
8	**Estudio de los módulos**	**jue 02/09/04**	**mar 28/09/04**
9	Análisis de requisitos de los módulos del framework	jue 02/09/04	sáb 25/09/04
10	Diseño de los módulos del framework	vie 03/09/04	mar 28/09/04
11	**Ampliación del framework y Diseño del entorno real**	**mar 28/09/04**	**lun 03/01/05**
12	Análisis de los requisitos de los nuevos módulos	mar 28/09/04	mar 23/11/04
13	Diseño de los nuevos módulos	mié 29/09/04	vie 26/11/04
14	Análisis de la integración con proyecto [NCAP02]	mié 15/12/04	jue 30/12/04
15	Diseño de la integración con [NCAP02]	jue 16/12/04	lun 03/01/05
16	Revisión del diseño para integración	jue 30/09/04	lun 03/01/05
17	**Codificación**	**vie 01/10/04**	**lun 10/01/05**
18	Implementación de los módulos internos	vie 01/10/04	jue 09/12/04
19	Implementación del entorno de explotación	lun 20/12/04	lun 10/01/05
20	**Pruebas**	**jue 14/10/04**	**mar 11/01/05**
21	Pruebas de unidades	jue 14/10/04	lun 10/01/05
22	Pruebas de integración	lun 13/12/04	mar 11/01/05
23	**Instalación de los componentes**	**mié 12/01/05**	**lun 17/01/05**
24	Manual de instalación	mié 12/01/05	jue 13/01/05
25	Instalación física en los nodos	jue 13/01/05	lun 17/01/05
26	**Documentación del framework**	**mar 18/01/05**	**vie 11/02/05**
27	Guía de instanciación	mar 18/01/05	jue 20/01/05
28	Modelo de características	jue 20/01/05	vie 11/02/05

2.5 Seguimiento del proyecto

Finalmente, las estimaciones iniciales han resultado ser optimistas y el desarrollo del proyecto se ha alargado 62 días laborables aunque hay que señalar que las horas establecidas a partir del mes de Mayo son 4, los día laborables utilizados son 4,5. Es significativo observar, que muchas de las iteraciones han visto prolongado su tiempo de desarrollo.

No se ha sabido identificar la magnitud real del desarrollo al comienzo del mismo y aunque en el tiempo estaba muy bien planificado, no se han podido cumplir los plazos previstos pues surgieron cuatro grandes problemas:

1. A partir del día 7 de Mayo estoy trabajando en Madrid para una gran empresa de software, el echar horas extras y la presión del trabajo es algo común y a influido en el desarrollo del proyecto, como se tuvo presente en los riesgos.
2. La resolución de ciertos problemas que han surgido. De forma teórica estaban solucionados pero en la práctica chocaban con funciones y objetivos anteriormente previstos. En la Parte I de este proyecto podremos ver con mas profundidad los cambios que se han realizado.
3. La adaptación de los proyectos iniciales, a alargado el tiempo. Primero con la evolución del framework y posteriormente con el de *Mecanos Representando Proyectos Together: Inserción automática en el repositorio de GIRO*, optando por respetar al máximo posible el análisis de ambos proyectos.
4. La documentación se ha extendido más de lo previsto, pues se ha intentando hacer un documento que marque por un lado la evolución y por otro nos sirva de base en futuras evoluciones del framework, y aunque se tenía presente que iba a ser un proceso laborioso, ha sido aún mas de lo planificado.

En consecuencia, se han conseguido cumplir los objetivos principales y los secundarios, dejando abiertos aspectos pertenecientes a los dos tipos. Peculiaridad marcada en la definición de los objetivos.

2.5.1 Riesgos aparecidos en el desarrollo del proyecto

Durante el desarrollo, han aparecido algunos de los riesgos que se habían identificado y que más incidencia tenían sobre el proyecto, retrasando éste considerablemente, además apareció otro que no se había tenido en cuenta, lo cual aún retraso más el proyecto.

Por ello el tiempo de desarrollo planificado, ha sido insuficiente y para intentar conseguir los objetivos se alargó la planificación. Los motivos de que han hecho fallar la planificación inicial son:

1.- Estoy trabajando en Madrid.

2.- Errores en la formación de la DTD. Tanto errores sintácticos como errores “semánticos”.

3.- Estructuración inacabada en alguno de los modelos (Principalmente Iteración 3). Con lo que ha habido que alterar los modelos de clases.

4.- Cambios en la formación del mensaje. Y que hicieron mella en la DTD.

5.- El riesgo no identificado y que más problemas ha dado ha sido el de la redefinición de los métodos y clases en la mayoría de las iteraciones por no encontrarse éstos implementados del todo o por chocar querían conseguir.

6.- La documentación paso a formar dos fases, una que es la revisión del proyecto anterior propiamente dicho, y otra en la que se elaboró esta.

La solución principal y la que nos ha permitido conseguir los objetivos ha sido la de alargar el tiempo de planificación del proyecto.

2.5.2 Distribución real de tiempo por tareas

La distribución de esfuerzos y de tiempo es diferente de la planificada en un principio. Por un lado se redujo el esfuerzo de análisis de requisitos, ya que debido al trabajo previo al desarrollo, consistente en la definición del entorno, el conocimiento del dominio del problema y de los requisitos era muy grande. Por otro lado, las tareas de diseño y codificación se han prolongado bastante más de lo esperado, por los problemas ya mencionados.

Porcentaje	Tarea	Días
1%	Planificación	3
8%	Estudio de la arquitectura	17
11%	Estudio de los módulos	23
15%	Ampliación del framework y Diseño del entorno real	31
28%	Codificación	54
15%	Pruebas	30
2%	Instalación de los componentes	4
20%	Documentación framework	40

Como se puede observar, el mayor incremento de tiempo, se dio en las tareas de implementación aunque las tareas que se han visto alteradas por los riesgos que hemos encontrado han sido las pruebas y la documentación, pues los cambios que se han realizado se han tenido que probar y documentar.

2.5.3 Distribución real de tiempo por iteraciones

Se muestra a continuación la duración real de las iteraciones en que se ha dividido el desarrollo.

Las últimas dos iteraciones no pudieron ser finalizadas en su totalidad.

Iteración	Parte a desarrollar	Días Estimados	Días Empleados
1	Utilidades	14	13
2	Comunicación	10	8
3	Modelos	14	26
4	Datos de mensajes	16	36
5	Núcleo de los nodos	22	40
6	Servicios	17	11
7	Adaptación en Together 6.2	12	8

Como vemos el tiempo invertido en los nuevos servicios y en la adaptación con Together ha sido menor que el planificado. Esto es debido a que se han reducido el número de servicios que se

pretendían realizar. Se a adaptado lo establecido en el proyecto [NCAP03] pero con un único servicio y dejando para líneas futuras el resto,.

2.5.4 Planificación Real

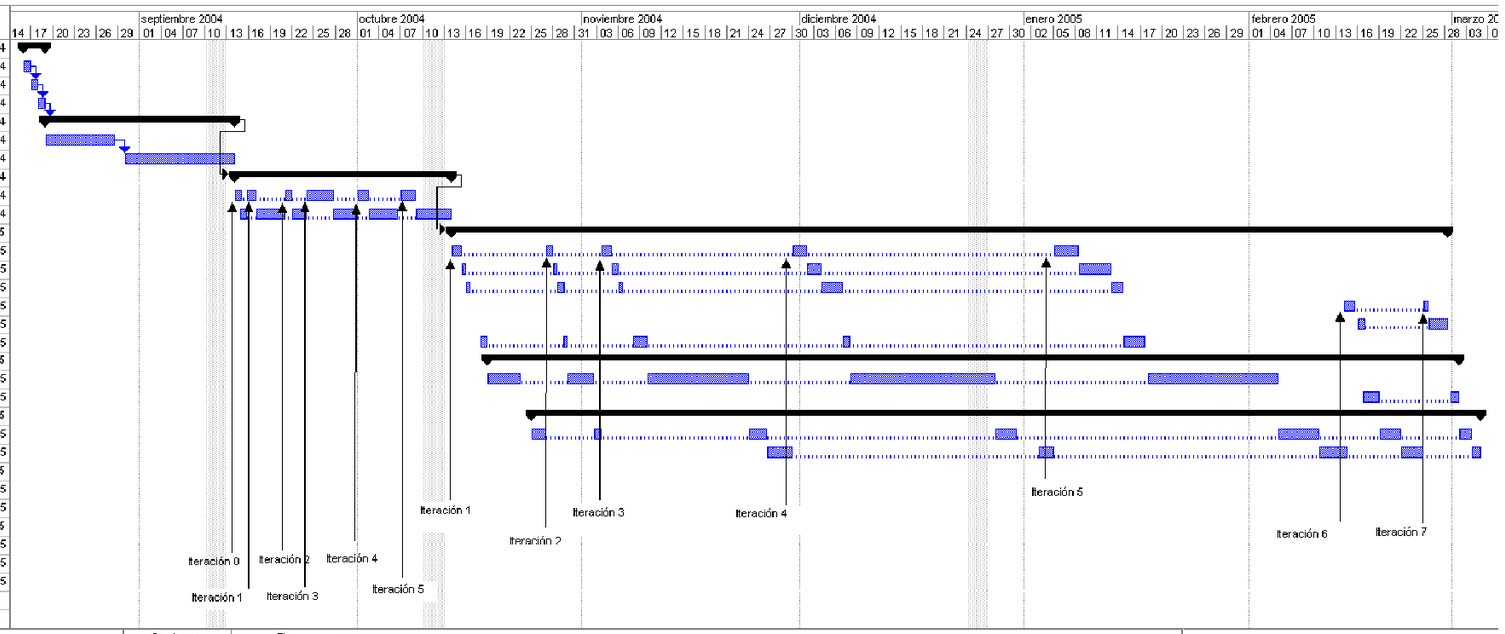
Se muestran a continuación las tablas y gráficos finales con la duración y las fechas de comienzo y finalización, de tareas e iteraciones:

Gantt de seguimiento

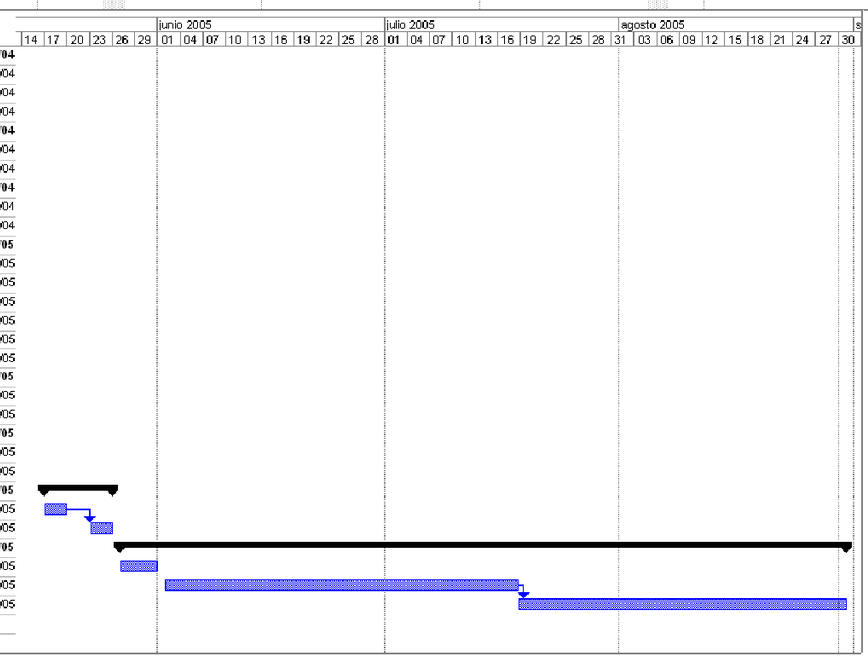
	Nombre de tarea	Comienzo	Fin
1	**Planificación**	**lun 16/08/04**	**mié 18/08/04**
2	Ámbito, objetivos y recursos	lun 16/08/04	lun 16/08/04
3	Plan de riesgos	lun 16/08/04	mar 17/08/04
4	Estimaciones temporales	mar 17/08/04	mié 18/08/04
5	**Estudio de la arquitectura**	**mié 18/08/04**	**lun 13/09/04**
6	Análisis de requisitos de la arquitectura	mié 18/08/04	sáb 28/08/04
7	Diseño de la arquitectura	lun 30/08/04	lun 13/09/04
8	**Estudio de los módulos**	**mar 14/09/04**	**mié 13/10/04**
9	Análisis de requisitos de los módulos del framework	mar 14/09/04	vie 08/10/04
10	Diseño de los módulos del framework	mar 14/09/04	mié 13/10/04
11	**Ampliación del framework y Diseño del entorno real**	**jue 14/10/04**	**lun 28/02/05**
12	Análisis de los requisitos de los nuevos módulos	jue 14/10/04	sáb 08/01/05
13	Diseño de los nuevos módulos	vie 15/10/04	mié 12/01/05
14	Rediseño de los módulos existentes	sáb 16/10/04	vie 14/01/05
15	Análisis de la integración con proyecto [NCAP02]	lun 14/02/05	vie 25/02/05
16	Diseño de la integración con [NCAP02]	mar 15/02/05	lun 28/02/05
17	Revisión del diseño para integración	lun 18/10/04	lun 17/01/05
18	**Codificación**	**mar 19/10/04**	**mar 01/03/05**
19	Implementación de los módulos	mar 19/10/04	vie 04/02/05
20	Implementación del entorno de explotación	mié 16/02/05	mar 01/03/05
21	**Pruebas**	**lun 25/10/04**	**mié 02/03/05**
22	Pruebas de unidades	lun 25/10/04	mar 01/03/05
23	Pruebas de integración	vie 26/11/04	mié 02/03/05
24	**Instalación de los componentes**	**lun 16/05/05**	**mié 25/05/05**
25	Manual de instalación	lun 16/05/05	jue 19/05/05
26	Instalación física en los nodos	dom 22/05/05	mié 25/05/05
27	**Documentación del framework**	**jue 26/05/05**	**mar 30/08/05**
28	Guía de instanciación	jue 26/05/05	mar 31/05/05
29	Revisión de la arquitectura y documentación	mié 01/06/05	lun 18/07/05
30	Modelo de características	lun 18/07/05	mar 30/08/05

2.5 Seguimiento del proyecto

Id	Nombre de tarea	Comienzo	Fin
1	**Planificación**	**lun 16/08/04**	**mié 18/08/04**
2	Ámbito, objetivos y recursos	lun 16/08/04	lun 16/08/04
3	Plan de riesgos	lun 16/08/04	mar 17/08/04
4	Estimaciones temporales	mar 17/08/04	mié 18/08/04
5	**Estudio de la arquitectura**	**mié 18/08/04**	**lun 13/09/04**
6	Análisis de requisitos de la arquitectura	mié 18/08/04	sáb 28/08/04
7	Diseño de la arquitectura	lun 30/08/04	lun 13/09/04
8	**Estudio de los módulos**	**mar 14/09/04**	**mié 13/10/04**
9	Análisis de requisitos de los módulos del framework	mar 14/09/04	vie 08/10/04
10	Diseño de los módulos del framework	mar 14/09/04	mié 13/10/04
11	**Ampliación del framework y Diseño del entorno real**	**jue 14/10/04**	**lun 28/02/05**
12	Análisis de los requisitos de los nuevos módulos	jue 14/10/04	sáb 08/01/05
13	Diseño de los nuevos módulos	vie 15/10/04	mié 12/01/05
14	Rediseño de los módulos existentes	sáb 16/10/04	vie 14/01/05
15	Análisis de la integración con proyecto [NCAP02]	lun 14/02/05	vie 25/02/05
16	Diseño de la integración con [NCAP02]	mar 15/02/05	lun 28/02/05
17	Revisión del diseño para integración	lun 18/10/04	lun 17/01/05
18	**Codificación**	**mar 19/10/04**	**mar 01/03/05**
19	Implementación de los módulos	mar 19/10/04	vie 04/02/05
20	Implementación del entorno de explotación	mié 16/02/05	mar 01/03/05
21	**Pruebas**	**lun 25/10/04**	**vie 04/03/05**
22	Pruebas de unidades	lun 25/10/04	jue 03/03/05
23	Pruebas de integración	vie 26/11/04	vie 04/03/05
24	**Instalación de los componentes**	**lun 16/05/05**	**mié 25/05/05**
25	Manual de instalación	lun 16/05/05	jue 19/05/05
26	Instalación física en los nodos	dom 22/05/05	mié 25/05/05
27	**Documentación del framework**	**jue 26/05/05**	**mar 30/08/05**
28	Guía de instanciación	jue 26/05/05	mar 31/05/05
29	Revisión de la arquitectura y documentación	mié 01/06/05	lun 18/07/05
30	Modelo de características	lun 18/07/05	mar 30/08/05



Id	Nombre de tarea	Comienzo	Fin
1	**Planificación**	**lun 16/08/04**	**mié 18/08/04**
2	Ámbito, objetivos y recursos	lun 16/08/04	lun 16/08/04
3	Plan de riesgos	lun 16/08/04	mar 17/08/04
4	Estimaciones temporales	mar 17/08/04	mié 18/08/04
5	**Estudio de la arquitectura**	**mié 18/08/04**	**lun 13/09/04**
6	Análisis de requisitos de la arquitectura	mié 18/08/04	sáb 28/08/04
7	Diseño de la arquitectura	lun 30/08/04	lun 13/09/04
8	**Estudio de los módulos**	**mar 14/09/04**	**mié 13/10/04**
9	Análisis de requisitos de los módulos del framework	mar 14/09/04	vie 08/10/04
10	Diseño de los módulos del framework	mar 14/09/04	mié 13/10/04
11	**Ampliación del framework y Diseño del entorno real**	**jue 14/10/04**	**lun 28/02/05**
12	Análisis de los requisitos de los nuevos módulos	jue 14/10/04	sáb 08/01/05
13	Diseño de los nuevos módulos	vie 15/10/04	mié 12/01/05
14	Rediseño de los módulos existentes	sáb 16/10/04	vie 14/01/05
15	Análisis de la integración con proyecto [NCAP02]	lun 14/02/05	vie 25/02/05
16	Diseño de la integración con [NCAP02]	mar 15/02/05	lun 28/02/05
17	Revisión del diseño para integración	lun 18/10/04	lun 17/01/05
18	**Codificación**	**mar 19/10/04**	**mar 01/03/05**
19	Implementación de los módulos	mar 19/10/04	vie 04/02/05
20	Implementación del entorno de explotación	mié 16/02/05	mar 01/03/05
21	**Pruebas**	**lun 25/10/04**	**vie 04/03/05**
22	Pruebas de unidades	lun 25/10/04	jue 03/03/05
23	Pruebas de integración	vie 26/11/04	vie 04/03/05
24	**Instalación de los componentes**	**lun 16/05/05**	**mié 25/05/05**
25	Manual de instalación	lun 16/05/05	jue 19/05/05
26	Instalación física en los nodos	dom 22/05/05	mié 25/05/05
27	**Documentación del framework**	**jue 26/05/05**	**mar 30/08/05**
28	Guía de instanciación	jue 26/05/05	mar 31/05/05
29	Revisión de la arquitectura y documentación	mié 01/06/05	lun 18/07/05
30	Modelo de características	lun 18/07/05	mar 30/08/05



2.6 Plan de pruebas

Algunas de las pruebas, que se han realizado a lo largo de todo el desarrollo del proyecto, se localizan en la finalización de cada iteración de desarrollo, comprobando que los métodos que íbamos obteniendo eran correctos y fiables. Se han ejecutado las pruebas anotando los resultados esperados y obtenidos en cada caso. Los resultados de estas pruebas, se han recogido en tablas de registro. Esta documentación aparecerá al final de la documentación de cada iteración correspondiente.

Aunque el desarrollo inicialmente es de un framework y no de una aplicación se platearon las pruebas, para las primeras iteraciones, de manera unitaria. Una vez que el framework ha estado desarrollado y apoyándose en las pruebas unitarias de cada iteración, se han hecho pruebas de integración de partes, así como de la instanciación del framework, todas ellas bajo condiciones reales y en la herramienta en la que se ha instanciado el framework.

Se describen a continuación los campos que aparecen en las tablas de registro de las pruebas unitarias:

Prueba	Nombre de la prueba
Clase ejecutable	Clase que contiene el método main de la prueba, y que debe ejecutarse para realizar la prueba.
Clases implicadas	Clases sobre las que se realiza la prueba, o que intervienen de algún modo en la misma.
Procedimiento	Descripción del procedimiento de la prueba.
Resultado deseado	Resultados que se espera que se obtengan de la prueba cuando el funcionamiento de las clases implicadas es correcto.
Resultado obtenido	Resultado real que se obtiene tras ejecutar la prueba.
Errores encontrados	Defectos encontrados en las clases, que han causado la obtención de unos resultados diferentes de los esperados y que deben ser subsanados.

Las pruebas integrales implican la instanciación de varias clases, analizando la cohesión y el acoplamiento. Par no alargar aún más esta documentación, estas pruebas se resumen en el entorno de explotación, ya que sin ellas, la instancia del proyecto sería una utopía.

Parte I

Definición de un entorno de desarrollo distribuido basado en MECANOS

Capítulo 3

Arquitectura del entorno de desarrollo

3.1 Introducción al entorno

El entorno de desarrollo distribuido basado en reutilización, se compone de una serie de entidades genéricas, que denominamos Nodos, capaces de enviar y recibir mensajes, procesar y generar peticiones de diversos tipos, dependiendo del tipo de actuación que se requiera. Para la coordinación de estos entes se define un protocolo de entorno, con el que podremos transmitir estos mensajes que solicitarán determinados servicios o funciones sobre objetos pertenecientes al modelo reutilizable que empleemos.

3.1.1. Tipos de los nodos

Como ya hemos dicho los nodos se comunican entre ellos, para intercambiar objetos del modelo reutilizable sobre los que ejecutaremos servicios. El entorno define cuatro tipos de nodos sintetizados de las tareas que se necesitan realizan en un entorno de desarrollo *para y con reutilización* y de las herramientas necesarias para el:

Tool, Herramientas interactivas utilizadas por el usuario. En la mayoría de los casos se tratará de herramientas CASE con funcionalidades específicas para reutilización utilizando un modelo de componentes reutilizables. Para este proyecto utilizaremos como herramienta CASE, Together 6.2 bajo el modelo de elementos reutilizables MECANO.

Broker Nodos encaminadores, actúan como distribuidores de mensajes entre el resto de los nodos del entorno. Realizan diversas funciones de gestión del protocolo. Entre las tareas más importantes de las que llevan a cabo, está la de validar los mensajes intercambiados para garantizar que sólo contienen objetos y modelos pertenecientes al modelo de componente reutilizable empleado.

Processor Nodos específicos para realizar un servicio concreto sobre objetos y/o modelos y devolver el resultado. Interaccionan exclusivamente con otros nodos.

Repository Nodos del entorno de desarrollo que ofrecen servicios de repositorio. Realizan tareas de almacenamiento y consulta para objetos pertenecientes al modelo de componente reutilizable empleado.

A los nodos **Processor**, **Repository** y **Tool**, se les denominará **Nodos de Servicios**, pues son los que realizan las tareas realmente útiles para el trabajo con *MECANOS*, mientras que las tareas que realiza el nodo **Broker**, tienen que ver con la gestión del protocolo y de la comunicación entre los nodos de servicios.

Los repositorios de componentes reutilizables pueden ser referenciales o de contención. Los repositorios de contención incluyen los componentes reutilizables en su propio esquema de almacenamiento, mientras que los referenciales actúan como catálogos de información, conteniendo las direcciones donde se deben solicitar los componentes deseados (en un caso se envía sólo la información del modelo necesaria para catalogar el componente y su referencia, y en el otro caso, en lugar de la referencia se envía el componente).

Para conseguir una transparencia en la recuperación de los elementos reutilizables no diferenciando si es repositorio es de tipo referencial o de contención se ha definido un quinto elemento:

FileBroker, este nodo ofrece servicios de recuperación de ficheros y de repositorio de contención interno. Ofrecerán servicios de almacenamiento y recuperación local para los ficheros que contienen los objetos del modelo referencial, así como de recuperación para los ficheros almacenados en otros hosts. Este tipo de nodo se encuentra más cercano al grupo de nodos de servicio, pero no se le va a incluir de momento en este grupo.

A continuación podemos ver un ejemplo de un entorno de desarrollo con los distintos tipos de nodos definidos.

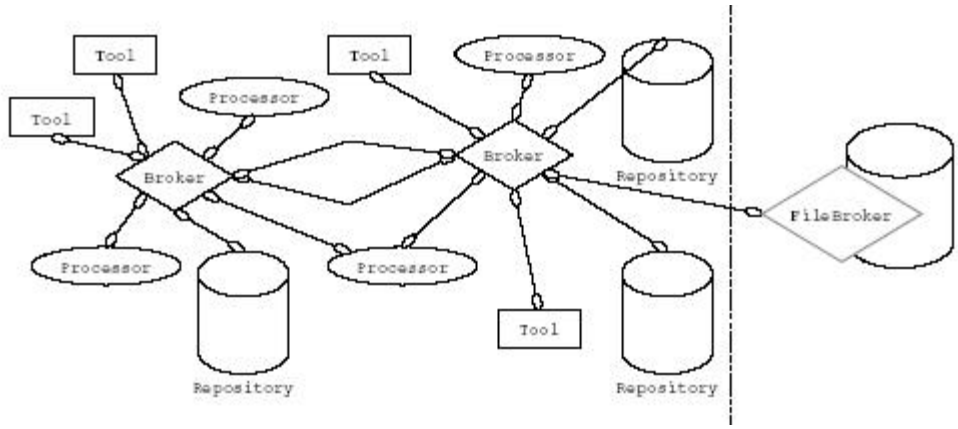


Figura 3.1 *Ejemplo de colaboración entre entidades del entorno que desarrollamos*

En principio, todos los nodos tienen la capacidad de realizar una petición a cualquier otro nodo, y de responder a estas peticiones siempre y cuando se realicen mediante el protocolo propio del entorno. El tipo de peticiones que puede atender cada nodo dependerá de los servicios que ofrezca. Los mensajes siempre serán encaminados por los nodos Broker.

3.1.2. Servicios de los nodos

Los servicios que ofrece un nodo estarán dados por, *los servicios comunes* a todos los tipos de nodos, los servicios predefinidos para el tipo de nodo genérico al que corresponde, los servicios adicionales que se definan para ser ofrecidos por un tipo especializado de uno de los tipos genéricos, más los servicios particulares que añada la implementación específica de un nodo.

Servicios Comunes

Todos los nodos son capaces de identificar al usuario y de registrar sus accesos mediante el establecimiento de sesiones. Además pueden proporcionar información acerca de su estado, los servicios que ofrecen y las funciones que pueden ejecutar.

Servicios de los nodos Tool

Los nodos Tool generalmente no ofrecerán servicios al resto de los nodos del entorno, con excepción de los servicios comunes. Sólo se ha definido uno inicialmente que permite preguntar a un nodo Tool por información relativa a un usuario. Algunos aspectos de este servicio se dejan a decisión de la implementación específica de cada nodo, como por ejemplo si se permite que el nodo proporcione automáticamente la información solicitada.

Este servicio será especialmente útil de cara a integrar de forma transparente en el entorno repositorios que respondan al modelo referencial, a través de nodos FileBroker, como se verá más adelante en el Apartado *4.3*.

Servicios de los nodos Broker

Las funciones principales de los nodos Broker incluyen el encaminamiento, la distribución de los mensajes en el entorno y la verificación de los mismos.

A la hora de distribuir los mensajes, la tarea se ve facilitada al conocer los nodos Broker las direcciones de cada uno de los nodos. Para ello, mantienen un registro de los nodos activos en un entorno, con información enviada por los propios nodos, acerca de la ubicación y el estado de cada uno. A través del Broker un nodo puede conocer el estado de cualquier otro nodo.

En cuanto a la comprobación de los mensajes, el Broker se encarga de validar que los objetos intercambiados responden al modelo de componente reutilizable empleado. Se

garantiza por lo tanto, que los nodos sólo recibirán objetos pertenecientes al modelo con el que son capaces de trabajar

Bajo petición de un nodo, un Broker puede registrar los objetos intercambiados en una comunicación. La principal utilidad de este servicio, es que en vez de reenviar un objeto, un nodo puede enviar su referencia en el registro del Broker, de modo que este se recupera desde el registro actuando el Broker como una caché de objetos. Con este servicio se reduce considerablemente el tráfico de red.

Otros servicios importantes del Broker incluyen por ejemplo el control de acceso a los diferentes servicios ofrecidos en el entorno, según el rol del usuario solicitante. Se ha definido una política general de accesos según los roles que marca el desarrollo basado en reutilización y según los servicios que ofrecen los nodos. Dicha política puede extenderse en cada nodo, siempre añadiendo restricciones a las que se impone mediante la política predefinida, es decir, la nueva política resultante siempre es una conjunción de las restricciones de las de la predefinida con las de la extensión.

Servicios de los nodos Processor

Los nodos Processor pueden ofrecer cualquier tipo de servicio no interactivo sobre un objeto o conjunto de objetos. Posibles servicios pueden ser, comprobar un conjunto de objetos para verificar que pertenecen al modelo de componente reutilizable, realizar cálculos de métricas, etc.

Servicios de los nodos Repository

Los nodos Repository ofrecen servicios de un repositorio de componentes reutilizables, pero diseñados para trabajar directamente con el modelo de componente reutilizable empleado.

El servicio de recuperación permite obtener componentes reutilizables siguiendo las reglas del modelo de reutilización empleado, es decir extrayendo además de los componentes estrictamente solicitados, todos aquellos relacionados con ellos que indica el modelo, normalmente según el tipo o la semántica de las relaciones que los unen.

3.1.3 Protocolo de intercambio de objetos y servicios

De la coexistencia de herramientas y entornos trabajando sobre un mismo modelo de componente reutilizable surge la necesidad de comunicación entre ellas, principalmente la comunicación con repositorios, hace necesario el desarrollo de un protocolo común que permita esta comunicación. La definición del protocolo (formato de mensajes, conexiones, etc.) es lo suficientemente general, para que pueda ser utilizado por el mayor conjunto posible de herramientas y aplicaciones. Por otra parte, ciertos aspectos del protocolo deberán ser específicos para poder recoger el modelo de componente reutilizable, de forma que, por ejemplo, se puedan realizar las comprobaciones sobre la corrección respecto al modelo de los elementos intercambiados.

El protocolo desarrollado define los objetos y servicios que pueden intercambiarse entre los nodos del entorno, así como el formato de los mensajes que se utilizan para intercambiarlos.

Para permitir que los mensajes puedan procesarse fácilmente con independencia de la tecnología utilizada en la implementación del protocolo, se ha optado por utilizar XML como lenguaje de codificación para los mismos.

Además, un protocolo para un entorno de estas características debe permitir reducir lo máximo posible el tráfico de red y el uso de recursos, dedicados a la comunicación. Para ello, se ha intentado que se pueda enviar el máximo volumen de objetos y se puedan realizar el máximo número de servicios con el mínimo número de mensajes.

3.1.3.1 Formato de los mensajes

El formato de los mensajes responde principalmente a estas necesidades:

- Permitir el intercambio de cualquier conjunto de objetos pertenecientes al modelo de componente reutilizable junto con las relaciones entre ellos.
- Permitir que mediante una única petición y a partir de un mismo conjunto de datos de entrada, se puedan realizar varias operaciones simples (servicios), que constituyan conjuntamente una operación más compleja.
- Delimitar y separar los elementos dependientes del modelo de reutilización.

El formato general puede verse en la Figura 3.2. Puede distinguirse:

la parte independiente del modelo de componente reutilizable: constituida por las cabeceras del mensaje (MIPHeader), las cabeceras de los bloques de petición (Request, Result y Dialog) y los datos de gestión del protocolo (XMD)

la parte dependiente del modelo de componente reutilizable (X-REM): consistente en el conjunto de objetos y relaciones pertenecientes al modelo El conjunto principal de datos de trabajo de los servicios es el modelo de componente reutilizable. Además de estos datos, los servicios pueden tomar como argumentos y generar como resultados, datos pertenecientes a tipos simples, concretamente a los tipos básicos definidos en el lenguaje Java [6].

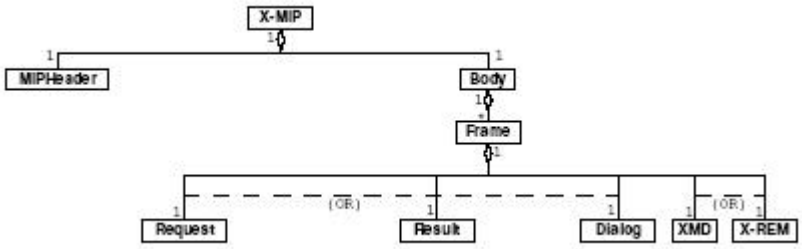


Figura 3.2 *Estructura general de los mensajes*

3.1.3.2 Modelo de componente reutilizable en XML

Para permitir el intercambio de objetos pertenecientes al modelo de reutilización, se ha trasladado un modelo de componente reutilizable completo a XML, partiendo de la descripción del modelo en la forma de un diagrama de clases UML. La manera en que se ha definido en XML el modelo de reutilización, hace posible la ampliación posterior del modelo, ya que la traducción del modelo UML al modelo XML es inmediata, y los cambios o ampliaciones en el modelo original pueden trasladarse al modelo XML. El formato, permite intercambiar conjuntos de objetos y las relaciones entre ellos.

La DTD para la formación del fichero XML es:

```
<!ELEMENT X-REM ( Object*, Association*)>
<!ELEMENT Object ( Asset | Mecano | Author | Representation | Domain | ... ) >
<!ELEMENT Association ( assocFrom_Asset | assocFrom_Mecano | ...)>
```

El modelo de MECANO se muestra en la figura 3.3. Cada clase del modelo, se representará en XML como un elemento del mismo nombre que la clase. Los atributos de cada clase, se incorporarán como elementos y todos se declararán como opcionales. Se muestra sólo el ejemplo de la clase Asset:

```
<!ELEMENT Asset(Name? , Abstract? , AbstractionLevel? , ...) >
```

Por último, los objetos del documento XML (elementos Object) contienen información necesaria para la gestión del protocolo, como un identificador específico para poder establecer referencias a ellos. Las asociaciones (elementos Association), contienen las referencias a los objetos origen y

destino de las mismas. Para poder incluir las relaciones en el documento, manteniendo la semántica del modelo, se declaran bajo Association todos los posibles orígenes de asociación como: asocFrom_ClassName, que actuará como contenedor de las relaciones. En cada uno de estos elementos se incluyen las clases de destino de la relación y se les añade la cardinalidad que establece el modelo de reutilización.

```
<!ATTLIST Object objID ID #REQUIRED>
<!ATTLIST Association sObjLink              CDATA   #REQUIRED
                      sObjLocation          (MSG|TALK|SHELF|REP)   "MSG"
                      sLinkOBehaviour       (REPLACE|MIX)          "MIX"
                      sLinkLBehaviour       (IGNORE|REPLACE|MIX)   "IGNORE"
>
<!ELEMENT assocFrom_Asset
                      (assocTo_Author*,  assocTo_Representation*,
                      assocTo_Mecano*, assocTo_AssetType?,
                      assocTo_Relationship_asRol_Source*,
                      assocTo_Relationship_asRol_Target*,
                      assocTo_FunctionalDescriptor*,
                      assocTo_OtherClass*)
>
```

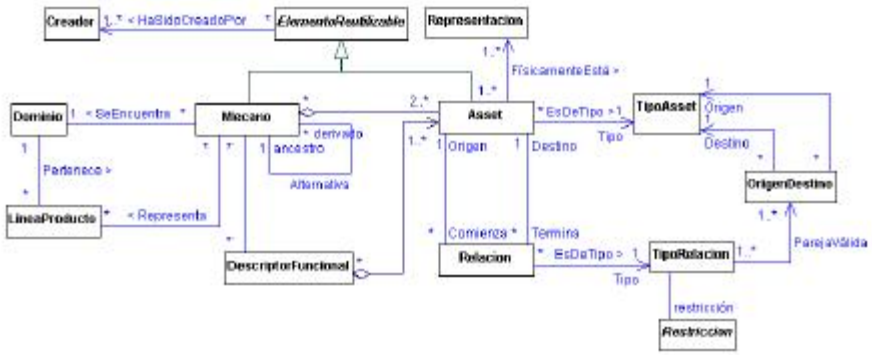


Figura 3.3 *Estructura del modelo de MECANO*

3.1.3.3 Solicitud de Servicios

La solicitud de servicios se realiza mediante el bloque de petición *Request*, cuya estructura se muestra en la figura 3.4 .

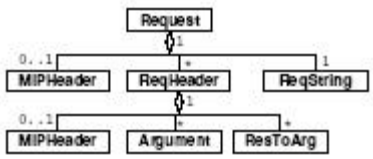


Figura 3.4 *Estructura del bloque de petición Request*

Dentro de este bloque, se utiliza un lenguaje sencillo que permite solicitar la ejecución combinada de varios de ellos (elemento Reqstring). Una petición de servicios realizada mediante este lenguaje, está constituida por los identificadores de cada servicio, enlazados mediante operadores que expresan ejecución en secuencia ";", encadenamiento "."(composición de servicios) y operaciones condicionales"service0?service1:service2".

En una misma petición, y mediante un único mensaje, se puede solicitar, por ejemplo, la apertura de una sesión, el almacenamiento de un conjunto de elementos reutilizables, la consulta de esos componentes para obtener sus identificadores en el repositorio y el cierre de la sesión:

```
<reqString >
OPENSESSION;STORE.QUERY;CLOSESESSION;
</ reqString >
```

El protocolo permite también indicar el nodo en el que se desea ejecutar cada servicio, lo que ofrece la posibilidad por ejemplo, de encadenar servicios entre varios nodos.

Para ello se incluirán cabeceras de mensaje en el bloque de petición, elementos MIPHeader. En la cabecera del bloque de petición se incluye una subcabecera para cada servicio solicitado, dónde se añaden los datos de entrada no pertenecientes al modelo de componente reutilizable (elementos Argument), información acerca de como debe realizarse la composición de servicios (elemento ResToArg), y se puede incluir también la cabecera MIPHeader que indica qué nodo que debe ejecutar ese servicio.

3.1.3.4 Extensión y adaptación del protocolo

Las posibilidades de adaptación y extensión del protocolo son varias. La más importante es, quizá, la posibilidad de extender el modelo de componente reutilizable o incluso de utilizar otros modelos. También existe la posibilidad de utilizar el protocolo a través de diferentes medios de transmisión (conexiones) y la posibilidad de aplicar filtros de compresión y encriptación para mejorar el rendimiento y la seguridad del mismo.

3.1.4 Diseño y solución basado en frameworks

La arquitectura ofrece, por un lado, la escalabilidad de funciones y servicios en los nodos, permitiendo que un nodo incorpore nuevos servicios, y por otro lado, la adaptabilidad de los servicios, permitiendo que la funcionalidad independiente del modelo de reutilización pueda recaer en diferentes soluciones tecnológicas.

El framework implementa las funcionalidades básicas para trabajar con objetos del modelo de componente reutilizable y las funciones necesarias para la ejecución de los diferentes servicios.

En esta sección se presenta una visión general del framework y se detallan algunos aspectos de éste. También se introducen detalles acerca de los nodos *FileBroker*.

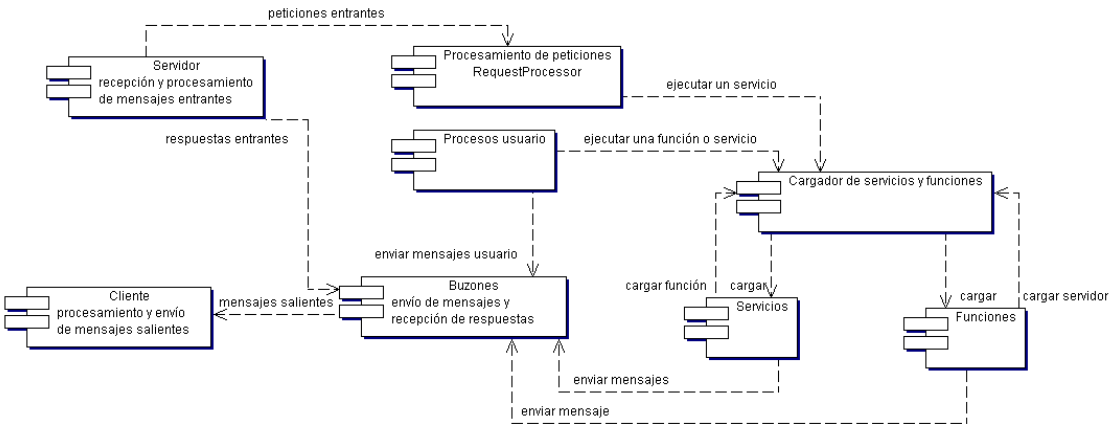


Figura 3.5 *Visión general del Framework*

3.1.4.1 Framework genérico para los nodos del entorno de desarrollo

La figura 3.5 representa el framework genérico que mantiene la funcionalidad de los nodos, mostrando la visión general mediante su descomposición modular. Cada nodo dispone de la funcionalidad básica para enviar y recibir mensajes empleando el protocolo particular del entorno, y de realizar y solicitar servicios sobre objetos pertenecientes al modelo de componente reutilizable empleado.

La funcionalidad que es capaz de ejecutar un nodo sobre modelos de componentes reutilizables, se divide en *funciones* y *servicios*. Las funciones engloban a cualquier funcionalidad del nodo y son ejecutadas desde otras funciones del propio nodo, por ejemplo, la validación de mensajes que realiza un nodo Broker. Los servicios son funciones con la capacidad añadida de que pueden ser solicitados desde otros nodos y una vez finalizados devuelven los resultados generados al nodo que los solicitó.

La adaptabilidad del nodo, permite configurar qué funciones y servicios será capaz de ejecutar y dependerá tanto del tipo del nodo, como de las configuraciones adicionales que realice el desarrollador que lo instancie. La escalabilidad de un nodo, permite que un desarrollador, incluya nuevas funciones y servicios en los nodos. Inicialmente, se han elaborado las pertinentes configuraciones por defecto, para instanciar los diferentes tipos de nodos con los servicios predefinidos, comunes y específicos, correspondientes.

3.1.4.2 Repositorios de Contención

Un repositorio referencial no mantiene los ficheros de los componentes reutilizables, sino que gestiona un catálogo de información, conteniendo las direcciones desde dónde el desarrollador o proveedor ofrece dichos componentes a sus clientes. Esto pasa generalmente por utilizar una URL como referencia para recuperar el fichero de un componente reutilizable, lo que supone en la mayoría de los casos que el fichero requerido se puede obtener directamente desde esta dirección, y que de algún modo es accesible públicamente. En ocasiones puede ser que la URL no sea suficiente para obtener un fichero. Puede ser que el proveedor requiera el envío de un formulario para recopilar información acerca del uso que se le va a dar al componente, la autentificación de un usuario autorizado, o la utilización de una aplicación cliente específica para recuperar el fichero en cuestión.

En una URL se puede añadir información diversa, como un login y un password de un usuario, y puede ir incluso dirigida a una aplicación web que resuelva la petición, incluyendo una 'querystring' que contenga la información requerida por el proveedor. Pero en ambos casos, se requiere del envío de información variable y dependiente del usuario. Esta información no se puede incluir estáticamente en la referencia asociada al componente reutilizable que aparece en el catálogo.

Para mantener la posibilidad de acceder al fichero del componente a partir de su URL y permitir, de la manera más transparente posible al usuario, el envío de la información adicional, requerida por el proveedor, la URL, que figura como referencia del componente reutilizable, no tiene que identificar directamente el recurso, sino que puede apuntar a una ubicación intermedia dónde se pueden alojar herramientas que permiten resolver la ubicación definitiva y realizar las tareas pertinentes (envío de un formulario, solicitud de datos al usuario de forma interactiva, etc.) para obtener el fichero que almacena el proveedor.

Un repositorio de contención integrado en el entorno de desarrollo del modelo referencial: el nodo FileBroker. La parte central para integrar un repositorio de contención es incluir en el entorno de desarrollo del modelo referencial, un nodo que ofrece servicios de contención local y de recuperación de componentes reutilizables de forma transparente.

Este nodo, denominado FileBroker, actuará como interfaz entre la URL de un objeto y el fichero del componente, ubicado en el repositorio de contención local o en el del proveedor. Al realizar la

recuperación, se realizarán todas la tareas que se requieren para obtener el fichero del proveedor añadiéndose, de forma transparente, toda la información adicional necesaria.

Un nodo FileBroker almacenará las referencias reales a los ficheros y generará URL's relativas consistentes en su propia dirección y algún identificador adicional. Estas URL's relativas serán las que se almacenen en el repositorio referencial, El FileBroker ofrecerá también un servicio de repositorio de contención local para los ficheros de componentes, orientado a proveedores que no puedan o no quieran mantener sus propios repositorios.

3.2 Revisión del framework

Inicialmente la revisión del framework iba dirigida a partes que no se habían implementado, a partes del framework que pudieran chocar con otras ya implementadas, incluso en definir nuevos servicios que pudieran ampliar la funcionalidad de éste. En el estudio realizado sobre el proyecto inicial, pude constatar, que lo desarrollado hasta el momento .se basaba en módulos aislados, base para la implementación propiamente dicha del framework. Los módulos instanciados estaban localizados en el la iteración 1 y en la iteración 2, que eran base para partes muy concretas del resto de las iteraciones, dejándolas muy abiertas.

El grueso del framework, basado en los modelos de protocolo y de reutilización, estaban sin codificar, por lo que se procedió a ello. Este hecho originó diversos problemas, pues es la parte más importante del framework y la que marca las directrices en el funcionamiento del mismo. La revisión se convirtió en un rediseño y nueva codificación de las partes que ya existían.

Por otro lado, la definición de la DTD del mensaje, al no haber sido utilizada, no era correcta, poseía errores sintácticos.

Según fuimos desarrollando encontramos partes del modelo que no se podía representar correctamente, como pueden ser la representación de las asociaciones entre elementos de un modelo de UML. Esto ocasionó tener que volver a cambiar la DTD que marca la configuración del mensaje.

Estableciendo la relación del diseño con la implementación, surgieron clases propias de esta última etapa y que no se habían localizado en el diseño, esto lo podemos ver en el iteración 3, con los servicios de los nodos, en la que ha surgido una estructura para que éstos estén bien ordenados.

Otro tipo de cambios en los métodos básicos, su redefinición, pues el acoplamiento con otros, no se podía realizar correctamente

Toda la estructura de representación interna de los datos del mensaje, ha sido creada nuevamente, pudiendo almacenar en el nodo una estructura DOM y de grafos, dando la posibilidad de convertir esta estructura nuevamente a texto, pudiéndola filtrar o encriptar para que sea objeto de envío por el canal.

Se han desarrollado servicios como pueden ser el encaminamiento en el nodo Broker, la creación de mensajes, el procesamiento del mensaje utilizando características intrínsecas de la definición del mensaje, tales como el lenguaje de utilización de servicios.

En resumen, los grandes cambios realizados en cada una de las iteraciones son:

- Iteración 1: Representación DOM, de texto a DOM y de DOM a texto.
- Iteración 2: Creación de módulos para codificar y encriptar.
- Iteración 3: Reestructuración de la clase Skill. Implementación de todas las clases independientemente del modelo. Representación de los datos en grafos.
- Iteración 4: Implementación para la formación de mensajes.
- Iteración 5: Implementación de servicios, funciones y características básicas de los nodos.

3.3 Definición del entorno de explotación

Se ha definido un entorno en el que aplicaremos todo lo desarrollado para el framework y donde anexionaremos lo propuesto en el proyecto fin de carrera [*NCAP02*], el entorno cumple los rasgos principales de este último proyecto. En él tenemos definidas tres partes claras:

- *Servidor*: encargado de la comunicación e inserción en la BD.
- *Cliente*: Nodo desarrollado en Together 6.2 (no en 5.2 como esta elaborado este trabajo), y desde el cual el cliente demanda los servicios que requiere para conseguir insertar elementos reutilizables (de tipo Asset y Mecano) en el repositorio.
- *Protocolo*: Forma con la que ponemos de acuerdo al *Servidor* y al *Cliente*.

Está claro que esta estructura desaparece con la utilización del framework, pues este ya nos marca los servicios propios de un cliente y un servidor y la forma con la que se comunicarán ambas partes, por ello la parte de los requisitos, tanto funcionales como no funcionales, de los *servicios* que ofrece el proyecto [*NCAP002*] serán las pautas tomadas para formar los servicios que se formarán en el framework.

La estructura que instanciaremos será

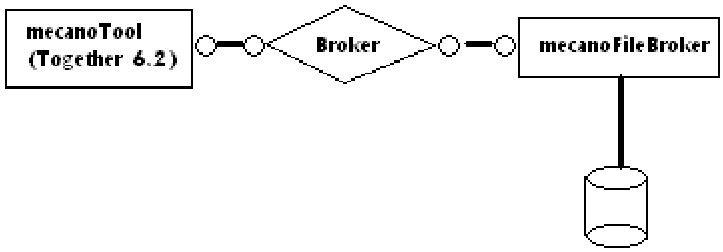


Figura 3.6 *Entorno de desarrollo*

Donde mecanoTool es el elemento que definiremos en la herramienta Together 6.2, el broker será un elemento que instanciaremos en la misma máquina que Together 6.2 u otra cualquiera, por configuración ambos elementos tienen que estar relacionados. Lo mismo sucede con el Broker y el FileBroker, tienen que estar asociados mediante configuración.

El mecanoFileBroker es el nodo que se encargará de la comunicación con la BD, así como del almacenamiento de los elementos.

En la BD almacenaremos características y cualidades de los distintos elementos reutilizables. La BD se encuentra en la máquina MARTE, localizada en el departamento de GIRO. Para conocer más sobre su estructura podemos dirigirnos a *[PLPG00]*.

Para el caso práctico que hemos desarrollado, aunque los datos que definen un elemento se almacenan en MARTE, el elemento en sí será almacenado en el nodo MecanoFileBroker, con la misma estructura que se encuentre en el proyecto inicial desarrollado en Together 6.2.

Intuimos como mecanoTool hace las veces que la estructura Cliente del proyecto [*NCAP002*], mecanoFileBroker, simula al Servidor y el protocolo queda definido por el framework. El nodo Broker será el encargado de validar la estructura del mensaje así como de encaminar como ya hemos visto en la definición de servicios de este tipo de nodos.

Los servicios implementados para los nodos a los que hacemos referencia son los básicos: servicio Cliente, servicio Servidor, encaminamiento para el Broker, validación básica del mensaje,

procesamiento del mensaje recibido, ejecución de los servicios definidos por el lenguaje de servicios utilizado y formación del nuevo mensaje para constituir el "resultado".

3.3.2 Consideraciones prácticas

Para que la parte del trabajo dedicada a la implementación práctica sea posible, se ha reestructurado la DTD para que podamos trabajar en un entorno de pruebas.

- No se ha considerado la existencia de la etiqueta ***<Dialog>*** en *x-mip.dtd*; esto quiere decir que la comunicación con los nodos se hará de una manera rígida, no se podrán mandar mensajes, en medio de un mensaje X-MIP para poder pedir algún tipo de información a un nodo. Por ello todas las funciones así como instanciación de la clase Dialog (como veremos en Capitulo 7) se han mantenido inalterable.
- La estructura que se mostrará es la de un nodo Broker único, sobre el que se centralizarán todos los nodos, en este caso, los nodos que realizarán la petición de los servicios (Tool, y construido bajo Together) y el nodo que hará de conexión con el repositorio de GIRO, (nodo FileBroker).
- El protocolo planteado no transmite elementos físicos, por lo que se utilizarán características de X-MIP para intentar solapar lo establecido en el proyecto [*NCAP02*] y el protocolo iniciado en [*Per03*], y perfilado en este trabajo, consiguiendo almacenar estos elementos en el mensaje. En el primer proyecto se mandan los elementos físicos para que sean almacenados en el repositorio. Se planteó la opción de que cada nodo, sea del tipo que sea, almacenase sus elementos reutilizables y el repositorio fuese de tipo referencial, pero por intentar guardar similitudes con los proyectos estudiados habrá un nodo FileBroker en el que se almacenen de forma física los elementos reutilizables y se ponga en comunicación con la BD.

- La comunicación se realizará con tres nodos
  1. *Tool*, con el que crearemos en Together 6.2 un Asset o Mecano, siempre partiendo del XMI que nos proporciona esta herramienta.
  2. *Broker*, con el que enrutaremos y validaremos (en algún grado).
  3. *FileBroker,* en el que almacenaremos los elementos reutilizables y con el que podremos comunicarnos con el repositorio de GIRO.

     El nodo *FileBroker* obtendrá los elementos provenientes del mensaje y los almacenara de forma física y referencial. Esta opción se ha dejado muy abierta para que las futuras ampliaciones de inserción y extracción sean lo mas cómodas posibles.
- En el proyecto [NCAP02] la comunicación entre Together 6.2 y el repositorio se hace mediante *servlets*, y se estableció un protocolo único y cerrado para la comunicación. En el caso práctico la comunicación la realiza el framework.

Considerar que por ahora la autentificación del usuario, está en la propia implementación la conexión, siempre fiable, que el usuario es constante y tiene permisos para insertar. Esto se ha dejado abierto para que desde la sesión establecida en la estructura distribuida, se pueda conectar con el repositorio.

Parte II

Diseño de un framework para los nodos del entorno de desarrollo

Capítulo 4

Iteración 1: Utilidades

A continuación se detallan un conjunto de clases y relaciones, que serán la base para el desarrollo y especificación de otros paquetes. Se ha conseguido un conjunto de herramientas útiles y reutilizables, tanto para este framework como para servicios que se puedan implementar. Estas herramientas han sido definidas inicialmente en el proyecto del que partimos [Per03] pero aquí se redefinen consiguiendo una funcionalidad mas refinada.

4.1 Requisitos

Número	reqUtil01
Descripción	Se necesitan clases que proporcionen la base del núcleo multihilo del framework, y que puedan ser reutilizadas en otros diseños que necesiten un núcleo servidor o cliente multihilo.
Prioridad	Alta

Número	reqUtil02
Descripción	Se necesitan estructuras de datos para permitir la comunicación asíncrona entre los diferentes hilos de ejecución que constituirán un nodo.
Prioridad	Alta

Número	reqUtil03
Descripción	Se necesita una estructura para acceder a un archivo de registro común que pueda ser utilizado por cualquier clase del framework, y que pueda ser accedido de forma segura en un entorno de ejecución multihilo.
Prioridad	Alta

Número	reqUtil04
Descripción	Se necesita poder mantener y especificar la configuración de un nodo de forma sencilla mediante, por ejemplo, un fichero de configuración. Se necesita diseñar un formato para el fichero y una interfaz, para poder leer la configuración fácilmente desde cada parte del framework teniendo en cuenta siempre que se trata un entorno multihilo.
Prioridad	Alta

Número	reqUtil05
Descripción	Se necesita de un mecanismo para proporcionar identificadores únicos a los distintos elementos de un mensaje: objetos, frames y los propios mensajes.
Prioridad	Alta

4.1 Requisitos

Número	reqUtil06
Descripción	Se necesita un mecanismo para obtener instancias de clases dinámicamente a partir de su nombre y el nombre del paquete, cuando no se conocen las clases previamente en tiempo de desarrollo, pero sí en tiempo de ejecución.
Prioridad	Alta

Número	reqUtil07
Descripción	Se necesita definir clases genéricas para todos los datos que necesitan soporte de representación en XML, se definirán en ellas elementos y métodos genéricos para XML.
Prioridad	Alta

4.2 Diseño

Soporte para un servidor en etapas multihilo

Descripción

Con esta parte se define un mecanismo para atender los mensajes que se reciben del exterior, y a la vez enviar al exterior los mensajes que se generan en el propio nodo.

Se ha diseñado el núcleo del nodo como cadenas de procesamiento multietapa, ejecutándose cada etapa en un hilo independiente. De este modo, se mantiene la capacidad de atender múltiples peticiones simultáneamente pero a la vez se mantiene controlado el número de procesos concurrentes existentes en el sistema al limitar su número al total de las etapas.

Cada una de las etapas se ejecuta en un hilo independiente y se comunican entre sí por medio de colas y eventos. Cada etapa dispone de una cola de entrada y de una tubería para esperar eventos, a su vez, puede enviar datos a la siguiente etapa y enviarle eventos. Existen dos tipos de eventos, uno indica que existen datos en la cola y otro indica que se debe detener la ejecución de la etapa. Cada etapa toma un dato, realiza el procesamiento correspondiente y envía el resultado a la etapa siguiente a través de la cola de entrada de esta. Le notifica que existe un nuevo dato para procesar enviándole un evento a través de la tubería de eventos.

La evolución de esta parte ha sido prácticamente nula, se ha retocado algo los métodos de finalización o parada del servicio, o incluso alguna función que nos permita conocer si los servicios que utilizan este mecanismo están parados o activos y la utilización de una opción de Java para que el procesador se puede dedicar a otro Thread (Thread. yield()).

Clases relacionadas

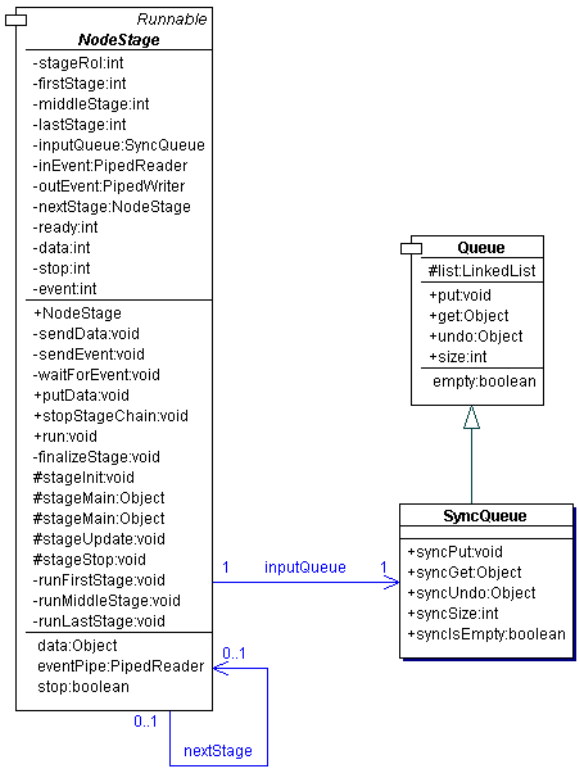
public class **SyncQueue** extends **Queue**	Se trata de un tipo de cola, heredada de la clase Queue, que puede ser instanciada y utilizada como recurso compartido, y ser empleada para comunicar entre sí varios procesos, marcando así el comienzo o finalización de una etapa. Implementa los mismo métodos que la clase Queue, pero *añadiendo bloqueo* al declararlos como *synchronized,* no se redefinen los métodos sino que se diferencian del os anteriores por el prefijo 'sync', de este modo pueden emplearse los dos métodos: sin bloque cuando se tenga la seguridad de que sólo van a ser accedidos por un solo proceso simultáneamente y con bloque cuando no tengamos esta seguridad.
public class **Queue**	Se trata de una cola de elementos Objects.
abstract public class **NodeStage** implements **runnable**	Etapa de un Servidor o Cliente multietapa, *diseñada para ejecutarse como un proceso independiente y comunicarse con otras etapas*. En tiempo de instanciación se indica el rol de la etapa, en una cadena de etapas hay que indicar si se trata de una etapa inicial (firstStage), de una etapa intermedia (middleStage) o de una etapa final (lastStage). Solo podremos encontrar una etapa firstStage y otra lastStage, por el contrario la etapa middleStage la podemos encontrar repetida, pues con ella solo marcamos un rol, no su

	funcionalidad.

Diagramas relacionados

	Diagrama de clases del soporte para clientes y servidores multihilo en etapas.
	Diagrama de secuencia del procesamiento y el intercambio de datos a través de las etapas, cada etapa constituye un proceso concurrente.
	Diagrama de secuencia del proceso de detener la cadena de etapas. Se inicia con el método 'stopStageChain()' que debe ser llamado en la primera etapa, desde un proceso externo o desde la propia etapa.
	Diagrama de estados de una etapa inicial.
	Diagrama de estados de una etapa intermedia.
	Diagrama de estados de una etapa final.

Modelo estático: diagrama de clases



Soporte para clientes y servidores multihilo en etapas

Señalar que todos los métodos de las clases se han revisado, utilizando una serie de iconos para hacer referencia al trabajo efectuado sobre el proyecto inicial:

-  : Referencia asociada a la copia, de métodos o atributos de una clase, del proyecto inicial.
-  : Referencia asociada a la modificación, de métodos en su código, del proyecto inicial.
-  : Referencia asociada a la creación o implementación de nuevos métodos o métodos que estaban sin implementar.

*Campos de la clase **Queue***

Tipo	Declaración	Acción
protected LinkedList	list	

*Métodos de la clase **Queue***

Tipo	Declaración	Acción
public void	put(Object v)	
public Object	get()	
public Object	undo()	
public boolean	isEmpty()	
public int	size()	

*Métodos de la clase **SyncQueue***

Tipo	Declaración	Acción
public synchronized void	syncPut(Object o)	
public synchronized Object	syncGet()	
public synchronized Object	syncUndo()	
public synchronized int	syncSize()	
public synchronized boolean	syncIsEmpty()	

*Campos de la clase **NodeStage***

Tipo	Declaración	Acción
private int	stageRol	
private int	firstStage	
private int	middleStage	

private int	lastStage	
private SyncQueue	inputQueue	
private PipedReader	inEvent	
private PipedWriter	outEvent	
private NodeStage	nextStage	
private int	ready	
private int	data	
private int	stop	
private int	event	

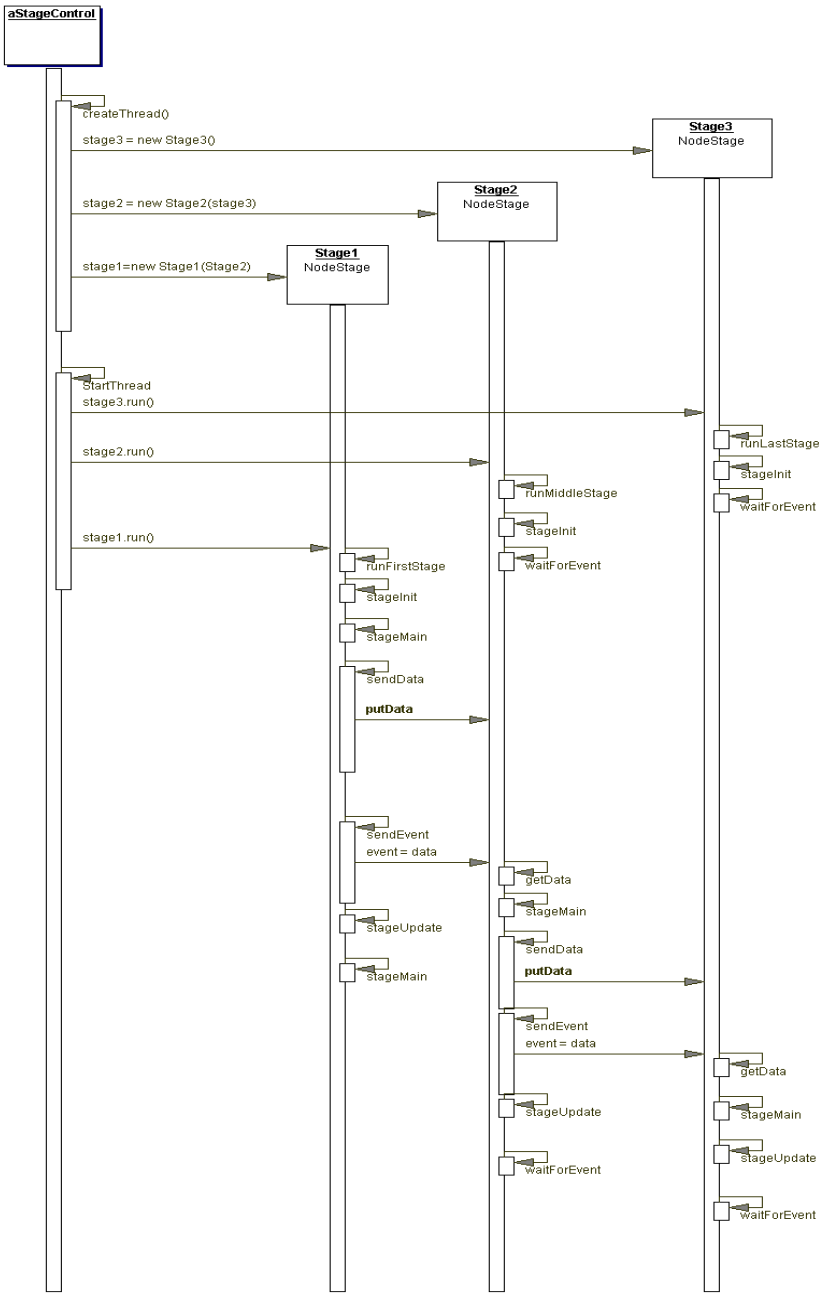
Constructores de la clase ***NodeStage***

Tipo	Declaración	Acción
public	NodeStage (NodeStage nextStage, boolean hasInput)	

Métodos de la clase ***NodeStage***

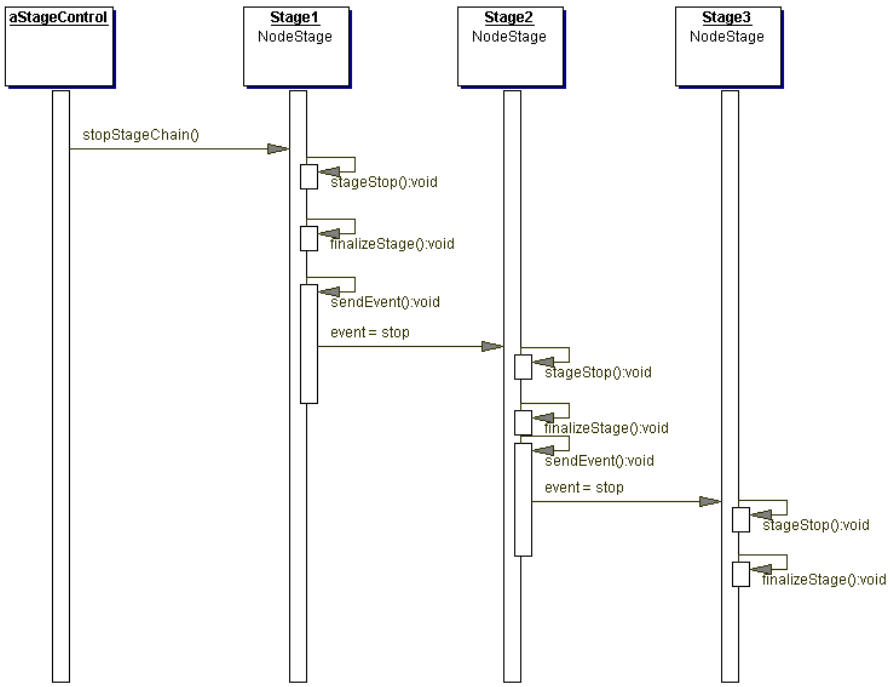
Tipo	Declaración	Acción
private Object	getData()	
private void	sendData (Object data)	
private void	sendEvent()	
private void	waitForEvent()	
public void	putData (Object data)	
public PipedReader	getEventPipe()	
public void	stopStageChain()	
public void	run()	
private void	finalizeStage()	
protected void	stageInit()	
protected Object	stageMain()	
protected Object	stageMain (Object data)	
protected void	stageUpdate()	
protected void	stageStop()	
private void	runFirstStage()	
private void	runMiddleStage()	
private void	runLastStage()	
public boolean	isStop()	

Diagrama de secuencia del procesamiento e intercambio de datos a través de las etapas



En el diagrama vemos el ejemplo de un elemento (cliente o servidor) con tres estados o etapas. La definición de lo que hace el cliente o servidor se define en los métodos *StageInit()*, *StageMain()*, *StageUpdate()*. Todos los estados que creamos son concurrentes, y los implementamos en *JAVA* con clases *Threads*. No se pasa de un estado a otro si no hay un objeto dato, objetivo de la comunicación entre los estados; el evento marca a los estados, que no son el primero, que se debe hacer algo con el objeto dato que ha llegado.

Diagrama de secuencia del proceso de detener la cadena de etapas



Así podemos ver la secuencia que se produce en los objetos que forman un elemento servidor o cliente cuando se quiere finalizar los estados. Los distintos métodos liberan los recursos que son utilizados. El método *stageStop()* es particular de cada uno de los estados, siendo definido con posterioridad.

Diagrama de estados de una etapa inicial

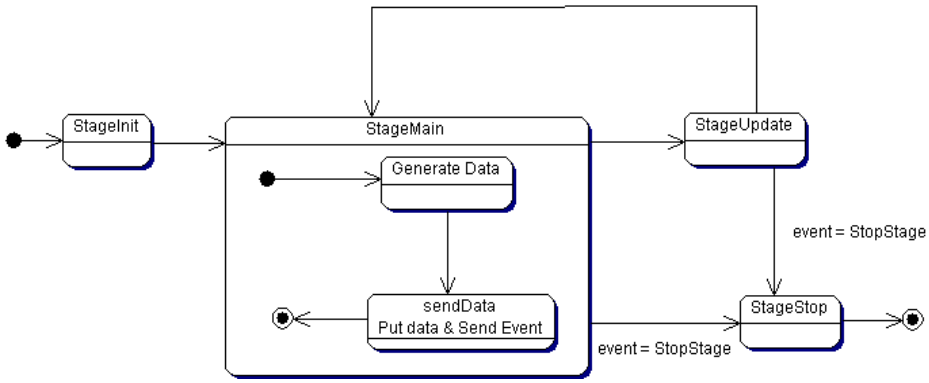


Diagrama de estados de una etapa intermedia

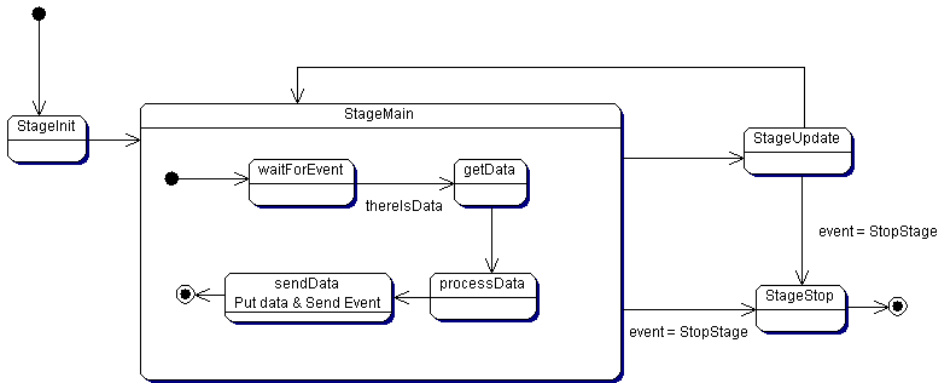
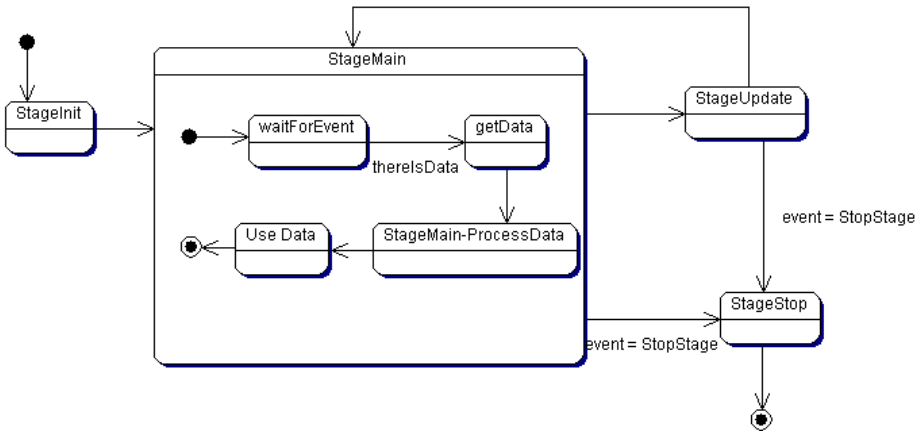


Diagrama de estados de una etapa final



Soporte para un archivo de registro

Descripción

Los nodos que resulten de instanciar el framework se ejecutarán en forma de varios procesos concurrentes y consistirán tanto de una parte que actuará como servidor como de otra que simultáneamente lo hará como cliente. Para poder recoger los diferentes mensajes de error o de notificación de la realización de determinadas operaciones, se ha creído necesaria la existencia de una estructura que represente a un archivo de registro. Este archivo de registro tiene que poder ser creado y escrito desde cualquier clase y proceso que constituya el framework.

Esta parte no ha sufrido ningún cambio.

Clases relacionadas

public class **NodeLog**	Estructura de datos representando un archive de registro, que puede ser accedido desde múltiples procesos concurrentes. Se ha diseñado siguiendo un patrón 'Singleton', lo que garantiza que todas las clases clientes utilizarán la misma instancia para acceder al archivo de registro. Por esta característica hay que implementar bloqueos en los métodos de escritura en el archivo de registro, para garantizar que se realizan de forma correcta en un entorno multiproceso.

Diagramas relacionados

	Diagrama de clases de la estructura de datos de archivos de registro.

Modelo estático: diagrama de clases

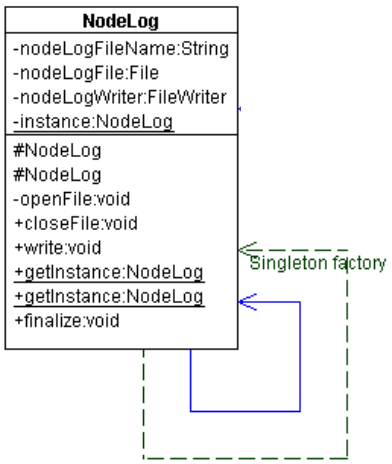


Diagrama de clases del soporte de archivo de registro.

Diccionario de Clases

Campos de la clase ***NodeLog***

Tipo	Declaración	Acción
private String	nodeLogFileName	
private File	nodeLogFile	
private FileWriter	nodeLogWriter	
private static NodeLog	instance	

Constructores de la clase ***NodeLog***

Tipo	Declaración	Acción
protected	NodeLog()	
protected	NodeLog (String fileName)	

*Métodos de la clase **NodeLog***

Tipo	Declaración	Acción
private void	openFile()	
public void	closeFile()	
public synchronized void	write(String logMessage)	
public static NodeLog	getInstance()	
public static NodeLog	getInstance(String fileName)	
public void	finalize()	

Soporte para la configuración de nodos en tiempo de ejecución

Descripción

Se ha pensado en ficheros de configuración para poder configurar el framework fácilmente. A la hora de instanciar el framework y ejecutarlo como un nodo determinado, este leerá su configuración de un archivo, dónde se podrá indicar, desde el tipo de compresión que se utilizará para enviar y recibir mensajes, hasta el tipo de nodo, pasando por los servicios que ofrecerá, modelo de reutilización que empleará, etc...

Debido a que desde el principio del proyecto se ha optado por utilizar XML, como el formato de intercambio de mensajes, se ha tomado la decisión de emplear también este lenguaje para el fichero de configuración.

El formato del fichero de configuración se ha descrito en una sencilla DTD, que se reproduce a continuación:

(nodeConfig.dtd)

```
<?xml version="1.0" encoding="UTF-8"?>
      <!ELEMENT NodeConfig    (Unit*)>
      <!ELEMENT Unit          (RequiredKey?, OptionalKey?)>
      <!ATTLIST Unit
            name  CDATA #REQUIRED
      >
      <!ELEMENT RequiredKey   (Key*)>
      <!ELEMENT OptionalKey   (Key*)>
      <!ELEMENT Key     (#PCDATA)>
      <!ATTLIST Key
            name  CDATA #REQUIRED
      >
```

Cada clase, función, etc., que necesite un soporte de configuración dinámica, se representará en el archivo de configuración como Unit, y recibirá un nombre con el que se reconocerán las opciones que le pertenecen.

Las diferentes opciones no serán más que pares atributo-valor, que se agruparán en opcionales y obligatorias.

Un posible ejemplo de fichero de configuración sería:

(nodeConfig.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE NodeConfig PUBLIC   "SYSTEM"
                                                "nodeConfig.dtd">
<NodeConfig>
     <Unit name="connection">
          <RequiredKey>
               <Key name="package">tcpsocketconnection</Key>
               <Key name="connectionClassName">
                  TCPConnection
               </Key>
          </RequiredKey>
     </Unit>

     <Unit name="filterChain">
          <RequiredKey>
               <Key name="chainConfigMode">fixed</Key>
          </RequiredKey>
          <OptionalKey>
               <Key name="compressor">zipfilter.ZIP</Key>
          </OptionalKey>
     </Unit>

     <Unit name="filterFactory">
          <RequiredKey>
               <Key name="package">zipfilter</Key>
          </RequiredKey>
     </Unit>

     <Unit name="connectionListener">
          <RequiredKey>
               <Key name="port">61300</Key>
          </RequiredKey>
     </Unit>

     <Unit name="router">
          <RequiredKey>
               <Key name="defaultBroker.name">Rhin</Key>
               <Key name="defaultBroker.ip">
                  157.88.124.87
               </Key>
               <Key name="defaultBroker.port">61301</Key>
          </RequiredKey>
     </Unit>
```

```
        <Unit name="model">
             <RequiredKey>
                  <Key name="name">MECANO</Key>
                  <Key name="package">mecano</Key>
                  <Key name="modelClass">MecanoModel</Key>
             </RequiredKey>
        </Unit>

        <Unit name="node">
             <RequiredKey>
                  <Key name="name">Carrion</Key>
                  <Key name="type">processor</Key>
                  <Key name="servicePackage">services</Key>
                  <Key name="functionpackage">functions</Key>
             </RequiredKey>
        </Unit>
   </NodeConfig>
```

Para mas información ver el apéndice B localizado en el CD.

La modificación que ha sufrido esta clase se ha orientado hacia métodos que son necesarios para implementar la captura del fichero de configuración, apareciendo métodos privados para capturar partes específicas y obligatorias del fichero. Se ha requerido tocar lo métodos existentes para acoplar los métodos que se han implementado.

Clases relacionadas

public class **ConfigFile**	Estructura par acceder a ficheros de configuración. Se ha diseñado siguiente un patrón 'Singleton', lo que garantiza que todas las clases clientes utilizarán la misma instancia para acceder al archivo de configuración.

Diagramas relacionados

	Diagrama de clases del soporte de archivos de configuración en XML.

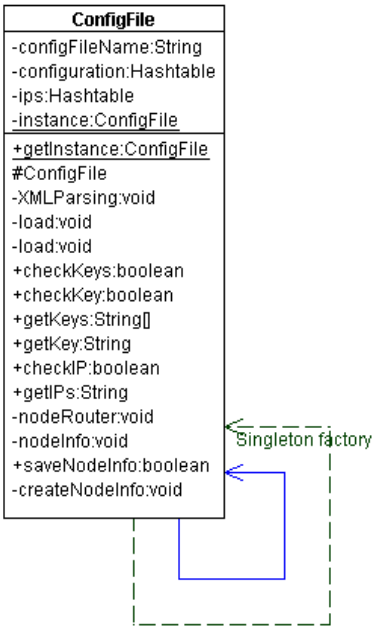


***Diagrama de clases del soporte de archivos de configuración* XML.**

Campos de la clase ***ConfigFile***

Tipo	Declaración	Acción
private String	configFileName	
private Hashtable	configuration	
private Hashtable	ips	
private static ConfigFile	instance	

Constructores de la clase ***ConfigFile***

Tipo	Declaración	Acción
protected	ConfigFile()	

Métodos de la clase ***ConfigFile***

Tipo	Declaración	Acción
public static ConfigFile	getInstance()	

private void	XMLParsing(Node nodo)	
private void	load()	
private void	load(String fileName)	
public boolean	checkKey(String unit,String key)	
public boolean	checkKeys(String unit,String[] keys)	
public String[]	getKeys(String unit,String[] keys)	
public String	getKey(String unit,String key)	
public boolean	checkIP(String IP)	
public String	getIPs(String ip)	
private void	nodeRouter(Document docu,Node nodo, String alias,NodeInfo addr,String mode)	
private void	nodeInfo(Document docu, Node nodo, String alias,NodeInfo addr,String mode)	
private void	createNodeInfo(Document docu,String alias, NodeInfo addr)	
public boolean	saveNodeInfo(String alias,NodeInfo addr, String mode)	

Soporte para la Generación de Identificadores Únicos

Descripción

Cuando se crean en un nodo nuevos elementos para ser enviados, como objetos de un modelo, bloques de petición (frames), o mensajes, se necesita que todos ellos puedan ser diferenciados mediante un identificador único por cada tipo de elemento y para los generados en ese nodo. Para ofrecer está función al nodo, se utiliza una clase generadora de identificadores que se diseñará siguiendo un patrón 'singleton'.

Se han creado nuevos métodos para facilitar la gestión de creación y obtención de identificadores, y se han implementado métodos que existían pero no estaban codificados.

Clases relacionadas

public class **Numberers**	Actúa como interfaz para crear un conjunto de clases Numberer, una por cada tipo de elemento que se necesita numerar. Actúa también como interfaz común para acceder a las clases de cada tipo *Numberer*. Se ha diseñado siguiendo un patrón '*Singleton*', lo que garantiza que todas las clases clientes utilizarán la misma instancia.

public class **Numberer**	Clase que genera identificadores para un tipo de elementos concretos. Añade al identificador la inicial del tipo al que se encuentra asociado. El método para generar identificadores se bloquean para que puedan ser accedidos por múltiples procesos y garantizar que se genera una secuencia coherente de identificadores y que estos sean únicos.

Diagramas relacionados

	Diagrama de clases de las estructuras de generación de identificadores únicos.

Modelo estático: diagrama de clases

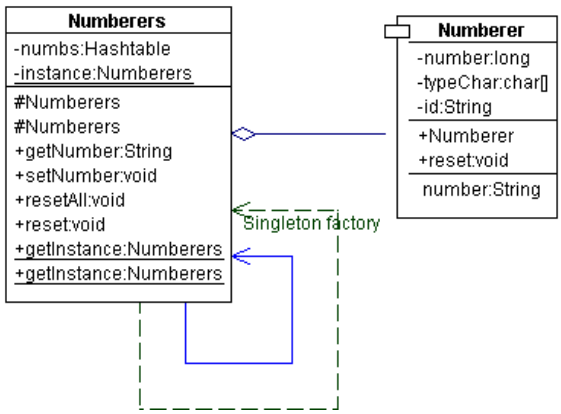


Diagrama de clases de las clases generadoras de identificadores únicos.

Diccionario de Clases

*Campos de la clase **Numberer***

Tipo	Declaración	Acción
private long	number	
private char[]	typeChar	
private String	id	

*Constructores de la clase **Numberer***

Tipo	Declaración	Acción
public	Numberer(char type)	

*Métodos de la clase **Numberer***

Tipo	Declaración	Acción
public synchronized String	getNumber()	
public synchronized void	reset()	

*Campos de la clase **Numberers***

Tipo	Declaración	Acción
private Hashtable	numbs	
private static Numberers	instance	

*Constructores de la clase **Numberers***

Tipo	Declaración	Acción
protected	Numberers(String[] numberer)	
protected	Numberers(String numberers)	

*Métodos de la clase **Numberers***

Tipo	Declaración	Acción
public String	getNumber(String type)	
public void	setNumber(String type)	
public void	resetAll()	
public void	reset(String[] numberer)	
public static Numberers	getInstance(String[] numberers)	
public static Numberers	getInstance(String numberers)	

Cargador dinámico de Clases

Descripción

Con este mecanismo conseguimos configurar, fácilmente, los nodos en tiempo de ejecución, a partir de las distintas funciones, servicios y diferentes implementaciones que pueden estar disponibles. Para esto se necesitaba una forma indicarle al nodo, cuando arrancase, la configuración con la que debería ejecutarse. Con los ficheros de configuración se resuelve este problema, pero además se necesitaba de un mecanismo para poder utilizar clases de las que no se conoce, en tiempo de desarrollo, ni su nombre ni su ubicación.

El enfoque típico para resolver este problema es utilizar clases diseñadas según el patrón "Factory". Esto es, clases encargadas exclusivamente de instanciar otras clases de las que sólo se conocen, en tiempo de ejecución, los datos necesarios para ello. La manera de instanciar estas clases puede encontrarse implementada en las clases "Factory", pero esto no servía para los requisitos del

proyecto, ya que se deseaba evitar que la instanciación de un Framework implicara 'tocar' el código.

Finalmente se han utilizado clases que siguen el patrón 'Factory' y que utilizan la información del fichero de configuración para instanciar las clases desconocidas en tiempo de desarrollo. Para obtener una instancia de una clase en Java se utiliza el método forName(String className), de la clase java.lang.Class. Como finalmente, todos las clases 'Factory', utilizan este método para obtener instancias de clases, se ha encapsulado este método, junto con alguna funcionalidad adicional en una clase específica que se ha incorporado al paquete de utilidades. Se trata de la clase Loader, que se describe a continuación. y que constituye la interfaz que se utilizará en el framework para acceder a instancias de clases desconocidas en tiempo de desarrollo.

Esta clase no ha sufrido ningún cambio.

Clases relacionadas

public class **Loader**	Clase que efectúa la carga dinámica de clases.

Diagramas relacionados

	Diagrama de clases del cargador dinámico de clases.
	Diagrama de secuencia de la carga dinámica de clases.

Modelo estático: diagrama de clases

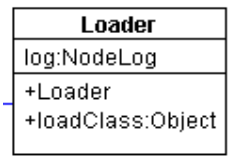


Diagrama de clases del cargador dinámico de clases

Diccionario de Clases

*Constructores de la clase **Loader***

Tipo	Declaración	Acción
public	Loader()	

Métodos de la clase ***Loader***

Tipo	Declaración	Acción
public Object	loadClass (String packageName, String className)	

Modelo dinámico: diagrama de secuencia

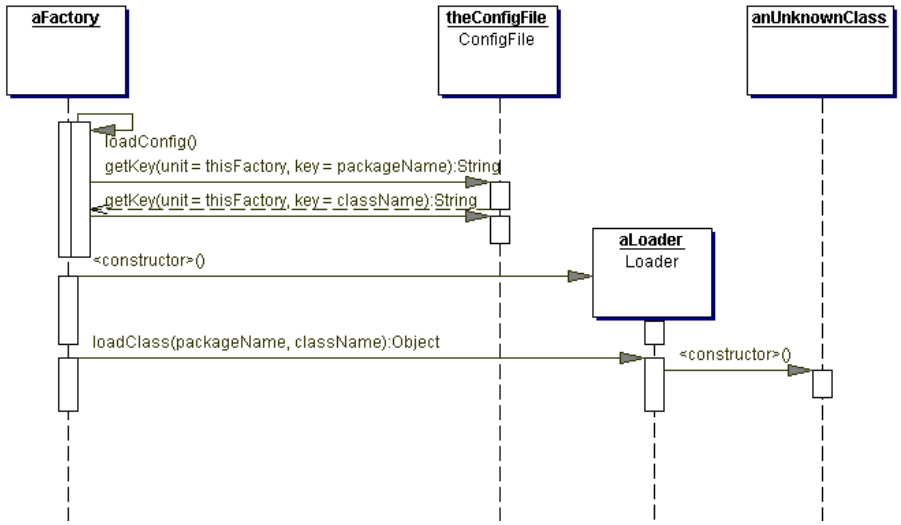


Diagrama de secuencia del cargador dinámico de clases

Podemos ver como hacemos uso de la clase Loader en un supuesto caso práctico, pues cargamos del fichero de configuración, valores procedentes de una unidad, *thisFactory*, de donde cargamos el paquete, packageName y la clase en cuestión, className. El objeto aFactory, representa cualquier instancia de clase, que requiera el uso de la clase Loader.

Funciones para la manipulación de datos en XML

Descripción

Una parte fundamental de este trabajo es el intercambio de objetos y servicios entre nodos mediante mensajes en XML. Para ello, se ha dotado a cada clase que represente información susceptible de ser enviada mediante mensajes, de las funciones para entender su representación en XML, en DOM y para poder generar estas representaciones a partir de su representación interna.

Se han diseñado dos clases para dotar a las clases de esta funcionalidad.

Las clases implicadas en este soporte han sufrido un gran cambio, pues la mayoría de los métodos no estaban implementados, además se han añadido nuevos métodos que aportan una mayor funcionalidad a la gestión de los ficheros XML.

Clases relacionadas

public class **DOMizable**	Representa el conjunto de funciones que una clase debe implementar, para poder generar su representación en *XML*, siguiendo el formato definido en la *DTD x-mip.dtd*, y así poder entender esta representación, tanto *XML* como *DOM,* obteniendo a partir de ellas su propia representación interna.
public class **DOMSource**	Se utilizará para generar una representación *DOM* a partir de un mensaje en formato de texto plano *XML*, y para generar documentos *DOM* que puedan ser utilizados para crear nuevos mensajes en *XML*.

Diagramas relacionados

	Diagrama de clases de las clases relacionadas con la manipulación de datos en formato *XML*.

Modelo estático: diagrama de clases

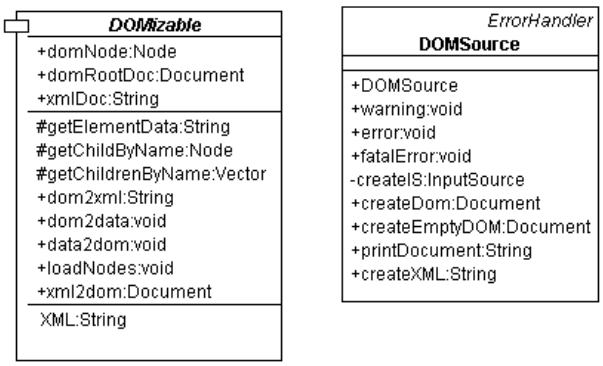


Diagrama de clases de las funciones para la representación de los datos de los nodos en XML

Diccionario de Clases

Métodos de la clase ***DOMSource***

Tipo	Declaración	Acción
public void	warning (SAXParseException e)	

public void	error (SAXParseException e)	
public void	fatalError (SAXParseException e)	
private InputSource	createIS (String message)	
public Document	createDom (String messageXMLString)	
public Document	createEmptyDOM (String namenode, String dtddoc)	
public String	printDocument(Document dom)	
public String	createXML(Document doc)	

Campos de la clase ***DOMizable***

Tipo	Declaración	Acción
public Node	domNode	
public Document	domRootDoc	
public String	xmlDoc	

Métodos de la clase ***DOMizable***

Tipo	Declaración	Acción
protected String	getElementData(Node n)	
protected Node	getChildByName(Node n, String tagName)	
protected Vector	getChildrenByName(Node n, String tagName)	
public String	dom2xml(Document doc)	
public void	dom2data()	
public void	data2dom()	
public void	loadNodes(Node node, Document doc)	
public Document	xml2dom(String xmip)	
public String	getXML()	

4.3 Registro de pruebas unitarias

Se recoge a continuación el registro de las pruebas realizadas.

Prueba	Cargar los atributos provenientes del fichero de configuración NodeConfig.xml
Clase ejecutable	pruebas.utils.AtributosConfig.java
Clases implicadas	REMAssoc, ModelAssoc
Procedimiento	Con esta prueba cargamos el fichero de configuración NodeConfig.xml, y hacemos alguna visualización de los datos obtenidos, probando con alguna clase del framework, como es REMAssoc y ModelAssoc.
Resultado deseado	El fichero de configuración se lee por completo y según su contenido llegamos a cargar alguna de las clases que nos indica.
Resultado obtenido	Se ha obtenido el resultado deseado.
Errores encontrados	Ninguno

Prueba	Creamos una estructura DOM
Clase ejecutable	pruebas.utils.CreateXML
Clases implicadas	DOMSource
Procedimiento	Cargamos una estructura DOM con una serie de valores en cada uno de los nodos que lo forman.
Resultado deseado	Visualización de los datos que hemos introducido en la estructura. Verificando así que esta está bien formada.
Resultado obtenido	El esperado
Errores encontrados	Ninguno

Prueba	Archivo de configuración
Clase ejecutable	pruebas.utils.CreateNodoXREM
Clases implicadas	pruebas.utils.Nodo, utils.DOMSource. (La clase Nodo es la encargada de construir un nodo del documento que hemos creado en la clase CreateNodoXREM
Procedimiento	Con esta prueba cargamos una estructura DOM insertando un nodo a la estructura, para ello utilizamos la clase generada Node. Obtenemos un único documento en el que hemos insertado el nodo.
Resultado deseado	Insertar en una estructura DOM un nuevo elemento. Esto nos servirá para ampliar una estructura DOM añadiendo los elementos que requiramos.

4.3 Registro de pruebas unitarias

Resultado obtenido	Insertar un nodo creado desde otro elemento Document supone el no poder insertar este elemento después en otra estructura, por ello hay que pasar a la clase el objeto Document y crear el nodo en ella directamente. Por ellos necesitamos pasar la clase Document a la clase que genera el nodo, obteniendo una única estructura. No podemos insertar un Nodo creado por otro elemento Document.
Errores encontrados	Ninguno.

Prueba	Generación de identificadores
Clase ejecutable	prueba.utils.ParsingXML
Clases implicadas	import com.ibm.xml.parsers.TXDOMParser,
Procedimiento	Cargamos del fichero Asset.xml en una estructura DOM, y vamos recorriendo los primeros hijos, llamando a una función, recorrido(Node), para obtener los "sub-hijos" de éste.
Resultado deseado	Visualizar todos los valores de los nodos recorriéndolos por nodos.
Resultado obtenido	El esperado.
Errores encontrados	Ninguno.

Prueba	Cargador de clases
Clase ejecutable	prueba.utils.GenerateId
Clases implicadas	utils.Numbers, utils.Number
Procedimiento	Instanciamos la clase Numbers y para un elemento Message y Frame lo que hacemos es generar identificadores probando los distintos métodos que poseemos. Y visualizando los resultados en todo momento.
Resultado deseado	Obtener los identificadores sucesivos de los elementos Message y Frame.
Resultado obtenido	El deseado.
Errores encontrados	No errores.

Prueba	Interfaz XML-DOM
Clase ejecutable	prueba.utils.Dom2xml
Clases implicadas	utils.NodeLog, utils.DOMSource

Procedimiento	Se crea una cadena con la estructura XML, que forma un mensaje en el protocolo. Este mensaje se formará creado de una estructura DOM convirtiendo cada uno de los elementos que forma la estructura en una parte textual del mensaje.
Resultado deseado	Visualizar por pantalla un fichero XML, el creado desde la estructura DOM. En el fichero Log iremos viendo los pasos que se van ejecutando.
Resultado obtenido	El deseado.
Errores encontrados	Ninguno.

Las clases NodeLog y NodeStage, se han probado con las clases obtenidas de la memoria inicial. No olvidemos que todas estas clases darán soporte al desarrollo de las iteraciones siguientes por lo que las prueba de las siguientes iteraciones consolidarán el buen funcionamiento de éstas.

Capítulo 5

Iteración 2: Comunicación

5.1 Requisitos

Número	reqCom01
Descripción	El framework debe ser capaz de enviar y recibir peticiones a través de la red, en forma de mensajes X-MIP.
Prioridad	Alta

Número	reqCom02
Descripción	Se debe garantizar que todos los mensajes enviados a la red, llegan a su destino y lo hacen en el mismo orden en el que fueron enviados.
Prioridad	Alta

Número	reqCom03
Descripción	Las funciones que faciliten el acceso a la red deben diseñarse de forma adaptable y extensible, de manera que puedan añadirse nuevos métodos de acceso al medio, como por ejemplo diferentes tipos de protocolo.
Prioridad	Alta

Número	reqCom04
Descripción	En un principio las comunicaciones entre nodos serán asíncronas, sin que se descarte la utilización de comunicaciones síncronas en un futuro. Por ello, las estructuras que ofrezcan las funciones de acceso al medio tienen que proporcionar los recursos necesarios para que el framework obtenga un canal de comunicación, sin que estas estructuras queden bloqueadas, pudiendo generar nuevos recursos. Es decir, una vez generado un canal de comunicación, las estructuras que lo han proporcionado, no necesitan encargase de mantenerlo.
Prioridad	Alta

Número	reqCom05
Descripción	Se deben poder añadir funciones de filtrado, como compresión, encriptación, etc., en el envío y recepción de mensajes, con el propósito de mejorar la eficiencia, versatilidad y seguridad del protocolo.
Prioridad	Media

5.1 Requisitos

Número	reqCom06
Descripción	Las funciones de filtrado de mensajes, deben diseñarse de manera extensible y adaptable para que puedan añadirse nuevas funcionalidades al protocolo, como por ejemplo, el formatear mensajes en HTML para mostrarlos resultados de una petición mediante una interfaz web.
Prioridad	Alta

Número	reqCom07
Descripción	Las diferentes funciones de filtrado que se aplicarán a los mensajes tanto a su recepción como a su envío, deben poder ser configurables mediante el fichero de configuración.
Prioridad	Media

Número	reqCom08
Descripción	Las diferentes funciones de filtrado que se aplicarán a los mensajes tanto a su recepción como a su envío, deben poder ser configurables mediante el intercambio de mensajes de negociación entre nodos.
Prioridad	Baja

Número	reqCom09
Descripción	Las diferentes funciones de filtrado que se aplicarán a los mensajes tanto a su recepción como a su envío, deben poder activadas automáticamente para el procesamiento correcto de cada mensaje en función de la codificación que presente.
Prioridad	Baja

5.2 Diseño

Conexiones entre nodos a nivel de red (*connection* y *tcpSokectConnection*)

Descripción

Para soportar las funciones relacionadas con el acceso a la red de los nodos, se han diseñado un conjunto de formatos genéricos que se utilizarán como canales de comunicación y un conjunto de funcionalidades para generar estos canales.

Como canales de comunicación se han establecido las clases Java java.io.InputStream y java.io.OutputStream, para los canales de recepción y de envío de datos respectivamente. Se han seleccionado estas clases debido a que representan el tipo de canal de comunicación más básico y genérico posible que ofrece Java. Estos canales ofrecen funciones genéricas de flujos de lectura y escritura de bytes, independientemente del tratamiento que se les dé posteriormente a los datos y de que se utilicen para desarrollar otros canales de comunicación más complejos o específicos para un medio concreto.

Para proporcionar un canal de comunicación a través de la red utilizando los canales genéricos, se han desarrollado una serie de clases que emplean conexiones basadas en sockets y en el protocolo TCP. La utilización de TCP garantiza un transporte fiable de los mensajes a través de la red.

Esta es la iteración que menos cambios ha sufrido. La mayoría de los cambios se producen para acoplar las modificaciones que se han producido las clases en la iteración anterior, para controlar aún mas comprobaciones de uso de recursos en la comunicación.

Diagrama de paquetes

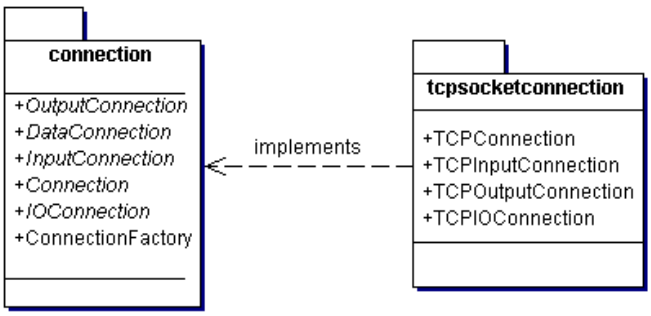


Diagrama de paquetes del soporte de conexión a la red

Diagramas relacionados

	Clases genéricas para las funciones de comunicación a través de la red, del paquete Connection.

	Clases genéricas para las funciones de comunicación a través de la red, del paquete Connection.
	Clases que implementan las funciones de conexión mediante *TCP* y *sockets*.
	Secuencia de establecimiento de un canal de comunicación.

Clases relacionadas: Connection

abstract public class **DataConnection**	Clase genérica que representa un canal de comunicación de datos.
abstract public class **InputConnection** extends DataConnection	Clase genérica que representa un canal de comunicación de entrada de datos.
abstract public class **OutputConnection** extends DataConnection	Clase genérica que representa un canal de comunicación de salida de datos.
abstract public class **IOConnection** extends DataConnection	Clase genérica que representa un canal de comunicación simultáneamente de entrada y salida de datos.
public interface **Connection**	Clase genérica que define las funciones genéricas para crear canales de comunicación de datos.
public class **ConnectionFactory**	Clase de patrón “Factory”, que permite obtener una instancia de las clases concretas que implementan los canales de comunicación y las funciones definidas por las clases genéricas.

..

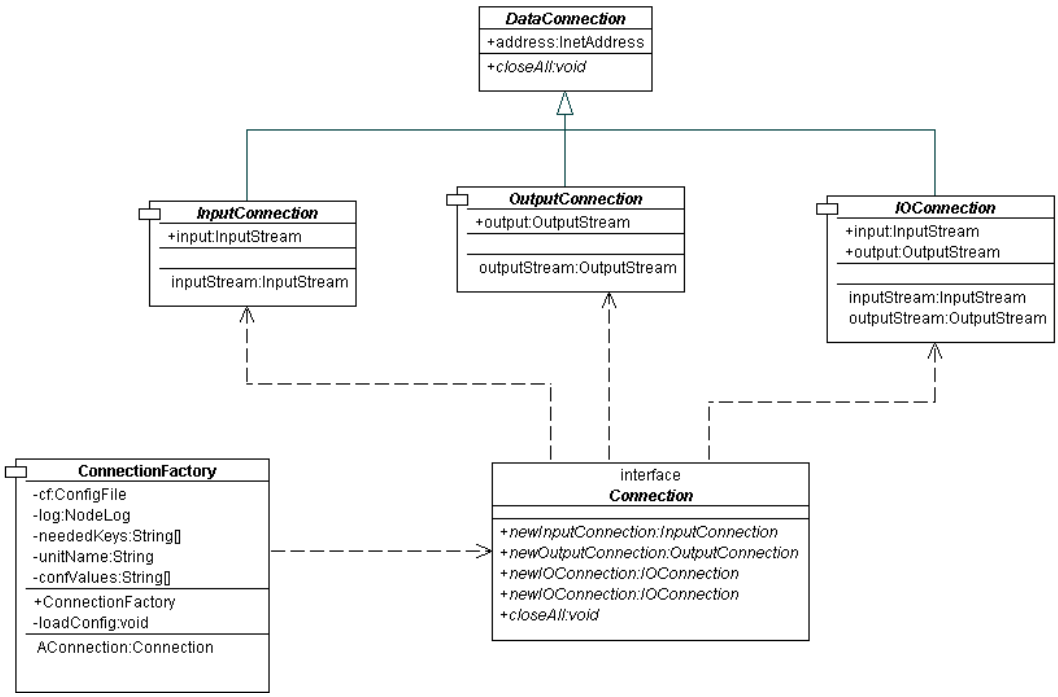


Diagrama de clases de las clases genéricas para crear canales de comunicación

Diccionario de Clases: connection

Campos de la clase ***DataConnection***

Tipo	Declaración	Acción
public InetAddress	address	

Métodos de la clase ***DataConnection***

Tipo	Declaración	Acción
public abstract void	closeAll()	

Campos de la clase ***InputConnection***

Tipo	Declaración	Acción
public InputStream	input	

*Métodos de la clase **InputConnection***

Tipo	Declaración	Acción
public InputStream	getInputStream()	

*Campos de la clase **OutputConnection***

Tipo	Declaración	Acción
public OutputStream	output	

*Métodos de la clase **OutputConnection***

Tipo	Declaración	Acción
public OutputStream	getOutputStream()	

*Campos de la clase **IOConnection***

Tipo	Declaración	Acción
public InputStream	input	
public OutputStream	output	

*Métodos de la clase **IOConnection***

Tipo	Declaración	Acción
public InputStream	getInputStream()	
public OutputStream	getOutputStream()	

*Métodos de la clase **Connection***

Tipo	Declaración	Acción
public InputConnection	newInputConnection(int port)	
public OutputConnection	newOutputConnection(String address, int port)	
public IOConnection	newIOConnection(int port)	
public IOConnection	newIOConnection(String address, int port)	
public void	closeAll()	

*Campos de la clase **ConnectionFactory***

Tipo	Declaración	Acción
private ConfigFile	cf	
private NodeLog	log	
private String[]	neededKeys	
private String	unitName	

private String[]	confValues	

Constructores de la clase ***ConnectionFactory***

Tipo	Declaración	Acción
public	ConnectionFactory()	

Métodos de la clase ***ConnectionFactory***

Tipo	Declaración	Acción
private void	loadConfig()	
public Connection	getAConnection()	

Clases relacionadas: TcpSokectConnection

public class **TCPInputConnection** extends **InputConnection**	Clase que implementa un canal de comunicación de salida basado en sockets y el protocolo TCP.
public class **TCPOutputConnection** extends **OutputConnection**	Clase que implementa un canal de comunicación de salida basado en sockets y el protocolo TCP.
public class **TCPIOConnection** extends **IOConnection**	Clase que implementa un canal de comunicación de entrada y salida basado en sockets y el protocolo TCP.
public class **TCPConnection** implements **Connection**	Clase que implementa las funciones que general canales de comunicación basados en sockets y el protocolo TCP.

5.2 Diseño

Modelo estático: diagrama de clases tcpSokectConnection

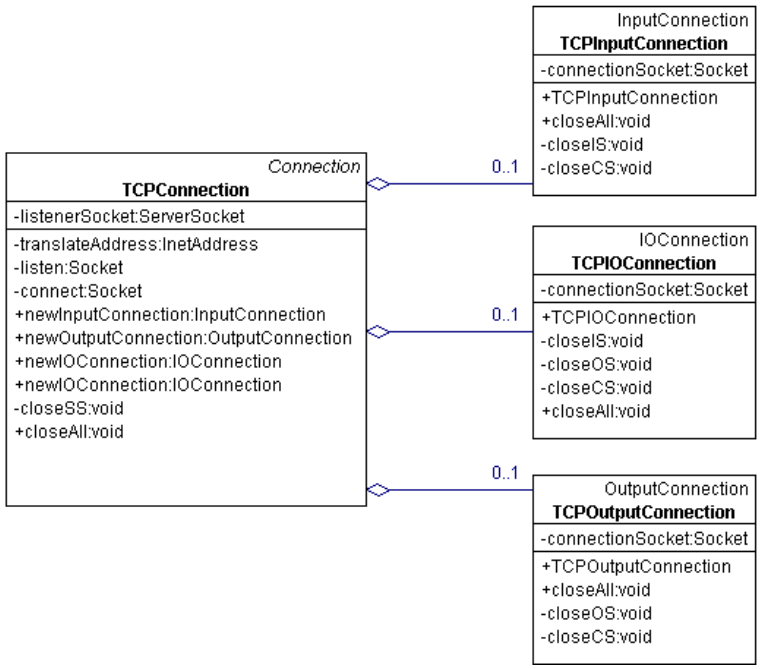


Diagrama de las clases que implementan las funciones genéricas de conexión mediante TCP y sockets.

Diccionario de Clases tcpSokectConnection

Campos de la clase ***TCPInputConnection***

Tipo	Declaración	Acción
private Socket	connectionSocket	

Constructores de la clase ***TCPInputConnection***

Tipo	Declaración	Acción
public	TCPInputConnection(Socket aSocket)	

Métodos de la clase ***TCPInputConnection***

Tipo	Declaración	Acción
public void	closeAll()	
private void	closeIS()	
private void	closeCS()	

Campos de la clase ***TCPOutputConnection***

Tipo	Declaración	Acción
private Socket	connectionSocket	

Constructores de la clase ***TCPOutputConnection***

Tipo	Declaración	Acción
public	TCPOutputConnection(Socket aSocket)	

Métodos de la clase ***TCPOutputConnection***

Tipo	Declaración	Acción
public void	closeAll()	
private void	closeOS()	
private void	closeCS()	

Campos de la clase ***TCPIOConnection***

Tipo	Declaración	Acción
private Socket	connectionSocket	

Constructores de la clase ***TCPIOConnection***

Tipo	Declaración	Acción
public	TCPIOConnection(Socket aSocket)	

Métodos de la clase ***TCPIOConnection***

Tipo	Declaración	Acción
private void	closeIS()	
private void	closeOS()	
private void	closeCS()	
public void	closeAll()	

Campos de la clase ***TCPConnection***

Tipo	Declaración	Acción
private ServerSocket	listenerSocket	

Métodos de la clase ***TCPConnection***

Tipo	Declaración	Acción
private InetAddress	translateAddress(String host)	
private synchronized	listen(int port)	

Socket		
private Socket	connect(String address, int port)	
public InputConnection	newInputConnection(int port)	
public OutputConnection	newOutputConnection(String address, int port)	
public IOConnection	newIOConnection(int port)	
public IOConnection	newIOConnection(String address, int port)	
private void	closeSS()	
public void	closeAll()	

Diagrama secuencia

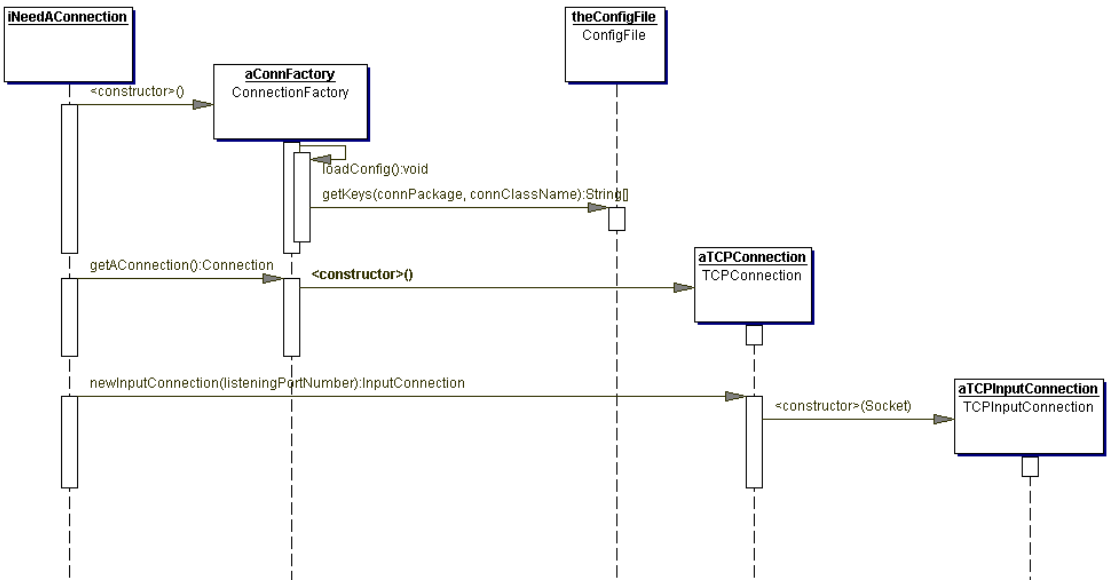


Diagrama de secuencia de la creación de un canal de comunicación para recibir mensajes, que escucha por un determinado puerto.

Todas las secuencias de creación de canales de comunicación son idénticas a esta, a excepción de que cambia la clase que implementa el canal, dependiendo del tipo que se ha solicitado.

Filtros de codificación de mensajes *Controlador*, *GZIP* y *ZIP*

Descripción

Para diseñar el soporte a la funcionalidad de filtrado de mensajes, se ha tomado la idea, que utilizan las clases Java que heredan de java.io.FilterInputStream y java.io.FilterOutputStream, para añadir diversas funciones de procesamiento, a los flujos de entrada y salida respectivamente.

Se ha separado, al igual que ocurría en el caso de las clases relacionadas con las conexiones entre nodos a través de la red, las clases genéricas, que especifican como se aplican las funcionalidades de filtrado, de las clases que realmente realizan las funciones de filtrado.

Así mismo, se han contemplado dos modos de filtrado:

- Uno de configuración estática, que establece los filtros y las fases a aplicar, a partir de la información de los ficheros de configuración o de la información intercambiada entre dos nodos en mensajes de negociación.
- Otro de configuración dinámica, que detecta, a partir de información adicional presente en los datos intercambiados, el formato de codificación que se ha empleado en un mensaje. Además, a la hora de enviar un mensaje, si el nodo dispone de información suficiente acerca de qué filtros acepta el nodo con el que se intenta comunicar, la configuración dinámica permite que se seleccionen los filtros que ofrezcan mayor seguridad y rendimiento.

De momento, aunque se han planteado todos los modos de configuración de los filtros, sólo se ha implementado la configuración a partir del fichero de configuración. Debido a esto, dos nodos que deseen comunicarse entre sí, deberán utilizar la misma configuración de filtros, señalándola en sus ficheros de configuración.

Se han planteado dos tipos de filtros, de encriptación y de compresión, aunque se pueden añadir nuevos tipos en desarrollos posteriores. De estos dos tipos de filtros, se han implementado dos de compresión, utilizando los filtros que proporciona Java:

java.util.zip.GZIPInputStream, java.util.zip.GZIPOutputStream, java.util.zip.ZipInputStream y java.util.zip.ZipOutputStream.

Se han adaptado los métodos que utilizaban algún cambio de las iteraciones anteriores, se han implementado métodos vacíos y se ha construido la clase ZIP marcando unas pautas de implementación:

Forma de compresión

Las consideraciones de implementación que se han tenido presente son:

1. El canal siempre irá comprimido en una entrada (*java.util.zip.ZipEntry*) y en la cual introduciremos el nombre.
    - getName(), setName()
2. Solo habrá una única entrada cuando mandemos el fichero. Por el momento no habrá más.

El fichero Zip esta probado en la clase de prueba Zipp.java, aquí vemos como leemos un fichero y lo comprimimos, para posteriormente hacer la operación inversa.

Diagrama de Paquetes

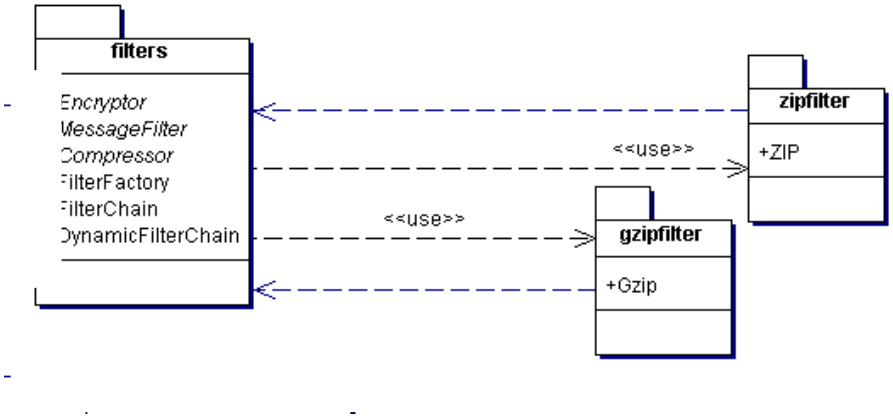


Diagrama de paquetes de filtros de mensajes

Diagramas relacionados

	Diagrama de clases de filtrado de mensajes.
	Diagrama de clases del filtro *GZIP*.
	Diagrama de secuencia de la creación de un filtro y de su utilización en una cadena de filtros de configuración estática, con el objetivo de descomprimir los datos de un canal de entrada.
	Diagrama de clases del filtro ZIP.

Clases relacionadas Controlador

abstract public class **MessageFilter**	Clase genérica que representa a un filtro.
abstract public class **Encryptor** extends **MessageFilter**	Clase genérica que representa a un filtro de compresión. No contiene métodos ni atributos. De momento sólo se utiliza para clasificar los filtros por tipos.
abstract public class **Compressor** extends **MessageFilter**	Clase genérica que representa a un filtro de encriptación. No contiene métodos ni atributos. De momento sólo se utiliza para clasificar los filtros por tipos.
public class **FilterChain**	Clase que implementa las funciones necesarias para establecer una cadena de filtros para procesar los datos recibidos y enviados a través de canales de comunicación. Contiene los métodos para aplicar filtros utilizando configuración estática. Construye los filtros necesarios y genera canales de comunicación ya procesados mediante estos.
public class **FilterFactory**	Clase de patrón '*Factory*' que obtiene una instancia de una clase concreta, que implementa un filtro de alguno de los tipos definidos.

public class **DynamicFilterChain** implements **Runnable**	Implementa la configuración dinámica de filtros. Es utilizada por la clase *FilterChain*, cuando esta necesita soporte para la configuración automática de filtros. La clase implementa la interfaz ***Runable***, para poder ser ejecutada como un proceso independiente. Esto es necesario para poder realizar el filtrado dinámico sin bloquear el canal de datos. De otro modo, la clase debería primero recibir y filtrar el mensaje completo para luego poder ofrecerlo por el canal de datos en forma de texto plano. Al lanzarse el filtro en un proceso concurrente se puede ir ofreciendo el mensaje en texto plano por el canal de datos a medida que se van aplicando los filtros "***byte*** a ***byte***".

Modelo estático: diagrama de clases *Controlador*

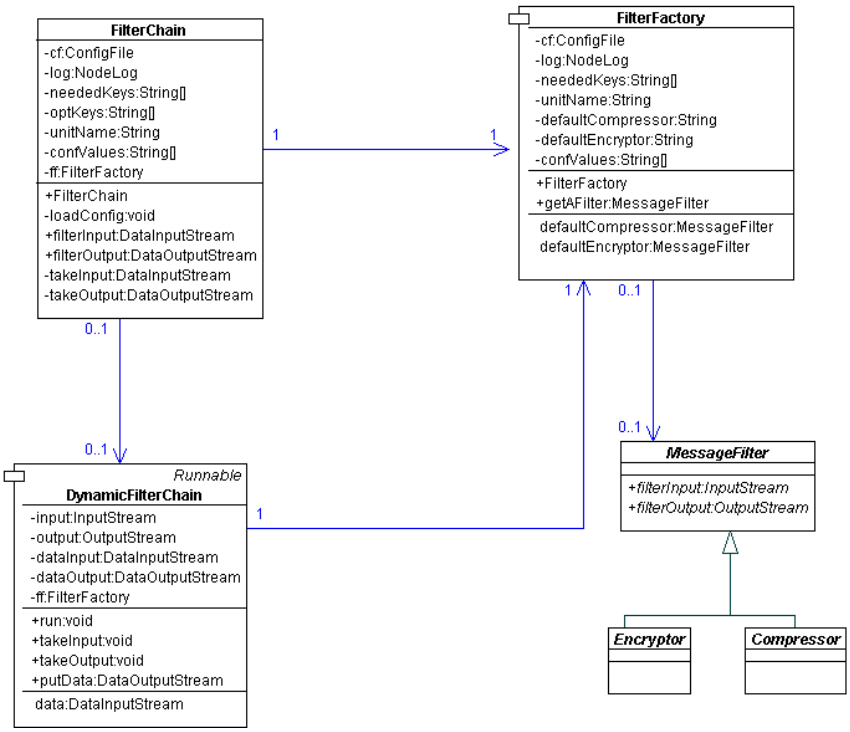


Diagrama de las clases de filtrado de mensajes

Diccionario de Clases *Controlador*

Métodos de la clase ***MessageFilter***

Tipo	Declaración	Acción
public abstract InputStream	filterInput(InputStream input)	
public abstract	filterOutput(OutputStream output)	

OutputStream		

Campos de la clase ***FilterChain***

Tipo	Declaración	Acción
private ConfigFile	cf	
private NodeLog	log	
private String[]	neededKeys	
private String[]	optKeys	
private String	unitName	
private String[]	confValues	
private FilterFactory	ff	

Constructores de la clase ***FilterChain***

Tipo	Declaración	Acción
public	FilterChain()	

Métodos de la clase ***FilterChain***

Tipo	Declaración	Acción
private void	loadConfig()	
public DataInputStream	filterInput(InputStream input)	
public DataOutputStream	filterOutput(OutputStream output)	
private DataInputStream	takeInput(InputStream input)	
private DataOutputStream	takeOutput(OutputStream output)	

Campos de la clase ***FilterFactory***

Tipo	Declaración	Acción
private ConfigFile	cf	
private NodeLog	log	
private String[]	neededKeys	
private String	unitName	
private String	defaultCompressor	
private String	defaultEncryptor	
private String[]	confValues	

Constructores de la clase ***FilterFactory***

Tipo	Declaración	Acción
public	FilterFactory()	

Métodos de la clase ***FilterFactory***

Tipo	Declaración	Acción
public MessageFilter	getAFilter(String filterID)	
private MessageFilter	getDefaultCompressor()	
private MessageFilter	getDefaultEncryptor()	

Campos de la clase ***DynamicFilterChain***

Tipo	Declaración	Acción
private InputStream	input	
private OutputStream	output	
private DataInputStream	dataInput	
private DataOutputStream	dataOutput	
private FilterFactory	ff	

Métodos de la clase ***DynamicFilterChain***

Tipo	Declaración	Acción
public void	run()	
public void	takeInput(InputStream input)	
public void	takeOutput(OutputStream output)	
public DataInputStream	getData()	
public DataOutputStream	putData()	

Clases relacionadas GZIP

public class **Gzip** extends **Compressor**	Implementación de un filtro de compresión que utiliza GZIP.

Modelo estático: diagrama de clases GZIP

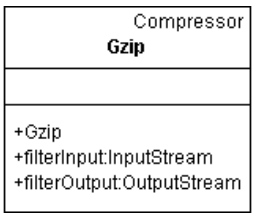


Diagrama de clases del filtro GZIP

Diccionario de Clases GZIP

Métodos de la clase ***Gzip***

Tipo	Declaración	Acción
public InputStream	filterInput(InputStream input)	
public OutputStream	filterOutput(OutputStream output)	

Diagrama de Secuencia

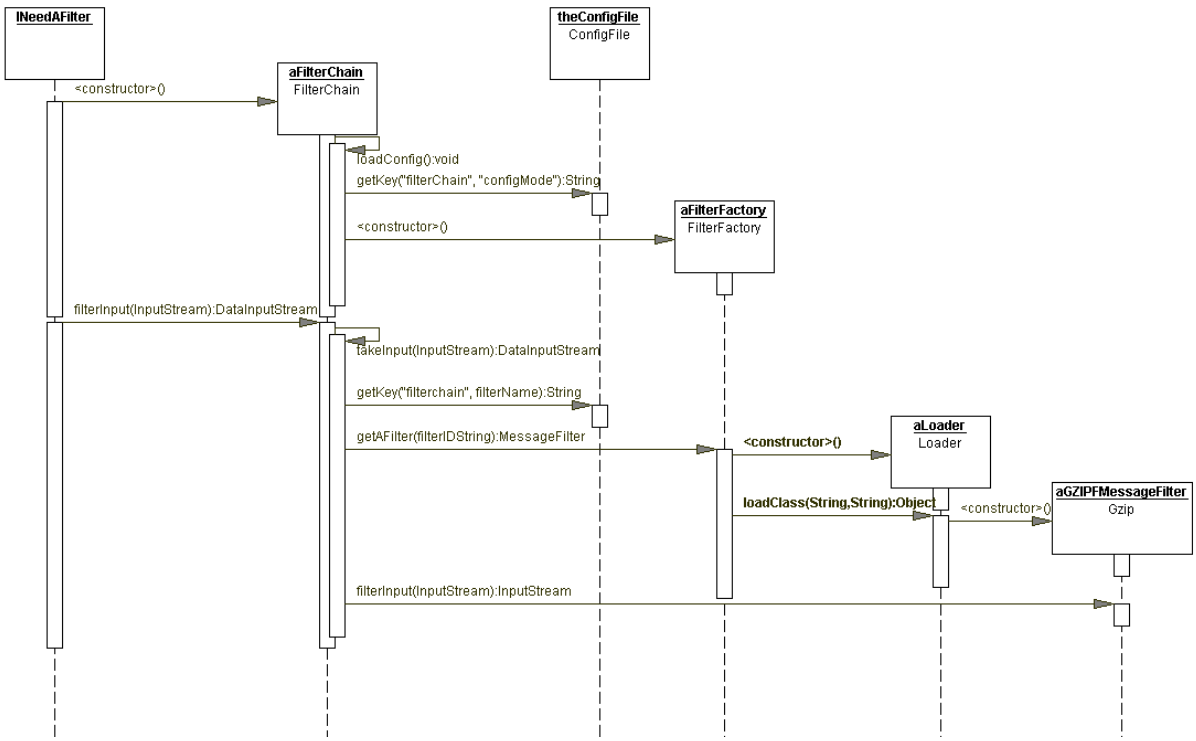


Diagrama de secuencia de la creación de un filtro (GZIP).

Vemos como creamos un filtro y su utilización en una cadena de filtros de configuración estática, con el objetivo de descomprimir los datos de un canal de entrada.

Clases relacionadas ZIP

public class **ZIP** extends **Compressor**	Implementación de un filtro de compresión que utiliza ZIP.

Modelo estático: diagrama de clases ZIP

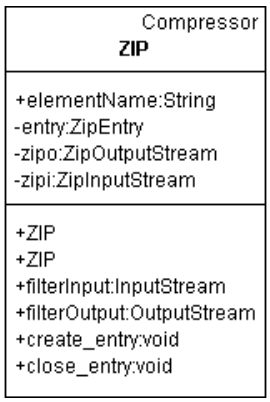


Diagrama de clases del filtro ZIP.

Diccionario de Clases ZIP

*Campos de la clase **ZIP***

Tipo	Declaración	Acción
public String	elementName	
private ZipEntry	entry	
private ZipOutputStream	zipo	
private ZipInputStream	zipi	

*Constructores de la clase **ZIP***

Tipo	Declaración	Acción
public	ZIP (String name)	
public	ZIP ()	

*Métodos de la clase **ZIP***

Tipo	Declaración	Acción
public InputStream	filterInput(InputStream input)	
public OutputStream	filterOutput(OutputStream output)	
public void	create_entry(String name)	
public void	close_entry()	

5.3 Registro de pruebas unitarias

Prueba	Prueba de las conexiones
Clase ejecutable	testConnection
Clases implicadas	utils.NodeLog, utils.ConfigFile connection.Connection, connection.ConnectionFactory, connection.InputConnection, connection.OutputConnection, connection.DataConnection, tcpsocketconnection.TCPConnection, tcpsocketconnection.TCPInputConnection, tcpsocketconnection.TCPOutputConnection
Procedimiento	Se utilizan dos clases ejecutándose en procesos concurrentes. Una de ellas actuará como servidor y la otra como cliente. La clase que actúa como servidor, utiliza la clase ConnectionFactory para obtener una instancia una conexión. La clase que implementa la conexión, así como los parámetros para configurarla, se obtienen del archivo de configuración. A continuación se lanza la clase cliente, que utiliza también la clase ConnectionFactory para crear una conexión, pero esta vez de salida (TCPOutputConnection), para conectarse con la clase servidor. A continuación, carga un fichero XML (sin obtener su representación DOM), y se lo envía como una cadena de texto a la clase servidor. La clase servidor, atiende la conexión entrante, lee el fichero enviado y almacena en el sistema de ficheros, una copia con otro nombre.
Resultado deseado	Se debe obtener una copia del fichero empleado.
Resultado obtenido	Se ha obtenido la copia del fichero.
Errores encontrados	Ninguno

Prueba	Filtros de codificación de mensajes.
Clase ejecutable	filterTest

Clases implicadas	utils.NodeLog, utils.ConfigFile connection.Connection, connection.ConnectionFactory, connection.InputConnection, connection.OutputConnection, connection.DataConnection, tcpsocketconnection.TCPConnection, tcpsocketconnection.TCPInputConnection, tcpsocketconnection.TCPOutputConnection, filter.FilterChain, filter.FilterFactory, filter.MessageFilter, filter.Compressor, gzipfilter.gzip
Procedimiento	Se ha seguido un procedimiento similar al caso de prueba anterior, pero antes de enviar y recibir el fichero, se ha utilizado una cadena de filtros FilterChain. La cadena se ha preparado según las opciones del fichero de configuración, para utilizar un sólo filtro de compresión/descompresión: Gzip. El fichero se ha comprimido al enviar y se ha descomprimido al recibir.
Resultado deseado	Se debe obtener una copia del fichero.
Resultado obtenido	Se ha obtenido la copia del fichero esperada.
Errores encontrados	Ninguno.

Prueba	Filtros de codificación de mensajes.
Clase ejecutable	zipp
Clases implicadas	zipfilter.ZIP
Procedimiento	Utilizaremos el método de compresión utilizado por la clase ZIP. Para ello utilizaremos un canal de salida sobre el que iremos introduciendo los datos que vamos a comprimir, estos datos serán los obtenidos de otro fichero. Al mismo tiempo una vez que hemos finalizado la compresión descomprimiremos para ver si obtenemos los mismos datos que los originales. Para ello volvemos a crear un canal de entrada con el fichero comprimido obteniendo los datos que deseamos.
Resultado deseado	Visualizar el contenido del fichero que hemos comprimido en la primera etapa. Al mismo tiempo hemos obtenido el fichero ZIP con la entrada de la primera etapa por lo que podremos descomprimirlo con cualquier aplicación de compresión y ver si está bien construido.
Resultado obtenido	Se ha obtenido la copia del fichero esperada.
Errores encontrados	Ninguno.

Capítulo 6

Iteración 3: Datos estructurados como modelos

6.1 Requisitos

Número	reqMod01
Descripción	Se necesita poder representar datos interconectados entre sí, cuya estructura está definida en diagramas de clases UML. La naturaleza de estos datos requiere que se puedan representar tanto los propios tipos de datos como las relaciones que existan entre ellos.
Prioridad	Alta

Número	reqMod02
Descripción	Los conjuntos de datos que formen un modelo tienen que poder ser representados como grafos, dónde los vértices serán los objetos de los modelos, y los arcos serán las asociaciones entre los objetos. Se debe ofrecer una interfaz para crear y acceder a los modelos basado en operaciones y elementos de grafos.
Prioridad	Alta

Número	reqMod03
Descripción	El protocolo y por lo tanto también los nodos, deben reconocer dos grandes tipos de datos estructurados como modelos, los datos de gestión del protocolo y los datos pertenecientes a un modelo de reutilización.
Prioridad	Alta

Número	reqMod04
Descripción	Además de incluir todos los elementos de gestión del protocolo que puedan intercambiarse con otros nodos, constituyendo modelos, se podrán integrar otras representaciones de este tipo de datos que sean útiles para el nodo. En concreto, existen elementos utilizados en el protocolo, y no integrados en modelos, como los elementos Identiy, NodeAddress, que forman parte de los encabezados MIPHeader. Estos datos deben poder ser utilizados mediante las dos representaciones: elementos aislados, y elementos integrables en un modelo.
Prioridad	Alta

6.1 Requisitos

Número	reqMod05
Descripción	El modelo de componente reutilizable, debe representarse de forma que se puedan separar las funcionalidades básicas de los aspectos estrictamente únicos del modelo concreto utilizado. De esta forma se podrán utilizar otros modelos de componente reutilizable además del escogido para este trabajo. Se debe independizar todo lo posible, la funcionalidad relacionada con la manipulación de estos datos de la información relacionada con el modelo particular (como las clases, atributos y asociaciones que definen el modelo concreto).
Prioridad	Alta

Número	reqMod06
Descripción	Al construir un modelo, tanto directamente como a partir de su representación XML, se debe poder utilizar la definición del modelo, en concreto la especificación de las asociaciones permitidas en él, para comprobar su validez.
Prioridad	Alta

Número	reqMod07
Descripción	Las estructuras desarrolladas tienen que ser capaces de entender su propia representación en XML y generar su representación interna a partir de esta.
Prioridad	Alta

Número	reqMod08
Descripción	Descripción Las estructuras desarrolladas tienen que ser capaces de generar a partir de su representación interna, su representación en XML.
Prioridad	Alta

Número	reqMod09
Descripción	Descripción El formato para la representación en XML será el definido en las DTD's x-rem.dtd y xmd.dtd, para los modelos de elementos reutilizables y de datos de gestión del protocolo respectivamente.
Prioridad	Alta

6.2 Diseño

6.2.1 Soporte de datos estructurados *ModelData*, *ProtocolData*, *Mecano*

Para dar soporte a los datos que pueden ser representados como modelos, se han diseñado primero las estructuras genéricas. Según los requisitos para desarrollar esta parte, todos los modelos y los objetos que pertenecen a estos tienen que poder ser accedidos como si de grafos se tratase. Es decir se debe diseñar una estructura de datos basada en grafos y se debe separar la naturaleza de estos datos, del contenido que se define en cada modelo concreto.

En las DTD's que definen cada uno de los dos tipos de modelos, se puede comprobar que el formato de los modelos de componentes reutilizables, es un súper conjunto del modelo de datos de gestión del protocolo. Debido a esto, se ha desarrollado una estructura genérica que es utilizada directamente por el modelo de datos de gestión, y otro tipo de estructura que hereda de la primera y que ofrece soporte para los modelos de reutilización, más complejos. Se pueden consultar las DTD's en los apéndices A y B, y en el Apéndice C se define se define el entorno de desarrollo con los cambios realizados después de esta evolución.

Para las estructuras de datos basadas en grafos, se ha utilizado una librería de estructuras de datos en Java: JDSL [jds]. Se trata de una librería de libre acceso si no se emplea en proyectos comerciales. Está constituida por una colección de interfaces y clases que implementan estructuras de datos y algoritmos fundamentales como:

- Secuencias, árboles, colas de prioridad, árboles de búsqueda, tablas hash, ...
- Algoritmos de búsqueda y ordenación
- Grafos
- Algoritmos de recorrido de grafos, de búsqueda de caminos, etc.

Se describen a continuación los diferentes elementos que componen las estructuras genéricas para tratar los datos de tipo 'modelo' como grafos:

Representación de los objetos de un modelo

Para representar los objetos de un modelo, se ha tenido en cuenta que lo único que diferencia a unos objetos de otros (independientemente del tipo de modelo) es el listado de atributos que se definen para cada clase que existe en el modelo. De este modo, se ha diseñado un elemento genérico que representa a todos los objetos, y un contenedor para albergar a atributos, independientemente del tipo de estos. De este modo, se puede ofrecer una interfaz para acceder a los atributos de un objeto de forma homogénea, independientemente de su tipo, nombre o número.

```
protected   Attributes              attribs;
protected   String[]                attribSet;
```

Como se puede observar, se utilizan dos elementos:

- Un contenedor genérico para almacenar los atributos y sus valores: attribs.
- Un Array de Strings que se utilizará para declarar en cada objeto, los atributos definidos por el modelo concreto para esa clase.

Por ejemplo, en el caso de una clase que representa a un Asset, se inicializará esta lista en la declaración de los atributos de la clase.

```
public String[] attribSet = { "Identifier" , "Name" , "Abstract",
      "CreationDate ", "ModificationDate", "KeyWords",
      "Environment", "Constraints", "SecurityLevel",
       "Cost", "Version", "CertifyingLevel",
```

```
        "RetrievingCount", "Comments","InsertionDate",
        "AbstractionLevel" , "Methodology ", "Language" } ;
```

Finalmente, se debe mencionar que para que estos objetos puedan 'comprender' y generar también su representación en X-MIP, contienen los atributos que poseen todos los objetos del protocolo:

```
public boolean          opTarget ;
public String           objID;
```

Representación de las asociaciones de un modelo

La definición de una estructura para las asociaciones responde a la necesidad de dar soporte sintáctico a las relaciones entre objetos. En los modelos concretos no será necesario definir ningún elemento de tipo asociación, sino solamente especificar cuáles son las asociaciones permitidas en ese modelo. Se define entonces una estructura para las asociaciones que pueda manejarse, insertarse en grafos, etc. al igual que los objetos. Además esto facilitará la traducción de modelos entre su representación interna y su representación en X-MIP. Las asociaciones genéricas contienen aquellos atributos que poseen las asociaciones de los dos grandes tipos de modelo en X-MIP:

```
public      boolean          opTarget ;
public      Strings          ObjLink ;
public      String           tObjLink ;
```

Especificación de las asociaciones existentes en el modelo

Las asociaciones se pueden definir de modo sencillo utilizando esta estructura, que además servirá para verificar la validez de una asociación cuando esta vaya a insertarse en un modelo.

Se define para ello una operación sencilla que se deberá utilizar para crear las definiciones de las asociaciones existentes en el modelo:

```
protected void addAssoc(String source,
            String target, String cardinality, String label)
```

dónde:

source es el nombre de la clase origen de la asociación.

target es el nombre de la clase destino de la asociación.

cardinality es la cardinalidad máxima del extremo destino de la asociación. Podrá ser "*" o cualquier número entero mayor que cero, aunque en los mensajes sólo se soportan las cardinalidades "1" y "*".

label etiqueta que se utiliza para identificar la asociación, si entre esos dos elementos existen varias diferentes. Se utilizará como etiqueta en primer lugar la etiqueta de la asociación. Si no la hay, se utilizará el rol de la clase origen, y si tampoco lo hubiera, se utilizará el rol de la clase destino.

Las definiciones de asociaciones se almacenarán en la clase para poder ser consultadas posteriormente. Para crear las definiciones, en cada modelo concreto se definirá una clase que herede de modeldata.AssocRules con un único método, el constructor. En este método, se incluirá las operaciones para crear las definiciones de las asociaciones del modelo utilizando la operación mencionada. Se muestra como ejemplo, como se crea la definición de algunas asociaciones en el modelo de Mecano:

```
public MecanoAssocRules ( ) {
addAssoc ( "Asset" , "Author" , "*" , " " ) ;
```

```
addAssoc ( "Asset" , "Representation " , "*" , " " ) ;
addAssoc ( "Asset" , "Mecano" , "*" , " " ) ;
addAssoc ( "Asset" , "AssetType" , "1" , " " ) ;
addAssoc ( "Asset" , "Relationship" , "*" , "Source" ) ;
addAssoc ( "Asset" , "Relationship" , "*" , "Target" ) ;
. . . . .
}
```

Representación de un modelo

Por último, y para representar un modelo, se utiliza la clase Model, que ofrece la interfaz de grafos para crear y acceder a los conjuntos de datos estructurados. Contiene operaciones para insertar objetos y asociaciones en el grafo del modelo y también para recorrer el grafo. De momento no se ha implementado ningún algoritmo de recorrido de grafos, aunque no se descarta hacerlo en un futuro, si se estima necesario. Para acceder a los elementos de un modelo, se proporcionan operaciones simples que devuelven:

- Todos los objetos a los que no llega ninguna asociación.
- Todos los objetos de los que no sale ninguna asociación.
- Los objetos origen y destino de una asociación.
- Las asociaciones que llegan y salen de un objeto.
- ...

En general, con este conjunto de operaciones se podrá recorrer el grafo, decidiendo el modo de exploración que más convenga, según la operación a realizar sobre el modelo. La clase actúa también como un contenedor de los objetos y asociaciones que existen en un modelo concreto...

Para soportar los modelos de componentes reutilizables, se han definido un conjunto de clases que heredan de las genéricas. La única necesidad es poder añadir los atributos adicionales que los modelos de componentes reutilizables presentan en X-MIP. Junto con estos atributos se han añadido las operaciones relacionadas.

Además, se ha diseñado una clase de patrón 'Factory' REMFactory, que permita instanciar modelos de reutilización concretos. Para configurar el modelo que utilizará el nodo, se empleará el fichero de configuración. La clase REMFactory, cargará el modelo que se haya indicado en el fichero de configuración.

En las siguientes iteraciones las clases existentes sufrirán un gran cambio, pues se implementarán la mayoría de los métodos que antes sólo figuraban.

En esta iteración conseguiremos instanciar el grafo con la estructura del modelo, tanto para XMD como para el X-REM, incluso se llegará a crear nuevas clases para facilitar la estructuración de los datos; tendremos almacenados en memoria todos y cada uno de los objetos que constituyen cualquiera de los modelos que utilicemos y que provienen de un mensaje, y podremos construir un mensaje a partir de los objetos que tenemos en memoria; no olvidemos que siempre hay que tener en cuenta los cambios que se han realizado en las iteraciones anteriores.

Para empezar, se crearán una serie de clases, que como ya hemos indicado, nos indicarán las generalidades de los elementos de los modelos, X-REM y XMD.

Diagrama de paquetes

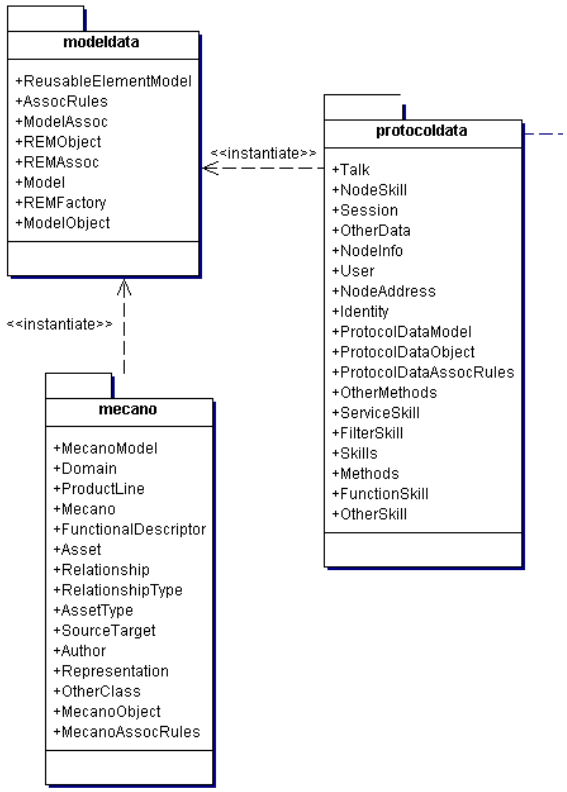


Diagrama de paquetes del soporte para datos estructurados como modelos.

Diagramas relacionados

	Diagrama de clases de las estructuras genéricas para representar datos de tipo “modelo”, con una interfaz de grafos.
	Diagrama de secuencia del proceso de insertar dos objetos y una asociación entre ellos, en un modelo.
	Diagrama de clases de la representación del modelo de Mecano
	Diagrama de secuencia de la creación de un modelo de MECANO a partir de su representación DOM.
	Diagrama de secuencia de la creación del grafo de un modelo de MECANO partir de su representación DOM, convirtiéndose los elementos a su representación interna.
	Diagrama de secuencia de la creación de un nuevo modelo de Mecano vacío
	Diagrama de secuencia de la creación de un grafo de un modelo de Mecano, creando objetos y asociaciones e insertándolos en el grafo.
	Diagrama de la secuencia de la conversión de un modelo de

	Mecano y los elementos que contiene a su representación DOM
	Primera parte del diagrama de clases del modelo de datos de gestión del protocolo.
	Segunda parte del diagrama de clases del modelo de datos de gestión del protocolo.
	Diagrama de secuencia de la creación de un modelo de datos de gestión del protocolo a partir de su representación DOM, y de la creación del grafo, convirtiendo los elementos a su representación interna
	Diagrama de secuencia de la creación de un grafo de un modelo de datos de gestión del protocolo, creando objetos y asociaciones e insertándolos en el grafo.
	Diagrama de la secuencia de la conversión de un modelo de datos de gestión del protocolo y los elementos que contiene a su representación DOM.

6.2.2 ModelData

Clases relacionadas Modeldata

public class **Model** extends **DOMizable**	Representa un modelo de tipo genérico y lo presenta como un grafo. Almacena los distintos elementos del modelo: Objetos y Asociaciones. Ofrece métodos para manipular el modelo como un grafo: insertar nodos, insertar arcos, recorrer el grafo, ... etc.. Será la base para establecer el modelo del protocolo y el modelo reutilizable, así como para cualquier posible modelo futuro.
public class **ModelObject** extends **DOMizable**	Elemento genérico que representa un objeto de un modelo de datos de gestión. Almacena aquellos atributos que son comunes a todos los tipos de modelos y ofrece las operaciones necesarias para acceder a ellos.
public class **ModelAssoc** extends **DOMizable**	Elemento genérico que representa a una asociación de un modelo de gestión del protocolo, almacena aquellos atributos que son comunes a las asociaciones de los dos tipos de modelos y las operaciones para acceder a ellos.
public class **AssocRules**	Representación de una asociación, de un modelo de componente reutilizable, genérica. Incluye todos los atributos propios de este tipo de relaciones y permite entre otras cosas, especificar las ubicaciones alternativas de un objeto que no se encuentra en el mensaje.

public class **REMObject** extends **ModelObject**	Representación de un objeto de un modelo de componente reutilizable genérico. Presenta todos los atributos específicos de este tipo de objetos y las operaciones para acceder a ellos.
public class **ReusableElementModel** extends **Model**	Estructura genérica de un modelo de componente reutilizable. Almacena el nombre del modelo y presenta alguna operación necesaria para poder instanciar un modelo.
public class **REMFactory**	Clase de patrón Factory que se utilizará para poder obtener una instancia de un modelo de componente reutilizable concreto, ya que en tiempo de desarrollo no se conoce el modelo exacto que empleará el nodo. El modelo que utilizará el nodo se determina dinámicamente en tiempo de ejecución, a partir de los ficheros de configuración.
public class **REMAssoc** extends **ModelAssoc**	Representación de una asociación de un modelo de componente reutilizable. Incluye todos los atributos propios de este tipo de relaciones y que permiten entre otras cosas, especificar las ubicaciones alternativas de un objeto que no se encuentra en el mensaje.

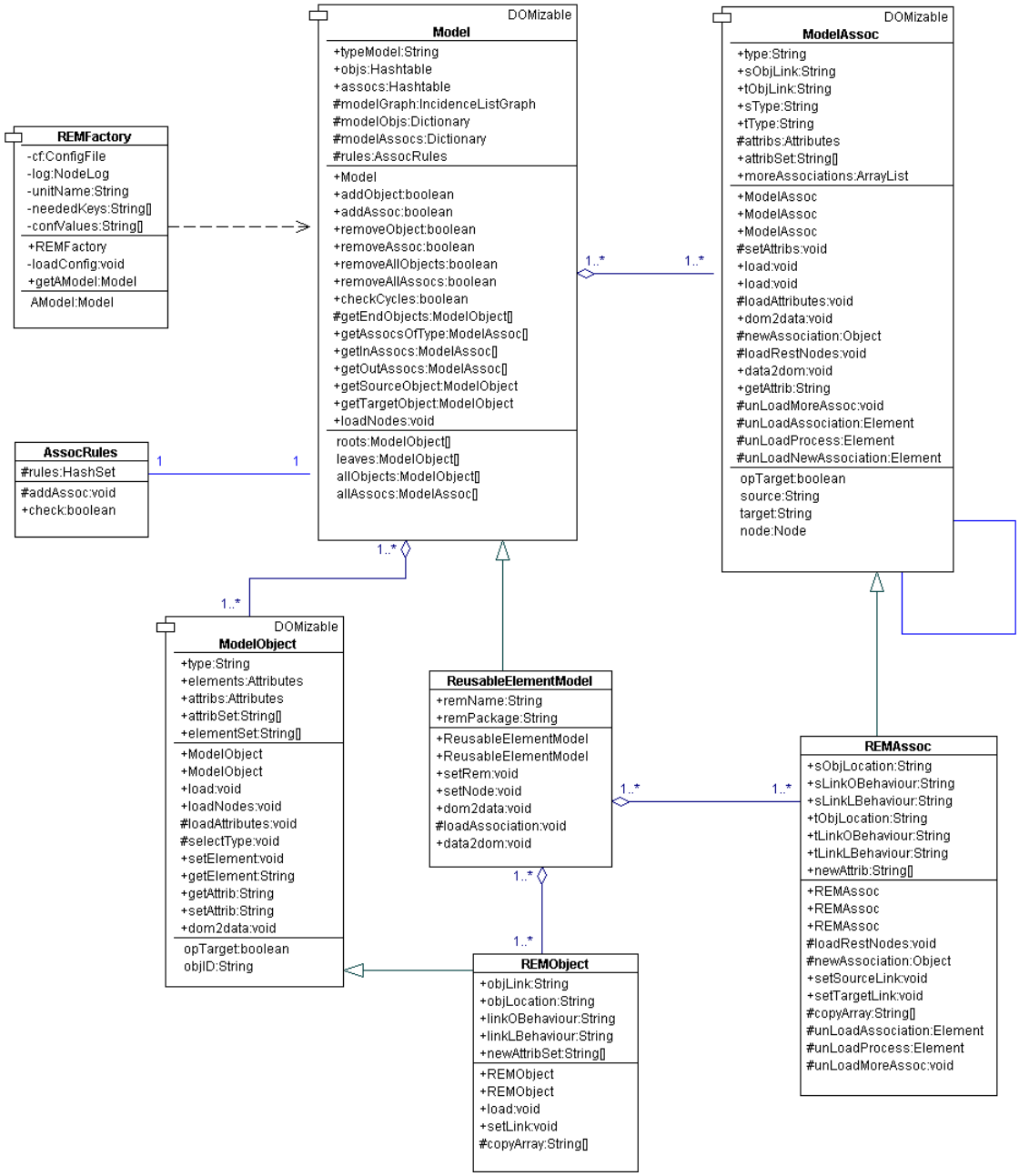


Diagrama de clases de las estructuras genéricas para representar datos de tipo 'modelo', con una interfaz de grafos.

Campos de la clase ***Model***

Tipo	Declaración	Acción
public String	typeModel	
public Hashtable	objs	
public Hashtable	assocs	
protected IncidenceListGraph	modelGraph	
protected Dictionary	modelObjs	
protected Dictionary	modelAssocs	
protected AssocRules	rules	

Constructores de la clase ***Model***

Tipo	Declaración	Acción
public	Model	

Métodos de la clase ***Model***

Tipo	Declaración	Acción
public boolean	addObject(obj:ModelObject)	
public boolean	addAssoc(obj1:ModelObject, bj2:ModelObject, assoc:ModelAssoc)	
public boolean	removeObject(obj:ModelObject)	
public boolean	removeAssoc(assoc:ModelAssoc)	
public boolean	removeAllObjects()	
public boolean	removeAllAssocs()	
public boolean	checkCycles()	
protected ModelObject[]	getEndObjects(type:int)	
public ModelAssoc[]	getAssocsOfType(obj:ModelObject, type:int)	
public ModelAssoc[]	getInAssocs(obj:ModelObject)	
public ModelAssoc[]	getOutAssocs(obj:ModelObject)	
public ModelObject	getSourceObject(assoc:ModelAssoc)	
public ModelObject	getTargetObject(assoc:ModelAssoc)	
public void	loadNodes(node:Node, doc:Document)	
public ModelObject[]	getRoots()	
public ModelObject[]	getLeaves()	

public ModelObject[]	getAllObjects()	
public ModelAssoc[]	getAllAssocs()	

*Campos de la clase **ModelObject***

Tipo	Declaración	Acción
public String	type	
public String	objID	
public boolean	opTarget	
public Attributes	elements	
public Attributes	attribs	
public String[]	attribSet	
public String[]	elementSet	

*Constructores de la clase **ModelObject***

Tipo	Declaración	Acción
public	ModelObject(objNode:Node, docu:Document)	
public	ModelObject()	

*Métodos de la clase **ModelObject***

Tipo	Declaración	Acción
public void	load()	
public void	loadNodes(node:Node, doc:Document)	
protected void	loadAttributes()	
protected void	selectType()	
public void	setElement(name:String, value:String)	
public String	getElement(name:String)	
public String	getAttrib(name:String)	
public String	setAttrib(name:String, value:String)	
public void	dom2data()	
public void	setObjID(objId:String)	
public void	setOpTarget(opTarget:boolean)	

*Campos de la clase **ModelAssoc***

Tipo	Declaración	Acción
public boolean	opTarget	
public String	sObjLink	
public String	tObjLink	
public String	sType	

public String	tType	
protected Attributes	attribs	
public String[]	attribSet	
public ArrayList	moreAssociations	

Constructores de la clase ***ModelAssoc***

Tipo	Declaración	Acción
public	ModelAssoc(assocNode:Node, docu:Document)	
public	ModelAssoc()	
public	ModelAssoc(n:Node, doc:Document, a:Attributes)	

Métodos de la clase ***ModelAssoc***

Tipo	Declaración	Acción
protected void	setAttribs()	
public void	load()	
public void	load(node:Node, docu:Document)	
protected void	loadAttributes()	
public void	dom2data()	
protected Object	newAssociation(n:Node, a:Attributes)	
protected void	loadRestNodes()	
public void	data2dom()	
protected void	unLoadMoreAssoc(assocFrom:Element)	
protected Element	unLoadAssociation()	
protected Element	unLoadProcess()	
protected Element	unLoadNewAssociation()	
public void	setNode(node:Node)	
public String	getAttrib(name:String)	
public void	setSource(sourceObjLink:String)	
public void	setOpTarget(opTarget:Boolean)	
public void	setTarget(targetObjLink:String)	

Campos de la clase ***AssocRules***

Tipo	Declaración	Acción
protected HashSet	rules	

Métodos de la clase ***AssocRules***

Tipo	Declaración	Acción
protected void	addAssoc(source:String, target:String, cardinality:String, label:String)	

public boolean	check(source:String, target:String)	

Campos de la clase ***ReusableElementModel***

Tipo	Declaración	Acción
public String	remName	
public String	remPackage	

Constructores de la clase ***ReusableElementModel***

Tipo	Declaración	Acción
public	ReusableElementModel(n:Node,docu:Document)	
public	ReusableElementModel()	

Métodos de la clase ***ReusableElementModel***

Tipo	Declaración	Acción
public void	setRem (modelName:String,modelPackage:String)	
public void	setNode(modelNode:Node, doc:Document)	
public void	dom2data()	
protected void	loadAssociation(asc:REMAssoc)	
public void	data2dom()	

Campos de la clase ***REMObject***

Tipo	Declaración	Acción
public String	objLink	
public String	objLocation	
public String	linkOBehaviour	
public String	linkLBehaviour	
public String[]	newAttribSet	

Constructores de la clase ***REMObject***

Tipo	Declaración	Acción
public	REMObject(Node objNode, Document docu)	
public	REMObject()	

Métodos de la clase ***REMObject***

Tipo	Declaración	Acción
public void	load()	
public void	setLink(obj:String, location:String,	

	oBehaviour:String, lBehaviour:String)	
protected String[]	copyArray(els:String[])	

*Campos de la clase **REMAssoc***

Tipo	Declaración	Acción
public String	sObjLocation	
public String	sLinkOBehaviour	
public String	sLinkLBehaviour	
public String	tObjLocation	
public String	tLinkOBehaviour	
public String	tLinkLBehaviour	
public String[]	newAttrib	

*Constructores de la clase **REMAssoc***

Tipo	Declaración	Acción
public	REMAssoc(assocNode:Node, docu:Document)	
public	REMAssoc(n:Node, doc:Document, a:Attributes)	

*Métodos de la clase **REMAssoc***

Tipo	Declaración	Acción
protected void	loadRestNodes()	
protected Object	newAssociation(n:Node, a:Attributes)	
public void	setSourceLink(objLocation:String, oBehaviour:String, lBehaviour :String)	
public void	setTargetLink(objLocation:String, oBehaviour:String, lBehaviour :String)	
protected String[]	copyArray(String[] els)	
protected Element	unLoadAssociation()	
protected Element	unLoadProcess()	
protected void	unLoadMoreAssoc(Element assocFrom)	

*Campos de la clase **REMFactory***

Tipo	Declaración	Acción
private ConfigFile	cf	
private NodeLog	log	
private String	unitName	
private String[]	neededKeys	
private String[]	confValues	

Constructores de la clase ***REMFactory***

Tipo	Declaración	Acción
public	REMFactory()	

Métodos de la clase ***REMFactory***

Tipo	Declaración	Acción
private void	loadConfig()	
public Model	getAModel()	
public Model	getAModel(modelNode:Node, docu:Document)	

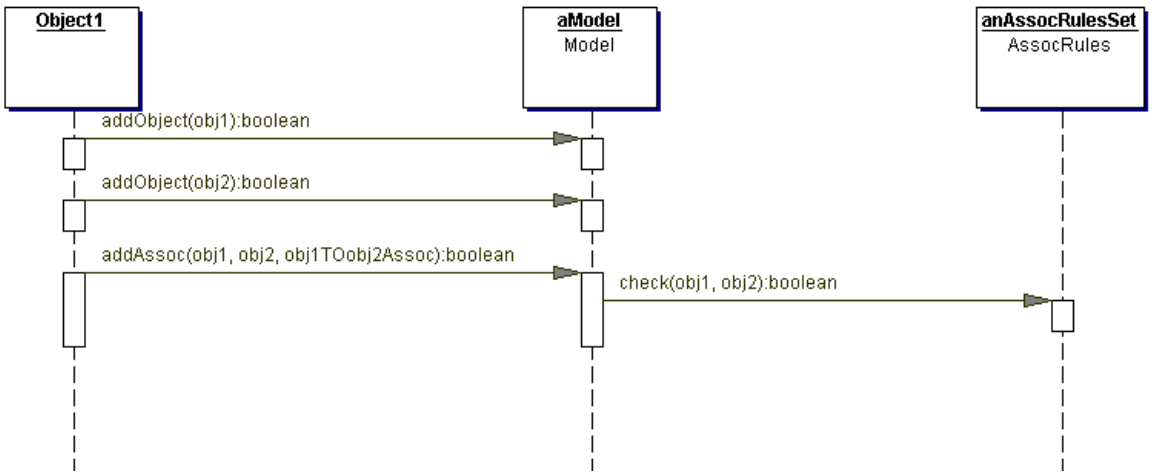


Diagrama de secuencia del proceso de insertar dos objetos y una asociación entre ellos, en un modelo.

6.2.3 Modelo de Mecano

Descripción

Para que los nodos puedan trabajar con el modelo de MECANO y para mostrar como se utiliza las estructuras definidas en el apartado anterior, se ha implementado la descripción del modelo de MECANO.

La evolución que ha sufrido esta parte se resumen en que todos los elementos del modelo de MECANO, pueden ser instanciados, utilizando las clases genéricas del modelo anterior y métodos que se han construido para terminar de especializar estas generalidades.

Clases relacionadas Mecano

public class **MecanoAssocRules** extends **AssocRules**	Conjunto de asociaciones que existen en el modelo *MECANO*. Se crean y almacenan las definiciones de las asociaciones para poder ser consultadas.
public class **MecanoModel** extends **ReusableElementModel**	Representación de un modelo *MECANO*, como una instancia del modelo genérico de componente reutilizable.
public class **MecanoObject** extends **REMObject**	Representación genérica de los objetos del modelo.
public class **SourceTarget** extends **MecanoObject**	Representación de la clase *SourceTarget* del modelo de *MECANO*.
public class **AssetType** extends **MecanoObject**	Representación de la clase *AssetType* del modelo de *MECANO*.
public class **FunctionalDescriptor** extends **MecanoObject**	Representación de la clase *FunctionalDescriptor* del modelo de *MECANO*.
public class **ProductLine** extends **MecanoObject**	Representación de la clase *ProductLine* del modelo de *MECANO*.
public class **RelationshipType** extends **MecanoObject**	Representación de la clase *Relationship* del modelo de *MECANO*.
public class **Asset** extends **MecanoObject**	Representación de la clase *Asset* del modelo de *MECANO*.
public class **Mecano** extends **MecanoObject**	Representación de la clase *Mecano* del modelo de *MECANO*.
public class **Domain** extends **MecanoObject**	Representación de la clase *Domain* del modelo de *MECANO*.
public class **Author** extends **MecanoObject**	Representación de la clase *Author* del modelo de *MECANO*.
public class **Representation** extends **MecanoObject**	Representación de la clase *Representation* del modelo de *MECANO*.
public class **Relationship** extends **MecanoObject**	Representación de la clase *Relationship* del modelo de *MECANO*.
public class **OtherClass** extends **MecanoObject**	Representación de la clase *OtherClass*, que aunque no pertenece al modelo de *MECANO*, si que es usada en el protocolo como parte del modelo.

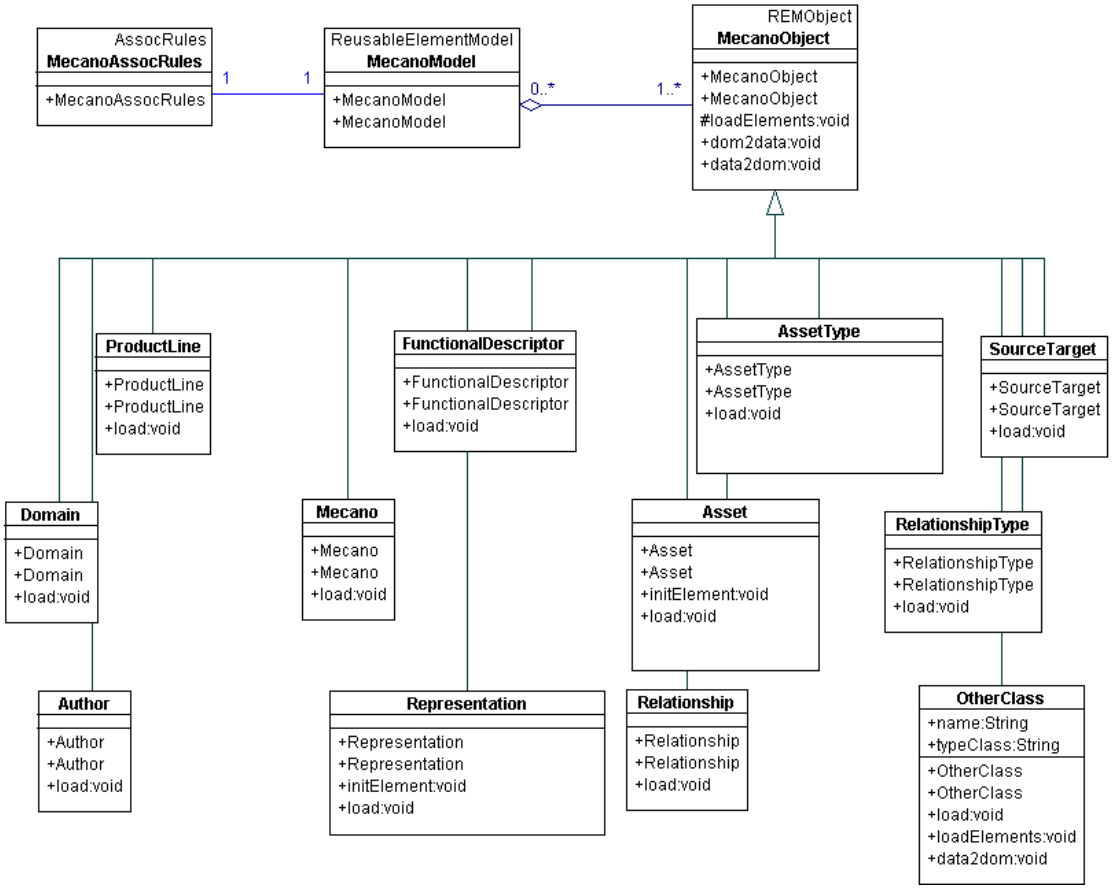


Diagrama de clases de la representación del modelo de Mecano

Vamos a comentar algunas consideraciones prácticas, aunque el diseño no es la mejor parte para ver cosas de implementación el crear un nuevo documento para la implementación hace todavía más tediosa la documentación.

Estructura de los datos para la clase OtherClass

En esta clase el atributo ***element*** seguirá siendo del tipo Attributes, pero almacenaremos en él el atributo del elemento ***atribute*** y como valor su ***CDATA.***

Para ello detallaremos la estructura que empleamos en la DTD.

```
<!ELEMENT OtherClass          (atribute*)>
<!ATTLIST OtherClass
  name      CDATA             #REQUIRED
  type      (class abstractClass    |
             interface)       #REQUIRED "class"
```

```
>
<!ELEMENT atribute              (#PCDATA)>
<!ATTLIST atribute
  name      CDATA               #REQUIRED
>
```

El atributo de la clase ***elements*** será de tipo Attributes, y su contenido:

Key	Value
name **(su contenido)** **-de atribute-**	**CDATA** **-de atribute-**

Y en **attribs** tenemos ***name*** y ***type*** de OtherClass.

*Constructores de la clase **MecanoAssocRules***

Tipo	Declaración	Acción
public	MecanoAssocRules()	

*Constructores de la clase **MecanoModel***

Tipo	Declaración	Acción
public	MecanoModel(Node n, Document docu)	
public	MecanoModel()	

*Constructores de la clase **MecanoObject***

Tipo	Declaración	Acción
public	MecanoObject(Node objNode, Document docu)	
public	MecanoObject()	

*Métodos de la clase **MecanoObject***

Tipo	Declaración	Acción
protected void	loadElements()	
public void	dom2data()	
public void	data2dom()	

*Constructores de la clase **SourceTarget***

Tipo	Declaración	Acción
public	SourceTarget(Node objNode, Document docu)	

public	SourceTarget()	

Métodos de la clase ***SourceTarget***

Tipo	Declaración	Acción
public void	load()	

Constructores de la clase ***AssetType***

Tipo	Declaración	Acción
public	AssetType(Node objNode, Document docu)	
public	AssetType()	

Métodos de la clase ***AssetType***

Tipo	Declaración	Acción
public void	load()	

Constructores de la clase ***FunctionalDescriptor***

Tipo	Declaración	Acción
public	FunctionalDescriptor (Node objNode, Document docu)	
public	FunctionalDescriptor()	

Métodos de la clase ***FunctionalDescriptor***

Tipo	Declaración	Acción
public void	load()	

Constructores de la clase ***ProductLine***

Tipo	Declaración	Acción
public	ProductLine(Node objNode, Document docu)	
public	ProductLine()	

Métodos de la clase ***ProductLine***

Tipo	Declaración	Acción
public void	load()	

Constructores de la clase ***RelationshipType***

Tipo	Declaración	Acción
public	RelationshipType (Node objNode, Document docu)	

public	RelationshipType()	

*Métodos de la clase **RelationshipType***

Tipo	Declaración	Acción
public void	load()	

*Constructores de la clase **Asset***

Tipo	Declaración	Acción
public	Asset(Node objNode, Document docu)	
public	Asset()	

*Métodos de la clase **Asset***

Tipo	Declaración	Acción
public void	initElement()	
public void	load()	

*Constructores de la clase **Mecano***

Tipo	Declaración	Acción
public	Mecano(Node objNode, Document docu)	
public	Mecano()	

*Métodos de la clase **Mecano***

Tipo	Declaración	Acción
public void	load()	

*Constructores de la clase **Domain***

Tipo	Declaración	Acción
public	Domain(Node objNode, Document docu)	
public	Domain()	

*Métodos de la clase **Domain***

Tipo	Declaración	Acción
public	void load()	

*Constructores de la clase **Author***

Tipo	Declaración	Acción
public	Author(Node objNode, Document docu)	
public	Author()	

Métodos de la clase ***Author***

Tipo	Declaración	Acción
public void	load()	

Constructores de la clase ***Representation***

Tipo	Declaración	Acción
public	Representation(Node objNode, Document docu)	
public	Representation()	

Métodos de la clase ***Representation***

Tipo	Declaración	Acción
public void	initElement()	
public void	load()	

Constructores de la clase ***Relationship***

Tipo	Declaración	Acción
public	Relationship(Node objNode, Document docu)	
public	Relationship()	

Métodos de la clase ***Relationship***

Tipo	Declaración	Acción
public void	load()	

Campos de la clase ***OtherClass***

Tipo	Declaración	Acción
public String	name	
public String	typeClass	

Constructores de la clase ***OtherClass***

Tipo	Declaración	Acción
public	OtherClass(Node objNode, Document docu)	
public	OtherClass()	

Métodos de la clase ***OtherClass***

Tipo	Declaración	Acción
public void	load()	
public void	loadElements()	

public void	data2dom()	

Diagrama de secuencia crear un modelo Mecano

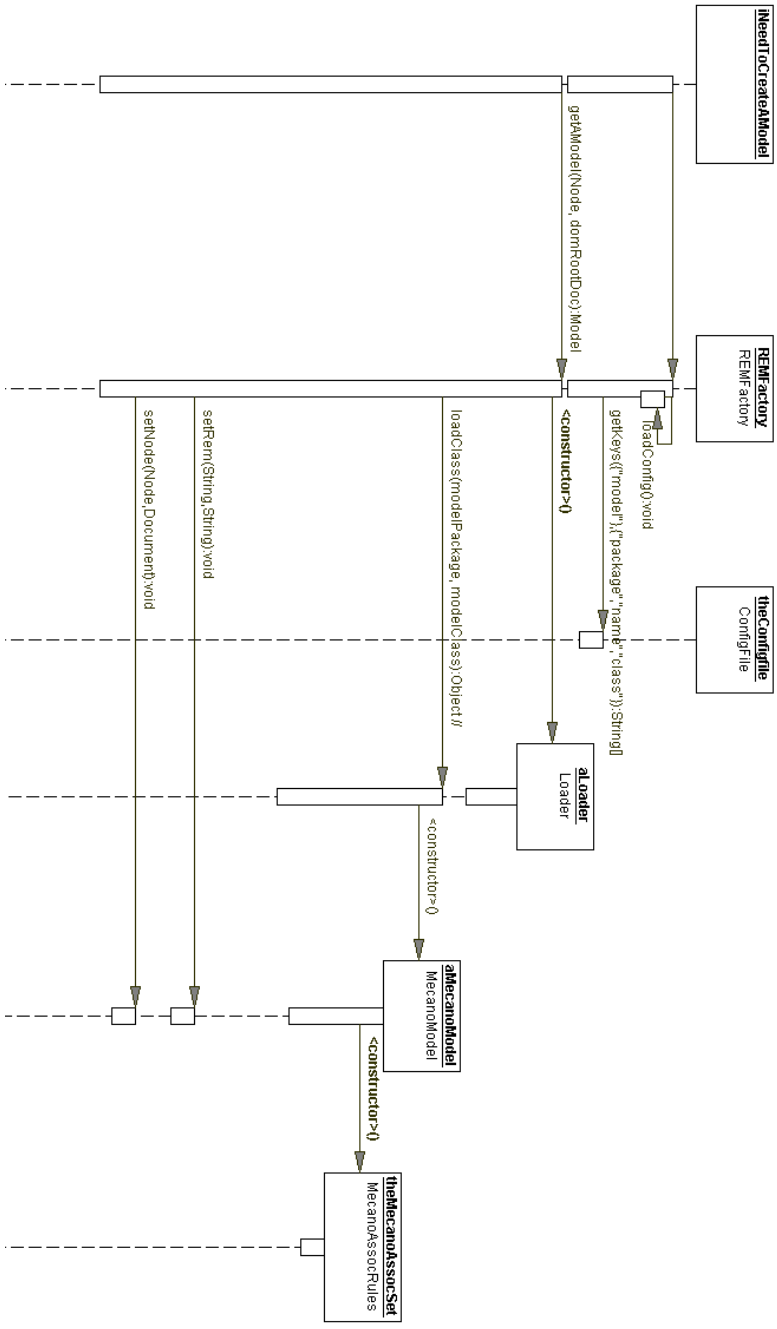


Diagrama de secuencia de la creación de un modelo de MECANO a partir de su representación DOM

Diagrama secuencia creación de un grafo

Diagrama de secuencia de la creación del grafo de un modelo de MECANO partir de su representación DOM, convirtiéndose los elementos a su representación interna

Diagrama secuencia creación de un modelo vacío

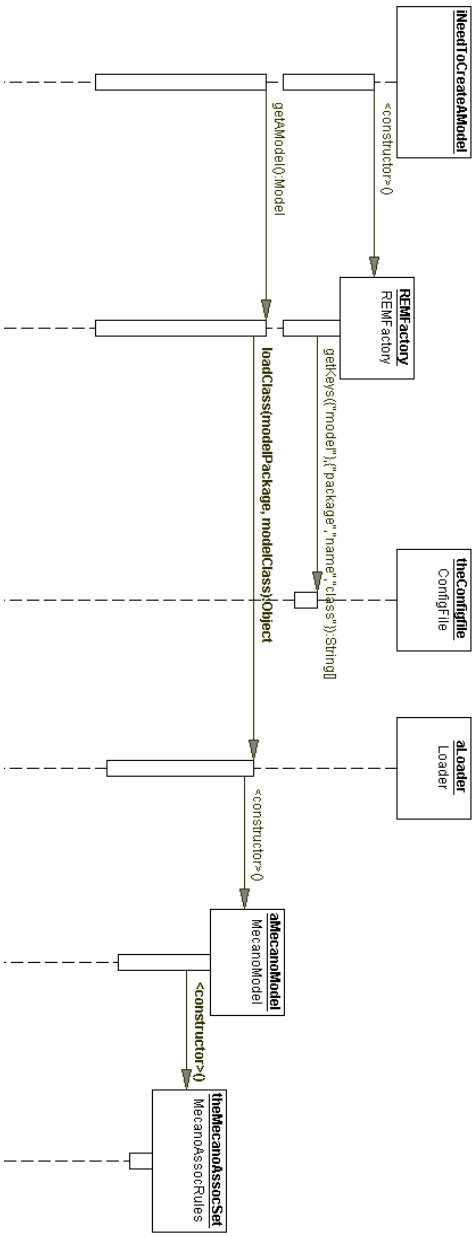


Diagrama de secuencia de la creación de un nuevo modelo de Mecano vacío

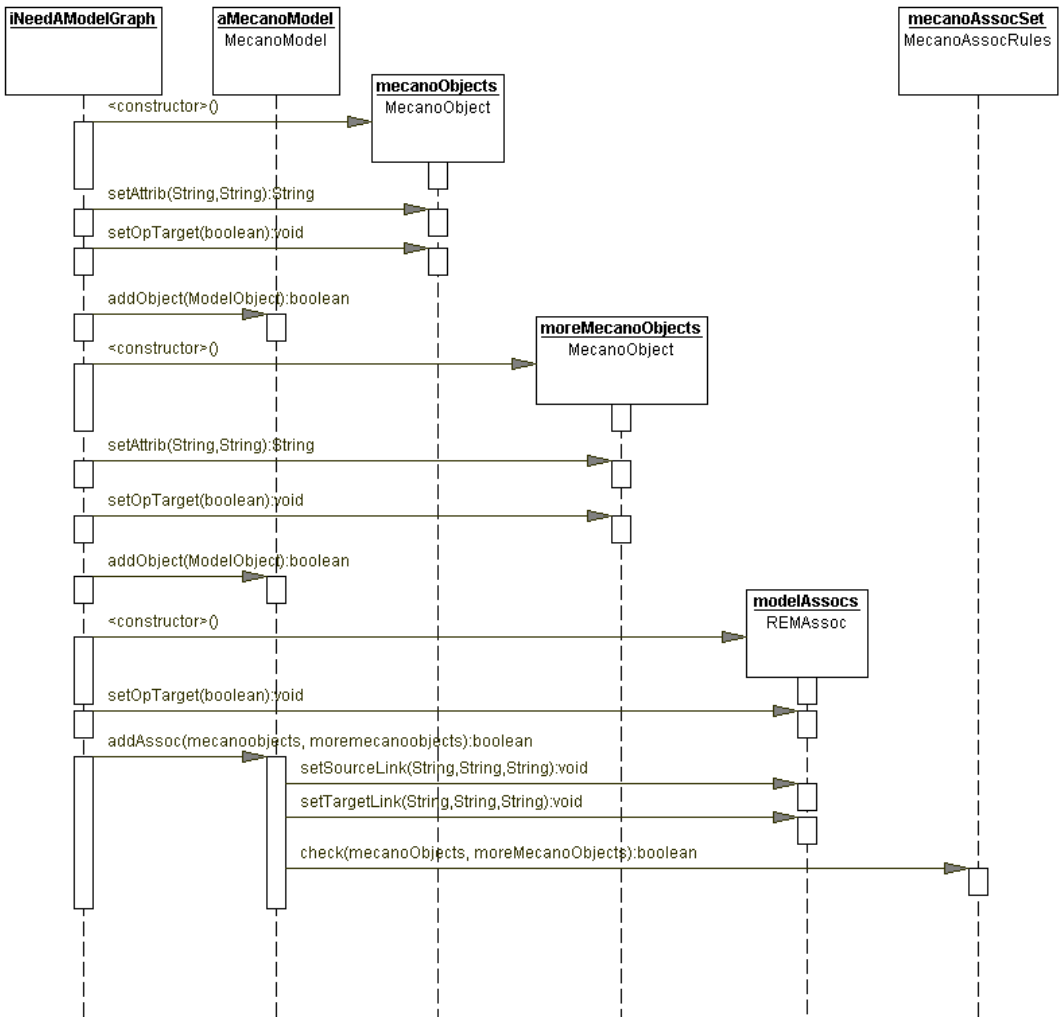


Diagrama de secuencia de la creación de un grafo de un modelo de Mecano, creando objetos y asociaciones e insertándolos en el grafo

Diagrama de secuencia para pasar de Mecano a DOM

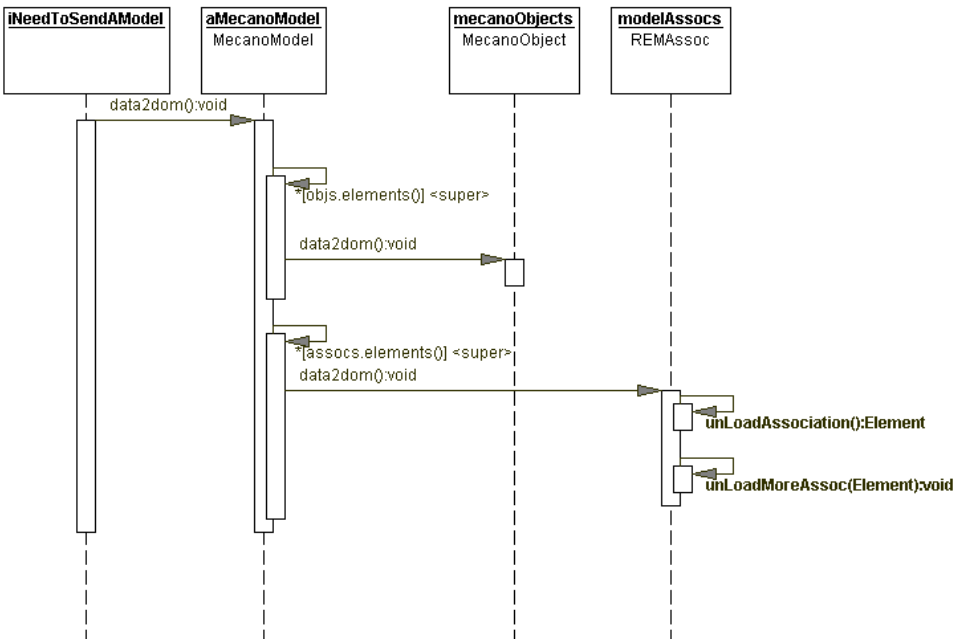


Diagrama de la secuencia de la conversión de un modelo de Mecano y los elementos que contiene a su representación DOM

6.2.4 Datos de gestión del protocolo

Descripción

El conjunto de datos de gestión del protocolo, también ha sido implementado como modelo. La principal diferencia con el modelo de componentes reutilizables, es que las clases de este modelo heredan directamente de las clases genéricas. Esto es así, porque la estructura y los atributos del modelo de datos de gestión del protocolo es común y aparece también en el modelo de componentes reutilizables. Al estar la estructura y los atributos comunes, definidos en las clases genéricas, este modelo se construye heredando directamente de las clases genéricas.

Se incluyen en este modelo algunas peculiaridades. Además de ofrecer una representación en forma de modelos y grafos, ofrecen una vista alternativa, pudiéndose acceder a sus atributos a través del contenedor de atributos y también de forma natural (objeto.atributo). Esto se ha diseñado así, para facilitar la utilización de estos objetos desde el nodo, ya que se trata de un modelo bien definido en tiempo de desarrollo, cerrado y común a todos los nodos.

Se incluyen además, algunas clases en el modelo, que no pertenecen estrictamente a él, es decir, no siguen las estructuras de objetos pertenecientes a un modelo y no heredan de ProtocolDataObjetcts. Se trata de datos que pueden encontrarse en las cabeceras MIPHeader de los mensajes y que ofrecen una representación distinta de la de los objetos del modelo.

Al igual que en el diagrama anterior, la evolución de esta parte consiste en formar cada uno de los elementos del modelo de gestión del protocolo. Se han creado métodos especializados, partiendo de los métodos generales, creados en el diagrama inicial de este tema. Además de los cambios originados por estas especializaciones hay una clase, NODESKILL, que se ha tenido que generalizar, debido a la gran cantidad de información que podía albergar. Este último cambio se detalla en la sección .....

Clases relacionadas Datos de Gestión de Protocolo

Al igual que para el modelo anterior, el tener presente el formato de la ***DTD*** será muy aclaratorio, pues en los siguientes pasos veremos como hemos plasmado la estructura de una ***DTD*** en unos diagramas de clases.

public class **ProtocolDataAssocRules** extends **AssocRules**	Conjunto de asociaciones que existen en el modelo de datos de gestión del protocolo. Se crean/almacenan las definiciones de las asociaciones para que puedan ser consultadas posteriormente, pudiendo hacer comprobaciones.
public class **ProtocolDataModel** extends **Model**	Representación de un modelo de datos de gestión del protocolo como una instancia del modelo genérico.
public class **ProtocolDataObject** extends **ModelObject**	Representación genérica de los objetos del modelo.
public class **OtherData** extends **ProtocolDataObject**	Representación de la clase *OtherData* del modelo de datos de gestión del protocolo.

public class **NodeInfo** extends **ProtocolDataObject**	Representación de la clase *NodeInfo* del modelo de datos de gestión del protocolo
public class **Talk** extends **ProtocolDataObject**	Representación de la clase *Talk* del modelo de datos de gestión del protocolo.
public class **User** extends **ProtocolDataObject**	Representación de la clase *User* del modelo de datos de gestión del protocolo.
public class **NodeSkill** extends **ProtocolDataObject**	Representación de la clase NodeSkill del modelo de datos de gestión del protocolo.
public class **NodeAddress** extends **DOMizable**	Clase que representa la dirección de un nodo, bien mediante el identificador del nodo (*nombre_del_broker.nombre_del_nodo*), único en el entorno de desarrollo, o bien mediante la dirección ***IP*** del nodo y el puerto en el que atiende las peticiones. Aunque forma parte del conjunto de datos de gestión del protocolo, no forma parte del modelo, las instancias de esta clase no se consideran objetos del modelo de datos de gestión del protocolo.
public class **Session** extends **ProtocolDataObject**	Representación de la clase *Session* del modelo de datos de gestión del protocolo.
public class **Identity** extends **DOMizable**	Clase que representa la identidad de un usuario en un nodo, bien a través de un identificador de sesión o bien por medio de los datos login, password y rol del usuario. Aunque forma parte del conjunto de datos de gestión del protocolo, no forma parte del modelo, las instancias de esta clase no se consideran objetos del modelo de datos de gestión del protocolo.

La clase NodeSkill, y como podemos ver en la DTD, esta compuesta de otra serie de elementos funcionales, por ello se obtienen del proceso de implementación una serie de nuevas clases. De ahí que estas clases tengan una especificación y funcionalidad más detallada, vemos en la siguiente sección su estudio.

public class **Skills** extends **DOMizable**	Clase estructural que representa los principales elementos de las características. Como cualquier clase que hereda de *DOMIzable*, posee características para manejar y tratar estructuras ***DOM***.
public class **OtherMethods** extends **Skills**	Es la generalización de las clases *FilterSkill* y *OtherSkill*, representando las principales características de éstas.
public class **Methods** extends **Skills**	Es la generalización de las clases *ServiceSkill* y *FunctionSkill*, representando las principales características de éstas.
public class **OtherSkill** extends **OtherMethods**	Esta clase nos sirve para detallar características no definidas en la *DTD*.
public class **FilterSkill** extends **OtherMethods**	Esta clase representa características de Filtrado, tanto para la compresión como para la encriptación.

public class **ServiceSkill** extends **Methods**	Sabemos que un nodo nos puede ofrecer servicios y funciones. Con esta clase representaremos las características de los servicios de los nodos.
public class **FunctionSkill** extends **Methods**	Sabemos que un nodo nos puede ofrecer servicios y funciones. Con esta clase representaremos las características de las funciones de los nodos.

Estructura de los datos para la clase NodeSkill en ProtocolData

Debido a la gran información que va a almacenar, esta clase merece un estudio detallado, y la especificación, de forma clara, de la solución tomada para la carga de sus datos.

Realizando un riguroso estudio de la DTD, en un primer estudio y realización se optó por que sus atributos lleven el control de toda la información de XML.

Para ello detallaremos la estructura que empleamos.

```
<!ELEMENT
NodeSkill((ServiceSkill|FunctionSkill|FilterSkill|OtherSkill)*)
>
```

Con esto hemos creado un atributo que se llamará nodeNodeSkill, como vemos almacenar varias "especializaciones" por ello este atributo será **ArrayList**.

Este array tendrá como elementos un **Hashtable;**

```
NodeNodeSkill = { Hashtable1, Hastable2, Hastable3... };
```

A continuación detallamos cada uno de los valores que toma el HastableX:

Key	Value
"ServiceSkill"	HashtableX1
"FunctionSkill"	HashtableX2
"FilterSkill"	ArrayListX3(Attribute)
"OtherSkill"	ArrayListX4(Attribute)

Como vemos cada valor de este hashtable inicial será una llamada a otro hashtable, debida a la implementación de la DTD.

1. A continuación estructuramos **HasshtableX1 y HashtableX2**:

Key	Value
"ServiceName"	String
"ServiceID"	String
"PostCondition"	ArrayList(String)
"PreCondition"	ArrayList(String)
"ClientNode"	ArrayList(String)
"ServerNode"	ArrayList(String)
"Input"	HasTableX11
"output"	HasTableX12

   a. Visto esta estructura vamos a al estructura del **hashtableX11** y **HashtableX12**, pues tienen exactamente los mismos nodos en XML**:**

Key	Value
“PlainData”	ArrayList(HashtableX111)
“ModelData”	String

i. Y para terminar la estructura de **HashtableX111**:

Key	Value
“argType”	String
“usage”	String
“PDName”	String
“Constraint”	ArrayList(Atributes)

Donde los atributos tienen los elementos

Key	Value
“value”	String
“consType”	String

2. Seguimos con **ArrayListX3(Attribute),** donde cada elemento puede tener:
   El primer elemento será siempre el atributo FilterSkill

Key	Value
“skillType”	String

   y los siguientes elementos podrán ser:

Key	Value
“value”	**String**
“name”	**String**

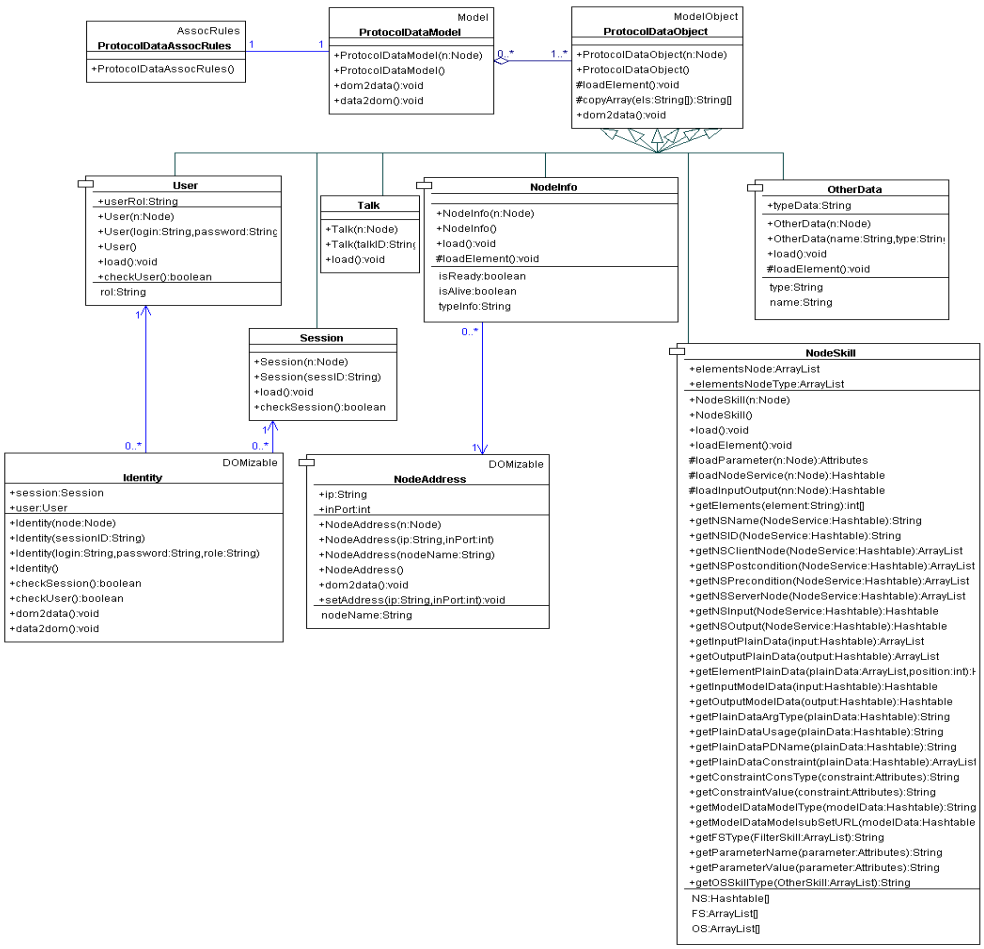
3. Seguimos con **ArrayListX4(Attribute)**:
   El primer elemento será siempre el atributo FilterSkill

Key	Value
“type”	String

   y los siguientes elementos podrán ser:

Key	Value
“value”	String

Como vemos esta solución es un poco compleja, obteniendo un diagrama:



La clase NodeSkill se hace muy extensa y con ello el problema de tener siempre presente la estructura de los datos tomada; por ello se ha decidido romper algunos aspectos de la estructuración de la DTD.

Una manera menos complicada sería almacenar todo en cadenas, y separarlo mediante caracteres. Teniendo delante la *DTD* del *XML*, podemos tener siempre en la cabeza lo que quedemos introducir, que será siempre una cadena. Ahora se han definido tres caracteres especiales que son *S1*, *S2* y *S3*, detallados en la clase *SKILLS*. Cada uno de estos caracteres simbolizará una acción a tomar a la hora de cargar los datos. Para explicar su significado partimos del contexto de niveles, en paréntesis he reflejado a que nivel pertenecen las etiquetas (nivel 1, nivel 2, o nivel 3).

- S1: Simboliza la separación entre etiquetas repetidas, si solo puede albergar un elemento (según la DTD) entonces significará que el elemento está en el siguiente elemento permitido, si puede albergar múltiples entonces pertenecerán al mismo elemento.
- S2: Simboliza el salto de un tag*km* a otro tag*k(m+1)* en el que hemos insertado S1. (Salto entre tags hermanos).
- S3: Simboliza el final del tag padre en el que hemos insertado S1.

Podemos ver en un ejemplo como sería esta estructura:

```
<FunctionSkill>
      <NodeService>
            <ServiceName>Nombre del servicio</ServiceName>
            <ServiceID>Service_001</ServiceID>
            <Precondition>
                  Primera Precondicion service 001
            </Precondition>
            <Precondition>
                  Segunda Precondicion service 001
            </Precondition>
            <Postcondition>
                  Primera Postcondicion service 001
            </Postcondition>
            <Postcondition>
                  Segunda Postcondicion service 001
            </Postcondition>
            <Input>
                  <PlainData argType = "short" usage="optional">
                        <PDName>Nombre 1 de Plain Data</PDName>
                        <Constraint consType="range">
                              Restriccion 1
                        </Constraint>
                        <Constraint consType="range">
                              Restriccion 1
                        </Constraint>
            </PlainData>
            <PlainData argType = "short" usage="optional">
                  <PDName>Nombre 2 de Plain Dataaa</PDName>
            </PlainData>
            <ModelData modelType = "xmm">
                  <ModelsubSetURL>
                        URL del subconjunto del modelo
                  </ModelsubSetURL>
            </ModelData>
      </Input>
      <Output>
            <PlainData argType = "int" usage="optional">
                  <PDName>Nombre 1 de Plain Data</PDName>
                  <Constraint consType="range">
                        Restriccion 1
                  </Constraint>
```

```
            </PlainData>
            <PlainData argType = "boolean" usage="optional">
                  <PDName>Nombre 2 de Plain Dataaa</PDName>
                  <Constraint consType="range">
                        Limitacion 2
                  </Constraint>
                  <Constraint consType="range">
                        Limitacion 3
                  </Constraint>
                  <Constraint consType="range">
                        Limitacion 4
                  </Constraint>
            </PlainData>
            </Output>
            <ClientNode nodeType="tool"></ClientNode>
            <ClientNode nodeType="broker"></ClientNode>
            <ServerNode nodeType="processor"></ServerNode>
            <ServerNode nodeType="fileBroker"></ServerNode>
      </NodeService>
</FunctionSkill>
```

La estructura que generaremos será:

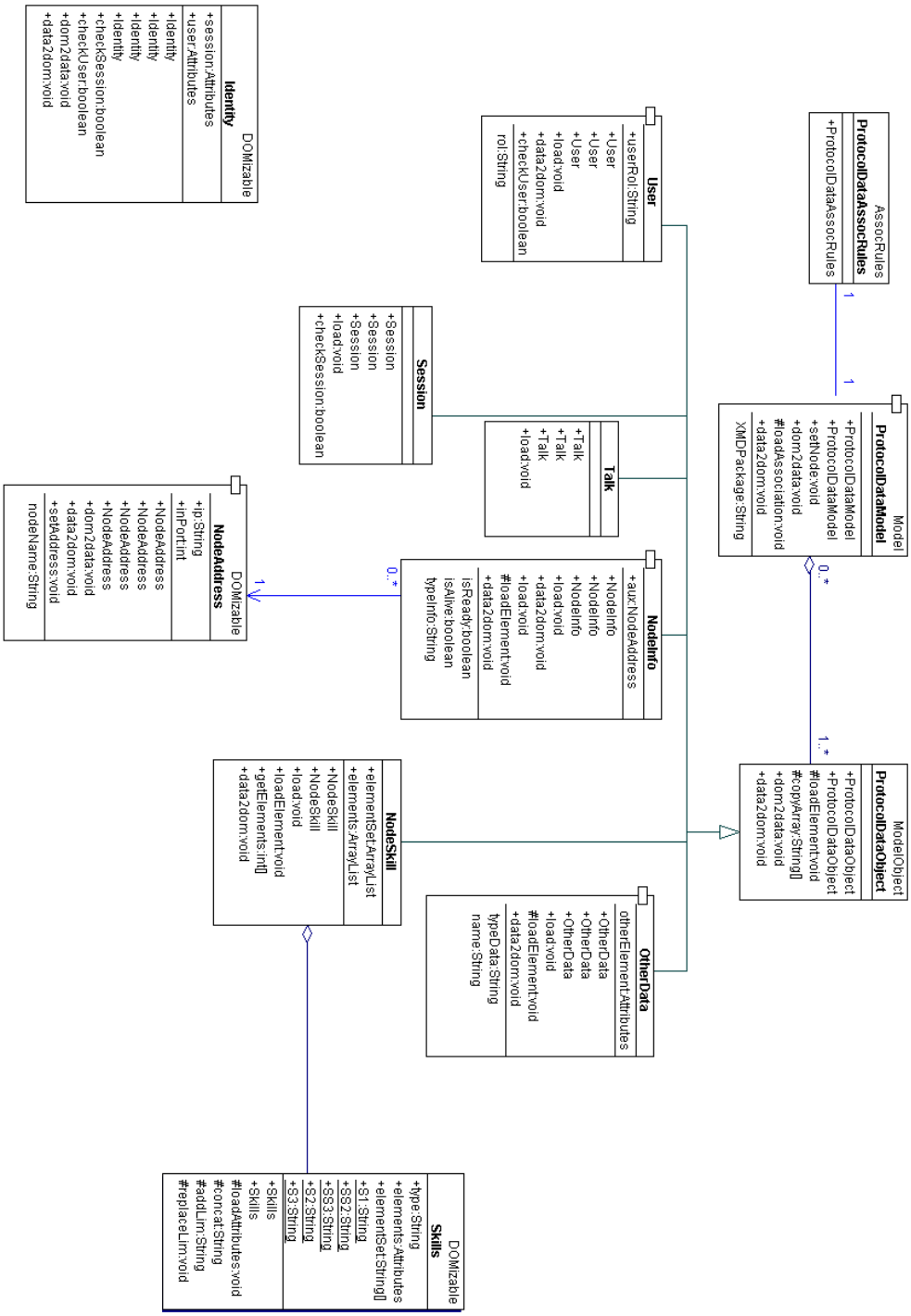
Elementos	***Valores***
"ServiceName"	Nombre del servicio
"ServiceID"	*Service_001* Solo puede aparecer una vez, no hay problema.
"Precondition"	*Primera Precondicion service 001**S1** Segunda Precondicion service 001**S1*** Aquí vemos que sólo nos hace falta S1 pues no hay más elementos en ningún otro sitio, sólo separaremos los elementos dentro del mismo tag.
"Postcondition"	*Primera Postcondicion service 001**S1** Segunda Postcondicion service 001**S1*** Aquí vemos que sólo nos hace falta S1 pues no hay más elementos en ningún otro sitio, sólo separaremos los elementos dentro del mismo tag.
"ClientNode"	*tool**S1**broker* Aquí vemos que sólo nos hace falta S1 pues no hay más elementos en ningún otro sitio, sólo separaremos los elementos dentro del mismo tag.
"ServerNode"	*processor**S1**fileBroker* Aquí vemos que sólo nos hace falta S1 pues no hay más elementos en ningún otro sitio, sólo separaremos los elementos

	dentro del mismo tag.
“argType”	*short**S1**short **S3**int**S1**boolean **S3*** Aquí los shorts, separados por el S1 nos indican que argType ocurre en la primera etiqueta que podemos encontrar en la que haya este elemento PlainData, y después el S3 nos indica que se ha terminado el Input, el int entonces está en el siguiente elemento (output) al igual que el boolean, ambos seguidos, S3 nos indica que finaliza output.
“usage”	*optional**S1**optional **S3**optional**S1**optional **S3*** Aquí los optionals, separados por el S1 nos indican que usage ocurre en la primera etiqueta que podemos encontrar en la que haya este elemento PlainData, y después el S3 nos indica que se ha terminado el Input, el optional, entonces está en el siguiente elemento (output) al igual que el segundo optional, ambos seguidos, S3 nos indica que finaliza output.
“PDName”	*Nombre 1 de Plain Data**S1**Nombre 2 de Plain Dataaa **S3**Nombre 1 de Plain Data**S1**Nombre 2 de Plain Dataaa **S3*** Ahora “Nombre 1 de Plain Data” está en un elemento pero como solo se permite que esté en un elemento el siguiente, Nombre 2 de Plain Dataaa, estará en el siguiente permitido; con S3 termina el padre, que es input, y no PlainData, porque PDName es obligatorio para PlainData, por eso cuando usamos S1, queremos decir que esta en otro PlainData, y S3 ha terminado Input.
“Constraint”	*Restriccion 1**S1**Restriccion 1 **S2** **S3**Restriccion 1 **S2**Limitacion 2**S1**Limitacion 3**S1**Limitacion 4 **S3*** Ahora constraint puede estar en un mismo elemento varias veces por ello S1 nos indicará que es un atributo que se repite dentro del elemento. Por ello Restricción 1 y Restriccion1, son tags distintos dentro de un mismo elemento, PlainData. S2 que ha finalizado el padre, PlainData, donde hemos introducido las dos primeras restricciones y con S3 que se ha finalizado input. Lo mismo para lo siguiente Restricción 1 cuenta que solo hay una restricción en el primer PlainData, el S2 indica que las siguientes restricciones están en el siguiente PlainData, donde localizaremos Limitacion2, Limitacion 3 y Limitación 4. El último S3 nos indica la finalización del output.
“consType”	*range**S1**range **S2** **S3**range **S2**range**S1**range**S1**range **S3*** Ahora constype puede estar en un mismo elemento varias veces por ello S1 nos indicará que es un atributo que se repite dentro del elemento. Por ello range y range, son tags distintos dentro de un mismo elemento. S2 que ha finalizado el padre, PlainData, donde hemos introducido las dos primeras características y con S3 que se ha finalizado input. Lo mismo para lo siguiente, range cuenta que solo hay un elemento con range, en el primer PlainData, el S2 indica que las siguientes características están en el siguiente PlainData, donde localizaremos los tres ranges que quedan. El último S3 nos indica la finalización del output.
“ModelType”	*xmm **S3** **S3***

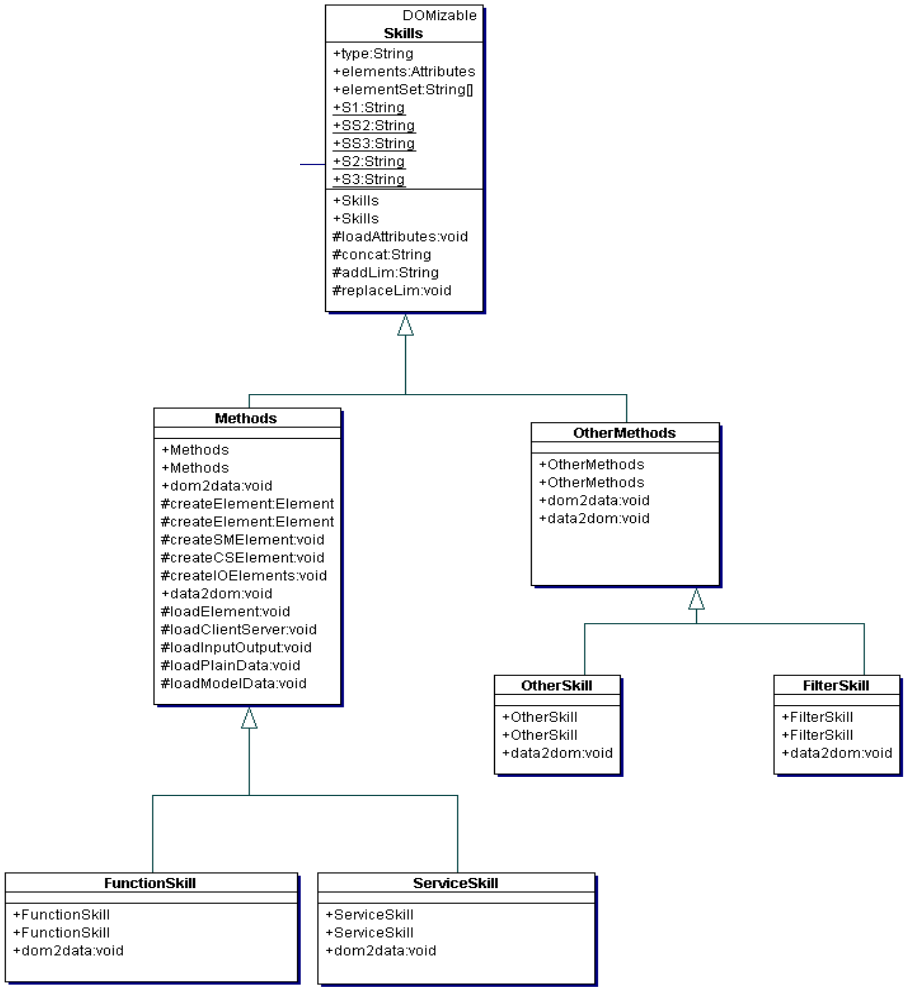
	Indica que xmm está dentro del primer input y solo se puede localizar dentro de ModelData, por lo cual no nos hace falta S2, por ello finalizamos con S3 diciendo que el siguiente ModelType que hay es vacío, en el Output no habrá.
"ModelsubSetURL"	*URL del subconjunto del modelo* ***S3*** ***S3*** Indica que ModelsubSetURL está dentro del primer input y solo se puede localizar dentro de ModelData, por lo cual no nos hace falta S2, por ello finalizamos con S3 diciendo que el siguiente ModelSubSetURL que hay es vacío, en el Output no habrá.
"InputOrOutput"	Input**S1**Output *Indica las etiquetas input y output que haya. En su mismo orden*

Esto es mucho más sencillo, sólo hay que tener presente el nivel, que es marcado por la DTD, del documento. Indicar que hay que respetar las separaciones de espacios, pues marcan que no hay elementos.

Modelo estático: diagrama de clases Datos de Gestión de Protocolo



Primera parte del diagrama de clases del modelo de datos de gestión del protocolo



Segunda parte del diagrama de clases del modelo de datos de gestión del protocolo

Nuevamente se han querido plasmar algunas consideraciones practicas:

Estructura de datos para la clase OtherData en ProtocolData

En esta clase el atributo ***element*** seguirá siendo del tipo Attributes, pero almacenaremos en él el atributo del elemento ***Parameter*** y como valor su ***CDATA.***

Para ello detallaremos la estructura que empleamos en la DTD.

```
<!ELEMENT OtherData     (Parameter*)>
<!ATTLIST OtherData
      name  CDATA              #IMPLIED
      type  CDATA              #IMPLIED
>
<!ELEMENT Parameter           (#PCDATA)>
<!ATTLIST Parameter
      name  CDATA              #REQUIRED
      type  CDATA              #REQUIRED
>
```

El atributo ***element*** será de tipo Attributes, y su contenido:

Key	Value
name (su contenido) -de Parameter-	CDATA -de parameter-

Debido a los cambios que ha sufrido la *DTD*, el parámetro *type* de *Parameter* ha sido introducido con bastante anterioridad, por ello se ha añadido una nueva variable que lo almacena, de tipo Attributes:

Key	Value
name (su contenido) -de Parameter-	Type (su contenido) -de Parameter-

Estructura de los datos para la clase NodeInfo de ProtolData

Al igual que hemos hecho en otros casos aquí tampoco vamos a respetar la estructura del XML de forma exacta, para tener los datos en una estructura lo menos compleja y uniforme.

```
<!ELEMENT NodeInfo
      (NodeAddress?, OutPort?, Signature?, Info?)
>
<!ELEMENT NodeAddress
      (NodeName | (IP, InPort) | (NodeName, IP, InPort) )
>
<!ATTLIST NodeInfo
  type      (tool       |
             broker     |
             processor  |
             repository |
             fileBroker)      #REQUIRED
  isAlive    (true|false)     #IMPLIED
  isReady   (true|false)      #IMPLIED
>
<!ELEMENT IP                  (#PCDATA)>
<!ELEMENT InPort              (#PCDATA)>
```

```
<!ELEMENT OutPort          (#PCDATA)>
<!ELEMENT Signature        (#PCDATA)>
<!ELEMENT Info             (#PCDATA)>
<!ELEMENT NodeName         (#PCDATA)>
```

Ahora element almacenará:

Key	Value
"Nodename"	String
"IP"	String
"InPort"	String
"OutPort"	String
"Signature"	String
"Info"	String

Como vemos el elemento *NodeAddress* pierde su aparición, pero no su valor o componentes.
Como siempre en ***attribs***, almacenaremos los valores de los atributos que para esta caso serán:

"***type***", "***isAlive***", "***isReady***"

Constructores de la clase ***ProtocolDataAssocRules***

Tipo	Declaración	Acción
public	ProtocolDataAssocRules()	

Campos de la clase ***ProtocolDataModel***

Tipo	Declaración	Acción
public String	XMDPackage	

Constructores de la clase ***ProtocolDataModel***

Tipo	Declaración	Acción
public	ProtocolDataModel(Node n, Document docu)	
public	ProtocolDataModel()	

Métodos de la clase ***ProtocolDataModel***

Tipo	Declaración	Acción
public void	setNode(Node modelNode, Document doc)	
public void	dom2data()	
protected void	loadAssociation(ModelAssoc asc)	
public void	data2dom()	
public void	setXMDPackage(String modelPackage)	

Constructores de la clase ***ProtocolDataObject***

Tipo	Declaración	Acción
public	ProtocolDataObject(Node n, Document docu)	
public	ProtocolDataObject()	

Métodos de la clase ***ProtocolDataObject***

Tipo	Declaración	Acción
protected void	loadElement()	
protected String[]	copyArray(String[] els)	
public void	dom2data()	
public void	data2dom()	

Campos de la clase ***OtherData***

Tipo	Declaración	Acción
public String	name	
public String	typeData	
Attributes	otherElement	

Constructores de la clase ***OtherData***

Tipo	Declaración	Acción
public	OtherData(Node n, Document docu)	
public	OtherData()	
public	OtherData(String name, String type)	

Métodos de la clase ***OtherData***

Tipo	Declaración	Acción
public void	load()	
protected void	loadElement()	
public void	data2dom()	
public void	setName(String name)	
public void	setTypeData(String value)	

Campos de la clase ***NodeInfo***

Tipo	Declaración	Acción
public String	typeInfo	
public boolean	isReady	
public boolean	isAlive	

public NodeAddress	aux	

*Constructores de la clase **NodeInfo***

Tipo	Declaración	Acción
public	NodeInfo(Node n, Document docu)	
public	NodeInfo (String typ, String isA, String isR, Document root)	
public	NodeInfo()	

*Métodos de la clase **NodeInfo***

Tipo	Declaración	Acción
public void	load()	
protected void	loadElement()	
public void	data2dom()	
public void	setIsReady(boolean ready)	
public void	setIsAlive(boolean alive)	
public void	setTypeInfo (String t)	

La siguiente clase, no se ha redefinido, pues aunque se ha marcado su funcionalidad no se ha especificado su diseño.

*Constructores de la clase **Talk***

Tipo	Declaración	Acción
public	Talk(Node n, Document docu)	
public	Talk(String talkID)	
public	Talk()	

*Métodos de la clase **Talk***

Tipo	Declaración	Acción
public void	load()	

*Campos de la clase **User***

Tipo	Declaración	Acción
public String	userRol	

*Constructores de la clase **User***

Tipo	Declaración	Acción
public	User(Node n, Document docu)	
public	User(String login, String password, String rol)	

public	User()	

*Métodos de la clase **User***

Tipo	Declaración	Acción
public void	load()	
public void	data2dom()	
public boolean	checkUser()	
public void	setRol(String rol)	

*Métodos de la clase **NodeSkill***

Tipo	Declaración	Acción
public ArrayList	elementSet	
public ArrayList	elements	

*Constructores de la clase **NodeSkill***

Tipo	Declaración	Acción
public	NodeSkill(Node n, Document docu)	
public	NodeSkill()	

*Métodos de la clase **NodeSkill***

Tipo	Declaración	Acción
public void	load()	
public void	loadElement()	
public int[]	getElements(String element)	
public void	data2dom()	

*Campos de la clase **NodeAddress***

Tipo	Declaración	Acción
public String	nodeName	
public String	ip	
public int	inPort	

*Constructores de la clase **NodeAddress***

Tipo	Declaración	Acción
public	NodeAddress(Node n, Document docu)	
public	NodeAddress (String ip, int inPort, Document docu)	
public	NodeAddress(String nodeName,Document docu)	

public	NodeAddress()	

Métodos de la clase ***NodeAddress***

Tipo	Declaración	Acción
public void	dom2data()	
public void	data2dom()	
public void	setAddress(String ip, int inPort)	
public void	setNodeName(String nodeName)	

Constructores de la clase ***Session***

Tipo	Declaración	Acción
public	Session(Node n, Document docu)	
public	Session(String sessID)	
public	Session()	

Métodos de la clase ***Session***

Tipo	Declaración	Acción
public	void load()	
public boolean	checkSession()	

La siguiente clase puede confundirnos pues en la DTD vemos que posee los elementos User y Session, pero estos elementos no son los mismos que los que heredan de los objetos de la gestión del protocolo, por ello sus datos se cargarán como atributos y no como objetos.

Campos de la clase ***Identity***

Tipo	Declaración	Acción
public Attributes	session	
public Attributes	user	

Constructores de la clase ***Identity***

Tipo	Declaración	Acción
public	Identity(Node node, Document docu)	
public	Identity(String sessionID)	
public	Identity(String login, String password, String role)	
public	Identity()	

*Métodos de la clase **Identity***

Tipo	Declaración	Acción
public boolean	checkSession()	
public boolean	checkUser()	
public void	dom2data()	
public void	data2dom()	

*Campos de la clase **Skills***

Tipo	Declaración	Acción
public String	type	
public Attributes	elements	
public String[]	elementSet	
static public String	S1	
static public String	SS2	
static public String	SS3	
static public String	S2	
static public String	S3	

*Constructores de la clase **Skills***

Tipo	Declaración	Acción
public	Skills(Node n, Document docu)	
public	Skills()	

*Métodos de la clase **Skills***

Tipo	Declaración	Acción
protected void	loadAttributes(Node n)	
protected String	concat(String name, String data)	
protected String	addLim(String name, String lim)	
protected void	replaceLim(String name, String lim)	

*Constructores de la clase **OtherMethods***

Tipo	Declaración	Acción
public	OtherMethods(Node n, Document docu)	
public	OtherMethods()	

*Métodos de la clase **OtherMethods***

Tipo	Declaración	Acción
public void	dom2data()	
public void	data2dom()	

*Constructores de la clase **Methods***

Tipo	Declaración	Acción
public	Methods(Node n, Document docu)	
public	Methods()	

*Métodos de la clase **Methods***

Tipo	Declaración	Acción
public void	dom2data()	
protected Element	createElement(String nameNode)	
protected Element	createElement(String nameNode, String value)	
protected void	createSMElement(String name)	
protected void	createCSElement(String name)	
protected void	createIOElements()	
public void	data2dom( )	
protected void	loadElement()	
protected void	loadClientServer(Node n, String rol)	
protected void	loadInputOutput(Node nn)	
protected void	loadPlainData(Node sons)	
protected void	loadModelData(Node nn)	

*Constructores de la clase **OtherSkill***

Tipo	Declaración	Acción
public	OtherSkill(Node n, Document docu)	
public	OtherSkill()	

*Métodos de la clase **OtherSkill***

Tipo	Declaración	Acción
public void	data2dom()	

*Constructores de la clase **FilterSkill***

Tipo	Declaración	Acción
public	FilterSkill(Node n, Document docu)	
public	FilterSkill()	

*Métodos de la clase **FilterSkill***

Tipo	Declaración	Acción
public void	data2dom()	

*Constructores de la clase **ServiceSkill***

Tipo	Declaración	Acción
public	ServiceSkill(Node n, Document docu)	
public	ServiceSkill()	

*Métodos de la clase **ServiceSkill***

Tipo	Declaración	Acción
public void	dom2data()	

*Constructores de la clase **FunctionSkill***

Tipo	Declaración	Acción
public	FunctionSkill(Node n, Document docu)	
public	FunctionSkill()	

*Métodos de la clase **FunctionSkill***

Tipo	Declaración	Acción
public void	dom2data()	

Diagrama de secuencia del modelo de datos de gestión del protocolo partiendo de DOM

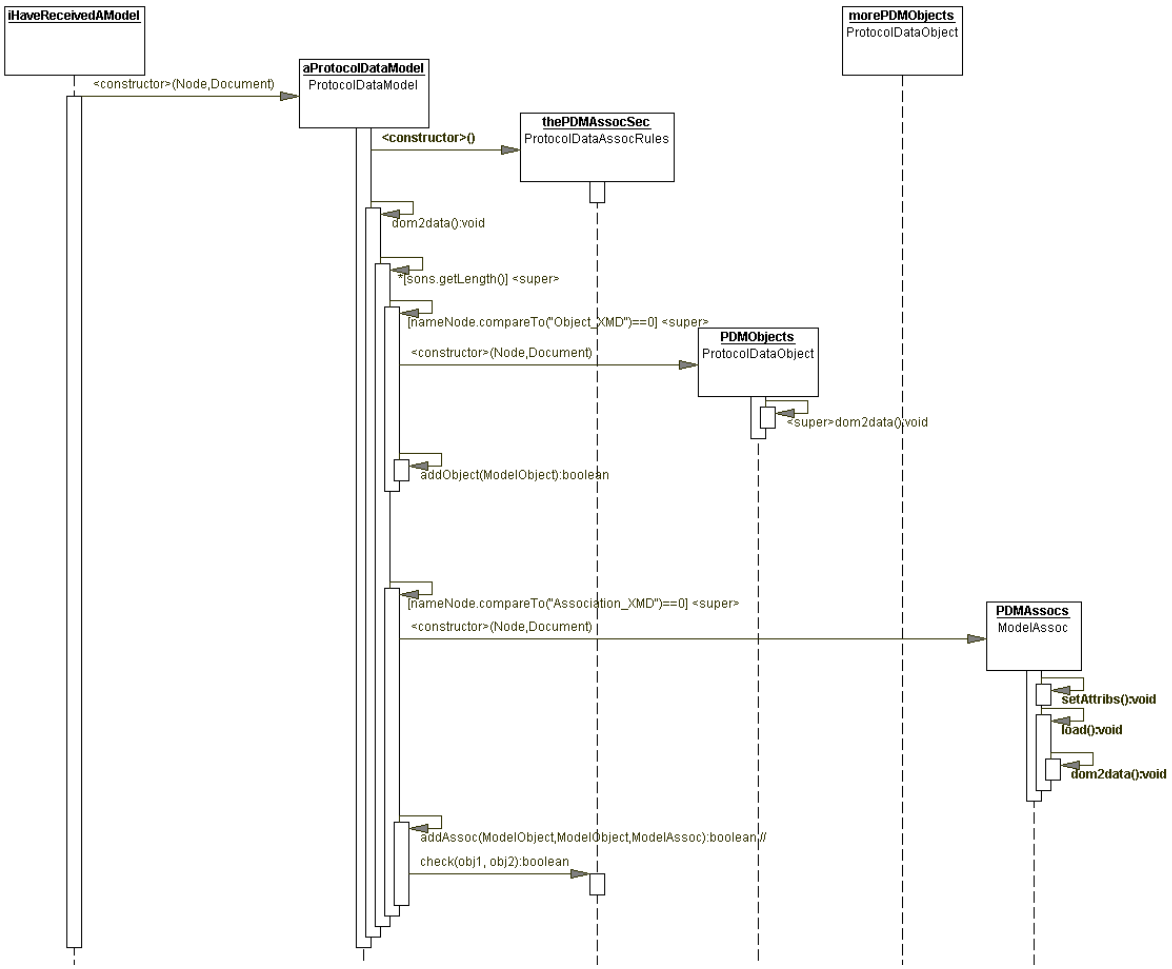


Diagrama de secuencia de la creación de un modelo de datos de gestión del protocolo a partir de su representación DOM, y de la creación del grafo, convirtiendo los elementos a su representación interna

Diagrama de secuencia de la creación de un grafo en el modelo de gestión de protocolo

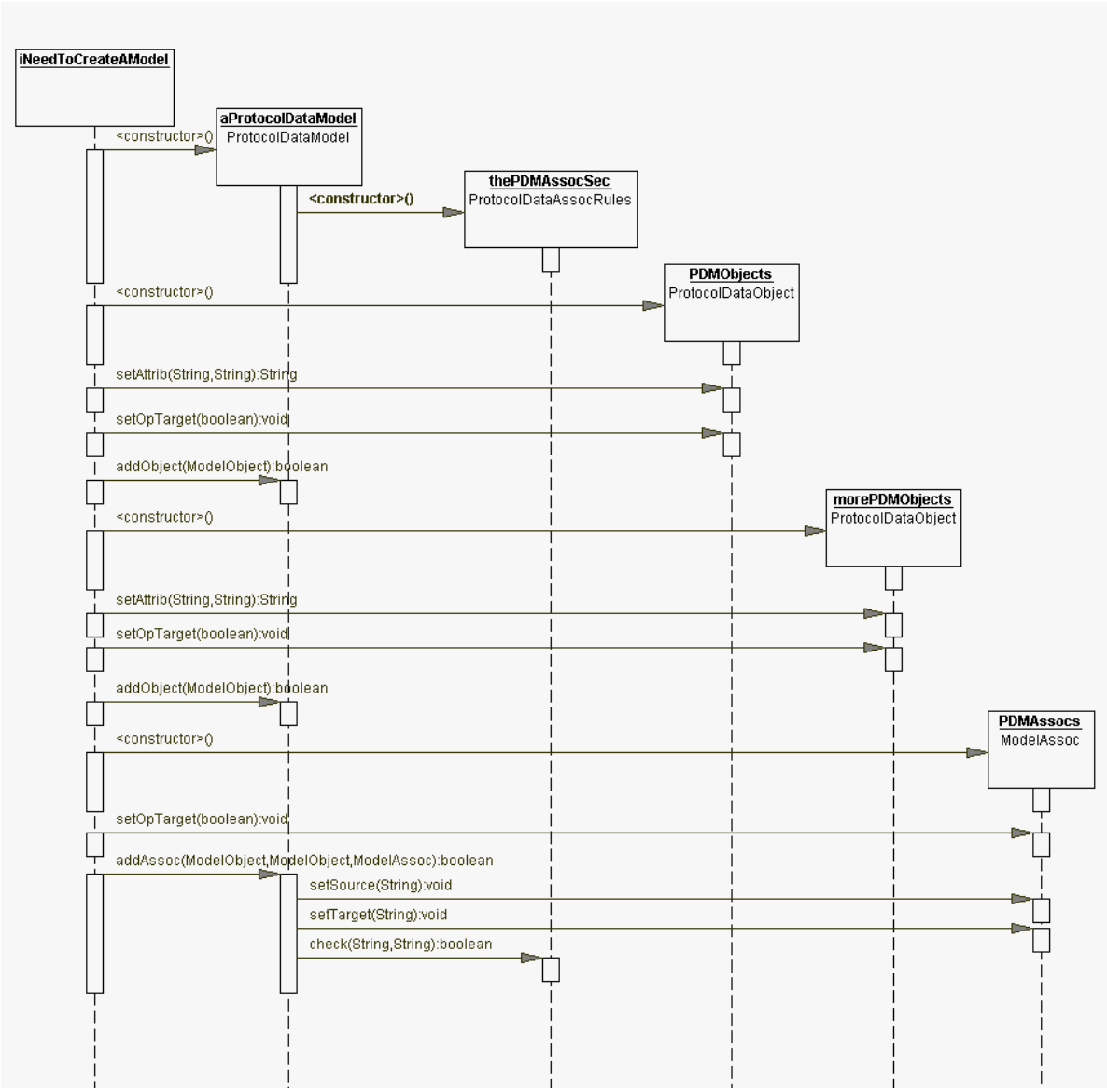


Diagrama de secuencia de la creación de un grafo de un modelo de datos de gestión del protocolo, creando objetos y asociaciones e insertándolos en el grafo

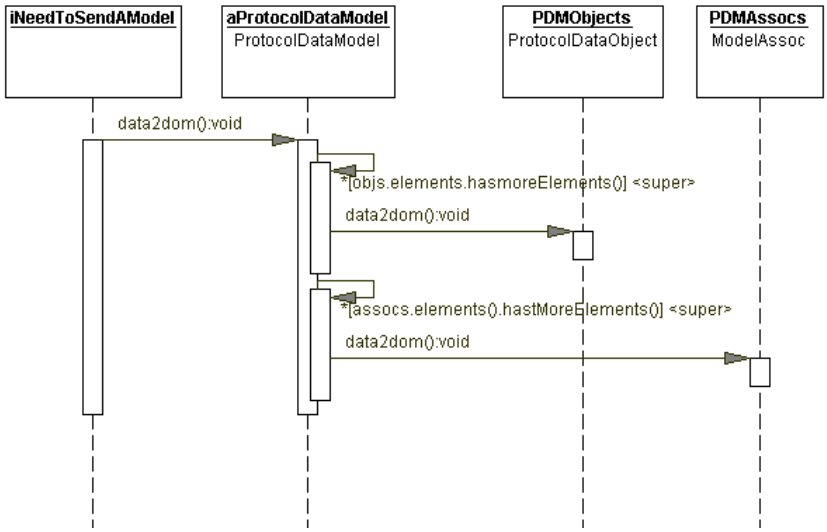


Diagrama de la secuencia de la conversión de un modelo de datos de gestión del protocolo y los elementos que contiene a su representación DOM

6.3 Registro de pruebas unitarias

Prueba	Prueba del modelo de datos del protocolo.
Clase ejecutable	XMDCarga
Clases implicadas	utils.DOMSource, utils.DOMizable, utils.ConfigFile, utils.NodeLog, todas las clases del paquete modeldata y las clases del paquete protocoldata.
Procedimiento	Se crea un fichero XML conteniendo un modelo de datos de gestión del protocolo, con numerosos objetos y asociaciones. Se utilizan las clases del paquete modeldata y las del paquete protocoldata, para cargar este fichero y en la clase propia del modelo del protocolo la cual formará el grafo. En la creación del grafo se valida que los objetos y las asociaciones insertadas, son correctos y están permitidas. Si se localizan objetos no existentes o asociaciones no permitidas, se rechazan y se escriben en el archivo de registro. A continuación se realiza el proceso inverso y se genera una representación XML de la estructura que hemos obtenido, que se almacena como una copia del fichero original. En el fichero original se incluyen algunas asociaciones que no están permitidas en el modelo.

Resultado deseado	Se debe obtener una copia del fichero original, pero no deben aparecer las asociaciones no permitidas. En el fichero de registro deben aparecer las notificaciones de las asociaciones no permitidas encontradas.
Resultado obtenido	Se ha construido correctamente el grafo, y la obtención del fichero XML es correcta.
Errores encontrados	Sin errores.

Prueba	Prueba del modelo de datos MECANO.
Clase ejecutable	REMCarga
Clases implicadas	utils.DOMSource, utils.DOMizable, utils.ConfigFile, utils.NodeLog, todas las clases del paquete modeldata y las clases del paquete protocoldata.
Procedimiento	Se ha seguido el mismo procedimiento que en el caso anterior, pero utilizando un fichero XML conteniendo un modelo de MECANO.
Resultado deseado	Una copia del fichero original sin las asociaciones no permitidas, de las que se habrá notificado en el fichero de registro.
Resultado obtenido	Se ha obtenido el resultado esperado.
Errores	encontrados Ninguno.

Prueba	Prueba de la clase Talk.
Clase ejecutable	CargaTalk
Clases implicadas	utils.DOMSource
Procedimiento	En estas últimas pruebas se han apoyado las anteriores. Cargamos de un fichero en el que hemos formado un elemento Talk del modelo de protocolo, lo cargamos en memoria con clase constructor, obteniendo su representación DOM y a partir de está visualizamos el atributo de todos los elementos que forman la clase y nuevamente el fichero XML.
Resultado deseado	Una copia del fichero original.
Resultado obtenido	Se ha obtenido el resultado esperado.
Errores	encontrados Ninguno.

En la documentación podemos ver que se han hecho muchísimas pruebas con todas las clases que forman parte del modelo de reutilización y del modelo del protocolo. Las pruebas Unitarias serían del mismo estilo que las última mostrada. Además de construir la clase podemos ver como se han implementado los método *xml2dom* y *dom2xml*.

Capítulo 7

Iteración 4: Estructuras de datos de los mensajes

7.1 Requisitos

Número	reqMen01
Descripción	Se deben diseñar estructuras de datos que representen las diferentes partes que componen la estructura de los mensajes X-MIP: “Message”, “Frame”, “Request”, “Result”, “Dialog” y “PlainData”.
Prioridad	Alta

Número	reqMen02
Descripción	Al igual que ocurría en el caso de los datos de tipo “modelo”, las estructuras que representen las partes de los mensajes tienen que poder soportar su representación DOM. Esto implica poder construir su representación interna a partir de su representación DOM y viceversa.
Prioridad	Alta

Número	reqMen03
Descripción	Se tienen que diseñar para estas estructuras, métodos que permitan crear nuevos mensajes fácilmente. Esto significa que el usuario (o función), que desee crear un mensaje, necesite conocer lo menos posible acerca de la estructura de los mismos, y sólo necesite aportar la información mínima necesaria y en un formato más natural posible.
Prioridad	Alta

Número	reqMen04
Descripción	Se tienen que diseñar las estructuras para que sea sencillo extraer la información que contienen, por parte de las distintas funciones internas del nodo que las utilicen. Por ejemplo, en el caso de las peticiones (“Request”), sería útil que ofrecieran operaciones para ir extrayendo las peticiones individuales según el orden especificado por la secuencia de ejecución. Esta operación podría ser utilizada por las funciones que se encarguen de procesar la petición.
Prioridad	Alta

7.2 Diseño

7.2.1 Descripción

Para desarrollar la representación de la estructura de un mensaje, simplemente se han creado clases para los grandes bloques en los que se compone un mensaje:

- Mensaje (Message)
- Cabeceras (MIPHeader)
- Bloques de petición (Frame)
- Peticiones (Request, especialización de Frame)
- Resultados (Result, especialización de Frame)
- Diálogos (Dialog, especialización de Frame)
- Datos simples (PlainData)

Se han diseñado las estructuras para que los mensajes se puedan construir en cascada tanto al enviar como al recibir. La diferencia reside en que al recibir, la representación interna se construye desde el elemento mayor hasta el elemento menor, mientras que al construir un mensaje para enviar, este se construye desde el elemento menor hasta el mayor.

Las estructuras implementan las operaciones de las clases relacionadas con XML, del paquete de utilidades (utils): DOMizable y DOMSource. En estas operaciones, cada clase implementa el modo de generar su propia representación, entre XML, DOM y representación interna. Mientras que todas las clases son capaces de 'entender' y generar su representación DOM, la única clase capaz de trabajar con la representación XML, es la clase Message, que representa un mensaje. Al ser la clase que contiene a todas las demás y ser trivial la traducción entre DOM y XML.

En esta iteración se tienen presentes los cambios realizados en las iteraciones anteriores, se implementan todos los métodos, se redefinen las maneras de crear los objetos, se añaden funcionalidades a las clases para gestionarlas de manera más cómoda y se consiguen cargar los distintos tipos de mensajes definidos, partiendo de elementos más pequeños que lo forman.

Clases relacionadas

public class **Message** extends **DOMizable**	Representación de un mensaje del protocolo X-MIP.
public class **FrameBuilder** extends **DOMizable**	Clase que se encarga de instanciar un tipo concreto de frame a partir de su representación DOM, extrayendo de esta representación la información necesaria para poder determinar el tipo de que se trata.
public class **MIPHeader** extends **DOMizable**	Representación de una cabecera del protocolo X-MIP.
abstract public class **Frame** extends **DOMizable**	Representación de un frame genérico del protocolo X-MIP.
public class **ReqString** extends **DOMizable**	Representación de la definición de una secuencia de ejecución de servicios. Que se utiliza para especificar cómo deben ejecutarse las peticiones individuales contenidas en un objeto "Request".
public class **Request** extends **Frame**	Representación de un frame de tipo "Request" del protocolo X-MIP.
public class **Dialog** extends **Frame**	Representación de un frame de tipo "Dialog" del protocolo X-MIP.
public class **Result** extends **Frame**	Representación de un frame de tipo "Result" del protocolo X-MIP.
public class **PlainData** extends **DOMizable**	Representación de los datos simples que pueden intercambiarse como argumentos, resultados, ... etc., mediante el protocolo X-MIP.
public class **SingleRequest** extends **DOMizable**	Petición de servicios individual, de las que se componen los objetos "Request".
public class **ResToArg** extends **PlainData**	Tipo especial de datos simples utilizado para definir en un servicio, argumentos que toman su valor de los resultados de otro servicio. Se conocen como argumentos encadenados.

Diagramas relacionados

	Diagrama de clases de las estructuras de datos que representan la estructura de los mensajes.
	Diagrama de secuencia del proceso de recibir un mensaje y solicitar su conversión, de su representación XML a su representación interna.
	Diagrama de secuencia del proceso de solicitar la conversión de una cabecera, de su representación DOM a su representación interna.
	Diagrama de secuencia del proceso de solicitar la conversión de un frame de petición, de su representación DOM a su representación

	interna.
	Diagrama de secuencia del proceso de solicitar la conversión de un frame de resultados, de su representación DOM a su representación interna.
	Diagrama de secuencia del proceso de solicitar la conversión de un frame diálogo, de su representación DOM a su representación interna.
	Diagrama de secuencia del proceso d extraer de un frame de petición, las peticiones individuales que la componen, siguiendo la secuencia de ejecución que tenga establecida.

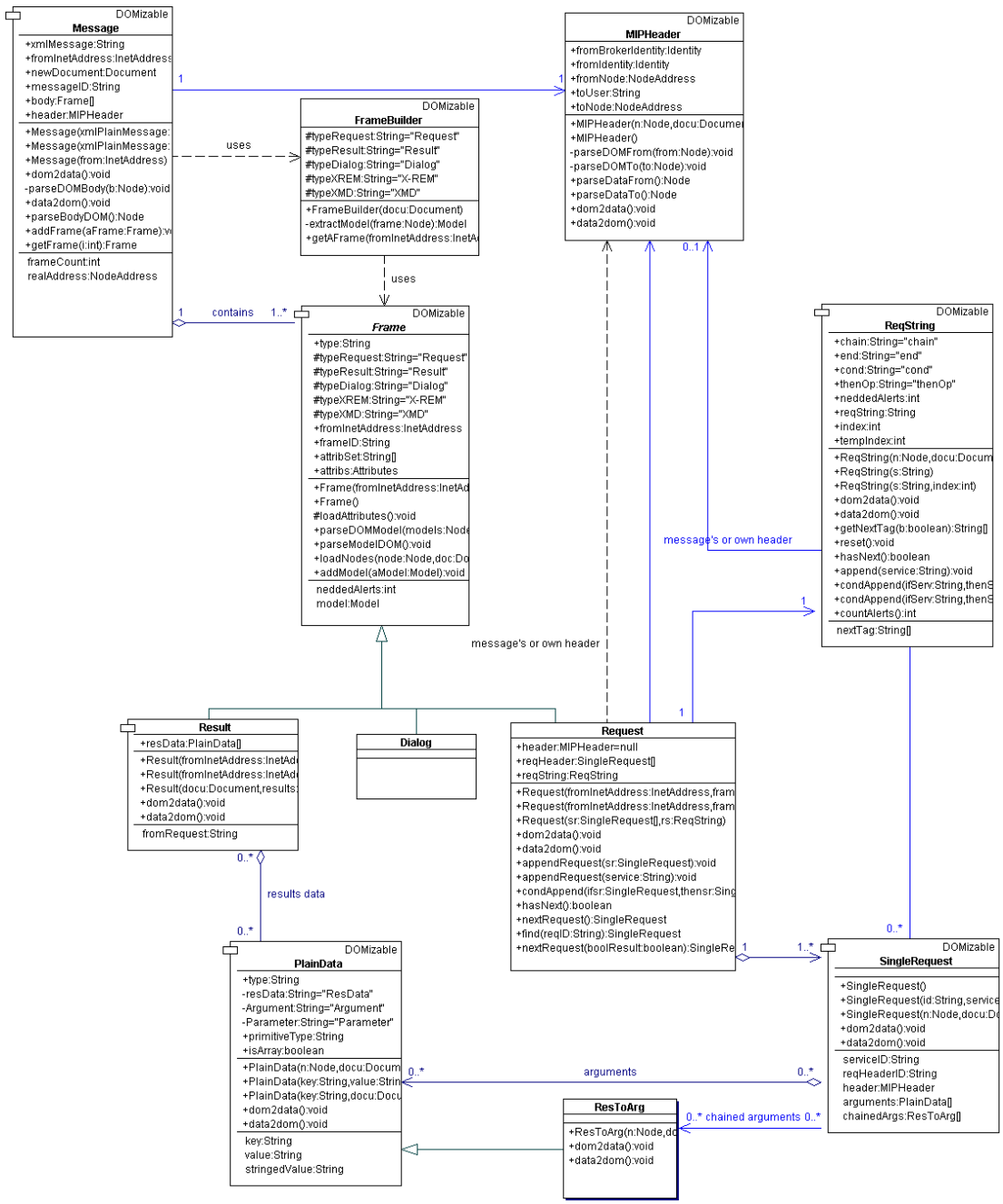


Diagrama de clases de las estructuras de datos que representan la estructura de los mensajes

*Campos de la clase **Message***

Tipo	Declaración	Acción
public String	xmlMessage	
public InetAddress	fromInetAddress	
public String	messageID	
public Frame[]	body	
public MIPHeader	header	
public NodeAddress	realAddress	

*Constructores de la clase **Message***

Tipo	Declaración	Acción
public	Message(String xmlPlainMessage)	
public	Message (String xmlPlainMessage, InetAddress from)	
public	Message(InetAddress from)	

*Métodos de la clase **Message***

Tipo	Declaración	Acción
public void	dom2data()	
private void	parseDOMBody(Node b)	
public void	data2dom()	
public Node	parseBodyDOM()	
public void	addFrame(Frame aFrame)	
public Frame	getFrame(int i)	
public void	setRealAddress(NodeAddress add)	
public int	getFrameCount()	

*Campos de la clase **FrameBuilder***

Tipo	Declaración	Acción
protected String	typeRequest	
protected String	typeResult	
protected String	typeDialog	
protected String	typeXREM	
protected String	typeXMD	

Constructores de la clase ***FrameBuilder***

Tipo	Declaración	Acción
public	FrameBuilder(Document docu)	

Métodos de la clase ***FrameBuilder***

Tipo	Declaración	Acción
private Model	extractModel(Node frame)	
public Frame	getAFrame (InetAddress fromInetAddress, Node frame)	

Campos de la clase ***MIPHeader***

Tipo	Declaración	Acción
public Identity	fromBrokerIdentity	
public Identity	fromIdentity	
public NodeAddress	fromNode	
public String	toUser	
public NodeAddress	toNode	

Constructores de la clase ***MIPHeader***

Tipo	Declaración	Acción
public	MIPHeader(Node n, Document docu)	
public	MIPHeader()	

Métodos de la clase ***MIPHeader***

Tipo	Declaración	Acción
private void	parseDOMFrom(Node from)	
private void	parseDOMTo(Node to)	
public Node	parseDataFrom()	
public Node	parseDataTo()	
public void	dom2data()	
public void	data2dom()	

Campos de la clase ***Frame***

Tipo	Declaración	Acción
public String	type	
protected String	typeRequest	
protected String	typeResult	

protected String	typeDialog	
protected String	typeXREM	
rotected String	typeXMD	
public InetAddress	fromInetAddress	
public String	frameID	
public String[]	attribSet	
public Attributes	attribs	
protected int	neddedAlerts	
public Model	model	

Constructores de la clase ***Frame***

Tipo	Declaración	Acción
public	Frame(InetAddress fromInetAddress, String frameID, String type)	
public	Frame()	

Métodos de la clase ***Frame***

Tipo	Declaración	Acción
public int	getNeddedAlerts()	
protected void	loadAttributes()	
public void	parseDOMModel(Node models)	
public void	parseModelDOM()	
public void	loadNodes(Node node, Document doc)	
public void	addModel(Model aModel)	
public Model	getModel()	
public void	setModel(Model m)	

Campos de la clase ***ReqString***

Tipo	Declaración	Acción
public String	chain	
public String	end	
public String	cond	
public String	thenOp	
public int	neddedAlerts	
public String	reqString	
public int	index	
public int	tempIndex	

Constructores de la clase ***ReqString***

Tipo	Declaración	Acción
public	ReqString(Node n, Document docu)	
public	ReqString(String s)	
public	ReqString(String s, int index)	

Métodos de la clase ***ReqString***

Tipo	Declaración	Acción
public void	dom2data()	
public void	data2dom()	
public String[]	getNextTag()	
public String[]	getNextTag(boolean b)	
public void	reset()	
public boolean	hasNext()	
public void	append(String service)	
public void	condAppend (String ifServ, String thenServ, String elseServ)	
public void	condAppend(String ifServ, String thenServ)	
public int	countAlerts()	

Campos de la clase ***Request***

Tipo	Declaración	Acción
public MIPHeader	header	
public SingleRequest[]	reqHeader	
public ReqString	reqString	

Constructores de la clase ***Request***

Tipo	Declaración	Acción
public	Request(InetAddress fromInetAddress, String frameID, Node node, Document docu)	
public	Request (InetAddress fromInetAddress, String frameID)	
public	Request (SingleRequest[] sr, ReqString rs)	

Métodos de la clase ***Request***

Tipo	Declaración	Acción

public void	dom2data()	
public void	data2dom()	
public void	appendRequest(SingleRequest sr)	
public void	appendRequest(String service)	
public void	condAppend(SingleRequest ifsr, SingleRequest thensr, SingleRequest elsesr)	
public boolean	hasNext()	
public SingleRequest	nextRequest()	
public SingleRequest	find(String reqID)	
public SingleRequest	nextRequest(boolean boolResult)	

Campos de la clase ***Result***

Tipo	Declaración	Acción
private String	fromRequest	
public PlainData[]	resData	

Constructores de la clase ***Result***

Tipo	Declaración	Acción
public	Result (InetAddress fromInetAddress, String frameID, Node node, Document docu)	
public	Result (InetAddress fromInetAddress, String frameID, String fromReq)	
public	Result(Document docu, PlainData[] results, String fromReq)	

Métodos de la clase ***Result***

Tipo	Declaración	Acción
public void	dom2data()	
public void	data2dom()	
public String	getFromRequest()	

Campos de la clase ***PlainData***

Tipo	Declaración	Acción
public String	type	
private String	resData	
private String	Argument	
private String	Parameter	
public String	key	
public String	value	

public String	stringedValue	
public String	primitiveType	
public boolean	isArray	

Constructores de la clase ***PlainData***

Tipo	Declaración	Acción
public	PlainData (Node n, Document docu, String type)	
public	PlainData (String key, String value, String tprimitive, Document docu,String type)	
public	PlainData (String key, Document docu)	

Métodos de la clase ***PlainData***

Tipo	Declaración	Acción
public void	dom2data()	
public void	data2dom()	
public void	setKey(String key)	
public void	setValue(String value)	
public String	getKey()	
public String	getValue()	
public String	getStringedValue()	

Campos de la clase ***SingleRequest***

Tipo	Declaración	Acción
public String	serviceID	
public String	reqHeaderID	
public MIPHeader	header	
public PlainData[]	arguments	
public ResToArg[]	chainedArgs	

Constructores de la clase ***SingleRequest***

Tipo	Declaración	Acción
public	SingleRequest()	
public	SingleRequest(Node n, Document docu)	
public	SingleRequest(String id, String service, PlainData[] args, ResToArg[] chained, MIPHeader h)	

Métodos de la clase ***SingleRequest***

Tipo	Declaración	Acción
public void	dom2data()	
public void	data2dom()	
public void	setServiceID(String service)	
public void	setReqHeaderID(String reqHead)	
public void	setHeader(MIPHeader h)	
public void	setArguments(PlainData[] pd)	
public void	setChainedArgs(ResToArg[] ra)	
public String	getServiceID()	
public String	getReqHeaderID()	
public MIPHeader	getHeader()	
public PlainData[]	getArguments()	
public ResToArg[]	getChainedArgs()	

Constructores de la clase ***ResToArg***

Tipo	Declaración	Acción
public	ResToArg(Node n, Document docu)	

Métodos de la clase ***ResToArg***

Tipo	Declaración	Acción
public void	dom2data()	
public void	data2dom()	

Diagrama de secuencia del proceso de recibir un mensaje y solicitar su conversión

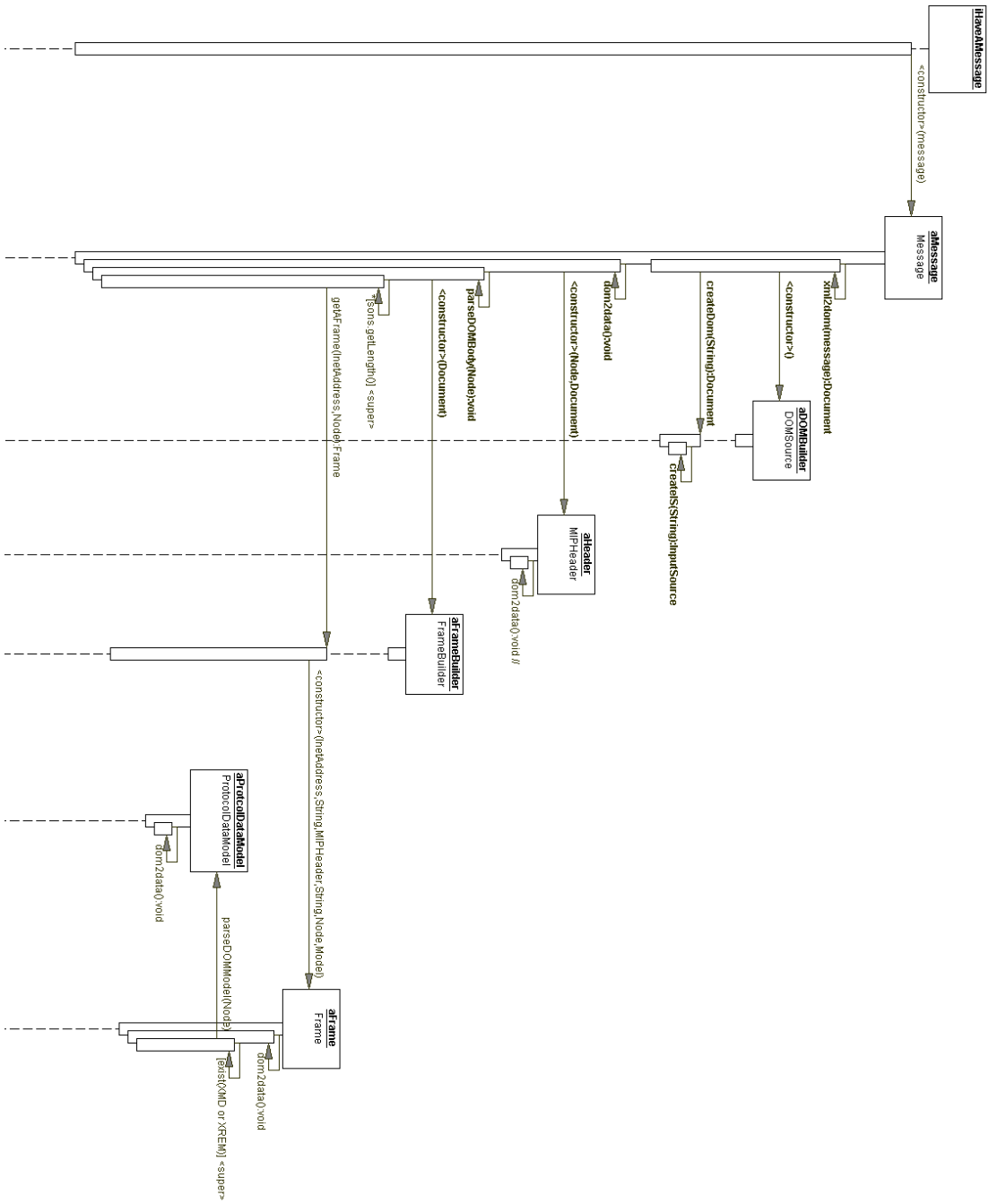


Diagrama de secuencia del proceso de recibir un mensaje y solicitar su conversión, de su representación XML a su representación interna

Diagrama de secuencia de la conversión de una cabecera

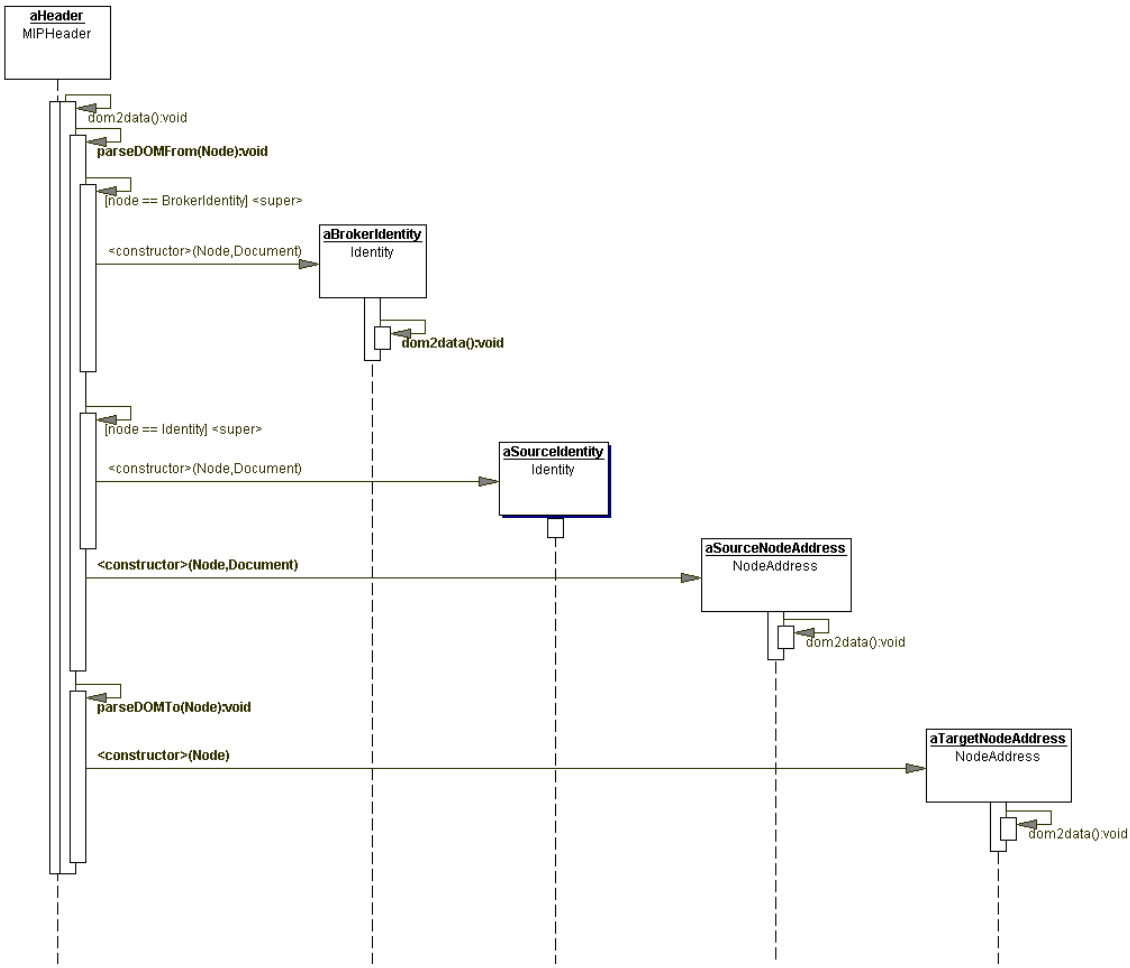


Diagrama de secuencia del proceso de solicitar la conversión de una cabecera, de su representación DOM a su representación interna

Diagrama de secuencia para un frame request

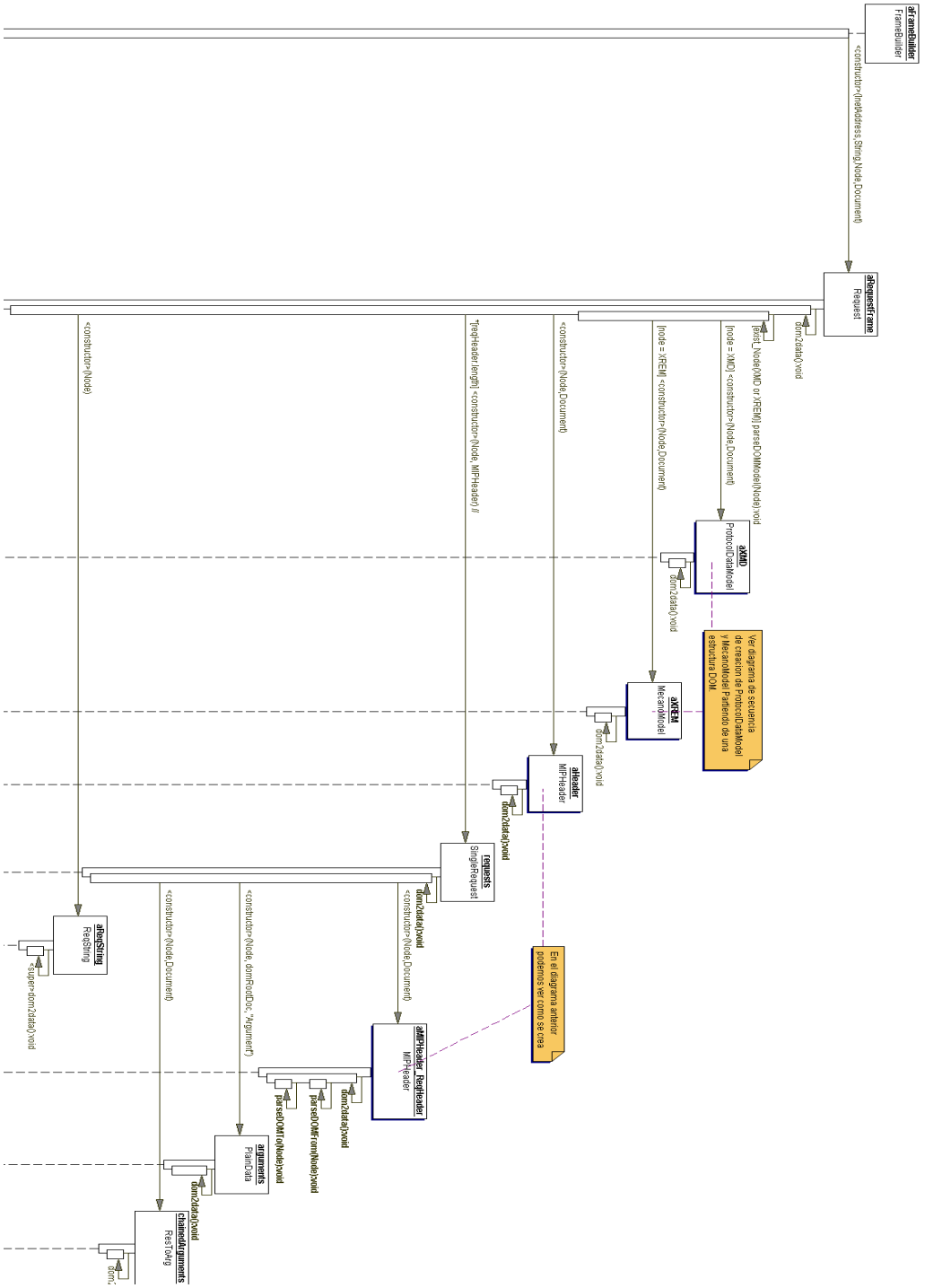


Diagrama de secuencia del proceso de solicitar la conversión de un frame de petición, de su representación DOM a su representación interna

Diagrama de secuencia para un frame result

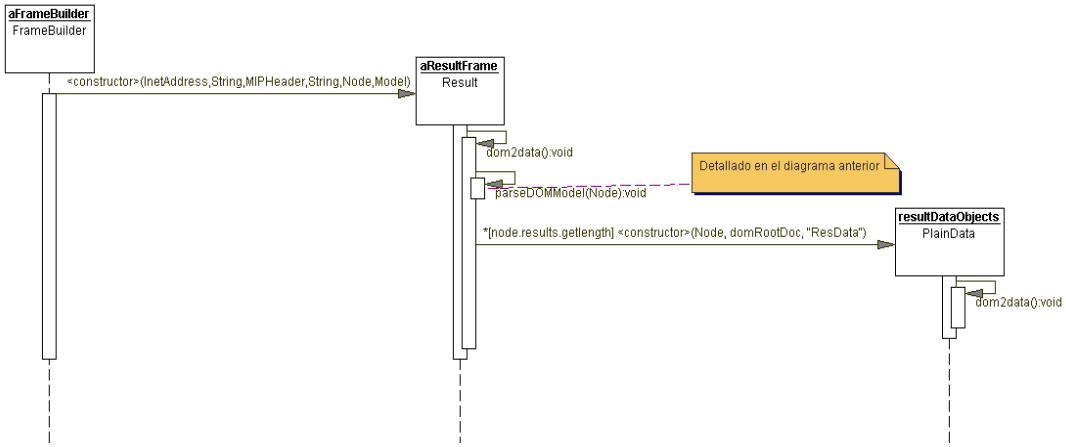


Diagrama de secuencia del proceso de solicitar la conversión de un frame de resultados, de su representación DOM a su representación interna

Diagrama de secuencia para un frame Dialog

Este diagrama es copiado del proyecto anterior, pues no se ha llegado a implementar aunque de manera funcional está documentado, siguiendo los pasos de los otros dos tipos de frames el funcionamiento de éste se deja abierto al desarrollador.

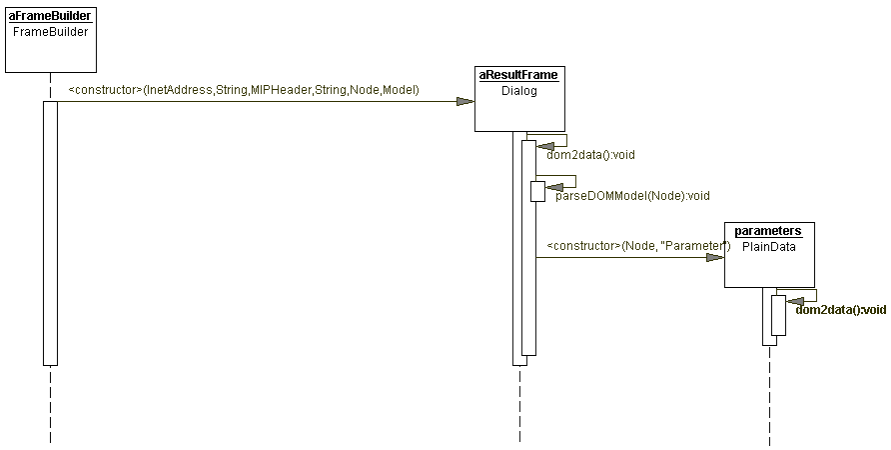


Diagrama de secuencia del proceso de solicitar la conversión de un frame diálogo, de su representación DOM a su representación interna

Diagrama de secuencia para la extracción de frames individuales

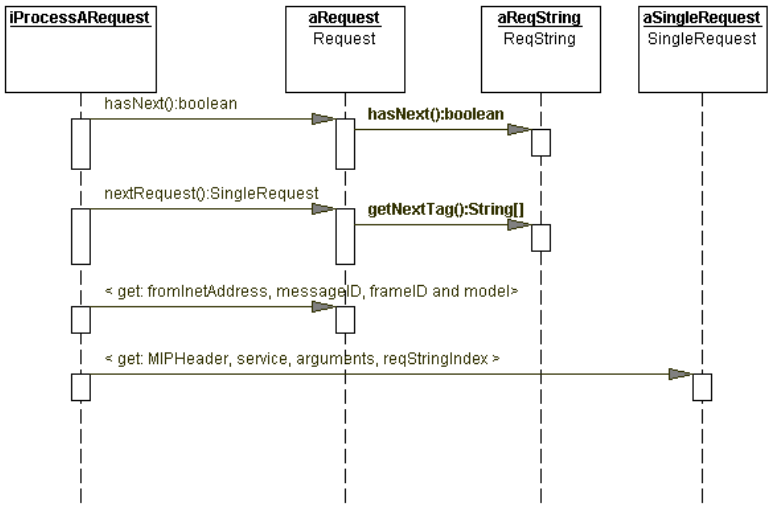


Diagrama de secuencia del proceso d extraer de un frame de petición, las peticiones individuales que la componen, siguiendo la secuencia de ejecución que tenga establecida

7.3 Registro de pruebas unitarias

Prueba	Carga de un mensaje.
Clase ejecutable	XMIPCarga
Clases implicadas	utils.NodeLog, utils.ConfigFile, nodedata.Message,
Procedimiento	Pasamos por ficheros la estructura de un mensaje, este lo que hará en construir la clase Message, que formará todas las clases necesarias para cargar el fichero, así mismo se procederá a cargar los modelos de protocolo y de reutilización. Visualizaremos por pantalla valores procedentes de la carga de los distintas clases cargadas.
Resultado deseado	Visualización de los datos ubicados, en el mensaje, en partes muy concretas.
Resultado obtenido	Resultado esperado
Errores encontrados	No había errores.

Prueba	Creación y envío de mensajes con frames de resultado.
Clase ejecutable	Clientes y Servidores
Clases implicadas	utils.NodeLog, utils.ConfigFile connection.Connection,

	connection.ConnectionFactory, connection.InputConnection, connection.OutputConnection, connection.DataConnection, tcpsocketconnection.TCPConnection, tcpsocketconnection.TCPInputConnection, tcpsocketconnection.TCPOutputConnection, nodedata.Message, nodedata.Frame, nodedata.FrameBuilder, nodedata.MIPHeader, nodedata.Result, nodedata.PlainData
Procedimiento	Se utilizan dos clases, una como cliente y otra como servidores, ejecutándose en procesos concurrentes. En la clase cliente se crea un mensaje conteniendo exclusivamente frames, que son enviados como cadenas de texto a la clase servidor. En esta otra clase, se reciben los mensajes y se obtiene su representación interna. A partir de ella se construyen, nuevos mensajes, pero con el mismo contenido, que se envía al cliente. Se almacenan los ficheros que reciben cada uno de los nodos.
Resultado deseado	Los ficheros generados tienen que recoger exactamente los mismos contenidos que se enviaron desde la primera clase.
Resultado obtenido	Los esperados.
Errores encontrados	No se encontraron.

Cada una de estas pruebas detalladas se han modificado, intentando utilizar el máximo posible de funciones que se han implementado. El detalle de todas pruebas hacen que la documentación sea todavía más larga, por lo que se decidió plasmarla de manera general.

.

Capítulo 8

Iteración 5: Núcleo de los nodos

8.1 Requisitos

Número	reqNúcleo01
Descripción	El núcleo del nodo tiene que utilizar todas las partes desarrolladas con anterioridad, para controlar y gestionar la ejecución de las siguientes acciones, que describen las fases necesarias para el procesamiento de los mensajes, tanto entrantes como salientes: ▪ Atender peticiones, aceptando las conexiones entrantes. ▪ Decodificar los mensajes, aplicando las cadenas de filtros necesarias, para obtenerlos en formato de texto plano ▪ Convertir los mensajes de su representación XML a una representación interna, que pueda ser utilizada por los diferentes módulos del nodo. ▪ Entregar el contenido de los mensajes, a los servicios y a las funciones correspondientes que puedan procesarlos, o a los procesos que los estén esperando. ▪ Generar automáticamente los resultados de los mensajes de petición. ▪ Reenviar automáticamente las peticiones que deban ser atendidas por otros nodos. ▪ Al enviar mensajes de los que se espere una respuesta, se debe preparar el nodo para esperar esta respuesta, sin bloquear ninguna de las funcionalidades del nodo. ▪ Convertir los mensajes a su representación en XML. ▪ Codificar los mensajes, aplicando las cadenas de filtros necesarias para ser enviados ▪ Enviar los mensajes salientes utilizando nuevas conexiones.
Prioridad	Alta

Número	reqNúcleo02
Descripción	El núcleo del nodo debe poder atender simultáneamente múltiples peticiones entrantes y salientes. Deben existir procesos independientes para: ▪ Recibir los mensajes ▪ Procesar los mensajes ▪ Esperar los mensajes de respuesta ▪ Enviar los mensajes
Prioridad	Alta

8.1 Requisitos

Número	reqNúcleo03
Descripción	Si un proceso envía un mensaje y espera una respuesta a éste, debe poder permanecer bloqueado a la espera de la respuesta sin que esto interfiera en el funcionamiento del nodo.
Prioridad	Alta

Número	reqNúcleo04
Descripción	Las funciones y los servicios que debe ofrecer el nodo deben poder configurarse y comprobarse al arrancar el nodo. El tipo de nodo de qué se trate, y los servicios y funciones que se hayan indicado en el fichero de configuración, determinarán cuáles se ofrecerán en el nodo.
Prioridad	Alta

Número	reqNúcleo05
Descripción	Las funciones y los servicios que estén disponibles en un nodo, deben poder cargarse dinámicamente cuando cualquier módulo del nodo lo solicite. Cualquier módulo del nodo debe poder acceder a cualquier función ▪ Servicio de los que estén disponibles en el nodo.
Prioridad	Alta

8.2 Diseño

8.2.1 Descripción

El núcleo del framework, se ha planteado principalmente como un conjunto de clases de control que se encargan de establecer y gestionar las diferentes fases que constituyen el procesamiento de los mensajes.

La arquitectura del núcleo del nodo, representada en el diagrama de clases de este capítulo, recoge los principales componentes que componen el núcleo.

Se describen a continuación.

Servidor: Conjunto de etapas de recepción y procesamiento de los mensajes entrantes. Cada etapa se ejecuta en un proceso concurrente independiente y realiza una fase concreta del procesamiento de un mensaje. Las etapas son:

- Obtención del canal de comunicación entrante por dónde se recibe un mensaje.
- Decodificación del mensaje.
- Conversión del mensaje, de XML a una representación interna.
- Enrutado del mensaje si es que estamos en un nodo Broker y no va dirigido a él.
- Distribución del contenido del mensaje

Las tres últimas fases podrían incluirse en una única etapa, para reducir el número de procesos concurrentes en el sistema, lo que implicaría una reducción en el número de cambios de contexto (en un sistema monoprocesador) y por consiguiente una mejora en el rendimiento del nodo. A pesar de ello se han mantenido separadas, ya que al tratarse el presente trabajo de un desarrollo 'experimental', se ha preferido mantener las fases bien diferenciadas y en etapas independientes, para facilitar la comprensión del diseño de la arquitectura del sistema, frente a la mejora del rendimiento del mismo.

Cliente: Conjunto de etapas de procesamiento y envío de los mensajes salientes. Cada etapa se ejecuta en un proceso concurrente independiente y realiza una fase concreta del envío de un mensaje. Las etapas son:

- Conversión del mensaje, de su representación interna a su representación XML
- Codificación del mensaje.
- Creación del canal de comunicación por el que se enviará el mensaje.

Las dos últimas fases son realizadas por una única etapa. En realidad, debido a la manera en la que se realiza la codificación, el orden de estas fases es el inverso. La codificación no se aplica al mensaje, sino al canal de comunicación a través del que se transmite, por ello, primero se crea el canal de comunicación, a continuación se aplican filtros de codificación al canal, y finalmente se envía el mensaje.

Procesadores de peticiones: Por cada mensaje entrante, se creará un procesador de peticiones, que se ejecutará como un proceso concurrente independiente. Los procesadores de peticiones se encargarán de solicitar la ejecución de los servicios correspondientes para poder atender las peticiones Tomarán los resultados generados para enviarlos a sus destinatarios.

Gestionarán también la ejecución de peticiones que deban ser encadenadas entre varios nodos, reenviando las peticiones a los nodos que corresponda.

Cargador de servicios y funciones: Componentes que se encargan de obtener instancias de las funciones y de los servicios disponibles en el sistema, para que puedan ser utilizados por cualquier módulo del sistema que lo necesite. Junto a estos mecanismos, se utiliza también una definición genérica de servicios y funciones, dónde se declaran las interfaces principales de estos. Los cargadores de servicios y funciones, junto con las definiciones

genéricas de estos, permiten que el acceso al conjunto de la funcionalidad del nodo se efectúe de un modo homogéneo desde cualquier módulo y proceso del sistema.

Funciones: Conjunto de funciones disponibles en el nodo. Las funciones se refieren tanto a funciones internas como a servicios. Todas las funciones y servicios tienen cada una su propia interfaz, aunque presentan algunas operaciones definidas de modo genérico para poder ofrecer a los diferentes módulos y procesos del nodo una interfaz homogénea.

Servicios: Conjunto de servicios disponibles en el nodo. Además de la interfaz establecida en la definición genérica de funciones, la definición de un servicio, aporta la interfaz para aceptar solicitudes que llegan mediante mensajes desde otros nodos.

Buzones de envío de mensajes y recepción de respuestas: Componentes dónde se depositan los mensajes que son enviados al exterior. Si al enviar un mensaje, se espera recibir una respuesta, los buzones ofrecen la funcionalidad de 'esperar' esta respuesta. Para ello, se lo notifican al distribuidor de mensajes, y le proporcionan la información necesaria para que les distribuya los mensaje que están esperando. Cuando se espera una respuesta, los buzones se bloquean hasta que ésta es recibida. Cada proceso, ya se interno del nodo o externo, que necesite enviar un mensaje, poseerá y utilizará su propio buzón.

Procesos de usuario: No forma parte del núcleo del nodo, propiamente dicho. Se ha incluido este componente en el diagrama, para representar la interacción con el nodo de otros procesos no pertenecientes a él.

En esta iteración se compactan todos los elementos y características que se han definido hasta ahora, en ella llegamos a instaciar un nodo de la estructura del framework, con los servicios y funcionalidades que lo caracterizan.

Hemos tenido en cuenta todas las iteraciones anteriores, además hemos añadido una nueva etapa al servicio Servidor, encargada de enrutar, esta etapa sólo es para los Brokers. Se consigue mandar un mensaje, recibirlo y procesarlo ejecutando los servicios que solicita.

public abstract class **Node**	Representación de un nodo genérico. Contiene las estructuras y las operaciones que son comunes a todos los nodos y que se utilizan para configurar y arrancar el nodo. Se trata de la clase de control principal del nodo.
public class **ConnectionListener** extends **NodeStage**	Primera etapa del servidor del nodo, que se encarga de atender las peticiones que el nodo recibe, y de establecer a partir de ellas los canales de comunicación desde dónde se obtendrán los mensajes entrantes.
public class **CommunicationIn** extends **NodeStage**	Segunda etapa del servidor del nodo, que se encarga de decodificar los mensajes recibidos, aplicando entre otros, filtros de compresión y encriptación. Obtiene del canal de comunicación, mensajes ***X-MIP*** en texto plano para que puedan ser procesados por el nodo.
public class **MessageProcessor** extends **NodeStage**	Tercera etapa del servidor del nodo que se encarga de convertir los mensajes ***X-MIP***, de su representación ***XML***, a su representación interna para que puedan ser utilizados por el nodo.
public class **Message Router** extends **NodeStage**	Cuarta etapa del servidor del nodo, que se encarga de enrutar el nodo, si va dirigido para algún nodo de un broker o para el propio broker. Se pone en contacto con el broker al que va dirigido y si no lo conoce, al broker por defecto. Tiene asociada una pequeña tabla de enrutamiento.
public class **MessageDispatcher** extends **NodeStage**	Quinta y última etapa del servidor del nodo, que se encarga de extraer de los mensajes, los frames que estos contengan y de distribuirlos a las unidades funcionales correspondientes que se encargarán de procesar cada uno de ellos.
public class **MessageBuilder** extends **NodeStage**	Primera etapa del cliente del nodo, que se encarga de preparar un mensaje de la manera más adecuada para ser enviado, por ejemplo eliminando información redundante, y de convertirlo a su representación ***XML***.
public class **CommunicationOut** extends **NodeStage**	Segunda y última etapa del cliente del nodo, que se encarga de aplicar una cadena de filtros al mensaje, para codificarlos de la manera más apropiada posible, y de crear un canal de comunicación de salida por donde será enviado.
public class **RequestProcessor** implements **Runnable**	Procesador de peticiones, que se encarga de atender las peticiones contenidas en los frames de tipo 'Request', y de construir y enviar los resultados que se generen.
public class **Mailbox**	Buzón de salida dónde se depositan los mensajes para ser enviados.
public class **ReceiverMailbox** extends **Mailbox**	Buzón de recepción de mensajes, especialización del buzón normal, que al enviar un mensaje es capaz de permanecer a la espera de los mensajes de respuesta correspondientes.

public class **AlertObject**	Clase que contiene las estructuras necesarias para poder enviarle a un buzón receptor los frames que está esperando, y para poder avisarle cuando se haya hecho esto.
Public class **AlertsTable**	Contenedor de *AlertObject*, donde se almacenan junto con estos objetos, las referencias que son necesarias para identificar los *frames* entrantes y para relacionarlos con el objeto *AlertObject*, que debe ser utilizado para entregarlos a su destinatario.
public class **DialogProcessor** implements **Runnable**	Procesador de diálogos, que se encarga de hacer llegar a la función correspondiente, el mensaje de diálogo contenido en el frame, y en caso de que exista, se encarga también de construir y enviar la respuesta a este diálogo.
public class **FunctionLoader** extends **Loader**	Cargador de funciones, que se encarga de obtener instancias de las funciones que se le indiquen.
public class **ServiceLoader** extends **Loader**	Cargador de servicios, que se encarga de obtener instancia de los servicios que se le indiquen.
public class **WorkingSet**	Conjunto de datos que toma un servicio como entrada, y que está constituido por los datos propiamente dichos y por la información adicional que pueda ser útil al servicio, como por ejemplo la identidad del solicitante, la dirección IP desde la que se envió la petición, etc... .
public abstract class **Function**	Representación genérica de un servicio, donde se declara la operación que deben implementar todas las funciones, y por lo tanto, servicios del sistema.
public abstract class **Service** extends **Function**	Representación genérica de un servicio, dónde se declara la operación que deben implementar todos los servicios.
public class **ResultSet**	Conjunto de datos de resultado que genera un servicio, y que está constituido por un modelo y un conjunto de datos simples.
public class **OutgoingQueue**	Cola de salida de mensajes, dónde los buzones depositan los mensajes para que estos sean enviados por el nodo. Notifica a la primera etapa cliente, de la existencia de nuevos mensajes para ser enviados.
public class **Broker** extends **Node**	Especialización de un nodo, que se encarga de configurar y comprobar las funciones y servicios específicos de su tipo.
public class **Processor** extends **Node**	Especialización de un nodo, que se encarga de configurar y comprobar las funciones y servicios específicos de su tipo.
public class **Repository** extends **Node**	Especialización de un nodo, que se encarga de configurar y comprobar las funciones y servicios específicos de su tipo.
public class **Tool** extends **Node**	Especialización de un nodo, que se encarga de configurar y comprobar las funciones y servicios

	específicos de su tipo.
public class **FileBroker** extends **Node**	Especialización de un nodo, que se encarga de configurar y comprobar las funciones y servicios específicos de su tipo.

Diagramas relacionados

	Modelo estático: diagrama de clases.
	Diagrama de componentes del núcleo del nodo.
	Diagrama de secuencia para la creación de un nodo genérico.
	Diagrama de secuencia del proceso de creación de un nodo específico.
	Diagrama de secuencia del proceso de creación del servicio Servidor.
	Diagrama de secuencia del proceso de creación del servicio Cliente.
	Diagrama de Secuencia, del proceso general de recibir un mensaje en un nodo broker.
	Diagrama de Secuencia del proceso de crear y enviar un mensaje por parte del cliente.
	Diagrama de secuencia de la etapa ConnectionListener.
	Diagrama Secuencia de la ejecución CommunicationIn.
	Diagrama Secuencia de la ejecución MessageProcessor.
	Diagrama Secuencia de la ejecución MessageRouter.
	Diagrama Secuencia de la ejecución MessageDispatcher.
	Diagrama de secuencia del módulo encargado de procesar los mensajes de petición.
	Diagrama de secuencia de un mensaje del que se espera respuesta
	Diagrama de Secuencia de la etapa MessageBuilder
	Diagrama de Secuencia de la etapa CommunicationOUT

8.2 Diseño

Modelo estático: diagrama de clases

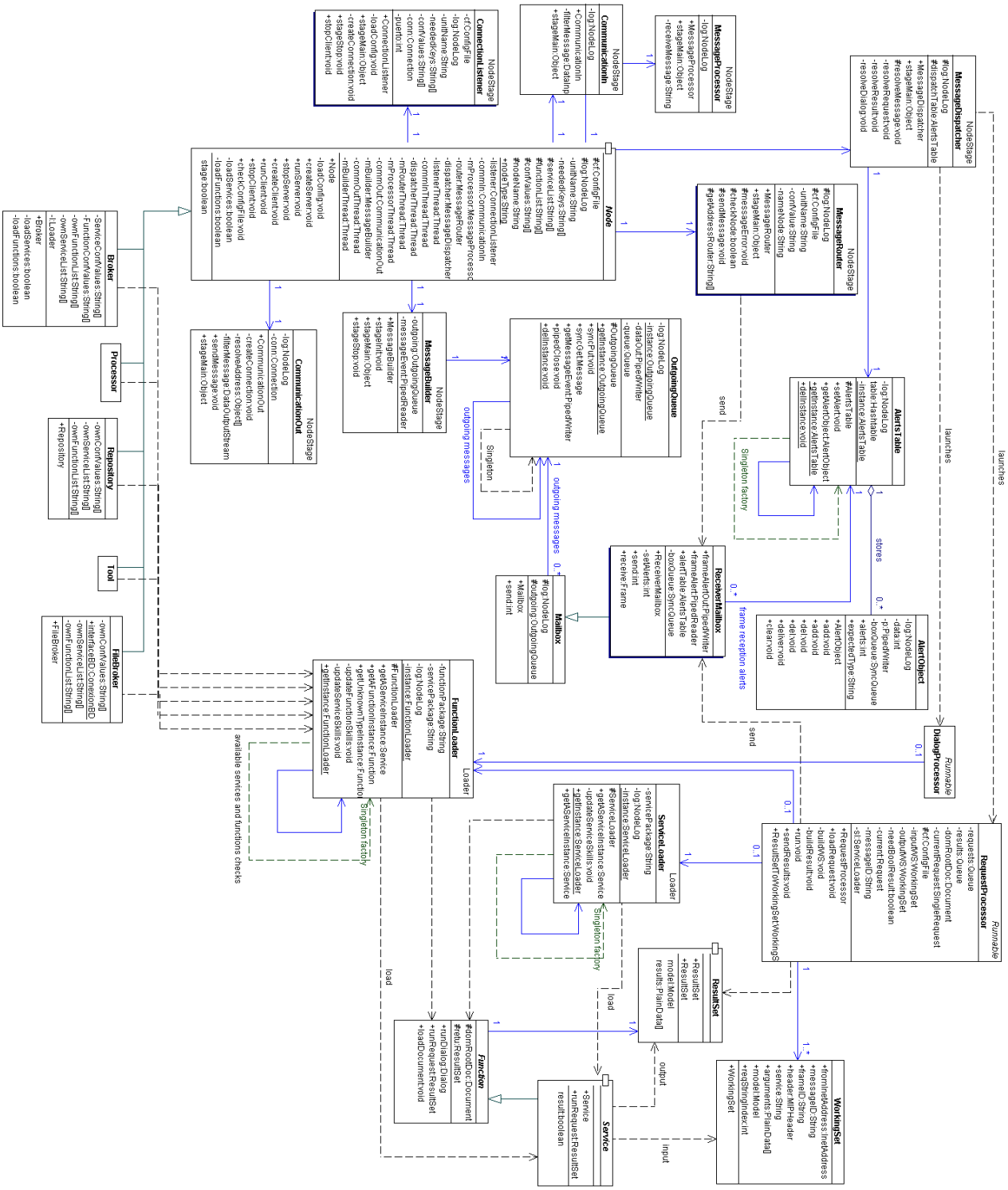


Diagrama de componentes del núcleo del nodo

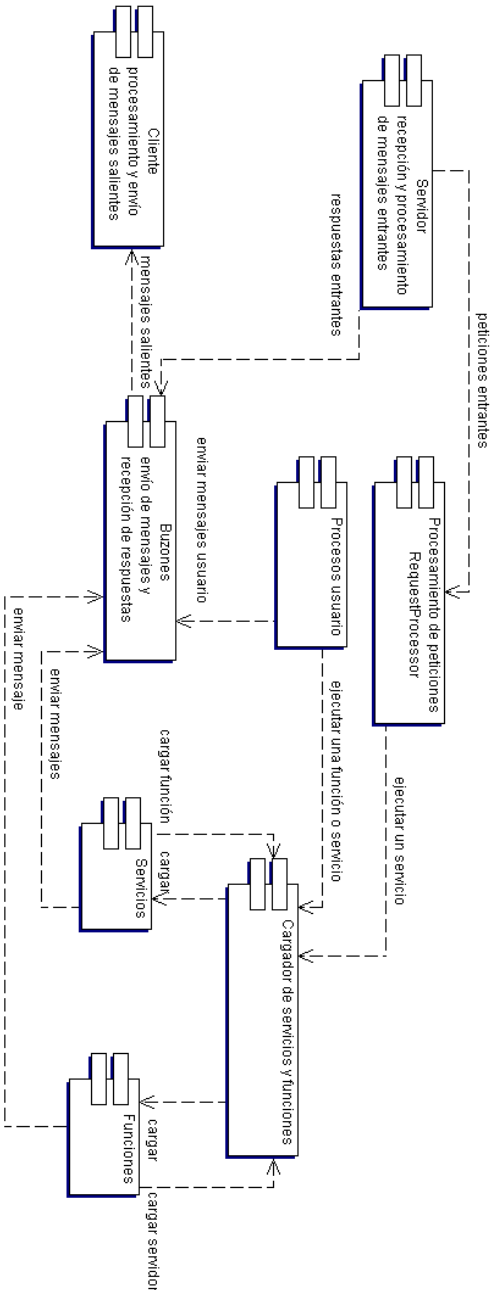


Diagrama de componentes del núcleo de nodos, con la representación de los principales bloques funcionales que constituyen el núcleo de los nodos.

Campos de la clase ***Node***

Tipo	Declaración	Acción
protected ConfigFile	cf	
protected NodeLog	log	
private String	unitName	
private String[]	neededKeys	
protected String[]	serviceList	
protected String[]	functionList	
protected String[]	confValues	
protected String	nodeName	
public static String	nodeType	
private ConnectionListener	listener	
private CommunicationIn	commIn	
private MessageProcessor	mProcessor	
private MessageRouter	router	
private MessageDispatcher	dispatcher	
private Thread	listenerThread	
private Thread	commInThread	
private Thread	dispatcherThread	
private Thread	mRouterThread	
private Thread	mProcessorThread	
private CommunicationOut	commOut	
private MessageBuilder	mBuilder	
private Thread	commOutThread	
private Thread	mBuilderThread	

Constructores de la clase ***Node***

Tipo	Declaración	Acción
public	Node (String type)	

Métodos de la clase ***Node***

Tipo	Declaración	Acción
private void	loadConfig()	
public void	createServer()	
public void	runServer()	

public void	stopServer()	
public void	createClient()	
public void	runClient()	
public void	stopClient()	
public boolean	getStage()	
public void	checkConfigFile()	
private boolean	loadServices()	
private boolean	loadFunctions()	

Campos de la clase ***ConnectionListener***

Tipo	Declaración	Acción
private ConfigFile	cf	
private NodeLog	log	
private String	unitName	
private String[]	neededKeys	
private String[]	confValues	
private Connection	conn	
private int	puerto	

Constructores de la clase ***ConnectionListener***

Tipo	Declaración	Acción
public	ConnectionListener(NodeStage nextStage)	

Métodos de la clase ***ConnectionListener***

Tipo	Declaración	Acción
private void	loadConfig()	
ublic Object	stageMain()	
private void	createConnection()	
public void	stageStop()	
public void	stopClient()	

Campos de la clase ***CommunicationIn***

Tipo	Declaración	Acción
private NodeLog	log	

Constructores de la clase ***CommunicationIn***

Tipo	Declaración	Acción

public	CommunicationIn(NodeStage nextStage)	

Métodos de la clase ***CommunicationIn***

Tipo	Declaración	Acción
private DataInputStream	filterMessage(InputStream input)	
public Object	stageMain(Object aConn)	

Campos de la clase ***MessageProcessor***

Tipo	Declaración	Acción
Private NodeLo	log	

Constructores de la clase ***MessageProcessor***

Tipo	Declaración	Acción
public	MessageProcessor(NodeStage nextStage)	

Métodos de la clase ***MessageProcessor***

Tipo	Declaración	Acción
public Object	stageMain(*Object* connect)	
private String	receiveMessage(*DataInputStream* data)	

Campos de la clase ***MessageRouter***

Tipo	Declaración	Acción
protected NodeLog	log	
protected ConfigFile	cf	
private String	nameNode	
private String	unitName	
private String	confValue	

Constructores de la clase ***MessageRouter***

Tipo	Declaración	Acción
public	MessageRouter (NodeStage nextStage)	

Métodos de la clase ***MessageRouter***

Tipo	Declaración	Acción
public Object	stageMain(Object aMessage)	
protected void	messageError(Message message)	
protected boolean	checkNode(String ip, int port)	

protected void	sendMessage(Message message)	
protected String[]	getAddressRouter(String name)	

*Campos de la clase **MessageDispatcher***

Tipo	Declaración	Acción
protected NodeLog	log	
protected AlertsTable	dispatchTable	

*Constructores de la clase **MessageDispatcher***

Tipo	Declaración	Acción
public	MessageDispatcher()	

*Métodos de la clase **MessageDispatcher***

Tipo	Declaración	Acción
public Object	stageMain(Object aMessage)	
protected void	resolveMessage(Message m)	
private void	resolveRequest(Request r, RequestProcessor rp, String messageID)	
private void	resolveResult(Result r)	
private void	resolveDialog(Dialog d)	

*Campos de la clase **MessageBuilder***

Tipo	Declaración	Acción
private OutgoingQueue	outgoing	
private PipedReader	messageEvent	

*Constructores de la clase **MessageBuilder***

Tipo	Declaración	Acción
public	MessageBuilder(NodeStage nextStage)	

*Métodos de la clase **MessageBuilder***

Tipo	Declaración	Acción
public void	stageInit()	
public Object	stageMain()	
public void	stageStop()	

8.2 Diseño

Campos de la clase ***CommunicationOut***

Tipo	Declaración	Acción
private NodeLog	log	
private Connection	conn	

Constructores de la clase ***CommunicationOut***

Tipo	Declaración	Acción
public	CommunicationOut()	

Métodos de la clase ***CommunicationOut***

Tipo	Declaración	Acción
private void	createConnection()	
private Object[]	resolveAddress(Message m)	
private DataOutputStream	filterMessage(OutputStream output)	
public void	sendMessage(String plainMessage, DataOutputStream dataOut)	
public Object	stageMain(Object message)	

Campos de la clase ***RequestProcessor***

Tipo	Declaración	Acción
private Queue	requests	
private Queue	results	
private Document	domRootDoc	
private SingleRequest	currentRequest	
protected ConfigFile	cf = ConfigFile.getInstance()	
private WorkingSet	inputWS	
private WorkingSet	outputWS	
private Request	current	
private String	messageID	
private ServiceLoader	sl	
private boolean	needBoolResult	

Constructores de la clase ***RequestProcessor***

Tipo	Declaración	Acción
public	RequestProcessor()	

Métodos de la clase ***RequestProcessor***

Tipo	Declaración	Acción
public void	loadRequest(Request r, String MessageID)	
private void	buildWS(SingleRequest sr, String serviceID)	
private void	buildResult(ResultSet out)	
public void	run()	
public void	sendResults()	
public WorkingSet	ResultSetToWorkingSet (ResultSet r, String serviceID)	

Campos de la clase ***Mailbox***

Tipo	Declaración	Acción
protected NodeLog	log	
protected OutgoingQueue	outgoing	

Constructores de la clase ***Mailbox***

Tipo	Declaración	Acción
public	Mailbox()	

Métodos de la clase ***Mailbox***

Tipo	Declaración	Acción
public int	send(Message m)	

Campos de la clase ***ReceiverMailbox***

Tipo	Declaración	Acción
public PipedWriter	frameAlertOut	
public PipedReader	frameAlert	
public AlertsTable	alertTable	
private SyncQueue	boxQueue	

Constructores de la clase ***ReceiverMailbox***

Tipo	Declaración	Acción
public	ReceiverMailbox()	

Métodos de la clase ***ReceiverMailbox***

Tipo	Declaración	Acción
private int	setAlerts(Message m)	
public int	send(Message m)	

public Frame	receive()	

Campos de la clase ***AlertObject***

Tipo	Declaración	Acción
private NodeLog	log	
private int	data	
private PipedWriter	p	
private SyncQueue	boxQueue	
public int	alerts	
public String	expectedType	

Constructores de la clase ***AlertObject***

Tipo	Declaración	Acción
public	AlertObject(String expectedType, PipedWriter p, SyncQueue boxQueue, int alerts)	

Métodos de la clase ***AlertObject***

Tipo	Declaración	Acción
public void	add(int i)	
public void	add()	
public void	del(int i)	
public void	del()	
public void	deliver(Frame f)	
public void	clear()	

Campos de la clase ***AlertsTable***

Tipo	Declaración	Acción
private NodeLog	log	
Hashtable	table	
private static AlertsTable	instance	

Constructores de la clase ***AlertsTable***

Tipo	Declaración	Acción
protected	AlertsTable()	

Métodos de la clase ***AlertsTable***

Tipo	Declaración	Acción
public synchronized void	setAlert(String alertID,String expectedType,	

	PipedWriter p, SyncQueue boxQueue, int alerts)	
public synchronized AlertObject	getAlertObject (String alertID,String expectedType)	
public static AlertsTable	getInstance()	
public static void	delInstance()	

Campos de la clase ***DialogProcessor***

Tipo	Declaración	Acción
Dialog	incomingDialog	
Dialog	outgoingDialog	

Métodos de la clase ***DialogProcessor***

Tipo	Declaración	Acción
public void	loadDialog(Dialog d)	
public void	run()	
private void	sendNewDialog()	

Campos de la clase ***FunctionLoader***

Tipo	Declaración	Acción
private String	functionPackage	
private String	servicePackage	
private NodeLog	log	
private static FunctionLoader	instance	

Constructores de la clase ***FunctionLoader***

Tipo	Declaración	Acción
protected	FunctionLoader (String functionPackage, String servicePackage)	

Métodos de la clase ***FunctionLoader***

Tipo	Declaración	Acción
public Service	getAServiceInstance(String serviceID)	
public Function	getAFunctionInstance(String functionID)	
public Function	getUnknownTypeInstance(String functionID)	
private void	updateFunctionSkills(String function)	
private void	updateServiceSkills(String service)	
public static	getInstance	

FunctionLoader	(String functionPackage, String servicePackage)	

Campos de la clase ***ServiceLoader***

Tipo	Declaración	Acción
private String	servicePackage	
private NodeLog	log	
private static ServiceLoader	instance	

Constructores de la clase ***ServiceLoader***

Tipo	Declaración	Acción
protected	ServiceLoader(String servicePackage)	

Métodos de la clase ***ServiceLoader***

Tipo	Declaración	Acción
public Service	getAServiceInstance (String serviceID, Document docu)	
private void	updateServiceSkills(String service)	
public static ServiceLoader	getInstance(String servicePackage)	

Campos de la clase ***WorkingSet***

Tipo	Declaración	Acción
public InetAddress	fromInetAddress	
public String	messageID	
public String	frameID	
public MIPHeader	header	
public String	service	
public PlainData[]	arguments	
public Model	model	
public int	reqStringIndex	

Constructores de la clase ***WorkingSet***

Tipo	Declaración	Acción
public	WorkingSet (InetAddress fromInetAddress, String messageID, String frameID, MIPHeader header, String service, PlainData[] arguments, Model model, int reqStringIndex)	

*Campos de la clase **function***

Tipo	Declaración	Acción
protected Document	domRootDoc	
protected ResultSet	retu	

*Métodos de la clase **function***

Tipo	Declaración	Acción
public Dialog	runDialog(Dialog d)	
public ResultSet	runRequest()	
public void	loadDocument(Document docu)	

*Campos de la clase **Service***

Tipo	Declaración	Acción
protected boolean	result	

*Constructores de la clase **Service***

Tipo	Declaración	Acción
public	Service()	

*Métodos de la clase **Service***

Tipo	Declaración	Acción
public ResultSet	runRequest(WorkingSet ws)	
public boolean	getResult()	

*Campos de la clase **ResultSet***

Tipo	Declaración	Acción
public PlainData[]	results	
public Model	model	

*Constructores de la clase **ResultSet***

Tipo	Declaración	Acción
public	ResultSet(Model model, PlainData[] results)	
public	ResultSet()	

*Métodos de la clase **ResultSet***

Tipo	Declaración	Acción
public void	setModel(Model model)	
public void	setResults(PlainData[] results)	

*Campos de la clase **OutgoingQueue***

Tipo	Declaración	Acción
private NodeLog	log	
private static OutgoingQueue	instance	
private PipedWriter	dataOut	
private Queue	queue	

*Constructores de la clase **OutgoingQueue***

Tipo	Declaración	Acción
protected	OutgoingQueue()	

*Métodos de la clase **OutgoingQueue***

Tipo	Declaración	Acción
public static OutgoingQueue	getInstance()	
public synchronized void	syncPut(Message m)	
public synchronized Message	syncGet()	
public PipedWriter	getMessageEvent()	
public void	pipedClose()	
public static void	delInstance()	

*Campos de la clase **Broker***

Tipo	Declaración	Acción
private String[]	ServiceConfValues	
private String[]	FunctionConfValues	
private String[]	ownFunctionList	
private String[]	ownServiceList	
private Loader	l	

*Constructores de la clase **Broker***

Tipo	Declaración	Acción
public	Broker()	

*Métodos de la clase **Broker***

Tipo	Declaración	Acción
private boolean	loadServices()	
private boolean	loadFunctions()	

*Campos de la clase **Processor***

Tipo	Declaración	Acción
private String[]	ownConfValues	
private String	ownServiceList	
private String	ownFunctionList	

*Constructores de la clase **Processor***

Tipo	Declaración	Acción
public	Processor()	

*Campos de la clase **Repository***

Tipo	Declaración	Acción
private String[]	ownConfValues	
private String[]	ownServiceList	
private String[]	ownFunctionList	

*Constructores de la clase **Repository***

Tipo	Declaración	Acción
public	Repository()	

*Campos de la clase **Tool***

Tipo	Declaración	Acción
private String[]	ownConfValues	
private String	ownServiceList	
private String	ownFunctionList	

*Constructores de la clase **Tool***

Tipo	Declaración	Acción
public	Tool()	

*Campos de la clase **FileBroker***

Tipo	Declaración	Acción
private String[]	ownConfValues	
public static ConexionBD	interfaceBD	
private String[]	ownServiceList	
private String[]	ownFunctionList	

*Constructores de la clase **FileBroker***

Tipo	Declaración	Acción
public	FileBroker()	

Diagrama de secuencia para la creación de un nodo genérico

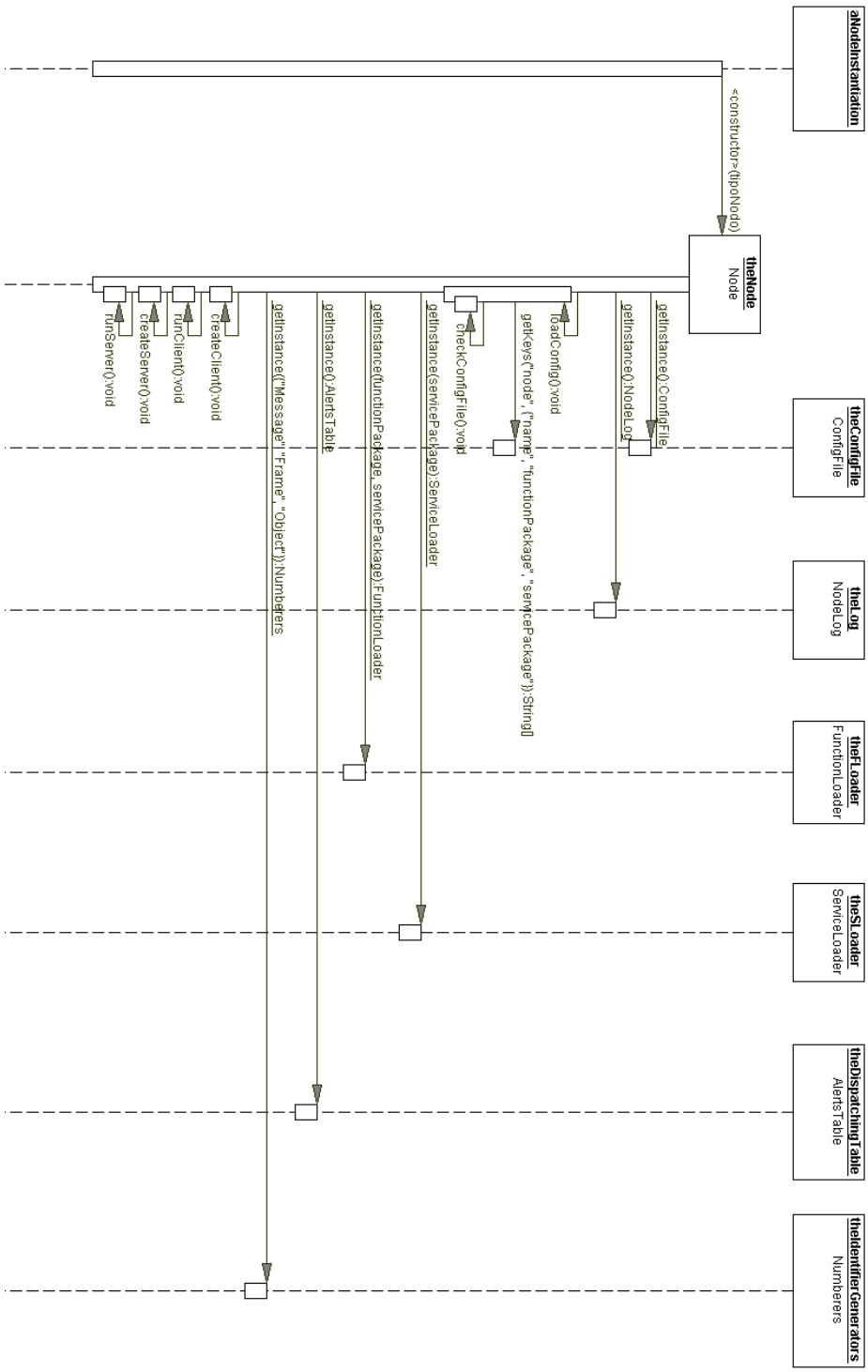


Diagrama de secuencia del proceso de creación de un nodo genérico, dónde se muestra el proceso de inicialización de todos los módulos comunes a todos los tipos de nodos.

Diagrama de secuencia del proceso de creación de un nodo específico

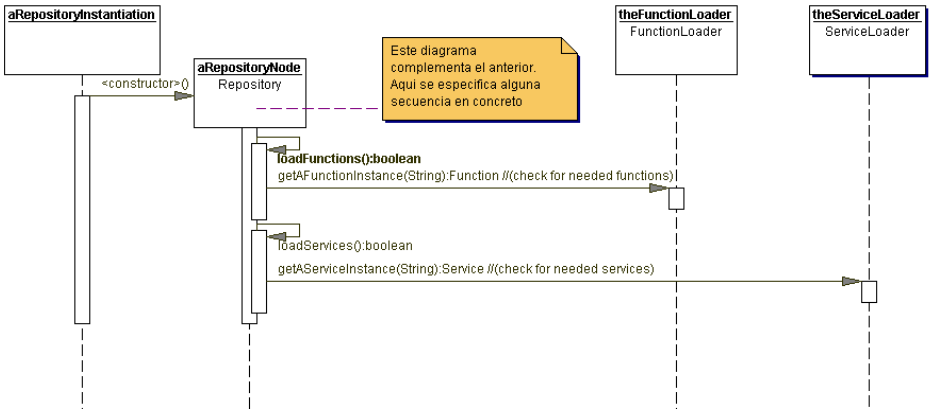


Diagrama de secuencia del proceso de creación de un nodo específico, en este caso un repositorio. El nodo comprueba los servicios y las funciones obligatorios dependientes de su tipo de nodo y aquellas que se indiquen en el fichero de configuración.

Diagrama de secuencia del proceso de creación del servicio Servidor

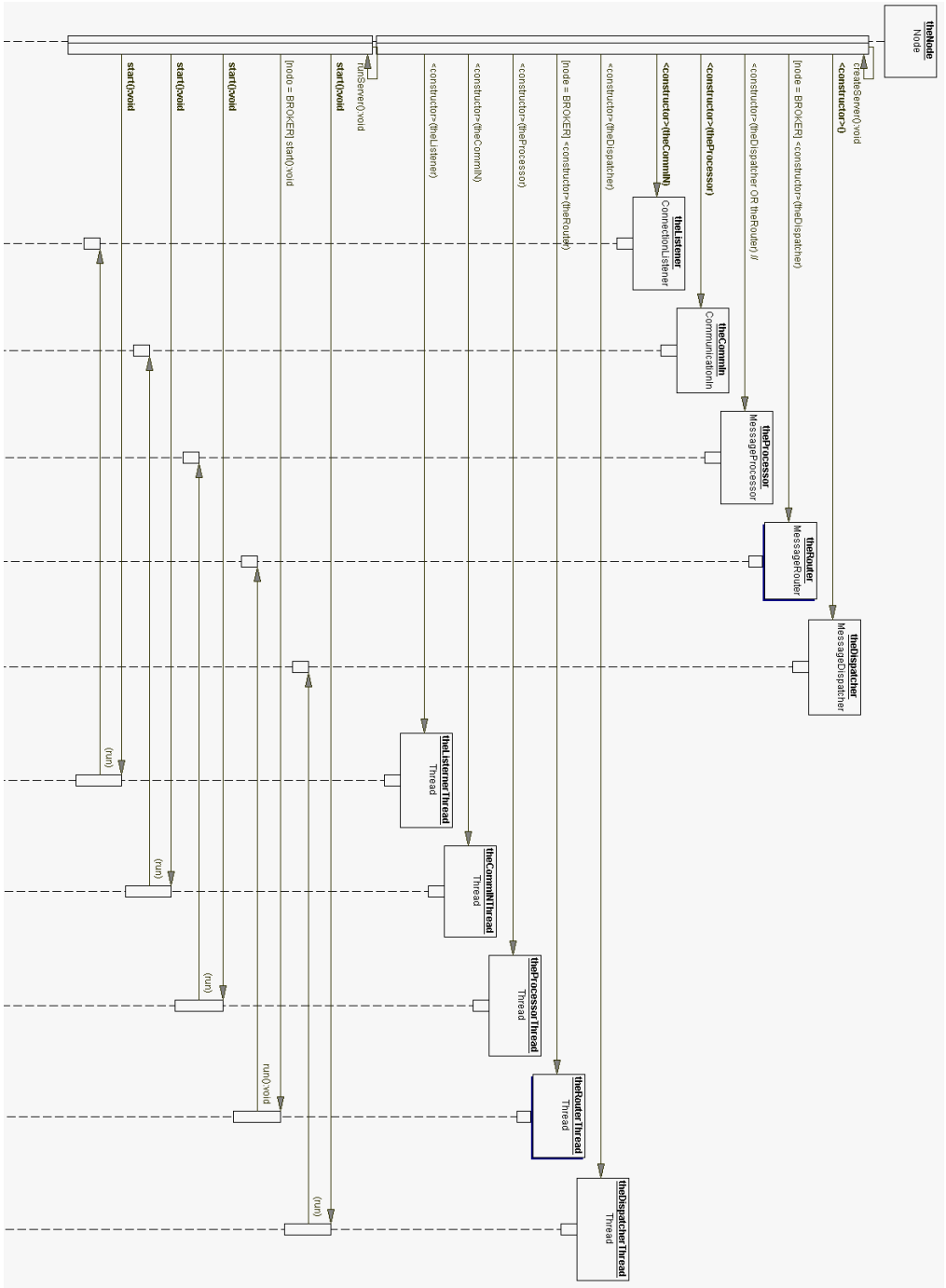
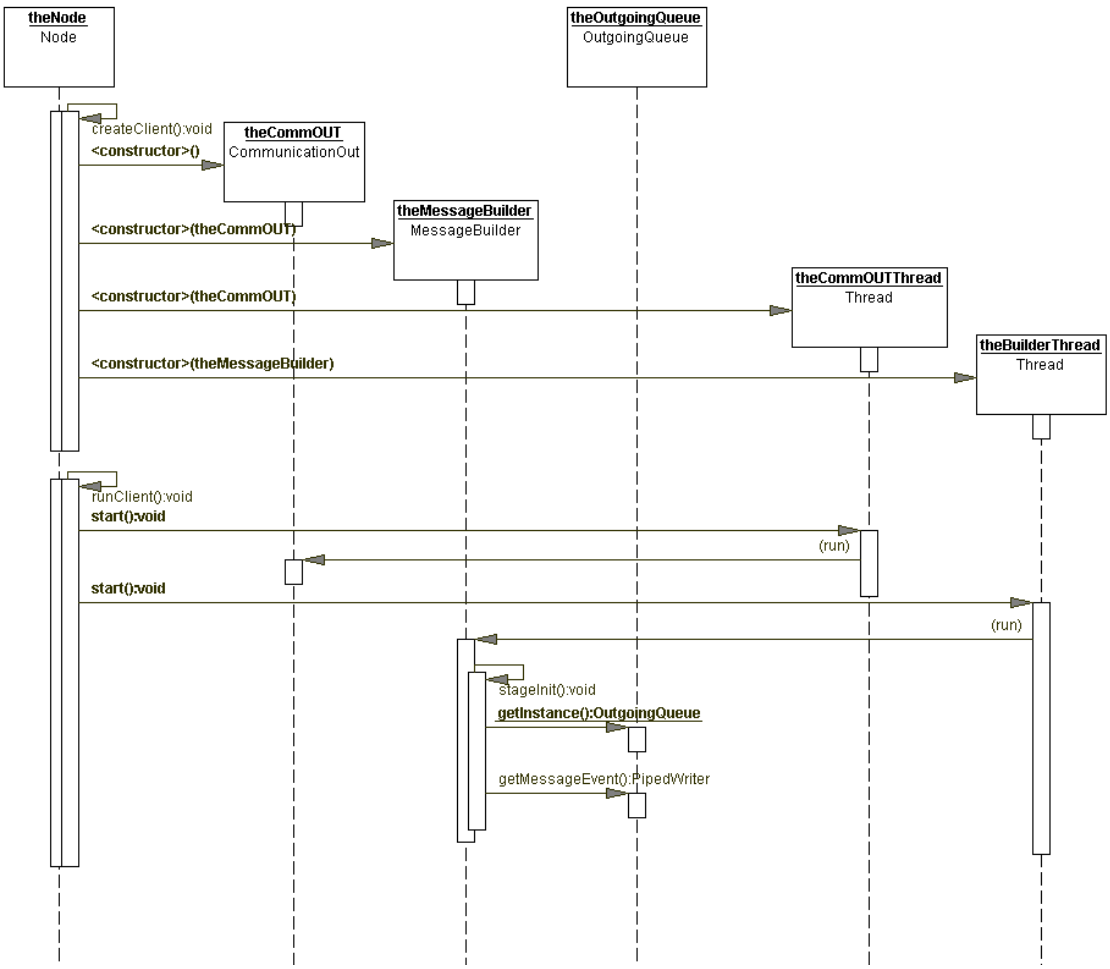


Diagrama de secuencia del proceso de creación, y puesta en marcha, del módulo servidor del nodo. Las etapas son creadas y lanzadas desde la última hasta la primera

Diagrama de secuencia del proceso de creación del servicio Cliente



Al igual que en la etapa Servidor, el Cliente se crea atendiendo una estructura de hilos, de esta manera conseguimos concurrencia. Cada una de las etapas que forman el cliente se forman con la ayuda de la clase NodeStage, la cual nos ayudan a establecer una comunicación y una secuencia.

Diagrama de Secuencia, del proceso general de recibir un mensaje en un nodo broker

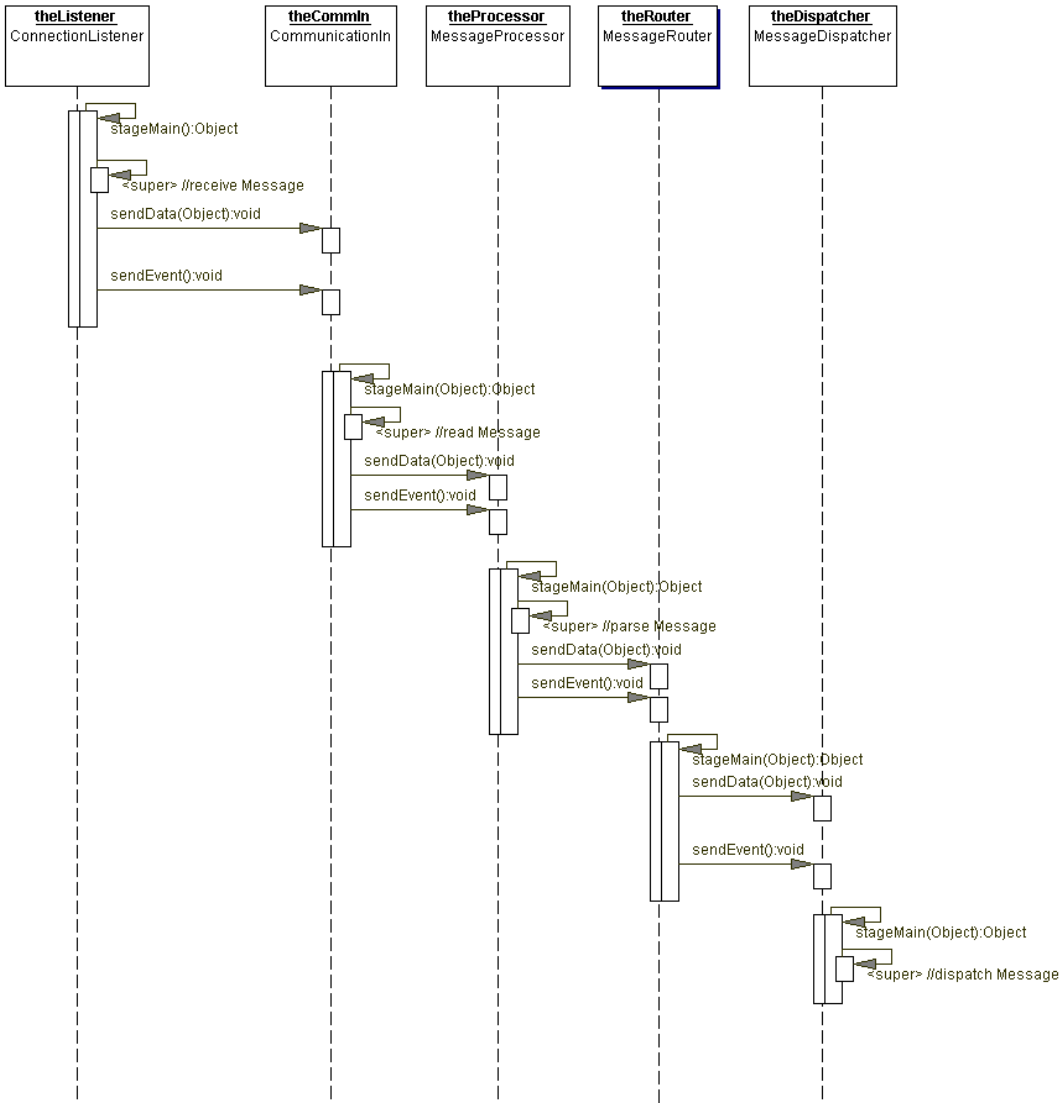


Diagrama de secuencia orientativo, del proceso general de recibir y procesar un mensaje, por parte del módulo del servidor. Esta secuencia ha sido creada para un nodo *Broker*, siendo más completa, pues en otro tipo de nodo tendremos que eliminar el objeto *theRouter*, y pasar directamente al objeto *TheDispatcher*. En los siguientes diagramas veremos de forma mas detallada la funcionalidad de cada uno de estos nodos.

Diagrama de Secuencia del proceso de crear y enviar un mensaje por parte del cliente

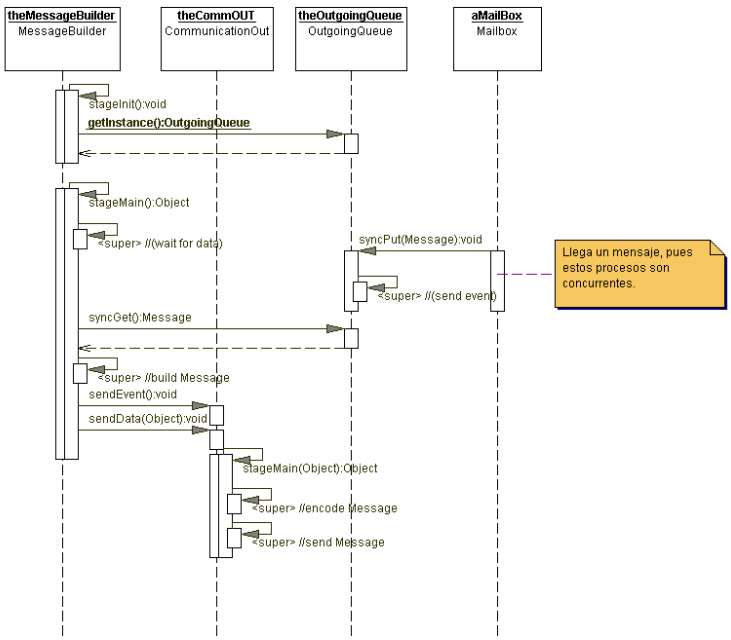


Diagrama de secuencia orientativo, del proceso general de crear y enviar un mensaje por parte del módulo cliente.

Diagrama de secuencia de la etapa ConnectionListener

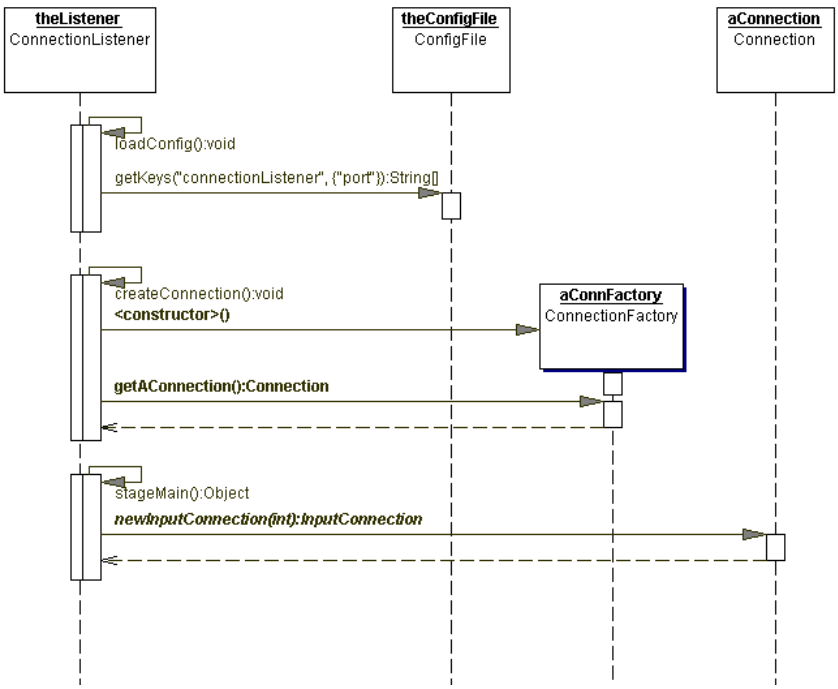


Diagrama de secuencia de la ejecución de la etapa "*ConnectionListener*", perteneciente al módulo servidor. Se obtiene la conexión correspondiente al mensaje entrante. Es la secuencia correspondiente a la creación del objeto. De esta manera podemos detallar aún mas la secuencia vista en otros diagramas.

Diagrama Secuencia de la ejecución CommunicationIn

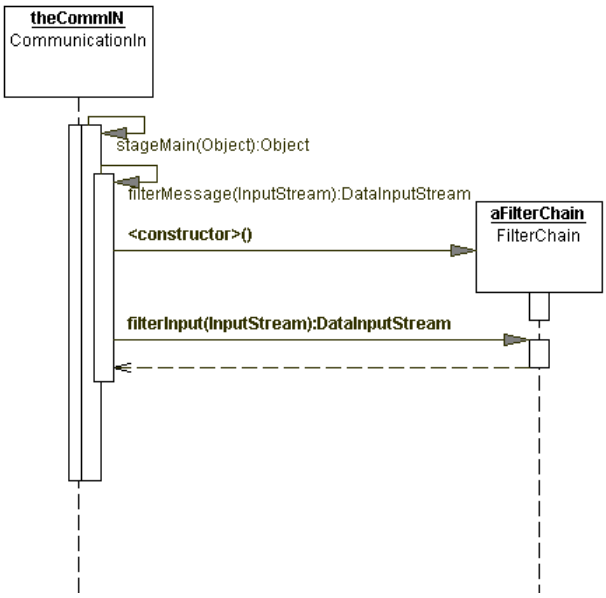


Diagrama de secuencia de la ejecución de la etapa "*CommunicationIn*", perteneciente al módulo Servidor. Se construye una cadena de filtros para decodificar el mensaje. Pasaremos este canal de entrada, con los filtros oportunos, a la siguiente etapa.

Diagrama Secuencia de la ejecución MessageProcessor

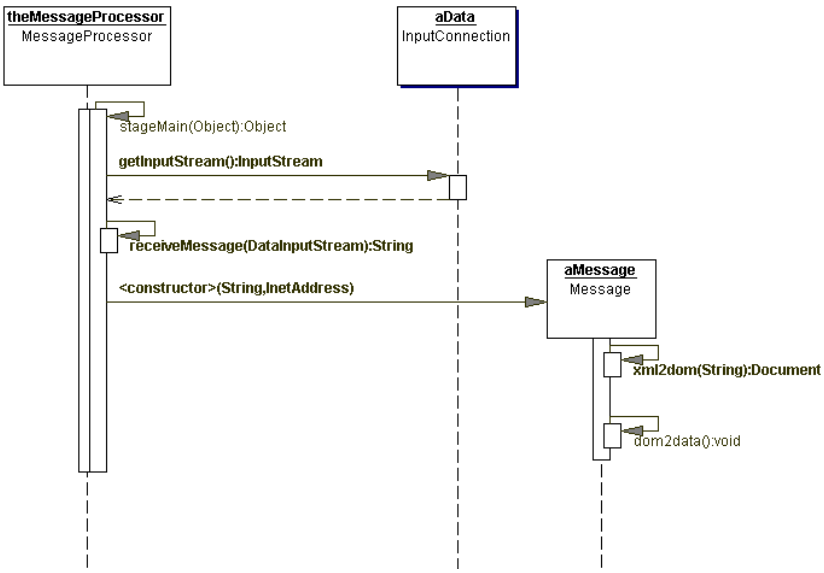


Diagrama de secuencia de la ejecución de la etapa "*MessageProcessor*", perteneciente al módulo Servidor. Se convierte el mensaje, de su representación como cadena de texto (XML), a su representación interna, obteniendo el mensaje del canal de la etapa anterior.

Diagrama Secuencia de la ejecución MessageRouter

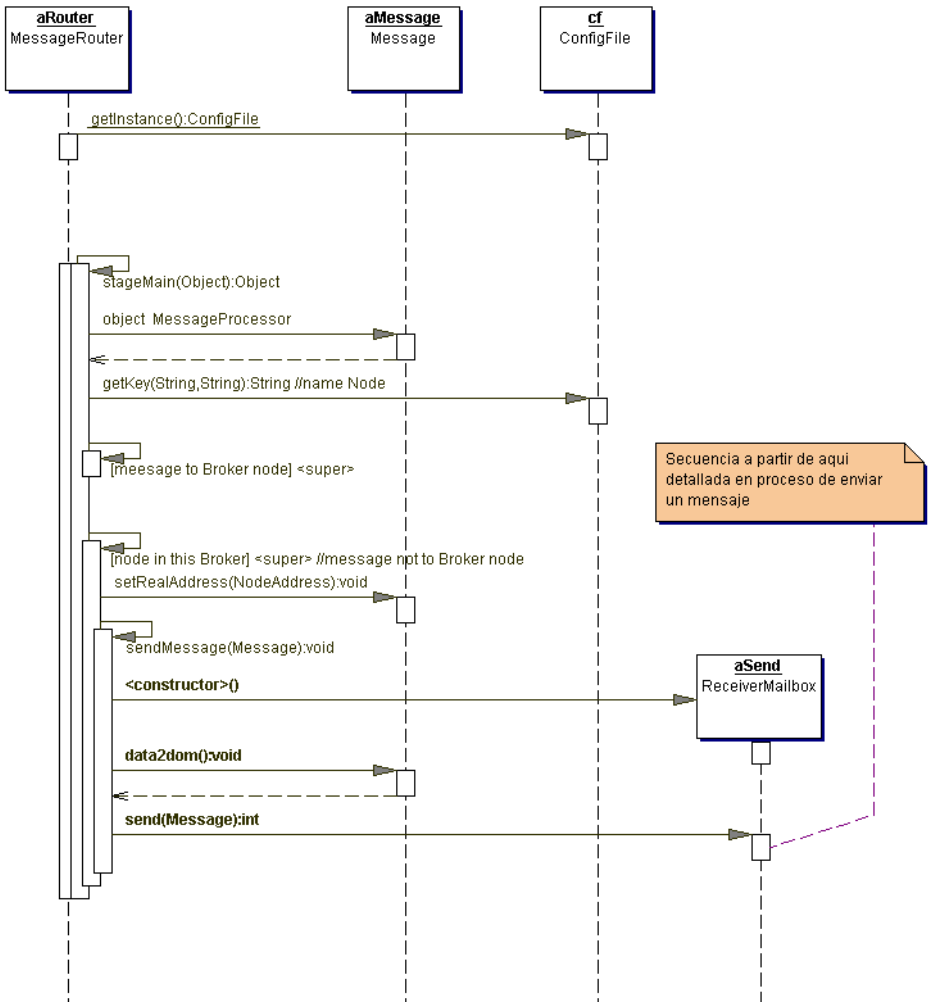


Diagrama de secuencia de la etapa de enrutamiento. Esta etapa es propia únicamente de los nodos de tipo *Broker* y su función consiste en analizar el mensaje resolviendo la dirección de éste. El mensaje puede ir dirigido al Broker, o bien a alguno de los nodos a los que atiende, así sabe como comunicarse con otros *Brokers;* posee un *Broker Default* en última instancia, al que se dirigirá si no encuentra el *Broker* de un determinado mensaje.

Diagrama Secuencia de la ejecución MessageDispatcher

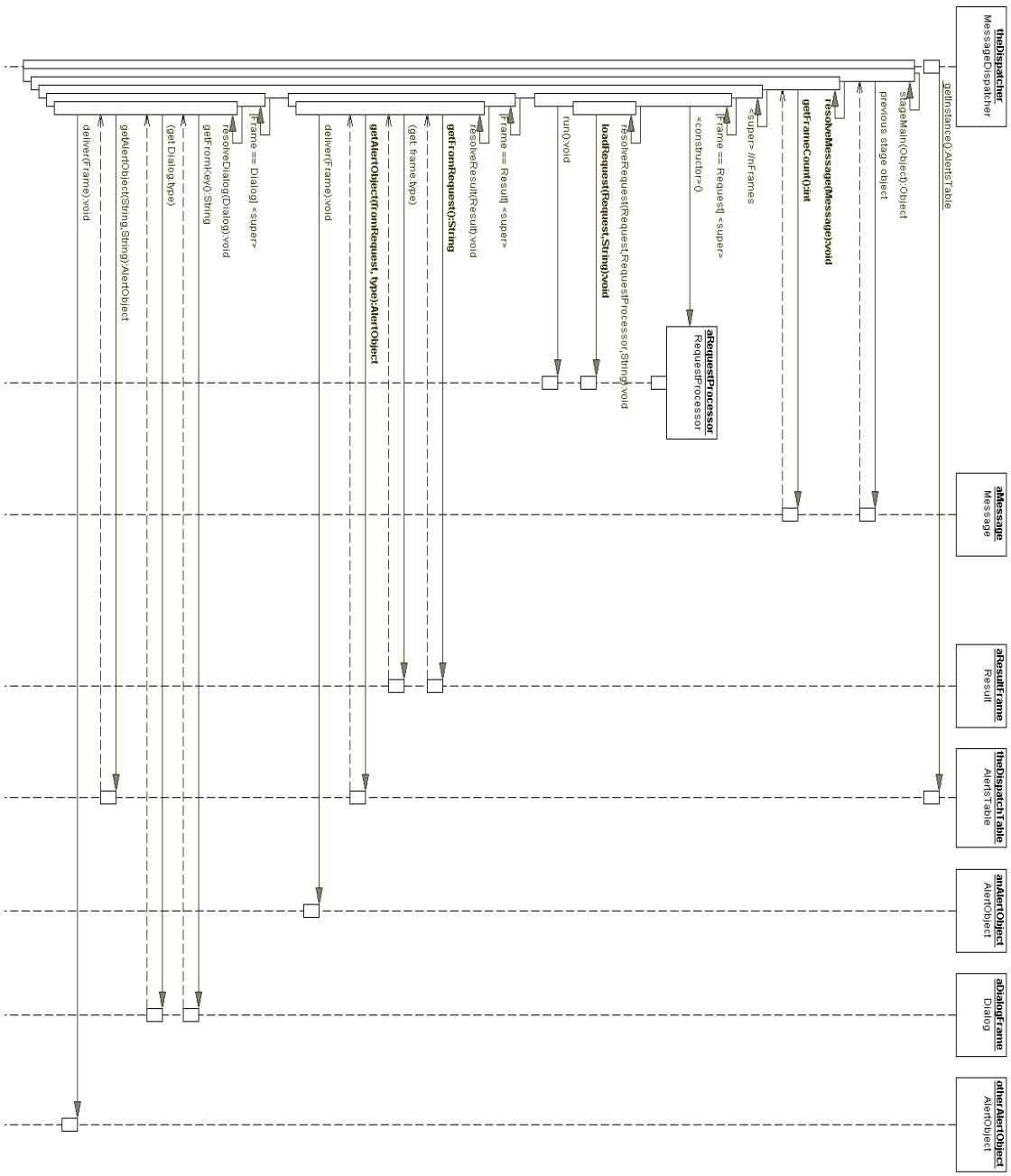


Diagrama de secuencia de la ejecución del módulo encargado de procesar los mensajes de petición de servicios

Diagrama de secuencia del módulo encargado de procesar los mensajes de petición

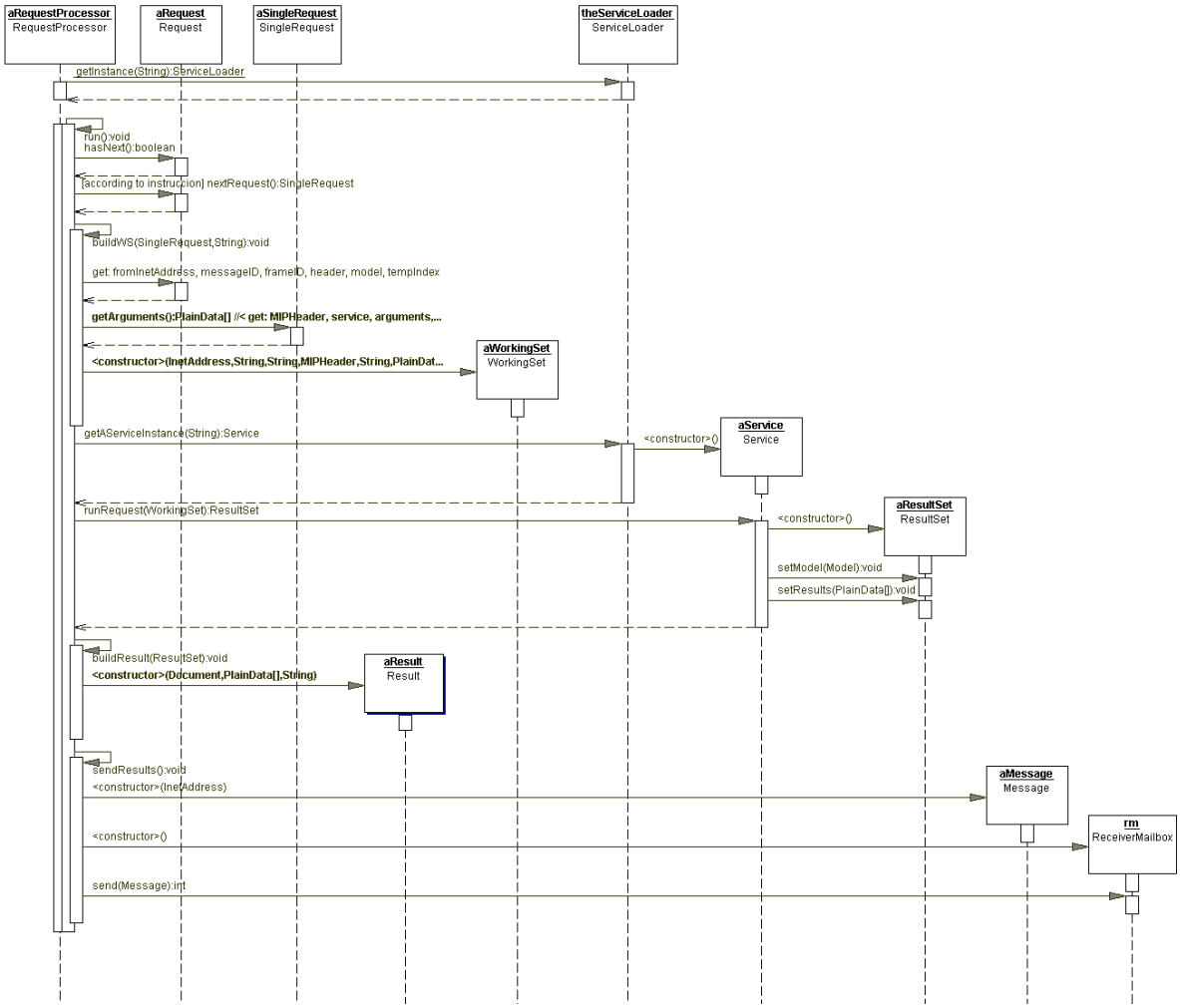


Diagrama de secuencia de la ejecución del módulo encargado de procesar los mensajes de petición de servicios (run()). La ejecución del módulo procesar los mensajes de diálogo es muy similar. Tenemos que tener en cuenta que esta secuencia está simplificada caracterizando las acciones más marcadas, pero éstas dependerán de datos de configuración y sobre todo del lenguaje de formación de los servicios.

Diagrama de secuencia de un mensaje del que se espera respuesta

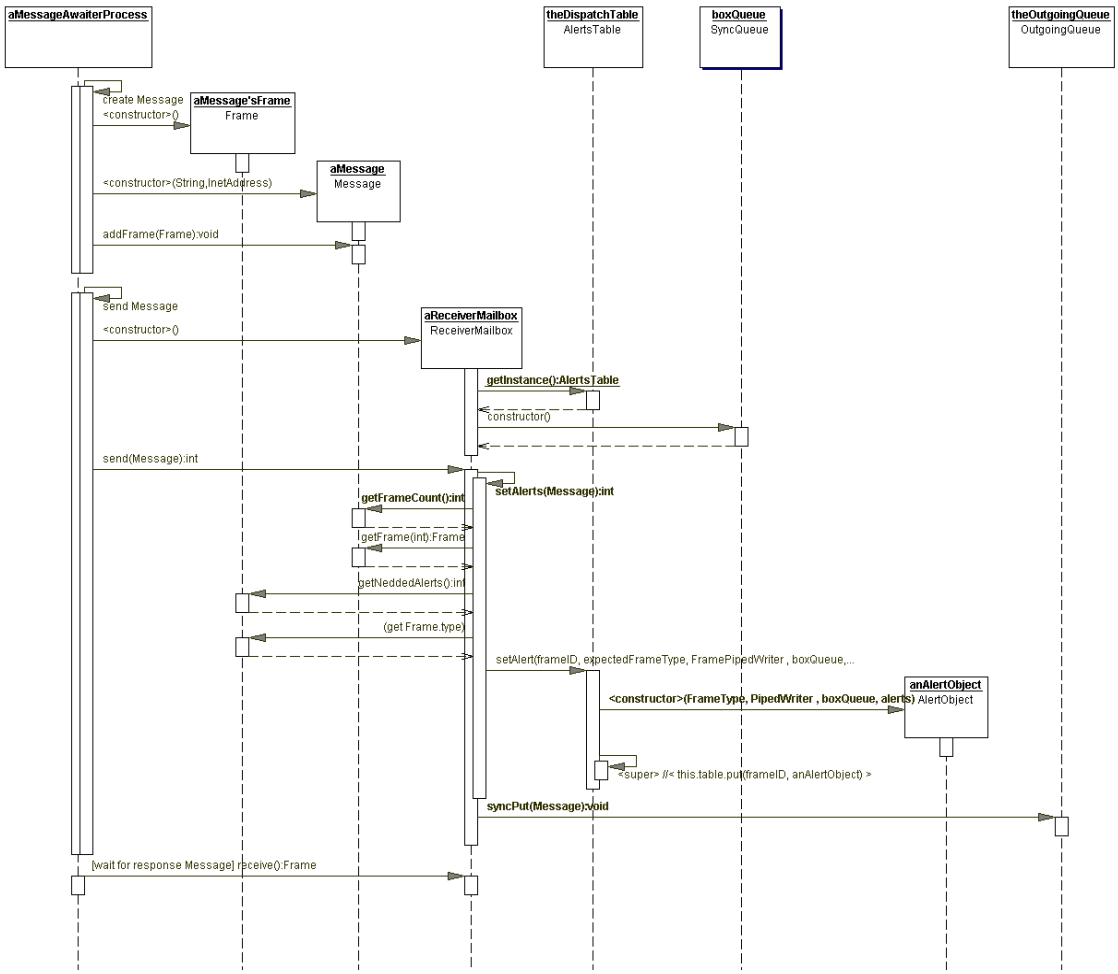


Diagrama de secuencia del proceso de enviar un mensaje del que se espera una respuesta. El mensaje se envía a través de un buzón receptor, que se encarga de anotar en la tabla la distribución, la información relativa al mensaje que se está esperando, y a los canales de comunicación por lo que el buzón puede recibir el mensaje. A continuación el buzón envía el mensaje dejándolo en la cola de salida del sistema.

Diagrama de Secuencia de la etapa MessageBuilder

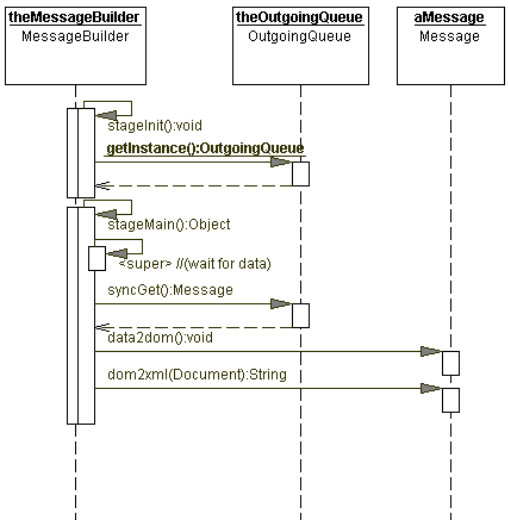


Diagrama de secuencia de la ejecución de la etapa "MessageBuilder", perteneciente al módulo cliente. El mensaje se prepara para ser enviado, obteniéndose su representación DOM y posteriormente su representación XML.

Diagrama de Secuencia de la etapa CommunicationOUT

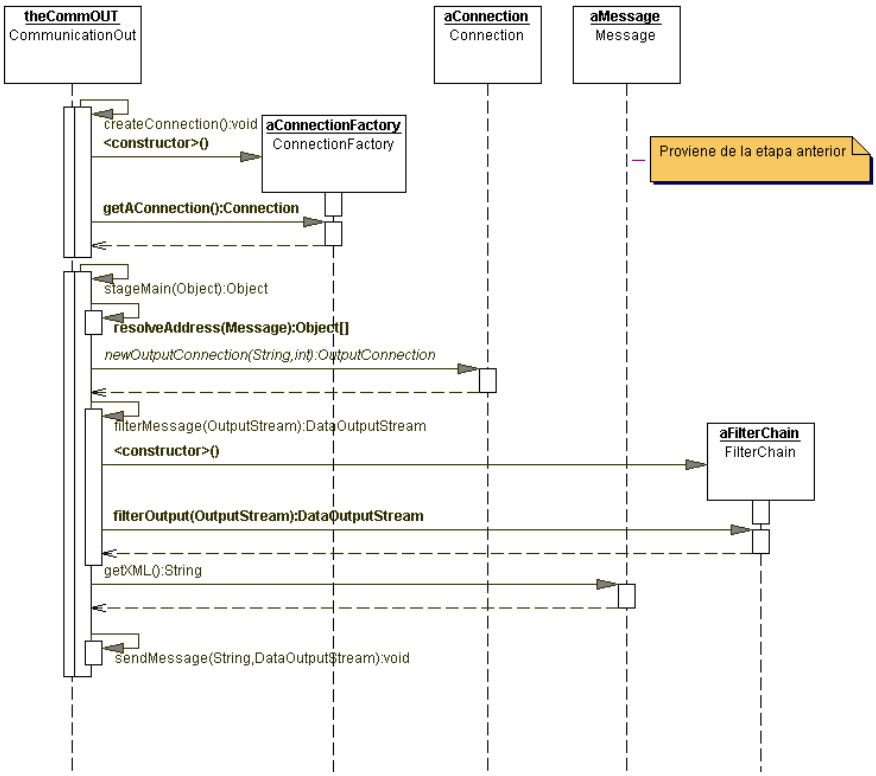


Diagrama de secuencia de la ejecución de la etapa “CommunicationOUT”, perteneciente al módulo cliente. Se crea una conexión para enviar el mensaje, se prepara una cadena de filtros para codificar el mensaje y se envía el mensaje.

8.3 Registro de pruebas unitarias

Prueba	Prueba de los módulos servidor y cliente
Clase ejecutable	sinTogether.FileB.java y sinTogether.Cliente.java
Clases implicadas	nodecore.NodeStage, nodecore.ConnectionListener, nodecore.CommunicationIn, nodecore.MessageProcessor, nodecore.MessageDispatcher, nodecore.Node, nodecore.Tool, nodecore.OutgoingQueue, nodecore.MessageBuilder, nodecore.CommunicationOut y clases de los paquetes: nodedata, filter, connection, mecano.
Procedimiento	Se crean dos nodos, uno actuará como cliente y otro como servidor y en este caso un FileBroker. Desde el cliente se envía un mensaje que hemos formado desde un fichero. El servidor es capaz de procesarlo y enviar una respuesta al cliente dependiendo de lo que obtenga del servicio que el cliente solicita. En todo momento tanto por parte del cliente como del servidor se reflejan en un fichero los pasos que se van realizando.
Resultado deseado	Se espera recoger por pantalla los mensajes que el FileB recibe del cliente y el que el cliente obtiene del FileB.
Resultado obtenido	El esperado
Errores encontrados	Ninguno

La siguiente prueba se ejecuta siguiendo bajo el entorno de Together. Según el apéndice F debemos instalar las clases necesarias para que Together actúe como un nodo del repositorio.

Prueba	Distribución de los mensajes
Clase ejecutable	conTogetherCliente.Cliente2.java y conTogetherBroker.Servidor.java
Clases implicadas	nodecore.NodeStage, nodecore.ConnectionListener, nodecore.CommunicationIn, nodecore.MessageProcessor, nodecore.MessageDispatcher, nodecore.Node, nodecore.Tool, nodecore.OutgoingQueue, nodecore.MessageBuilder, nodecore.CommunicationOut, nodecore.AlertsTable, nodecore.AlertObject, nodecore.Mailbox, nodecore.ReceiverMailbox y clases de los paquetes: nodedata, filter, connection, mecano.
Procedimiento	Se realiza un procedimiento similar al caso anterior pero ahora el cliente será un nodo Together. Se envían una serie de mensajes desde el cliente hasta el servidor, y se utilizan para ello buzones receptores, que permanecen bloqueados a la espera de una respuesta. En el servidor al llegar a la etapa de distribución de mensajes, se crean automáticamente las respuestas que son devueltas al cliente. El cliente, escribe las confirmaciones de recepción de los mensajes al recibir las respuestas que estaba esperando.
Resultado deseado	Se espera obtener en el archivo de registro, las notificaciones de todos los mensajes enviados.
Resultado obtenido	No hay errores.
Errores encontrados	Había un error en el modo en el que la clase AlertObject, contabilizaba las respuestas que se esperaban de un mensaje.

Parte III

Definición de un entorno de desarrollo distribuido basado en MECANOS

Capítulo 9

Iteración 6: Comunicación con el repositorio, Nodo FileBroker y Nodo Broker

En este tema se detallan elementos propios del nodo FileBroker y Broker. El nodo FileBroker es aquel que nos permite procesar el mensaje y tratarlo, pudiendo enviar cada una de sus partes al repositorio. El nodo Broker nos sirve de unión entre el nodo Tool, definido en la herramienta CASE Together y el nodo FileBroker.

9.1 Requisitos

Número	reqCtoR01
Descripción	Recibir un mensaje X-MIP propio de la estructura diseña e implementada en la parte anterior y procesarlo.
Prioridad	Alta

Número	reqCtoR02
Descripción	Almacenar el elemento recibido en el mensaje X-MIP tanto de forma lógica (en la BBDD) como de forma física, (repositorio de contención localizado en el nodo FileBroker). Y bajo la misma estructura que la utilizada en el nodo emisor del mensaje.
Prioridad	Alta

Número	reqCtoR03
Descripción	Informar al usuario, mediante un mensaje X-MIP, de cualquier evolución en el procesamiento del mensaje recibido.
Prioridad	Alta

Número	reqCtoR04
Descripción	Utilizar el diseño y la implementación del frame, llevada acabo en la parte II de este documento, para conseguir implementar el servicio de inserción en la *BD*.
Prioridad	Alta

Número	reqCtoR05
Descripción	Existe en el proyecto [NCAP02] la estructura servidor que se debe intentar respetar para obtener el servicio del mecanoFileBroker.
Prioridad	Media

9.2 Diseño

Descripción

En esta parte se detalla cómo nos comunicaremos con la base de datos y cómo desde la estructura del framework procesaremos el mensaje para poder terminar ejecutando los servicios que nos facilitan y permiten la inserción en el repositorio.

No olvidamos que esta parte fue definida en otro proyecto [NCAP002], en concreto es la parte *servidor*, y como ya hemos dicho se ha intentando respetar al máximo la estructura de éste, aunque podamos ver elementos que en principio hemos dicho que no están definidos en el framework, se han dejado en esta parte pues aunque su gran funcionalidad ha cambiado, su esencia sigue siendo la misma, consiguiendo de esta manera un nexo entre los proyectos. Señalando que esto ya no sería propiamente el framework, sino una instanciación del mismo.

A parte del diagrama de clases, con el que identificamos las funcionalidades de FileBroker, se han definido unos diagramas de secuencia, en los cuales se detallan la funcionalidad de los servicios del FileBroker y del Broker, que se han implementado.

En el diagrama de clases podemos ver que para implementar el servicio hay que heredar de la clase Service e implementar una serie de métodos. Pero esto lo podemos ver mejor en el diccionario de clases, el manual de instanciación, definido en el Capítulo 14, del framework también nos será de gran ayuda.

Ninguno de los capítulos que hacen referencia a esta parte había sido desarrollado con anterioridad, por eso no hay un estudio de evolución. Todas las clases y métodos son nuevas.

Clases relacionadas

public abstract class **ReusableElement**	Implementa el concepto de Elemento Reutilizable según el modelo de Mecano conteniendo todos los atributos y métodos comunes para los distintos Assets y para Mecano, así como diversas funciones que utilizarán aquellos.
public class **AssetElement** extends **ReusableElement**	Esta clase se encarga de crear un asset objeto, modelado a partir de la *DTD* de Mecano y de proporcionar métodos adecuados para realizar consultas relacionadas con assets a la base de datos y por supuesto de insertar un asset en ella.
public class **SendAssetOptionHandlerForTextEditor** extends Service	Clase que implementa un servicio de la estructura framework. Éste servicio se encarga de establecer comunicación con la base de datos para almacenar la información correspondiente al asset, así como almacenar de manera física el asset en el repositorio.

public class **RepresentationElement**	Esta clase se encarga de crear un objeto representación, modelado a partir de la *DTD* de *Mecano* y de proporcionar métodos adecuados para realizar consultas relacionadas con representaciones en la base de datos y por supuesto de insertar una representación en ella. Representación informa de los parámetros físicos del asset. Esta clase mantiene una dependencia con la clase asset.
public class **UserValidation**	Clase que conecta con la base de datos y obtiene si el usuario y password están presentes y por tanto tiene acceso a ella.

Diagramas relacionados

	Modelo estático: diagrama de Clases.
	Diagrama de secuencia de la creación de servicio según el framework.
	Diagrama de Secuencia para el servicio SendNodeInfo
	Diagrama de Secuencia para el servicio SendAssetOptionHandlerForTextEditor.

Modelo estático: diagrama de Clases Cliente Together 6.2

Esta es la parte que se va a instalar en el nodo FileBroker

Campos de la clase ***ReusableElement***

Tipo	Declaración	Descripción
public String	authors = null	Nombres de los desarrolladores del elemento reutilizable.
public String	words = null	Palabras clave que designan al elemento del *Mecano*.
protected	String id = null	Variable que almacena el identificador para la BD del elemento reutilizable, que coincidirá con el de asset o mecano.
public String	nameReusableElement = null	Variable que almacena el nombre del elemento que será el del asset o mecano.
public String	creationDate = null	Fecha de creación del elemento reutilizable.
public String	insertDate = null	Fecha de introducción del elemento en el repositorio.
public String	modificationDate = null	Fecha de la última modificación del elemento reutilizable.
public String	keyWords = null	Frases que describen aspectos importantes del elemento reutilizable, para facilitar la búsqueda del mismo.
public String	plataform = null	Plataforma sobre la que se desarrolló el elemento reutilizable.
public String	restrictions = null	Información legal sobre el uso del elemento reutilizable, incluyendo copyright, patentes, derechos de explotación, limitaciones de explotación y licencias.
public String	idSecurityLevel = null	Almacena el nivel de seguridad más alto al elemento reutilizable o cualquier parte constituyente del elemento.
public String	price = null	Cuantía que debe pagarse para la reutilización del elemento reutilizable.
public String	idLanguage = null	Lengua en el que se encuentra el elemento reutilizable.
public String	version = null	Designación de la versión de un elemento reutilizable.

Constructores de la clase ***ReusableElement***

Tipo	Declaración	Descripción
public	ReusableElement()	Método de creación. Crea una instancia de la clase.

Métodos de la clase ***ReusableElement***

Tipo	Declaración	Descripción
public void	insertBD	Método que se encarga de insertar un

	(Connection conn)	elemento reutilizable en la *BD*. Recibe como parámetro el objeto conexión a la *BD*. Hay que controlar las excepciones: *MecanoServletException, SQLException, ClassNotFoundException, IOException*.
public void insertReusable	insertReusableElement AsoUser (Connection conn, String idReusableElement, String idUser)	Método que inserta en la BD la relación entre el elemento reutilizable y el usuario que lo envió. Recibe como parámetro el objeto conexión a la BD y los identificadores de elemento reutilizable y el usuario. Se deben controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException.*
public static int	searchIdRE (Connection conn, String idRE)	Método que busca el identificador del elemento reutilizable en la BD. (El id coincide con el de asset). Recibe como parámetros el objeto conexión con la BD y el Id del elemento reutilizable. Obtenemos 1 si halla el id, 0 si no halla el id en la BD. Se deben controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException.*
public static int	searchNameRE (Connection conn, String nameRE)	Método que busca un nombre de elemento reutilizable en la BD. Recibe como parámetro el objeto de conexión con la BD y el nombre del elemento reutilizable. Obtenemos 1 si halla el nombre, 0 si no halla el nombre en la BD. Se deben controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	searchReusableElement AsoUser (Connection conn, String idReusableElement, String idUser)	Método que busca la relación entre un elemento reutilizable y un usuario. Recibe como parámetro el objeto conexión a la BD y los identificadores de elemento reutilizable y el usuario. Obtenemos 1 si existe la relación, 0 en caso contrario. Se deben controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	checkIdSecurityLevel (Connection conn, String idSecurityLevel)	Comprueba que el id del nivel de seguridad del elemento reutilizable sea conocido en la BD. Recibe como parámetro el objeto conexión con la BD, el identificador del nivel de seguridad. Obtiene 1 si el nivel de seguridad existe; 0 si no existe. Se deben controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.

public static int	checkIdEvaluationLevel (Connection conn, String idEvaluationLevel)	Comprueba que el id del nivel de evaluación del elemento reutilizable sea conocido en la BD. Recibe como parámetro el objeto conexión con la BD, el identificador del nivel de evaluación. Obtiene 1 si el nivel de evaluación existe, 0 si no existe. Se deben controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	checkIdLanguage (Connection conn, String idLanguage)	Comprueba que el id del idioma del elemento reutilizable sea conocido en la BD. Recibe como parámetros el objeto conexión con la BD, el identificador del idioma. Obtenemos 1 si el idioma existe, 0 si no existe.
public String	getIdReusableElement()	Método que devuelve el parámetro id del elemento reutilizable. Obtenemos la cadena id.

Campos de la clase ***AssetElement***

Tipo	Declaración	Descripción
public String	idAssetType = null	Variable que nos indica el tipo de asset: Casos de uso, diagrama de clases,...
public String	idAbstractionLevel = null	Variable que indica el nivel de abstracción al que pertenece el asset: Análisis, diseño o implementación.
public String	idMethod = null	Variable que indica al método de creación del asset: Orientado a objetos,...
public String	comments = null	Con este almacenamos en la base de datos los comentarios que vayan asociados al elemento *asset*.
public Asset	asset = null	Elemento representado en el framework, que nos servirá para definir el asset. Este asset viene definido en el mensaje y de esta manera accederemos a los datos que definen el elemento.
public RepresentationElement	repre	Variable que apunta al objeto de tipo representación donde se muestran las características físicas del asset.

Constructores de la clase ***AssetElement***

Tipo	Declaración	Descripción
public	AssetElement (Asset ast, Representation re)	Método de creación del asset. Recibe como parámetros un objeto de tipo Node (*DOM*) donde se encuentran los

		datos para rellenar los campos de asset, elemento reutilizable y la representación.

Métodos de la clase ***AssetElement***

Tipo	Declaración	Descripción
public void	insertBD (Connection conn)	Método que se encarga de insertar un *asset* en la *BD*, así como su representación y su elemento reutilizable. Debemos pasarle el objeto conexión con la BD. Se deben de controlar las excepciones, *MecanoServletException, SQLException, ClassNotFoundException, IOException*.
public void	insertAssetAsoRepresentation (Connection conn, String idAsset, String idRepresentation)	Inserta la asociación y dependencia existente entre una representación y un asset en la correspondiente tabla de la BD. Recibe como parámetros, el objeto conexión con la BD y los identificadores de asset y de representación. Se deben de controlar las excepciones, *MecanoServletException, ClassNotFoundException, SQLException*.
public int	getId(Connection conn)	Método que obtiene un Id para un asset según la secuencia generada por la BD. Recibe como parámetro el objeto de conexión con la BD. Obtenemos 1 si consigue un nuevo id, 0 si no halla un nuevo id en la BD. Se deben de controlar las excepciones, *SQLException, ClassNotFoundException, MecanoServletException*.
public static int	searchId (Connection conn, String idAss)	Método que busca un Id de un asset en la BD. Recibe como parámetro el objeto conexión con la BD y el id del asset. Obtenemos 1 si halla el id, 0 si no halla el id en la BD. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	searchName (Connection conn, String nameAsset)	Método que busca un nombre de asset en la BD. Recibe como parámetros el objeto conexión con la BD y el nombre del asset. Obtenemos 1 si halla el nombre, 0 si no halla el nombre en la base de datos. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static String	searchAssetType (Connection conn, String idAsset)	Se deben de controlar las excepciones, *ClassNotFoundException,SQLException, MecanoServletException*.

public static int	searchAssetInMecano (Connection conn, String idMecano, String idAsset)	Método que busca el identificador del tipo de asset del asset que se le pasa por parámetro. Recibe como parámetros el objeto conexión con la BD y el id del asset. Obtenemos el *id* del tipo de asset si lo encuentra; sino devuelve *null*. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	searchRepresentationInAsset (Connection conn, String idAsset, String idRepresentation)	Método que obtiene si una determinada representación pertenece a un determinado asset. Recibe como parámetros el objeto conexión con la BD, el id de asset y el id de representación. Obtenemos 1 si halla el asset en el mecano, 0 si no halla el asset en el mecano. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static String	searchIdAbstractionLevelAsset (Connection conn, String idAsset)	Método que obtiene el *id* del nivel de abstracción de un asset en la *BD*. Recibe como parámetros el objeto de conexión con la *BD* y el id del asset. Si halla el id del tipo de asset, lo devuelve, sino devuelve *null*. Se deben de controlar las excepciones, *SQLException, ClassNotFoundException, MecanoServletException*.
public static int	searchLevelAbstractionAsset (Connection conn, String idAsset)	Método que obtiene el nombre del nivel de abstracción de un asset por el id del asset en la BD. Le pasamos como parámetros el objeto de conexión con la *BD* y el id del asset. Obtenemos 1 si halla el nombre, 0 si no halla el nombre.
public static int	checkIdAssetType (Connection conn, String idAssetType)	Comprueba que el *id* del tipo de asset sea conocido en la base de datos. Recibe como parámetros el objeto conexión con la *BD*, el identificador del tipo de asset. Obtenemos 1 si el tipo de relación existe, 0 si no existe. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	checkIdAbstractionLevel (Connection conn, String idAbstractionLevel)	Comprueba que el id del nivel de abstracción del asset sea conocido en la base de datos. Recibe como parámetro el objeto conexión con la *BD*, el identificador del nivel de abstracción. Obtenemos 1 si el tipo de nivel de abstracción existe, 0 si no

		existe. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public int	checkAssetTypeAndAbstractionLevel (Connection conn, String idAssetType, String idAbstractionLevel)	Comprueba que el nivel de abstracción del *asset* a insertar es coherente con su tipo de *asset*. Recibe como parámetro el objeto conexión con la BD el identificador de asset y el identificador de su nivel de abstracción. Se deben de controlar las excepciones, *MecanoServletException, ClassNotFoundException, SQLException*.

Campos de la clase ***SendAssetOptionHandlerForTextEditor***

Tipo	Declaración	Descripción
private String	code = null	Esta variable nos sirve para indicar la descripción del código del resultado de la operación, desde un error, hasta resultado correcto.
private Asset	asset = null	Elemento *asset* de la estructura del framework, representa el objeto reutilizable.
private Representation	representation = null	Objeto representación

Constructores de la clase ***SendAssetOptionHandlerForTextEditor***

Tipo	Declaración	Descripción
public	SendAssetOptionHandlerForTextEditor()	Crea el objeto servicio del nodo e inicializa la variable resultado heredada de la clase *Service*.

Métodos de la clase ***SendAssetOptionHandlerForTextEditor***

Tipo	Declaración	Descripción
public ResultSet	runRequest (WorkingSet ws)	En esta función se almacena la funcionalidad del servicio, es un método que hay que implementar en todas sus intancias. Almacena en la URL indicada por el servicio, el asset textual que recibe.

Campos de la clase ***RepresentationElement***

Tipo	Declaración	Descripción
public String	nameRepresentation = null	Identificador del objeto en la base de datos.
public	String URL = null	Nombre de la representación.
public String	idFormat = null	Dirección donde se guarda el asset físico

		(Código fuente o diagrama).
public String	sizeFile = null	Identificador del formato en el que se encuentra el fichero físico: ASCII, Postscript,...
public String	idMedia = null	Tamaño en bytes del fichero físico.
public String	comments = null	Identificador del medio en el que se puede obtener el asset: CD-ROM,papel,...
public Representation	representation	Posibles notas sobre el formato físico del asset u otros.

Constructores de la clase ***RepresentationElement***

Tipo	Declaración	Descripción
public	RepresentationElement (Representation repre)	Método constructor. Obtiene la información para rellenar los atributos de la clase, del elemento de tipo Node obtenido a través del parser de *DOM* del fichero *XML* enviado por el cliente. Recibe como parámetros la URL donde está almacenado el fichero y un objeto de tipo *Node* de donde se obtiene la información para los atributos de representación. Obtenemos un objeto de tipo representación

Métodos de la clase ***RepresentationElement***

Tipo	Declaración	Descripción
public void	insertBD (Connection conn)	Método que añade el objeto representación a la *BD* basándose en los atributos de la misma y obtenidos del fichero *XML* enviado por el cliente. Recibe como parámetros el objeto conexión a la *BD*. Se deben de controlar las excepciones, *MecanoServletException, SQLException, ClassNotFoundException*.
public int	getId (Connection conn)	Método que obtiene un *Id* para un objeto representación según la secuencia generada por la BD. Recibe como parámetro el objeto conexión a la BD. Obtenemos 1 si consigue un nuevo *id*, 0 si no halla un nuevo id en la BD. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	searchId (Connection conn, String idRep)	Método que busca un *Id* de un objeto representación en la BD. Recibe como parámetros el objeto conexión con la BD y el *id* del objeto representación. Obtenemos

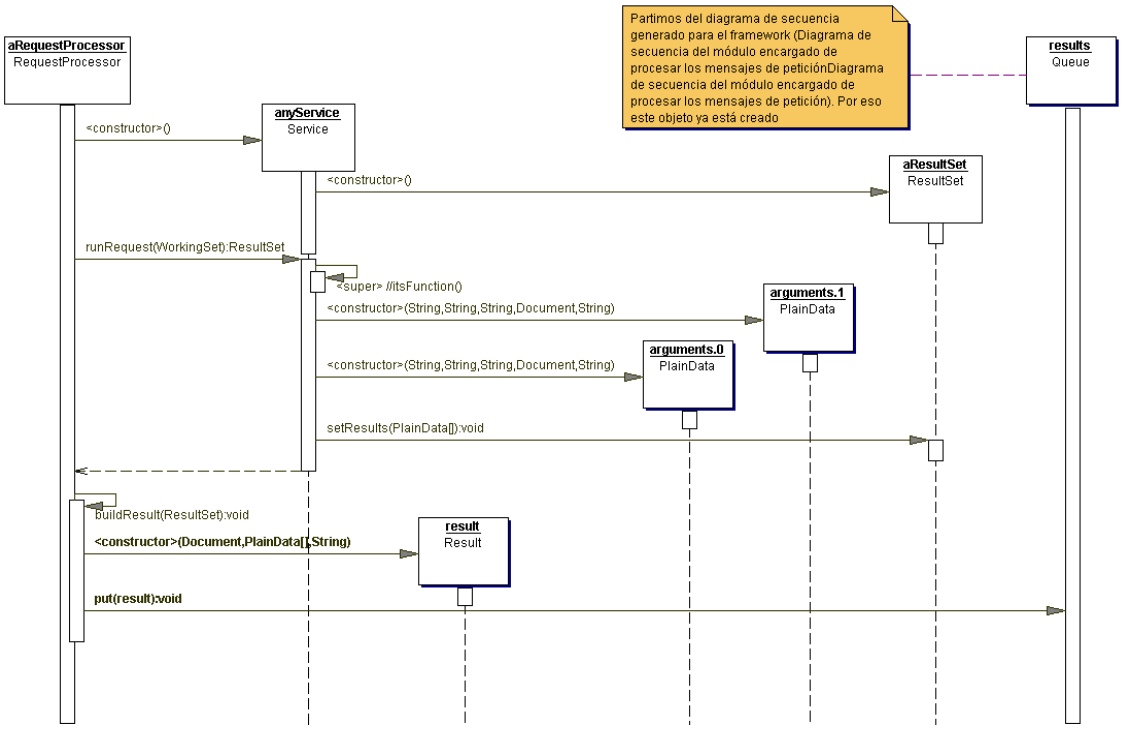
		1 si halla el *id*, 0 si no halla el *id* en la base de datos. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	searchURL (Connection conn, String _URL)	Método que busca una URL en la tabla representación en la *BD*. Recibe como parámetros el objeto de conexión con la BD y la URL. Obtenemos 1 si halla la URL, 0 si no halla la URL en la base de datos. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	searchName (Connection conn, String nameRepresentation)	Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	checkIdFormat (Connection conn, String idFormat)	Comprueba que el id del formato de la representación sea conocido en la base de datos. Recibe como parámetros el objeto conexión con la BD, el identificador del formato. Obtenemos 1 si el formato existe, 0 si no existe. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException*.
public static int	checkIdMedia (Connection conn, String idMedia)	Comprueba que el *id* del medio de la representación sea conocido en la base de datos. Recibe como parámetros el objeto conexión con la BD, el identificador del medio. Obtenemos 1 si el medio existe, 0 si no existe. Se deben de controlar las excepciones, *ClassNotFoundException, SQLException, MecanoServletException.*
public String	getIdRepresentation()	Método que devuelve el parámetro idRepresentation. Obtenemos la cadena *id*.

Métodos de la clase ***UserValidation***

Tipo	Declaración	Descripción
public static int	UserValidate (String login, String password)	Método que comprueba que el usuario y su password estén ambos presentes en la base de datos y por tanto tenga acceso a ella. Lanza excepciones de *SQLException* para errores de la sentencia SQL y *ClassNotFoundEception* para errores con el driver de la *BD*. Le pasamos como parámetros el nombre de usuario y su

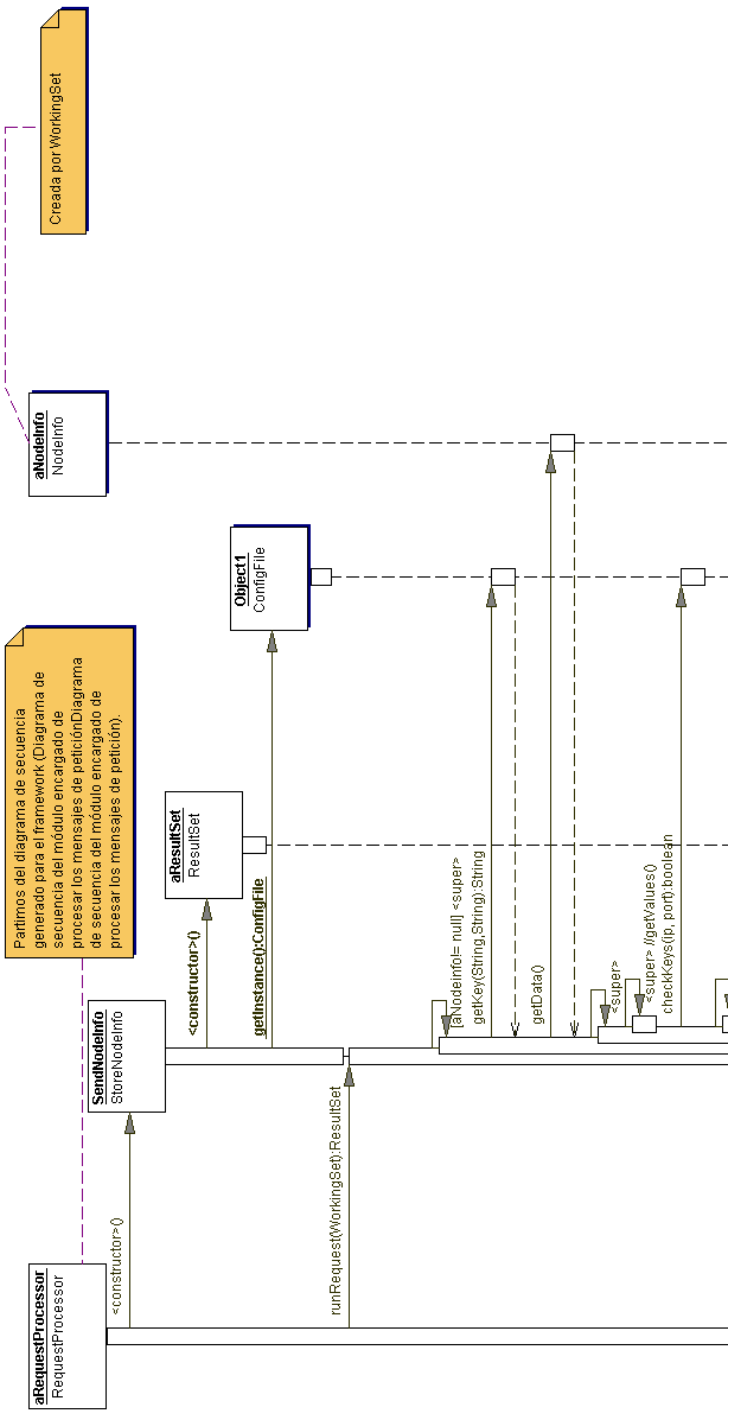
		clave. Obtenemos 1 si el usuario tiene acceso, 0 en caso contrario.
public static String	getIdUser (Connection conn, String login)	Método que averigua el identificador de un usuario. Lanza excepciones de *SQLException* para errores de la sentencia *SQL* y *ClassNotFoundEception* para errores con el driver de la *BD*. Recibe como parámetro el nombre del usuario. Obtenemos el identificador del usuario si lo encuentra; *null* en caso contrario.

Diagrama de secuencia de la creación del servicio según el framework



Ya hemos visto en la parte de definición del framework, como se procesan los mensajes, pero ahora vemos como funciona el servicio, las clases que debe utilizar para comunicarse con el framework, y las pautas que debe tomar para que su funcionamiento sea correcto.

Diagrama de Secuencia para el servicio SendNodeInfo



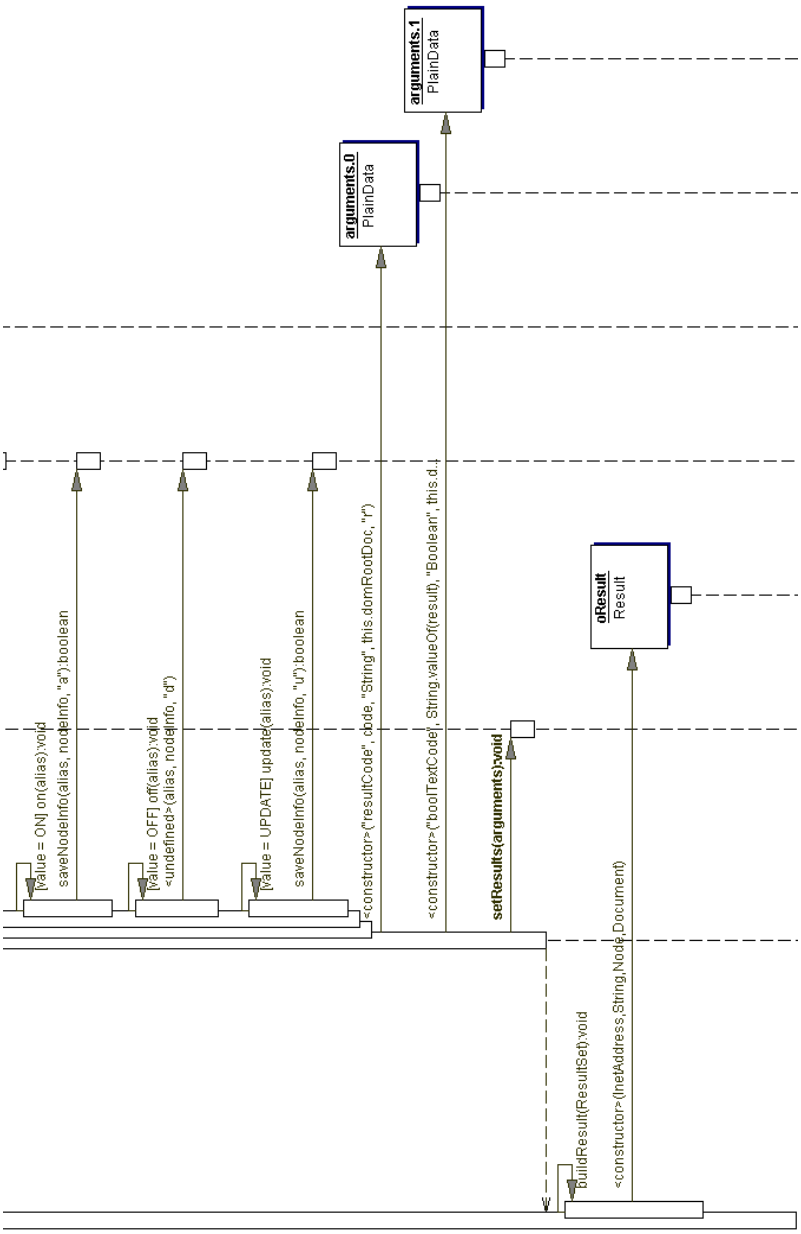
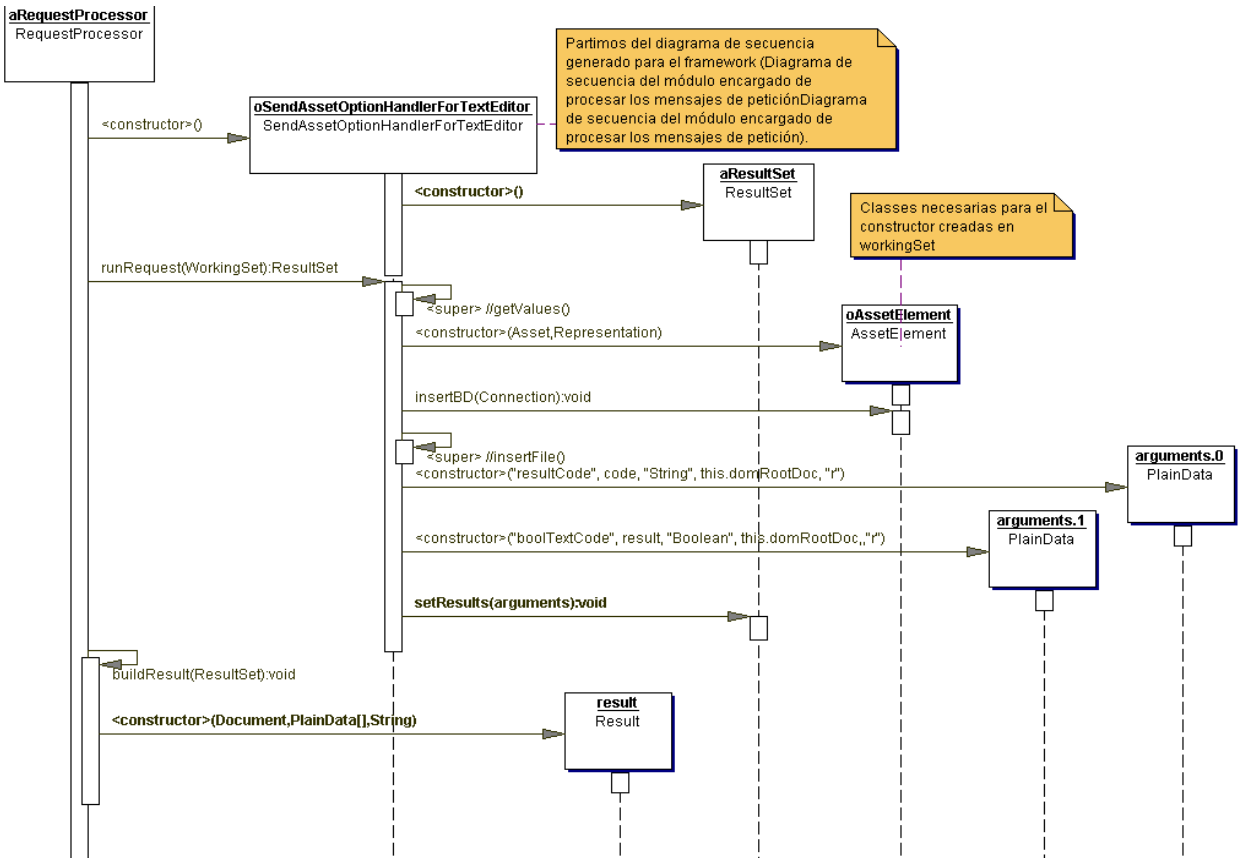

arguments.1
PlainData
arguments.0
PlainData
oResult
Result
[value = ON] on(alias):void
saveNodeInfo(alias, nodeInfo, "a"):boolean
[value = OFF] off(alias):void
<undefined>(alias, nodeInfo, "d")
[value = UPDATE] update(alias):void
saveNodeInfo(alias, nodeInfo, "u"):boolean
<constructor>("resultCode", code, "String", this.domRootDoc, "r")
<constructor>("boolTextCode", String.valueOf(result), "Boolean", this.d...
setResults(arguments):void
BuildResult(ResultSet):void
<constructor>(InetAddress,String,Node,Document)

Diagrama de Secuencia para el servicio SendAssetOptionHandlerForTextEditor



Capítulo 10

Iteración 7: Nodo Tool en Together 6.2

10.1 Requisitos

Número	reqNoToR01
Descripción	Ampliar la funcionalidad de la herramienta CASE (*Together*) para permitir la creación de componentes software reutilizables, a saber, *assets* individuales o *assets* formando un *mecano*. Creación de un módulo *Together.*
Prioridad	Alta

Número	reqNoToR02
Descripción	Permitir la realización de operaciones complejas (creación de elementos reutilizables y su almacenamiento), de una forma sencilla para el usuario.
Prioridad	Media

Número	reqNoToR03
Descripción	Procesamiento transparente al usuario, sólo se mostrará la información necesaria para el usuario. Con un solo “click” de ratón podrá realizar la operación.
Prioridad	Media

Número	reqNoToR04
Descripción	Ampliar la funcionalidad de la herramienta CASE (*Together*) para posibilitar la conexión con un repositorio, utilizando el diseño e implementación del framework, utilizando como modelo de reutilización el de *mecano* y consiguiendo la inserción automática en el repositorio.
Prioridad	Alta

Número	reqNoToR05
Descripción	Poder mandar un elemento reutilizable de tipo Asset con el que operará el nodo *BrokerFile* insertándolo en el repositorio.
Prioridad	Alta

Número	reqNoToR06
Descripción	Crear algún servicio propio de la estructura del framework e implantarlo en la herramienta CASE. Para ver la comunicación y resultados de éstos entre nodos.
Prioridad	Alta

10.2 Diseño

Descripción

En esta sección se detallará el análisis y parte funcional (entrando incluso un poco en su implementación) tenida en cuenta para aportar a la herramienta Together los elementos necesarios para acoplar el *framework* detallado.

Esta sección se compone de dos grandes partes, una que sería la instalación en Together 6.2, y otra que sería el servicio propiamente dicho del nodo.

No nos olvidamos que el nodo Together es también un nodo que puede proporcionar servicios.

Lo que haremos es instanciar el framework desarrollado e insertarlo con dos módulos en Together 6.2, de manera que podemos realizar servicios propios del framework, con el nodo Broker, y algún servicio de otro nodo, para este caso de tipo FileBroker.

Al igual que en el tema anterior, hay determinadas clases que se han mantenido, aunque en su propia definición podemos ver que son casi constantes, como por ejemplo *Consultation*, se ha dejado así pues el objetivo principal era instanciar una entorno con servicios funcionando, se deja para líneas futuras hacer funcionar la inserción de todos los tipos de Assets y Mecano, así como la recuperación de los elementos reutilizables. De esta manera el estudio y recuperación de los elementos así como la unificación de las clases es más cómoda.

Clases relacionadas nodo Togehter 6.2

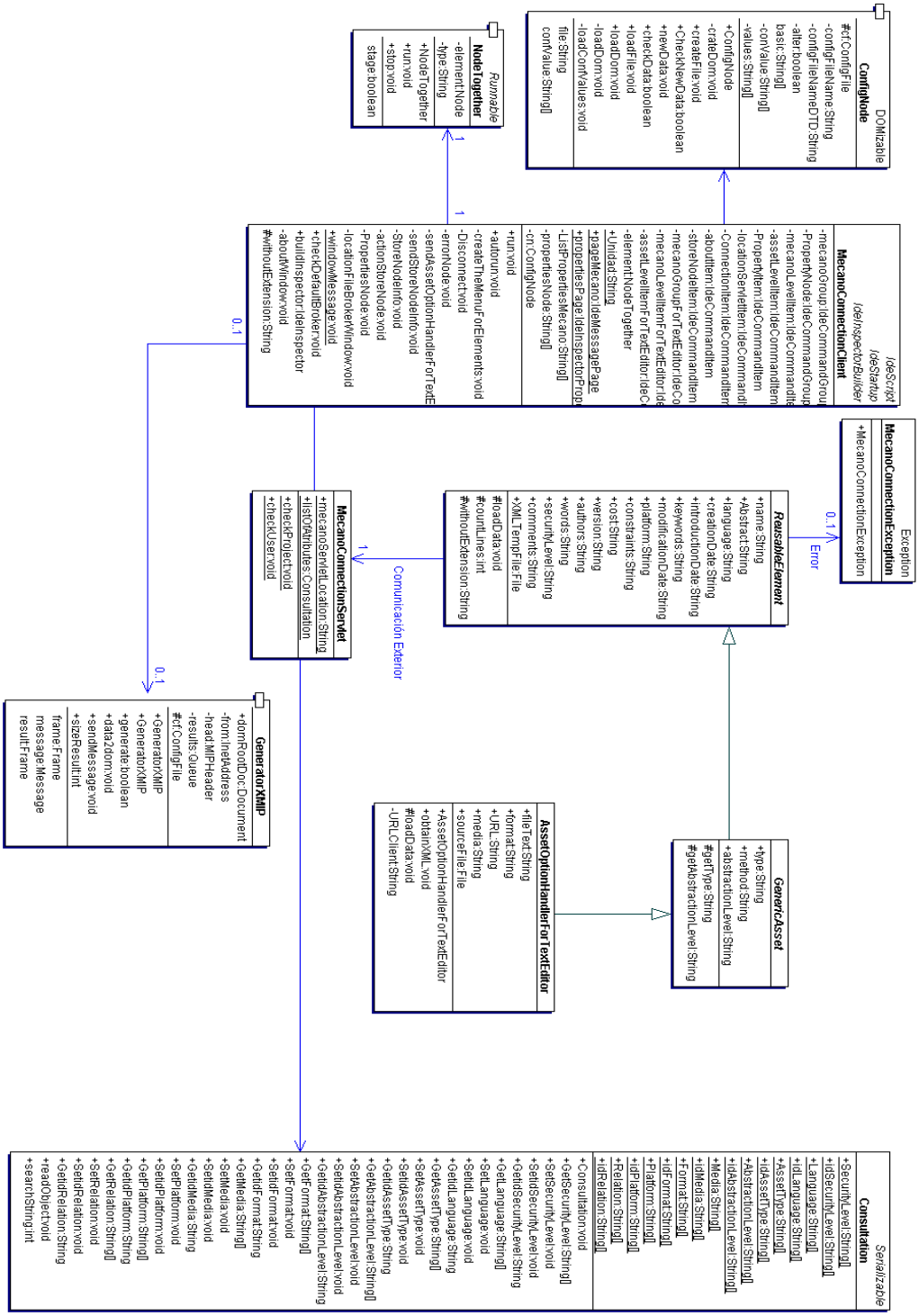
public class **MecanoConnectionClient** implements **IdeScript**, **IdeStartup**, **IdeInspectorBuilder**	Esta es la clase raíz del cliente (del módulo de *Together*). Se encarga de generar las distintas opciones o posibilidades que se ofrecen al usuario de *Together* y de delegarlas a las respectivas clases o métodos, obtenidos del diseño, que las ejecutarán.
public class **ConfigNode** extends **DOMizable**	Esta clase nos muestra la configuración del nodo Together, en ella podremos definir valores que después convertirá en datos formados por una ***DTD***, en un ***XML***. Esta clase forma el ***XML*** si no existe, o lo carga si es que existe.
public class **MecanoConnectionException** extends **Exception**	Maneja errores, excepciones propias del cliente.
public abstract class **ReusableElement**	Implementa el concepto de Elemento Reutilizable según el modelo de Mecano conteniendo todos los atributos y métodos comunes para los distintos Assets y para Mecano, así como diversas funciones que utilizarán aquellos.
public abstract class **GenericAsset** extends **ReusableElement**	Trata de poner en común las características de los distintos Assets, los de diagramas, los códigos. Separamos los assets en diagramas y códigos fuente debido a *Together*, que mantiene dos modelos distintos para su tratamiento.

public class **AssetOptionHandlerForTextEditor** extends **GenericAsset**	Se encarga del tratamiento de la opción por parte del usuario de enviar el código fuente como un asset a la base de datos. También proporciona métodos para que MecanoOptionHandler pueda generar un asset de un código fuente y lo envíe.
public class **MecanoConnectionServlet**	Esta clase se encarga de encargar valores de las propiedades de los elementos reutilizables; inicialmente son constantes pero pueden cargarse de la BBDD. Así nos pueden proporcionar funciones de chequeo y validación del proyecto.
public class **GeneratorXMIP**	Esta es la clase encargada de la comunicación con el exterior, formando el mensaje con los datos oportunos y mandando éste al buzón respectivo para que sea atendido por el servicio cliente del nodo.
public class **NodeTogether** implements **Runnable**	Con esta clase conseguimos crear un elemento propio del framework, definido en la otra etapa del proyecto. Este elemento es un nodo en concreto de la estructura. Es una clase Runnable, para que se quede ejecutando hasta que solicitemos sus servicios.
public class **Consultation** implements **Serializable**	Esta clase posee una estructura de constantes que concuerdan con la BD, para que la información sea coherente a la hora de almacenar.

Diagramas relacionados

	Diagrama de Clases para Servidor Together 6.2 .
	Diagrama de Secuencia de la opción send associated source code file as an asset.
	Diagrama de Secuencia para la opción Test Broker.
	Diagrama de secuencia para la opción StartNode.
	Diagrama de Secuencia de creación del nodo con el framework.
	Diagrama de secuencia para “tirar” un nodo.
	Diagrama de Secuencia para el servicio StoreNodeInfo.

Diagrama de Clases para Servidor Together 6.2



Esta es la parte que va a ser instalada en la herramienta Together

Campos de la clase ***MecanoConnectionCliente***

Tipo	Declaración	Descripción
private IdeCommandGroup	mecanoGroup	Con esta variable lo que hacemos es crear un "grupo", con el que definiremos una opción en el menú de *Together*, en la opción de Tools.
private IdeCommandGroup	PropertyNode	Con esta opción almacenamos la información del nodo en el fichero de configuración, para poder volver a levantarlo y tener siempre presente la última configuración.
private IdeCommandItem	mecanoLevelItem	Con esta variable definimos un elemento del menú, definido anteriormente; para esta variable definimos la opción: "Send project as a Mecano".
private IdeCommandItem	assetLevelItem	Aquí definimos otro elemento del menú, definido anteriormente; con esta variable definimos dos opciones: "Send current diagram as an Asset" y "Send current source code as an Asset".
private IdeCommandItem	PropertyItem	Es otro elemento del menú, con esta definimos las propiedades del mecano
private IdeCommandItem	locationServletItem	Aquí definimos otra opción con la que podemos cambiar la localización del servidor.
private IdeCommandItem	ConnectionItem	Esta opción es la que nos marcará la opción "tirar" la estructura de servidor y nodo creada, destruyendo el nodo.
private IdeCommandItem	aboutItem	Aquí definimos la opción de la ventana de "About".
private IdeCommandItem	storeNodeItem	Este es uno de los servicios creados para el nodo; se pone en contacto con el nodo Broker y almacena la información propia del nodo en el broker. Esta información ha sido previamente informada en una ventana emergente de Together.
private IdeCommandGroup	mecanoGroupForTextEditor	Aquí definimos otro grupo para formar un menú dentro de la edición de texto. Aquí definimos la opción del menú: "Send project as a Mecano".

private IdeCommandItem	mecanoLevelItemForTextEditor	Aquí definimos la primera opción para el último grupo de menú creado, en la edición de texto, esta opción es: “Send project as a Mecano”.
private IdeCommandItem	assetLevelItemForTextEditor	Creamos el segundo elemento para el grupo creado en la edición de texto, esta opción será: “Send current code source file as an Asset”.
public static IdeInspectorPropertySetComponent	propertiesPage	Página de propiedades Together del asset.
public static IdeMessagePage	pageMecano	Página en la ventana de mensajes de *Together*.
private String[]	ListPropertiesMecano	Lista los atributos propios de Mecano, para la formación de XML.
private String[]	propertiesNode = null	Es un array cadena, el cual almacena de forma estructurada los elementos que forman la información que se almacenará en el fichero de configuración. OJO, pues esta información guarda un orden.
private ConfigNode	cn = null	Instancia de la clase ConfigNode, para la creación y almacenamiento de la configuración del nodo.
private NodeTogether	element = null	Este es el elemento que formaremos como Broker, en la clase NodeTogether, de esta forma controlaremos si el nodo ha sido o no creado.

Métodos de la clase ***MecanoConnectionClient***

Tipo	Declaración	Descripción
public void	run(IdeContext context)	Método que será ejecutado cuando se ejecute explícitamente el módulo del menú de módulos de *Together*. Recibe como parámetro el contexto actual.
public void	autorun()	Método que será ejecutado cuando se ejecute *Together*. El parámetro que recibe es el contexto actual.
private void	createTheMenuForElements()	Crea el popup menú (añadiendo menús y comandos) para el contexto de los elementos de modelo (diagramas).

private void	Disconnect()	Si el nodo había sido levantando como una característica mas de Together, procedemos a terminar los procesos instanciados para conseguir esta característica. Si no había levantando tan solo informamos de que el nodo no estaba activo.
private void	errorNode()	Comprueba si hay una instancia en el atributo node, esto significa que el nodo ha sido levantando en Together y si además no ha habido algún error en los estados del nodo, para ello utilizamos el método getStage() propuesto en la arquitectura del ***frameWork***.
private void	sendAssetOptionHandlerForTextEditor (IdeCommandEvent event)	Con esta opción lo que hacemos es mandar el fichero asociado a una clase, al repositorio en Marte que tenemos levantando. Comprueba si el nodo estaba levantando, y si lo está ejecuta la opción.
private void	sendStoreNodeInfo (String[] values)	Envía al broker, la información del nodo Together, que previamente ha sido introducida en una ventana emergente al nodo *Broker*, espera por la respuesta del *Broker*.
private void	StoreNodeInfo()	Comprueba si hay un nodo Broker escuchando, si no lo informa de ello, y si lo hay ejecuta el método *actionStoreNode*().
private void	actionStoreNode()	Este método forma una ventana con el que podremos determinar la posición del nodo Broker. Posteriormente pasa el control de flujo al método *sendStoreNodeInfo()*.
private void	PropertiesNode()	Muestra una ventana emergente con los datos de configuración del nodo, son las propiedades del nodo, para ello instanciará la clase ConfigNode, en la que formalizamos esta información. Después de comprobar que los datos han sido bien informados, los datos se almacenan y se cargan en memoria. Lo primero que hace es comprobar si el nodo está levantado y sin errores.
private void	locationFileBrokerWindow()	Método que permite cambiar la dirección de conexión del nodo filebroker en tiempo de ejecución, y realiza la petición de diccionario, que por ahora está formado de constantes.

public static void	windowMessage (String message)	Con éste formamos una ventana informativa en Together, para que el cliente sepa lo que ocurre en todo momento.
public void	checkDefaultBroker()	Este método da servicio a la función que vemos en el menú <<*Properties Node/ Text Broker >>,* con la que comprobamos que el nodo Broker está activo.
public IdeInspector	buildInspector (IdeContext _context, IdeInspector _inspector)	Función encargada de crear una ventana para la recogida de los valores de Mecano. Esta función es llamada por Together cada vez que elegimos la opción "*properties*" de los elementos que creamos en el entorno Together.
private void	aboutWindow()	Ventana de información Acerca de... información relativa al proyecto.
protected String	withoutExtension (String fileNameString)	Éste método nos devuelve la cadena localizada antes de un punto.

Campos de la clase ***ConfigNode***

Tipo	Declaración	Descripción
private String	configFileName = "NodeConfig.xml"	Nombre del fichero de configuración del nodo.
private String	configFileNameDTD = "nodeConfig.dtd"	Nombre del fichero ***DTD***, con el que formaremos el fichero de configuración.
private boolean	alter	Nos indica si el atributo conValue, ha sido alterado desde fuera de la propia funcionalidad de la clase. Con ello conseguiremos mantener una coherencia entre los valores del fichero de configuración y los valores almacenados en memoria, para el fichero de configuración.
String[]	basic = {"fixed", "zipfilter", "zipfilter.ZIP", "tcpsocketconnection", "TCPConnection", "61300","Rhin", "157.88.124.87", "61301", "MECANO", "mecano", "MecanoModel", “Duero", ”Tool", "services", "functions"};	Valores por defecto que tomará el fichero de configuración si es que no existiera. Ojo con la disposición de los elementos hay que tenerla en cuenta. Estos son los valores *default* de configuración del nodo.
private String[]	conValue = null	Nos sirve para almacenar los valores de los elementos definidos en el fichero de configuración, estas unidades son: connection, filterFactory, filterChain,

		connectionListener, router, model, node.
private String	file =""	En esta variable insertamos todo el fichero de configuración, en formato cadena, para poder tratarlo como texto plano. El fichero de configuración viene definido por el atributo *configFileName*.

Constructores de la clase ***ConfigNode***

Tipo	Declaración	Descripción
public	ConfigNode()	Lo primero que hacemos es cargar el fichero de configuración, para ello utilizamos *loadFile()*, llamamos a *loadDom()* y *loadConfValues()*, si es que existe, si no creamos nuestro propio fichero con los parámetros que hemos definido en *basics*, y se lo pasaremos a la función *newData* para que forme el nuevo fichero.

Métodos de la clase ***ConfigNode***

Tipo	Declaración	Descripción
private void	crateDom()	Crea la estructura ***DOM*** dependiendo de los valores que haya en el atributo *conValue*.
public void	createFile()	Almacena en el fichero de configuración nombrado según el atributo *configFileName*, lo que tenemos en memoria en el atributo *file*.
public boolean	CheckNewData (String[] data)	data, su argumento nos muestra la nueva configuración que obtenemos por la ventana emergente de ***Together***. Este argumento nos servirá para actualizar el método *conValue*, si después de chequear los datos con *CheckData()*, no hay ningún problema creamos la estructura ***DOM***, con *createDom()* y el fichero en texto plano.
public void	newData (String[] data)	Crea una nueva estructura ***DOM*** y el fichero de configuración en texto dato. Las unidades que requerimos en el fichero serán marcadas por el argumento *data*. El argumento se almacena en el atributo *conValue* y ejecuta la función *getXML()* heredada.
public boolean	checkData()	Con este método comprobamos que todos los valores introducidos son correctos: que las clases de configuración existen, si los puertos a los que hacemos referencia están disponibles. Si ha habido algún error devuelve *false y* visualiza en una pantalla que ha habido un problema, si no devuelve *true*.
public void	loadFile()	Carga el fichero de configuración, indicado por configFileName, en el atributo *file*. Si el fichero no existe entonces iguala conValue a *null* y file a

		cadena vacía, que nos servirá como evento para saber que no hay fichero de configuración y hay que crearlo.
public void	loadDom()	Utilizando las clases del paquete *utils*, generamos una estructura ***DOM***, almacenando document, raíz del documento en *domRootDoc*. El fichero que pasamos debe estar cargado en el atributo *file*.
private void	loadDom (String file)	En file, introducimos el fichero en cadena, lo cargamos en el atributo de la clase *file*, y llamamos a *loadDom()* para crear la estructura ***DOM***.
private void	loadConfValues()	Carga los valores básicos que deben estar formados en el fichero de configuración, en el atributo *conValue*.
public void	setFile (String file)	Actualizamos el valor del atributo *file*, con el del argumento de este método.
public String	getFile()	Devolvemos el valor del atributo *file*.
public String[]	getConfValue()	Nos devuelve el array con todos los parámetros de configuración que hemos utilizado para formar el fichero de configuración.
public void	setConfValue (String[] conf)	Actualizamos el valor del array del atributo *conValue*, con el valor del array argumento. No generamos el nuevo fichero, por lo que tenemos un atributo, *alter*, que nos indica que hay cambios que se deben tener en cuenta.

Constructores de la clase ***MecanoConnectionException***

Tipo	Declaración	Descripción
public	MecanoConnectionException (String message)	Método de creación para manejar errores, excepciones propias del cliente; recibe una cadena que será el mensaje presentado.

Campos de la clase ***ReusableElement***

Tipo	Declaración	Descripción
public String	name = null	Variable que designa el nombre del asset o mecano según lo que trate.
public String	Abstract = null	Explicación o definición general del elemento reutilizable.
public String	language = null	Lengua en la que se encuentra el elemento reutilizable.
public String	creationDate = null	Fecha de creación del elemento reutilizable.
public String	introductionDate = null	Fecha en la que el elemento reutilizable fue introducido en el repositorio.
public String	keywords = null	Palabras clave, frases que describen aspectos importantes del elemento reutilizable. De esta

		forma se facilitarán las búsquedas de los elementos reutilizables.
public String	modificationDate = null	Fecha de la última modificación del elemento reutilizable.
public String	platform = null	Plataforma de desarrollo del elemento reutilizable.
public String	constraints = null	Información legal sobre el uso del elemento reutilizable, incluyendo copyright, patentes, derechos de explotación, limitaciones de explotación y licencias.
public String	cost = null	Cuantía que debe pagarse por la reutilización del elemento reutilizable.
public String	version = null	Designación de la versión de un elemento reutilizable.
public String	authors = null	Nombre del autor/es que desarrolló el elemento reutilizable.
public String	securityLevel = null	El nivel de seguridad más alto asignado al elemento reutilizable o cualquier parte constituyente del elemento reutilizable.
public String	comments = null	Comentarios acerca del fichero físico.
public File	XMLTempFile	Fichero que almacenará el XML con la información administrativa del asset antes de ser enviado al servidor.

Métodos de la clase ***ReusableElement***

Tipo	Declaración	Descripción
protected void	loadData()	Método que extrae los datos de un elemento reutilizable de la página de propiedades generada en Together y los almacena en sus respectivos atributos.
protected int	countLines (String file)	Función de miscelánea para contar las líneas de un fichero mediante el API de *Together*.
protected String	withoutExtension (String fileNameString)	Función de miscelánea para extraer el nombre de un fichero sin extensión. Reciba la cadena con el nombre del fichero con extensión y devuelve la cadena con el nombre del fichero sin extensión.

Campos de la clase ***GenericAsset***

Tipo	Declaración	Descripción
public String	type	Tipo del asset.
public String	method = null	Método de desarrollo utilizado para generar el asset.
public String	abstractionLevel = null	Nivel de Abstracción.

Métodos de la clase ***GenericAsset***

Tipo	Declaración	Descripción
protected String	getType (String _name)	Esta función pasa de Inglés a Español el tipo que devuelve *Together* para que sea acorde con la *BD*. Recibe como parámetro el tipo obtenido de *Together*, devuelve la cadena según la *BD*.
protected String	getAbstractionLevel (String _type)	Obtenemos los niveles de abstracción del diagrama actual dependiendo de su tipo. Recibe como parámetro el nombre obtenido de *Together* del tipo de diagrama, devuelve una cadena según contenido de la *BD*.

Campos de la clase ***AssetOptionHandlerForTextEditor***

Tipo	Declaración	Descripción
public String	fileText = ""	Fichero asociado a la clase.
public String	format = null	Formato en el que se encuentra el *asset*.
public String	URL = null	Dirección donde se encuentra el fichero físico.
public String	media = null	Medio en el que se encuentra el *asset*.
public File	sourceFile	Fichero que almacena el asset (código fuente).

Constructores de la clase ***AssetOptionHandlerForTextEditor***

Tipo	Declaración	Descripción
public	AssetOptionHandlerForTextEditor (IdeCommandEvent event)	Método de creación utilizado por Mecano para crear un Asset Código Fuente. Se rellenan los datos propios del Asset. Recibe como parámetros el objeto Together, con el que obtendremos el elemento que esta debajo de la posición del ratón.

Métodos de la clase ***AssetOptionHandlerForTextEditor***

Tipo	Declaración	Descripción
public void	obtainXML()	Creamos el *Asset* de tipo textual, para ello utilizamos las funciones *setAttribute()* y *setElement()*, propias de la clase *Attribs* creada en el *frameWork*. También formamos su estructura ***DOM.*** Forma el mensaje y lo envía, esperando la respuesta del nodo destino.
protected void	loadData()	Carga los datos propios de esta clase, lo primero que hace es llamar al método heredado, pues cargará las propiedades comunes a otros objetos.
private String	URLClient (String sourceFileURL, String nameMecano)	Método que devuelve la dirección de almacenaje de los ficheros fuente del proyecto pero sólo a partir del directorio del proyecto. Esta dirección se utilizará en el servidor reproduciendo la estructura de almacenaje del proyecto en el

		cliente. Recibe como parámetro, la dirección completa del fichero fuente y nombre del proyecto.

Campos de la clase ***MecanoConnectionServlet***

Tipo	Declaración	Descripción
public static Consultation	listOfAtributes	Este atributo nos servirá en un futuro para crear el paquete de intercambio de información de tipos entre cliente y la *BD*, con los datos formados según esta. Para ello habrá que crear un método que se ponga en contacto con la *BD*, y obtenga todos los formatos propios de los elementos que podremos formar.

Métodos de la clase ***MecanoConnectionServlet***

Tipo	Declaración	Descripción
public static void	checkProject()	Comprueba si hay un proyecto abierto. Si no hay ninguno abierto sale de la aplicación.. Debemos controlar la excepción MecanoConnectionException.
public static void	checkUser()	Esta función se queda pendiente de la definición de la autentificación del usuario en el *Broker* o *FileBroker*.

Campos de la clase ***NodeTogether***

Tipo	Declaración	Descripción
private Node	element	Es el tipo de nodo que se ha cargado en la arquitectura Together. Èste puede ser Broker, Processor, Repository, FileBroker, Tool.
private String	type	Es la descripción textual del tipo, que nos servirá para cargar las diferentes clases. El valor de éste puede ser: Broker, Processor, Repository, FileBroker, Tool.

Constructores de la clase ***NodeTogether***

Tipo	Declaración	Descripción
public	NodeTogether (String type)	Crea una instancia de esta clase y marca el tipo del nodo, valor que se pasa como argumento.

Métodos de la clase ***NodeTogether***

Tipo	Declaración	Descripción
public void	run()	Implementa el método de la clase *Runnable*, para conseguir un nuevo hilo de ejecución. Carga el nodo dependiendo del tipo definido al crear la instancia de la clase.
public void	stop()	Para los servicios clientes y servidores definidos en el framework, para poder parar las

		propiedades del nodo.
public boolean	getStage()	Llama a la función *getStage()* del atributo *element*, si es que existe. Esta función nos dice si el nodo se levantó sin ningún problema.

Campos de la clase ***GeneratorXMIP***

Tipo	Declaración	Descripción
public Document	domRootDoc	Atributo raíz de la estructura DOM que formaremos para el mensaje que enviaremos. Este elemento debe ser común a todos los objetos que forman parte del mensaje.
private Message	message	Atributo que nos muestra el mensaje objeto de la comunicación. Esta clase ha sido definida en la estructura del *frameWork*.
private InetAddress	from	Objeto definido en la estructura framework, con la que definimos la dirección origen del mensaje.
private MIPHeader	head	Atributo nos marca el elemento cabecera del mensaje.
private Queue	results	Es la cola en la que introduciremos el mensaje respuesta recibido, dependiente del mensaje original enviado.
protected	ConfigFile cf = ConfigFile.getInstance()	Instancia para cargar los valores del fichero de configuración del nodo.

Constructores de la clase ***GeneratorXMIP***

Tipo	Declaración	Descripción
public	GeneratorXMIP()	Genera la instancia de la clase, con los atributos anteriores inicializados a *null*, a excepción de *result*, que crea la cola y de *documentRoot*, que obtiene el raíz de la clase *DOMSource*, definida en la estructura del framework.
public	GeneratorXMIP (String xmlPlainMessage, InetAddress from)	Genera una instancia del la clase, inicializando sus atributos; *documentRoot*, que obtiene el raíz de la clase *DOMSource; result*, con la clase *Queue*; El argumento *xmlPlainMessage* nos sirve para poder crear el objeto *Message*, pues es el mensaje que vamos a enviar en texto plano; y *from*, es el elemento-dirección del que partirán los mensajes.

Métodos de la clase ***GeneratorXMIP***

Tipo	Declaración	Descripción
public boolean	generate (NodeAddress toNode)	Forma el mensaje con todos los elementos que lo conforman. El parámetro toNode nos indica a que nodo va dirigido.
public void	data2dom()	Pasa de los datos formados en memoria, a una estructura ***DOM***, para ello llama a la función

		data2dom() de la clase *Message*.
public void	sendMessage()	Envía el mensaje que hemos conseguido formar, y controla el resultado obtenido. Lo primero que hace es crear un objeto, *ReceiverMailbox,* para enviar el mensaje, utilizando el método *send()* de *ReceiverMailbox*, que nos indica el número de resultados que debe esperar, uno por cada *frame* del mensaje *Request*. Por último introduce los mensajes resultados en la cola de resultados.
public int	sizeResult()	Nos indica el tamaño de la cola de resultados.
public Frame	getResult()	Devuelve el objeto de la cola de resultados.
public void	setFrame (Frame item)	Añade un frame, al mensaje que vamos formando, para ello utiliza el método *addFrame()* de la clase *Message*.
public Message	getMessage()	Devuelve el objeto *Message* que vamos formando.

Campos de la clase ***Consultation***

Tipo	Declaración	Descripción
public static String[]	SecurityLevel	Nombre del nivel de seguridad.
public static String[]	idSecurityLevel	Identificador asignado al nivel de seguridad en la BD.
public static String[]	Language	Nombre del lenguaje empleado.
public static String[]	idLanguage	Identificadores de los lenguajes designados en la BD.
public static String[]	AssetType	Nombre del tipo de Asset.
public static String[]	idAssetType	Identificador del tipo de asset según la BD.
public static String[]	AbstractionLevel	Nombre del nivel de abstracción.
public static String[]	idAbstractionLevel	Identificador del nombre del nivel de abstracción según BD.
public static String[]	Media	Nombre de Media.
public static String[]	idMedia	Identificador del nombre de Media según BD.
public static String[]	Format	Nombre de formato.
public static String[]	idFormat	Identificador según BD del formato.
public static String[]	Platform	Nombre de la plataforma.
public static String[]	idPlatform	Identificador de la plataforma.

public static String[]	Relation	Nombre de la relación.
public static String[]	idRelation	Identificador del nombre de la relación.

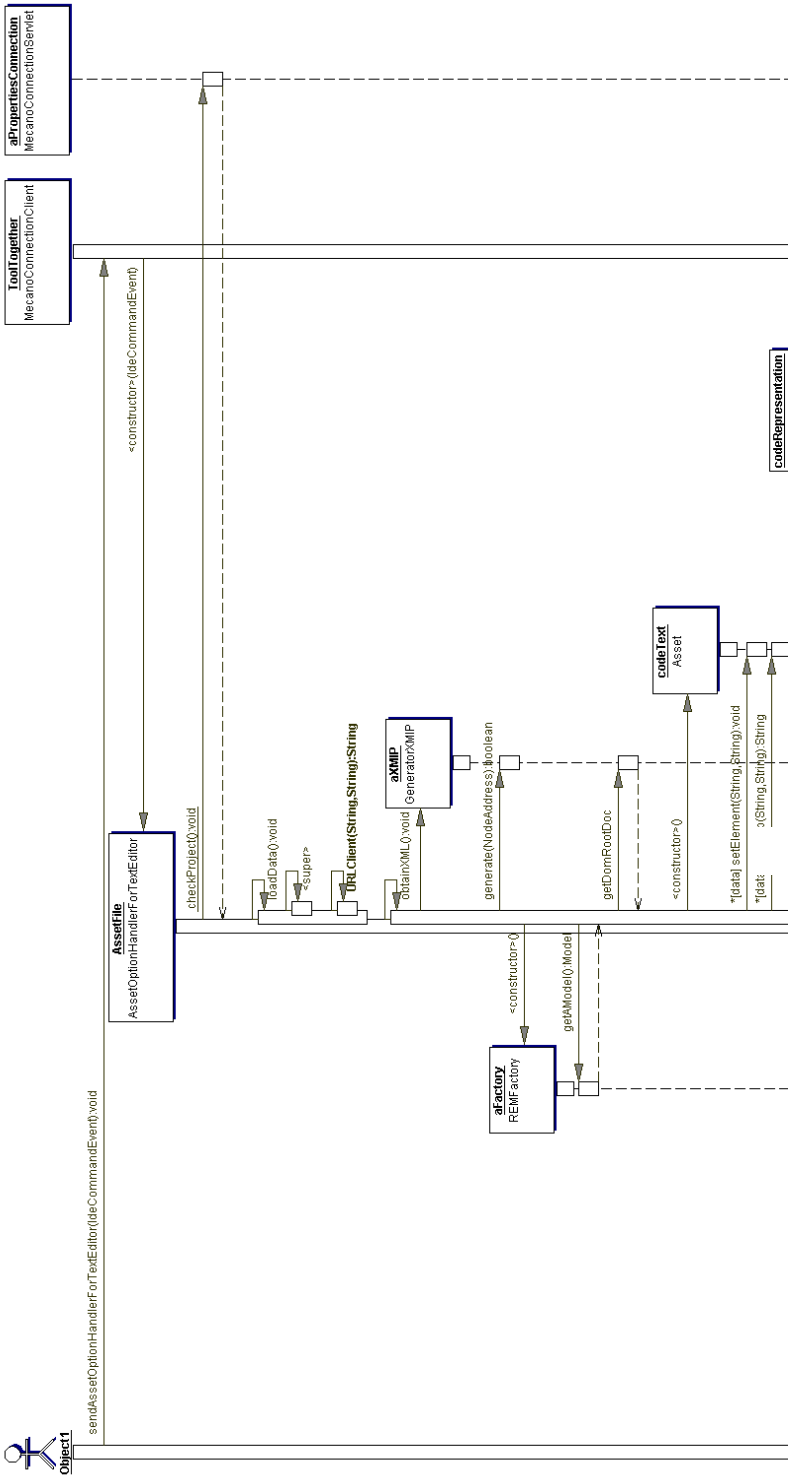
Constructores de la clase ***Consultation***

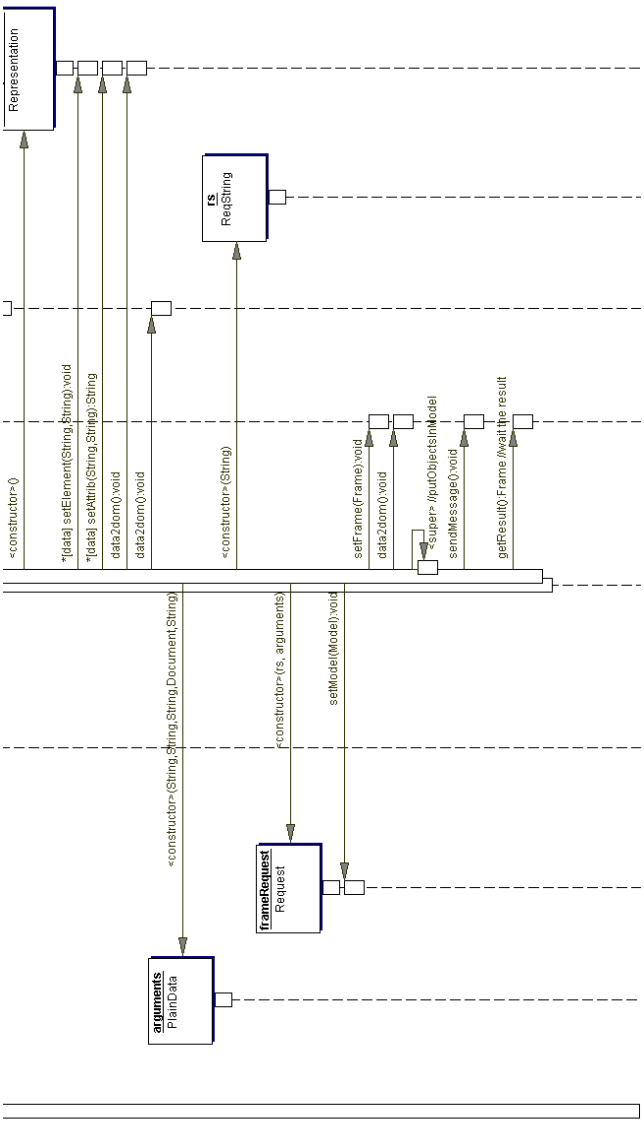
Tipo	Declaración	Descripción
public void	Consultation()	Creamos el objeto Consultation en memoria e inicializamos todos los atributos.

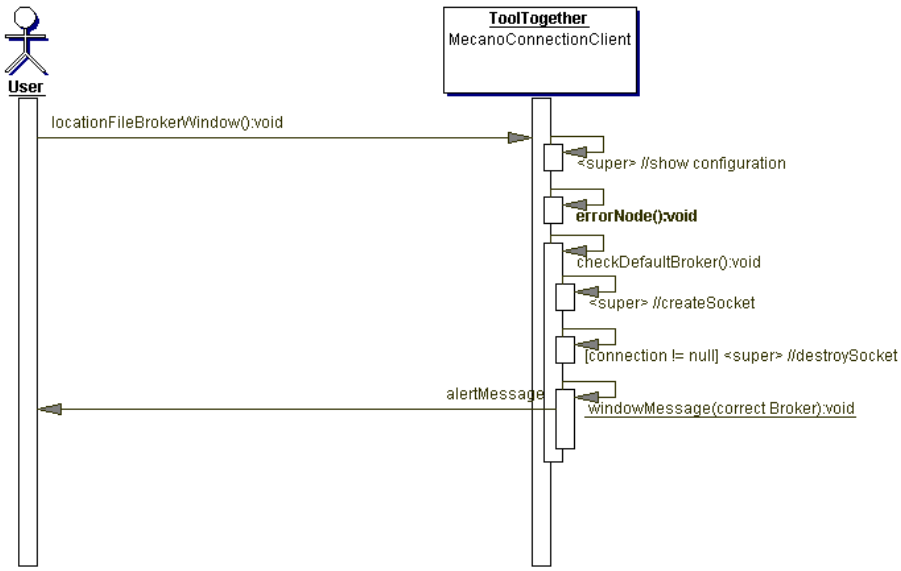
Métodos de la clase ***Consultation***

Los métodos de esta clase son los típicos para dar valor y extraer los atributos antes indicados. Todos poseerán el siguiente formato: Get<atributo>, Set<atributo>

Diagrama de Secuencia de la opción: send associated source code file as an asset




Representation
rs
RepString
«constructor»()
*[data] setElement(String,String):void
*[data] setAttrib(String,String):String
data2dom():void
data2dom():void
«constructor»(String)
setFrame(Frame):void
data2dom():void
«super» //putObjectsInModel
sendMessage():void
getResult():Frame //wait the result
«constructor»(String,String,String,Document,String)
«constructor»(rs, arguments)
setModel(IModel):void
frameRequest
Request
arguments
PlainData

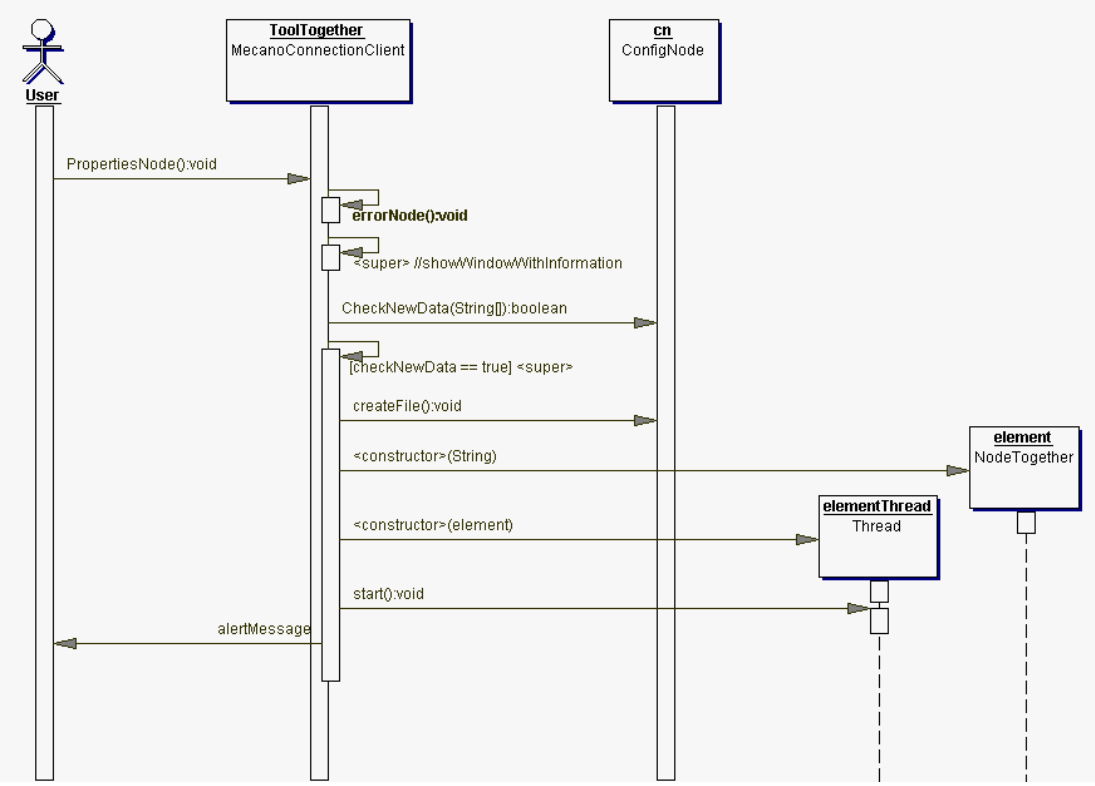


Aquí podemos ver como comprobamos que haya un broker escuchando a un nodo de la estructura. Desde esta opción se tiene la posibilidad de cambiar la dirección a la que está asociado el broker.

10.2 Diseño

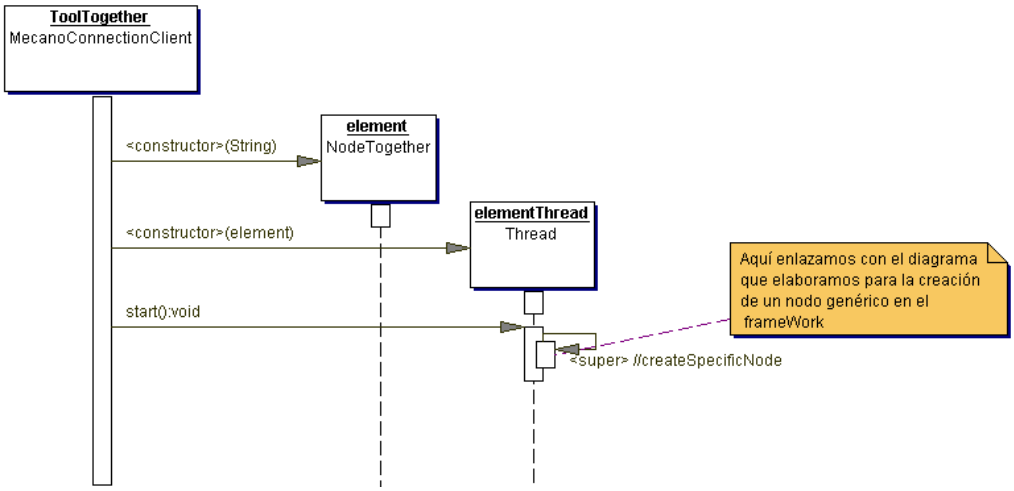
Tenemos claro que la creación del framework supone la construcción de una arquitectura y podemos emplear esta característica para utilizar las clases y conocer más profundamente las características del framework y de estas últimas; en este diagrama podemos ver como hemos utilizado alguna de las clases de la arquitectura aunque, se ha hecho así para dar más pruebas y visiones sobre el framework. Claro que lo podríamos haber desarrollado directamente mandando construir el mensaje desde un fichero patrón almacenado.

Diagrama de secuencia para la opción StartNode



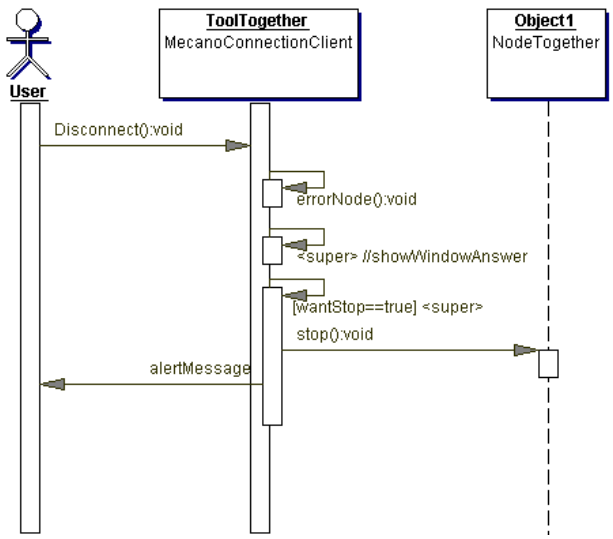
Levantamos el nodo y todas sus particularidades. El siguiente diagrama nos servirá para entender la conexión con la estructura del framework creada.

Diagrama de Secuencia de creación del nodo con el framework



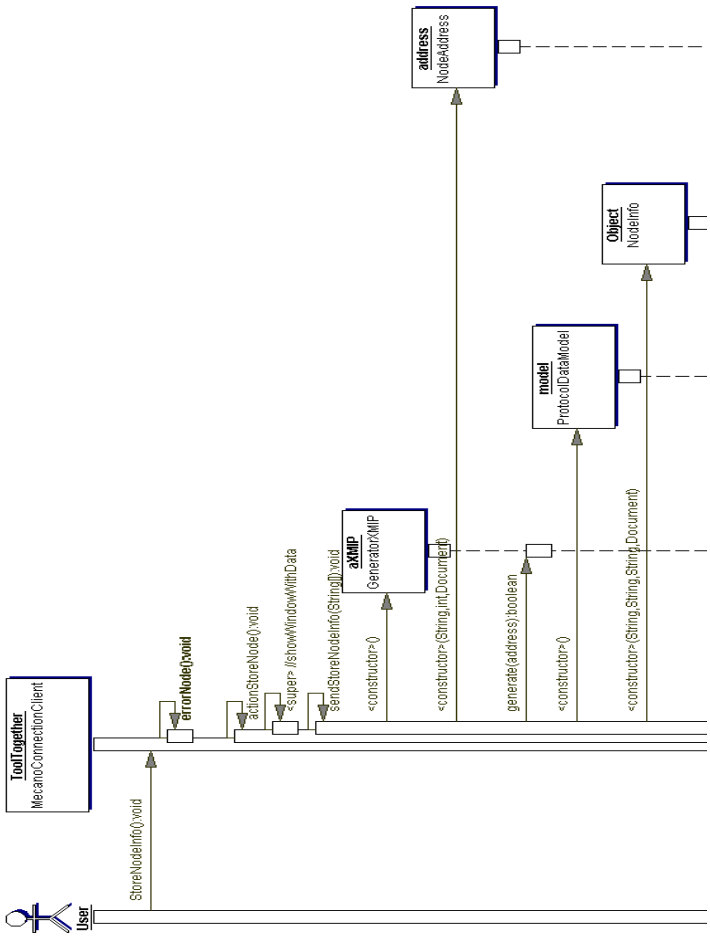
En este diagrama vemos como creamos un nodo específico ya sea Broker, FileBroker, Repository, Tool o Processor. Enlazamos directamente con el diagrama de secuencia presentado en el paquete nodecore, creación de un nodo genérico.

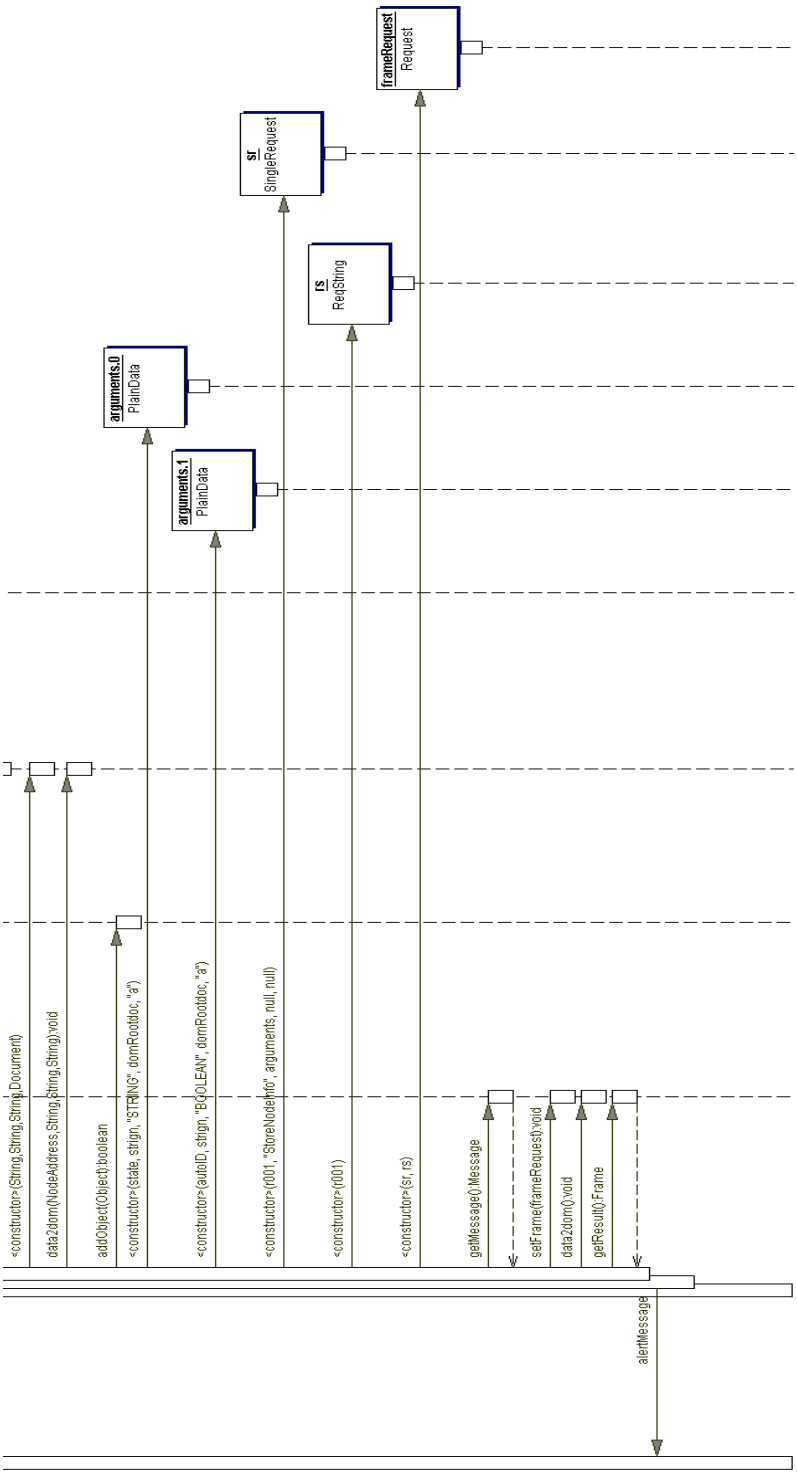
Diagrama de secuencia para "tirar" un nodo



***Así** terminamos con el servicio del nodo Together, eliminando sus propiedades*

Diagrama de Secuencia para el servicio StoreNodeInfo




frameRequest
Request
sr
SingleRequest
rs
RegString
arguments.0
PlainData
arguments.1
PlainData
<constructor>(String,String,String,Document)
data2dom(NodeAddress,String,String):void
addObject(Object):boolean
<constructor>(state, strign, "STRING", domRootdoc, "a")
<constructor>(autoID, strign, "BOOLEAN", domRootdoc, "a")
<constructor>(r001, "StoreNodeInfo", arguments, null, null)
<constructor>(r001)
<constructor>(sr, rs)
getMessage():Message
setFrame(frameRequest):void
data2dom():void
getResult():Frame
alertMessage

En el diagrama anterior podemos ver como se hace uso del servicio NodeStoreInfol. Hemos creado el mensaje a mano, esto se puede hacer de múltiples formas, pues podríamos tener un fichero patrón, en el que hubiese formado un ejemplo con un fichero en el que hubiésemos formado el mensaje de forma correcta. de esta manera se podría cargar los objetos llamando directamente a la clase Message, y ella se encargaría de formar todos los elementos. Puntualizando esta opción es mucho más "clara" que la empleada, pero opté por hacerlo de esta manera para que quedase plasmado la manera manual de hacer las cosas, pues en todas las pruebas efectuadas la carga de los elementos se ha hecho con el fichero patrón.

10.3 Registro de pruebas Unitarias

Al igual que en la iteración anterior, las pruebas unitarias que se han realizado son el entorno de explotación que hemos obtenido. Todas las pruebas se han realizado en él y nos han servido para crear un manual de Usuario, ahí podemos ver las pruebas, las clases utilizadas son las propia implementación de los servicios, pues el solicitar un servicio va implementado en la herramienta Together. El resultado ha sido el esperado.

Capitulo 11

Conclusiones y Líneas Futuras

11.1 Conclusiones

Este trabajo se ha extendido demasiado en funcionalidad y por ello en el tiempo. Como resultado se ha obtenido una framework, un entorno de explotación y una extensa documentación, la cual pretende ser una revisión de la memoria [*Per03*], y que se ha tenido que particionar creando apéndices que se localizan en el CD.

En todo momento, se ha pretendido que el documento esté orientado a cualquier tipo de usuario del framewok, consiguiendo una documentación lo menos pesada posible y diferenciando las partes, para que ésta sea a la vez una pequeña guía de cambios y definiciones.

La primera memoria sobre que la que me baso, posee muchas consideraciones teóricas, que se han intentando matizar desde un punto de vista práctico. Considero que se ha conseguido poner en marcha la instanciación y evolución del framework, está en la labor de nuevos usuarios ampliar esta aplicación o instanciar nuevas, utilizando componentes que ya han sido probados e instanciados.

Se ha intentado conseguir todos los objetivos, aunque acotar un proyecto de estas dimensiones es algo que, en muchos momentos, se escapaba de las manos, entrando en nuevas funcionalidades y partes del framework que no eran objeto de este proyecto.

Ha quedado patente la complejidad y extensión de un proyecto de este tipo, y como ya hemos dicho, se han dejado partes definidas y partes que, además, han sido desarrolladas dándolas un carácter abierto, para que la futura evolución sea lo mas flexible posible.

11.2 Líneas de acción y trabajo futuro

Finalmente, hemos conseguido un núcleo funcional y un entorno real sobre el que hemos instanciado el framework con las partes principales, para poder comenzar a integrar herramientas de desarrollo basadas en reutilización.

El presente proyecto seguirá evolucionando sobre servicios que no se han implementado, incluso sobre otros nuevos, como puede ser el enrutado de mensajes con más de un Broker.

La línea de trabajo futura puede discurrir por dos corrientes distintas, una que se encargue del entorno de explotación propuesto, añadiendo mas servicios definidos en [*NCAP02*]. Otra sería la evolución e implementación del framework (se pueden seguir implementando nuevos servicios).

Se ha intentado respetar el análisis realizado en los proyectos de los que partimos, por lo que el desarrollo de los servicios del entorno de explotación y el proyecto [*NCAP02*] son muy parecidos, intentado lograr el objetivo de acoplar los servicios de manera rápida y eficiente.

Se podrá seguir trabajando en la adaptación sobre herramientas, para que hagan uso de la tecnología desarrollada en el presente trabajo, pudiéndose convertir en un soporte completo para el proceso de reutilización del grupo GIRO [*LGBLG03*].

Apéndices

(Todos los apéndices se encuentran ubicados en el CD adjunto en la documentación).

Apéndice A

Documento en el que se encuentra definido los elementos que forman la DTD de los mensajes que podemos formar con este framework .

Apéndice B

Documento en el que se encuentra definido la DTD de configuración de los nodos con un ejemplo práctico del mismo.

Apéndice C

En el Capítulo C.1 se presenta la arquitectura propuesta, la definición de los tipos de entidades (nodos) que participan y los servicios que estas pueden ofrecer.

En la Capítulo C.2 se describe el protocolo de comunicación empleado, prestando especial atención al formato de los mensajes. Se describe su utilización de forma genérica para cualquier modelo de componente reutilizable y se expone su adaptación particular para el modelo de MECANO.

En el Capítulo C.3 se describen más extensamente los servicios que pueden ofrecer los nodos del entorno de desarrollo.

Todos estos capítulos son la revisión de los capítulos que se indican en la memoria [Pero03], con las actualizaciones producidas por este trabajo.

Apéndice D

Diccionario de clases con la descripción de cada método y la acción efectuada. Posee 7 secciones dependiendo de la iteración en el que se encuentre la clase.

Apéndice E

Manual de Instanciación del framework

Apéndice F

Manual de Instalación

Apéndice G

Manual de Usuario

Bibliografía y Referencias

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Java Series. Sun Microsystems, 1996.

[DOM01] Document object model (dom) level 3 core specification, Septiembre 2001. W3CWorking Draft.

[CLP03] Y. Crespo, M. A. Laguna, F. J. Pérez. Integrando un modelo de reutilización en la producción de software: entorno distribuido para el desarrollo basado en reutilización. VIII Jornadas ISBD, Pág. 79-94 , 12 Noviembre de 2003.

[PGCG03] F. J. Pérez and Y. Crespo. Framework para la integración de herramientas de desarrollo basado en reutilización. In IV *Jornadas de Trabajo DOLMEN*, 12-14 de Noviembre de 2003.

[Gar00] F.J. García. *Modelo de reutilización soportado por estructuras complejas de reutilización denominadas mecanos*. PhD thesis, Departamento de Informática y Automática, Universidad de Salamanca, Febrero 2000.

[HGL01] C. Hernández, F.J. García, and M. A. Laguna. La biblioteca de reutilización giro. In *I Jornadas de Trabajo DOLMEN*, pages 102–112, 12-13 Junio de 2001- Sevilla (España), Junio 2001.

[jav03] The java web services tutorial, Febrero 2003. en: http://java.sun.com.

[jds] Java data structures library. en: http://jdsl.org.

[NCAP02] Alberto Nistal Calvo and Julián Andújar Puertas. *Mecanos representando proyectos Together: Inserción automática en el repositorio de GIRO*. Julio 2002. Proyecto de fin de carrera.

[para] Parser xml de ibm. en: http://www.alphaworks.ibm.com/tech/xml4j.

[parb] Parser xml del proyecto apache. en: http://xml.apache.org/xerces2-j/index.html.

[PLPG00] Mª Elena Pérez León and Almudena Pulpón García. *Diseño y Desarrollo de un Repositorio para la Reutilizacón de Mecanos*. Junio 2000. Proyecto de fin de carrera.

[Pre02] Roger S. Pressman. *Ingeniería del Software, Un enfoque práctico*. Mc Graw Hill, quinta edición edition, 2002.

[Per03] Francisco Javier Pérez García. *Entorno de Desarrollo Distribuido Basado en Reutilización para el Modelo de Mecano*. Septiembre 2003. Proyecto fin de carrera.

[RH01] Elliotte Rusty Harold. *XML Bible, Second Edition*, 2001.

[T6.2] http://www.borland.com/downloads/download_together.html