UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Documentación automática de API basadas en peticiones SOAP

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Roig Cerdá, Pablo

Tutor/a: Letelier Torres, Patricio Orlando

CURSO ACADÉMICO: 2023/2024

A mi profesor de Tecnología Gonzalo Tormo Lillo, que me hizo enamorarme de la informática y la ingeniería.

A mis padres, que siempre me han apoyado con mis decisiones, a la hora de estudiar y de trabajar.

Y finalmente, a mis amigos, que han estado conmigo en los mejores años de mi vida.
Lok'tar Ogar.

Agradecimientos

Me gustaría agradecer a la empresa, a mi tutor de empresa Jose Carabaño y a mi tutor de TFG Patricio Letelier por permitirme realizar no solo mis prácticas, sino este TFG.

Este trabajo no sería posible tampoco sin la ayuda de mi amigo Alejandro Barceló, que me ha ayudado en numerosas ocasiones. Sin él, dudo que me hubiera lanzado a realizar estas prácticas; sin duda me habría olvidado de presentar el resumen del TFG; y definitivamente me hubiera hundido tratando de salvar un proyecto perdido.

Resumen

Una API (Interfaz de Programación de Aplicaciones) es una pieza de código que permite a diferentes aplicaciones comunicarse entre sí y compartir información y funcionalidades. Es conveniente que los métodos expuestos por esta API estén documentados, preferiblemente con peticiones detalladas que se ofrezcan como ejemplo. El objetivo de este TFG es facilitar las tareas de documentación de la API, automatizando en parte dichas tareas mediante una aplicación que accederá a la información necesaria para completar la documentación de las peticiones. Este trabajo se ha desarrollado en el marco de una práctica de empresa, en el departamento de desarrollo donde se realiza el mantenimiento de su producto, un ERP para el sector sociosanitario, con una API que acepta peticiones SOAP.

Palabras clave: .NET, API, SOAP, XML, documentación.

Abstract

An API (Application Programming Interface) is a piece of code that allows different applications to communicate with each other and share information and functionality. It is desirable that the methods exposed by this API be documented, preferably with detailed requests provided as examples. The objective of this TFG is to facilitate API documentation tasks, partly automating these tasks through an application that will access the information necessary to complete the documentation of the requests. This work has been developed within the framework of a company practice, in the development department where the maintenance of its product is carried out, an ERP for the social and healthcare sector, with an API that accepts SOAP requests.

Keywords : .NET, API, SOAP, XML, documentation.

Tabla de contenidos

Tabla de figuras

1. Introducción

1.1 Motivación

La documentación de una API es un proceso necesario para que sea utilizada por externos e integradores. Es vital que la documentación sea completa y de calidad.

Una API (*Application Programming Interface*) es, esencialmente, un conjunto de métodos que exponemos al público (a otro programa) para que se integre en otro entorno. Puede ser de varios tamaños, ya sea para un subsistema concreto, como puede ser la autenticación de Google, o un sistema más grande, como es el caso del ERP.

Muchas de estas API utilizan SOAP, o Simple Object Access Protocol, que es un protocolo que se utiliza para el traspaso de objetos entre programas diferentes, utilizando REST para el transporte.

No obstante, para este proceso resulta necesario un conocimiento avanzado de la API que se está documentando, a fin de conocer los cuerpos de las peticiones que se rellenan para crear la documentación final, además de baterías de pruebas y más. Por lo general, estos cuerpos son excesivamente complejos y extensos, y requieren de muchas horas de trabajo para entender y rellenar.

Este trabajo tiene como motivación crear una herramienta que facilite el proceso de documentación de una API SOAP[1]. Para hacer esto, apoyará al ciclo manual de documentación generando las peticiones, datos y estructuras relacionadas. También se incluirán otras herramientas que apoyan al proceso, como limpiadores o acortadores de peticiones.

La empresa comercializa un producto ERP sociosanitario, con diversas otras componentes, como versiones para tablets, y lo que es relevante para nosotros, una API para que sea integrada en otros programas.

Esta API, junto con el resto del programa, pasa por una serie de baterías de pruebas para verificar el comportamiento correcto entre versiones a modo de pruebas de regresión. Para esto, se hacen baterías de peticiones SOAP mediante el programa SOAPUI[2], donde se crean Test Suites que incluyen:

-Una petición por método

-Una comprobación Regex por cada método

-Todo esto va sujeto a que también exista una imagen de la base de datos poblada con los datos necesarios para ejecutar las pruebas.

[1] https://www.w3.org/TR/soap12-part1/Overview.html
[2] https://www.soapui.org

La estructura del ámbito de testing para la API son uno o varios clientes que lanzan las suites (solo es necesario 1), y una máquina virtual que es la que aloja los datos, y que tiene diversas imágenes (snapshots) y Scripts SQL asociados para restaurar un estado concreto para las pruebas.

Cada cliente hace uso de una copia de las Test Suites hechas con SOAPUI y lanzadas desde el mismo, que nos indica según las comprobaciones con Expresiones Regulares si ha sido el resultado esperado o no.

Hay una Test Suite para cada servicio documentado, y tenemos unos 6 servicios documentados con pruebas automatizadas (unas 400 pruebas). El porcentaje de éxito en las pruebas oscila el 95% cada mes que se actualiza la API, con fallos menores por alguna Expresión Regular que hay que cambiar o algún cambio en la base de datos o en alguna petición.

El proceso de documentación de la API engloba:

- Rellenar la petición
- Crear el Regex que comprueba la ejecución correcta
- Poblar la base de datos
- Redactar la propia documentación

La sección que más tiempo toma es la de rellenar la petición. Hay excepciones, por supuesto, pero normalmente la base de datos no cuenta con la información necesaria para directamente realizar las peticiones. Lo normal es que se tengan tres tipos de peticiones:

- Añadir
- Borrar
- Obtener / modificar

Lo ideal sería empezar por el método de añadir para poblar la base de datos y luego documentar el resto de métodos. No obstante, esto no es nada fácil en la mayoría de casos, porque suelen tomar como argumentos construcciones de datos (DTO) que a su vez pueden (y suelen) tener más DTO dentro, y acabamos con una petición excesivamente larga que esencialmente es imposible de rellenar.

Estamos hablando de peticiones con un espacio en disco de 30 MB, 400.000 líneas, y que ni siquiera se pueden abrir en un editor de texto porque este deja de responder. Además, es posible que ni siquiera se pueda guardar el proyecto de SOAPUI una vez creada una de estas peticiones, porque se queda sin espacio para el heap y la Máquina Virtual de Java deja de funcionar.

Con todo esto podemos empezar a ver los numerosos problemas que existen a la hora de documentar no solo una API que es desconocida para el usuario que la está documentando, sino a un usuario experimentado que trate de hacer su trabajo en un tiempo razonable.

Debido a todos estos problemas, sería conveniente tener una herramienta que ayude a rellenar estas peticiones, con el objetivo de aligerar la carga del trabajador, con herramientas para limpiar las peticiones para ponerlas en la documentación en vez de hacerlo manualmente, para revisar los componentes de la petición en el código fuente sin tener que buscarlos, para sugerir valores para cada campo, para editar más cómodamente el archivo XML, para
reducir las peticiones grandes que hemos comentado antes, o para regenerar peticiones parciales

1.2 Objetivos

Este TFG tiene por objetivo desarrollar una herramienta que permita solucionar o aliviar algunos de los problemas anteriormente comentados. Las finalidades concretas son:

- Hacer posible el manejo de peticiones grandes, borrando los comentarios y las secciones innecesarias, además de los cuerpos de DTO que no sean necesarios.
- Regenerar peticiones a partir de una incompleta, usando la información de los DTO proporcionada por la empresa para este fin.

- Limpiar peticiones de comentarios y darles formato tabulado.

- Proporcionar valores ejemplo para los campos de una petición proporcionada por el usuario.

- Tener una herramienta para editar XML de forma más cómoda, con navegación entre campos, selección de contenido y más.

- Proporcionar una manera fácil y sencilla de integrar nuevos generadores de datos personalizados a las necesidades de cada servicio a documentar. Esto es porque ciertos campos en ciertos servicios deseamos que tengan valores diferentes aunque el nombre sea igual.

1.3 Estructura del trabajo

La organización de este documento está compuesta por los siguientes capítulos:

- **Capítulo 2: Pruebas de regresión, Test Suites y Documentación de API**: está formado por una introducción en detalle al ámbito de la documentación de API, pruebas de regresión de software, y la organización en Test Suites.
- **Capítulo 3: Estado del arte**: una visión de las diferentes herramientas del mercado relacionadas con la documentación, así como el proceso actual.
- **Capítulo 4: Tecnologías utilizadas**: detalla el lenguaje de programación, editores, librerías y frameworks utilizados en el desarrollo de la herramienta.
- **Capítulo 5: Desarrollo de la solución:** expone el diseño de la solución, y los retos y soluciones que se han creado para formar el producto final.

- **Capítulo 6: Conclusiones y trabajo futuro**: una retrospectiva del proyecto en su totalidad, con un énfasis en lo que se ha conseguido y lo que podría desarrollarse en el futuro.
- **Capítulo 7: Referencias**: registro de las fuentes de información consultadas para la redacción de este trabajo.

2. Pruebas de regresión, Test Suites y Documentación de API

Las pruebas de regresión son conjuntos de pruebas que se realizan al final de un ciclo de desarrollo o Sprint, y que tratan de verificar que el funcionamiento de la herramienta se corresponde con su comportamiento especificado, es decir, si la herramienta hace lo que se supone que tiene que hacer [1].

La utilidad de las pruebas de regresión está en ejecutarlas cada vez que se completa un ciclo, de forma acumulativa, de manera que cada vez que se modifica un programa y se prueba por completo, tanto las pruebas nuevas como las de regresión deberían ser correctas, asegurándonos así de que el comportamiento no se ha modificado desde la última vez que se verificó, es decir, sigue siendo el correcto.

Estas pruebas se suelen agrupar o clasificar en Test Suites, o conjuntos de pruebas. Normalmente se agrupan por servicios, pantallas, o último resultado, aunque hay otros. De esta manera las tenemos organizadas, y podemos lanzarlas según queramos probar ciertos aspectos del programa.

Además, estas pruebas de regresión se pueden automatizar, y son en general mucho más eficientes en cuanto a tiempo para verificar el comportamiento del programa.

Dependiendo de la dimensión de la prueba, se pueden tener diferentes tipos de pruebas:

- Pruebas unitarias (*unit testing*): verifica el comportamiento de una parte concreta del código, como un método.

- Pruebas de integración: verifica la ejecución de diversos componentes del programa entre sí, en el ámbito de programa o subsistema.

- Pruebas de sistema (PS): verifica un comportamiento sobre un sistema completamente ensamblado. Estas pruebas en la empresa son las que están automatizadas para la aplicación como tal, y se serializan con el prefijo PS seguido de números, como por ejemplo PS003839.

- Pruebas de aceptación (PA): Tratan de verificar que el producto final bajo inspección cumple con las necesidades y expectativas del interesado o *stakeholder*. En el sistema de la empresa también están serializadas con el prefijo PA y números, como por ejemplo PA004539. Varias PS pueden ir asociadas a una misma PA (por ejemplo, para probar los casos no válidos).

No obstante, las pruebas sobre API carecen de códigos serializados y PA, aunque cuentan con baterías clasificadas por servicio.

Las pruebas sobre API se clasifican bajo pruebas de integración.

Como todo código, es conveniente que esté documentado para que sea mantenible, y a ser posible, que esté diseñado de manera que sea modificado fácilmente, o expandido, en un futuro. No obstante, a la hora de documentar una API, no es suficiente con eso, ya que idealmente deberíamos tener una documentación completa externa al código que podamos entregar a los integradores, no es el propio código, a ser posible con ejemplos de uso si se trata de peticiones particularmente complejas, y también detallando cualquier restricción, precondición o proceso especial necesario para poner en marcha el sistema o usarlo.

Esto es importante por ejemplo en aquellas peticiones que requieran de permisos concretos, será necesario indicar qué nivel y conjunto de permisos se requieren con el fin de ayudar y no confundir al usuario y facilitarle el uso de nuestra API. Además, también ayuda tener un estilo unificado y global en la extensión completa del documento: seguir una guía de estilo.

Las guías de estilo son documentos que expresan cómo deben escribirse o modelarse otros documentos. También se pueden aplicar a código fuente. En esta empresa, se ha especificado que cada petición debe tener:

- Una breve descripción de lo que hace
- Una tabla con los argumentos, su tipo y si son o no opcionales
- Una petición ejemplo
- La respuesta de dicha petición, tanto el HTTP como el propio mensaje
- Cualquier indicación apropiada (si necesita permisos, por ejemplo)

Y a ser posible, si existen varios resultados posibles para la petición, detallar dichos casos con la misma estructura mencionada. Esto se da, por ejemplo, en métodos de inicio de sesión, donde introducir las credenciales incorrectas puede dar una respuesta diferente a la de introducir un usuario inexistente.

Como nota menor, también se incluyen plantillas de Word para que todo tenga un formato concreto y unificado.

2.1 Demostración del ciclo manual de documentación

Para visualizar a más detalle el proceso de documentación manual, veamos una serie de figuras ilustrativas:

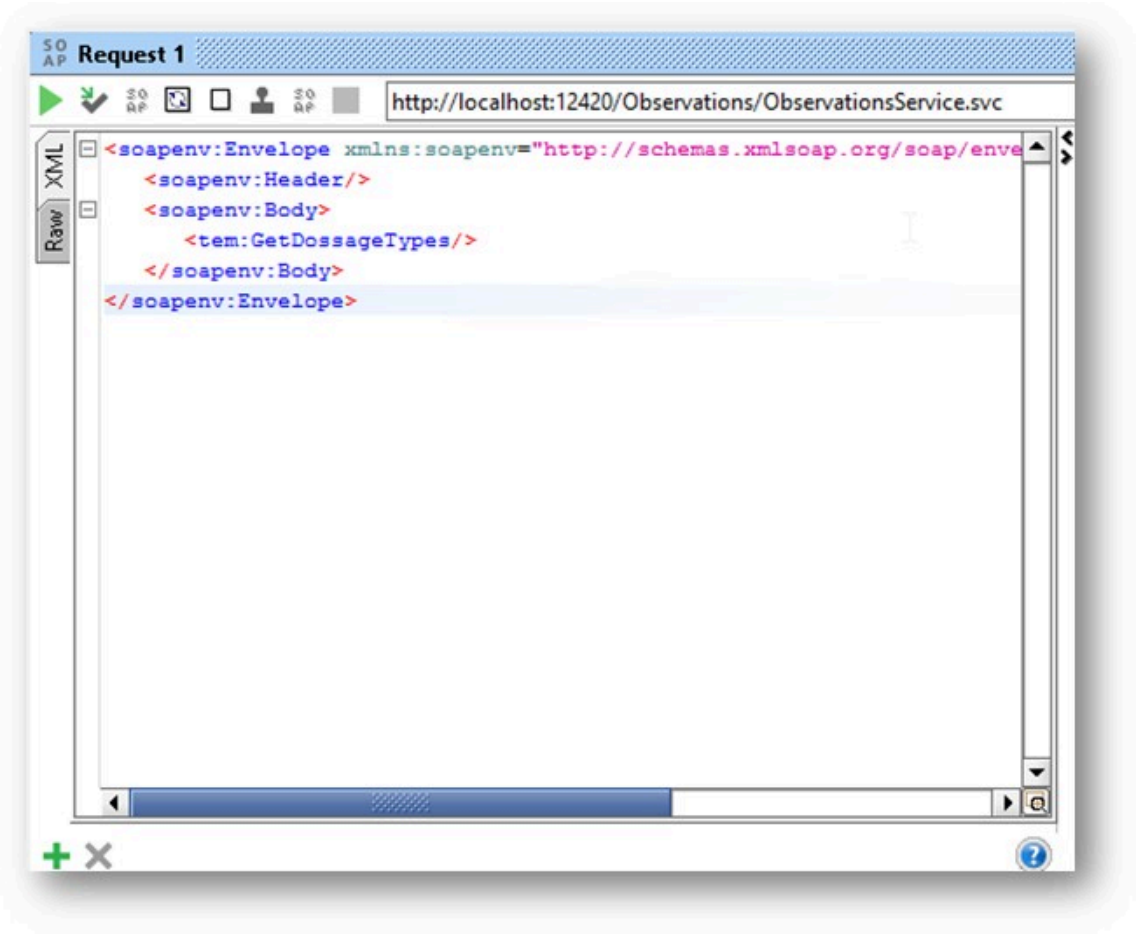


Figura 1. Petición estándar SOAP

Como podemos ver en la Figura 1, tenemos una petición que no admite ningún argumento, así que no hace falta rellenarla. Si tuviese argumentos, se vería como en la Figura 2:

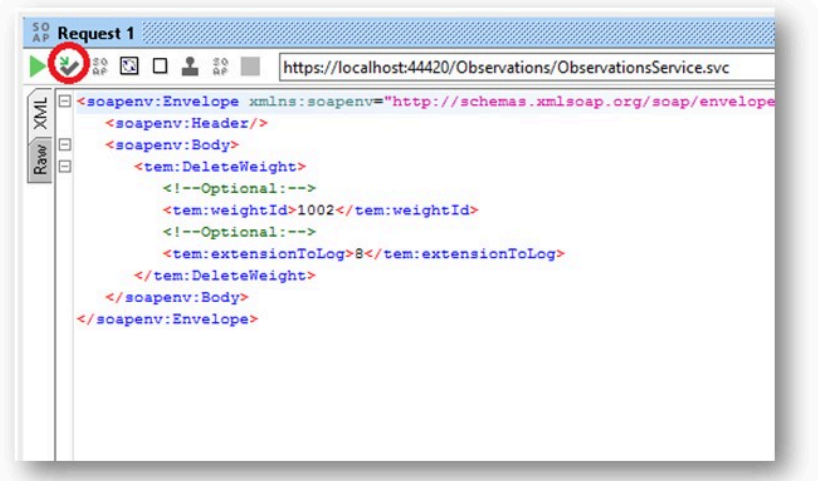


Figura 2. Petición SOAP con argumentos

Como podemos ver en la Figura 2, tenemos una petición que sí admite argumentos, aunque en este caso está ya rellena. Si no lo estuviera, los valores se mostrarían como interrogaciones.

Para saber qué valores hemos de rellenar, hemos de buscar en las bases de datos o conocer los valores por experiencia.

Una vez lanzamos la petición y se comprueba que devuelve una respuesta satisfactoria, podemos hacer la prueba automatizada.

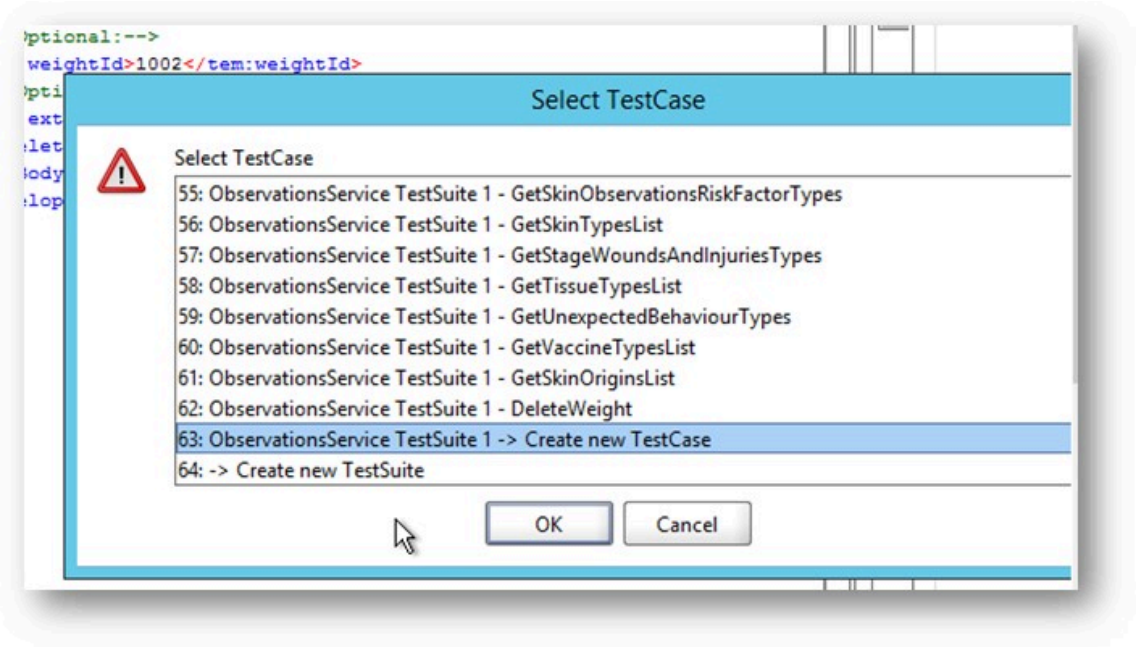


Figura 3. Creación de un caso de prueba

Como podemos ver en la Figura 3, SOAPUI nos pide seleccionar el caso de prueba que va a estar asociado a esta petición. Una vez creado esto, podemos crear la expresión regular que verifica el contenido de la respuesta que nos ha llegado.

Finalmente, deberemos redactar la documentación de la prueba en la plantilla Word proporcionada por la empresa.

3. Estado del Arte

El mercado de pruebas automatizadas es muy grande, y se han popularizado diversos entornos para los que se han desarrollado programas asistentes, ya sean peticiones SOAP, o generadores de datos para *mocking*, o simulación, y vamos a estudiar algunos en esta sección.

Cualquier empresa que tenga una API es vital que tenga una documentación adecuada, actualizada y completa. Es por ello que la empresa está poniendo un alto énfasis en documentar sus API, y tiene un proceso definido que vamos a discutir en esta sección.

Para empezar, comentaremos la tecnología empleada en este proceso:

- Lenguaje de programación C#[3]: Al estar en un entorno moderno centrado en Microsoft y sus tecnologías, es la opción natural en este caso, y es la elección de esta empresa.

- Entorno .NET Framework[4], IIS[5]: Son complementos para C# que permiten interactuar con los diferentes programas que ofrece Microsoft, como servidores de datos, servicios Web, y rutinas simplificadas para acceder a dichos servicios.

- XML[6]: eXtensible Markup Language, el formato de documentos que se usa para la transferencia de datos, notablemente influenciado por el uso SOAP. Existen librerías para deserializar datos en estos formatos, proporcionadas por Microsoft

- SOAP[7]: Simple Object Access Protocol, un protocolo que se utiliza para el traspaso de objetos entre programas diferentes, utilizando REST y cuerpos XML. Se utiliza primariamente en APIs.

- REST[8]: Representational State Transfer, un estilo de arquitectura software para paso de información y llamadas de API transfiriendo su estado usando HTTP y servicios Web.

- WSDL[9]: Web Services Definition Language, es un lenguaje que se utiliza para definir los cuerpos de peticiones de un servicio Web. Se pueden generar automáticamente, y se procesan también por ciertos programas. Indican cosas

[3] https://dotnet.microsoft.com/es-es/languages/csharp
[4] https://dotnet.microsoft.com/es-es/download/dotnet-framework
[5] https://www.iis.net/overview
[6] https://www.w3.org/XML/
[7] https://www.w3.org/TR/soap12-part1/Overview.html
[8] https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
[9] https://www.w3.org/TR/wsdl20/

como los tipos de los argumentos, si son opcionales o no, el orden y los nombres, espacios de nombres, y definiciones externas y estilos para XML.

En cuestión de programas, utilizamos:

- Visual Studio 2015[10]: el entorno de desarrollo para C# por parte de Microsoft. Lo utilizamos para poner en funcionamiento el servidor de API, y ver su código para determinar las causas de errores al lanzar peticiones, así como para ver el significado de los códigos de error.

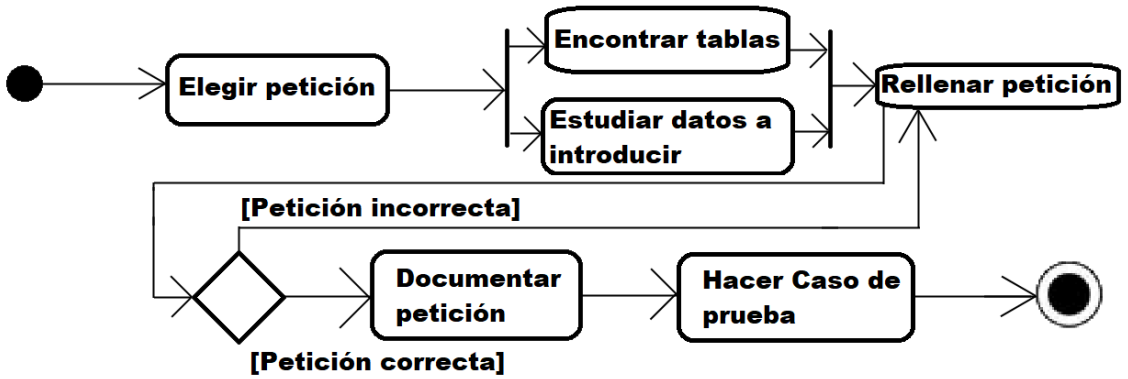
- SOAPUI[11]: un programa creado por SmartBear, escrito en Java y de código abierto, que permite manejar y generar peticiones SOAP ejemplo en base al WSDL del servidor.

El proceso de documentación que seguimos ya ha sido mencionado anteriormente, pero ahora vamos a verlo con más detalle.

El primer paso es asegurarse de que la máquina virtual que ofrece los datos está conectada, y entonces podemos abrir Visual Studio y arrancar el proyecto. En la parte de API tenemos dos proyectos diferentes: uno es el que da el servicio como tal y el otro es para apoyo de documentación, se llama Models, y ofrece información sobre la estructura, los datos, y los tipos de los elementos que se almacenan en la base de datos y que se usan en las peticiones.

Una vez hecho esto, debemos abrir el proyecto de SOAPUI asociado con el servicio que estamos documentando. Estos proyectos están subidos en el sistema de control de versiones de la empresa (TFS[12] o Team Foundation Server, es una tecnología de Microsoft, similar a Git[13] pero sin ramas). Con esto completado, ya estamos listos para empezar a documentar.

Veamos una ilustración global usando un diagrama de secuencia, como el mostrado en la Figura 1.



[10] https://visualstudio.microsoft.com/es/
[11] https://www.soapui.org
[12] https://learn.microsoft.com/en-us/previous-versions/azure/devops/
[13] https://git-scm.com

Figura 4. Diagrama de secuencia documentación manual

Como podemos ver en la Figura 4, el proceso es mayoritariamente lineal, con pocas ramificaciones. Únicamente tenemos un caso en el que según el resultado de la petición, debemos volver atrás.

Escogiendo una petición cualquiera, encontraremos que sigue la estructura de un archivo XML con un par de diferencias:

- No tiene una definición de versión XML
- Tiene un envoltorio SOAPenv (SOAP envelope)[2]
- Tiene numerosos espacios de nombres (namespaces) definidos

SOAPUI nos permite hacer diversas cosas con esta petición. Podemos añadir cabeceras HTTP, lo cual es necesario para prácticamente todas las peticiones; podemos rellenar la petición o modificarla a nuestro antojo, pudiendo comprimir y esconder los grupos de XML que no nos interesen; o podemos enviar la petición directamente.

Si decidimos enviar la petición, pueden pasar dos cosas:

- Que la petición no se responda con un código 200 OK HTTP, en cuyo caso nos dará un error 500 Internal Server Error. Esto es porque la petición que hemos enviado es defectuosa en algún campo o cabecera. Es la principal causa de ralentización a la hora de documentar la API, dado que no ofrece ninguna información particularmente útil para corregir el problema que lo causa. En algunos casos es posible discernir que es por alguna cabecera que falta, pero no dice qué cabecera es. En el caso de argumentos de la petición como tal, es muy difícil saber cuál es el dato que hace fallar la petición.

- Que la petición se responda con un 200 OK HTTP, que significa que el formato es correcto, pero tendremos que ver el contenido de la respuesta para ver qué es lo que realmente ha pasado. Tenemos dos posibilidades:
    - Que la petición se nos conteste con un cuerpo que contenga los datos que esperamos, ya sea un OK a la hora de crear o borrar algo, o bien el elemento que estábamos buscando si es una consulta. Si estamos en este caso, la petición ha ido bien y estamos listos para escribir su documentación.

    - Que la petición se nos conteste con un cuerpo vacío, o un código de error del ERP, lo que significa que lo que esperábamos que sucediese no ha sucedido por algún motivo. Esto puede ser causado, si estamos borrando algo, porque lo que hemos indicado a borrar no existe, o si estamos tratando de obtener algo que se nos devuelva un cuerpo vacío significa que la base de datos no contiene algo con ese filtro o

identificador. Esta es la segunda traba que más cuesta de resolver a la hora de documentar.

Una vez tengamos una petición que se envía y responde correctamente, podemos proceder a documentarla. Esto implica copiar la petición y las respuestas HTTP y SOAP a otra máquina, que es donde tenemos acceso al documento para escribirlo. Buscamos la petición escogida y describimos brevemente lo que hace, sus argumentos, si son obligatorios o no, su tipo, y después de eso pegamos la petición, y sus respuestas con el formato concreto que se utiliza en la empresa.

3.1 Dificultades y trabas durante el proceso de documentación

Durante la labor de documentación de APIs surgen diversas problemáticas, como las que hemos mencionado de pasada anteriormente, y que ahora vamos a estudiar en detalle para ver cómo se podrían solucionar más tarde.

- Lo primero que nos puede pasar, antes de que siquiera seamos capaces de enviar la petición, es que directamente ni nos genere una petición ejemplo. Esto pasa porque el archivo WSDL del servidor se ha movido o actualizado y hay que volver a configurarlo en SOAPUI. Esto se soluciona de manera manual y rara vez pasa de todas formas.

- Una vez enviamos la petición es posible que nos diga que es incorrecta, como ya hemos mencionado anteriormente. Las causas de esto pueden ser desde que falta una cabecera a que hay algún dato que no hemos rellenado o que no tiene el formato correcto. Para las cabeceras normalmente son siempre dos y están documentadas en la guía del documentador, así que se soluciona de forma manual. Para los datos o formatos incorrectos no existe una solución como tal, es cuestión de conocer la infraestructura interna de la aplicación completa del ERP, cosa que los usuarios no conocen.

- También es posible que nos de un código de error del ERP, distinto del de HTTP que sería 200 OK, esto quiere decir que alguna de las comprobaciones del método que estamos llamando ha fallado. Normalmente esto es un número, como -2 o -6, que tienen un significado dentro del propio fichero de código fuente. Esto es difícil de diagnosticar, puesto que tenemos que ir haciendo *debugging* por las distintas comprobaciones del propio código para ver qué es lo que falla.

- Otro problema muy común es el de que simplemente los usuarios no conocen el entorno del ERP lo suficiente como para saber lo que hay que rellenar y donde, el tipo que hay que ponerle, las partes que son opcionales, conocer los diversos DTO y sus componentes internos, y no hay tiempo para formarlos en estos conocimientos. Simplemente es algo que se conoce con el tiempo y que, hasta que se adquiere destreza, ralentiza y dificulta la documentación.

- Algo que contribuye también a dificultar la documentación es que no hay un sitio concreto donde se pueda saber qué tablas se están modificando o consultando, además de que hay más de 600 tablas donde buscar. Esto tiene el efecto secundario de que si queremos poblar la base de datos y no podemos rellenar una petición de añadir o crear datos, y queremos hacerlo manualmente, nos es muy difícil hacerlo.

- Además de todo esto, el código fuente de la API está traducido al inglés de forma incorrecta, y no se corresponde en absoluto muchas veces con los nombres en castellano que se encuentran en la base de datos y en el resto de la aplicación, lo que dificulta aún más encontrar en la base de datos lo que se supone que buscamos.

- Algunas peticiones son demasiado largas, unas 400.000 líneas, que esencialmente son imposibles de rellenar. Se requiere un alto nivel de conocimiento sobre los datos que son opcionales para poder "podar" las partes de la petición que no son necesarias. Además, si se intenta guardar el proyecto de SOAPUI con una de estas peticiones directamente se congela la aplicación y se produce un fallo de falta de memoria en el heap de la máquina virtual de Java. Si intentamos abrirlo con un editor de texto, como el Bloc de notas de Windows, éste deja de responder. La única manera de guardarlo es con el programa Notepad++ en un archivo externo a SOAPUI, que acaba siendo de unos 30 MB, y de todas formas esto nos sirve de bien poco, porque seguimos teniendo una petición con medio millón de líneas que no podemos rellenar de forma manual en un plazo de tiempo sensato.

- Y finalmente un último problema, aunque menor, es que para poner la petición en la documentación debemos eliminar los comentarios, de acuerdo al estilo de la empresa. Esto puede tardar de meros segundos a un par de minutos dependiendo de la petición, y es algo que se podría automatizar fácilmente.

En general, todo esto contribuye a que el proceso de documentar la API sea uno arduo, monótono y con muchísimas trabas que ralentizan el proceso, haciendo que un servicio tome, de media, de 150 a 200 horas por persona.

3.2 Herramientas presentes en el mercado

SOAPUI

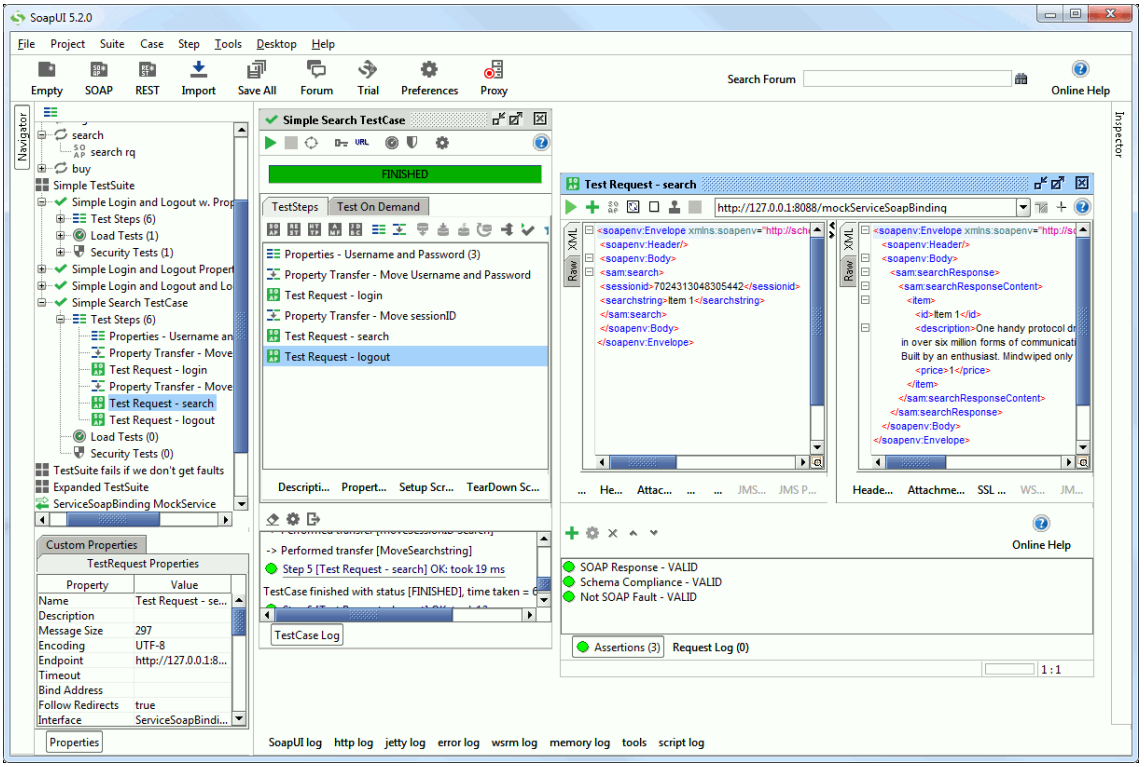


Figura 5. Interfaz de SOAPUI

La Figura 5 muestra una sesión de SOAPUI[14], que es una herramienta desarrollada por SmartBear para el manejo, organización, generación, verificación, rellenado y enviado de peticiones SOAP de forma automática para servicios Web locales o remotos basados en la estructura software SOAP. Es la herramienta que usamos actualmente.

Con esta aplicación se pueden crear proyectos para cada servicio Web que se quiera probar. De esta manera se pueden tener organizados según convenga, y lanzar *Test Suites* según servicios o cualquier otra clasificación.

Esta aplicación permite realizar acciones como las siguientes:

- Creación de diferentes proyectos: esto nos permite crear, para cada servicio un proyecto, o quizás queremos tener varios proyectos para un mismo servicio, por ejemplo para las peticiones de casos válidos y no válidos, o diferentes organizaciones que podamos tomar.

[14] https://www.soapui.org

- Generación de peticiones SOAP en base al WSDL del servicio Web: una de las herramientas más útiles de este programa, pues no hay una manera integrada y sencilla de hacer esto en C# o en otro lenguaje. Nos permite crear peticiones completas, aunque sin datos, de forma automática, con comentarios para indicar los tipos si así lo deseamos.

- Edición básica de archivos XML: Con utilidades para visualizar partes del archivo, comprimir o seleccionar de golpe secciones del XML, nos permite rellenar las peticiones de forma algo más sencilla.

- Lanzamiento de peticiones individuales y en batería: Permite probar el servicio Web lanzando peticiones de forma manual o automática, con informe de resultados fácilmente visible.

- Autogenerar datos: se basa en los tipos definidos en el WSDL para generar los datos. No obstante, no sabe distinguir ni generar datos útiles para ciertos campos. Por ejemplo, el campo “comentarios” y el campo “DNI” tienen ambos tipo String, pero no podrían ser más diferentes. SOAPUI no es capaz de ver esta diferencia, y tampoco hay manera de programarlo de forma externa para apuntarlo en la dirección correcta.

Como ya hemos comentado, esta aplicación tiene sus puntos débiles también, y nos impide usarla directamente para nuestros propósitos, aunque nos basaremos mucho en su rol actual en la empresa, puesto que no la estamos sustituyendo, sino complementando. Los principales problemas son:

- Problemas de memoria y al guardar cuando manejamos peticiones grandes: las máquinas donde operamos con este programa no tienen suficiente memoria RAM para albergar a todos los usuarios y también manejar y almacenar las peticiones grandes que a veces genera el programa. Como ya hemos comentado anteriormente, hay veces que no se puede guardar en disco la petición porque el programa se queda sin memoria, haciendo que la máquina virtual de Java deje de responder.

- El rellenado automático de peticiones es impreciso y poco útil: es una herramienta con mucho potencial y muy útil si tuviera en cuenta el entorno del ERP. No obstante, los datos a los que tiene acceso únicamente provienen del WSDL y sólo se indican tipos primitivos, con lo que cualquier cosa que no sea un entero, decimal, fecha o similar se considera simplemente como un String, ya sea realmente un comentario, un DNI, el nombre de una persona, o el código unificado de un medicamento.

Swagger

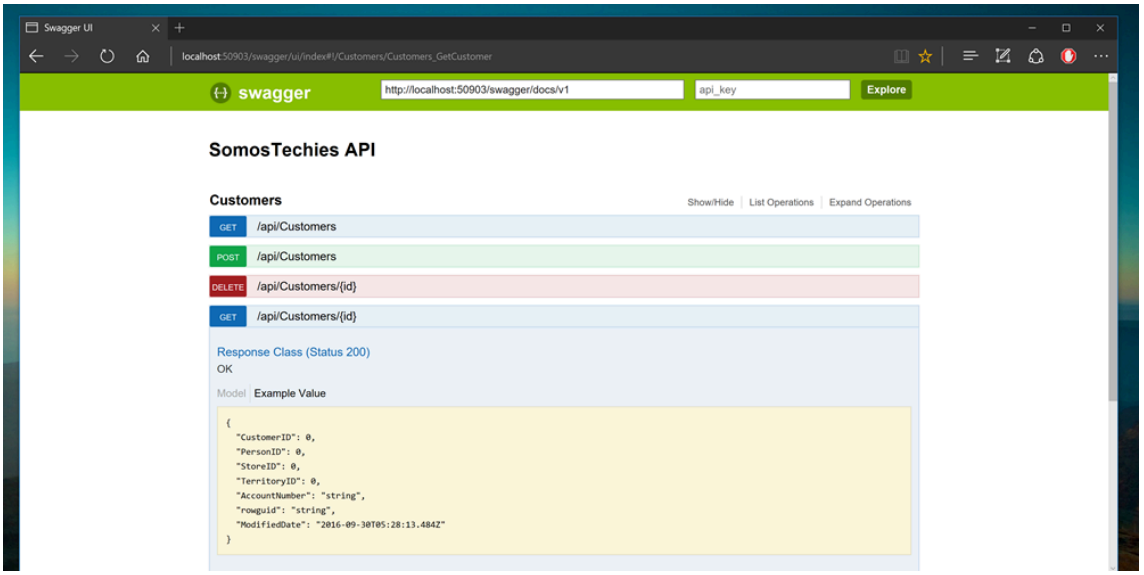


Figura 6. Interfaz de Swagger

La Figura 6 muestra Swagger[15], que es otra herramienta desarrollada por SmartBear, es una herramienta específicamente para diseñar y documentar APIs de servicios Web que usen REST.

Esta herramienta está escrita en Java y corre en cualquier entorno de trabajo, incluido en la propia Web. Es también una herramienta de código abierto, como SOAPUI, e implementa la especificación OpenAPI, que se usa por la fundación Linux y apoyada por las grandes empresas tecnológicas como Google. Se puede usar en entornos de equipos de trabajo.

Algunas de las ventajas que nos ofrece son:

- Una interfaz estandarizada a nivel de mercado para ofrecer nuestra documentación: Swagger está muy bien posicionado en el mercado, siendo el producto líder del sector, proporcionando mucha de la documentación pública para APIs del mercado actual. Además, es capaz de generarse de forma asistida, y todas las documentaciones que se han generado con Swagger tienen una vista común, con lo que cualquier especialista en la tecnología de Swagger podría entender con un golpe de vista nuestra API.

- Si tuviéramos que crear la API desde cero, nos vendrían muy bien las herramientas que ofrece Swagger para diseñar, ya que tiene muchas integraciones con diversos sistemas, y nos permite especificar nuestra API en un lenguaje estilo *low code* que se puede traducir a cualquier otro lenguaje, ya sea C# + .NET o Java, por ejemplo. Las ventajas de esto residen en que la portabilidad de nuestra API sería posiblemente la mejor, pero no es algo que

[15] https://swagger.io

sea relevante en nuestro entorno de trabajo, donde el lenguaje predominante es C# + .NET, el sistema operativo es Windows, y tenemos ya una API creada.

Mockaroo

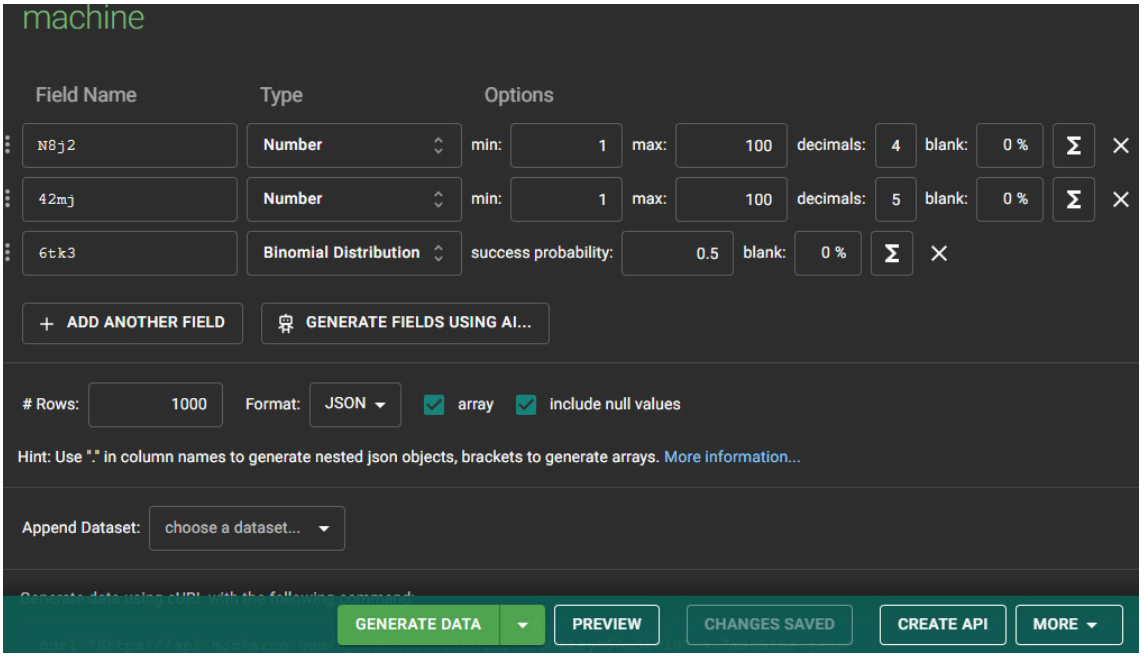


Figura 7. Interfaz de Mockaroo

La Figura 7 muestra Mockaroo[16], que es una herramienta Web potenciada por inteligencia artificial que nos permite generar datos según ciertos tipos que ya tiene registrados en su base de datos, y nos permite exportarlos a diferentes formatos, incluyendo CSV, SQL, JSON, Excel y XML.

Tiene una versión gratuita que nos permite generar hasta 1000 datos, y una versión de pago con una versión para Docker[17] por 60$ al año.

Este programa ofrece algunas capacidades interesantes a considerar, como las siguientes:

- Nos permite generar datos en masa: útil si queremos muchos datos en archivos para usar con diferentes aplicaciones, o almacenar para entrenar otro generador, por ejemplo.

- Tiene tipos de datos muy específicos: formatos para números de la seguridad social, nombres de medicamentos, datos basados en expresiones regulares, o incluso strings para causar problemas a los intérpretes que podamos tener programados. Esto es útil para ciertas pruebas de ciberseguridad, o para generar datos concretos.

No obstante, tiene serios problemas que nos impiden usarlo para nuestro propósito:

[16] https://www.mockaroo.com
[17] https://www.docker.com

- Es una aplicación Web, se tendría que automatizar su uso con Selenium o bien usar la versión de pago de Docker, y comunicarse con esa instancia, además de que no se usa Docker en esta empresa, sino la tecnología de máquinas virtuales de Microsoft, y que se controla muchísimo el tráfico de internet por motivos de seguridad, usando una VPN para teletrabajar y no permitiendo el acceso a ninguna página web quitando aquellas acordadas con el departamento de ciberseguridad de la empresa.

- Los datos específicos que puede generar son útiles para el ámbito norteamericano: los números de la seguridad social que genera, y demás datos, son para esa región, no para los españoles. Se podría hacer una expresión regular para cada dato, pero es un sobreesfuerzo cuando realmente no nos interesa tanto tener muchos datos diferentes como tener la mayoría de datos cubiertos con al menos un ejemplo.

- Además, se tendría que integrar de alguna manera en otro programa que sea el que rellena las peticiones, o hacer el rellenado de manera manual. Puestos a hacer el rellenado de forma externa, usando esto sólo para generar los datos, podemos obtener los datos directamente de las pruebas de sistema que se realizan y poblar las bases de datos con eso.

3.3 Conclusiones

Entre las herramientas que hemos descrito, aunque muy populares, ninguna en concreto es capaz de realizar todo lo que queremos al nivel de rendimiento que deseamos. Todas ofrecen utilidades muy diversas y potentes por separado, y son definitivamente algo que las hace muy interesantes a la hora de proponer un ciclo de desarrollo y documentación de API. No obstante, como ya hemos comentado, nuestro caso de aplicación es bastante diferente al propuesto por estas herramientas, y dado que tenemos un entorno basado en Windows, con tecnología de Microsoft, y una API ya hecha, a falta de documentar, con un uso muy integrado ya en el ciclo de SOAPUI, no podemos elegir realísticamente ninguna otra opción.

Además, al querer implementar una funcionalidad avanzada que requiere de ciertos datos exclusivos a esta empresa, y diversas otras utilidades, es importante discutir dos puntos de vista para el desarrollo que deseamos hacer: implementar desde cero un sistema de documentación integrando la API de SOAPUI, o hacer una aplicación compañera a SOAPUI para uso del documentador.

Ambas propuestas son interesantes, y vamos a discutir brevemente las implicaciones de cada una:

En el caso de desarrollar una sistema entero desde cero para efectivamente sustituir a SOAPUI, necesitaríamos crear:

- Un intérprete en C# del .dll que proporciona la empresa

- Una aplicación en Java para usar la API de SOAPUI

- Recrear la mayoría de la funcionalidad de SOAPUI: los proyectos, el lector de WSDL, editor de XML, enviar peticiones y recibir respuestas, Test Suites, y más

- Alguna manera de comunicar estas dos aplicaciones diferentes, idealmente por sockets.

Esta forma de afrontar el problema plantea varios problemas. Por una parte, estaríamos planteando un proyecto muy grande, y escrito en una tecnología ajena a la empresa.

Además, implicaría volver a entrenar a la plantilla, y desarrollar un programa que funcione similar a SOAPUI, de una forma verificable. Obviamente, esto no es práctico ni factible con el tiempo y recursos que tenemos.

La otra opción es una mucho más sensata y menos ambiciosa, pero igualmente potente y cumple con el objetivo igual de bien: Desarrollar una aplicación compañera a SOAPUI que nos permita solucionar algunos de los problemas que se presentan a la hora de documentar. La estructura general de este proyecto es la siguiente:

- Un intérprete en C# del .dll que proporciona la empresa

- Integrar eso en una aplicación de escritorio en C# y Windows Forms[18] que tenga la capacidad de manejar XML, usando las librerías de Microsoft para ello

- Lo único que tiene que poder hacer la aplicación es rellenar una petición usando la información obtenida del .dll, y los datos que se introducen de forma manual o se generan automáticamente. Además de las herramientas mencionadas como acortar peticiones o quitar comentarios.

Esta otra forma de afrontar el problema se apoya en SOAPUI para realizar todo excepto el rellenado de peticiones. La generación por WSDL, lanzamiento, organización en proyectos, Test Suites, verificado, y demás se siguen haciendo por SOAPUI. Lo único que los documentadores tienen que entrenar ahora es el rellenado con el nuevo programa, que sería capaz de reconocer la petición y sus componentes, los tipos asociados, y permitirá editarla como se desee.

Se han considerado ambas propuestas y nos hemos decantado por la última, puesto que es la que menos acople tiene, es más sencilla y rápida de desarrollar, y requiere el menor tiempo de entrenamiento.

[18] https://learn.microsoft.com/es-es/dotnet/desktop/winforms/overview

En definitiva, el procedimiento a seguir es el siguiente: el desarrollo de una aplicación independiente, escrita en C# con Windows Forms, que integre el .dll de la empresa, y los formatos usados por SOAP.

4. Tecnologías Utilizadas

Para el desarrollo de la solución usaremos diversas herramientas populares del mercado, y en uso por la empresa. Debido al entorno donde se produce este proyecto, nos encontramos limitados en la variedad de lenguajes, entornos de desarrollo, y tecnologías que podemos utilizar.

En las siguientes secciones discutiremos los distintos aspectos tecnológicos a considerar en las diferentes partes del proyecto, desde lenguajes de programación a formatos de archivos, librerías, *frameworks* y más.

4.1 Lenguaje de programación y entorno de desarrollo

La empresa tiene un uso muy alto de C# y Visual Basic .NET en sus códigos fuente. No obstante, Visual Basic .NET está desfasado y no se recomienda su uso, quedando relegado a papeles de parches sobre programas antiguos que aún lo usen, que es el caso de los programas de esta empresa.

Además, no tenemos conocimientos de este lenguaje de programación más allá de los conceptos básicos de semántica y sintaxis, aunque contamos con un manual que se puede consultar. Si nos remitimos al requisito de entorno de desarrollo por un momento, así como lo ha especificado la empresa, es recomendable que usemos Visual Studio 2019, que es la versión más moderna que se puede manejar en la empresa.

Visual Studio 2019 y versiones posteriores ya no incluyen los paquetes necesarios para desarrollar en Visual Basic .NET por defecto, se tienen que conseguir de forma separada. Microsoft ha publicado un informe de soporte para Visual Basic .NET donde explican que darán soporte de mantenimiento, pero no lo desarrollarán más, como sí es el caso con C#.

Es natural, pues, elegir C# como lenguaje preferido para este proyecto: es un lenguaje moderno, actualizado, con soporte por parte de Microsoft, con acceso a todas las tecnologías de .NET, y además es el que más experiencia tenemos, siendo parte del temario oficial de la carrera universitaria.

Además, C# es un lenguaje con numerosas librerías que vamos a utilizar más adelante para manejo sencillo de XML, interacción con la interfaz de usuario, manejo de archivos, y al ser orientado a objetos, nos permite tener una visión más genérica de lo que compone una petición.

Es por todo esto que escogeremos C# como nuestro lenguaje de programación. Para poder desarrollar en él, necesitaremos un entorno de desarrollo. Microsoft ofrece Visual Studio, y en esta empresa se utilizan numerosas versiones del mismo, las de 2013, 2015 y a la que tenemos acceso para el desarrollo de la solución, la de 2019.

Para la parte gráfica de la aplicación usaremos la opción natural en nuestro entorno: Windows Forms. Es un *framework* sencillo de usar, con el que tenemos experiencia, y tiene soporte de Microsoft, integración en Visual Studio, e interacción con C#.

4.2 Formatos de peticiones - SOAP, XML y restricciones varias

La API que estamos documentando genera y procesa peticiones SOAP con cuerpos XML, así que no hay mucha opción es este aspecto: nuestra solución debe dar soporte al manejo de archivos o formatos XML y concretamente los mensajes SOAP, que son ligeramente diferentes, ya que no tienen, por ejemplo, cabecera de versión XML.

Microsoft proporciona una librería para tratar XML por defecto, incluida con .NET Framework. No tenemos tampoco mucha opción a la hora de usar librerías, con las restricciones de seguridad que tiene la empresa que hemos mencionado anteriormente, no se nos permite, a menos que lo solicitemos:

- Navegar a páginas Web fuera de la empresa
- Utilizar código en librerías que no sean las oficiales de Microsoft o de la empresa
- Usar una base de datos o crearla nosotros mismos, a menos que sea estrictamente necesario

Esto pone serias restricciones al tipo de herramienta y al planteamiento que podemos hacer. Por ejemplo, todo lo que debe hacer el herramienta ha de ser bajo demanda y procedural, no deberíamos almacenar nada, y si tenemos que almacenar algo entre ejecuciones, no debe ser en una base de datos, sino en ficheros.

Tampoco podemos almacenar libremente todo lo que queramos, ya que hay poco espacio disponible para todos los usuarios y se debe compartir.

Para generar datos de ejemplo para las peticiones, no podemos utilizar páginas Web y extraer los datos con Selenium, porque no tenemos acceso a dichas páginas. Tampoco podemos utilizar librerías externas que generen datos por nosotros.

En resumen: debemos programar toda la funcionalidad que necesitemos manualmente o usando librerías estándar, incluyendo la generación de datos, el tratamiento y edición de XML, y el guardado o editado de archivos.

5. Desarrollo de la solución

Hay diversas soluciones que podrían darse a cada problema por separado: scripts de automatización para limpiar peticiones, formaciones a los usuarios para conocer mejor el ERP, redactar una guía de datos y valores, hacer plantillas para DTO concretos, importar partes de las bases de datos de las baterías de pruebas automatizadas en vez de tener que poblar la base de datos manualmente, tener un mínimo de documentación sobre la equivalencia entre los nombres en inglés y en español, una guía sobre qué tablas (o a veces vistas SQL) accede cada servicio, un mínimo de información sobre lo que es obligatorio o no en cada DTO, y más que podríamos proponer.

No se tratan de soluciones definitivas ni viables en algunos casos. No es viable formar a un usuario que va a estar en la empresa un par de meses en total. El volumen de datos es tan grande que sería complejo hacer un conjunto de plantillas para cada DTO. Importar los datos de las pruebas es complejo también, porque son scripts SQL diseñados para cada prueba y se tendría que estudiar que no haya conflictos entre lo que hay en esos scripts, la base de datos ya existente, y los futuros scripts que añadamos.

Lo único a lo que sí que tenemos acceso es un archivo .dll que se usa en la API para dotarla de las estructuras necesarias para procesar y comunicarse con el ERP, lo que también incluye las estructuras de los DTO, sus argumentos, tipos, nombres, orden, y demás información útil. No obstante, esto no es una guía documentada para humanos, es un .dll que realmente solo es entendible y observable por un programa.

Una solución efectiva es hacer una herramienta que nos permita solucionar los diversos problemas que hemos discutido anteriormente, y eliminar o reducir el requisito de conocimiento sobre el ERP del documentador, poniendo esa carga sobre el .dll al que tenemos acceso, y haciendo que se puedan rellenar automáticamente las peticiones, con la información proporcionada por el .dll y el formato deseado.

Como hemos descrito, debemos desarrollar una herramienta algo compleja en un entorno que no nos da muchas opciones. Hemos descartado muchas de las opciones modernas que hemos estudiado en algunas asignaturas (como aquellas basadas en el uso de Selenium[19] para los datos) porque no se nos permite usar herramientas externas ni Internet, y podemos desarrollar soluciones manuales.

Durante el desarrollo de la solución, afrontaremos los problemas que se nos plantean usando una metodología ágil, con objetivos claramente definidos, discutiendo las diversas opciones y los entornos relevantes que usaremos.

[19] https://www.selenium.dev

5.1 Metodología

Empecemos por hacer un repaso del equipo involucrado en el desarrollo, y los diferentes roles que se llevan a cabo en un desarrollo ágil.

Llamarlo equipo es quizás muy generoso, teniendo en cuenta que este trabajo ha sido realizado por una persona, el autor del TFG. Debido a esto, la planificación y priorización ha sido realizada por el desarrollador.

El Product Owner el jefe de departamento de testing automatizado, que ha expresado varias propuestas respecto al proyecto. Estas propuestas incluyen el uso del DLL de la empresa para obtener los datos, o el rellenado parcial de ciertos cuerpos que únicamente requieren de un componente.

Como Stakeholders tenemos a diversas personas y departamentos de la empresa, ya que ha sido propuesto como una herramienta de utilidad para los departamentos de soporte, desarrollo e integración, y testing manual.

5.2 Requisitos

Sigamos por hacer un repaso del entorno, de los requisitos y deseos de la solución que queremos desarrollar.

El entorno en el que nos encontramos es el de documentación de una API, con peticiones SOAP y cuerpos XML. Es un trabajo pesado, no se ajusta a trabajadores nuevos, que desconocen los componentes internos del programa. Es completamente manual en la parte de rellenado de peticiones, que es sin duda la que más tiempo cuesta, y vamos con una media de unas 150-200 horas por persona por servicio de unas 40-50 peticiones cada uno. Es decir, una petición cuesta alrededor de 3-4 horas por persona.

Además de este problema de desconocimiento de datos y estructuras a la hora de rellenar peticiones, nos topamos con otro problema: el de limpiar o “podar” peticiones. Para la documentación nos interesa no tener comentarios, tener tabulado todo y eliminar ciertos tipos de datos que no son relevantes. Hacer eso manualmente cuesta mucho tiempo y es algo que se podría hacer de forma automática.

Y finalmente, y quizás lo más problemático, el hecho de que a veces tenemos peticiones excesivamente largas, que son imposibles de rellenar por una persona en un tiempo sensato, y que hacen que nuestros editores de texto dejen de responder. Son peticiones que también necesitamos rellenar, pero no hay una buena solución manual.

Antes de comenzar el trabajo se hicieron diversas entrevistas cerradas a los miembros del equipo de documentación de API, para determinar los problemas y entorno con el que trabajamos. En base a los datos obtenidos, se enumeraron diversos objetivos e

ideas de funcionalidad para la herramienta a desarrollar, usando técnicas de *brainstorming*, que luego pasan a formar parte del *backlog* del proyecto.

Una vez puestas todas las ideas en una lista, deberíamos darles prioridad. Hay que saber reconocer qué partes de la herramienta son esenciales, cuáles son deseables, y cuáles son prescindibles por el momento. Para esto podemos usar una técnica conocida como la priorización MoSCoW[20]. Consiste en encajar cada idea del *backlog* en un marco concreto de cuatro disponibles: "Must have" o esencial, "Should have" o recomendable, "Could have" o idealmente, y "Won't have" o desestimado.

MUST
- Rellenador de peticiones
- Acortador de peticiones
- Limpiador de peticiones
- Lectura de contenido desde código fuente o .dll

SHOULD
- Inferencia de contenido de la petición
- Regeneración de peticiones
- Modo sencillo de añadir datos a los generadores de datos
- Procesamiento de peticiones en Stream
- Importar y exportar peticiones a archivos de texto

COULD
- Integración WSDL
- Buscador de elementos en código fuente
- Estadísticas de la petición (nº de componentes, omitidos, etc)
- Generación de documentación de la petición en Word

WON'T
- Lanzador de peticiones
- Generador de Regex para las respuestas
- Base de datos
- Actualizador automático del programa

El proyecto tiene un tiempo límite, y aunque empezamos con mucho más tiempo del habitual, es importante no infravalorar el esfuerzo que va a llevar el desarrollo de esta solución.

XML como tal es un lenguaje que no es fácil de usar para interpretar o crear archivos, está pensado para ser leído e interpretado. La información que tenemos disponible tampoco está hecha para ser leída directamente, hay que tomar varios pasos previos en preparar esos datos para poder usarlos, ya que provienen de un .dll.

También hemos marcado como algo recomendable la posibilidad de añadir generadores de datos de forma sencilla. Esto es algo que trataremos más adelante,

[20] https://en.wikipedia.org/wiki/MoSCoW_method

pero idealmente lo querríamos por código, integrado con la propia solución, y no simplemente una base de datos.

El tiempo designado a este proyecto es del 1 de diciembre de 2023 al 14 de marzo de 2024.
Tenemos un plan de 3 *Sprints*, de longitud variable según los objetivos. Serán planificados con Trello[21].

Un Sprint 0, dedicado a la investigación de alternativas y formación de tecnologías, cuyos resultados se han comentado anteriormente. Su duración fue de una semana, antes de diciembre.

Un Sprint 1, dedicado a la creación de un editor y navegador de XML, con posibilidad de guardar, cargar y limpiar peticiones, diseñar el esquema básico de la ventana del programa, y la posibilidad de navegar por el archivo XML, con edición de campos rápida. Esto ayudaría al proceso de documentación de API en este estado. También se hizo la inferencia de la estructura de la petición automáticamente en base al texto. Se dedicaron dos semanas a esto.

Un último Sprint 2, dedicado enteramente al rellenador de peticiones, y a la inferencia de los datos automáticamente del .dll oficial, así como la regeneración de peticiones, y el generador de datos. Se ha dedicado el resto del tiempo a este Sprint.

Notablemente, hemos tenido que retirar del alcance del proyecto las secciones de Could y Won't.

Detallando los motivos para cada descarte:

- Integración WSDL: No hay una librería que podamos usar para esto, aunque sería muy útil para saber, por ejemplo, el tipo de los valores de la petición. Sin esto, ponemos esa carga sobre el usuario. No obstante, SOAPUI genera esta información para el usuario.

- Buscador de elementos en código fuente: Cada instalación de cada usuario es diferente, tendríamos que pedir al usuario que nos indique su localización. Lo que estaríamos facilitando al usuario es no tener que hacer una búsqueda en el proyecto de Visual Studio que siempre tienen abierto, algo que no toma tanto tiempo.

- Estadísticas de la petición (nº de componentes, omitidos, etc): Esto realmente no es una información necesaria para el usuario, normalmente el deseo va a ser rellenar la petición por completo.

- Generación de documentación de la petición en Word: Como ya hemos comentado anteriormente, esta parte es bastante corta, y tampoco podríamos hacer mucho, el fichero Word que estamos rellenando y la herramienta con las

[21] https://trello.com

peticiones están en máquinas virtuales diferentes, incomunicadas excepto por unas carpetas compartidas.

- Lanzador de peticiones: El objetivo de esta solución no es sustituir a SOAPUI, sino hacer una herramienta compañera. Obviamente, esto implica que no vamos a hacer un lanzador de peticiones.

- Generador de Expresiones Regulares para las respuestas: No tenemos manera de saber qué clase de respuesta nos van a devolver, y es algo que tampoco ahorra mucho tiempo al usuario.

- Base de datos común: No tenemos necesidad de esto, y además sabemos que no hay mucho espacio para compartir con el resto del departamento.

- Actualizador automático del programa: Esto es algo que estaría bien, también es código que está hecho en otra herramienta a la que tenemos acceso en la empresa: el lanzador de pruebas automáticas ATUN. El código de actualizaciones automáticas en cuestión lo hemos arreglado personalmente, con lo que lo conocemos bien. No obstante, en ATUN tiene sentido tener este código, porque es una herramienta distribuida en varias máquinas virtuales, con parte cliente y servidor, y tener la misma versión es importante. En nuestro caso, es una aplicación sin ninguna conexión externa, no es tan importante. Podemos hacer que el usuario compile la nueva versión manualmente, o distribuirla nosotros con las carpetas compartidas.

Veamos un *Mockup* para hacernos una idea de la clase de interfaz que ofreceremos al usuario:

Tener claro un boceto rápido de cómo va a ser visualmente nuestra aplicación nos da una idea de qué controles vamos a necesitar, cómo los vamos a distribuir, y cuál va a ser el flujo de trabajo de nuestra aplicación.

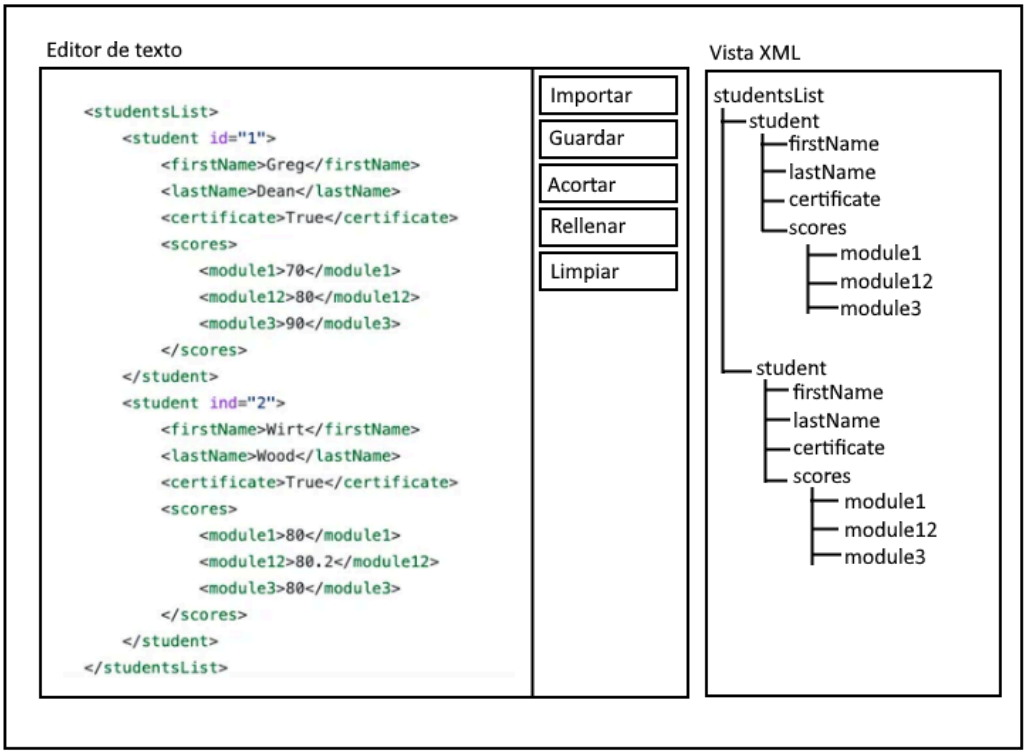


Figura 8. Mockup de interfaz

Como podemos ver en la Figura 8, nuestra interfaz es bastante similar a un editor de texto convencional, como puede ser Word. Tiene una vista XML para poder navegar rápidamente por el documento, y los botones con los casos de uso que hemos mencionado anteriormente.

El flujo de trabajo también es similar: Importamos manualmente o con el botón designado la petición, y entonces tenemos a nuestra disposición un esquema del documento, y diversas acciones que podemos realizar sobre el mismo.

Sigamos con un diagrama de casos de uso, para ilustrar el flujo de trabajo con la herramienta:

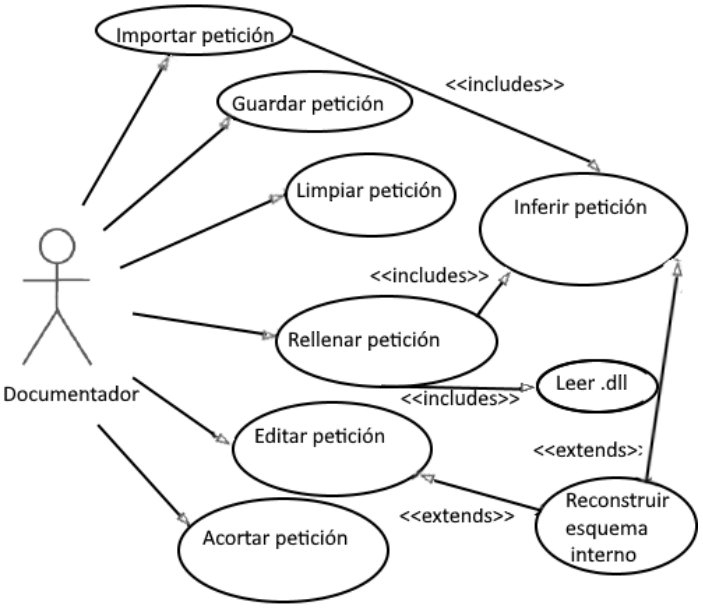


Figura 9. Caso de uso de la herramienta

Podemos ver en la Figura 9 que tenemos los casos de uso esperados: podemos importar y guardar peticiones, editarlas manualmente, acortarlas, y finalmente, rellenarlas. Este último caso tiene un par de pasos internos de los que se ocupa la herramienta.

Al rellenar la petición, la herramienta debe inferir los contenidos de la propia petición que tenemos, consultar el .dll, y posiblemente reconstruir el esquema interno que tiene de la petición mientras hace esto.

Podemos describir cada Caso de Uso (CU) como sigue:

- Importar petición: La acción de tomar una petición de un archivo o del portapapeles y ponerlo dentro del programa.

- Guardar petición: La acción de tomar la petición dentro de la herramienta y guardarla en un archivo.

- Rellenar petición: El acto de editar de forma automática la petición, usando la información de la herramienta para ello. El usuario sólo debería confirmar si desea el dato propuesto u otro.

- Acortar petición: El proceso por el cual tomamos una petición larga y la acortamos para poder procesarla con el programa.

- Limpiar petición: El proceso por el cual tomamos una petición y la reescribimos con los tabuladores nivelados, sin comentarios, y con la estructura correcta para un archivo XML.

- Editar petición: Alterar manualmente el contenido de una petición, ya sea cambiando valores o borrando elementos.

- Leer .dll: La herramienta obtiene la información del .dll proporcionado por la empresa que corresponde a las estructuras y contenidos, nombres y tipos de los datos de las peticiones.

- Inferir petición: La herramienta obtiene de la petición proporcionada la estructura y contenidos disponibles. Esto puede ser el contenido en su totalidad, parcial, o ninguno, ya que depende del estado en el que el usuario nos haya dado la petición.

- Reconstruir esquema interno: La herramienta realiza una exploración de la petición que se le proporciona. Toma notas de qué orden y qué contenidos tiene cada elemento, si son DTO o solo atributos, sus tipos, y más. Esta información se usa para reconstruir la petición desde cero.

Y para finalizar esta sección, repasemos los requisitos no funcionales recolectados desde el comienzo de la elicitación:

La herramienta debe ser accesible y fácil de usar por un usuario sin experiencia. Preferiblemente, debería ocultar todas las partes complejas del proceso que hemos comentado (conocimiento de los datos, estructuras del archivo XML, espacios de nombres, componentes y número de los mismos dentro de DTO). El objetivo final ideal es dar la petición a la herramienta y que obtengamos una petición rellena y válida con el mínimo esfuerzo. Esto será validado usando una métrica de tiempo de completado de la tarea: Sabemos que una petición estándar toma de media unas 2 horas para documentarse. Usando la herramienta debería tener un tiempo menor, alrededor de 1 hora y cuarto.

Por supuesto, la herramienta debería ser mantenible, estar documentada, ser escalable y robusta. Diversos de los programas con los que contamos en la empresa ya han tenido problemas por cambios en la estructura de las máquinas donde son ejecutados, y se han arreglado modificando su código, pero no ha sido una tarea fácil. Muchos de estos programas no tienen apenas comentarios ni guías. Hemos establecido que un nivel aceptable para este requisito es que la herramienta cuente con un código fuente comentado en detalle, y una guía de usuario.

En cuestión de rendimiento también tenemos ciertas restricciones: la máquina virtual que va a ejecutar esta herramienta tiene varios usuarios al mismo tiempo, con lo que nuestra solución debería ser ligera y usar el mínimo de memoria RAM y CPU posible. Hemos establecido que un uso de memoria de unos 150 MB o menos es aceptable.

Además, tenemos preocupaciones por el espacio del disco duro, con lo que debería ser completamente un proceso en tiempo de ejecución, con el mínimo componente en disco posible. Un espacio de disco de menos de 50 MB se consideraría aceptable.

Tenemos también que recordar que la herramienta no debe usar una base de datos, toda la información necesaria debe ser generada en tiempo de ejecución.

Y por último, como potencialmente tratamos archivos muy grandes, deberíamos afrontar el problema usando Streams, en vez de cargar todo en memoria, ya que si no caeremos en el mismo problema que tienen el resto de herramientas que ya usamos. Esto se puede validar también con el requisito de memoria: no debería subir excesivamente el uso de memoria mientras tratamos archivos grandes.

5.3 Arquitectura global del sistema

Durante esta sección veremos en más detalle la arquitectura de componentes propuesta para darnos una idea más clara de cómo van a interactuar las diferentes piezas del proyecto.

Para esto usaremos el diagrama de componentes mostrado en la Figura 7.

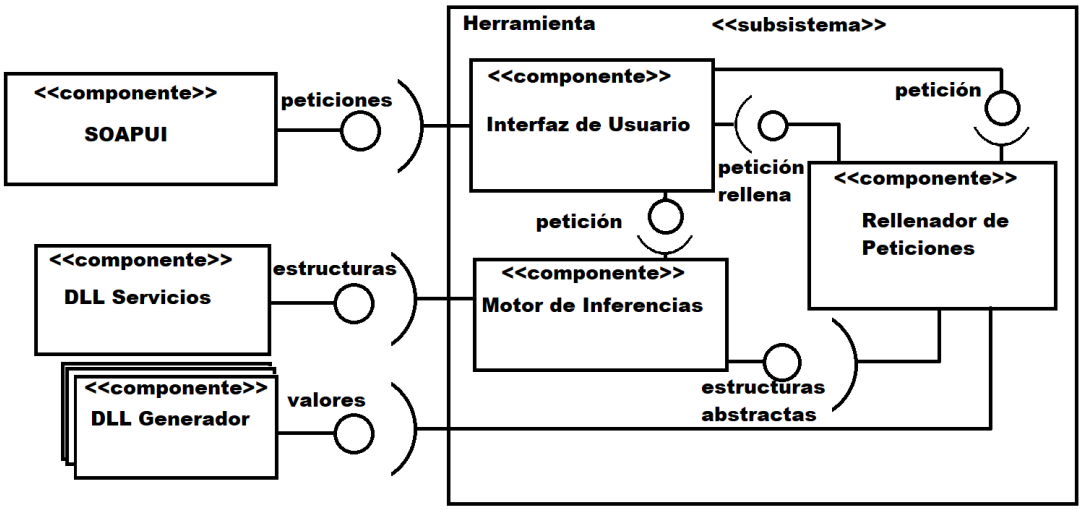


Figura 10. Diagrama de componentes

Como podemos ver en la Figura 10, tenemos a SOAPUI como componente, que ofrece peticiones. Esas peticiones son requeridas por la herramienta, ya que es lo que se desea rellenar. También hemos mencionado anteriormente que no tenemos el WSDL para definirlas, con lo que necesitamos que las genere SOAPUI.

Una vez recibida en la Interfaz de Usuario, es entregada al Motor de Inferencias y al Rellenador de Peticiones, que interactúan entre sí para proporcionar los datos necesarios para rellenarla.

Concretamente, el objetivo del Motor de Inferencias es obtener una abstracción de la estructura de la petición que se ha recibido. Una vez hecho esto, se obtiene la

información disponible sobre las estructuras inferidas de un componente externo, el DLL de Servicios de la empresa.

Una vez generadas todas las estructuras necesarias, se envían al Rellenador de Peticiones, que se encarga de iterar por cada componente de las estructuras generadas. Dependiendo de qué parámetros tengan esos componentes, se llamará a un generador u otro de valores. Una vez tengamos el valor obtenido de los generadores, se introduce en la petición. Una vez se ha insertado todo, se devuelve la petición rellena a la Interfaz de Usuario.

5.4 Diseño

Durante esta sección tomaremos los requisitos descritos anteriormente junto con las restricciones que hemos de tener en cuenta, e iremos describiendo los cambios o rediseños que se deban hacer.

Para empezar, veamos cuáles son los puntos que vamos a tratar:

- Por una parte, tenemos el problema de hacer los generadores de valores para que sean sencillos de crear y usar por un usuario normal, sin necesidad de conocer el código.

- Por otra parte, tenemos varios tipos de componentes en una petición: Valores, DTO, Atributos y Enumerados (o Enums). Cada uno va a necesitar su generador distinto, y dependiendo del caso va a acceder a diversos subsistemas que tendremos que programar.

- También está la cuestión de la relación entre las numerosas clases que tenemos. Hay que establecer el vínculo que existe entre Peticiones, Valores, DTO y Atributos, y lo ilustraremos más tarde con un diagrama de clases.

- Y por último pero no por ello menos importante: un *mockup* o prototipo visual de la interfaz del programa.

Empezando por una de las cuestiones más importantes del programa: la manera de crear valores personalizados para cada posible atributo o valor de petición de una manera sencilla, potente, y con el menor acople posible. Por suerte, tenemos un patrón que nos permite hacer esto mismo: El patrón estrategia. Además de esto, nos hemos impuesto un requisito más restrictivo aún: queremos que estos generadores sean *plug-and-play*, es decir, que no se requiera recompilar la herramienta principal para usarlos, únicamente el componente en sí.

Esto tiene numerosas ventajas. En nuestra empresa no es la primera vez que se pierde el código fuente de un programa. Es posible que en un futuro exista un generador particularmente útil del que no se tiene el código, y entonces se descubra un fallo en uno de los valores que genera. No podemos editar el código fuente porque

no lo tenemos, y no podemos saber cómo genera esos valores por el mismo motivo. No obstante, con este sistema, podemos hacer un “parche”, es decir, un nuevo generador que tome más prioridad que el anterior y que sobreescriba los valores concretos que no deseamos del anterior.

De esta manera, podemos crear un generador parche y distribuirlo para su uso por el resto de usuarios sin que estos tengan que recompilar nada.

Esto también implica que tenemos un sistema de orden de carga, por el cual tomamos un orden concreto para cargar los componentes del patrón estrategia. En nuestro caso, lo hemos decidido hacer por orden alfabético.

Esto nos permite definir un orden invariable entre ejecuciones. Si es necesario sobreescribir cierto valor concreto, se puede añadir un generador extra a la lista, y tomará efecto según el orden y los requisitos.

Veamos más en detalle cómo, a nivel de código, vamos a dar este control al usuario final de la aplicación. No olvidemos que, aunque esta solución está siendo creada para ayudar a documentar, no tenemos forma de generar los datos de todos y cada uno de los valores posibles de forma automática con la información que tenemos: necesitamos intervención de un humano que especifique qué poner en cada hueco.

Para ello, hemos decidido usar un sistema de *plugins* implementando el patrón Estrategia. Esto nos permite cumplir los objetivos que hemos propuesto anteriormente, concretamente el de evitar recompilar la herramienta por completo, y el de poder hacer parches y generadores de forma sencilla para el usuario, y distribuirlos. También los hace bastante potentes a la hora de ofrecer información para crear las reglas de generación de información, ya que estas clases tienen acceso a toda la información que tiene acceso el programa: nombre, tipo, de quién es hijo (si es un atributo), su espacio de nombres, etc.

Recordemos también que esta información es muy relevante, ya que no nos basta con el tipo para poder dar un valor correcto a un atributo o valor de petición. Un Apellido y un DNI ambos se clasifican internamente como una cadena de texto, pero no podrían ser más diferentes.

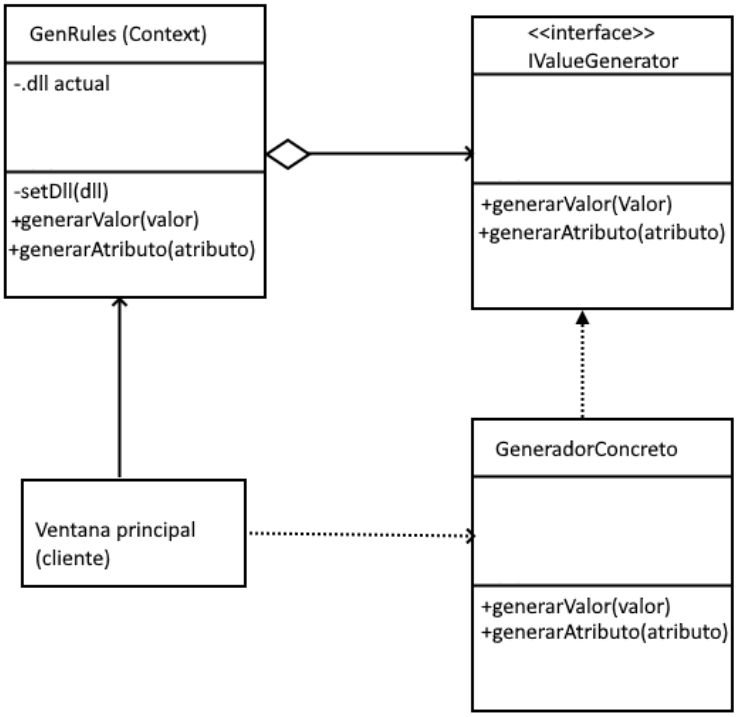


Figura 11. Diagrama de clases de patrón Estrategia

En la Figura 11 podemos apreciar que la clase GenRules es el contexto del patrón Estrategia, que tiene una referencia a un .dll que contiene una o más clases que implementan la interfaz IValueGenerator. Dichas clases son los generadores concretos. Desde la ventana principal solicitamos generar un valor. Dependiendo de si es un valor o un atributo se llama a un método generarValor() o generarAtributo().

Para elegir la estrategia, o el generador que se va a usar, se escoge por orden alfabético, como hemos detallado antes con el orden de carga.

Sigamos con la generación de estructuras o valores para los diferentes componentes de las peticiones.

Una petición contiene Valores, que están definidos en el WSDL, y que tienen un tipo concreto (Entero, Cadena de texto, etc). No obstante, también puede contener DTO.

Estos DTO pueden, por una parte, ser Enumerados, cuyos valores los obtenemos del .dll. Por otra parte, pueden contener Atributos, de un número variable de 1 a muchos.

Cada uno de estos Atributos tiene un tipo asignado (Entero, Cadena de texto, etc), y puede convertirse potencialmente en otro DTO, con su lista de Atributos, que recursivamente cada uno de ellos podría ser otro DTO.

Esto es un problema que trataremos en el siguiente segmento, pero por ahora, tenemos el problema de generar, con sus procesos y subprocesos, datos para cada uno de estos tipos.

La manera en la que lo hemos afrontado es usando un patrón Fachada, que oculta todos los subsistemas encargados de la generación de valores y estructuras.

Generar la estructura de un DTO o dar valores a los Enumerados, Atributos o Valores de la petición es algo que la herramienta necesita hacer muy a menudo. También es un proceso complejo que interactúa con diversos componentes internos y externos al programa. Es por ello que hemos decidido colocar el código encargado de los generadores detrás de un patrón Fachada, una clase sencilla al exterior que encapsula la complejidad y nos da una manera de utilizar ese código complejo.

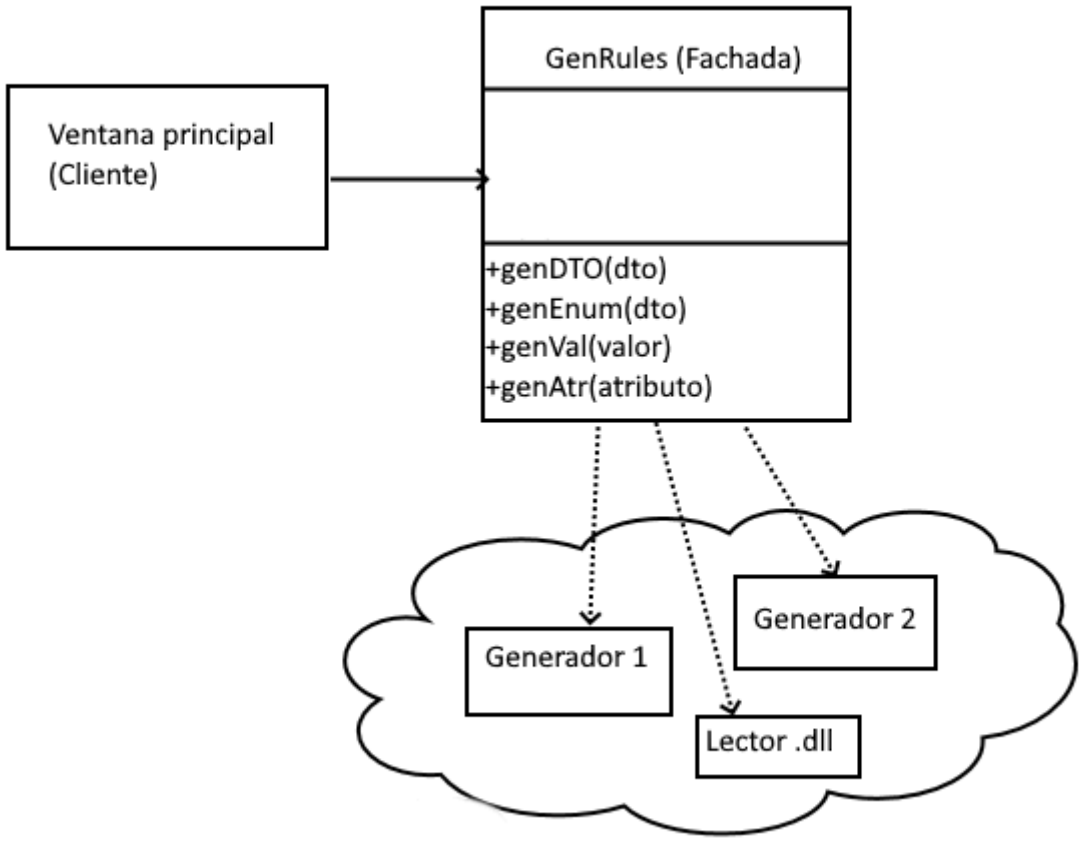


Figura 12. Diagrama de clases para patrón Fachada

En la Figura 12 podemos ver que la clase GenRules hace de fachada para los numerosos generadores que podamos tener, y la funcionalidad compleja de leer del .dll las propiedades de cada DTO.

Proporciona una manera opaca y sencilla de generar las estructuras y valores que son necesarios para el rellenado de la petición, ocultando la complejidad de la ventana principal. También oculta las relaciones complejas que tiene con el resto de clases.

Nuestro proyecto tiene varias clases utilitarias, y diversas clases entidad. Son en estas últimas en las que nos vamos a centrar en esta sección.

Nuestra aproximación ha sido la de abstraer los conceptos de los componentes de una petición. De esta manera, nos podemos centrar en las relaciones que tienen cada par de elementos.

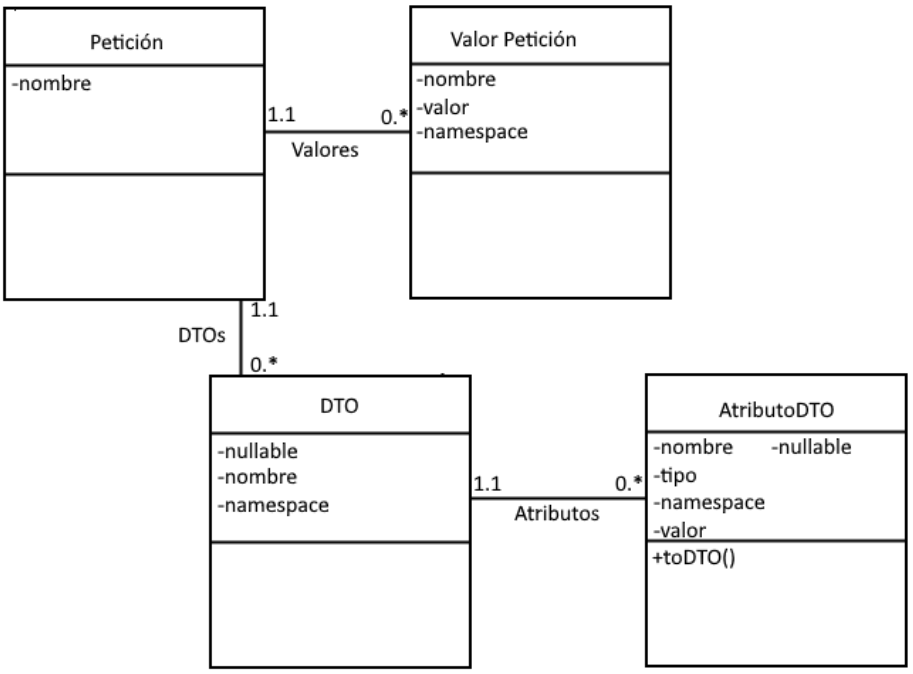


Figura 13. Diagrama de clases entidad

En la Figura 13 podemos apreciar que tenemos una clase denominada Petición, que representa la información relevante de la petición que estamos editando. Tiene una colección Valores, que son representados usando la clase Valor Petición. Previsiblemente, queremos almacenar su nombre, namespace o espacio de nombres XML, y su valor.

De Petición también tenemos una colección de DTO. Tiene nombre y espacio de nombres XML. A esto le añadimos una capacidad de ser nulo, y una lista de atributos, que es una colección de clases de AtributoDTO.

Cada AtributoDTO tiene un nombre, un tipo, un espacio de nombres XML, un valor, y si puede ser nulo o no. Además, tienen la capacidad de convertirse en DTO.

Esto coincide con la presentación que hemos hecho anteriormente sobre la estructura de peticiones que tratamos: Una petición puede tener tanto valores como DTO, y los DTO tienen atributos que a su vez pueden ser DTO de forma recursiva.

El resto de atributos que hemos mencionado, como el tipo o el espacio de nombres, lo usaremos para rellenar la petición. El espacio de nombres lo utilizamos para saber dónde colocar el componente dentro de la petición. El tipo obviamente lo necesitamos para dárselo a los generadores y que se ocupen de generar un valor apropiado teniéndolo en cuenta, junto con el nombre y demás información que almacenamos.

5.5 Programación

La doctrina común de programación de software suele ser la de no “perder el tiempo” diseñando y crear un código funcional lo más rápido posible, y dejar las refactorizaciones para más tarde según sea necesario. Esto implica varias cosas [3]:

Por una parte, obtenemos rápidamente una versión usable del programa. También somos capaces de descubrir si una idea es viable o no más rápido. No obstante, los problemas aparecen conforme avanzamos en el tiempo y se produce algún fallo o se requiere alguna modificación. Normalmente el código hecho sin diseño no suele ser mantenible, y requiere de grandes rediseños y posteriores pruebas, lo cual implica un esfuerzo mucho mayor al del resultante de partir de un buen diseño inicial.

Un factor contribuyente a la mantenibilidad es el diseño, como hemos comentado, pero también lo es la documentación, especialmente en una herramienta compleja como la que estamos creando. La política de documentación de la empresa suele ser de dejar anotaciones en cabeceras de métodos con su uso, ramificaciones y segmentos de código particularmente complejos o con variables que sean potencialmente importantes para depuración. Los segmentos de código deprecados se suelen eliminar del fichero por completo, usando el control de versiones para su visualización. Hemos seguido estas recomendaciones en el transcurso del proyecto.

En esta sección recorreremos la organización de la solución, el diseño de la arquitectura, programación de diversos algoritmos, y el detalle de los problemas y soluciones que hemos encontrado, teniendo en cuenta nuestros objetivos de usabilidad y mantenibilidad.

5.5.1 Estructura del proyecto

Empecemos con una visión general de la estructura del proyecto. Tenemos dos soluciones: Una es la propia herramienta, el rellenador; la otra es un generador básico. Podríamos tener varios generadores.

Dentro de la solución principal tenemos tres secciones y un archivo que es la vista principal, de acuerdo con el proyecto base que genera Windows Forms.

Las tres secciones que tenemos son:

- Clases Entidad
- Clases y código de utilidad
- Rellenado e importación de generadores externos

Bajo Clases Entidad tenemos las clases representadas en el diagrama de clases, con algún atributo extra que se usa en algunos algoritmos en algunos casos.

Bajo Utilidad encontramos código para manejo de XML, con jerarquía, tabulaciones, interpretación de elementos y más. También tenemos una clase para leer Streams de texto de forma inversa, que será importante más adelante.

Bajo Rellenado está uno de los pilares del proyecto: La clase GenRules y la Interfaz encargada de proporcionar el acople necesario a los generadores externos, IValueGenerator.

El tratamiento de texto y código para interactuar con el usuario está ubicado en el archivo de código de la vista principal de Windows Forms.

Y por último, nos queda la otra solución que hemos mencionado: el generador básico. Contiene un conjunto de reglas para fechas, nombres y booleanos. Se podría extender o duplicar y hacer nuevos generadores, y está pensado como plantilla.

Esta separación de soluciones nos permite tener una arquitectura global con *plugins*, como ya hemos mencionado en la sección de diseño. La solución principal contiene el código necesario para la interfaz y el manejo de XML y peticiones. Los *plugins* contienen el código ocupado de dar valores y son externos a la herramienta principal.

5.5.2 Lectura de peticiones

Ahora vamos a explorar en detalle algunos de los algoritmos que hemos diseñado y que usamos para las diversas acciones del programa.

Empecemos dando una visión global del procedimiento, y echaremos un vistazo a los trozos de código interesantes que tengamos por el camino.

Lo primero que hace la herramienta una vez insertamos (o modificamos) la petición es tratar de reconstruir las estructuras internas de XML. Son los espacios de nombres y la

vista XML con la que navegamos por el documento, además de los contenidos a alto nivel de la petición, información propia de la petición (como su nombre y componentes).

El código encargado de esto es la función UpdateTreeView(), y el detalle que vamos a explorar es el de la funcionalidad de obtención de los espacios de nombres.

```
//Lectura de namespaces
TextBoxXML.Text =  TextBoxXML.Text.Trim();
namespacesLine = TextBoxXML.Lines.FirstOrDefault();
if (namespacesLine == null || namespacesLine.Length == 0) return;
var working = TextBoxXML.Lines.FirstOrDefault().Split(' ');
Namespaces.Clear();
foreach (var e in working)
{
    if (e.Contains("xmlns:") && !e.Contains("envelope") && !e.Contains("Envelope") && !e.Contains("tempuri"))
    {
        String ns = e.Split(':')[1].Split('=')[0];
        String name = e.Split('.').Last();
        if (name.Last() == '>')
            name = name.Remove(name.Length - 1, 1);
        name = name.Remove(name.Length - 1, 1);
        Namespaces.Add(name, ns);
    }
}
```

Figura 14. Obtención de espacios de nombres XML

En la Figura 14 se describe el algoritmo encargado de inferir los espacios de nombres para el XML. Estos espacios de nombres están declarados en la primera línea del documento, y los necesitaremos para contrastarlos con los obtenidos del .dll.

Previsiblemente, TextBoxXML es la caja de texto de la petición, y namespacesLine es la primera línea de dicha caja de texto, que contiene los espacios de nombre.

Realizamos una comprobación de seguridad por la que si namespacesLine es vacía, el texto es inválido y no debemos continuar. Nunca tendremos una petición sin espacios de nombre.

Namespaces es un Dictionary<String, String>, donde enlazamos los nombres de dominio (obtenidos del .dll) a los espacios de nombre XML. Nos aseguramos de borrarlo para asegurar un estado consistente.

Iterando por cada línea, que hemos separado por espacios vacíos en una línea anterior, descartamos todo lo que tenga “envelope”, “Envelope” o “tempuri”. Estos son los espacios de nombres XML que están presentes en todas las peticiones SOAP, y no es relevante para las peticiones porque siempre las vamos a tener ahí.

La estructura de las líneas que estamos manipulando es la siguiente:

xmlns:add=”http://schemas.datacontract.org/2004/07/ADDInformatica.Resiplus.Services.ServicesDTO.Residents”

Queremos las partes de “add” y la de “Residents”. Para ello utilizamos los Splits que se pueden ver en el código. Por último enlazamos “Residents” con “add”.

La función inversa a esta recupera, dado el nombre de dominio, obtenemos el espacio de nombre.

```
public static string getNamespace(string key)
{
    if (key == null || key == "")
        return "KEY_NO_ASIGNADA";
    Namespaces.TryGetValue(key, out string wat);
    return wat;
}
```

Figura 15. Función para obtener espacios de nombres

Como podemos ver en la Figura 15, es una función que envuelve la función de Dictionary de TryGetValue() con un par de comprobaciones de seguridad para mayor comodidad.

Una vez hecho el enlace entre los espacios de nombres y los nombres de dominio, añadimos los nodos inferidos de la petición XML (usando un lector XML del paquete de Utilidades) a la estructura interna del programa. Cabe destacar que esta estructura interna en este momento no contiene más que la información que podemos leer directamente del texto de la petición, es decir, no hemos buscado nada en el .dll, no sabemos qué componentes tiene un DTO, por ejemplo, a menos que estén en la petición. Desconocemos los tipos, si son colecciones, *nullables*, y más. No obstante, esto lo solucionaremos de manera recursiva en el siguiente paso.

Hemos de tener en cuenta también que los objetos en las estructuras internas son simplificaciones genéricas de los objetos reales proporcionados por la empresa, también se pueden llamar *wrappers* o envoltorios, y que hemos de darles la información necesaria basándonos en lo que nos proporciona la empresa.

Esto lo hacemos de manera recursiva por la naturaleza de que los Atributos de un DTO podrían ser también DTO. El siguiente código que veremos se encarga de identificar el tipo de elemento que estamos viendo y llamar al generador (o estrategia) concreta.

```
//Método para generar los valores de la petición de forma recursiva, para Valores y DTOs
3 references
private void GenRulesRecursivo(Entities.ValorPeticion b)
{
    if (b is Entities.DTO)
    {
        Rellenado.GenRules.GenDTO((Entities.DTO)b);
        foreach (Entities.AtributoDTO c in ((Entities.DTO)b).Atributos)
        {
            Entities.DTO d = c.ToDTO();
            if (d != null) { GenRulesRecursivo(d); }//DTO
            else c.Valor = Rellenado.GenRules.Gen(c);//Atributo
        }
    }
    else b.Valor = Rellenado.GenRules.Gen(b);//Valor
}
```

Figura 16. Selección de estrategia en base al tipo de elemento a generar

Como podemos ver en la Figura 16, esta función toma un valorPetición. En caso de que sea, en efecto, un valor, lo daremos directamente al generador que nos dará un valor apropiado.

Si es un DTO, generamos las estructuras internas de ese DTO, lo que nos dará un listado de Atributos, tipos, nombres, y más, obtenidos del .dll. Una vez hecho esto, iteramos por cada atributo del DTO, y tratamos de convertir cada uno a DTO.

Como hemos mencionado, un Atributo de un DTO puede convertirse a DTO, en cuyo caso no tendrá valor asignado, y lo pasaremos de forma recursiva. En caso contrario, generamos el valor de forma similar a un Valor de Petición.

5.5.3 Generación de valores

Veamos cómo se genera un valor para atributos y valores de petición, como se detalla en la siguiente figura:

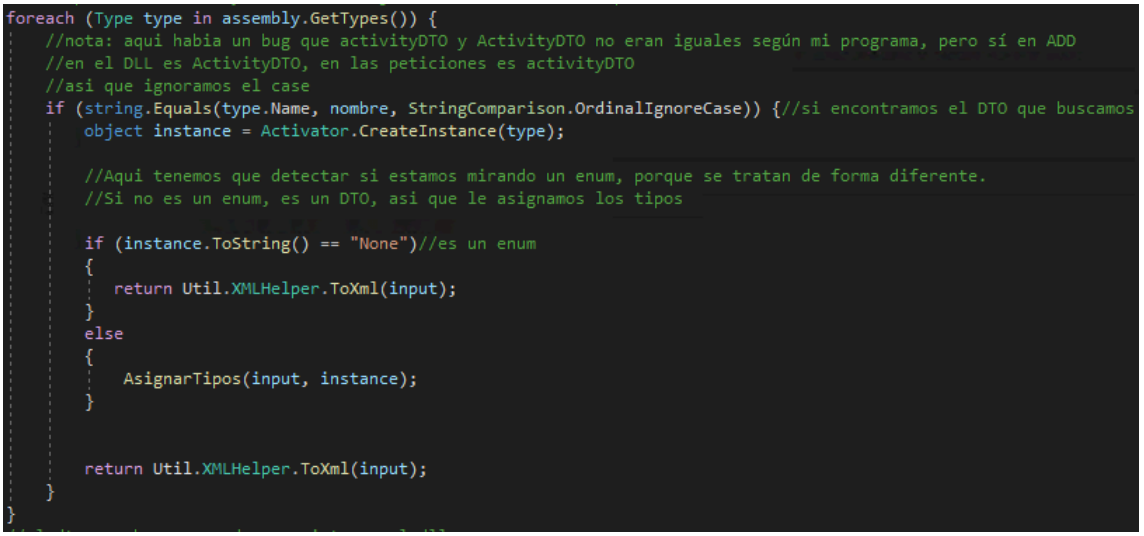
```
2 references
public static string Gen(Entities.AtributoDTO a)
{
    List<String> genVals = new List<String>();
    string pluginFolderPath = Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "plugins");
    if (Directory.Exists(pluginFolderPath))
    {
        String[] dllFiles = Directory.GetFiles(pluginFolderPath, "*.dll");
        Array.Sort(dllFiles);
        //usamos todos los dll en la carpeta plugins para generar valores, pero tenemos en cuenta la prioridad
        //zzHola > Hola
        //si has puesto mods a un juego de Bethesda es algo similar
        foreach (string dllFile in dllFiles)
        {
            try
            {
                var assembly = Assembly.LoadFrom(dllFile);
                var types = assembly.GetTypes();

                foreach (var type in types)
                {
                    if (typeof(IValueGenerator).IsAssignableFrom(type))
                    {
                        var generator = (IValueGenerator)Activator.CreateInstance(type);
                        string generatedValue = generator.GenCustomAtributo(a);
                        if(generatedValue != "Tipo no definido!!")
                            genVals.Add(generatedValue);
                    }
                }
            }
            catch (Exception e)
            {
                return "Error con el DLL";
            }
        }
    }
    else
    {
        return "No existe la carpeta plugins";
    }
    if(genVals.Count < 1)
        return "Tipo no definido!!";
    return genVals.Last();
}
```

Figura 17. Generación de valores para atributos

Como podemos ver en la Figura 17, el procedimiento para generar valores para los valores de petición es idéntico. Tenemos una carpeta que contiene los *plugins*, o generadores, y tomamos cada .dll que haya en esa carpeta en orden alfabético, y solicitamos un valor generado para cada uno de ellos.

Este es el patrón estrategia, dependiendo de qué estrategia o generador deseemos usar, si queremos generar valores para atributos, valores, enumerados o DTO.

Si queremos generar valores para DTO, es decir, su estructura, el procedimiento es el siguiente:

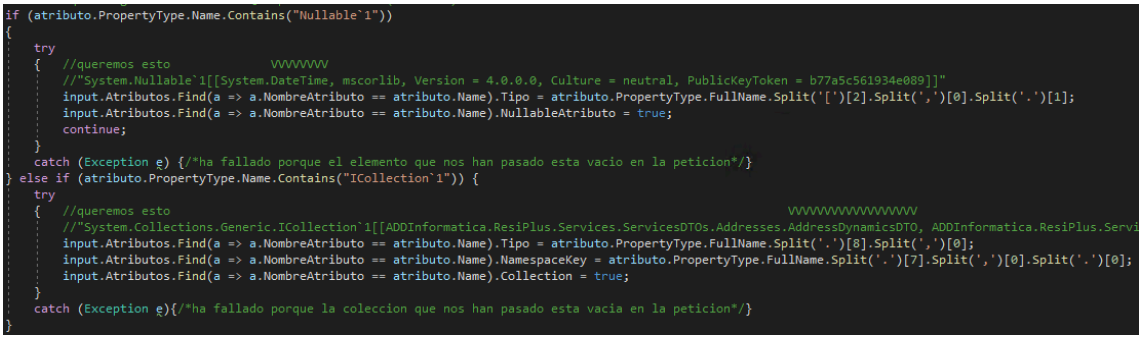


```
foreach (Type type in assembly.GetTypes()) {
    //nota: aqui habia un bug que activityDTO y ActivityDTO no eran iguales según mi programa, pero sí en ADD
    //en el DLL es ActivityDTO, en las peticiones es activityDTO
    //asi que ignoramos el case
    if (string.Equals(type.Name, nombre, StringComparison.OrdinalIgnoreCase)) {//si encontramos el DTO que buscamos
        object instance = Activator.CreateInstance(type);

        //Aqui tenemos que detectar si estamos mirando un enum, porque se tratan de forma diferente.
        //Si no es un enum, es un DTO, asi que le asignamos los tipos

        if (instance.ToString() == "None")//es un enum
        {
            return Util.XMLHelper.ToXml(input);
        }
        else
        {
            AsignarTipos(input, instance);
        }


        return Util.XMLHelper.ToXml(input);
    }
}
```

Figura 18. Generación de estructura para DTO

En la Figura 18, vemos cómo previamente a esto cargamos el .dll de la empresa, y ahora procedemos a buscar todas las estructuras de datos definidas en el mismo. Cuando encontramos el que estamos buscando, creamos una instancia, y detectamos si es un enumerado o no. Realmente, esto no es necesario, pero es una comprobación de seguridad, porque esta llamada siempre se va a hacer con DTO que no son enumerados. Si no es un enumerado, es decir, es un DTO normal, procedemos a asignar los tipos de sus atributos.

Iterando por cada atributo que tiene el DTO, exploramos el siguiente código:



```
if (atributo.PropertyType.Name.Contains("Nullable`1"))
{
    try
    {   //queremos esto             VVVVVVV
        //"System.Nullable`1[[System.DateTime, mscorlib, Version = 4.0.0.0, Culture = neutral, PublicKeyToken = b77a5c561934e089]]"
        input.Atributos.Find(a => a.NombreAtributo == atributo.Name).Tipo = atributo.PropertyType.FullName.Split('[')[2].Split(',')[0].Split('.')[1];
        input.Atributos.Find(a => a.NombreAtributo == atributo.Name).NullableAtributo = true;
        continue;
    }
    catch (Exception e) {/*ha fallado porque el elemento que nos han pasado esta vacio en la peticion*/}
} else if (atributo.PropertyType.Name.Contains("ICollection`1")) {
    try
    {   //queremos esto                                                                    VVVVVVVVVVVVVVV
        //"System.Collections.Generic.ICollection`1[[ADDInformatica.ResiPlus.Services.ServicesDTOs.Addresses.AddressDynamicsDTO, ADDInformatica.ResiPlus.Servi
        input.Atributos.Find(a => a.NombreAtributo == atributo.Name).Tipo = atributo.PropertyType.FullName.Split('.')[8].Split(',')[0];
        input.Atributos.Find(a => a.NombreAtributo == atributo.Name).NamespaceKey = atributo.PropertyType.FullName.Split('.')[7].Split(',')[0].Split('.')[0];
        input.Atributos.Find(a => a.NombreAtributo == atributo.Name).Collection = true;
    }
    catch (Exception e){/*ha fallado porque la coleccion que nos han pasado esta vacia en la peticion*/}
}
```

Figura 19. Asignación de tipos, parte 1

En la Figura 19, esencialmente estamos comprobando si cada atributo puede ser nulo, o si por el contrario es una colección, o es un tipo normal.

Hay que distinguir entre los tipos normales (int), los que pueden ser nulos (int?) y los que son colecciones (ICollection(int)), porque esto va a influir en si podemos o no omitirlos de la petición, o si podemos o no generar varios elementos en una misma colección.

Si no es una colección ni un posible nulo, entonces es un tipo normal, con lo que lo asignamos directamente. Si esto falla, es porque tenemos un atributo en el .dll que no hemos encontrado en la petición como texto, con lo que tenemos que generarlo. Es lo que se ilustra en la siguiente figura:

```
try{input.Atributos.Find(a => a.NombreAtributo == atributo.Name).Tipo = atributo.PropertyType.Name;}
catch (System.NullReferenceException f) {//hemos encontrado un atributo en un dto que la peticion dice que no tiene nada
    if (atributo.Name != "ValidationErrors")//no queremos ValidationErrors en las peticiones pq no sirve para nada
    {
        var g = new Entities.AtributoDTO();
        g.NamespaceKey = input.NamespaceKey;
        g.Namespace = VistaPrincipal.getNamespace(input.NamespaceKey);
        g.NombreAtributo = atributo.Name;
        g.Tipo = atributo.PropertyType.Name;
        g.TagPadre = nombreInstanciaDTO;
        g.Valor = Gen(g);
        input.Atributos.Add(g);
    }
}
```

Figura 20. Asignación de tipos, parte 2

Como vemos en la Figura 20, ignoramos los “ValidationErrors”, por motivos que explicaremos más adelante.

Por último, tenemos el caso de que sea un enumerado, y generamos sus valores obteniéndose del .dll como sigue:

```
if (instance.ToString() == "None")//es un enum
{
    Array enumVals = Enum.GetValues(type);
    foreach (var cosa in enumVals)
        sol.Add(cosa.ToString());
}
```

Figura 21. Generación de valores para Enumerados

Si vemos la Figura 21, es un código bastante sencillo, simplemente añadimos todos los posibles valores del enumerado a una lista que devolvemos.

Una vez hecho todo esto, tendremos en la estructura interna de la herramienta toda la información necesaria para poder rellenar la petición. No obstante, ahora tenemos que rellenar e insertar la petición en el texto.

5.5.4 Rellenado de peticiones

Esto es algo difícil, porque tenemos dos posibilidades a tener en cuenta: que el texto a rellenar esté en la petición, o que lo tengamos que generar nosotros mismos. Además de esto, es posible que sea una colección, o un DTO con nombre oculto, o que no admita colecciones, cosas que ya veremos más adelante.

Veamos la primera sección del código de insertar en el texto de la petición:

```
5 references
private void escribirValoresPeticion(String nodeText, String valor)
{
    String openingTag = $"<{nodeText}>";
    String closingTag = $"</{nodeText}>";
    int openingTagPosition = TextBoxXML.Text.IndexOf(openingTag);
    int closingTagPosition = TextBoxXML.Text.IndexOf(closingTag);
    if (openingTagPosition >= 0 && closingTagPosition >= 0)
    {
        TextBoxXML.Select(openingTagPosition + openingTag.Length, 0);
        TextBoxXML.SelectionLength = closingTagPosition - (openingTagPosition + openingTag.Length);
        TextBoxXML.ScrollToCaret();
        TextBoxXML.Text = TextBoxXML.Text.Remove(TextBoxXML.SelectionStart, TextBoxXML.SelectionLength);
        TextBoxXML.Select(openingTagPosition + openingTag.Length, 0);
        TextBoxXML.Text = TextBoxXML.Text.Insert(TextBoxXML.SelectionStart, valor);
    }
```

Figura 22. Método para escribir valores de la petición a texto, parte 1

En la Figura 22 tenemos una función toma por entradas el nombre de la etiqueta XML (con su espacio de nombres XML, detalle importante a destacar) y el valor que queremos asignar.

Esta combinación es única en todo el documento excepto en dos ocasiones:

- En los denominados “ValidationErrors”, que son secciones comunes en algunos DTO. Siempre llevan de espacio de nombre “add1” y se pueden eliminar. El limpiado de la petición que se hace de forma automática al importar la petición los elimina, así que la herramienta nunca los va a encontrar. También se ignoran si son componentes de un DTO que estemos generando.
- Si tenemos una colección de valores, obviamente cada elemento que se contenga tendrá una de estas combinaciones, provocando que ya no sea única. La solución que hemos dado a este caso es, una vez insertado un elemento en una colección, se elimina del texto y se guarda en memoria, pudiendo de esta manera repetir el proceso. Una vez generados todos los componentes que el usuario desee, se reinsertan en el texto todos los elementos de golpe.

El código expuesto en la Figura 19 no tiene mucha complicación: busca la combinación que conocemos como única en el documento, si la encontramos, borramos el valor que contenga e insertamos el que se nos ha pasado.

No obstante, este es un caso sencillo. En caso de que la petición que tenemos no contenga una estructura completa, tendremos que regenerar la petición. Veamos cómo lo solucionamos.

```
else
{//el campo generado no existe en la peticion que nos han pasado, asi que tenemos que generarlo...
    //esto busca el DTO que contiene el atributo que nos han pasado (solo puede ser un atributo)

    var atrib = FindAtributoPorNombre(Peticion, nodeText.Split(':')[1], nodeText.Split(':')[0]);

    var dtoPater = FindDTOPorAtributo(Peticion, atrib);

    int position;
```

Figura 23. Método para escribir valores de la petición a texto, parte 2

Dada la Figura 23, cabe destacar que sólo podemos llegar a esta situación si estamos rellenando un atributo. Recordemos que los valores de petición únicamente pueden estar en el nivel más alto, y por tanto, siempre los tendremos en el texto (asumiendo que no nos han pasado una petición defectuosa). También cabe recordar que un DTO no tiene valores, con lo que jamás invocamos esta función con un DTO.

Por tanto, buscamos en la colección abstracta de la petición el objeto atributoDTO que corresponde al texto que nos han pasado. Teniendo este objeto atributoDTO, buscamos el objeto DTO que contiene dicho atributoDTO.

Estos dos objetos son necesarios porque no tenemos conocimiento en este momento del espacio de nombres XML que se le tiene que asignar al atributo que estamos generando.

Como hemos generado antes toda la estructura, también contamos con los espacios de nombres para cada DTO.

```
if(Peticion.Contains(dtoPater))
    position = Util.XML.findAllPositionsOfPatterns(TextBoxXML.Text, new string[] {
    "</" + "tem" + ":" + atrib.TagPadre
}).ToList().FirstOrDefault();
else
position = Util.XML.findAllPositionsOfPatterns(TextBoxXML.Text, new string[] {
    "</" + dtoPater.Namespace + ":" + atrib.TagPadre
}).ToList().FirstOrDefault();
```

Figura 24. Método para escribir valores de la petición a texto, parte 3

Viendo la Figura 24, si la lista de componentes de la petición contiene el DTO que es padre del atributo que estamos intentando generar, es porque ese DTO es uno de primer nivel. Esto es relevante, porque en este caso el espacio de nombre será tem, no addX. Cabe destacar que esta colección únicamente contiene los DTO y valores de primer nivel. no los que puedan existir como atributos que se conviertan a DTO.

En caso de que el DTO padre no esté en esa colección como DTO, significa que proviene de un atributo que se ha convertido, entonces buscamos el padre de forma normal con sus espacios de nombres usuales.

En caso de que al buscar esta combinación sea el resultado la posición 0, es porque tenemos un DTO con nombre oculto. Lo hemos mencionado de pasada anteriormente,

y aquí es cuando es relevante. Hay ciertos DTO que internamente tienen un nombre, como MaintenancePlaceDTO, pero como atributo de otro DTO, es decir, antes de convertirse, tienen otro nombre (en este caso sería Place). En la petición aparece este segundo nombre, pero internamente en el .dll es el primero.

Si después de este último caso seguimos sin tener una posición correcta, entonces nos encontramos ante la última posibilidad: tenemos un DTO que tiene un nombre oculto y además, como es un atributo que se ha convertido a DTO, tiene dos espacios de nombre XML, y hemos escogido anteriormente la combinación incorrecta, así que volvemos a buscar con la combinación que nos queda. La figura 25 ilustra el código.

```
if(position == 0)
    position = Util.XML.findAllPositionsOfPatterns(TextBoxXML.Text, new string[] {
    "</" + dtoPater.TrueNamespace + ":" + dtoPater.TrueName
}).ToList().FirstOrDefault();
//si position sigue siendo 0 es porque tenemos que buscar esta otra combinacion
if (position == 0)
    position = Util.XML.findAllPositionsOfPatterns(TextBoxXML.Text, new string[] {
    "</" + getNamespace(dtoPater.NamespaceKey) + ":" + dtoPater.TrueName
}).ToList().FirstOrDefault();
```

Figura 25. Método para escribir valores de la petición a texto, parte 4

Una vez tenemos la posición correcta finalmente, insertamos el texto que nos han pasado al principio, el espacio de nombre XML y el nombre de la etiqueta, en la posición encontrada, y llamamos recursivamente al método, que será capaz de encontrar el texto y rellenarlo.

Ahora que ya sabemos el proceso por el que insertamos en el texto, veamos cómo hacemos las llamadas a este método, dando un paso atrás en nuestra visión global.

Estamos iterando por cada elemento de la colección que representa la petición, y estamos identificando los dos posibles tipos de elementos que contiene dicha colección: valores de petición y DTO. Si nos encontramos con un valor de petición, se inserta directamente, puesto que siempre va a estar en el texto. Si es un DTO, tenemos que hacer un proceso largo y complicado que vamos a detallar a continuación.

```
private void confirmarValoresDTO(Entities.DTO g)
{
    InsertarDatos ventana = new InsertarDatos();
    if (!Namespaces.ContainsValue(g.Namespace))
        g.Namespace = getNamespace(g.NamespaceKey);
    foreach (Entities.AtributoDTO k in g.Atributos)
    {
        if (k.ToDTO() == null)
        {
            //aqui k.namespace es null, y tiene que ser el que le toque por el padre
            if(!Namespaces.ContainsValue(k.Namespace))
                k.Namespace = getNamespace(g.NamespaceKey);
            ventana.rellenarForm(k.NombreAtributo, k.Tipo, "AtributoDTO de " + g.Nombre, k.Valor, k.Namespace);
            if (ventana.ShowDialog(this) == DialogResult.OK)
            {
                k.Valor = ventana.getValor();//atributo
                escribirValoresPeticion(k.Namespace + ":" + k.NombreAtributo, k.Valor);
            }
        }
```

Figura 26. Manejo de rellenado de petición, parte 1

Como podemos apreciar en la Figura 26, lo primero que hacemos al recibir un DTO es crear una ventana para que el usuario nos confirme después si el valor generado es el que desea. También hacemos una comprobación de seguridad del espacio de nombres XML que hemos recibido, para asegurarnos de que es correcta.

Una vez hecho esto, iteramos por cada atributo del DTO, si al tratar de convertir a DTO falla, significa que es un atributo, por lo que le volvemos a dar el espacio de nombres correspondiente por ser un atributo, le mostramos al usuario el valor asignado en caso de que lo quiera cambiar, y lo escribimos en la petición como hemos visto anteriormente.

Vale la pena recordar que los atributos de un DTO tienen dos espacios de nombres, y dependiendo de dónde estemos mirando la referencia será uno u otro. Debemos revisar a menudo que el espacio de nombres XML asignado sea el correcto.

Si no es un atributo, puede ser un DTO, o también un Enum. Veamos cómo tratamos los enumerados.

```
if (k.ToDTO().Atributos.Count == 0)
{//es un enum
    List<String> posVals = Rellenado.GenRules.GenEnum(k.ToDTO());
    ventana.rellenarForm(k.NombreAtributo, k.Tipo, "Enum", posVals, k.Namespace);
    if (ventana.ShowDialog(this) == DialogResult.OK)
    {
        k.Valor = ventana.getValor();//atributo
        escribirValoresPeticion(k.Namespace + ":" + k.NombreAtributo, k.Valor);
    }
    continue;
}
```

Figura 27. Manejo de rellenado de petición, parte 2

Como se puede ver en la Figura 27, podemos saber que un DTO es un enumerado cuando su lista de atributos está vacía. Obtenemos sus posibles valores del .dll y se los mostramos al usuario para que escoja uno, y lo escribimos en la petición como si fuera un atributo.

```
var result = MessageBox.Show("Hemos detectado una colección, ¿quieres añadir un elemento? Se a
String staging = "";
while (result == DialogResult.OK)
{
    String openingTagSearch;
    String closingTagSearch;
    if (k.Namespace != "" && k.Namespace != null)
    {
        openingTagSearch = $"<{k.Namespace + ":" + k.NombreAtributo}>";
        closingTagSearch = $"</{k.Namespace + ":" + k.NombreAtributo}>";
    }
    else
    {
        openingTagSearch = $"<{k.ToDTO().Namespace + ":" + k.NombreAtributo}>";
        closingTagSearch = $"</{k.ToDTO().Namespace + ":" + k.NombreAtributo}>";
    }
    int openingTagPositionSearch = TextBoxXML.Text.IndexOf(openingTagSearch);
```

Figura 28. Manejo de rellenado de petición, parte 3

Como ilustra la Figura 28, en caso de que finalmente sea un DTO, preguntamos al usuario si quiere añadirlo a la petición, y declaramos una cadena de texto que contendrá el contenido que el usuario ha generado.

Si el usuario decide generar el DTO, buscamos la etiqueta del DTO que sea padre de este en la petición, que puede ser que no tenga su espacio de nombres correctamente colocado, con lo que lo revisamos previamente.

```
Entities.DTO ga = k.ToDTO();
if (!Namespaces.ContainsValue(ga.Namespace))
    ga.Namespace = getNamespace(ga.NamespaceKey);

TextBoxXML.Select(openingTagPositionSearch + openingTagSearch.Length, 0);
TextBoxXML.Text = TextBoxXML.Text.Insert(TextBoxXML.SelectionStart, Environment.NewLine + $"<{ga.Namespa

confirmarValoresDTO(ga);//DTO, recursivo

//la idea es meter el texto de dentro aqui, conforme escribas para que no se confunda con el que ya has
int closingTagPositionSearch = TextBoxXML.Text.IndexOf(closingTagSearch);
TextBoxXML.Select(openingTagPositionSearch + openingTagSearch.Length + 4, closingTagPositionSearch - (op
string staging2 = TextBoxXML.SelectedText;
if (!k.Collection && k.ToDTO().Atributos.Count != staging2.Count(f => f == '<') / 2)
{
    staging += TextBoxXML.SelectedText;
    TextBoxXML.SelectedText = "";
}

//hay DTOs que no admiten más de 1 elemento, que lo sabemos por si son colecciones o no (P.Ej Place)
if (k.Collection)
    result = MessageBox.Show("Hemos detectado una colección, ¿quieres añadir un elemento? Se añadirán "
else
{
    result = DialogResult.Cancel;
}
```

Figura 29. Manejo de rellenado de petición, parte 4

Como podemos ver en la Figura 29, si encontramos la etiqueta en el texto, insertamos la del DTO y llamamos de forma recursiva, que encontrará en el texto la etiqueta, y hará lo mismo para cada atributo que tenga. Una vez hemos metido un elemento, lo quitamos del texto y lo guardamos en memoria, en la cadena de texto que hemos declarado antes, y revisamos si es una colección.

Los DTO pueden ser o no colecciones. Si lo son, admiten 0, 1 o más elementos. Si no lo son, admiten 1 elemento, y opcionalmente 0, en caso de poder ser nulos. Si resulta ser una colección, volvemos a preguntar si el usuario desea generar otro elemento, y volvemos a hacer todo el procedimiento. En caso contrario paramos.

Si no encontramos la etiqueta en el texto, la tenemos que generar, y entonces podemos realizar el proceso anterior. Esto es algo complejo, porque tenemos que recordar que todo atributo tiene dos espacios de nombres XML. La siguiente figura 30 ilustra el código para manejar dicha situación:

```
if (openingTagPositionSearch < 0)
{
    if (!Namespaces.ContainsValue(k.Namespace))
        k.Namespace = getNamespace(g.NamespaceKey);
    ventana.rellenarForm(k.NombreAtributo, k.Tipo, "AtributoDTO de " + g.Nombre, k.Valor, k.Namespace);
    if (ventana.ShowDialog(this) == DialogResult.OK)
    {
        k.Valor = ventana.getValor();//atributo

        escribirValoresPeticion(k.Namespace + ":" + k.NombreAtributo, "");
    }

    if (k.Namespace != "")
    {
        openingTagSearch = $"<{k.Namespace + ":" + k.NombreAtributo}>";
        closingTagSearch = $"</{k.Namespace + ":" + k.NombreAtributo}>";
    }
    else
    {
        openingTagSearch = $"<{k.ToDTO().Namespace + ":" + k.NombreAtributo}>";
        closingTagSearch = $"</{k.ToDTO().Namespace + ":" + k.NombreAtributo}>";
    }
    openingTagPositionSearch = TextBoxXML.Text.IndexOf(openingTagSearch);



    TextBoxXML.Select(openingTagPositionSearch + openingTagSearch.Length, 0);
```

Figura 30. Manejo de rellenado de petición, parte 5

Revisamos como de costumbre los espacios de nombres, y solicitamos el valor al usuario, insertamos el texto que nos faltaba, y podemos proceder con normalidad.

Una vez finalizado todo este proceso, tendremos en memoria el texto que tenemos que insertar, por lo que lo insertamos y tenemos finalmente el DTO de la petición rellenado.

5.5.5 Diseño modular de generadores

Echemos ahora un vistazo a otro aspecto del proyecto: los generadores, y la estructura de *plugins*. Empecemos recordando cómo interactuamos con los generadores:

```
var generator = (IValueGenerator)Activator.CreateInstance(type);
string generatedValue = generator.GenCustomValor(a);
if (generatedValue != "Tipo no definido!!")
    genVals.Add(generatedValue);
```

Figura 31. Generación de valores por plugins

Como podemos ver en la Figura 31, se llama a la función GenCustomValor() de cada .dll, y lo que devuelve lo añadimos a una lista que usaremos posteriormente para determinar el valor a escoger.

La instancia del generador es de tipo IValueGenerator, que es una interfaz que definimos para los generadores, define un par de funciones, y la estructura es la siguiente:

```
7 references
public interface IValueGenerator
{
    2 references
    string GenCustomAtributo(Entities.AtributoDTO atrib);
    2 references
    string GenCustomValor(Entities.ValorPeticion valor);
}
```

Figura 32. Interfaz de generación de valores

Como podemos ver en la Figura 32, tenemos dos métodos que devuelven cadenas de texto, y toman un atributo y un valor respectivamente.

La ventaja de esta estructura es que tenemos acceso desde los generadores a toda la información que la herramienta ha inferido del .dll, como pueden ser nombres, tipos, espacios de nombres, si son colecciones, y más.

Veamos ahora un ejemplo de generador que se ha hecho a modo de plantilla:

```
0 references
public class DemoGenerator : IValueGenerator
{

    2 references
    string IValueGenerator.GenCustomAtributo(AtributoDTO a)
    {
        if (a.NombreAtributo.Contains("DNI")) { return GenDNI(); }
        if (a.Tipo == "Boolean") { return "false"; }
        if (a.Tipo == "Int32") { return "1"; }
        if (a.NombreAtributo.Contains("Date")) { return "2002-02-09T00:00:00"; }
        if (a.NombreAtributo.Contains("Email")) { return "test@mail.es"; }
        if (a.NombreAtributo.Contains("Comments")) { return "Comentarios importantes"; }
        if (a.NombreAtributo.Contains("NhsNumber")) { return "04563453"; }
        if (a.NombreAtributo.Contains("Surname")) { return "Díaz"; }
        if (a.NombreAtributo.Contains("MiddleName")) { return "Andreu"; }
        if (a.NombreAtributo.Contains("Name")) { return "Pepe"; }
        if (a.NombreAtributo.Contains("PlaceOfBirth")) { return "Valencia"; }
        return "Tipo no definido!!";
    }

    2 references
    string IValueGenerator.GenCustomValor(ValorPeticion a)
    {
        if (a.Nombre == "extensionToLog") { return "8"; }
        if (a.Nombre.Contains("Date")) { return "2002-02-09T00:00:00"; }
        return "Tipo no definido!!";
    }
```

Figura 33. Generador Plantilla

Como podemos ver en la Figura 33, este generador implementa ambas funciones, y devuelve según ciertas condiciones un valor para el atributo o valor de petición solicitado. Tenemos acceso al nombre y tipo, y podemos hacer reglas complejas para determinar los valores que queremos devolver. Además, podemos generar valores fijos o aleatorios, como por ejemplo con DNIs, al que hemos asociado un generador aleatorio de número de DNI válidos [4].

Y finalmente, para acabar con esta sección, veamos cómo solucionamos uno de los problemas que mencionamos anteriormente: las peticiones largas.

5.5.6 Tratado de peticiones largas

Para recordar el problema en cuestión, las peticiones largas son aquellas que son autogeneradas con unas 400.000 líneas y son esencialmente imposibles de rellenar manualmente, requiriendo de una “poda” eliminando gran parte de la petición, pero esto requiere de un alto conocimiento de la estructura interna del programa del ERP y su API, algo que los usuarios asignados al puesto carecen. Para solucionar esto, podemos hacer un acortador de peticiones, y que la herramienta se encargue de regenerar la petición con la información con la que ya contamos, como hemos explicado en las últimas páginas.

La estrategia que vamos a seguir es sencilla: queremos conservar los valores de petición, porque no tenemos forma de obtenerlos, dado que no tenemos acceso al WSDL. También queremos deshacernos del cuerpo de todos los DTO presentes en la

petición. Esto parece sencillo, pero la manera directa de hacerlo es extremadamente lenta, requiriendo de una iteración por las 400.000 líneas del documento para deshacernos de más de un 99% del mismo. Esto es poco eficiente, y podemos aprovecharnos de la estructura y propiedades de XML para reducir el tiempo significativamente.

La manera en la que podemos hacer esto es recorrer el documento de arriba a abajo, y detenernos al encontrar un DTO. Seguidamente de esto, recorremos el documento de abajo a arriba, buscando la etiqueta XML que cierra ese DTO, y una vez lo encontremos, podemos ignorar todas las líneas que quedan entre los dos puntos que hemos encontrado.

Esto es muy eficiente, y aprovecha las propiedades de XML, pero es sencillo únicamente si cargamos en memoria todo el documento, cosa que no queremos ni podemos hacer porque entonces caeríamos en el mismo error que los otros programas que ya usamos: un fallo de falta de memoria en el heap.

La solución es recorrer el documento como un Stream, para evitar la carga de todo el documento en memoria. No obstante, esto presenta un problema, porque los Streams no se pueden recorrer al revés. Para esto usaremos una clase obtenida de Internet, que se ocupa de leer un Stream al revés y darlo como un Stream normal, evitando la carga en memoria. Tiene algunas limitaciones con respecto a la codificación soportada, pero no aplican a nuestro caso. No vamos a enseñar el código de este lector de Streams al revés, pero se puede encontrar en el post de StackOverflow[22] que dejamos en la bibliografía para su consulta.

Veamos donde estamos ahora en el código, recordemos que hemos encontrado la etiqueta de apertura de un DTO, con lo que queremos omitir su cuerpo. Para más contexto, estamos leyendo del archivo openFileDialog, y tenemos un archivo temporal outputFilePath2, así como el archivo de la petición acortada donde estamos escribiendo que se llama outputFilePath.

[22] https://stackoverflow.com/questions/452902

```
List<string> append = new List<string>();
reversed.MoveNext();
do
{
    String u = reversed.Current;
    append.Add(u);
    if (u.Contains(dtotag))
        break;
} while (reversed.MoveNext());
//como estamos leyendo al reves, pues hay que darle la vuelta a todo otra vez
append.Reverse();
bool firstLineIns = false;
//cerramos el lector pq lo vamos a cambiar
reader.Close();
reader.Dispose();
//borramos el archivo temporal
File.Delete(outputFilePath2);
StreamWriter writerTemp = new StreamWriter(outputFilePath2);
```

Figura 34. Procesado de peticiones largas, parte 1

Como podemos ver en la Figura 34, declaramos una lista de cadenas de texto, que es lo que vamos a fusionar con el archivo de la petición acortada. Hasta ahora, dicho archivo contiene toda la petición hasta la etiqueta de apertura del DTO.

Procedemos a iterar por la petición completa al revés, indicado por el Stream reversed, hasta que encontramos la etiqueta que cierra al DTO.

Mientras no lo encontremos, añadimos todas las líneas del texto a la lista. Una vez hayamos dejado de iterar, le damos la vuelta a la lista, porque estamos leyendo al revés. Cerramos el lector de la petición larga, y creamos un archivo temporal.

```
foreach (string j in append)
{
    if (!firstLineIns)
    {
        firstLineIns = true;
        writer.WriteLine(j);
    }
    else
        writerTemp.WriteLine(j);
}
writerTemp.Close();
writerTemp.Dispose();
//actualizamos el contador para el usuario :)
lns = max - append.Count - 1;
prog.updateLabel(lns);

//y reset de datos a los normales para continuar iterando
dtotag = "";
reader = new StreamReader(outputFilePath2);
```

Figura 35. Procesado de peticiones largas, parte 2

Como ilustra la Figura 35, insertamos todas las líneas excepto la primera en el archivo temporal, poniendo la primera en el archivo final. Cerramos el escritor del archivo

temporal y redireccionamos el Stream de lectura de la petición al archivo temporal, haciendo así que el lector “salte” todas las líneas que no queríamos tener. Finalmente nos deshacemos de todos los lectores y escritores para cerrar los archivos.

Y con esto, tenemos una visión global y en detalle del diseño, arquitectura, funcionalidad y procedimiento de la herramienta desarrollada.

5.6 Pruebas

En esta sección destacaremos las diversas pruebas que hemos realizado sobre la herramienta para verificar que el producto es el esperado

Recordemos que contamos con unas métricas anteriormente definidas. En esta sección vamos a revisarlas una a una.

Debemos tener una herramienta que sea usable, y el objetivo concreto que nos habíamos propuesto era el de "dar la petición a la herramienta y que obtengamos una petición rellena y válida con el mínimo esfuerzo".

Para verificar esto, propusimos una métrica de tiempo cronometrado de documentación de una petición, y podemos afirmar que esto se ha verificado manualmente de forma satisfactoria.

Consideramos también que hemos cumplido el requisito de mantenibilidad, ya que el código está documentado en detalle, y hemos redactado guías del usuario, del mantenedor y del redactor de generadores.

Para el requisito de uso de CPU y RAM propusimos un bajo uso de CPU y un uso de memoria de unos 150 MB, que no superamos por bastante margen, llegando a los 110-120 MB. Esto se ha verificado de forma manual.

Para el requisito de espacio del disco duro, propusimos que la aplicación completa debiera ocupar menos de 50 MB. Podemos decir que este requisito se ha cumplido con creces, ya que ocupa menos de 5 MB.

Para terminar con las pruebas de rendimiento, propusimos que el uso de memoria durante el uso de peticiones largas y archivos grandes no supere un nivel excesivo. Gracias al uso de Streams, se evitan los problemas presentes en los otros programas que estudiamos como alternativas.

Hablemos ahora del cumplimiento y verificación de objetivos.

Propusimos que la herramienta hiciera posible el manejo de peticiones largas, lo que ha conseguido satisfactoriamente, verificando de manera manual.

Otro punto notable fue que debía ser capaz regenerar peticiones a partir de una incompleta, usando la información de los DTO proporcionada por la empresa. Hemos

verificado esto usando pruebas automatizadas que verifican contra peticiones completas y correctas.

También propusimos que debía ser capaz de limpiar peticiones y ser capaz de editar XML de forma cómoda. Esto se verifica siempre en tiempo real y de forma automática con la librería de XML de Microsoft. Dicha librería no permite el uso de XML mal formados, con lo que verificamos que nuestro XML es válido.

Y por último, nuestra herramienta debía ser capaz de generar valores para los campos de las peticiones, así como proporcionar una manera sencilla de integrar nuevos generadores de datos. También debe de ser capaz de generar peticiones válidas. Esto se ha verificado de manera automática contrastando con peticiones ya rellenadas.

En resumen, podemos decir que la herramienta ha pasado todas las pruebas de aceptación, y funciona correctamente, cumpliendo además con los objetivos de usabilidad, rendimiento y eficiencia propuestos.

5.7 Guía de uso de la aplicación

La aplicación se abre a una interfaz muy similar a la del prototipo, con un conjunto de acciones disponibles usando botones, una caja de texto para la petición, y un visor de estructura del XML, así como un apartado para mensajes del programa. Veamos una foto de la interfaz:

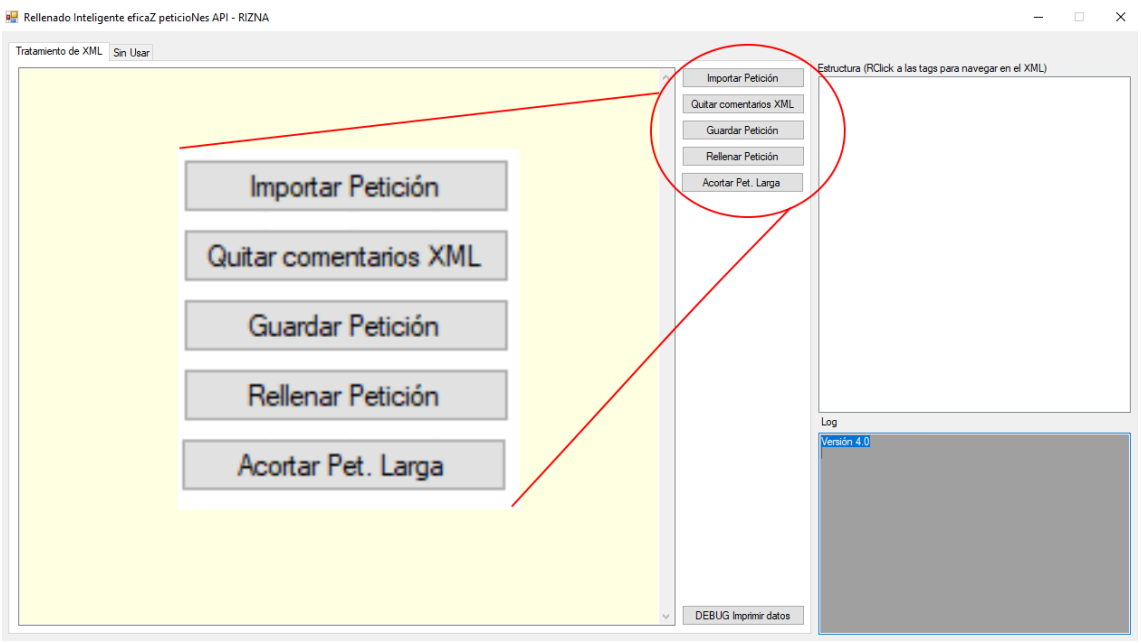


Figura 36. Aplicación recién iniciada

Como podemos ver en la Figura 36, hay 3 huecos con diferentes colores: el amarillo es para la petición XML, el blanco será ocupado para la vista XML, y el gris es para los mensajes de la aplicación.

Si cargamos una petición, ya sea con el botón designado o copiándola, veremos que se actualiza la vista para reflejar los cambios:

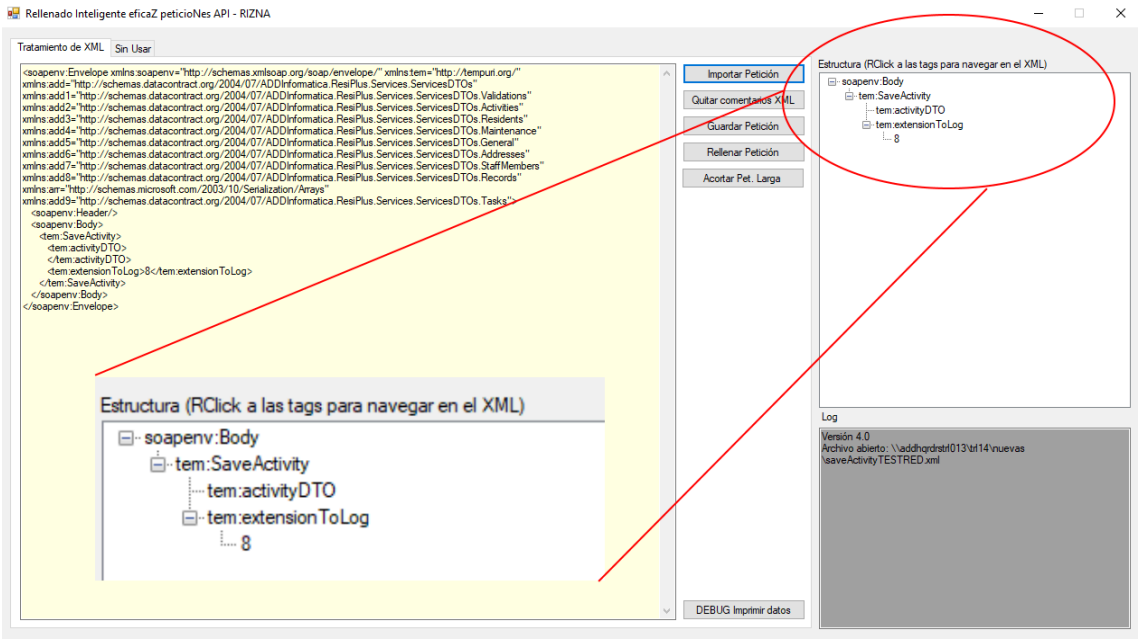


Figura 37. Aplicación con una vista actualizada

Tal y como muestra la Figura 37, haciendo click derecho en las etiquetas (algunos ejemplos son tem:extensionToLog o tem:SaveActivity) seleccionará su contenido. Esto es muy útil para “podar” cuerpos enteros de DTO, o editar campos a mano, copiar sus valores, o seleccionar y ver claramente el cuerpo de alguna estructura.

El botón llamado “Quitar comentarios XML” hace exactamente lo que dice: quita los comentarios, y también elimina las partes innecesarias de “ValidationErrors” que ya hemos comentado anteriormente, además de comprobar que la estructura XML sea válida y también tabula correctamente el documento.

También podemos guardar la petición, o acortar una petición larga. Esta última opción nos pedirá un documento que contenga la petición larga, abriendo un diálogo de explorador de Windows para buscarlo. Si le indicamos el archivo, procederá a acortarlo siguiendo el procedimiento que hemos descrito en una sección anterior, con una barra de progreso y una notificación cuando acabe. Aquí tenemos una foto de la notificación, como se ve en la Figura 38:

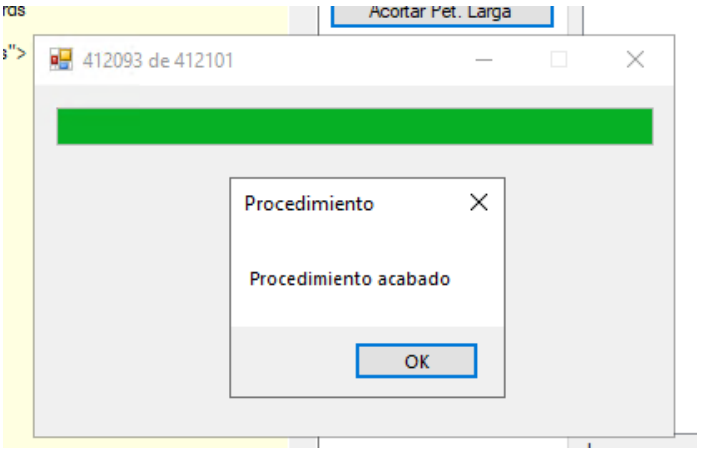


Figura 38. Aplicación con la notificación de acortar peticiones largas

Y finalmente, nos queda la parte más importante del programa: el rellenado de peticiones. Cuando hacemos click sobre el botón de rellenar petición y hay una petición válida en la caja de texto, tendremos una ventana por cada uno de los atributos y valores que podemos rellenar, y podremos indicar qué valor queremos, así como ver información diversa del elemento que estamos editando y qué valor nos sugiere el programa. Veamos dicha ventana en acción:

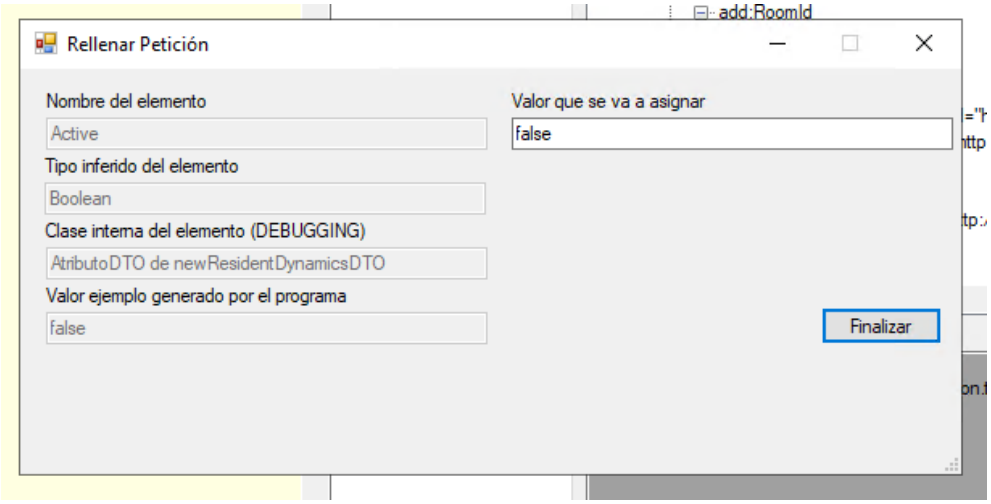


Figura 39. Aplicación con la ventana de rellenar petición

Como podemos ver en la Figura 39, estamos viendo un atributo del DTO newResidentDynamicsDTO, que tiene por nombre Active, y es de tipo Boolean. La herramienta nos sugiere el valor “false”. Podemos editar libremente el campo de “Valor que se va a asignar”. En el caso de un enumerado, la ventana es similar pero tendremos un DropDown con los distintos valores del enumerado.

Una vez hayamos editado nuestra petición, debemos copiarla de vuelta a SOAPUI o guardarla a un archivo de texto.

5.8 Cronología del proyecto

La filosofía de desarrollo de este proyecto ha sido la de tomar una lista de requisitos, priorizarlos, y darles una clara definición de qué es lo que se pretende conseguir con cada requisito. Esto es importante para no caer en definiciones ambiguas, estimando de forma incorrecta o añadiendo funcionalidad demasiado desarrollada a coste de otras más importantes [1].

Una vez hecho esto, se han definido secciones en el Backlog según los objetivos propuestos de cada Sprint. Hemos hecho Sprints de duración variable ya que esto nos permite visualizar rápidamente qué funciona y qué debe ser replanteado, y en los Sprints más largos, poder desarrollarlo en detalle.

Este proyecto ha sido desarrollado a lo largo de 18 semanas, desde un par de semanas antes de diciembre de 2023, hasta mitad de abril de 2024, incluyendo en este tiempo investigación, recogida de requisitos, programación y redacción de este documento. En total, unas 360 horas.

Empecemos con una visualización general de la línea de tiempo del proyecto:

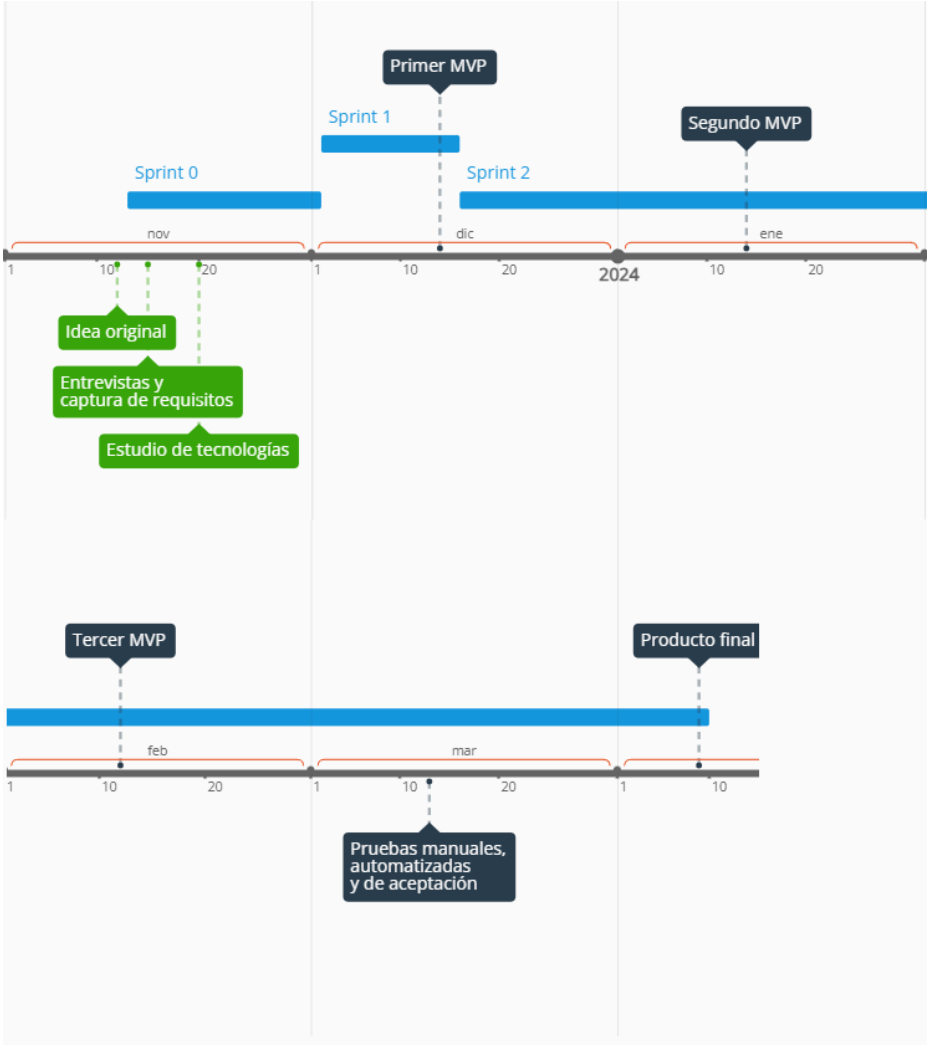


Figura 40. Línea temporal

Viendo más en detalle lo expuesto en la Figura 40, veamos un desglose en detalle:

13/11/2023 - 02/12/2023 Sprint 0: Captura de requisitos y brainstorming, entrevistas con los miembros del departamento, estudio de tecnologías, alternativas, y posibles diseños de arquitectura. Los resultados de estas investigaciones están expuestos anteriormente.

02/12/2023 - 16/12/2023 Sprint 1: Diseño de interfaz, implementación de carga y guardado de XML, y limpieza e inferencia de peticiones.

Llegados a este punto la ruta que se iba a tomar para obtener la información de los componentes de una petición iba a ser usando el proyecto de Visual Studio de Models, leyendo de alguna manera el código fuente. No obstante, se nos presentó la posibilidad de usar el .dll de la empresa, lo que ha resultado ser una gran ventaja. Empezamos el Sprint 2 con esta idea ya asentada en el diseño.

16/12/2023 - 10/04/2024 Sprint 2: Programación de la solución, pruebas manuales y de aceptación. En este Sprint se ha realizado todo lo relacionado con rellenado de peticiones, lectura de .dll, solución de errores, y depuración.

Al final del primer Sprint contábamos con una revisión, y a lo largo del Sprint 2 hemos tenido tres revisiones más.

La primera revisión mostraba la interfaz, con la vista XML, y limpiar, importar y guardar peticiones, y marca el final del Sprint 1.
La segunda revisión contaba con la inferencia de la estructura existente en la petición.
La tercera revisión tenía por objetivo definir los generadores y su interacción completa.
La cuarta y final revisión hizo que se pudieran regenerar peticiones a partir de un conjunto de datos muy escaso, y también hizo posible el acortar peticiones. Marca el final del proyecto.

Hablemos ahora del backlog completo del proyecto. Para el Sprint 1 teníamos los siguientes ítems:

- Limpiar una petición
- Importar una petición
- Guardar una petición
- Poner una petición con tabulación

Como podemos ver, muchos de estos son Épicas o Casos de Uso. No hemos dividido en acciones más pequeñas porque tampoco son tan complicadas.

Asignado al Sprint 2 teníamos lo siguiente:

- Inferencia de DTO desde código fuente
- Rellenador de peticiones automático
- Inferencia de DTO desde petición

Fuera del proyecto, tenemos lo siguiente:

- Integración con Word
- Integración con WSDL
- Buscador del proyecto de Visual Studio Models
- Generador de expresiones regulares para respuestas de petición

No obstante, los requisitos cambian, y muchas veces surgen ideas que serían grandes añadidos al proyecto. Esto se ve reflejado en la evolución que ha tenido el tablero Kanban de Trello que tenemos para cada uno de los Sprints.

Para el primer Sprint añadimos:
- Varios errores que fueron arreglados
- Navegar el XML usando la vista XML

Y para el segundo Sprint añadimos:
- Uso del .dll y programación de clases Wrapper a los objetos oficiales
- Una refactorización de la estructura de clases entidad (DTO, atributoDTO, valorPeticion, etc)
- Tratamiento de errores para peticiones malformadas
- Diseño recursivo para tratamiento de DTO
- Diseño de uso de *plugins* en vez de usar tuplas en archivos de texto separados, lo que permite libre distribución y más libertad a la hora de generar valores
- Regeneración de peticiones e insertado de datos no presentes en la petición
- Acortar peticiones largas

Como ya hemos expuesto a lo largo de este documento, todos estos objetivos han sido cumplidos excepto los que quedaron fuera del proyecto desde un principio.

6. Conclusiones y Trabajo Futuro

Podemos decir que todos los objetivos principales del proyecto se han cumplido. Por supuesto, aún quedan mejoras y funcionalidades en el tintero, pero son secundarias y no hay tiempo para hacerlas.

En materia de funcionalidades que nos propusimos y hemos conseguido, tenemos:

- Hacer posible el manejo de peticiones grandes, borrando los comentarios y las secciones innecesarias, además de los cuerpos de DTO que no sean necesarios.

- Regenerar peticiones a partir de una incompleta, usando la información de los DTO proporcionada por la empresa para este fin.

- Limpiar peticiones de comentarios y darles formato tabulado.

- Proporcionar valores ejemplo para los campos de una petición proporcionada por el usuario.

- Tener una herramienta para editar XML de forma más cómoda, con navegación entre campos, selección de contenido y más.

- Proporcionar una manera fácil y sencilla de integrar nuevos generadores de datos personalizados a las necesidades de cada servicio a documentar.

Nos propusimos implementar requisitos de rendimiento, usabilidad y eficiencia que hemos conseguido alcanzar con un buen diseño. Gracias a las pruebas de aceptación podemos comprobar que la herramienta hace, en efecto, peticiones correctas tan rápido como el usuario introduzca los datos, sin necesidad de recordar nada de la estructura interna de la API o del XML. El resultado es una herramienta de uso sencillo, potente y eficaz. Se han satisfecho todas las necesidades básicas que nos hemos propuesto en este proyecto.

En cuanto a mi formación, sin duda la asignatura que viene a la mente es la de Integración e Interoperabilidad (IEI), que trata con los temas de XML, SOAP, y demás tecnologías pilares de este proyecto. De no ser por esta asignatura, no tendría el dominio de XML y SOAP que me ha permitido realizar este TFG.

En el procedimiento han sido clave las asignaturas de Proceso del Software y Proyecto de Ingeniería (PSW y PIN), en ambas nos entrenamos en metodologías ágiles, que es la que hemos usado en este proyecto para desarrollarlo y planificar las pruebas de aceptación del mismo, lo que ha influenciado en gran medida decisiones de diseño, como por ejemplo los requisitos de rendimiento y de usabilidad.

Otra asignatura que ha sido indispensable en este proyecto es Diseño del Software (DDS), que me ha permitido aplicar los conocimientos de patrones de software para hacer funcionalidades complejas de forma escalable y sencilla.

Finalmente, en lo que a lo personal y profesional respecta, puedo asegurar con toda confianza que este proyecto no solo me ha ayudado al enseñarme buenas técnicas de programación para afrontar ciertos problemas, como por ejemplo el uso de *plugins*, sino que también me ha ayudado a hacer que el trabajo en mi puesto, que es documentador de API, sea infinitamente más llevadero.

En el frente de trabajo futuro e ideas en el tintero, hay muchas, pero por detallar algunas:

- Integración WSDL: Esto es algo que me encantaría haber hecho, porque supondría un replanteamiento entero de la aplicación para en vez de pasar por SOAPUI para generar las peticiones, se generarían automáticamente. El problema es que no tenemos una librería para interpretar WSDL, y al ser un lenguaje pensado para máquinas, es increíblemente complicado hacer un intérprete para tan poca ganancia, teniendo en cuenta el coste de tiempo y esfuerzo, los posibles errores que puedan surgir, y demás.
- Generación de expresiones regulares para los resultados: Esto es algo que no es muy complicado de hacer para ciertos resultados conocidos (true y false, Ids, cadenas de texto…) pero no tenemos forma de saber qué estructura o contenido tienen que tener las respuesta, y tendríamos que ir leyendo la respuesta para determinar qué expresión regular corresponde. Además, hay una guía del documentador escrita con una sección dedicada a esto, que está hecha para ser copiada.
- Generar documentación en Word automáticamente: Algo complicado de hacer porque tendríamos que usar las librerías de Word, tener Office instalado, y realmente ahorraríamos muy poco tiempo. Todo esto sin contar además que el Word que estamos editando está en otra máquina, dentro de un sistema de documentos compartidos en el que además hay un bloqueo por usuario que edite el documento. Es innecesariamente complicado para simplemente ahorrar unos segundos.
- Búsqueda de elementos en el proyecto Models: La API tiene un proyecto que se llama Models, describe muchas estructuras y peticiones, así como qué tablas de qué bases de datos se consultan en cada petición. Es algo que hubiera estado bien tenerlo, pero realmente solo ahorramos un Ctrl+Shift+F (Buscar en toda la solución) al usuario, y es posible que ni siquiera esté descargado el proyecto, o que esté en un sitio diferente en cada máquina.

7. Referencias

1. Kniberk, H., 2015, *Scrum and XP from the Trenches* (2nda Edición). C4Media

2. Richardson, L., Ruby, S., 2007, *RESTful Web Services*. O'Reilly.

3. Kanat-Alexander, M., 2012, *Code Simplicity: The Fundamentals of Software*. O'Reilly

4. Gałęzowski, G., 2023, *Test-Driven Development: Extensive Tutorial*. Leanpub

Anexo ODS

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible

Objetivo de Desarrollo Sostenible	Alto	Medio	Bajo	No Procede
1 Fin de la pobreza				X
2 Hambre cero				X
3 Salud y bienestar		X		
4 Educación de calidad				X
5 Igualdad de género				X
6 Agua limpia y saneamiento				X
7 Energía asequible y no contaminante				X
8 Trabajo decente y crecimiento económico			X	
9 Industria, innovación e infraestructuras			X	
10 Reducción de las desigualdades				X
11 Ciudades y comunidades sostenibles				X
12 Producción y consumo responsables				X
13 Acción por el clima				X
14 Vida submarina				X
15 Vida de ecosistemas terrestres				X
16 Paz, justicia e instituciones sólidas				X
17 Alianzas para lograr objetivos				X

Relación del TFG con los ODS

El trabajo realizado tiene un impacto positivo en la sociedad ya que forma parte de una mejora para dotar a la sociedad de un ERP para el ámbito sociosanitario de calidad.

El objetivo más destacable, por tanto, es el de “Salud y bienestar”. El ERP contribuye a mejorar las vidas de los residentes de tercera edad y grupos de atención especial. Al dotar de una mayor calidad a la API que está en uso por numerosos colaboradores, se

hace posible dar un mayor cuidado a estos grupos, previniendo que el programa y la infraestructura tengan fallos y funcionen correctamente.

También podemos marcar “Trabajo decente y crecimiento económico”, ya que con nuestra solución contribuimos en gran medida a mejorar el trabajo del encargado de documentar la API. Contribuimos a que sea un puesto más llevadero, un trabajo decente y motivador.

Y finalmente, marcamos también “Industria, innovación e infraestructuras”, ya que estamos mejorando la infraestructura del ERP y de la API, de la fase de pruebas, y estamos innovando para conseguir ese objetivo.