UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de repositorios de Modelos y de Componentes Software para dar soporte al desarrollo de sistemas autoadaptativos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Miguel Blanco Máñez
Tutor: Joan Fons i Cors
Curso 2018/2019

"La diferencia entre la teoría y la práctica es que, en teoría,
no hay diferencia entre la teoría y la práctica"

– Benjamin Brewster, "The Yale Literary Magazine", Febrero 1882.

Resumen

En este trabajo se van a desarrollar dos prototipos de servicios (un repositorio de modelos de componente y otro de componentes software) que actuarán como almacenes y gestores de recursos. Estos recursos se emplearán como parte de una herramienta de apoyo al diseño y desarrollo de sistemas autoadaptativos. Dada la naturaleza de esta herramienta, estos repositorios deberán poder emplearse para almacenar artefactos empleados por el propio sistema autoadaptativo tanto en tiempo de diseño como en el de ejecución.
La implementación se realizará mediante un servicio *REST*ful donde la aplicación podrá enviar los modelos de componente y sus componentes software al repositorio, el cual los almacenará bajo una estructura definida de directorios. Si en un determinado instante la aplicación necesita hacer uso de los archivos guardados, el repositorio se los debe suministrar, además la aplicación podrá actualizar en el repositorio cualquier archivo enviado previamente, así como borrarlo.
Para llevar acabo este trabajo realizaremos una primera aproximación creando el repositorio dentro del sistema de archivos de la máquina que contiene la aplicación. Posteriormente pasaremos a crear el servicio *REST*ful que contendrá el repositorio definitivo. Pondremos especial interés en la seguridad en las comunicaciones y en el envío de ejecutables como cadenas de texto.

Palabras clave: repositorio,modelos de componente,componentes software,MAPE-K,*REST*ful, autoadaptativo.

Abstract

In this work, two prototypes of services will be developed (a repository of component models and a repository of software components) that will act as warehouses and resource managers. These resources will be used as part of a tool to support the design and development of self-adaptive systems. Given the nature of this tool, these repositories should be able to be used to store artifacts used by the self-adaptive system itself, both in design time and in execution time.
This implementation will be done through a *REST*ful service where the application will be able to send the component models and their software components to the repository, which will store them under a defined directory structure. If in a certain moment the application needs to make use of the saved files, the repository must supply them. The application can update any previously sent file in the repository, as well as delete it.
To carry out this work we will make a first approximation by creating the repository within the file system of the machine that contains the application. Later we will create the *REST*ful service that will contain the final repository. We will take special interest in the security of communications and the sending of executables as text strings.

Keywords: repository, component model, software component, MAPE-K, *REST*ful, self-adaptive.

Resum

En aquest treball es van a desenvolupar dos prototips de serveis (un repositori de models de component i un altre de components de programari) que actuaran com magatzems i gestors de recursos. Aquests recursos es faran servir com a part d'una eina de suport al disseny i desenvolupament de sistemes auto-adaptatius. Donada la naturalesa d'aquesta eina, aquests dipòsits han de poder emprar-se per emmagatzemar artefactes emprats pel propi sistema autoadaptativo tant en temps de disseny com en el d'execució. Aquesta implementació es realitzarà mitjançant un servei *REST*ful on l'aplicació podrà enviar els models de component i els seus components de programari al repositori, el qual els emmagatzemarà sota una estructura definida de directoris. Si en un determinat instant l'aplicació necessita fer ús dels arxius guardats, el repositori se'ls ha de subministrar. L'aplicació podrà actualitzar en el repositori qualsevol arxiu enviat prèviament, així com esborrar-lo.
Per dur a terme aquest treball realitzarem una primera aproximació creant el repositori dins el sistema d'arxius de la màquina que conté l'aplicació. Posteriorment passarem a crear el servei *REST*ful que contindrà el repositori definitiu. Posarem especial interès en la seguretat en les comunicacions i en l'enviament de executables com cadenes de text.

Paraules clau: repositori, models de component, components de programari, MAPE-K, *REST*ful, autoadaptatiu.

Tabla de contenidos

1. Introducción

Históricamente una de las aspiraciones de la informática, que cobró fuerza a partir de la segunda guerra mundial, ha sido el crear artefactos que se asemejaran a la máquina más perfecta que existe, el ser humano. En 1950 Alan Turing publicó el articulo *Computing machinery and intelligence* [1] en el cuál se plantea si una máquina puede 'pensar' y actuar en consecuencia. Pero no fue hasta la década de los noventa del siglo XX cuando muchas de las empresas tecnológicas comenzaron realizar grandes inversiones en este terreno. La enorme cantidad de información que se genera en el creciente mundo digital empuja a mejorar la capacidad de procesamiento y análisis de estos datos.

Podemos pensar en una ciudad inteligente la cual está equipada con una red de sensores, estos sensores envían la información a un centro que los procesa y según los parámetros que se manejen, el sistema puede realizar acciones en consecuencia. Otro de los casos es el del coche autónomo. Estos vehículos van provistos de sensores que van enviando información al sistema operativo del coche, estos datos se interpretan y el vehículo va realizando acciones en respuesta a esos datos procesados.

Este sistema de recolección, procesado y almacenado de datos con la posterior interpretación de los mismos – lo que llamaríamos computación autónoma – no está completamente desarrollado. La computación autónoma ayuda a abordar esta complejidad mediante el uso de la tecnología para administrar la tecnología.

Tomando como modelo el ser humano el sistema nervioso autónomo controla los latidos del corazón, controla la dilatación de las pupilas, la frecuencia respiratoria y mantiene la temperatura del cuerpo cerca de 37 ºC sin ningún esfuerzo consciente . De la misma manera, las capacidades autónomas de autogestión anticipan los requisitos del sistema y resuelven problemas con una intervención humana mínima. Sin embargo, existe una distinción importante entre la actividad autónoma en el cuerpo humano y las actividades autónomas en los sistemas informáticos. Muchas de las decisiones tomadas por las capacidades autónomas en el cuerpo son involuntarias. En contraste, las capacidades autónomas de autogestión en sistemas informáticos realizan tareas que los ingenieros eligen delegar a la tecnología de acuerdo a unas condiciones o políticas. Un sistema autoadaptativo, en lugar de uno rígido, determina los tipos de decisiones y acciones que realizan las capacidades autónomas.

Desarrollar sistemas como estos tendrá mucho interés en el futuro conectado que nos viene, porque de alguna manera podremos desplegarlos para que cuando algún servicio falle, algún servidor no responda u ocurra cualquier error, la propia infraestructura pueda corregirlo y no dependa del factor humano.

1.1 Motivación

Los sistemas autoadaptativos son capaces de modificar su comportamiento en tiempo de ejecución para alcanzar los objetivos del sistema. Circunstancias impredecibles, como cambios en el entorno del sistema, fallas del sistema, nuevos requisitos y cambios en la prioridad de los requisitos, son algunas de las razones para desencadenar acciones de adaptación en un sistema autoadaptativo. Para hacer frente a estas incertidumbres, un sistema autoadaptativo se supervisa continuamente, recopila datos y los analiza para decidir si se requiere adaptación.

Una de las soluciones es utilizar bucles de control. El aspecto desafiante de diseñar e implementar un sistema autoadaptativo es que no sólo el sistema debe aplicar cambios en el tiempo de ejecución, sino que también deben cumplir con los requisitos del sistema hasta un nivel satisfactorio [7][8][9].

El método PROTeus que está siendo desarrollado por el grupo TaTami del Centro de Investigación PROS propone un método para el desarrollo de software con capacidades de computación autónoma mediante bucles de control, aplicando soluciones de modelado y generación de código y usando modelos en tiempo de ejecución – *models-at-runtime* – . Estos sistemas se diseñan de manera que tanto los modelos del sistema, como los componentes que lo implementan se desarrollan a la par usando un *framework* de implementación – que propone el propio método – para construir estos sistemas auto-adaptativos.

Para soportar un mayor grado de dinamismo, se requiere extender este *framework* y herramienta para que los diferentes artefactos que constituyen la solución – modelos y componentes software – puedan ser gestionados en tiempo de ejecución como un servicio de repositorio de estos elementos. Estos servicios podrán ser usados tanto en fase de diseño de los sistemas, como en tiempo de ejecución, haciendo incluso posibles acciones de adaptación dinámica y evolución de los sistemas desarrollados.

1.2 Objetivos

Actualmente en esta propuesta – PROTeus – no existen servicios que faciliten el desarrollo, y la herramienta proporciona una funcionalidad 'encapsulada y acoplada' que no permite una reutilización, ni facilita el despliegue de las soluciones, de ahí la necesidad de la creación de estos repositorios.

En este trabajo vamos a desarrollar e implementar un servicio de repositorio de modelos de componente y otro de componentes software como un servicio *REST*ful. Este repositorio formará parte de una arquitectura MAPE-K, que se utilizará para la gestión de la computación autónoma mediante bucles de control. El repositorio estará dentro de K (Knowledge) que se encarga de gestionar los datos compartidos entre los otros componentes MAPE.

Para implementar los repositorios vamos a crear un servicio *REST*ful, que son servicios web que se ajustan al estilo arquitectónico *REST* y que proporcionan interoperatividad entre los sistemas informáticos en Internet. Los servicios web *REST*ful permiten a los sistemas solicitantes acceder y manipular representaciones textuales de recursos web mediante el uso de un conjunto uniforme y predefinido de operaciones sin estado [2][3].

Para realizarlo crearemos el repositorio, el servicio web *REST* y la aplicación cliente como *plugins* mediante *OSGi* de *Java*. Para crear la funcionalidad del servicio *REST*ful emplearemos *Restlet* que es un *framework* en *Java* específico para servicios *REST*.

1.3 Metodología plan de trabajo

Vamos a mostrar el plan de trabajo que tenemos previsto. En primer lugar realizaremos una aproximación a la solución creando la estructura de directorios en la misma máquina donde se ejecuta la aplicación. Posteriormente implementaremos el servicio *REST*ful basándonos en la estructura anteriormente creada.

Comenzaremos por mostrar y comentar las herramientas que vamos a utilizar para el desarrollo del trabajo. Realizaremos una breve descripción de las mismas, señalando la función que desempeñaran cada una dentro del proyecto.

Para poner el proyecto en perspectiva y poder determinar su contexto haremos una introducción teórica donde veremos los fundamentos de la computación autónoma basada en bucles de control. Una vez hayamos montado el escenario de trabajo pondremos en contexto el proyecto.

Seguidamente realizaremos el análisis del problema y estableceremos los requisitos. Estudiaremos las acciones que los repositorios han de ejecutar. Veremos como estructurar los directorios que albergarán los archivos y qué funcionalidad deben aportar estos repositorios. Finalmente analizaremos cómo se van a comunicar los diferentes elementos de nuestra solución.

Llegados a este punto pasaremos a realizar el diseño de la solución, este diseño se hará desde un punto de vista modular, es decir nos centraremos en la estructura por módulos y cómo se relacionan entre ellos. Diseñaremos una primera aproximación donde el repositorio es local, en un diseño final se añadirá el servicio *REST*ful. Para visualizarlo se utilizarán diagramas basados en UML.

Para continuar pasaremos a implementar la solución siguiendo el guión del diseño, la implementación se divide en dos fases, una primera con el repositorio en local y la final con el repositorio en remoto. Después de cada implementación se harán las pruebas correspondientes. Se mostrarán las clases más importantes en tablas y los paquetes resultantes en diagramas basados en UML.

La implementación final se probará dentro de la herramienta PROTeus tool.

Finalmente comentaremos aquellas cuestiones o problemas que han sido de especial relevancia, como son el establecimiento del protocolo *TLS* y el envío y recepción de archivos ejecutables .

A continuación vamos a realizar una breve cronología que mostraremos en un diagrama de Gantt.

Tarea 1 **(T 1)** → Entrevista con el tutor y estudio del problema.

Tarea 2 **(T 2)** → Análisis de requisitos.

Tarea 3 **(T 3)** → Elección de herramientas.

Tarea 4 **(T 4)** → Diseño de la primera aproximación. Repositorio local.

Tarea 5 **(T 5)** → Implementación de la primera aproximación.

Tarea 6 **(T 6)** → Test de funcionamiento. Depuración.

Tarea 7 **(T 7)** → Diseño final.

Tarea 8 **(T 8)** → Implementación final.

Tarea 9 **(T 9)** → Test de funcionamiento. Depuración.

Tarea 10 **(T 10)** → Documentación

Tarea 11 **(T 11)** → Entrega

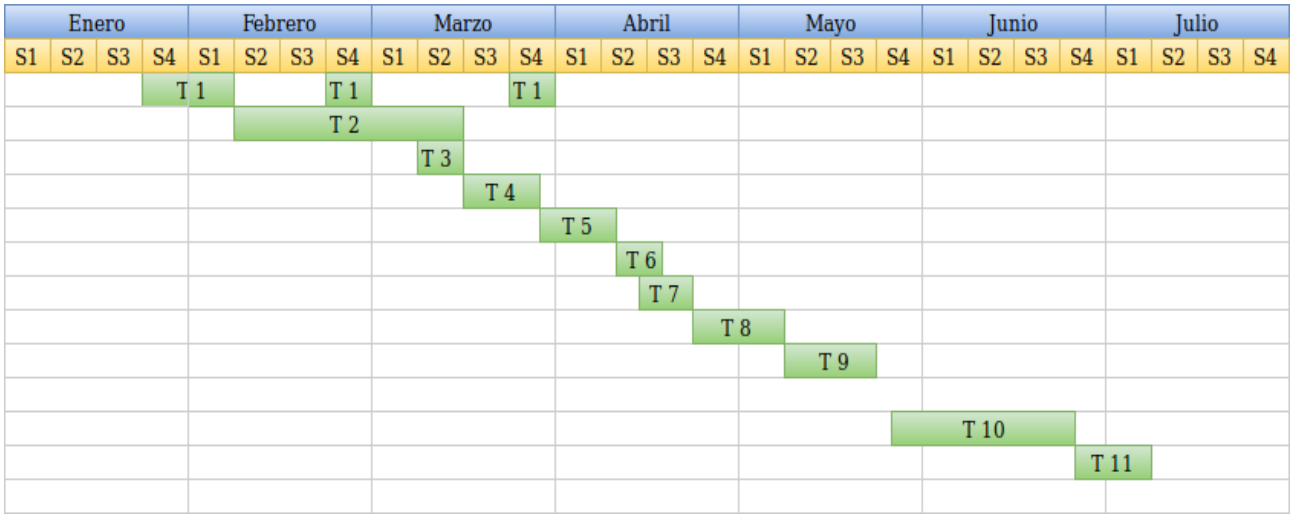


Diagrama de Gantt resultante.

2. Contexto tecnológico

En este capítulo vamos a mostrar las herramientas y entornos que emplearemos para el desarrollo del trabajo. Todo el software que se emplea es *open source software o* software de código abierto, es decir, aquel cuyo código fuente y otros derechos que normalmente son exclusivos para quienes poseen los derechos de autor, son publicados bajo una licencia de código abierto o forman parte del dominio público, con lo que no necesitamos de licencias del fabricante para su uso y además su descarga es gratuita. Usaremos como herramientas *OSGi* que nos permitirá crear una estructura modular, *Restlet framework* con la que conectaremos los repositorios a la red y así desarrollar el servicio web. Como entornos de desarrollo hemos elegido *Eclipse,* haciendo uso de la extensión para la creación de *plugins* y finalmente para simular un entorno real usaremos *VirtualBox.*

2.1 Herramientas

2.1.1 *OSGi*

OSGi es el sistema de módulos dinámicos para la plataforma *Java*. Es una especificación estándar desarrollada por un consorcio de proveedores de la industria y administrada por *OSGi Alliance*. Durante años, la tecnología *OSGi* ha florecido en el mercado de sistemas integrados y dispositivos en red. Hasta hace poco, seguía siendo una tecnología relativamente oscura para los desarrolladores, debido a su poco uso y a la imposibilidad de contrastar resultados. Hoy en día, *OSGi* está emergiendo como una tecnología viable y valiosa en la investigación y en la empresa.

El *framework OSGi* define una unidad modular llamada *bundle*. Un *bundle* es un archivo *JAR* de *Java*; un archivo *JAR* es un paquete *OSGi* válido, si contiene recursos que proporcionan funcionalidad y un adjunto – archivo *Manifest* – que contiene metadatos sobre el paquete. Los metadatos se definen utilizando pares clave-valor requeridos y opcionales; además pueden especificar un número de versión y un punto de entrada para la ejecución.

El pensamiento tradicional se centra en el desarrollo de aplicaciones web que requieren de otras aplicaciones para su funcionamiento, y los equipos dedican recursos valiosos a la identificación de la funcionalidad contenida en las aplicaciones requeridas y además el gasto se incrementa con la propia comunicación entre ellas. Con *OSGi* se desarrollan *bundle*s que se ensamblan en una aplicación, por lo que su filosofía de desarrollo pasa de centrada en la aplicación a centrada en el módulo. *OSGi* aplica el diseño de software modular a través de estos *bundle*s y dependencias administradas .

OSGi permite implementar *bundle*s individualmente, sin realizar un reinicio. Cuando la funcionalidad dentro de un *bundle* cambia, simplemente se puede volver a desplegar en la plataforma *OSGi* apropiada, y los clientes de ese *bundle* lo descubrirán automáticamente a través de los paquetes que exporta y su versión. No hay necesidad de volver a desplegar aplicaciones completas. Esto ofrece una gran flexibilidad al separar la implementación de la especificación en sus paquetes de aplicaciones [4].

Utilizando *OSGi* vamos a implementar el repositorio, el servicio *REST*ful y una aplicación cliente. Al hacerlo así, conseguiremos dar modularidad a nuestra propuesta y facilitará la integración en la infraestructura de la solución PROTeus.

2.1.2 Restlet

Restlet framework es un proyecto *open source software* para desarrolladores de *Java* que facilita el aprovechamiento de *REST* y la web en *Java*. Proporciona un extenso conjunto de clases y métodos a los que se puede llamar o extender, lo que evita tener que escribir una gran cantidad de código – p.e. usando librerías *Java* como HttpURLConnection o utilizando *servlets* – , y permite concentrarse en los requisitos del dominio. *Restlet* usa el amplio conjunto de funciones *HTTP*, como la negociación de contenido, el almacenamiento en caché, el procesamiento condicional y la autenticación segura. Puede usarse desde la web clásica a la web semántica, desde servicios web a *rich web clients* y sitios web, desde la web móvil a la computación en la nube [5].

Con *Restlet* implementaremos la funcionalidad del servicio *REST*ful, lo cual nos ahorrará muchas lineas de código. Además al ser un *framework* específico, una vez escrito el código, la lectura y comprensión del mismo será mas sencilla.

Se va a crear un servidor (repositorio) y un cliente (aplicación).

2.2 Entornos de desarrollo

2.2.1 Eclipse

Como entorno principal que nos permita desarrollar el código y realizar las pruebas de funcionamiento utilizaremos Eclipse en su última versión *Eclipse 2019-03-R.* Como vamos a basarnos en el sistema *OSGi* utilizaremos la extensión ‘Plug-in Development Environment (PDE)' que es un entorno específico para el desarrollo y ejecución de *plugins*. Gracias a la posibilidad de utilizar múltiples paquetes con diferente funcionalidad, podremos integrar en nuestro proyecto un servidor web, que será el que utilicemos en nuestro repositorio remoto.

2.2.3 VirtualBox

Virtualbox permite que un sistema operativo no modificado con todo su software instalado se ejecute en un entorno especial, además de su sistema operativo existente. Este entorno, denominado máquina virtual, es creado por el software de virtualización al interceptar el acceso a ciertos componentes de hardware y ciertas características. La computadora física generalmente se denomina "host", mientras que la máquina virtual a menudo se denomina "guest". La mayoría del código de invitado se ejecuta sin modificaciones, directamente en la computadora *host*, y el sistema operativo invitado "piensa" que se está ejecutando en una máquina real [6].

Con esta herramienta emularemos un escenario real, en el cuál la aplicación y el repositorio se encuentran en máquinas diferentes. Esto nos será de gran ayuda en el diseño e implementación del servicio *REST*ful.

3. Caso de estudio

3.1 Introducción teórica

Un sistema de computación autónoma es un sistema que detecta su entorno operativo, modela su comportamiento en ese entorno y toma medidas para adaptarse a este. Un sistema informático autónomo tiene las propiedades de autoconfiguración, autocorrección, autooptimización y autoprotección [7][8].

Para cumplir estas propiedades, se plantea la figura del *autonomic manager* (administrador autónomo) que es un componente que administra otros componentes de software o hardware mediante un bucle de control (*figura 1*) y que se encuentra por encima de los sistemas que no son autoadaptativos (o autónomos). El bucle de control del *autonomic manager* incluye funciones de monitorización, análisis, planificación y ejecución .

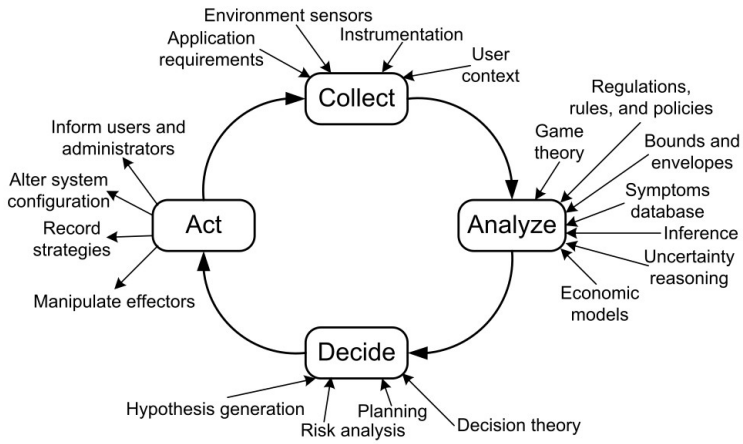


figura 1. Bucles de control en un autonomic manager.

3.1.1 MAPE-K

Un gran avance en hacer explícitos los bucles de control vino en 2005 con la iniciativa de computación autónoma de IBM mostrando su énfasis en los sistemas de autogestión. Uno de los hallazgos clave de esta iniciativa de investigación es el plan de acción para construir sistemas autónomos utilizando *MAPE-K control loops* [7][8][9][10]. Tiene este nombre por las fases principales: Monitor, Analizador, Planificador y Ejecutor, la K viene de Knowledge – conocimiento – .

A continuación haremos una breve descripción de los componentes:

3.1.1.1 Monitor

La función de monitorización recopila los detalles de los recursos administrados, a través conectores o sensores, y los relaciona con los datos que se pueden analizar. Los detalles pueden incluir información de topología, métricas, configuraciones de componentes, etc. Estos datos incluyen información sobre la configuración de los recursos administrados, el estado, la capacidad ofrecida y el rendimiento.

Algunos de los datos son estáticos o cambian lentamente, mientras que otros son dinámicos y cambian continuamente a lo largo del tiempo. La función de monitorización agrega, correlaciona y filtra estos detalles hasta que determina un conjunto de datos que debe analizarse.

3.1.1.2 Analizador

La función de análisis proporciona los mecanismos para observar y analizar situaciones para determinar si es necesario realizar algún cambio. Por ejemplo, el requisito para ejecutar un cambio puede ocurrir cuando la función de análisis determina que no se cumple alguna política. La función de análisis es responsable de determinar si el *autonomic manager* puede cumplir con la política establecida, ahora y en el futuro. En muchos casos, la función de análisis modela el comportamiento complejo, por lo que puede emplear técnicas de predicción como el pronóstico de series de tiempo y los modelos de colas. Estos mecanismos permiten al *autonomic manager* aprender sobre el entorno operativo y le ayudan a predecir el comportamiento futuro.

Los *autonomic manager* deben poder realizar análisis de datos complejos y razonar sobre las incidencias proporcionadas por la función de monitor. Si se requieren cambios, la función de análisis genera una solicitud de cambio y, lógicamente, pasa esa solicitud de cambio a la función de plan. La solicitud de cambio describe las modificaciones que el componente de análisis considera necesarias o convenientes.

El análisis está influenciado por los datos de conocimiento almacenados (K) .

3.1.1.3 Planificador

La función de plan crea o selecciona un procedimiento para ejecutar una alteración deseada en el recurso administrado. La función de plan puede tomar muchas formas, desde un solo comando hasta un flujo de trabajo complejo.

El planificador puede requerir de conocimientos (K) para construir la respuesta.

3.1.1.4 Ejecutor

La función de ejecución proporciona el mecanismo para programar y realizar los cambios necesarios en el sistema. Una vez que un *autonomic manager* ha generado un plan de cambio que corresponde a una solicitud de cambio, es posible que se deban tomar algunas medidas para modificar el estado de uno o más recursos administrados. La función de ejecución de un *autonomic manager* es responsable de llevar a cabo el procedimiento generado por la función de plan a través de una serie de acciones. Parte de la ejecución del plan de cambios podría incluir la actualización de los conocimientos (K) que utiliza el *autonomic manager*.

Esta función es interesante ya que hará uso del repositorio de componentes software que desarrollaremos.

3.1.1.5 Base de conocimiento K

Esta función es la más interesante para nuestro trabajo ya que el repositorio de modelos de componentes que vamos a desarrollar será pieza fundamental de la misma.

Una base de conocimiento K es una implementación de un registro, diccionario, base de datos u otro repositorio que proporciona acceso al conocimiento de acuerdo con las interfaces prescritas por la arquitectura. En un sistema autónomo, el conocimiento consiste en tipos particulares de datos con sintaxis y semántica arquitectónicas, tales como incidencias, políticas, solicitudes de cambio y planes de cambio. Estos datos se pueden almacenar en una base de conocimiento para que se pueda compartir entre los componentes del *autonomic manager*.

Los datos almacenados se pueden utilizar para ampliar las capacidades de conocimiento de un *autonomic manager*, el cual puede cargar datos de una o más bases de conocimiento, y el administrador del *autonomic manager* puede activar ese conocimiento, lo que permite realizar tareas de administración adicionales (como reconocer incidencias específicas o aplicar ciertas políticas) [7][10].

De las distintas formas para obtener la base de conocimiento, nuestro caso se produce cuando es el propio *autonomic manager* el que crea el conocimiento.

- El **conocimiento** utilizado por un *autonomic manager* en particular podría ser creado por la parte del **monitor**, según la información recopilada a través de sensores. La parte del monitor puede crear conocimiento basado en actividades recientes al registrar las notificaciones que recibe de un recurso administrado.
- La parte de **ejecución** de un *autonomic manager* puede actualizar el conocimiento para indicar las acciones que se tomaron como resultado del **análisis** y la **planificación** según los datos monitorizados , la parte de ejecución indicaría cómo esas acciones afectaron el recurso administrado – basado en los datos subsiguientes monitorizados – .

Esta base de conocimiento está contenida dentro del *autonomic manager*, como lo representa el bloque de "Knowledge" (*figura 2*).

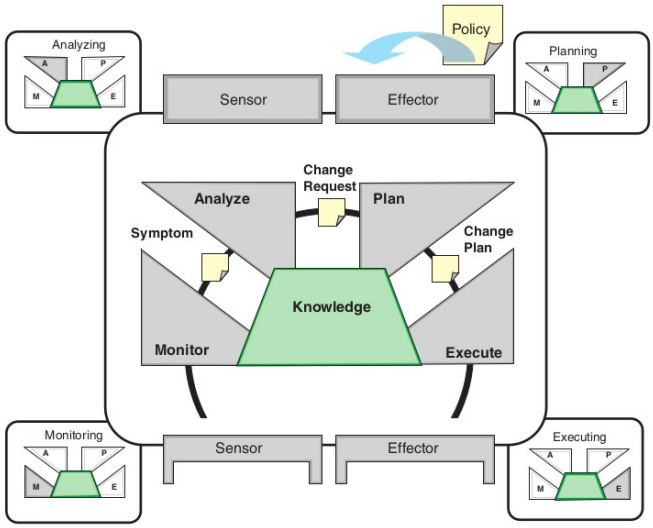


figura 2. Esquema de la estructura interna de un autonomic manager que utiliza MAPE-K.

Con el desarrollo de este trabajo se pretende añadir un elemento a esa base de conocimiento proporcionando un espacio donde los demás componentes se apoyen. El conocimiento será creado por una aplicación que contiene los componentes MAPE y nuestros prototipos serán la base donde almacenar, compartir y gestionar los modelos que se vayan creando.

Vemos un diagrama donde se ubican nuestros repositorios dentro de esta arquitectura (*figura 3*).

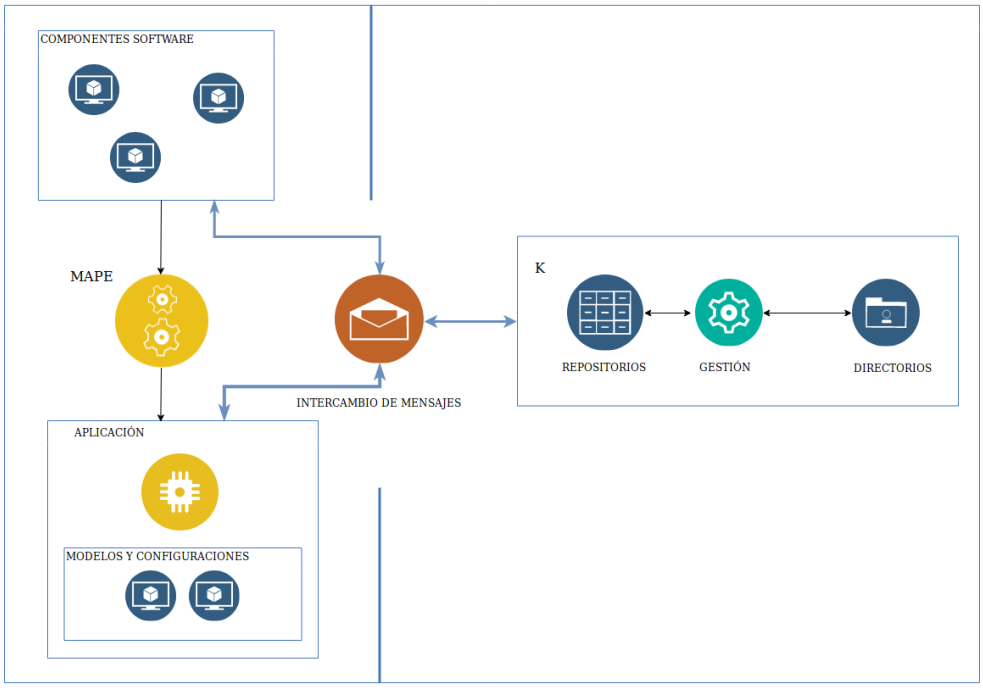


figura 3.Diagrama de situación de los repositorios en MAPE-K.

3.2 PROTeus Tool/fw

El método PROTeus dispone de una herramienta (PROTeus Tool) que permite configurar y desarrollar soluciones autoadaptativas empleando bucles de control que se superponen sobre sistemas que no son adaptativos. Para hacerlo, la herramienta proporciona un *framework* de implementación (PROTeus fw) con la que desarrollar desde cero sistemas preparados para ser adaptados, o reconvertir sistemas existentes en 'preparados para ser adaptados'. La herramienta también permite configurar los bucles de adaptación que permiten emplear sistemas preparados para ser adaptados, y hacerlos así autoadaptativos. Todo esto lo hace trabajando sobre modelos que describen los componentes del sistema.

Para que todo esto funcione, el sistema necesita en tiempo de ejecución consultar y manipular continuamente los modelos que forman el sistema (*ver 3.1.1*). La ingeniería de tales sistemas suele ser difícil, ya que el conocimiento disponible en el momento del diseño no es adecuado para anticipar todas las condiciones a las que el sistema se enfrentará cuando se esté ejecutando. Por lo tanto, esta incertidumbre es tratada en tiempo de ejecución, cuando hay más conocimiento disponible. La solución, entre otras cosas, emplea 'modelos-at-runtime' – modelos en tiempo de ejecución – .

Actualmente esa funcionalidad ya está desarrollada a través de un servicio 'local'. La base de conocimiento K almacena internamente los modelos debido a que no existe el repositorio, lo que hace que esté demasiado encapsulado y que no se puedan utilizar los modelos en varias soluciones.

Por ejemplo, si varias aplicaciones funcionan con los mismos modelos de referencia, cada una de ellas tiene internamente empotrada una copia de esos modelos en su K. ¿Qué ocurre? Que si se requieren hacer cambios a los modelos, se ha de hacer para cada aplicación expresamente e incluso recompilar las bases de conocimiento de cada aplicación para que sean ‘conscientes’ de los nuevos modelos.

En la propuesta PROTeus se han tomado dos decisiones:

- Se emplean modelos de componente como herramienta principal para funcionar internamente. Estos modelos serán compartidos a través del repositorio de modelos de componente y configuraciones.
- Se emplea una aproximación en la que se conciben los sistemas en base a componentes software los cuales se reconvierten/refactorizan en componentes adaptables – *Adaptive-Ready components* – . Los componentes serán compartidos mediante el repositorio de componentes software.

La duda que se plantea es cómo mover esos modelos a un espacio fuera de la aplicación donde se puedan gestionar globalmente. Una primera aproximación en tiempo de diseño es crear una estructura de directorios en la misma máquina donde se ejecuta la aplicación. Distintas aplicaciones corriendo en el mismo sistema operativo podrán almacenar y consultar modelos dinámicamente. El administrador podrá entonces realizar cambios en una sola acción, después serán las aplicaciones las que se actualicen bajo demanda.

Como esta solución también debe funcionar en tiempo de ejecución, las aplicaciones deben ser totalmente independientes de los repositorios. Se desarrolla entonces una aproximación final donde la solución reside en extraer los modelos a un servicio accesible vía *REST* al que se le pueden aplicar operaciones *CRUD* (Create, Read, Update, Delete) en tiempo de ejecución. Así, enfatizando en la modularidad, será totalmente independiente de la aplicación.

Para mostrar un ejemplo utilizaremos un despliegue básico de un sistema autoadaptativo. En este despliegue hay una aplicación que llamaremos ‘SmartCar’ y que es el sistema a hacer autoadaptativo. El *autonomic manager* (MAPE-K) está situado en un nodo al que SmartCar se conecta, realizando este nodo todas las acciones relacionadas con sus componentes – monitoriza, analiza, planifica y ejecuta – . Para finalizar tenemos el servicio de repositorio de modelos de componente que conecta directamente con la base de conocimiento K y el servicio de repositorio de componentes software que es usado por el ejecutor del MAPE-K. En este contexto los repositorios proporcionan un mecanismo donde la K puede ser accedida por los demás componentes en tiempo de ejecución.

En el siguiente diagrama podemos ver donde se ubica nuestros repositorios dentro de esta despliegue (*figura 4*).

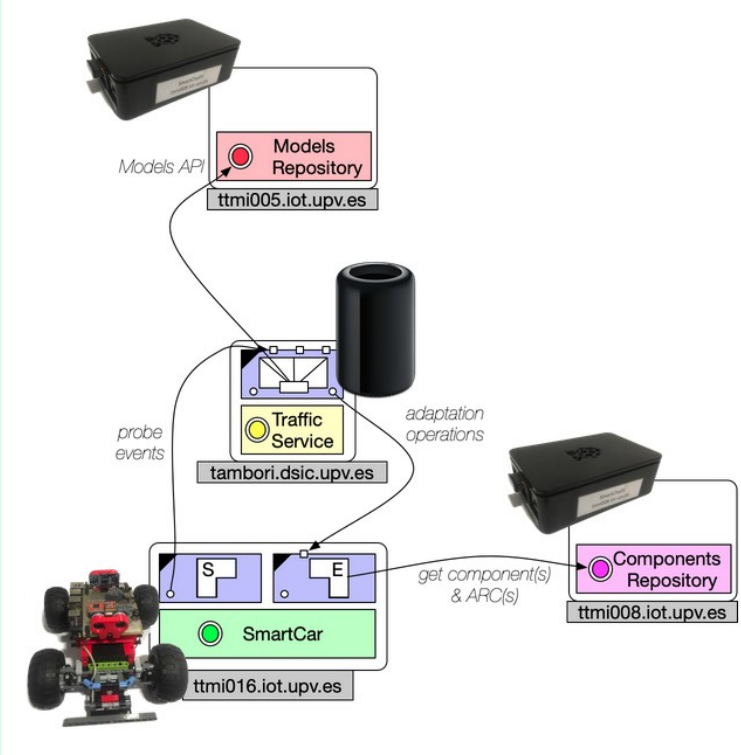


figura 4. Diagrama de un despliegue básico de un sistema autoadaptativo. Se observan la ubicación de los dos repositorios (en rojo y morado) dentro de esta arquitectura.

Después de esta introducción pasamos entonces a exponer el análisis de:

- El servicio repositorio que contendrá modelos de componentes y los modelos de configuraciones del sistema.

- El servicio repositorio de componentes software que contendrá los componentes adaptables (*Adaptive-Ready components*) .

4. Análisis del problema

Para crear los repositorios y que sirvan de soporte del desarrollo de sistemas autodaptativos – como lo es nuestro sistema SmartCar – se requiere de un análisis del problema y del diseño de las bases de la solución. Esto nos servirá de guía en la posterior fase de diseño eligiendo la función que debe realizar el software y establecer o indicar cuál es la interfaz más adecuada para el mismo.

En el contexto del problema tenemos por un lado una aplicación que irá creando y demandando modelos, configuraciones y componentes de forma dinámica, y por otro un almacén/gestor de estos artefactos. Las partes tienen que ser independientes una de otra, con lo que una opción válida será definirlos como módulos.

En tiempo de ejecución habrá un intercambio de mensajes entre ellos que habrá que manejar eficientemente. No todos los archivos intercambiados son del mismo tipo ni tienen la misma función con lo que tendrá que haber una forma de identificarlos (*figura 5*).

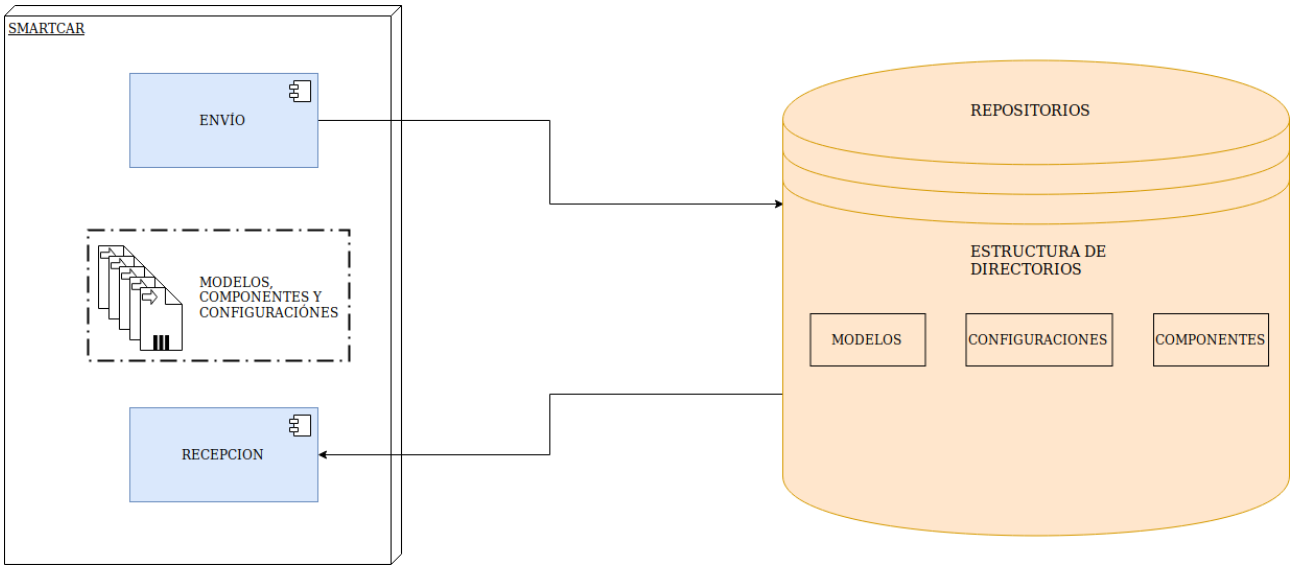


figura 5. Abstracción del funcionamiento esperado de la solución.

4.1 Funciones de los repositorios

Los repositorios deben proporcionar un mecanismo para aplicar operaciones *CRUD* sobre los archivos que la aplicación vaya manejando. Como objetivo principal está el resolver cómo guardar estos modelos y componentes de forma que se realice un almacenado ordenado. Este orden en el guardado proporcionará una búsqueda fácil y rápida. Así mismo deben identificarse los archivos unívocamente.

4.1.1 Estructura definida de directorios. Requisitos.

Repositorio de modelos y configuraciones

Vamos a definir ahora los requisitos necesarios para cumplir el objetivo:

- La aplicación va a trabajar con dos tipos de archivos, uno de modelos de componente y otro de modelos de configuraciones del sistema. El formato de los archivos es *XML*.
- Cada uno de los modelos de componente podrá tener varias versiones, las configuraciones del sistema podrán igualmente tener versiones.
- Cada uno de los modelos de configuraciones del sistema lleva asociada una versión de modelo de componente.
- Deberá proporcionar un modo de identificar los archivos unívocamente.

Será necesario crear un registro de entrada en el cual se almacenen algunos de los metadatos del archivo y facilitar así la posterior búsqueda.

Modelo conceptual

A continuación se muestra un diagrama de cómo deberá ser la estructura directorios en el repositorio de modelos de componente y modelos de configuraciones del sistema (*figura 6*) .

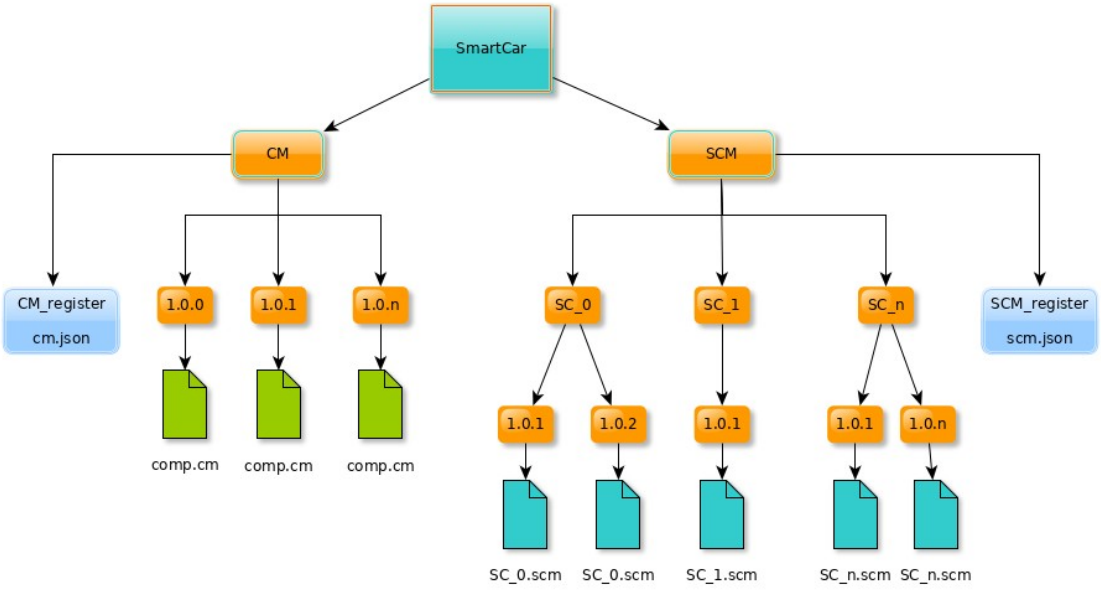


figura 6. Estructura directorios en el repositorio de modelos de componente (CM) y modelos de configuraciones del sistema (SCM).

Repositorio de componentes software

Definimos los requisitos de la solución :

- La aplicación va a trabajar con un sólo tipo de archivo. Estos archivos son binarios/ejecutables.
- Cada uno de los componentes software podrá tener varias versiones.
- Deberá proporcionar un modo de identificar los archivos unívocamente.

Se va a crear un registro de entrada en el cual se almacenen algunos de los metadatos del archivo facilitando el control y la consulta.

Modelo conceptual

A continuación se muestra un diagrama de cómo deberá ser la estructura directorios para el repositorio de componentes software (*figura 7).*

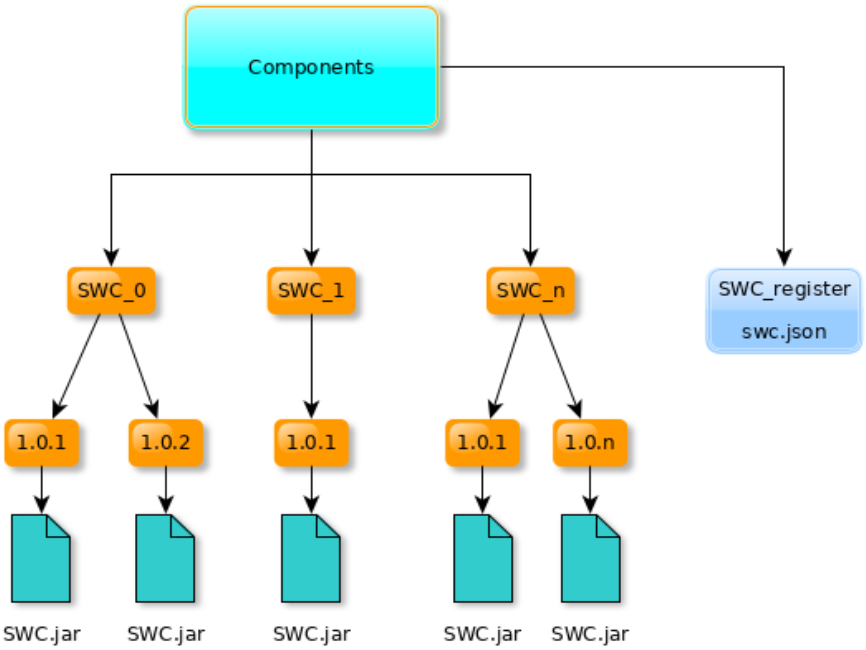


figura 7. Estructura de directorios para el repositorio de componentes software (SWC).

4.1.2 Funcionalidad de los repositorios

Ahora hay que determinar que funciones debe proporcionar el repositorio para cumplir los objetivos – operaciones *CRUD* – .

Los requisitos funcionales del repositorio serán:

- Deberá almacenar los archivos enviados por la aplicación siguiendo la estructura definida de directorios. Esto lo hará para modelos de componente, configuraciones del sistema y componentes software.
- Deberá proporcionar un mecanismo de respuesta a una petición de archivo almacenado.
- Deberá ser capaz de realizar actualizaciones basadas en su versión, de cualquier archivo almacenado.
- Deberá tener la capacidad de eliminar cualquier archivo que se le solicite.
- No se almacenarán archivos que hayan sido actualizados, una vez se produzca la actualización, el archivo obsoleto será eliminado.
- Si por alguna circunstancia, existe un modelo de configuración en la que el modelo de componente asociado ha sido borrado, no se reemplazará.
- En todas las operaciones, a excepción de la de lectura, deberá devolver un mensaje del estado de la petición.

Para cumplir con estos requisitos deberemos usar las clases que ofrece *Java* tanto para el tratamiento de archivos, como de navegación por el sistema de archivos del sistema operativo donde se ejecuta.

En este diagrama se muestra como debería actuar el repositorio:

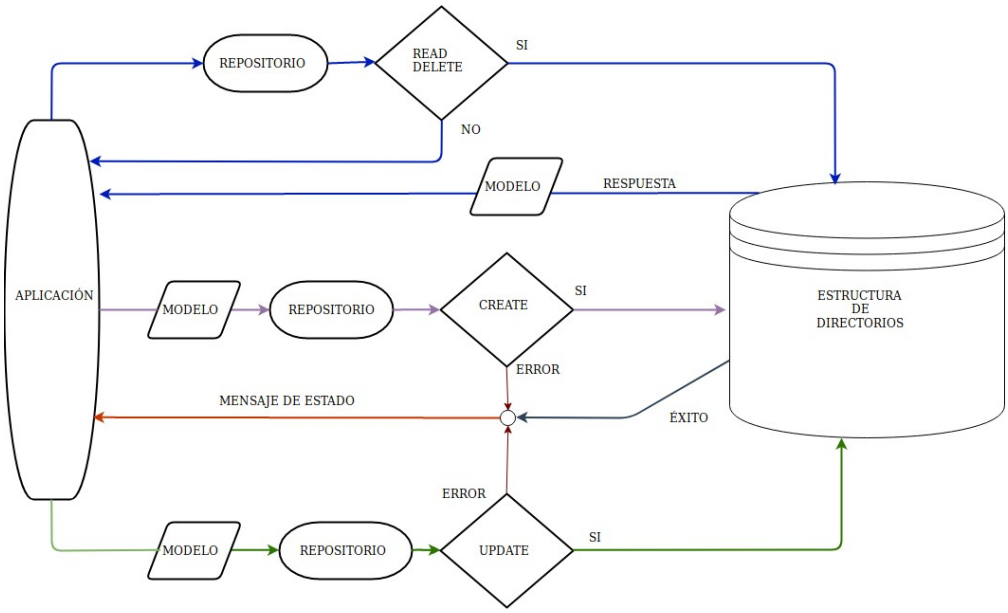


Diagrama de flujo de las acciones que se realizarán durante el funcionamiento.

4.2 Comunicaciones

Para conseguir independencia entre el repositorio y la aplicación necesitamos que estas partes se comuniquen de una forma lo más homogénea y acotada posible. Las partes pues, deben disponer de una interfaz en el que los métodos estén bien definidos y sean utilizados para la misma función. Además hay que potenciar la modularidad en la solución proporcionando así flexibilidad y un mantenimiento más sencillo.

El sistema *OSGi* proporciona los mecanismos necesarios para afrontar este problema. Por medio de interfaces y su servicio de registros podremos conseguir el objetivo.

Además nuestra propuesta es crear el repositorio como un servicio web, y concretamente utilizando *REST*. Habrá pues que definir los recursos y adecuarlos a la estructura de directorios que hayamos creado. Este servicio será una capa de software que se superpondrá al repositorio, será entonces este servicio web el que esté en comunicación con la aplicación. Si las operaciones a realizar van a ser crear, leer, actualizar y borrar, los métodos a utilizar deberán ser GET, PUT/POST, UPDATE Y DELETE [13]. Aquí entrará en juego *Restlet framework* y los beneficios que reporta al poder crear un cliente y un servidor con el mismo *framework.*

En una secuencia tipo la aplicación realiza una petición al servicio, este utiliza la interfaz que implementa para comunicarse con el repositorio. El repositorio realiza las acciones correspondientes a la petición, posteriormente y según lo demandado dará una contestación al servicio web. Este último utilizando la misma conexión abierta anteriormente devolverá una respuesta (*figura 8*).

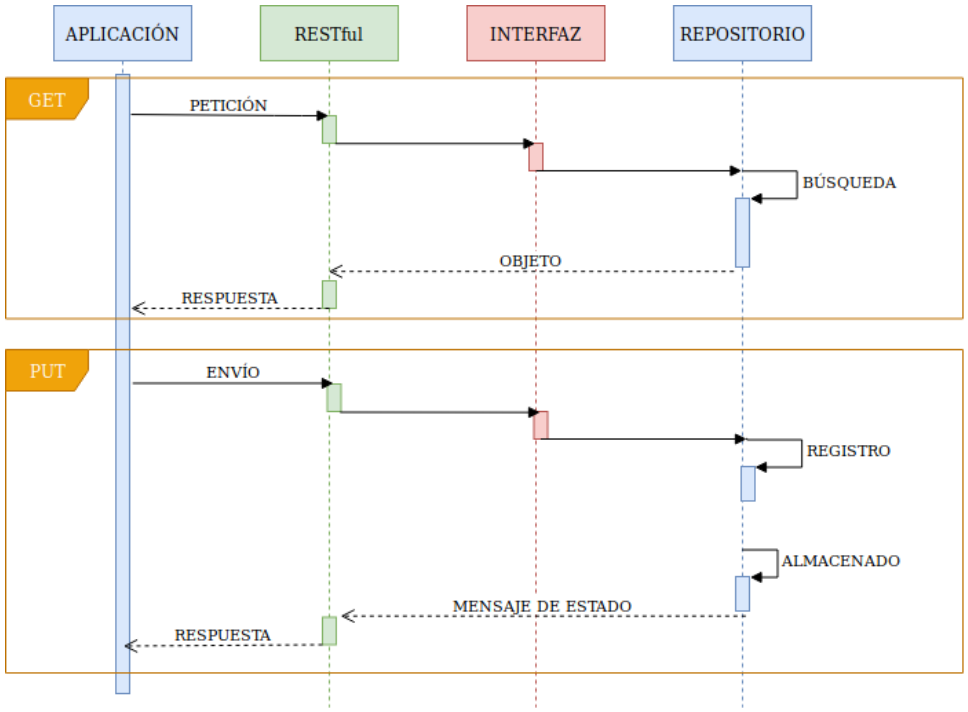


figura 8. Diagrama de secuencia de las operaciones GET y PUT.

5. Diseño de la solución

Iniciaremos el diseño con la estructura de directorios y el nombrado de archivos que residirán en los repositorios. Para continuar nos centraremos en la gestión de los repositorios, justificando el uso de las interfaces, y diseñando el registro de entrada y formato de los mensajes a intercambiar. También definiremos la interfaz del servicio *REST* que servirá para localizar los recursos. Seguidamente realizaremos una primera aproximación creando el repositorio en la misma máquina donde se ejecuta la aplicación – SmartCar – . En esta aproximación sólo se va a diseñar el repositorio de modelos de componente y configuraciones del sistema. Esto sentará las bases para la obtención de la solución final. Para concluir realizaremos el diseño definitivo.

5.1 Estructura definida de directorios

La primera fase para atacar el problema es diseñar la estructura de directorios que tendrá nuestros repositorios. Esta parte resulta clave porque posteriormente será la base sobre la que construyamos las peticiones al servicio *REST*ful.

5.1.1 Repositorio de modelos de componente y configuraciones

A continuación definimos la estructura que tendrá el repositorio de modelos:

Carpeta raíz → Será nombrada con el nombre de la aplicación.

- **Rama 1** → Contendrá a los modelos de componente.
  - **Hojas** → Cada una de las versiones
- **Registro** → Registro de entrada de modelos de componente
- **Rama 2** → Contendrá a los modelos de configuraciones del sistema.
  - **Rama 2.n** → Cada una de las configuraciones del sistema para componentes.
    - **Hojas** → Cada una de las versiones.
- **Registro** → Registro de entrada de modelos de configuraciones del sistema.

Nombrado de archivos

Los modelos de componente y los modelos de configuraciones del sistema están definidos en archivos *XML* y el registro de entrada se creará como un archivo *JSON*.

Para identificar unívocamente estos tres tipos de archivos seguiremos la siguiente política:

- Los modelos de componente – en inglés Component Model – se identificarán por su acrónimo en inglés, esto es “CM”. La extensión de estos archivos será por tanto “.cm” .
- Los modelos de configuraciones del sistema – en inglés System Configuration Model – se identificarán por su acrónimo en inglés, esto es “SCM”. La extensión de estos archivos será “.scm” .

- El registro de entrada seguirá con la identificación estándar para archivos *JSON*. La extensión de estos archivos será ".json" .

Los nombres de los archivos tanto de modelos de componente como de modelos de configuraciones, serán proporcionados por la propia aplicación, así mismo se proporcionará la versión de estos.

5.1.2 Repositorio de componentes software

A continuación realizamos el diseño de la estructura de directorios para el repositorio de componentes software. Como se ha mencionado en *5.1* esta parte resulta fundamental para el desarrollo de la solución.

Esta estructura resultará más simple ya que sólo trabaja con un tipo de archivo.

Carpeta raíz → Será nombrada con el nombre de "Components".

Rama 1 → Contendrá al componente software.
Hojas → Cada una de las versiones.

Rama 2 → Contendrá al componente software.
Hojas → Cada una de las versiones.

•
•

Rama N → Contendrá al componente software.
Hojas → Cada una de las versiones.

Registro → Registro de entrada de los componentes software.

Nombrado de archivos

Los componentes software son archivos ejecutables, habitualmente son archivos *JAR*/*bundle*s de *OSGi* (*ver 2.1.1*) y el registro de entrada se creará como un archivo *JSON*.

Para identificar unívocamente este tipo de archivos seguiremos la siguiente política:

- Los componentes software – en inglés Software Component – se identificarán por su acrónimo en inglés, esto es "SWC". La extensión de estos archivos será la identificación estándar para archivos *JAR* ".jar" .
- El registro de entrada seguirá con la identificación estándar para archivos *JSON*. La extensión de estos archivos será ".json" .

Los nombres de los archivos – la identificación del componente software – serán proporcionados por la propia aplicación, así mismo se proporcionará la versión de éstos.

5.2 Gestión de los repositorios

5.2.1 Interfaces

Las recomendaciones en *OSGi* animan a utilizar patrones de usabilidad. Es fácil crear y administrar las dependencias de los módulos lanzando todas las clases en un par de archivos *JAR*, pero al hacerlo, habremos creado un problema de mantenimiento.

Uno de los principales patrones de usabilidad es incluir una interfaz publicada. La interfaz publicada de un módulo debe ser bien conocida y además debe constar de los métodos públicos que contienen las clases que se encuentran en los paquetes exportados y que otros módulos pueden invocar.

Usando *Java* estándar, la manera más fácil de evitar que las clases externas accedan a una clase o método que no se desea, es usar interfaces. La interfaz define los métodos que se desea exponer, mientras que la implementación dentro del módulo puede definir métodos adicionales. Los usuarios del módulo deben interactuar con la interfaz, no con la implementación [4]. Estas interfaces dentro de *OSGi* sirven para registrar un servicio – el repositorio en nuestro caso – y de esta forma poder referenciarlo posteriormente, este es el mecanismo por el que distintos módulos pueden comunicarse.

Los requisitos establecen que el repositorio realizará operaciones *CRUD*. Definiremos la interfaz basándonos en estas cuatro acciones. La tabla muestra las acciones que la interfaz debe cubrir y los posibles métodos que debería implementar.

Operación	IModelRepository		IComponentRepository
	CM	SCM	SWC
Create	`setCM()`	`setSCM()`	`setSWC()`
Read	`getCM()`	`getSCM()`	`getSWC()`
Update	`updateCM()`	`updateSCM()`	`N/A`
Delete	`deleteCM()`	`deleteSCM()`	`deleteSWC()`

5.2.2 Registros de entrada

Para facilitar el control y la búsqueda de archivos se van a crear registros de entrada en formato *JSON*. Esta es su estructura:

CM_register.*JSON* :

```
[     {"App_name":"Smart_Car"},
{"CM_version":"1.0.0","CM_id":1,"Timestamp":"20190519_130209"},
{"CM_version":"1.0.1","CM_id":2,"Timestamp":"20190519_130210"},
{"Counter":3},
{"CM_version":"1.0.3","CM_id":3,"Timestamp":"20190519_130212"}    ]
```

SCM_register.*JSON* :

```
[      {"App_name":"Smart_Car"},
{"CM_version":"1.0.0","SCM_id":1,"SCM_name":"SC0_InitialSystemConfiguration","Timestamp":
"20190519_130212","SCM_version":"1.0.1"},
{"CM_version":"1.0.0","SCM_id":2,"SCM_name":"SC1_OnStdContextSystemConfiguration","Timest
amp":"20190519_130213","SCM_version":"1.0.1"},
{"Counter":3},
{"CM_version":"1.0.3","SCM_id":3,"SCM_name":"SC5_OnInClassroomEnvironmentSystemConfigurat
ion","Timestamp":"20190519_130215","SCM_version":"1.0.1"} ]
```

SWC_register.*JSON* :

```
[      {"Content":"Software Components"},
{"SWC_version":"1.0.0","Comp_id":"AFI.interfaces","SWC_id":1,"Timestamp":"20190519_182802
"},
{"SWC_version":"1.0.1","Comp_id":"AFI.Automovil","SWC_id":2,"Timestamp":"20190519_182803"
},
{"SWC_version":"1.0.1","Comp_id":"AFI.restservice","SWC_id":3,"Timestamp":"20190519_18280
4"},
{"Counter":4},
{"SWC_version":"1.0.0","Comp_id":"AFI.repository","SWC_id":4,"Timestamp":"20190519_183802
"}      ]
```

5.2.3 Formato de los mensajes

Para el intercambio de datos entre aplicación y repositorio vamos a utilizar archivos *JSON*. Usando este formato podremos envolver los datos y enviarlos de una 'pieza'. La característica principal de este formato es que tiene una estructura clave-valor, la cual utilizaremos para definir el tipo de archivo – CM, SCM o SWC – y su contenido. Existe una restricción a la hora de usar este formato y es que el valor a adjuntar debe ser un `String` – cadena de texto – , esto no supondrá problema cuando trabajemos con modelos de componente y configuraciones del sistema debido a que son archivos *XML* y pueden manejarse tal cual. La dificultad aparecerá cuando trabajemos con binarios y haya que convertirlos a una cadena de texto.

Ejemplo de mensaje de respuesta a una petición de configuración de sistema con modelo de componente asociado:

```
[
{"CM":"<?XML version="1.0" encoding="UTF-8"?>
       <es.upv.pros.tatami.adaptation.componentsModel:ComponentsModel  xmi:version="2.0"
       xmlns:xmi="HTTP://www.omg.org/XMI"xmlns:xsi="HTTP://www.w3.org/2001/XMLSchema-
       instance"xmlns:es.upv.pros.tatami.adaptation.componentsModel="HTTP://
       es.upv.pros.tatami.adaptation.componentsModel" name="AdaptiveSmartCar"> ... "},
{"SCM":"<?XMLversion="1.0"encoding="ASCII"?>
       <es.upv.pros.tatami.adaptation.systemConfigurationModelPackage:
       SystemConfigurationModel
       xmlns:es.upv.pros.tatami.adaptation.systemConfigurationModelPackage="HTTP://
       es.upv.pros.tatami.adaptation.systemConfigurationModelPackage" name="SC_1"> ... "}

]
```

Formato de los mensajes en el repositorio de modelos y configuraciones.

Para el repositorio de componentes software se nos plantea el desafío de insertar un ejecutable dentro de un archivo *JSON* como cadena de texto, la solución de esta cuestión se trata posteriormente en un apartado dedicado (*7.2*).

5.2.4 Interfaz *REST*ful

Para desarrollar un servicio *REST*ful es necesario establecer un contrato formal – *Uniform contract* – el cual se basa en tres elementos principales [12].

- Sintaxis del identificador de recursos: Expresa hacia dónde se transfieren los datos y desde dónde provienen. Representados por URI's / URL's.
- Métodos: Mecanismos de protocolo utilizados para transferir los datos – GET, PUT, DELETE , POST, OPTIONS ... – .
- *Media type*: Formato de los datos que se están transfiriendo – text/plain, application/*XML*, application/*JSON* ... – .

Sintaxis y métodos : En la siguiente tabla se refleja la elección de las URL's y los métodos *HTTP* a utilizar. El recurso `root` se encontrará en `https://<IP del servidor>/` .

URL	GET	PUT	DELETE
`root/cmodel/{app}/{version}`	Lectura	Almacenado Actualización	Borrado
`root/scmodel/{app}/{name}/{version}`	Lectura	Almacenado Actualización	Borrado
`root/component/{component_id}/{version}`	Lectura	Almacenado Actualización	Borrado

Media type : Definición del formato de los datos que se van a manejar.

Archivos	Dónde reside	*Media type* / Formato
Modelos de componente → ".cm"	`String` en un campo *JSON*	application/json
Configuraciones de sistema → ".scm"	`String` en un campo *JSON*	application/json
Componentes software → ".jar"	Binario en un campo *JSON*	application/json

Con esto hemos concluido la parte del diseño de la base de nuestra solución. Seguidamente veremos el diseño estructural que proponemos dividido en dos partes.

5.3 Primera aproximación. Repositorio local.

5.3.1 Componentes

Para comenzar vamos a definir dos componentes en nuestra solución y que están presentes dentro una arquitectura MAPE-K, como son el *autonomic manager* (*ver 3.1*) y la base de conocimiento K (*ver 3.1.1.5*).

En esta aproximación el *autonomic manager* tendrá la forma de una aplicación la cual:

- Envía al repositorio modelos de componentes y modelos de configuraciones del sistema que este debe almacenar.
- Realiza peticiones de archivos ya guardados basándose en su versión.
- Actualiza versiones de las configuraciones del sistema.
- Borra archivos almacenados basándose en su versión.

Con estas acciones se intenta simular un escenario en el que los modelos y las configuraciones van creándose y cambiando dinámicamente en tiempo de ejecución. Hay que decir que es una simulación muy simple, sin embargo nos sirve para testar el correcto funcionamiento del repositorio.

Esta aplicación estará representada como un *bundle* en *OSGi* y contendrá las clases y métodos necesarios para cumplir con la funcionalidad demandada. Al utilizar *OSGi* vamos a conseguir portabilidad y modularidad en nuestra solución.

La base de conocimiento K tendrá la forma de repositorio y estará representado como un *bundle* en *OSGi* que contendrá las clases y métodos necesarios. Recordamos que el repositorio está en la misma máquina que la aplicación.

5.3.2 Estructura

Como se ha mencionado en *5.2.1* la interfaz nos va a proporcionar el mecanismo por el cual los diferentes módulos se van a comunicar, con lo que nos quedaría una estructura de tres módulos. La aplicación y el repositorio se comunicarán a través de la interfaz (*figura 9*).

La interfaz servirá de conector entre aplicación y repositorio, se entiende que deberán implementarla los dos módulos. La aplicación maneja los archivos a enviar y los recibidos de una manera interna, es decir tiene su propio espacio dentro del sistema de archivos.

El módulo del repositorio creará dentro del sistema de archivos la estructura de directorios que necesita para funcionar. Este se encargará de almacenar los archivos en esa estructura, así como de la recuperación desde el repositorio y envío de estos.

La comunicación es posible gracias al servicio de registros que ofrece OSGi, en el cual se ha registrado el repositorio y es la aplicación la que hace referencia a ese servicio ofrecido.

Una vez hemos concluido la primera aproximación pasaremos al diseño final donde incluiremos el servicio *REST*ful como otro módulo.

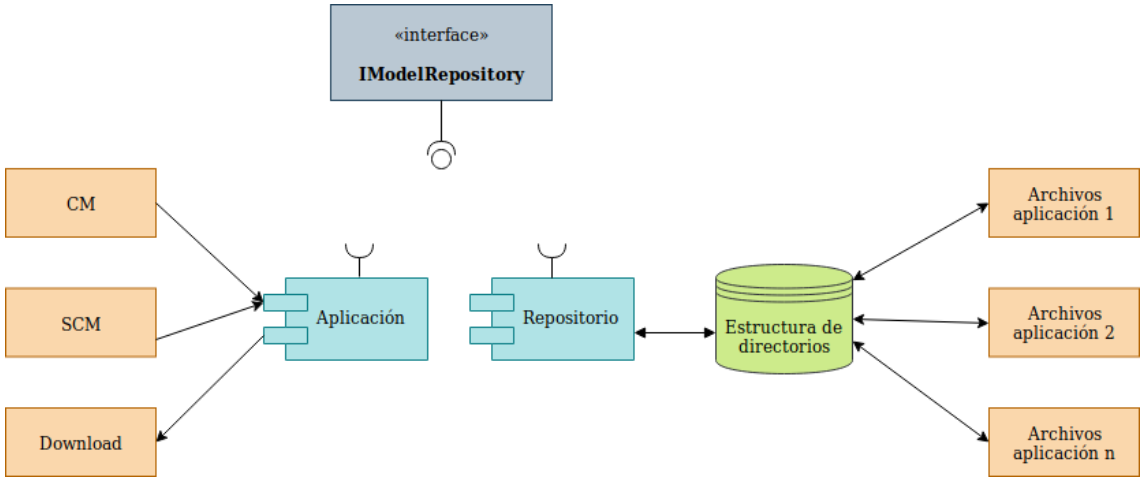


figura 9. Diagrama de la primera aproximación.

5.4 Diseño final. Repositorio remoto.

Una vez ya tenemos la estructura base diseñada, vamos a incluir el servicio *REST*ful. Este módulo implementará la interfaz, sustituyendo en su uso a la aplicación. En estos momentos la aplicación pasará a ser un módulo que será registrado como cliente del servicio web. Será pues función de este servicio el comunicarse e interactuar con el repositorio que ahora sí, se encuentra en una máquina diferente de la aplicación.

5.4.1 Componentes

Para incluir los dos repositorios necesitaremos la otra interfaz – IComponentRepository – que incluiremos en la estructura con lo que el servicio *REST*ful también deberá implementarla .

Por otro lado para esta parte necesitaremos modificar el módulo de la aplicación, que pasará a ser un cliente *Restlet* y debe por tanto:

- Cumplir el contrato formal establecido en *5.2.4* como parte cliente.
- Soportar el protocolo *TLS*.

El componente que nos falta es el servicio *REST*, que será un módulo dentro de la estructura el cual :

- Debe cumplir el contrato formal establecido en *5.2.4* como parte servidor.
- Utilizará la interfaz correspondiente para comunicarse con el repositorio.
- Deberá implementar el protocolo *TLS*.

5.4.2 Estructura

En el diseño final, la aplicación reside en una máquina distinta del repositorio, además como se aprecia en el diagrama (*figura 10*) se ha conseguido modularidad total, esto facilitará la integración, el despliegue y el mantenimiento.

En este diseño el servicio *REST*ful debe implementar las dos interfaces las cuales le servirán para comunicarse con los repositorios. La aplicación se conecta vía web con el servicio, será este quien deberá cubrir todas las necesidades de la aplicación. El servicio web supone una capa de software que estará por encima del repositorio. Los módulos del servicio web y el repositorio se encuentran bajo el mismo sistema operativo.

Los repositorios crean y gestionan la estructura de directorios, que han debido ubicar en el sistema de archivos de la máquina donde residen. Se comunican a través de la interfaz correspondiente al tipo de archivos que utilicen. Esta comunicación entre módulos se realizará por medio del registro de servicios que ofrece OSGi.

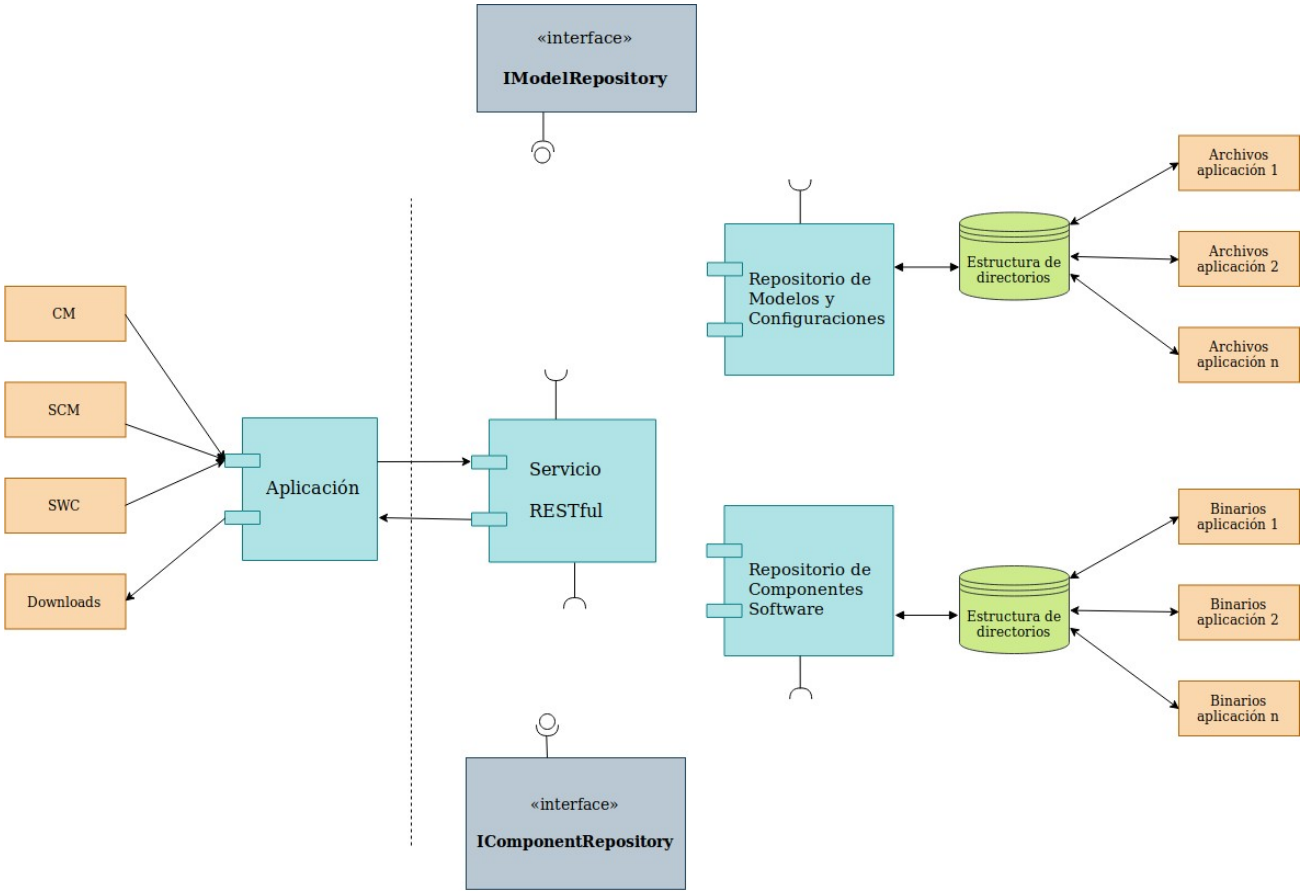


figura 10. Diagrama del diseño final con los dos repositorios.

6. Implementación

Siguiendo con el guión del diseño, realizaremos una primera aproximación implementando el repositorio en la misma máquina donde se encuentra la aplicación. En esta aproximación no se va a implementar el borrado de archivos en el repositorio. Concluiremos con la implementación final.

6.1 Implementación de la primera aproximación.

Tanto la interfaz, la aplicación, como el repositorio van a ser implementados como *bundle*s en *OSGi*. Se mostrarán estos módulos y las relaciones que se establecen entre ellos para una correcta comunicación y funcionamiento.

El procedimiento para exportar e importar paquetes se realiza a través de interfaz gráfica en Eclipse, pero queda todo reflejado en el *Manifest.mf* del *bundle* (*ver 2.1.1*), por lo que estas relaciones entre distintos paquetes las visualizaremos a través de este archivo en cada uno de los módulos. Los paquetes resultantes se mostrarán en diagramas basados en UML – *Unified Modeling Language* – .

Paralelamente crearemos los métodos en cada una de las clases mostrando su sintaxis y función dentro de la implementación.

6.1.1 Interfaz IModelRepository

Seguidamente crearemos la interfaz que definirá el tipo de repositorio y los métodos a utilizar.

En el repositorio de modelos uno de los requisitos es que se puedan realizar operaciones *CRUD*, por lo que los métodos a utilizar están bien definidos.

Operación	IModelRepository.java	
	CM	SCM
Create	`saveComponentModel()`	`saveSystemConfigurationModel()`
Read	`getComponentModel()`	`getSystemConfigurationModel()`
Update	`saveComponentModel()`	`saveSystemConfigurationModel()`
Delete	`deleteComponentModel()`	`deleteSystemConfigurationModel()`

El procedimiento es el habitual para crear una interfaz en *Java*, pero ha de cumplir ciertos requisitos para ser un *bundle*.

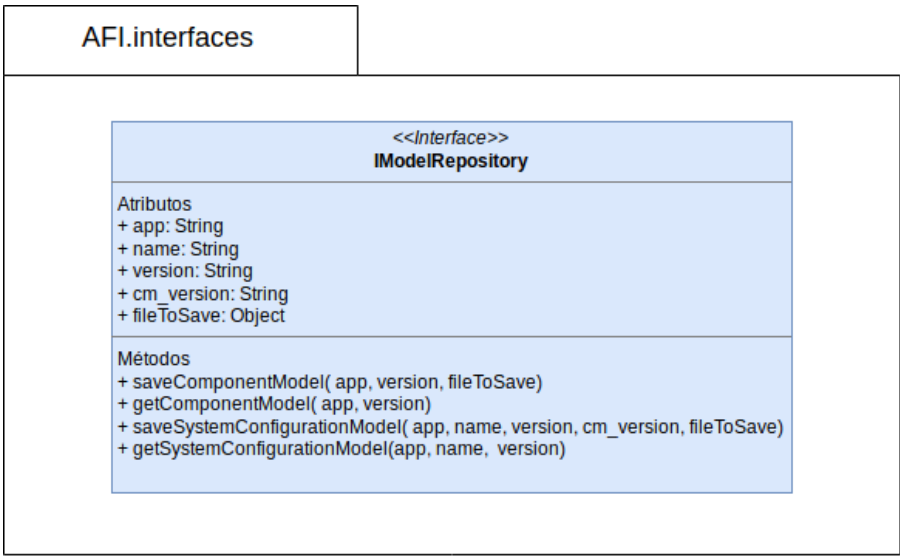
- Se crea el módulo utilizando los mecanismos *OSGi*, este módulo puede contener una o varias interfaces. Todas las interfaces creadas se envuelven en un paquete (*package*) que podrá ser exportado.
- Una vez creado el módulo se exporta el paquete (*export-package*) con las interfaces para que pueda ser invocado por otro módulo .

En el *Manifest* observamos el nombre de nuestro *bundle* – `AFI.Interfaces` – y el paquete que exportamos (*figura 11*).

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: AFI.Interfaces
Bundle-SymbolicName: AFI.interfaces
Bundle-Version: 1.0.0
Export-Package: afi.interfaces
```

figura 11. Manifest del módulo Interfaz.

El paquete de la interfaz quedará como sigue:



6.1.2 Aplicación cliente AFI.Automovil

Para crear la aplicación como un *bundle* seguiremos los siguientes pasos:

Se crea el módulo con *OSGi*, en este caso hay que destacar una clase que resulta fundamental para el funcionamiento modular, la clase `Activator.java` . En esta clase se establece la 'centralita' del módulo, ya que es el punto de entrada y salida de datos, así como la responsable del encendido y apagado de este y que tiene por defecto dos métodos `start()` y `stop()`.

Además en esta clase se establece la comunicación con el repositorio a través de la interfaz por medio de referencias a servicio. Estas referencias no son más que el alta de la aplicación en la funcionalidad ofrecida por el repositorio (*figura 12*).

```
private IModelRepository afi_IModelRepository;
private ServiceReference<IModelRepository> s_ref;

. . . . . .

s_ref = (ServiceReference<IModelRepository>)
       context.getServiceReference(IModelRepository.class.getName());
afi_IModelRepository = (IModelRepository) context.getService(s_ref);
```

figura 12. Fragmento de afi.automovil.Activator.java

La aplicación debe realizar las acciones descritas en *5.3.1*, para ello se van a crear – dentro de `Activator.java` – los siguientes métodos:

Para los modelos de componentes :

Operación	Método	Función
Create	`setComponents(String version)`	Busca un modelo de componente y lo pasa al repositorio como un archivo *JSON*.
Read	`afi_IModelRepository.getSystemConfigurationModel()`	Realiza una petición de modelo de componente
Update	`setComponents(String version)`	Actualiza un modelo

`setComponents()`: Este método busca un modelo de componente que está almacenado en el propio módulo, posteriormente hacemos uso de la interfaz – IModelRepository – a través del servicio que ofrece y se pasa el contenido del archivo al repositorio para que éste lo almacene.
Este método se utiliza igualmente para la actualización, cuando enviamos el contenido va siempre ligado a una versión, el repositorio sobrescribe el archivo almacenado.
Para realizar la petición de archivo se utiliza directamente el servicio referenciado.

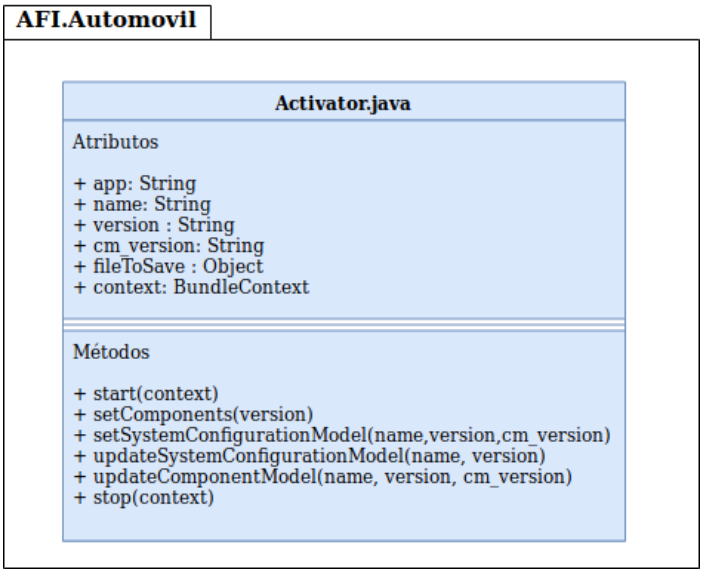
Para los modelos de configuraciones del sistema :

Operación	Método	Función
Create	`setSystemConfigurationModel()`	Busca una configuración y la pasa al repositorio como un archivo *JSON*.
Read	`afi_IModelRepository.getSystemConfigurationModel()`	Realiza una petición de configuración del sistema
Update	`updateSystemConfigurationModel()`	Actualiza la configuración del sistema según versión.
	`updateComponentModel()`	Actualiza el modelo de componente asociado a esta configuración

`setSystemConfigurationModel()`: Este método busca un modelo de configuración que se encuentra guardado en el módulo, a continuación y a través del servicio de la interfaz – IModelRepository – pasaremos este archivo al repositorio que lo almacenará.

`updateSystemConfigurationModel()`: Este método busca un modelo de configuración del sistema que está almacenado en el módulo en un directorio 'new-input' y lo pasa al servicio de la interfaz para que el repositorio actualice esa configuración, todo basado en su nombre y versión.
Al igual que con los modelos de componente para realizar la petición de lectura se usa directamente la referencia al servicio.

El paquete de la aplicación AFI.Automovil quedaría como sigue:



Es imprescindible importar el paquete que contiene la interfaz. Además es necesario el paquete del *framework OSGi* para el correcto funcionamiento (*figura 13*) .

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Automovil
Bundle-SymbolicName: AFI.Automovil
Bundle-Version: 1.0.0
Bundle-Activator: afi.automovil.Activator
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Automatic-Module-Name: AFI.Automovil
Bundle-Bundle-ActivationPolicy: lazy
Import-Package: afi.interfaces, org.OSGi.framework;version="1.9.0"
```

figura 13. Manifest del módulo aplicación.

6.1.3 Repositorio AFI.repository

Este módulo es el más importante y a la vez el más complejo de los tres que vamos a crear en esta primera aproximación. Como se ha mencionado en el punto anterior la parte más importante de cualquier módulo *OSGi* reside en la clase `Activator.java` que en este caso se utiliza para registrar el servicio permitiendo que pueda ser utilizado por otros módulos (*figura 14*).

```
ServiceRegistration<IModelRepository> s_ref;
Activator.context = BundleContext;

IModelRepository repoService = new ModelRepository();

s_ref=(ServiceRegistration<IModelRepository>)
BundleContext.registerService(IModelRepository.class.getName(),repoService, null);
```

figura 14. Fragmento del método start() en afi.repository.Activator.java

Con respecto a la funcionalidad, mostramos la clase principal para realizar el trabajo :

`ModelRepository.java`

Para los modelos de componente se crearán los siguientes métodos:

Operación	Método	Función
Create	`saveComponentModel()`	Almacena en la estructura de directorios el archivo pasado desde la aplicación.
Read	`getComponentModel()`	Busca un archivo CM dentro de la estructura y devuelve su contenido.El contenido se envia en un archivo *JSON*.
Update	`saveComponentModel()`	Actualiza un modelo de componente.

`saveComponentModel()`: Este método se utiliza igualmente tanto para la creación como para la actualización, en la implementación final se desarrolla esta característica.

Para los modelos de configuraciones del sistema se van a crear estos métodos:

Operación	Método	Función
Create	`saveSystemConfigationModel()`	Almacena en la estructura de directorios el archivo pasado desde la aplicación.
Read	`getSystemConfigurationModel()`	Busca un archivo SCM dentro de la estructura y devuelve su contenido . Busca en el registro el componente asociado y lo añade al contenido. El contenido se envia en un archivo *JSON*.
Update	`updateSystemConfigurationModel()`	Busca un archivo dentro de la estructura y actualiza la configuración del sistema según versión.
	`updateConfigurationModel()`	Busca dentro del registro el modelo de componente asociado a una configuración y lo actualiza.

Recordamos que en esta primera aproximación no se implementa la operación de borrado, esto lo realizaremos en la implementación final.

`ModelRepository.java` está relacionada con otras dos clases que realizan funciones de tratamiento de archivos, las cuales se encuentra en un paquete denominado `utils`, estas clases son `ManagementFiles.java` que gestiona los archivos genéricos y `ManagementJSON.java` que gestiona los archivos *JSON* de registro (*ver 5.2.2*) y de respuesta (*ver 5.2.3*) .

`ManagementFiles.java` va a ser utilizada por `ModelRepository.java` para la gestión de los archivos CM y SCM. Se encargará de guardar y leer los modelos y configuraciones dentro del sistema de archivos del sistema operativo.

`ManagementFiles.java`

Método	Función
`saveAsFile()`	Almacena un archivo en el repositorio
`saveJsonFile()`	Almacena un archivo *JSON* (registro) en el repositorio
`readFile()`	Lee y convierte a `String` cualquier archivo
`getFilefromRepository()`	Busca y devuelve el modelo de componente asociado a una configuración

`ManagementJSON.java` se va a encargar de la gestión de los archivos con formato *JSON*. Entre sus funciones estará la creación de los archivos de registro y de respuesta, así como de la actualización de los mismos.

`ManagementJSON.java`

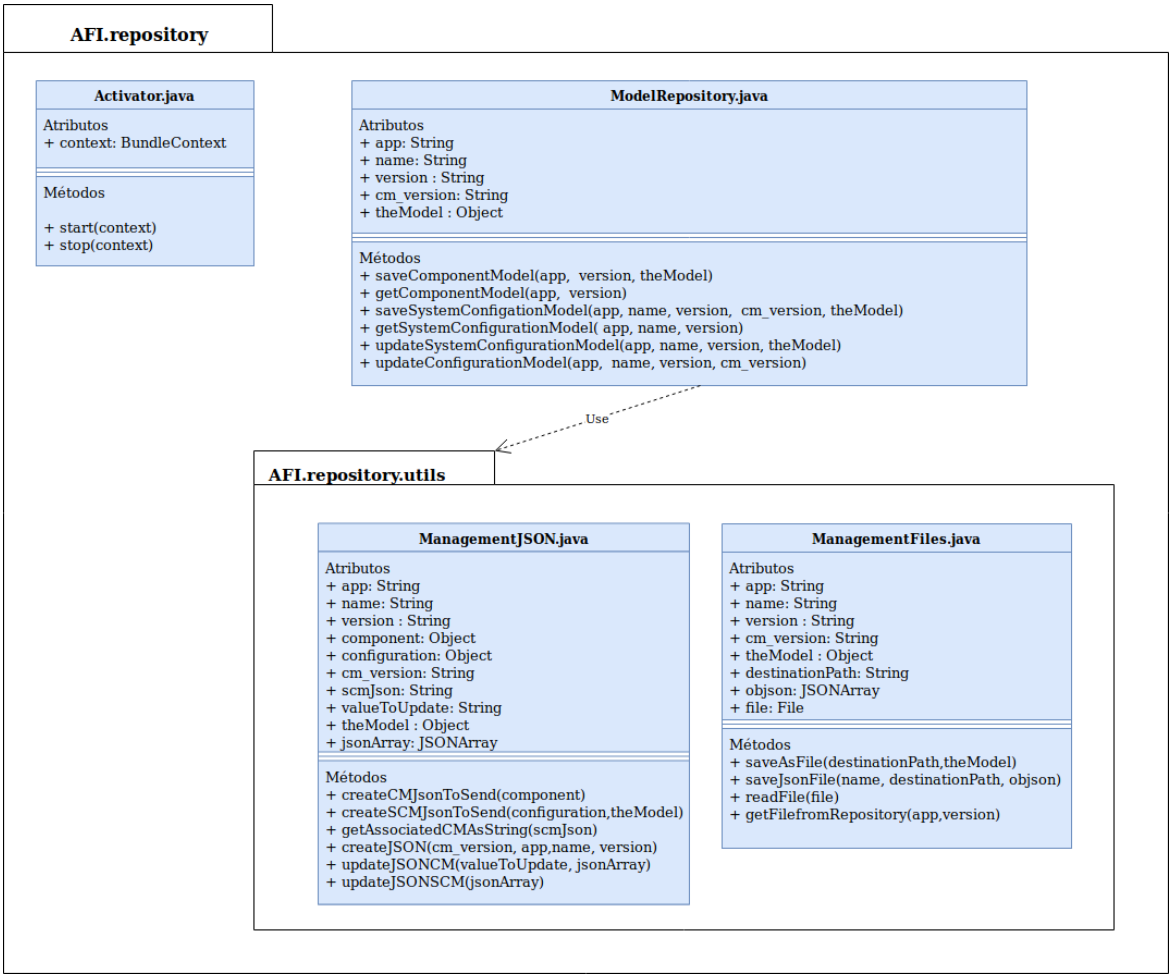
Método	Función
`createCMJsonToSend()`	Crea el archivo *JSON* que será devuelto como respuesta a una petición de modelo de componente
`createSCMJsonToSend()`	Crea el archivo *JSON* que será devuelto como respuesta a una petición de configuración del sistema
`getAssociatedCMAsString()`	Devuelve el modelo de componente asociado a una configuración como un `String`
`createJSON()`	Crea el registro de entrada tanto para CM como para SCM
`updateJSONCM()`	Actualiza registro de entrada para CM
`updateJSONSCM()`	Actualiza registro de entrada para SCM

Una vez hemos creado las clases y registrado el servicio hay que importar los paquetes necesarios para el funcionamiento, como es el de la interfaz y el de *OSGi*, además como vamos a manejar archivos *JSON* también serán necesarios los paquetes correspondientes. Del mismo modo hay que exportar los paquetes contenidos en el módulo (*figura 15*).

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Repository
Bundle-SymbolicName: AFI.repository
Bundle-Version: 1.0.0
Bundle-Activator: afi.repository.Activator
Import-Package: afi.interfaces,
 org.JSON,
 org.OSGi.framework
Export-Package: afi.repository,
 afi.repository.utils
```

figura 15. Manifest del módulo repositorio.

En este diagrama se muestra cómo queda este paquete AFI.repository:



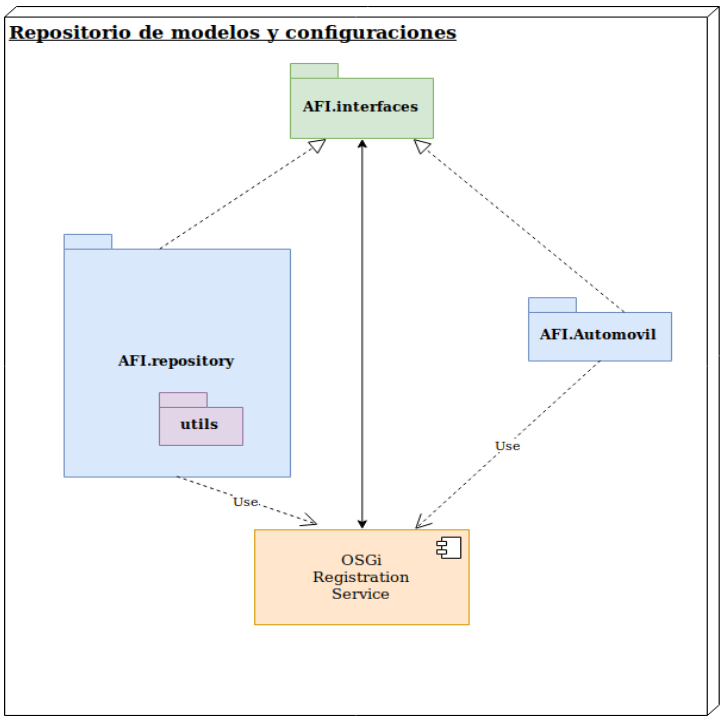
En estos momentos lo que nos queda es agrupar todos los módulos creados y testar el funcionamiento. Para ello crearemos un espacio de trabajo dentro de Eclipse en el cuál ejecutaremos los módulos.

Con el fin de realizar la simulación de un entorno real la aplicación irá realizando acciones a través de su *Activator*.

Características del test:

- Tanto los modelos de componentes como las configuraciones del sistema que la aplicación va a enviar al repositorio están almacenados en sendas carpetas dentro del módulo aplicación.
- Cuando realice una actualización de una configuración, el archivo a enviar se encontrará en una carpeta 'new-input' también localizada dentro del módulo aplicación.
- Las respuesta a una petición de archivo se visualizarán por consola.
- Los mensajes de estado se visualizarán igualmente por consola.

Centrándonos en los paquetes la estructura final y las relaciones que se producen quedarán entonces de la siguiente forma:



Los paquetes AFI.repository y AFI.Automovil implementan la interfaz IModelRepository que se encuentra en AFI.interfaces. La comunicación se hace efectiva a través del servicio de registros OSGi.

En esta implementación se han creado tres módulos con sus respectivos paquetes para exportar. Estos módulos importarán los paquetes que necesiten para su funcionalidad. La comunicación entre ellos se realiza por medio del servicio de registros OSGi, y utilizan la interfaz como medio para realizarla.

6.2 Implementación final.

Proseguimos con la implementación final, en la que entrará en juego el servicio *REST*ful. Este servicio web se va a crear como un módulo, conectándolo al repositorio a través de la interfaz. Esta será la forma que tendrá la aplicación de intercambiar archivos con el repositorio, se va a implementar con *Restlet framework (ver 2.1.2).*

Añadiremos la otra interfaz para crear el enlace necesario con el repositorio de componentes software. Además incluiremos las operaciones de borrado completando así el cumplimiento de los requisitos referidos al repositorio.

La aplicación deberá ahora crearse como cliente web mediante *Restlet*, con lo que habrá que modificar el código e importar los paquetes necesarios.

6.2.1 Módulo AFI.interfaces

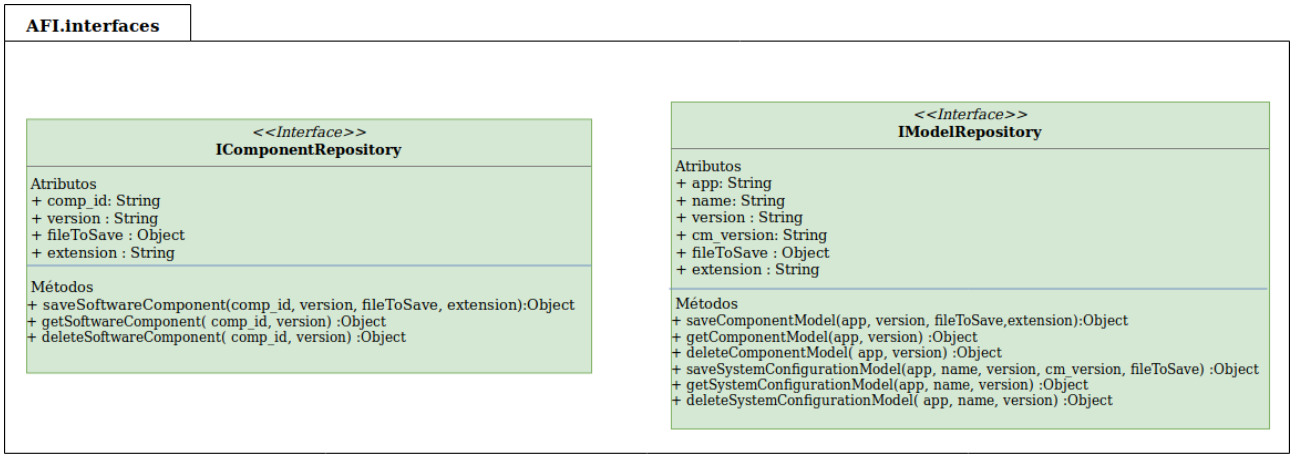
Lo primero que vamos a hacer y para cumplir con los requisitos es añadir la interfaz de componentes software `IComponentRepository.java`.

Para el caso de los componentes software no es necesaria la actualización.

Operación	IComponentRepository.java
	SWC
Create	`saveSoftwareComponent ()`
Read	`getSoftwareComponent ()`
Delete	`deleteSoftwareComponent ()`

Con el fin de poder diferenciar los distintos archivos desde origen se ha añadido el atributo ‘extension’ a los métodos que se encargan del almacenado. Reseñar también que los archivos – CM, SCM o SWC – serán tipados como *Object*.

Paquete AFI.interfaces definitivo:



6.2.2 Módulo AFI.Automovil

Para este módulo se crearán dos paquetes, uno que albergará al cliente *Restlet* y otro que contendrá la clase ya mencionada `Activator.java` .

6.2.2.1 Cliente Restlet

La clase `ClientResource` en *Restlet*, actúa como un proxy de un recurso determinado. Puede funcionar con cualquier recurso del lado del servidor remoto implementado en cualquier tecnología para la que haya un protocolo definido y un conector *Restlet* disponible, como *HTTP*, POP3 o FTP. Esta clase se puede considerar un cliente de nivel superior porque, aparte de la conectividad que ofrecen los conectores de cliente de nivel inferior – los elementos *Restlet* implementan concretamente un protocolo específico – , ofrece características como una estrategia de reintento para solicitudes fallidas, la capacidad de seguir automáticamente la redirección o la serialización transparente entre las representaciones de *Restlet* enviadas o recuperadas, y los objetos *Java* de nivel superior como representaciones [5].

Entre sus características destacaremos la denominada *parent* que es el recurso basado en la URL raíz al que el cliente puede realizar peticiones. Mediante esta característica se puede acceder fácilmente – en términos de código – a los recursos que ofrece el servidor (*figura 16*).

```
ClientResource resource;
resource = service.getChild("/cmodel/" + app + "/" + version);
respuesta = resource.get().getText();

. . . . . .

resource = service.getChild("/cmodel/" + app + "/" + version);
resource.put(resource.toRepresentation(mensaje.toString(), MediaType.APPLICATION_JSON));
```

figura 16. Ejemplo de operación GET y PUT utilizando la característica parent en Restlet .

Así mismo haremos un uso intensivo de la serialización transparente del envío y petición de archivos mediante representaciones *JSON*.

Hemos denominado a la clase como `RestletClient.java` y contiene los siguientes métodos:

Operación	Método	Función
Constructor	`RestletClient()`	Se conecta al servidor *Restlet* mediante una IP bien conocida.
GET/DELETE	`sendGetOrDel()`	Realiza una operación según el valor del parámetro ‘método’ .
PUT	`sendToServer()`	Envía archivos hacia el servidor.

En esta clase deberemos implementar el protocolo *TLS* en la parte cliente, esta cuestión se desarrolla más adelante.

6.2.2.2 Activator

Esta clase ya no tiene que realizar la referencia a servicio de *OSGi*, sin embargo tiene que utilizar el cliente *Restlet* `RestletClient.java`, con lo que los métodos difieren de la primera aproximación.

`Activator.java`

Operación	Método	Función
Activar *bundle*	`start()`	Pone en marcha el *bundle*. Realiza todas las acciones de módulo – envío y recuperación de archivos –
Parar *bundle*	`stop()`	Detiene la ejecución del *bundle*
Create	`sendComponent()`	Construye un archivo *JSON* con los campos necesarios para su envío. Busca un archivo CM o SWC dentro de la estructura y lo añade al archivo *JSON*.
	`sendSoftwareComponent()`	
	`sendSystemConfigurationModel()`	Construye un archivo *JSON* con los campos necesarios para su envío, para ello busca un archivo SCM dentro de la estructura y lo añade. Busca en el registro el componente asociado y lo añade al archivo.El archivo resultante será el que se envíe.
Update	`updateSystemConfigurationModel()`	Construye un archivo *JSON* con los campos necesarios para su envío.
	`updateComponentModel()`	Construye un archivo *JSON* con los campos necesarios para su envío.
Read	`N/A`	Utiliza directamente `RestletClient.java`
Delete	`N/A`	Utiliza directamente `RestletClient.java`

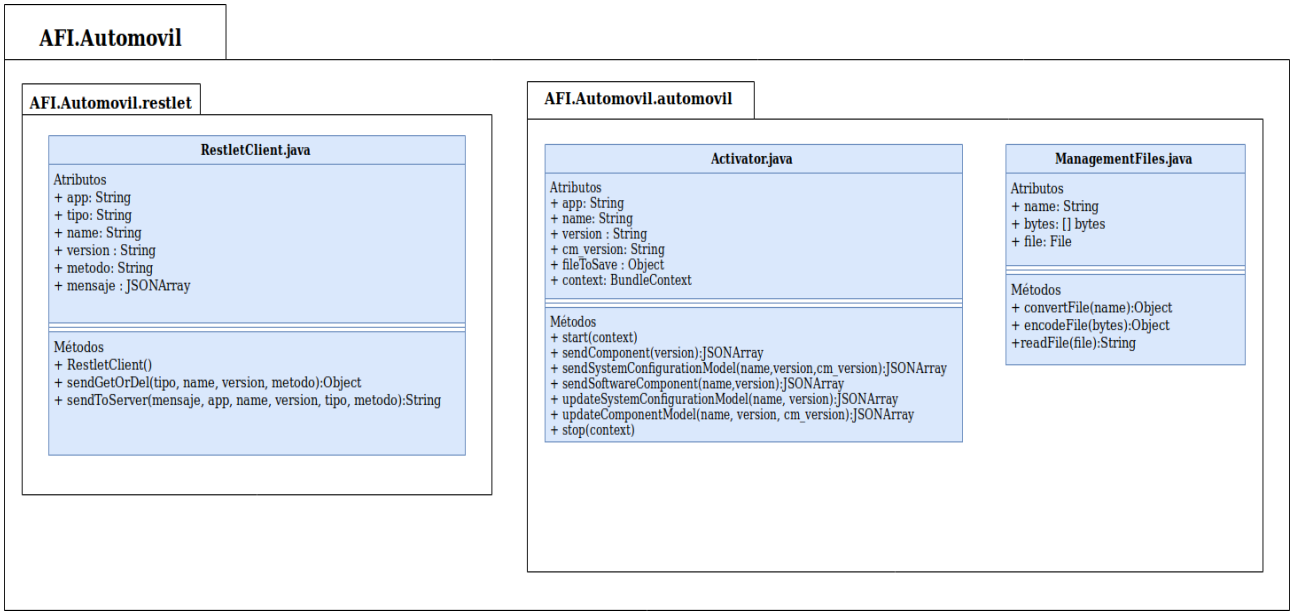
La clase `Activator.java` se apoya en otra para realizar tareas sobre archivos y se denomina `ManagementFiles.java` *(ver Apéndice)*.

Si observamos el *Manifest* podemos ver los paquetes *Restlet* que son necesarios importar aparte de los ya importados en la primera aproximación *(figura 17)*.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Automovil
Bundle-SymbolicName: AFI.Automovil
Bundle-Version: 1.0.0
Bundle-Activator: afi.automovil.Activator
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Automatic-Module-Name: AFI.Automovil
Bundle-ActivationPolicy: lazy
Import-Package: org.JSON;version="20080701.0.0", org.OSGi.framework;version="1.9.0",
 org.restlet,org.restlet.data,org.restlet.engine,org.restlet.ext.OSGi,org.restlet.util,
 org.restlet.resource,org.restlet.representation,org.restlet.routing,org.restlet.security
 org.restlet.service
```

figura 17. Manifest del módulo AFI.Automovil

A continuación se muestra cómo queda el paquete AFI.Automovil:



6.2.3 Módulo AFI.repository

Para la implementación definitiva deberemos añadir el repositorio de componentes software, para ello comenzaremos registrando el servicio utilizando `Activator.java` (*figura 18).*

```
Activator.context = bundleContext;

ServiceRegistration<IComponentRepository> serv_crs;
IComponentRepository crsService = new ComponentRepository();

serv_crs = (ServiceRegistration<IComponentRepository>)
bundleContext.registerService(IComponentRepository.class.getName(),crsService,null);
```

figura 18. Fragmento de afi.repository.Activator.java

Seguidamente creamos la clase `ComponentRepository.java` que contará con los siguientes métodos:

`ComponentRepository.java`

Operación	Método	Función
Create	`saveSoftwareComponent()`	Almacena en la estructura de directorios el archivo pasado desde el servicio web.
Read	`getSoftwareComponent()`	Busca un archivo SWC dentro de la estructura y devuelve su contenido en un archivo *JSON*.
Delete	`deleteSoftwareComponent()`	Elimina un componente software SWC.

Continuando con la implementación final habrá que añadir la operación de borrado a los métodos que ya teníamos en `ModelRepository.java`. Además es necesario añadir un método para gestionar y diferenciar los archivos que se reciben, esto es, determinar si se trata de una petición de almacenado o por el contrario es una actualización.

`ModelRepository.java`

Operación	Método	Función
Delete	`deleteComponentModel()`	Elimina un modelo de componente.
	`deleteSystemConfigurationModel()`	Elimina un modelo de configuración.
Identificación operación	`managementUpdates()`	Determina el tipo de operación – Create/Update –

Al paquete `utils` que contiene las clases que manejan los archivos genéricos y los archivos *JSON* hay que añadir las clases que operan con el repositorio de componentes software. El paquete definitivo contendrá las siguientes clases *(ver Apéndice)* .

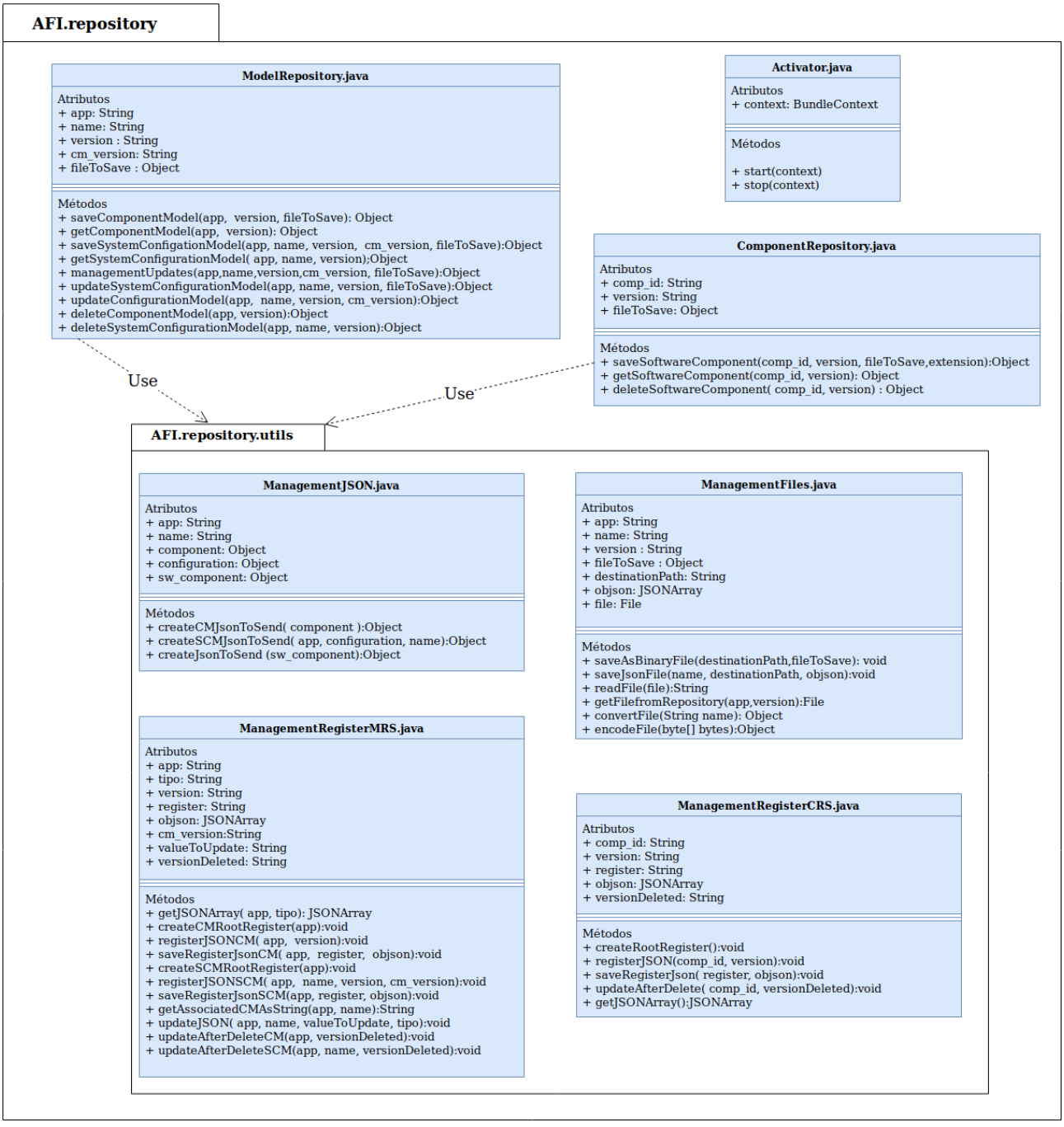
Paquete `utils`	`ManagementFiles.java`
	`ManagementJSON.java`
	`ManagementRegisterMRS.java`
	`ManagementRegisterCRS.java`

El repositorio sigue necesitando de la interfaz para comunicarse y exportar los paquetes propios para proporcionar el servicio, por lo tanto la posición dentro de la estructura no ha cambiado. El archivo *Manifest* permanece inalterado.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Repository
Bundle-SymbolicName: AFI.repository
Bundle-Version: 1.0.0
Bundle-Activator: afi.repository.Activator
Import-Package: afi.interfaces,
 org.JSON,
 org.OSGi.framework
Export-Package: afi.repository,
 afi.repository.utils
```

Manifest del módulo repositorio.

Seguidamente se muestra el diagrama del paquete resultante:



6.2.4 Módulo AFI.restservice

Para comenzar vamos a realizar una pequeña introducción al concepto de servicio y cual de sus diferentes tipos utilizaremos en nuestro trabajo.

Concepto de servicio

Desde una perspectiva general, un servicio es un programa de software que hace que su funcionalidad esté disponible a través de una interfaz publicada, denominada contrato de servicio. Un contrato de servicio puede dividirse en un conjunto de capacidades de servicio, cada una de las cuales expresa una función ofrecida por el servicio a otros programas de software.

Por otro lado tenemos al consumidor de servicios, que es el rol en tiempo de ejecución asumido por un programa de software cuando accede e invoca un servicio o más específicamente, cuando envía un mensaje a una capacidad de servicio expresada en el contrato. Al recibir la solicitud, el servicio comienza a procesarse y puede o no devolver el mensaje de respuesta correspondiente al consumidor.

Por último se encuentra la figura del agente de servicio que es un programa controlado por eventos – a menudo denominado interceptor, oyente o filtro – que no proporciona una interfaz publicada. En su lugar, está diseñado para actuar como un intermediario capaz de interceptar mensajes en tiempo de ejecución. Cuando se intercepta un mensaje, el agente de servicio puede realizar un procesamiento activo o pasivo sobre este. La lógica de procesamiento del agente generalmente se considera activa cuando termina alterando el contenido del mensaje, mientras que la lógica de procesamiento pasivo no lo hace [11].

Tipos de servicio

Existen varios tipos de servicio, están como los utilizados en el sistema *OSGi* donde se utiliza la interfaz *Java* y un registro de servicios, y hay algunos orientados a trabajar en red .Un tipo de estos servicios es un servicio web – *Web service* – , que es una interfaz que describe una colección de operaciones a las que se puede acceder en red a través de mensajes *XML* estandarizados. Un servicio web se describe utilizando una noción *XML* formal estándar, denominada contrato de servicio. Los servicios web cumplen una tarea específica o un conjunto de tareas [12].

Otro tipo de servicio orientado a la red es un servicio *REST* que está basado en el protocolo *HTTP* y sus métodos [13]. Este es un tipo de servicio web en el que no es necesaria la descripción del mismo usando *XML*. Al estar basado en *HTTP* el único compromiso o contrato entre productor y consumidor es la definición de los recursos, todas las demás acciones son estándar. *REST* impone una serie de restricciones de diseño y arquitectura al contrato de servicio y a la lógica, pero no restringe cómo y con qué propósito se pueden diseñar y utilizar los servicios de forma individual y en relación con los demás. Por lo tanto, tiene mucha libertad en cuanto a cómo puede llevar a cabo un proceso para modelar y diseñar servicios destinados a automatizar la lógica del sistema [11].

Servicios *REST*ful

Los objetivos de *REST*ful son principalmente de naturaleza técnica y deben alcanzarse tomando decisiones de diseño claras para producir una arquitectura tecnológica que se parezca mucho a la *Web*. Estas decisiones de diseño se definen a través de restricciones asociadas a propiedades arquitectónicas que representan objetivos de diseño.

Restricciones más importantes:

Cliente-Servidor → Requiere que un servicio ofrezca uno o más recursos y escuche las solicitudes de estos recursos. Un consumidor invoca un recurso enviando el mensaje de solicitud correspondiente, y el servicio rechaza la solicitud o realiza la tarea solicitada antes de enviar un mensaje de respuesta al consumidor. Las excepciones que impiden que la tarea continúe se devuelven al consumidor, y el consumidor es responsable de tomar medidas correctivas.

Sin estado → Cada solicitud de un consumidor del servicio debe contener toda la información necesaria para que el servicio comprenda el significado de la solicitud, y todos los datos del estado de la sesión deben devolverse al consumidor del servicio al final de cada solicitud.

Interfaz / Contrato formal → Todos los servicios y consumidores de servicios dentro de una arquitectura compatible con *REST* deben compartir una única interfaz general. Al igual que la restricción principal que distingue a *REST* de otros tipos de arquitectura, la interfaz generalmente se aplica utilizando los métodos y tipos de medios proporcionados por *HTTP* y otros estándares de Internet.

Sistema en capas → Una solución basada en *REST* puede estar compuesta de varias capas arquitectónicas, y ninguna capa puede "ver más allá" de la siguiente. Las capas se pueden agregar, eliminar, modificar o reordenar en respuesta a cómo la solución debe evolucionar. La restricción del sistema en capas se basa en el modelo cliente-servidor para agregar componentes de *middleware* – componentes intermedios que pueden existir como servicios o agentes de servicio – a una arquitectura.

REST proporciona el siguiente conjunto de "propiedades clave" que ayudan a establecer los objetivos de diseño que se encuentran detrás de la aplicación de las restricciones *REST*:

Rendimiento, escalabilidad, simplicidad, posibilidad de modificar, visibilidad, portabilidad, fiabilidad y seguridad.

Estas propiedades representan a un estado objetivo de una arquitectura que se parece a la *World Wide Web*. Aunque muchas de las decisiones de diseño llevadas a cabo durante la aplicación de restricciones *REST* ayudan a alcanzar estos objetivos, varias de estas propiedades pueden realizarse o mejorarse al tomar decisiones de diseño adicionales que no son necesariamente partes formales de *REST* [12].

6.2.4.1 Implementación

Este módulo albergará al servicio *REST*ful el cual se va implementar como un servidor utilizando *Restlet framework.* El módulo deberá ofrecer servicio a la aplicación la cual le hará peticiones sobre archivos y será el servicio web el que se comunique con el repositorio, para que a partir de este momento se considere repositorio remoto (*ver 5.4.2*) .

Contendrá dos clases principales y su cometido será el de gestionar las conexiones con la red e implementar los recursos como servicio *REST*. En un paquete adjunto se encontrarán las clases que se encargan de manejar las entradas y salidas de archivos a través de los métodos acordados (*ver 5.2.4*).

Empezaremos con la clase *Activator* – imprescindible en *OSGi* – y que aquí denominamos `RestActivator.java`. Si hemos de trabajar con dos repositorios deberemos crear sendas referencias a servicio o 'altas' en el servicio ofrecido (*figura 19*).

```
private static IModelRepository afi_ModelRepository;
private static IComponentRepository afi_ComponentRepository;
private ServiceReference<IModelRepository> serv_mrs;
private ServiceReference<IComponentRepository> serv_crs;
. . . . . .
RestActivator.context = bundleContext;

serv_mrs = (ServiceReference<IModelRepository>)
       context.getServiceReference(IModelRepository.class.getName());
afi_ModelRepository = (IModelRepository) context.getService(serv_mrs);

serv_crs = (ServiceReference<IComponentRepository>)
       context.getServiceReference(IComponentRepository.class.getName());
afi_ComponentRepository = (IComponentRepository) context.getService(serv_crs);
```

figura 19. Fragmento de AFI.restservice.RestActivator.java

Será pues cometido de esta clase – aparte de encender y apagar el módulo – el comunicarse con el repositorio a través del servicio ofrecido. Recibirá los parámetros a utilizar de las clases que se encuentran en el paquete adjunto y que gestionan el intercambio de mensajes con la aplicación. Así mismo deberá implementar el protocolo *TLS* en la parte del servidor, asunto que trataremos más tarde (*7.1*).

La siguiente tarea es crear la clase que implementa la referencia a los recursos como servicio *REST*, a esta clase la hemos denominado `MRServerApplication.java`. Utilizando *Restlet* y en concreto el objeto *Router,* se establecen las URL que se van a utilizar para localizar los recursos (*figura 20*).

```
Router router = new Router(getContext());
       router.attach("/", RootServerResource.class);
       router.attach("/cmodel/{app}/{version}", MRSComponentModel.class);
       router.attach("/scmodel/{app}/{name}/{version}", MRSConfigurationModel.class);
       router.attach("/component/{component_id}/{version}", CRSSoftwareComponent.class);
```

figura 20. Fragmento de AFI.restservice. MRServerApplication.java

Para continuar pasamos al paquete que contiene las clases que albergan el tratamiento de los métodos *HTTP* – GET, PUT y DELETE – y que hemos denominado `logic`.

Comenzamos con la implementación del repositorio de modelos de componente y modelos de configuraciones del sistema. Se han definido dos clases `MRSComponentModel.java` y `MRSConfigurationModel.java`.

`MRSComponentModel.java`

Operación	Método	Función
GET	`getComponentModel()`	Recibe una petición vía web, recoje los parámetros de la consulta y los pasa al repositorio. Espera la respuesta del repositorio y envía el contenido de esta utilizando la misma vía de la petición.
PUT	`saveComponentModel()`	Recibe un archivo CM vía web, el cual pasa al repositorio con los metadatos del archivo. Enviará una respuesta en forma de código que reflejará el resultado de la operación – éxito o fracaso – .
DELETE	`deleteComponentModel()`	Recibe una petición vía web y transfirere los parámetros pasados al repositorio. Enviará una respuesta en forma de código indicando el resultado de la operación.

`MRSConfigurationModel.java`

Operación	Método	Función
GET	`getSystemConfigurationModel()`	Recibe una petición vía web, recoje los parámetros de la consulta y los pasa al repositorio. Espera la respuesta del repositorio y envía el contenido de esta utilizando la misma vía de la petición.
PUT	`saveSystemConfigurationModel()`	Recibe un archivo SCM vía web, el cual pasa al repositorio con los metadatos del archivo. Enviará una respuesta en forma de código que reflejará el resultado de la operación – éxito o fracaso – .
DELETE	`deleteSystemConfigurationModel()`	Recibe una petición vía web y transfirere los parámetros pasados al repositorio. Enviará una respuesta en forma de código indicando el resultado de la operación.

Seguimos ahora con el repositorio de componentes software, a la clase la hemos denominado `CRSSoftwareComponent.java` y contiene los siguientes métodos:

`CRSSoftwareComponent.java`

Operación	Método	Función
GET	`getSoftwareComponent()`	Recibe una petición vía web, recoje los parámetros de la consulta y los pasa al repositorio. Espera la respuesta del repositorio y envía el contenido de esta utilizando la misma vía de la petición.
PUT	`saveSoftwareComponent()`	Recibe un archivo SWC vía web, el cual pasa al repositorio con los metadatos del archivo. Enviará una respuesta en forma de código que reflejará el resultado de la operación – éxito o fracaso – .
DELETE	`deleteSoftwareComponent()`	Recibe una petición vía web y transfirere los parámetros pasados al repositorio. Enviará una respuesta en forma de código indicando el resultado de la operación.

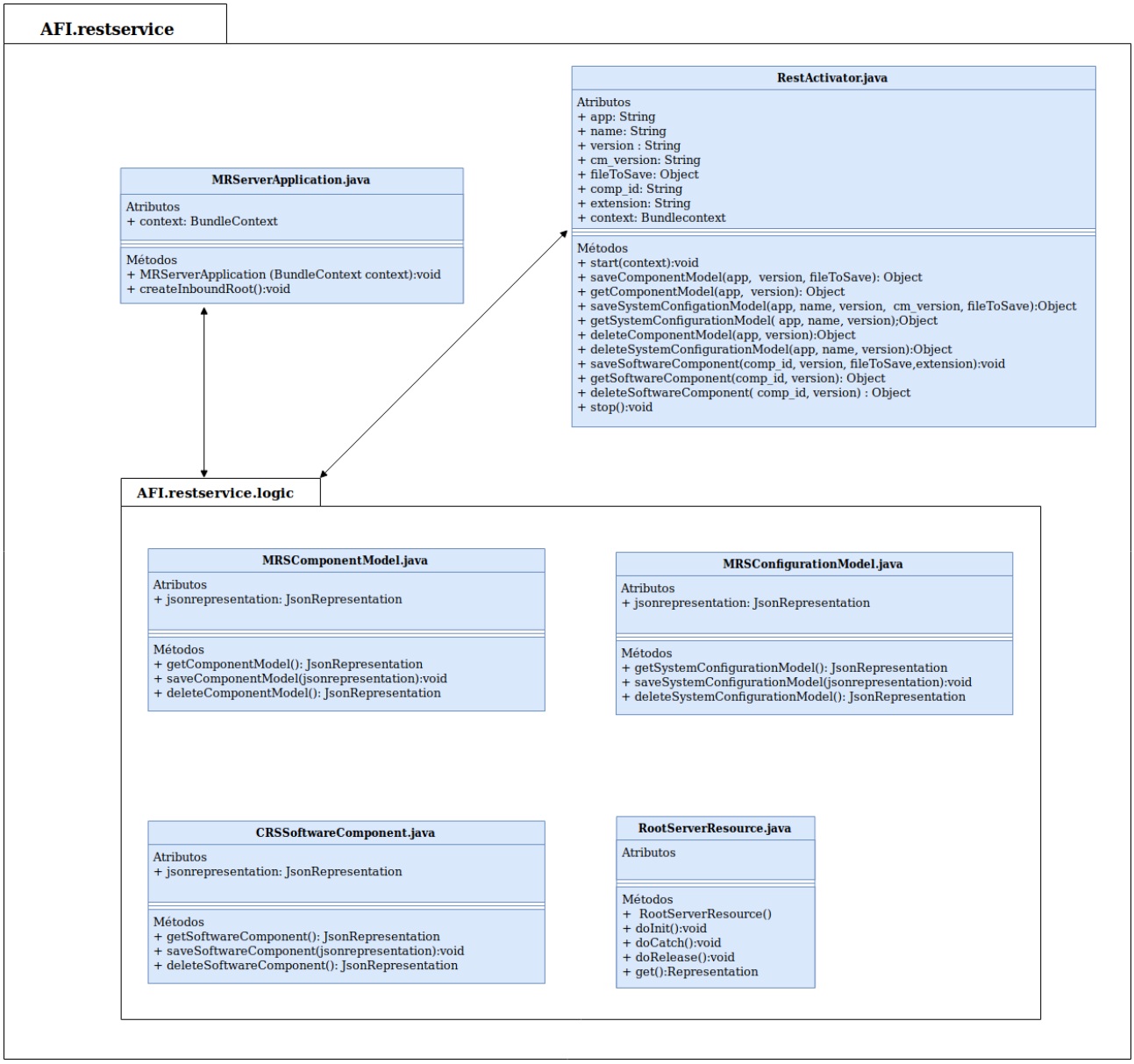
Se ha creado también la clase `RootServerResource.java` *(ver Apéndice)*, esta responde a una petición que se haga a la raiz del recurso o "/" – `www.repository.es/` –. A parte de testar el funcionamiento del servidor, devuelve un mensaje de bienvenida.

Vemos ahora como queda el archivo *Manifest*, observamos los paquetes necesarios para crear el servidor con *Restlet (figura 21).*

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Restservice
Bundle-SymbolicName: AFI.restservice
Bundle-Version: 1.0.0
Bundle-Activator: afi.restservice.RestActivator
Automatic-Module-Name: AFI.restservice
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Import-Package: afi.interfaces,
 org.JSON;version="20080701.0.0",
 org.OSGi.framework;version="1.9.0",
 org.restlet,
 org.restlet.data,
 org.restlet.ext.JSON,
 org.restlet.representation,
 org.restlet.resource,
 org.restlet.routing,
 org.restlet.service,
 org.restlet.util
```

figura 21. Archivo Manifest en AFI.restservice

El paquete resultante quedaría como sigue:



Una vez creados los módulos con sus respectivos paquetes, pasaremos a testar el funcionamiento siguiendo el diagrama de *5.4.2* .

Utilizaremos *VirtualBox (ver 2.2.3)* para colocar allí el módulo del cliente, por otro lado y ya en la máquina *host* , se ejecutarán los módulos del repositorio y del servicio *REST*ful. Se utilizarán las IP's proporcionadas por el router local y las direcciones a utilizar serán:

Modelos de componente → `https://192.168.1.102:8183/cmodel/{app}/{version}`

Modelos de configuraciones → `https://192.168.1.102:8183/scmodel/{app}/{name}/{version}`

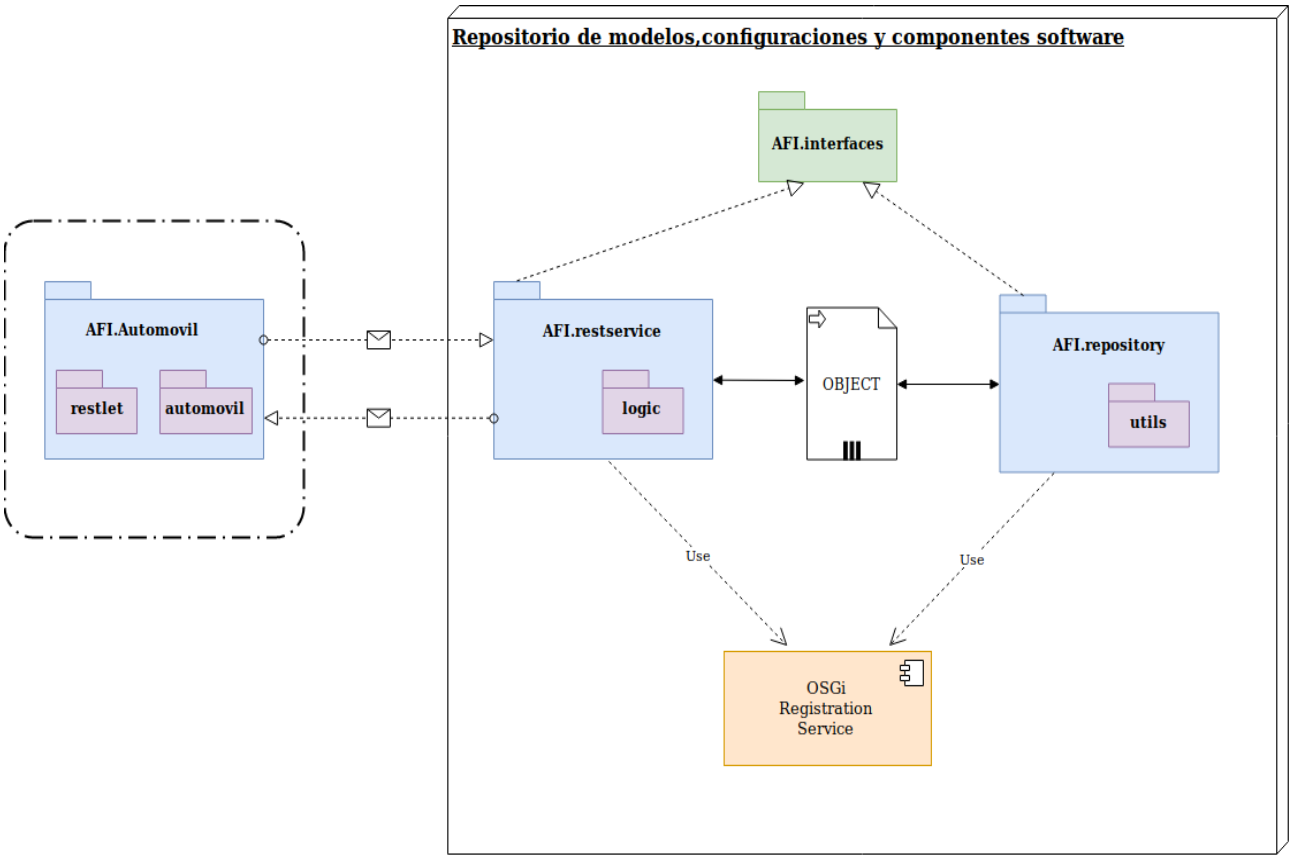
Componentes software → `https://192.168.1.102:8183/component/{component_id}/{version}`

El módulo de la aplicación realizará las mismas acciones descritas para la primera aproximación alternando el uso de los repositorios, esto es, una vez realizará acciones sobre modelos de componente y configuraciones y otra las realizará sobre componentes software.

Para comprobar el funcionamiento de las peticiones – GET – la aplicación las guardará en una carpeta denominada "downloads" *(ver 5.4.2)* .

Con esto habremos concluido con la implementación de los repositorios y del servicio *REST*ful.

Seguidamente se muestra un diagrama de la estructura y sus relaciones.



Los paquetes AFI.repository y AFI.restservice implementan la interfaz IModelRepository que se encuentra en AFI.interfaces. Estos módulos intercambian un mismo tipo de objeto. La comunicación se hace efectiva a través del servicio de registros OSGi. AFI.Automovil es ahora un módulo que se encuentra en una máquina externa al repositorio.

En la implementación final la aplicación AFI.Automovil intercambia mensajes con el servicio web AFI.restservice. Este servicio se comunica a su vez con el repositorio AFI.repository realizando un intercambio de objetos. El repositorio finalmente gestiona los directorios y da respuesta a las peticiones. Con esto habremos conseguido crear un repositorio remoto.

Se realizó el test de implementación con resultados favorables, seguidamente este repositorio será integrado dentro de la herramienta PROTeus como parte de su infraestructura y entonces procederemos con el test en tiempo de ejecución.

7.Principales desafíos

En el desarrollo de este trabajo nos hemos encontrado con algunas cuestiones que han supuesto un esfuerzo adicional, bien por la poca documentación encontrada, o por las dificultades de aplicar los conocimientos adquiridos a un entorno real, el cual en ocasiones es poco flexible y no permite "atajos".

Veremos como implementar del protocolo *TLS* en cliente y servidor, y finalizaremos con el envío de archivos ejecutables/binarios incrustados en archivos *JSON*.

7.1 Protocolo TLS

Un elemento que es fundamental hoy en día es el de la seguridad en las comunicaciones, el número de ataques e intrusiones en sistemas es un factor creciente. Dentro de la rama de *Tecnologías de la información* se nos ha hecho hincapié en esta cuestión, señalando que la seguridad debe ser un elemento mas dentro del diseño de soluciones y por tanto debe tomarse en consideración desde el principio.

Para securizar las comunicaciones existen diversas formas y estrategias, según al nivel que trabajemos se aplicarán unas u otras, por ejemplo si estamos trabajando en la capa de red, utilizaríamos VPN's junto a protocolos que cifran los datagramas que se intercambian. En nuestro caso vamos a trabajar a nivel de transporte y más concretamente con *TCP* que es el protocolo que se utiliza con aplicaciones web. El protocolo más seguro – aunque no infalible – para la comunicación vía web es el protocolo *TLS.*

El protocolo *TLS* [13] – *Transport Layer Secutity* – proporciona seguridad de comunicaciones a través de Internet. El protocolo permite que las aplicaciones cliente-servidor se comuniquen de una manera que está diseñada para evitar la interceptación, la manipulación o la falsificación de mensajes. Tiene dos características básicas, ofrece privacidad en las comunicaciones y la conexión es confiable. El transporte de mensajes incluye una verificación de integridad del mensaje utilizando un resumen cifrado del mismo.

El uso de *TLS* afecta a la forma de realizar las peticiones, hay que utilizar el protocolo *HTTPS* [14] y colocarlo en lugar de *HTTP*. Así mismo cambia el número de puerto estándar, del 80 pasamos al 443.

Las cuestiones técnicas del funcionamiento del protocolo se salen del contexto de este trabajo con lo que nos limitaremos a ofrecer una explicación sencilla del mismo.

TLS se basa en el uso de certificados:

- El cliente accede mediante *HTTPS* a un sitio web.
- El servidor le envía un certificado digital.
- El cliente pues comprueba que ese certificado es auténtico a través de la clave pública de una autoridad certificadora o simplemente confía en él.

- Comienza entonces una negociación – en términos criptográficos – o *handshake* que dará como resultado el cifrado de todos los mensajes intercambiados (*figura 22*).

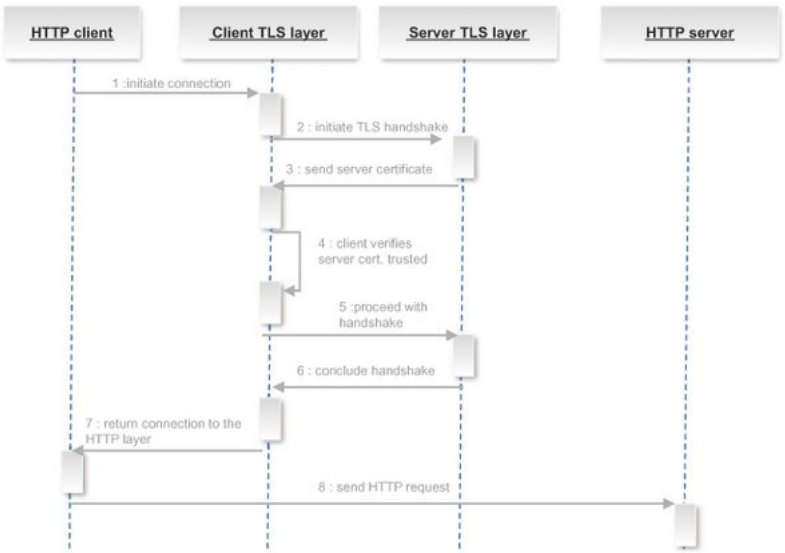


figura 22. Diagrama de secuencia en HTTPS

Con esto se establece una confianza mutua y se consigue privacidad en las comunicaciones.

TLS se usa casi por completo en páginas web, y es gestionado por los navegadores que implementan mecanismos de comprobación de certificados, o en su defecto avisan al usuario para que bajo su responsabilidad acepte el certificado.

El desafío que se plantea en nuestra solución es conseguir realizar está comprobación y posterior negociación de forma automática. La solución son los certificados 'autofirmados' que conseguiremos con las herramientas que proporciona *Java* y que implementaremos con *Restlet framework*.

7.1.1 Obtención de certificados

Para *TLS* se utilizan dos tipos de archivos, *keystores* – almacenes de claves – y *truststores* – almacenes de confianza – .

En este contexto, *keystore* es el archivo que tiene la información local de la aplicación. *Keystore* contiene el certificado y la clave privada de un servidor, o el certificado y la clave privada de un cliente. En contraste, *truststore* es el archivo utilizado para tomar decisiones de confianza con respecto a los certificados presentados por pares remotos. El cliente utiliza *truststore* para verificar los certificados de los servidores a los que se conecta y el servidor para verificar los certificados de clientes que recibe – si el servidor está configurado para solicitar certificados de clientes – .

7.1.1.1 Certificado autofirmado

Para obtener un certificado autofirmado utilizaremos *keytool* que es una herramienta de *Java* para la generación de claves. Los archivos *keystore* tiene formato *JKS – Java KeyStore –* .

La siguiente secuencia de ordenes creará un par de claves *2048-bit RSA* y luego lo almacenará en el alias del servidor de un archivo *keystore* llamado `serverKey.jks` .

```
keytool -keystore serverKey.jks -alias server -genkey -keyalg RSA -keysize 2048
        -dname "CN=inf.upv.es,OU=DSIC,O=The_MRS_team,C=ES" -sigalg "SHA1withRSA"
        -ext san=ip:127.0.0.1 -ext san=dns:localhost -ext san=ip:192.168.1.102
```

Este certificado puede luego exportarse como un archivo de certificado independiente ".crt", usando este comando:

```
keytool -exportcert -keystore serverKey.jks -alias server -file serverKey.crt
```

Ahora que ya tenemos el certificado, vamos a crear el archivo *truststore* para el cliente al que llamaremos `clientTrust.jks`. Debemos partir del archivo ".crt" del servidor para su creación:

```
keytool -import -keystore clientTrust.jks -trustcacerts -alias server -file serverKey.crt
```

En este momento ya disponemos de los dos archivos necesarios para implementar *TLS:*

- Servidor → *keystore* → `serverKey.jks`
- Cliente → *truststore* → `clientTrust.jks`

7.1.2 Implementación con *Restlet*

Una vez tenemos los archivos necesarios pasaremos a implementarlo en cliente y servidor.

7.1.2.1 Servidor *Restlet*

En nuestra solución la implementación se realizará en `AFI.restservice,` y más concretamente en la clase `RestActivator.java` (*figura 23*), los archivos necesarios están guardados en la carpeta 'keystore' que se encuentra localizada dentro del módulo .

```
private static final String keystorePath = "/AFI.restservice/keystore/serverKey.jks";

. . . . . .

this.component = new Component();
Server server = this.component.getServers().add(Protocol.HTTPS, port);

Series <Parameter> parameters = server.getContext().getParameters();
      parameters.add("keystorePath", WORKSPACE + keystorePath);
      parameters.add("keystorePassword","mrspass");
      parameters.add("keystoreType","JKS");
      parameters.add("keyPassword","mrspass");
```

figura 23. Fragmento de RestActivator.java

Ahora ya tenemos un servidor que ofrece servicios seguros mediante *HTTPS y TLS.*

7.1.2.2 Cliente *Restlet*

Para que el cliente pueda ahora acceder al servicio necesita utilizar el archivo *truststore* antes creado, esto se va a llevar a cabo en la clase `RestletClient.java` que se encuentra en `AFI.Automovil.restlet` *(figura 24),* el archivo necesario está guardado en una carpeta 'truststore' dentro del módulo.

```
private static final String keystorePath = "/AFI.Automovil/truststore/clientTrust.jks";

. . . . . .

this.client = new Client(new Context(), Protocol.HTTPS);

Series<Parameter> parameters = client.getContext().getParameters();
       parameters.add("truststorePath", WORKSPACE + keystorePath);
       parameters.add("truststorePassword", "mrspass");
       parameters.add("truststoreType", "JKS");
```

figura 24. Fragmento de RestletClient.java

Después de haber realizado estos pasos, el servicio *REST*ful y la aplicación podrán comunicarse de una forma segura, garantizando la confiabilidad, la integridad y la confidencialidad en el transporte de los datos.

7.2 Envío de archivos binarios como un campo *JSON*

La característica fundamental del repositorio de componentes software es el intercambio de archivos ejecutables entre cliente y servidor, lo que plantea la duda de como enviarlos y como tratarlos cuando se reciben.

Para afrontar esta cuestión se han barajado dos posibilidades, la que ofrece *Restlet framework* y la de seguir utilizando archivos *JSON* al igual que en el repositorio de modelos y configuraciones.

7.2.1 Dos alternativas posibles

Restlet se basa en los procedimientos estándar que se utilizan en *HTTP / HTTPS*, lo que es recomendable. Después de realizar un par de pruebas vimos que funcionaba bien pero resultaba poco flexible, ya que a cada tipo de archivo hay que explicitar su formato y tipo y gestionarlos según el tipo de contenido en el lado el servidor. Básicamente adjunta archivos a un formulario.

Si nuestra solución hubiera tratado con varios formatos y tipos esta opción hubiera sido la adecuada, pero no es nuestro caso – sólo trabajamos con ".jar" – . Necesitábamos algo que aprovechara el formato que habíamos estado empleando y que no condicionara la estructura y los métodos ya creados.

Los campos *JSON* sólo admiten el tipo `String` – cadena de texto – para definirlos, entonces la duda era como convertir un archivo *JAR* en un `String`. Al parecer esto que planteamos no es muy 'habitual' y no está está documentado – al menos hasta donde ha llegado nuestra búsqueda – con lo que centramos la investigación en experiencias previas de desarrolladores que comparten sus conocimientos a través de foros o páginas especializadas.

La solución que encontramos y que utilizan algunos desarrolladores profesionales es sencilla y simple, se trata de la siguiente:

Tenemos un archivo cualquiera (desde .txt a .zip) lo pasamos a un `byte[] array` – o matriz de bytes – , esto lo hacemos con `FileInputStream` y `FileOutputStream` que son clases de *Java-SE*.

Una vez tenemos el *array* lo codificamos en *Base64* como un objeto *MIME*, que es el tipo de archivo con el que trabaja *HTTP/HTTPS* de forma estándar para el envío de objetos (figura 25).

```
Base64.Encoder mimeEncoder = Base64.getMimeEncoder();

String encode = mimeEncoder.encodeToString(byte[]);
```

figura 25. Transformación de la matriz de bytes a objeto MIME tipado como String.

Ahora ese `String` lo podemos guardar en un campo de un archivo *JSON* y enviarlo. Así mismo una vez se tiene como un `String` en *Base64*, se puede cifrar con un algoritmo más potente (p.ej. *RSA*) y darle un plus de seguridad.

Después en destino simplemente hay que hacer las operaciones a la inversa:

`ModelRepository.java` llama a `ManagementFiles.saveAsBinaryFile()` (*figura 26*)

```
Base64.Decoder decoder = Base64.getMimeDecoder();

byte[] decodedByteArray = decoder.decode(theModel.toString());
```

figura 26. Fragmento de ManagementFiles.saveAsBinaryFiles

Finalmente con `FileOutputStream` reconstruimos el archivo a partir del `byte[] array` y habremos completado el envío y la recepción de un archivo binario dentro de un campo *JSON*.

7.2.2 Elección del procedimiento

Para tomar la decisión final nos planteamos las siguientes cuestiones que relatamos a continuación.

Tenemos dos casos:

1. Enviar un único documento (un *JSON*) y empotrar el binario ‘dentro’ de estos *JSON* (junto con los metadatos del archivo) → codificar en *Base64*

2. Enviar varios documentos (en un formulario múltiple), uno de los cuáles sería el *JSON* y el otro el binario → *Restlet framework*

¿Qué sería más estándar, o menos dependiente de tecnologías?

Caso 1: Se puede implementar con las librerías que van en *Java-SE*, con lo que las dependencias a librerías externas son nulas.

Caso 2: Para implementarlo es necesario importar estos paquetes extra:

en servidor → *apache.org.commons.fileupload, apache.org.commons.fileupload.disk, restlet.ext.fileupload* como dependencias.

en cliente → *restlet.ext.OSGi, restlet.ext.html,* igualmente como dependencias.

En esta solución se utilizan las cabeceras *HTTP* típicamente usadas para html, resulta obvio que es más estándar, pero a la vez más dependiente.

¿Qué sería más extensible si se quisieran añadir nuevos campos?

Caso 1: Se pueden añadir los campos que se deseen, simplemente hay que crear un *JSONObject* con el binario e introducir este en un *JSONArray* que será enviado al servidor. Con esta opción se puede poner más información adjunta al binario que con la opción 2, ya que con esta última sólo enviamos el objeto en si.

Caso 2: Igualmente se pueden añadir cuantos campos se quieran, *Restlet* no ofrece restricciones en ese aspecto. El contenido del objeto se compone únicamente del ejecutable.

¿Qué sería más portable o adaptable a otros sistemas operativos?

Estarían a la par, el fundamento de las dos soluciones es el mismo,se convierten en una matriz de bytes, se envían y se reconstruyen en destino. Al trabajar a bajo nivel, el sistema operativo no tiene especial relevancia, otra cosa es que al reconstruirlo se quiera ejecutar, entonces si dependerá del sistema operativo.

¿Qué es más eficiente?

Caso 1: El trabajo "extra" lo tendríamos a la hora de codificar y decodificar en *Base64*, pero al ser un algoritmo de codificación simple este trabajo es despreciable.

Caso 2: En esta solución hay que crear un formulario, procesar los distintos archivos y adjuntarlos con lo que ese trabajo es igualmente despreciable.

Conclusión: La opción idónea es el caso 1.

Si implementamos esta solución los cambios en `ModelRepository.java` serían minimos. Con la 2 hay que revisar todos los métodos debido a lo que llega al servidor – `InputStream` – . Igualmente con la solución 2 es innecesario la creación del archivo *JSON* a enviar, pero los metadatos del archivo si son necesarios , con lo que enviar un *JSON* con información del binario y el binario por separado sería poco eficiente.

También está el hecho de que en la 2 hay que especificar en cada campo el tipo de contenido. En la 1 siempre es el mismo, como mucho podríamos añadir un campo en *JSON* con la extensión del

archivo para que visualmente el sistema operativo mostrara el tipo de archivo. En cualquier caso el tipo de archivo está descrito en los primeros bytes de la matriz.
Pensamos que una solución simple si funciona resulta más robusta que otra que sea más compleja.

Una vez tomamos la decisión procedimos a su implementación, adecuamos el código que teníamos escrito para que los modelos y configuraciones fueran tratados de igual manera. Para identificarlos en destino, añadimos el campo ‘extension’ dentro del archivo *JSON* que proporcionará de una manera sencilla flexibilidad a la hora de tratar distintos archivos.

A continuación mostramos un mensaje *JSON* de intercambio en el que el ejecutable SWC está definido como una cadena de texto:


```
{   "SWC":"UEsDBAoAAAgAAHpcs04AAAAAAAAAAAAAAAAJAAQATUVUQS1JTkYv/soAAFBLAwQKAAAICAB5XLNO\r\
nn+2gHw4BAAAcAgAAFAAAAE1FVEEtSU5GL01BTklGRVNULk1GdVHBToNAEL2T8A+kZ9lAL1aMB0ww\r\
nqQnGtIn3cRnIWtipswvWv3eBitDU2+S9t2/evM1BqxKNDd+QjSKdBLGIfO+x1UWNYX5mJ3I9US/Q\r\
nYBLsHGmQOyVxYvbfzTvVSo6K9Gkr+Ipqvq/fmLaWGrBKhjkV7bTgv+c7/GwVY5GdULbWGWW6U0y6\r\
nQW2T4Bk62GdhLDa+t22OxDZ8BXmAyjlCqYTSFrkEieaGuBIfhvR9NwZ6WK2jaBPdRkOu1cCTqXwv\r\
nUKJkl+mL+PAnjsXdr6qPWaOdz6IACwvA+eDJDhsXOOPRjS489MdcUIZalnjpw+QO19UCPteOwJyq\r\
nnnpLpVUdWOKxiFm1ov/LifY93/sBUEsBAhQDCgAACAAAelyzTgAAAAAAAAAAAAAAAAkABAAAAAAA\r\
nAAAQAO1BAAAAAE1FVEEtSU5GL/7KAABQSwECFAMKAAAICAB5XLNOn+2gHw4BAAAcAgAAFAAAAAAA\r\
nAAAAAAAApIErAAAATUVUQS1JTkYvTUFOSUZFU1QuTUZQSwUGAAAAAAIAAgB9AAAAawEAAAA" }
```


Respuesta a una petición de componente software a https://192.168.1.102:8183/component/AFI.restservice/1.0.1

8.Conclusiones

En este trabajo se han creado dos prototipos de servicio en forma de repositorios de modelos y componentes software. Los objetivos que nos planteamos alcanzar han sido cumplidos, dotando al desarrollo de sistemas autoadaptativos de una flexibilidad y modularidad que antes no tenía.

El hecho de hacer independiente el conocimiento obtenido por la aplicación de la propia aplicación, permite a los desarrolladores poder realizar las modificaciones o actualizaciones que consideren tanto en el momento del diseño como en tiempo de ejecución sin tener que parar el sistema.

Hemos logrado construir un servicio web que, en combinación con los repositorios, hará que la aplicación se independice totalmente del conocimiento, pudiendo además realizar una gestión remota. Distintas aplicaciones en distintos lugares podrán conectarse al repositorio a través del servicio web y compartir así sus conocimientos y experiencias. De igual modo la gestión de posibles modificaciones o actualizaciones se simplifica, el administrador sólo deberá realizar la acción una vez sobre el repositorio, luego será el servicio web quien se encargue de entregarlas bajo petición.

Hemos incluido en la solución una característica que si bien recientemente está cobrando protagonismo es la gran olvidada en la fase de diseño, esto es, la seguridad. Diseñar el servicio web con *TLS*, nos ha permitido dotar a las comunicaciones de privacidad, integridad y confiabilidad. Esto a condicionado también el resto del diseño, teniendo que crear algunos métodos específicos para tratar el protocolo. De haber tenido que hacerlo después, y como se ha dicho hoy en día es casi imprescindible, nos hubiéramos encontrado con no pocos problemas.

Al haberlo realizado con certificados autofirmados, tenemos la seguridad de que los participantes en las comunicaciones están gestionados por nosotros. Una aplicación cliente a la que no le hayamos introducido personalmente el certificado de confianza no tendrá acceso al repositorio.

Otra de las características a resaltar es el envío de ejecutables dentro de un campo *JSON* , con esta solución se ha podido crear un único tipo de mensaje para todas las comunicaciones. Toda la información se envuelve en un archivo *JSON* sea cuál sea el contenido y es eso lo que se intercambia, después son los extremos los que desenvuelven e interpretan. El hecho de tener que utilizar mas de un formato de mensaje en un servicio web, complica la implementación. Lo que se pretende cuando se crean estos servicios es que sean los más homogéneos posible a la hora de intercambiar datos.

Con la nueva implantación del 5G, estos repositorios remotos cobrarán gran protagonismo, sobre todo en vehículos de conducción autónoma en los cuáles se podrán consultar configuraciones, realizar actualizaciones o compartir conocimientos con otros vehículos a través de la web casi en tiempo real.

En lo personal este trabajo me ha servido para aplicar los conocimientos obtenidos durante el grado y en especial aquellos relacionados con las *Tecnologías de la información*.También la investigación me ha resultado interesante, la búsqueda de soluciones a los problemas que han ido surgiendo y su posterior consecución han supuesto una gran motivación para mi. He aprendido que la solución no siempre tiene un sólo camino. Estoy seguro que esta experiencia me será útil en el futuro.

Relación del trabajo desarrollado con los estudios cursados

Este trabajo me ha servido para aplicar los conocimientos obtenidos durante el grado. La base que he adquirido en el área de redes a todos los niveles, me ha permitido desarrollar el trabajo casi de una forma intuitiva. Los conceptos que he visto en los libros o artículos que he consultado me eran todos familiares con lo que me he podido enfocar más rápidamente en la solución que proponían.

Así mismo todos los conceptos vistos en relación a sistemas distribuidos que se ven en segundo y tercero me han sido de gran utilidad y sin los fundamentos en sistemas operativos, la tarea del diseño de la estructura definida de directorios, así como el tratamiento de archivos hubiera sido imposible.

Para realizar este documento me he basado en el proceso de creación que nos enseñaron en 'Ingeniería del software' – asignatura de 3er curso – , no he utilizado el procedimiento estricto pero sí una adaptación que me ha servido mucho.

Por supuesto el uso de *Java* durante todo el grado me ha sido muy útil, la comprensión en la lectura del código de otros desarrolladores, así como el manejo de librerías y paquetes tan presentes en este proyecto no han resultado un problema.

En mi humilde entender pienso que he sacado partido a estos cuatro años en la escuela, me siento confiado para salir al mundo laboral y no 'morir en el intento'.

9.Referencias bibliográficas

[1] *Computing machinery and intelligence.Resumen.*
Disponible en: https://es.wikipedia.org/wiki/Computing_machinery_and_intelligence

[2] *Restful. Definición.*
Disponible en: https://en.wikipedia.org/wiki/Representational_state_transfer

[3] Fielding, T.R., 2000. *Architectural Styles and the Design of Network-based Software Architectures.Capítulo 5.*
Disponible en: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[4] Knoernschild, K., 2012. *Java Application Architecture: Modularity Patterns with Examples Using OSGi*. Published by Addison-Wesley Professional. ISBN 978-0-321-24713-1 .

[5] Louvel, J., Templier, T. y Boileau, T. , 2012. *Restlet in Action: Developing RESTful web APIs in Java*. Published by Manning Publications. ISBN 978-1-935-18234-4 .

[6] *Virtualbox. Virtualización de sistemas operativos.*
Disponible en: https://www.virtualbox.org/wiki/Virtualization

[7] IBM Corporation, 2005. *An architectural blueprint for autonomic computing.* Whitepaper published by IBM Software Group.

[8] Kephart, J. O. and Chess ,D. M., 2003. *The vision of autonomic computing*. Publisheb by the IEEE Computer Society. Ref. 0018-9162/03

[9] Brun,Y., Di Marzo Serugendo,G., Gacek,C., Giese,H., Kienle,H., Litoiu,M., Müller,H., Pezzè,M., y Shaw, M., 2009. *Engineering Self-Adaptive Systems through Feedback Loops*. Artículo académico.

[10] Arcaini,P., Riccobene,E., Scandurra,P., 2015. *Modeling and Analyzing MAPE-K Feedback Loops for Self-adaptation*. Artículo académico.

[11] Thomas, E., Carlyle,B., Pautasso,C. y Balasubramanian,R., 2012. *Principles, Patterns & Constraints for Building Enterprise Solutions with REST*. Prentice Hall.
ISBN 978-01-328-69904 .

[12] Kreger,K., 2001. *Web Services Conceptual Architecture (WSCA 1.0)*. Published by IBM Software Group.

[13] *RFC 2616 HTTP 1.1* . Request for comments by The Internet Society (1999).
Disponible en: https://tools.ietf.org/html/rfc2616

[14] *RFC 5246 The Transport Layer Security (TLS) Protocol Version 1.2 ,* 2008. Request for comments refer to Internet standards track protocol.
Disponible en: https://tools.ietf.org/html/rfc5246

[15] *RFC 2818 HTTPS Over TLS.* Request for comments by The Internet Society (2000).
Disponible en: https://tools.ietf.org/html/rfc2818

Acrónimos

Término	Definición
OSGi	Open Services Gateway initiative
MAPE-K	Monitor, Analize, Plan, Execute, Knowledge
JSON	JavaScript Object Notation
JAR	Java ARchive
XML	eXtensible Markup Language
REST	Representational State Transfer
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
CM	Component Model
SCM	System Configuration Model
SWC	Software Component
CRUD	Create, Read, Update, Delete
UML	Unified Modeling Language
TLS	Transport Layer Security
JKS	Java KeyStore
TCP	Transmission Control Protocol
MIME	Multipurpose Internet Mail Extensions
HTTP	HyperText Transport Protocol
HTTPS	HyperText Transport Protocol over TLS

Apéndice

En este apéndice se encuentran todas las clases involucradas en la solución. Se muestran con los métodos que las componen y la función que desempeñan. Se encuentran ordenadas por el módulo al que pertenecen de los cuatro que hay desarrollados.

AFI.interfaces

Operación	IModelRepository.java	
	CM	SCM
Create	`saveComponentModel ()`	`saveSystemConfigurationModel ()`
Read	`getComponentModel ()`	`getSystemConfigurationModel ()`
Update	`saveComponentModel ()`	`saveSystemConfigurationModel ()`
Delete	`deleteComponentModel ()`	`deleteSystemConfigurationModel()`

Operación	IComponentRepository.java
	SWC
Create	`saveSoftwareComponent ()`
Read	`getSoftwareComponent ()`
Delete	`deleteSoftwareComponent ()`

AFI.Automovil

AFI.Automovil.restlet

`RestletClient.java`

Operación	Método	Función
Constructor	`RestletClient()`	Se conecta al servidor Restlet mediante una IP bien conocida
GET/DELETE	`sendGetOrDel()`	Realiza una operación según el valor del parámetro ‘método’
PUT	`sendToServer()`	Envía archivos hacia el servidor

AFI.Automovil.automovil

`Activator.java`

Operación	Método	Función
Activar *bundle*	`start()`	Pone en marcha el *bundle*. Realiza todas las acciones de módulo – envío y recuperación de archivos –
Parar *bundle*	`stop()`	Detiene la ejecución del *bundle*
Create	`sendComponent()`	Construye un archivo *JSON* con los campos necesarios para su envío. Busca un archivo CM o SWC dentro de la estructura y lo añade al archivo *JSON*.
	`sendSoftwareComponent()`	
	`sendSystemConfigurationModel()`	Construye un archivo *JSON* con los campos necesarios para su envío, para ello busca un archivo SCM dentro de la estructura y lo añade. Busca en el registro el componente asociado y lo añade al archivo.El archivo resultante será el que se envíe.
Update	`updateSystemConfigurationModel()`	Construye un archivo *JSON* con los campos necesarios para su envío.
	`updateComponentModel()`	Construye un archivo *JSON* con los campos necesarios para su envío.
Read	`N/A`	Utiliza directamente `RestletClient.java`
Delete	`N/A`	Utiliza directamente `RestletClient.java`

`ManagementFiles.java`

Método	Función
`convertFile()`	Convierte cualquier archivo en una cadena de caracteres codificados en *Base64*. Se apoya en `encodeFile()`.
`encodeFile()`	Codifica el array de bytes pasados por `convertFile()` y los codifica en *Base64*.
`readFile()`	Pasa a cadena de caracteres archivos *JSON*.

AFI.repository

`Activator.java`

Operación	Método	Función
Activar *bundle*	`start()`	Pone en marcha el *bundle*. Se registra el servicio en *OSGi*.
Parar *bundle*	`stop()`	Detiene la ejecución del *bundle*

`ModelRepository.java`

Operación	Método	Función
Create	`saveComponentModel()`	Almacena en la estructura de directorios el archivo pasado desde la aplicación.
	`saveSystemConfigurationModel()`	
Read	`getComponentModel()`	Busca un archivo CM dentro de la estructura y devuelve su contenido.El contenido se envia en un archivo *JSON*.
	`getSystemConfigurationModel()`	Busca un archivo SCM dentro de la estructura y devuelve su contenido . Busca en el registro el componente asociado y lo añade al contenido. El contenido se envia en un archivo *JSON*.
Update	`saveComponentModel()`	Actualiza un modelo de componente.
	`managementUpdates()`	Determina el tipo de operación –Create/Update –.
	`updateSystemConfigurationModel()`	Busca un archivo dentro de la estructura y actualiza la configuración del sistema según versión.
	`updateConfigurationModel()`	Busca dentro del registro el modelo de componente asociado a una configuración y lo actualiza.
Delete	`deleteComponentModel()`	Elimina un modelo de componente.
	`deleteSystemConfigurationModel()`	Elimina una configuración del sistema.

`ComponentRepository.java`

Operación	Método	Función
Create	`saveSoftwareComponent()`	Almacena en la estructura de directorios el archivo pasado desde la aplicación
Read	`getSoftwareComponent()`	Busca un archivo SWC dentro de la estructura y devuelve su contenido en un archivo *JSON*
Delete	`deleteSoftwareComponent()`	Elimina un componente software

AFI.repository.utils

`ManagementFiles.java`

Método	Función
`saveAsBinaryFile()`	Almacena un archivo en el repositorio.
`saveJsonFile()`	Almacena un archivo *JSON* (registro) en el repositorio.
`readFile()`	Lee y convierte a `String` cualquier archivo.
`getFilefromRepository()`	Busca y devuelve el modelo de componente asociado a una configuración.
`convertFile()`	Convierte cualquier archivo en una cadena de caracteres codificados en *Base64*. Se apoya en `encodeFile()` .
`encodeFile()`	Codifica el array de bytes pasados por `convertFile()` y los codifica en *Base64*.

`ManagementJSON.java`

Método	Función
`createCMJsonToSend()`	Crea el archivo *JSON* que será devuelto como respuesta a una petición de modelo de componente CM.
`createSCMJsonToSend()`	Crea el archivo *JSON* que será devuelto como respuesta a una petición de configuración del sistema SCM.
`createJsonToSend()`	Crea el archivo *JSON* que será devuelto como respuesta a una petición de componente software SWC.

`ManagementRegisterMRS.java`

Método	Función
`createCMRootRegister()`	Crea el archivo *JSON* inicial de registro para modelos de componente CM.
`registerJSONCM()`	Gestiona el registro para modelos de componente CM.
`saveRegisterJsonCM()`	Guarda el archivo de registro en la estructura de directorios después de cada actualización.
`createSCMRootRegister()`	Crea el archivo *JSON* inicial de registro para configuraciones del sistema SCM.
`registerJSONSCM()`	Gestiona el registro para configuraciones del sistema SCM.
`saveRegisterJsonSCM()`	Guarda el archivo de registro en la estructura de directorios después de cada actualización.
`getAssociatedCMAsString()`	Devuelve el modelo de componente asociado a una configuración como un `String`
`updateJSON()`	Actualiza registro de entrada tanto para CM como para SCM.
`updateAfterDeleteCM()`	Actualiza el registro tras un borrado de modelo de componente CM.
`updateAfterDeleteSCM()`	Actualiza el registro tras un borrado de modelo de componente SCM.
`getJSONArray()`	Recupera el registro de la estructura de directorios para poder ser tratado.

`ManagementRegisterCRS.java`

Método	Función
`createRootRegister()`	Crea el archivo *JSON* inicial de registro para componentes software SWC.
`registerJSON()`	Gestiona el registro para componentes software SWC.
`saveRegisterJson()`	Guarda el archivo de registro en la estructura de directorios después de cada actualización.
`updateAfterDeleteCM()`	Actualiza el registro tras un borrado de un componente software SWC.
`getJSONArray()`	Recupera el registro de la estructura de directorios para poder ser tratado.

AFI.restservice

`RestActivator.java`

Operación	Método	Función
Activar *bundle*	`start()`	Pone en marcha el *bundle*.
Parar *bundle*	`stop()`	Detiene la ejecución del *bundle*.
PUT	`saveComponentModel()`	Usa los métodos de la interfaz correspondiente para comunicarse con el repositorio.
	`saveSystemConfigurationModel()`	
	`saveSoftwareComponent()`	
GET	`getComponentModel()`	Usa los métodos de la interfaz correspondiente para comunicarse con el repositorio.
	`getSystemConfigurationModel()`	
	`getSoftwareComponent()`	
DELETE	`deleteComponentModel()`	Usa los métodos de la interfaz correspondiente para comunicarse con el repositorio.
	`deleteSystemConfigurationModel()`	
	`deleteSoftwareComponent()`	

`MRServerApplication.java`

Método	Función
`MRServerApplication()`	Constructor
`createInboundRoot()`	Crea los recursos que serán referenciados posteriormente.

AFI.restservice.logic

`RootServerResource.java`

Método	Función
`RootServerResource()`	Constructor
`doInit()`	Devuelve un mensaje de inicio del servicio.
`doCatch()`	Testea las excepciones y devuelve un mensaje si las hay.
`doRelease()`	Devuelve un mensaje de inicio de los recursos.
`get()`	Devuelve un mensaje de bienvenida al usuario.

`MRSComponentModel.java`

Operación	Método	Función
GET	`getComponentModel()`	Recibe una petición vía web, recoje los parámetros de la consulta y los pasa al repositorio. Espera la respuesta del repositorio y envía el contenido de ésta utilizando la misma vía de la petición.
PUT	`saveComponentModel()`	Recibe un archivo CM vía web, el cual pasa al repositorio con los metadatos del archivo. Enviará una respuesta en forma de código que reflejará el resultado de la operación – éxito o fracaso – .
DELETE	`deleteComponentModel()`	Recibe una petición vía web y transfirere los parámetros pasados al repositorio. Enviará una respuesta en forma de código indicando el resultado de la operación.

`MRSConfigurationModel.java`

Operación	Método	Función
GET	`getSystemConfigurationModel()`	Recibe una petición vía web, recoje los parámetros de la consulta y los pasa al repositorio. Espera la respuesta del repositorio y envía el contenido de esta utilizando la misma vía de la petición.
PUT	`saveSystemConfigurationModel()`	Recibe un archivo SCM vía web, el cual pasa al repositorio con los metadatos del archivo. Enviará una respuesta en forma de código que reflejará el resultado de la operación – éxito o fracaso – .
DELETE	`deleteSystemConfigurationModel()`	Recibe una petición vía web y transfirere los parámetros pasados al repositorio. Enviará una respuesta en forma de código indicando el resultado de la operación.

`CRSSoftwareComponent.java`

Operación	Método	Función
GET	`getSoftwareComponent()`	Recibe una petición vía web, recoje los parámetros de la consulta y los pasa al repositorio. Espera la respuesta del repositorio y envía el contenido de esta utilizando la misma vía de la petición.
PUT	`saveSoftwareComponent()`	Recibe un archivo SWC vía web, el cual pasa al repositorio con los metadatos del archivo. Enviará una respuesta en forma de código que reflejará el resultado de la operación – éxito o fracaso – .
DELETE	`deleteSoftwareComponent()`	Recibe una petición vía web y transfirere los parámetros pasados al repositorio. Enviará una respuesta en forma de código indicando el resultado de la operación.