Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos.

Itinerario Ingeniería de Software (Código 10501)

Generador Automático de Pruebas sobre Servicios Web's.

Autor: José Antonio Pérez Reyes / josea.perezr@gmail.com.

Director: Ismael Abad Cardiel.

Curso: 2015 / 2016.

Convocatoria: Junio 2016.

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos.

Itinerario Ingeniería de Software (Código 10501)

Generador Automático de Pruebas sobre Servicios Web's.

Generador automático de código de pruebas unitarias bajo el lenguaje Java como DSL, sobre servicios web's para dar soporte a tareas de comprobación de funcionamiento y cumplimiento de requisitos.

Autor: José Antonio Pérez Reyes / josea.perezr@gmail.com.

Director: Ismael Abad Cardiel.

Impreso TFdM05_Autor. Autorización de publicación y difusión del TFdM para fines académicos

Autorización

Autorizo/amos a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del autor.

PEREZ REYES JOSE ANTONIO - 42234575C

Firmado digitalmente por PEREZ REYES JOSE ANTONIO - 42234575C
Nombre de reconocimiento (DN): c=ES, serialNumber=42234575C, sn=PEREZ REYES, givenName=JOSE ANTONIO, cn=PEREZ REYES JOSE ANTONIO - 42234575C
Fecha: 2016.06.13 22:25:42 +02'00'

Juan del Rosal, 16
28040, Madrid

Tel: 91 398 89 10
Fax: 91 398 89 09

www.issi.uned.es

RESUMEN

En el presente documento se estudia y analiza como problema la ausencia de código de cobertura de pruebas sobre servicios web's implementados. Y se analiza e implementa una generador de código que de cobertura de pruebas sobre un servicio determinado. Se desarrolla un prototipo funcional que realice la generación de los casos de pruebas para cada operación (método) que contenga el servicio web indica.

Estará enfocado a servicios web's y a pruebas funcionales. Y se basará en una herramienta desarrollada en versión prototipo, pero que a su vez está orientada a ser un producto que solvente las necesidades expuestas.

Por tanto, se desglosará: el problema, el diseño del generador propuesto, su implementación a rasgos generales y una detallada explicación de los casos de uso que cubre este generador automático de pruebas unitarias.

Este prototipo será desarrollado haciendo uso del lenguaje de programación Java como lenguaje específico de dominio (DSL)

LISTA DE PALABRAS CLAVES

SOAP, Junit, web services, generador, Java, Maven, mockup's y framework.

Tabla de Contenidos

Índice de contenido

1 Introducción.

Las pruebas es una fase de gran importancia en el ciclo de desarrollo y mantenimiento de software. Y dado que existen principalmente pruebas de funcionamiento, rendimiento y pruebas de cumplimiento y aseguramiento de requisitos, esto conlleva la necesidad de disponer horas de recursos dedicados a la generación de estas pruebas más recursos especializados en el negocio para definir las pruebas de cumplimiento de requisitos.

Este trabajo esta orientado en el planteamiento de un generador que sea capaz de generar automáticamente el código de pruebas y así cubrir lo antes expuesto, es decir, la generación de todo aquel elemento de programación y recursos necesarios para la realización de las pruebas de un software desarrollado.

El generador propuesto deberá cubrir la necesidad de generación de pruebas unitarias contextualizando el ámbito sobre servicios SOAP. Partiendo de la descripción de un servicio web (en lo sucesivo nos referiremos a ellos con el acrónimo WS), este generador básicamente deberá ser capaz de construir una aplicación que implemente los test's que cubran las operaciones que exponga dicho WS. Como DSL se elige Java, por lo que el software generado será bajo este lenguaje.

En principio el generador partiendo del WSDL generará el conector al WS y generará un proyecto en el DSL especificado, que contendrá todos los test necesarios. Este proyecto lo denominaremos Test Suites, ya que contendrán el conjunto de las pruebas unitarias que implementarán la petición necesaria para hacer la invocación de las operaciones disponibles en el WS, y posteriormente contendrá las condiciones de aserción que permitirá determinar el resultado del test. Adicionalmente este prototipo proporcionara el código fuente de las suites generadas, también permitirá la ejecución de los test y visualización de resultados de la ejecución de caja caso de prueba unitaria generada.

2 Ámbito del generador.

Dado que el contexto de pruebas sobre un sistema puede ser muy amplio ya que puede abarcar diferentes capas de aplicación, como:

- **Presentación**: refiriéndose a todas las interfaces de usuario que tiene un sistema y en donde se debe comprobar el correcto funcionamiento para las distintas entradas y operaciones efectuadas por los usuarios.

- **Lógica**: Se trata de las distintas operaciones definidas para la ejecución por parte del usuario. Incluye el flujo correcto entre ellas y las salidas esperadas por cada operación establecida en el sistema.

- **Acceso a datos**: es la capa de un sistema que contiene los objetos que realizan las operaciones sobre los datos que pueden estar contenidos en una BBDD, fichero, repositorio, etc. Aquí las pruebas son unitarias y se debe comprobar que cada operación de acceso de datos es efectuada correctamente.

- **Integración (WS)**: aquí nos referimos a los posibles servicios web que estén definidos para el consumo por otros sistemas. Y en donde se debe comprobar su funcionamiento en base a las salidas que se esperan de el.

- **Rendimiento**: son las pruebas que se realizan sobre un sistema para conocer su rendimiento en diferentes escenarios de carga de trabajo.

Nos centraremos sobre pruebas generadas sobre capas de servicios web de un sistema. Esto implicaría la generación de pruebas unitarias para cada servicio expuesto, con el fin de realizar pruebas de funcionamiento. Al igual que la generación de pruebas programáticas para el cumplimiento de un requisito determinado que estará definido.

3 Estado del arte.

3.1 Productos más populares existentes en el mercado.

3.1.1. SOAPUI.

Se trata de una herramienta Open Source orientada a la realización de pruebas sobre servicios web multiplataformas. Pudiendo hacer pruebas de funcionamiento, de regresión, de cumplimiento y de carga. Tiene una versión gratuita y una versión profesional mucho más completa. Cuenta con las siguientes características:

- Pruebas funcionales.

Permite generar pruebas funcionales y de regresión sin necesidad de conocimiento de programación.

- Escenarios complejos.

Permite probar un conjunto de pruebas estructuradamente.

- Depuración de pruebas.

Permite realizar el seguimiento del flujo de las pruebas por medio del depurador de la herramienta. Disponible en la versión de PRO.

- Pruebas de Data-Driven.

Creación de pruebas basadas en distintas fuentes de datos como pueden ser: Excel, XML, JDBC y Archivos.

- Soporte Multientorno

Cambio de forma rápida la configuración de las pruebas en función del entorno de destino.

- Generación de Mocks.

Facilita la generación de imitadores de servicios existentes. Este servicio genera una simulación de un WS basándose en la interfaz que este debe tener, haciendo posible por ejemplo poder hacer pruebas de integración entre distintos WS que no estén desarrollados.

3.1.2. JMeter.

Es una herramienta de código abierto desarrollada en java. Originalmente fue diseñada para probar aplicaciones web pero progresivamente fue expandiéndose a otro tipo de pruebas.

Una de las característica de esta herramienta es la capacidad de realizar carga y pruebas de rendimiento sobre servidores del tipo SOAP / REST. Las pruebas que permite hacer esta herramienta sobre servicios web son a nivel de peticiones específicas pudiendo precisar una carga específica pero no pudiendo modelar escenarios complejos.

Cuenta con las siguientes características:

- Pruebas unitarias sobre conexiones a BBDD vía JDBC.
- Pruebas unitarias sobre LDAP.
- Pruebas de funcionamiento y rendimiento sobre web (http/https)
- Framework con manejo de multihilos para la realización de pruebas de estrés.
- Pruebas de consumo sobre servicios SOAP / REST.

3.1.3. WebInject.

Es una herramienta gratuita para pruebas automatizadas de aplicaciones web y servicios web. Puede ser utilizado para probar los componentes individuales del sistema que tienen interfaces HTTP (JSP, ASP, CGI, PHP, AJAX, Servlets, formularios HTML, XML / Servicios web SOAP, REST, etc.), y se pueden utilizar como un instrumento de prueba para crear un conjunto automatizado pruebas funcionales, de aceptación y de regresión. Un mecanismo de pruebas que permite ejecutar muchos casos de prueba y reporta sus resultados. WebInject ofrece resultados en tiempo real y también se puede utilizar para el seguimiento de los tiempos de respuesta del sistema.

WebInject se puede utilizar como un marco de prueba completa que es controlado por la Interfaz de usuario WebInject. Opcionalmente, se puede utilizar como un ejecutor de prueba independiente (texto application / consola) que puede ser integrado y llamado desde otros marcos de prueba o aplicaciones.

Cuenta con las siguientes principales características:

- Casos de pruebas.

Los casos de prueba se escriben en archivos XML, utilizando elementos y atributos XML, y se pasan al motor WebInject para la ejecución sobre la aplicación / servicio que se está probando. Esta abstrae la parte interna de la aplicación de WebInject lejos del probador no técnico, durante el uso de una arquitectura abierta [escrito en Perl] para los que requieren más personalización o modificaciones.

- Reporte de resultados.

Los reportes de resultados son generador en formato HTML y XML Incluyen detalles como pueden ser: errores, test's correctos, test fallidos, tiempos de respuesta, etc.

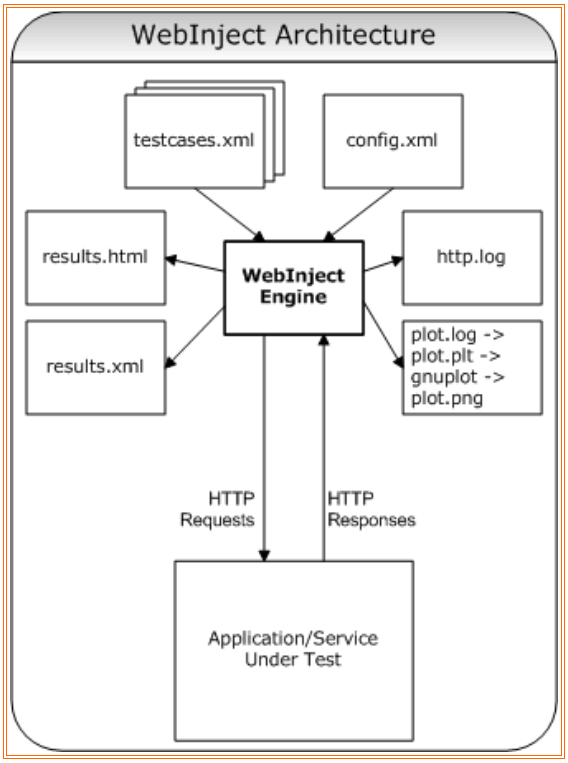


Figura 1: Ilustrción sobre la arquitectura WebInject.

3.2 Criterio de evaluación de soluciones existentes.

Los criterios que consideraremos de importancia en la evaluación de las herramientas más relevantes para la realización de pruebas son los siguientes:

Cod. Criterio	Descripción
CR01 – Facilidad de uso.	Todo lo relacionado a la facilidad para el uso de la aplicación. Sobre todo en la definición de las pruebas.
CR02 – Generación de código.	Contiene la posibilidad de generación de código.
CR03 – Generación de test's independientes.	Generación de pruebas unitarias por método expuesto en un WS.
CR04 - Generación de Test complejos.	Generación de test que incluyan la llamada estructurada a distintos métodos y distintos servicios web.
CR05 – Resultados.	Presentación de resultados.
CR06 – Herramienta libre.	Que se trate de una herramienta totalmente libre.

3.3 Evaluación y comparación de la solución.

Criterio por herramienta.	SOAPUI	Jmeter	WebInject
CR01	8	7	6
CR02	7	0	0
CR03	10	9	7
CR04	9	0	6
CR05	10	8	6
CR06	5	10	10
Totales	**49**	**34**	**35**

3.4 Análisis sobre la comparativa.

CR01 - Facilidad de uso. **Todo lo relacionado a la facilidad para el uso de la aplicación. Sobre todo en la definición de las pruebas.**	
SOAPUI	* Es una herramienta muy intuitiva que permite el diseño de los test fácilmente sin necesidad de conocimiento en programación. * Contiene muchas funcionalidades disponibles lo que dificulta un poco la curva de aprendizaje. * Muchas de las funcionalidades que facilitan el uso de la herramienta están disponibles solo en la edición profesional, la cual es de pago.
Jmeter	* A pesar de que solo se pueden realizar pruebas de consumo sobre métodos de manera independiente, y que no soporta la creación de casos de pruebas complejos la forma de realizar estas pruebas es bastante difícil, ya que por ejemplo una vez se indica el WS por medio del WSDL, la petición debemos realizarla nosotros, es decir, toda la petición xml que forma la petición.
WebInject	* En esta herramienta los casos de pruebas se deben diseñar directamente en formato XML utilizando elementos y atributos XML que son interpretados por el motor de WebInject para realizar la generación y ejecución de las pruebas. Esto dificulta mucho la creación de las pruebas ya que la herramienta no cuenta con una interfaz que permita el diseño de la prueba que genere estos XML's usados por el motor del generador.

CR02 - Generación de código. **Contiene la posibilidad de generación de código.**	
SOAPUI	* En el caso de esta herramienta es posible generar los clientes de consumo de un servicio determinado, incluso generando el fuente necesario para la compilación del cliente generado. Pero en ningún caso la definición de la prueba es generada a ningún lenguaje.
Jmeter	* No dispone de generación de código.
WebInject	* No dispone de generación de código.

CR03 - Generación de test independientes. **Generación de pruebas unitarias por método expuesto en un Web service.**	
SOAPUI	* Permite la generación de pruebas de consumo independiente por métodos de WS. * La generación del envoltorio de la petición es generada automáticamente. * La parametrización es directamente en el formato xml en la versión gratuita y en la versión de pago la herramienta cuenta con una interfaz. * Permite la creación de TestSuite's y añadir casos de aprobación (assert's).
Jmeter	* Permite la generación de pruebas de consumo independiente por métodos de WS. * La generación del envoltorio de la petición y la parametrización es manual directamente en formato xml.
WebInject	* Permite la generación de pruebas de consumo independiente por métodos de WS. * El diseño de los casos de pruebas es directamente en formato XML utilizando elementos y atributos XML que son interpretados por el motor de WebInject.

CR04 - Generación de Test complejos. **Generación de test que incluyan la llamada estructurada a distintos métodos y distintos servicios web.**	
SOAPUI	* Permite probar un conjunto de pruebas estructuradamente bajo escenarios complejos. * Cuenta con interfaz gráfica para el diseño de estos pruebas. * Permite diseñar secuencia de peticiones de un mismo WS.
Jmeter	* No soporta generación de casos de pruebas sobre escenarios complejos.
WebInject	* Es posible utilizarse como un instrumento de prueba para crear un conjunto automatizado pruebas funcionales, de aceptación y de regresión. * No cuenta con interfaz para ello por lo que se convierte en una tarea muy compleja.

CR05 – Resultados. **Presentación de resultados.**	
SOAPUI	* Cuenta con salida de resultados y estadísticas.
Jmeter	* Cuenta con salida de resultados y estadísticas.
WebInject	* Cuenta con salida de resultados y estadísticas.

CR06 – Herramienta libre. **Que se trate de una herramienta totalmente libre.**	
SOAPUI	* Cuenta con una versión gratuita muy completa y una versión de pago que contiene muchas mejoras interesantes y de alto nivel.
Jmeter	* Gratuita.
WebInject	* Gratuita.

4 Solución propuesta.

El siguiente trabajo de investigación se centra en el diseño de un generador que facilite la producción automatizada de pruebas unitarias enfocados a WS's, al igual que la generación de pruebas que congreguen una secuencia de llamadas ordenadas a métodos específicos, con el fin de emular un requisito determinado.

El fin que se persigue es poder realizar la generación de estas pruebas, que en muchos casos son repetitivas, de una forma homogénea, rápida y automática, abarcando un alto porcentaje de un sistema dado y pudiendo mejorar la cobertura de pruebas del mismo de una forma casi inmediata.

Los puntos que se establecen como objetivos que debe alcanzar la herramienta son los siguientes:

- Ser capaz de leer e interpretar el WSDL de un WS y generar los mecanismos de conexión al WS.
- Partiendo de un WSDL generar sobre un mecanismo acorde, el código fuente en el DSL seleccionado (java) que implemente las pruebas unitarias para cada operación expuesta por un WS.
- Aumentar de una forma rápida la cobertura de código.
- Generación automática de código fuente con implementación de pruebas unitarias con entradas de objetos mockups.
- Proporcionar el código fuente generado para su descarga.
- Poder hacer validaciones de cumplimiento de requisitos. Pudiendo cubrir un porcentaje alto de una manera rápida.
- Poder indicar pruebas unitarias que contengan una secuencia de llamadas entre métodos para comprobar el cumplimiento de requisitos definidos para un sistema dado.

Para esto la herramienta centralizará un mecanismo donde sea posible definir de una forma sencilla las reglas que se deben cumplir en cuanto:

- Lectura de un WS indicado.
- Generación de test automáticamente.
- Entradas esperadas por métodos.
- Salidas esperadas por métodos.
- Secuencia de llamadas de métodos.

4.1 Concepto.

4.2 ¿Que solventa?

Se considera la orientación que damos a la solución, como herramienta para la generación automática de prueba, basado en un modelo que a su vez es generado por los datos que serán introducidos por los usuarios finales, sobre los distintos servicios web publicados por un sistema dado. Este generador debe producir el código fuente encargado de realizar las distintas pruebas unitarias y funcionales, de modo que satisfaga la necesidad de generar los mecanismos necesaria para la realización de las pruebas de cobertura para un sistema dado.

El generador debe cumplir todos estos criterios:

- **Facilidad de uso.**

  Debe contener una interfaz que permita indicar los servicios web que se desean testear al igual que esta interfaz facilite el modelado del flujo de llamadas entre distintos métodos de distintos servicios web.

- **Generación de código: contiene la posibilidad de generación de código.**

  La herramienta debe generar el código fuente con los test generados y con la posibilidad de poder ser empaquetados (jar/war). Esto con el fin de poder incrementar la cobertura de un código existente de una manera automatizada.

- **Generación de test independientes: Generación de pruebas unitarias por método expuesto en un WS.**

  El generador haciendo uso del WSDL de un WS, debe generar de forma automatica el código que implementa las pruebas unitarias por operación expuesta.

- **Generación de Test complejos: Generación de test que incluyan la llamada estructurada a distintos métodos y distintos servicios web.**

  Uno de los objetivos de la herramienta generará test unitarios por métodos, incluso inyectando valores mocks para el consumo de los métodos. También podrá generar test complejos que abarquen un flujo que este conformado por peticiones a distintos métodos de distintos servicios web, siguiendo un flujo de trabajo que es preestablecido en las interfaces de parametrización.

- **Presentación de resultados.**

  Debe arrojar todos los resultados posibles de los test ejecutados.

4.3 Java como DSL.

Para la integración con WS's que puedan estar desarrollados con cualquier tipo de lenguaje o framework, es necesario contar con un mecanismo o medio que facilite la producción de los distintos componentes necesarios para hacer uso de esos WS. Por tanto, hemos seleccionado Java como lenguaje DSL, ya que este ya cuenta con unas características y robustez necesaria para llevar a cabo tareas de cierta complejidad, y así beneficiarnos de características, frameworks, librerías y tecnologías basadas en este lenguaje que agilizan el objetivo perseguido.

Los elementos o características de los que hacemos uso del DSL seleccionado, son:

- Junit: se trata de un framework creado para el lenguaje Java, el cual nos facilita la creación y ejecución controlada de pruebas unitarias.

- Maven: Se trata de un framework basado en Java para la gestión y construcción de proyectos. Fundamental para la idea propuesta, dado que el generador produce un software basado en pruebas unitarias, y ese software es construido con esta tecnología. Ya que facilita la inyección de parámetros en los ficheros de configuración para la posterior compilación, ejecución y extracción de resultados.

- Apache CXF: es un framework de código abierto basado en Java y orientado a WS. Usado para la generación de conectores combinado con Maven. Esta tecnología facilita mucho el trabajo de generación de clientes.

- Librería FileUtil: Esta librería tiene implementada una seríe de funcionalidades que facilitan la manipulación de ficheros, lo que nos ayuda en la gestión de los ficheros utilizados como plantillas.

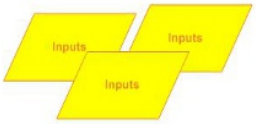
- Ya que producimos un software para su descarga, seleccionamos el lenguaje Java ya que es el más usado en la actualidad.

- Otra de las características fundamentales para el uso de este lenguaje y las librerías mencionadas, es que son libres y no requieren licencia.

4.4 Diseño.

Con el fin de ir conociendo el diseño del generador es necesario conocer la visión más general del flujo que se propone que siga y cumpla el generador. En el siguiente diagrama se ilustra los elementos que intervienen en la generación las pruebas al igual que las salidas del generador.

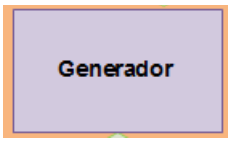
Como ya hemos mencionado que el punto de inicio para el generador sera la descripción del WS, esta vendrá determinada por el fichero estándar usado para este fin WSDL.



Este fichero contiene la información necesaria para poder determinar las siguientes características del WS:

- **Tipos de datos** (types) definidos y usados por el WS.
- **Tipos de puertos (portType).** Puertos definidos por el WS y operaciones y mensajes expuestos en estos.
- **Operaciones (operation)** o métodos que implementa en WS.
- **Mensajes (message),** representas las partes lógicas que representan las peticiones y respuestas de cada una de las operaciones.
- Protocolos de comunicación usados por el WS (**bindings**)
- **Servicios (service)** Conjunto de puertos y dirección de los mismos. Esta parte final hace referencia a lo aportado por las secciones anteriores.

Con estos elementos no podemos saber que hace un WS o como lo hace, pero sí disponemos de la información necesaria para interaccionar con él (funciones, mensajes de entrada/salida, protocolos, etc)



- Se trata del generador, aquí es donde se realiza la lectura e interpretación del WSDL para las posteriores generaciones necesarias. El generador se encarga básicamente de las siguientes tareas:

1. Generación por medio del framework Apache CXF, de los conectores o clientes necesarios para el consumo del WS.
2. Genera bajo petición la implementación de las pruebas unitarias que se encargan de probar el funcionamiento del WS y de los métodos que este contiene.

   Implementara un test por operación que exponga el WS generando automáticamente los valores que debe contener las peticiones según los tipos que se especifiquen en el WSDL.
3. Compilación del fuente generado con las pruebas unitarias.

4. La ejecución de las pruebas implementadas por el generador.
5. Salida de la compilación y ejecución de las pruebas.
6. Generación de informes de resultados de cada test unitario implementado.

Básicamente el producto principal del generador son las pruebas. Al decir pruebas, nos estamos refiriendo al software que es originado por el generador. Se tratan de proyectos construidos con el framework Maven, que concentra en ellos la implementación de las pruebas unitarias, junto con todos aquellos recursos que sean necesarios para la ejecución de ellas.

El generador, posterior a la generación de la suite que contiene las pruebas unitarias, realiza bajo petición la compilación de esta, la ejecución de cada uno de los test's y genera un informe con el resultado extraídos de la ejecución de las pruebas unitarias generadas. Estos resultados son el producto de la evaluación según el criterio que se establece para las condiciones de éxito de cada test. Estas condiciones son en principio condiciones generales, pero este prototipo esta abierto e inclusive se plantea como lineas de desarrollo posibles parametrizaciones de estos criterios de evaluación. De este modo la cobertura sería más robusta y adaptable a la diversidad de casos que puedan presentarse.

Todas las pruebas serán ejecutadas sobre los servidores que tengan desplegado el WS que se indica en la URL que contiene el WSDL. Debe considerarse tener los accesos permitidos para el consumo, es decir, el acceso HTTP al servidor y puerto necesario. Pero entendemos que la variedad de casos también puede abarcar los tipos de restricciones de seguridad, por lo que asumimos que es otra de las lineas de desarrollo de esta investigación inicial, es decir, poder especificar el método de autenticación sobre el WS si este tiene implementado algún mecanismo de seguridad.

A continuación mostramos un modelo que plasma la idea más global y general del flujo de trabajo del Generador:

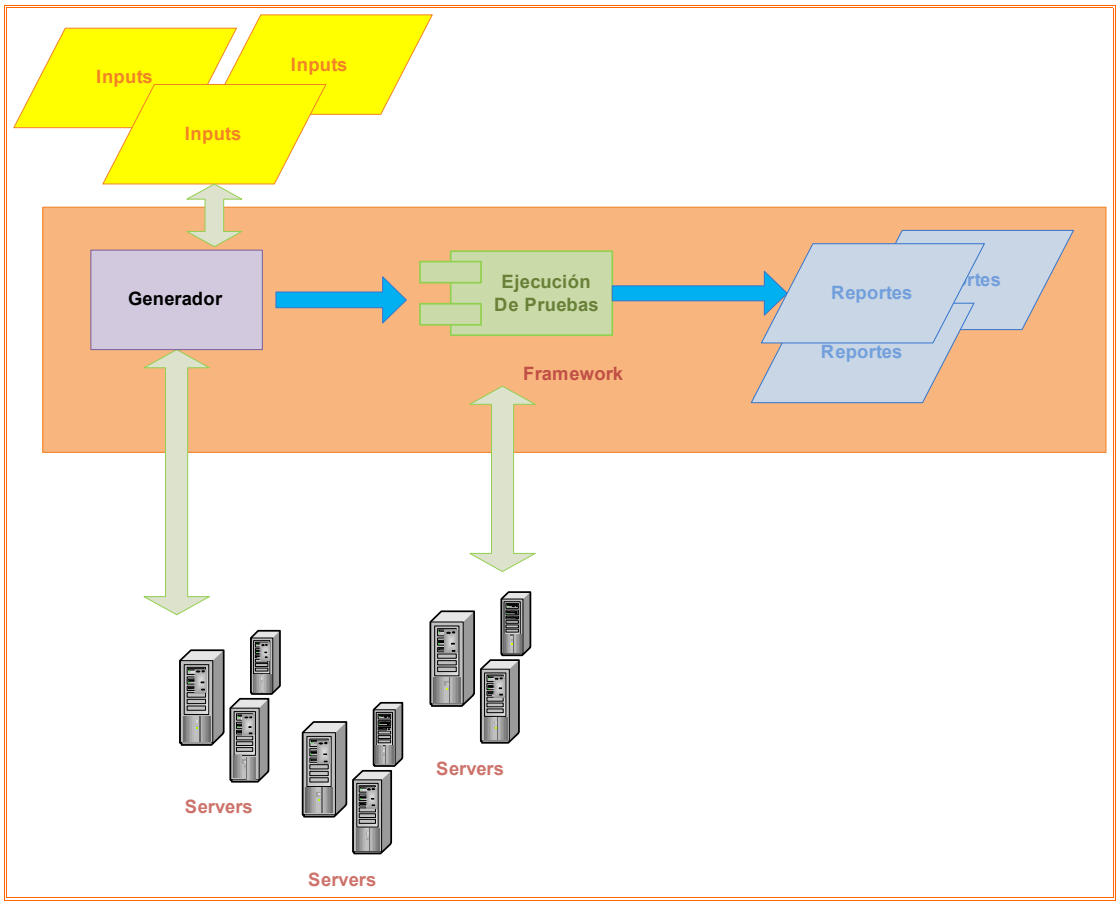


Figura 2: Vista general del modelo que implementa el generador.

Más adelante iremos explotando en varios niveles los detalles que contiene el generador, al igual que los detalles de las posibles lineas de desarrollo de este trabajo de investigación que esta dotado de un prototipo funcional.

5 Prototipo.

5.1 Diagrama general.

A continuación se muestra el diagrama general de las partes que componen el proceso de generación de las pruebas unitarias.

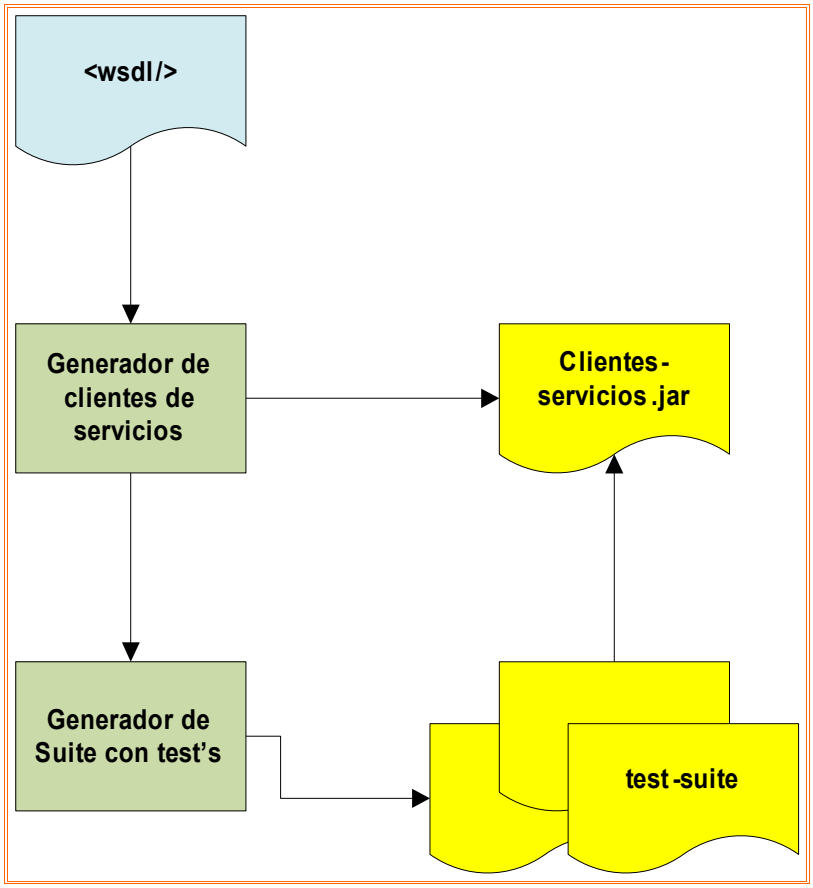


Figura 3: Vista general del proceso de generación de pruebas unitarias.

Como se puede observar, se ilustra como a partir de una entrada que será el descriptor (WSDL), es generado tanto el conector al WS como el conjunto de pruebas que cubren las operaciones que en teoría están expuestas por el WS.

5.2 Flujo de trabajo del generador.

A continuación hacemos explosión del diagrama anterior con el fin de detallar el flujo de trabajo del generador:

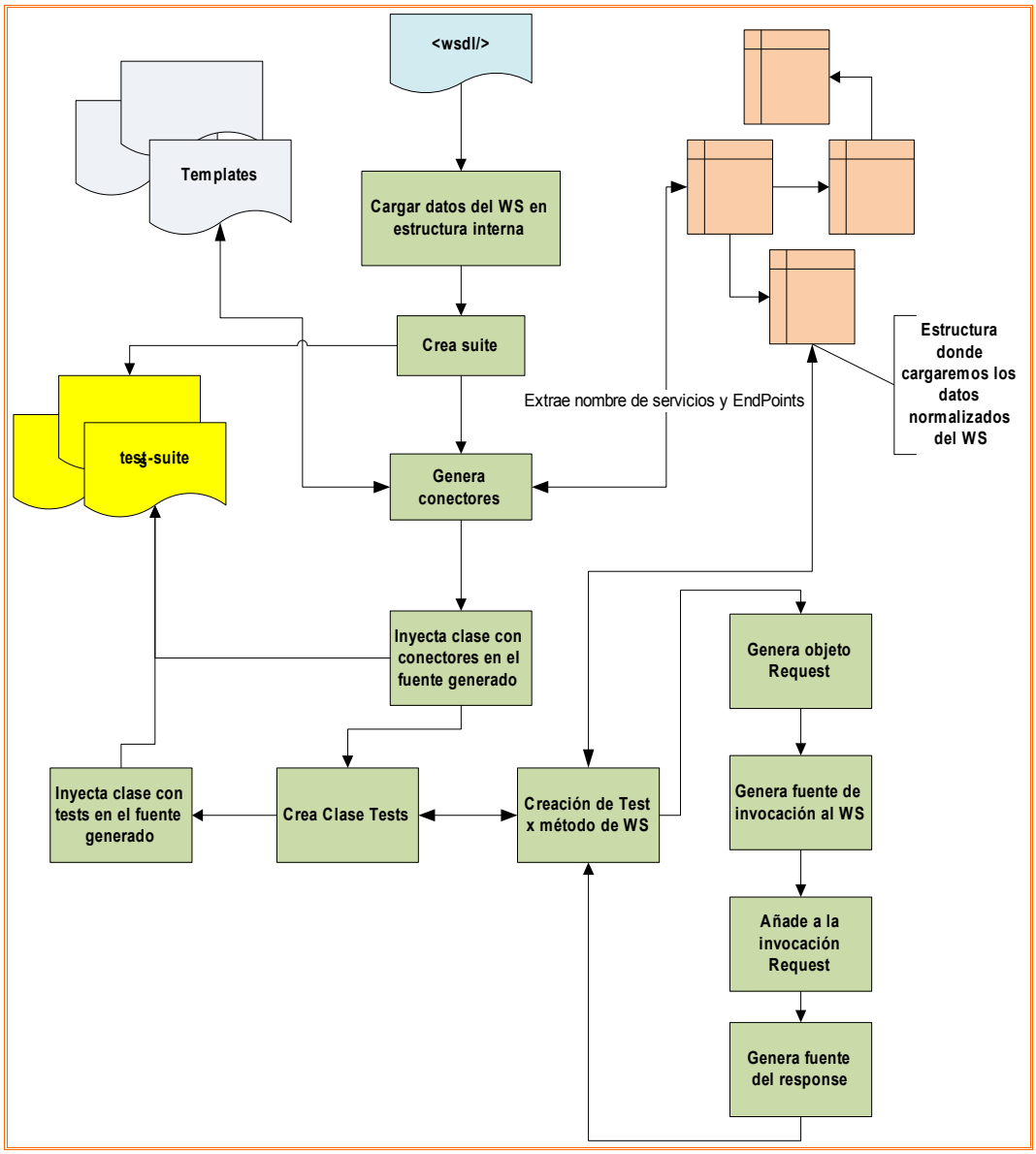


Figura 4: Detalle del flujo de trabajo del generador .

Descripción del flujo de trabajo:

1. En la ejecución del generador se indica como parámetro el nombre del fichero WSDL.

2. Se realiza la carga de los datos que se indican en el WSDL sobre una estructura de clases para posteriormente ser tratados. Los datos extraídos son:

    a) Nombre del servicio.

    b) PortType.

    c) Métodos u operaciones que contiene el WS.

    d) Los paquetes donde se encuentran los bean que representan los tipos complejos.

    e) Los mensajes (request y response) que componen la interacción con los métodos.

    f) Los tipos que componen los mensajes.

3. Se genera a partir de las plantillas el proyecto base. Para posteriormente añadirle todos los recursos necesarios para realizar los consumos al WS.

4. Se generan los conectores correspondientes al WS.

5. Se inyecta la clase en el paquete correspondiente donde la suite que contiene las pruebas unitarias espera ubicar los conectores.

6. Se genera la clase con un test por método que contenga el WS. La clase tendrá como nombre el nombre del WS más "Tests". <WSNameTests.java>

    El generador está preparado para que con pocos cambios pueda manejar varios wsdl's y así generar tantas clases <WSName>Tests.java sea necesario.

    Para ello se realizan los siguientes pasos

    a) Generación de los request, tomando los datos traducidos del WSDL y que los almacenamos en memoria en una estructura simple de leer. Se recorre todos los elementos que conforman la entrada a un método. Siendo capas de construir objetos complejos y haciendo las asignaciones de los valores correspondientes aleatoriamente (test's).

    b) Se construye el código para la invocación del método. Junto con esta invocación, es construido el código que representa la respuesta y la inyección de los parámetros de llamada.

7. Se inyecta la clase con los test en el paquete correspondiente en el proyecto.

8. El código fuente generado será un proyecto Maven denominado test-suite que implementara un conjunto de test de acuerdo al WSDL.

5.3 Modelo de datos (ORM).

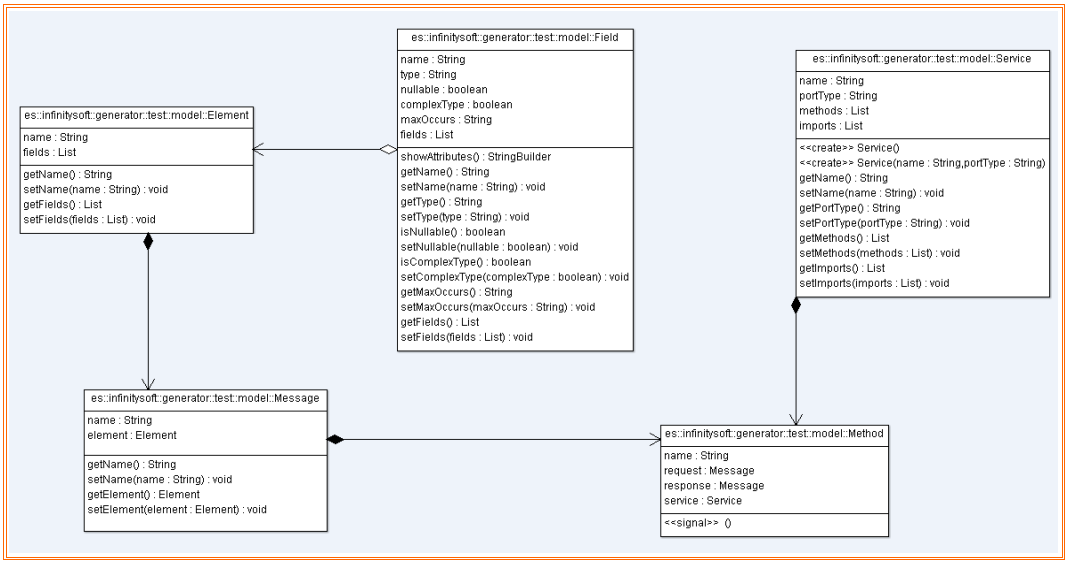


Figura 8: Modelo ORM.

En el proceso de extracción de la información que contiene el WSDL del WS, el generador hace uso del anterior ORM. Para posteriormente usar esos datos en el proceso de generación de los fuentes que implementas las pruebas.

Las entidades que contiene este ORM son:

- **Service.**

  En esta entidad se almacena la información de alto nivel de WS. Como pueden ser: el nombre del WS, port types, lista de métodos, etc.

- **Method.**

  Esta entidad representa una operación o método expuesto por el WS. En ella se relacionan tanto el mensaje de petición como el de respuesta que lo componen.

- **Message.**

  Entidad que representa los mensajes que componen un método y el elemento que compone este mensaje.

- **Element.**

  Representa el elemento que forma el mensaje, los campos y tipos que lo conforman.

- **Field.**

  Esta entidad contiene toda la información relacionada a un campo de un elemento de un mensaje de petición o respuesta. Aquí se almacenan datos como: nombre del campo, tipo, si es requerido, si es complejo, y la lista de campos que lo conforman en caso de ser complejo.

5.4 Diagrama de secuencia.

El siguiente diagrama muestra el flujo de interacción de las entidades que intervienen en el sistema propuesto. Esto nos da una visión general de la solución que se propone y la forma como solventa la necesidad expuesta.

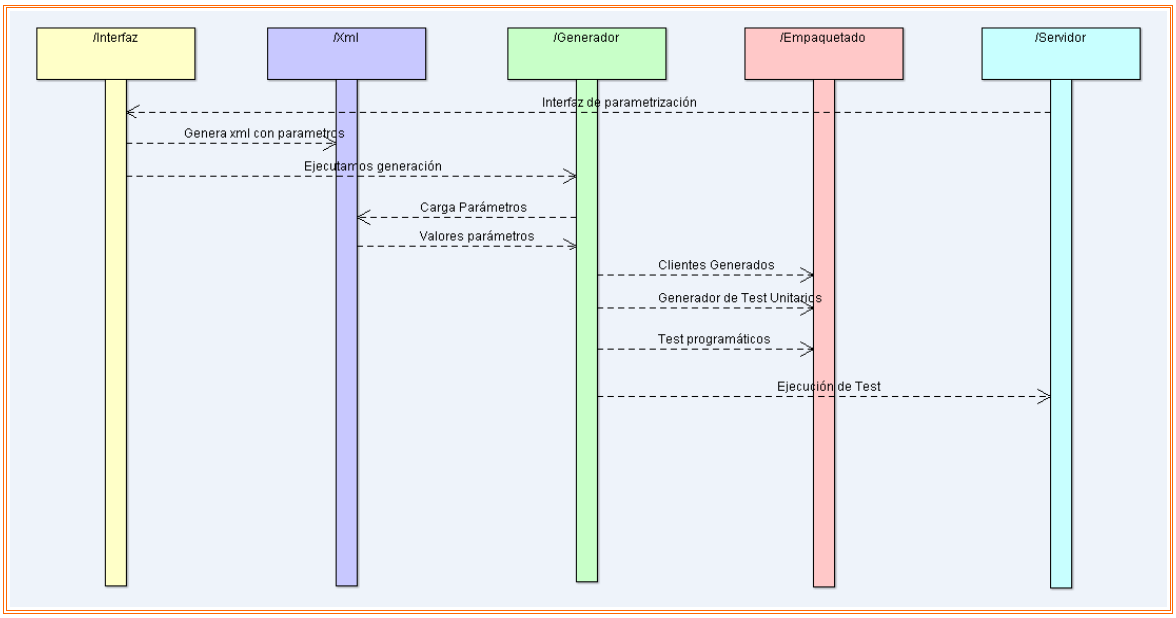


Figura 5: Diagrama de secuencia entre entidades que intervienen en el sistema propuesto.

1. El usuario haciendo uso de la interfaz indica los WS a los cuales se les generarán las pruebas. También indicará el flujo de ejecución entre métodos de un WS o diferentes WS, esto con el fin de modelar diferentes flujos de trabajo que representarán los requisitos funcionales. Aplicación web que permite la introducir los datos que alimentarán al generador.

2. De la parametrización se generará las entradas que usará el Generador. Estas entradas serán generadas en formato XML las cuales el generador interpretara y utilizará para la generación de los test unitarios y test programáticos sobre flujos de trabajo. Formatos generados haciendo uso de Java Dom y un motor de procesos de negocio similar a bpmn para modelar los flujos de trabajo.

3. Una vez establecidos los parámetros desde la misma interfaz se puede ejecutar la generación de las pruebas. Esto provocará que el generador cargue los parámetros y realice la generación del código fuente que contienen las pruebas. La generación de las pruebas se realizará haciendo uso de los frameworks JUNIT.

4. Posteriormente el generador realizará la compilación y empaquetado del fuente generado. La interfaz permitirá visualizar el resultado de la compilación. Para la gestión de construcción de software utilizaremos el framework Maven.

5. Tras la generación del empaquetado serán ejecutados cada una de las pruebas generadas, generando un reporte que muestra los resultados producto de la ejecución de todas las pruebas.

6 Casos de uso.

En este apartado vamos a explicar con mayor detalle cada uno de los casos de uso que lleva a cabo el generador. Intentando dejar claro el flujo de la secuencia del proceso del generador para cada uno de ellos. También se explicará la dependencia de cada caso de uso entre sí, dado que el objetivo principal y final del generador es la producción de un código que sea capaz de ejecutarse y realizar pruebas de otro código que se ejecuta en un entorno que puede ser independiente y diferente.

6.1 Generación del cliente o conector al WS.

Para poder hacer consumo de un WS es necesario disponer de los conectores propios del WS. Estos clientes conectores implementan las clases que representan los tipos complejos que usa el servicio, los beans de cada operación expuesta y los beans para establecer la conexión al WS.

Como ya hemos mencionado en puntos anteriores, el punto de partida para realizar esta operación es el fichero WSDL, el cual es solicitado en la interfaz principal para poder indicar al generador con que WS debe trabajar.

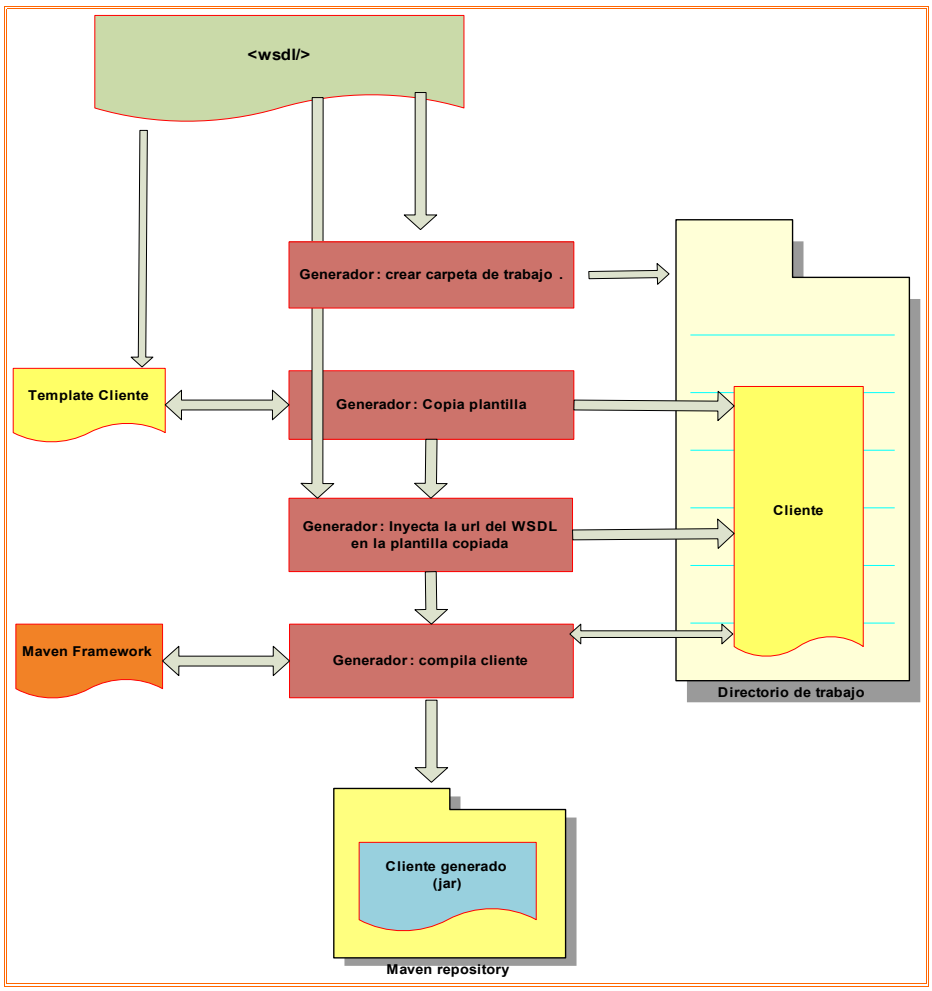


Figura 6: Flujo de generación de clientes o conectores al WS.

Descripción de secuencia:

1. Se especifica la URL del WS con el que se desea trabajar.

2. El generador crea una carpeta de trabajo donde se depositará los ficheros que genere de una forma organizada. Dicha carpeta se encuentra dentro del servidor donde se ejecuta el generador.

3. El generador hace uso de plantillas que se encuentran preconfiguradas y que contienen los recursos básicos para el cliente que se generará.

4. El generador copia la plantilla en la carpeta de trabajo y lo renombra con un nombre estándar (clientes-servicios).

5. El generador inyecta en un fichero pom.xml usado por Maven la URL del WSDL del WS, de modo que permita generación del cliente y la compilación haciendo uso del framework Maven.

6. El generador realiza la compilación del cliente provocando la generación automatica de todas los fuentes necesarios para consumir del WS.

7. Al realizar la compilación se generará una librería (.jar) que contendrá el cliente o conector. Maven, en el momento de la compilación lo instalará en el repositorio Maven que se encuentra en el propio servidor (.m2) donde se realizan las generaciones, compilaciones y ejecuciones de las pruebas. Esta librería será usada posteriormente por el software que se generará conteniendo las pruebas unitarias.

6.2 Generación de la suite.

La generación de la suite que contiene las pruebas unitarias, se divide en varios procesos que están formados por un conjunto de pasos que se describirán a continuación:

6.2.1. Carga de la información sobre el WS.

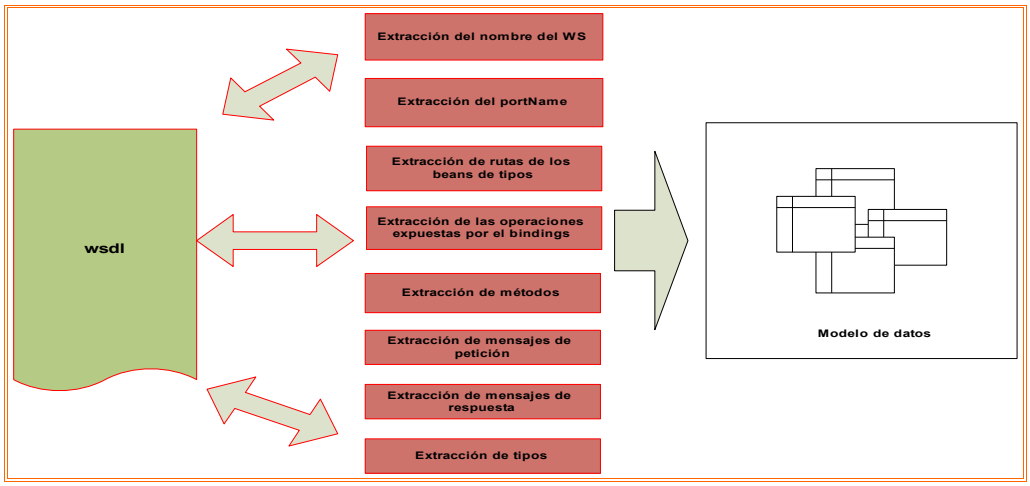


Figura 7: Flujo de extracción y análisis del WSDL.

Haciendo uso de la librería wsdl2j se realiza la lectura del WSDL. Realizando las siguientes operaciones:

1. Extracción del nombre de WS.
2. Extracción del portName del WS.
3. Extracción de la ruta de los Beans que implementan los tipos complejos contenidos en el WS.
4. Lectura de las operaciones expuestas por bindings.
5. Extracción de las operaciones o métodos expuestos.
6. Extracción de los mensajes de petición (request) vinculados a cada operación.
7. Extracción del elemento asociado al mensaje asociado a la operación.
8. Extracción de los campos que conforman el elemento de petición. Para cada campo se identifica:
    a) Nombre
    b) Tipo
    c) Si es requerido o no.
    d) Si se trata de un tipo complejo.
        - En caso de ser un tipo complejo, por medio de recursividad, se repite el proceso de carga de tipos asociados a este tipo complejo.
    e) El número máximo de repeticiones permitida.
9. Extracción de los mensajes de respuesta (response) vinculados a cada operación.
10. Extracción del elemento asociado al mensaje asociado a la operación.
11. Extracción de los campos que conforman el elemento de respuesta. Para cada campo se identifica:
    a) Nombre.
    b) Tipo.
    c) Si es requerido o no.
    d) Si se trata de un tipo complejo.
    e) El número máximo de repeticiones permitida.

6.3 Generación del código fuente que compone la suite con las pruebas.

Una vez es extraída la información del WS descrita en el WSDL y almacenada de forma temporal en el modelo de dato manejado por el generador y descrito en el punto anterior. Se procede a la generación del código fuente que contendrá las pruebas unitarias sobre el WS especificado.

A continuación enumeramos la secuencia de operaciones que lleva a cabo el generador para cumplir con su objetivo principal, la generación de pruebas unitarias:

- El generador crea una carpeta de trabajo donde se depositará los ficheros que genere de una forma organizada y sistemática. Dicha carpeta se encuentra dentro del servidor donde se ejecuta el propio generador.

- El generador lleva a cago la extracción de los datos descriptores del WS haciendo uso del WSDL. Esta información es almacenada de forma temporal en un modelo de datos definido para el uso por parte del generador.

- El generador hace uso de una serie de plantillas para la generación del proyecto base que finalmente contendrá la suite con las pruebas unitarias.

- Haciendo uso del modelo de datos se extrae la información de integración con el servicios web indicados. En nuestro caso, el prototipo solo permite la indicación de un WS.

  Para cada WS se crea un fichero que contendrá un método que implementa la prueba unitaria que corresponderá con un método que exponga el WS. Esta fichero se ubicará en un paquete destinado para este fin.

- Dado un WS, se recuperan todos los métodos que este contiene. Y se genera e inyecta en el fichero que se ha creado en el paso anterior, el método que implementa la prueba unitaria, extrayendo el nombre del método del ORM previamente cargado.

  En este paso para poder realizar la prueba que consume un método especifico debe también generarse:

  - Mensaje de petición (request): Para esto el generador es capaz de leer como se compone el mensaje de petición y traducirlo al lenguaje java, inyectando a su vez los valores generados aleatoriamente según el tipo que corresponda.

  - Mensaje de respuesta (response): De igual modo se determina como se compone la respuesta y se toma en cuenta para la recepción del mensaje de respuesta tras el consumo de un método específico.

  - Condición de verificación (assert): inmediatamente de la petición se inyecta esta condición en función del tipo de respuesta (response) generado.

- Tras la generación del código fuente que compone la suite, este queda disponible para las siguientes operaciones:

- Descarga del código fuente comprimido.
- Compilación en servidor de la suite generada con las pruebas unitarias.
- Visualización de la salida de la consola tras la ejecución de la compilación.
- Ejecución de las pruebas unitarias generadas.
- Extracción de resultados producidos en la ejecución de las pruebas unitarias generadas.

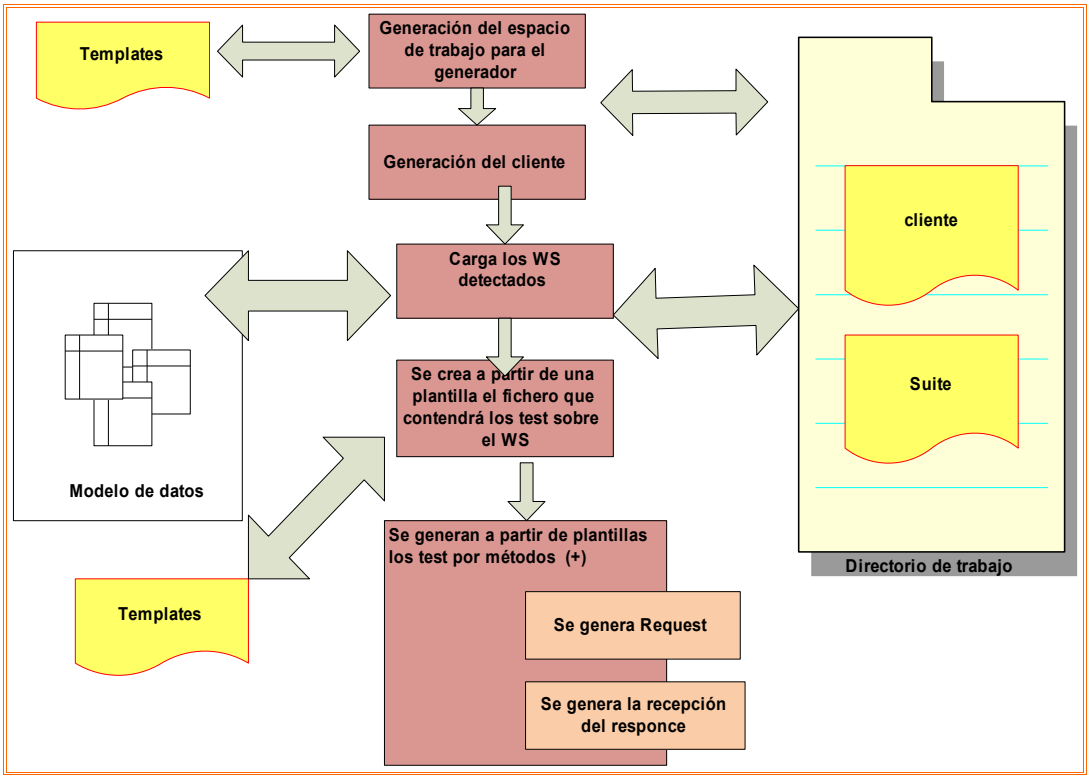


Figura 9: Flujo de de generación de la suite que contiene las pruebas unitarias.

6.4 Generación del mensaje de petición.

Para el consumo de los métodos expuestos por los servicios, en ocasiones hay que pasar en la petición parámetros que son requeridos. Por lo que una de las acciones más importantes y complejas que lleva a cabo el generador, es la capacidad de determinar los campos que se requieren y el tipo de dato que corresponda. Con esta información el generador puede crear los objetos que se necesiten y asignarle un valor aleatorio según el tipo de dato. Es decir, el generador es capaz de realizar los mockup's que simulan una petición cualquiera. El siguiente flujo ilustra las acciones que se llevan a cabo para la construcción del código que ejecutará las peticiones.

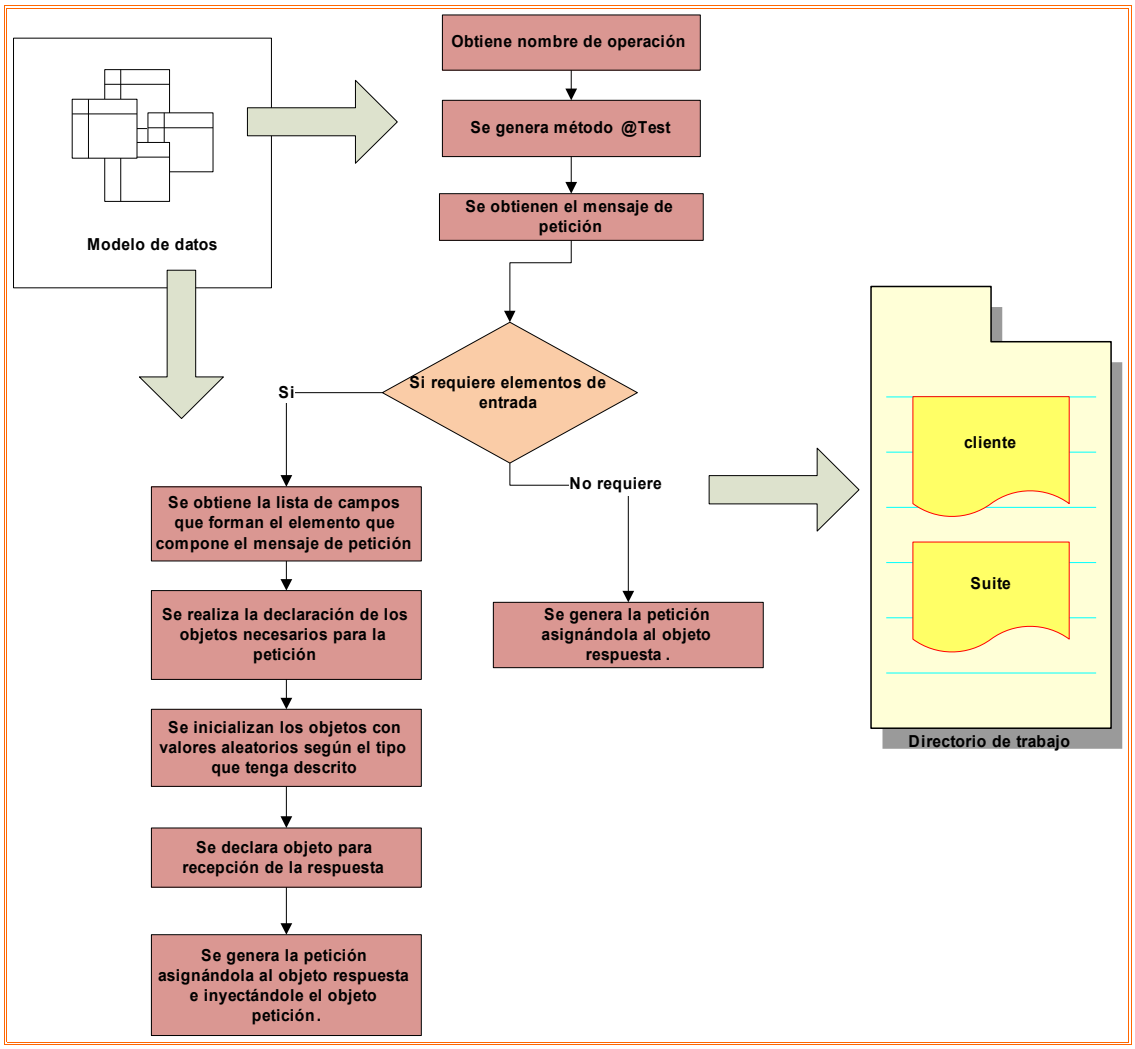


Figura 10: Flujo de construcción del código que ejecutará las peticiones.

A continuación describiremos el flujo:

1. En el momento de la generación del software que contendrá las pruebas unitarias de consumo de los servicios, es cargada la información del WSDL en el ORM para posteriormente ir leyendo esta información y así obtener los datos necesarios en la generación de cada parte necesaria. En este caso las peticiones.

2. El generador lee del modelo de datos el listado de métodos expuestos.

3. Por cada método el generador:

   a) De forma dinámica se construye un método (@Test) donde se implementará la prueba unitaria que contiene la petición a una operación específica expuesta por el WS.

   b) Se lee del ORM las características que debe tener el mensaje de petición, es decir , los campos que la conforman y que tipo de dato deben ser(Date, String, int, long, ComplexType, etc)

c) En caso de no requerir parámetros se procede a la construcción de la petición directamente. De lo contrario:

- Se crean los objetos que representarán los parámetros. Los cuales requiere la operación que se consumirá en esa prueba.
- Se inicializan con valores aleatorios estos objetos declarados. Según el tipo de dato que corresponda para el parámetro.
- Se crea el objeto que recibe la respuesta producto de la ejecución de la petición.
- Se genera la petición inyectándole los parámetros declarados e inicializados.
- Se añade el método generado a la clase que contiene los test's del servicio procesado.

6.4.1. Ejemplo de generación de una prueba unitaria.

```
<xs:complexType name="ImportControll">
   <xs:sequence>
   <xs:element minOccurs="0" name="contactKronos" type="xs:boolean"/>
   <xs:element minOccurs="0" name="dateImport" nillable="true" type="xs:dateTime"/>
   <xs:element minOccurs="0" name="dateImportEnd" nillable="true" type="xs:dateTime"/>
   <xs:element minOccurs="0" name="description" nillable="true" type="xs:string"/>
   <xs:element minOccurs="0" name="fileName" nillable="true" type="xs:string"/>
   <xs:element minOccurs="0" name="id" type="xs:long"/>
   <xs:element minOccurs="0" name="ignored" type="xs:long"/>
   <xs:element minOccurs="0" name="importName" nillable="true" type="xs:string"/>
   <xs:element minOccurs="0" name="mimeType" nillable="true" type="xs:string"/>
   <xs:element minOccurs="0" name="processed" type="xs:long"/>
   <xs:element minOccurs="0" name="status" type="xs:short"/>
   </xs:sequence>
</xs:complexType>
...
<xs:element name="updImportControllResponse">
  <xs:complexType>
   <xs:sequence> <xs:element minOccurs="0" name="return" type="xs:int"/> </xs:sequence>
 </xs:complexType>
</xs:element>
...
<xs:element name="updImportControll">
   <xs:complexType>
   <xs:sequence>
   <xs:element minOccurs="0" name="importControll" nillable="true" type="ax22:ImportControll"/>
   </xs:sequence>
   </xs:complexType>
</xs:element>
...
<wsdl:message name="updImportControllRequest">
<wsdl:part name="parameters" element="ns:updImportControll"/>
</wsdl:message>
<wsdl:message name="updImportControllResponse">
<wsdl:part name="parameters" element="ns:updImportControllResponse"/>
</wsdl:message>
...
<wsdl:operation name="updImportControll">
   <wsdl:input message="axis2:updImportControllRequest" wsaw:Action="urn:updImportControll"/>
   <wsdl:output message="axis2:updImportControllResponse" wsaw:Action="urn:updImportControllResponse"/>
</wsdl:operation>
```

Figura 11: Fragmento del WSDL que describe la operación updImportControll.

```
@Test
public void updImportControll() {
     try {

       ImportControll importControll = new ImportControll();
       importControll.setContactKronos((BaseUtils.getRandomBoolean()));
       importControll.setDateImport((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
       .createImportControllDateImport(BaseUtils.getRandomDate()));
       importControll.setDateImportEnd((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
                 .createImportControllDateImportEnd(BaseUtils.getRandomDate()));
       importControll.setDescription((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
                 .createImportControllDescription(BaseUtils.getRandomString()));
       importControll.setFileName((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
                 .createImportControllFileName(BaseUtils.getRandomString()));
       importControll.setId((BaseUtils.getRandomLong()));
       importControll.setIgnored((BaseUtils.getRandomLong()));
       importControll.setImportName((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
                          .createImportControllImportName(BaseUtils.getRandomString()));
       importControll.setMimeType((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
                 .createImportControllMimeType(BaseUtils.getRandomString()));
       importControll.setProcessed((BaseUtils.getRandomLong()));
       importControll.setStatus((new Short((short) 0)));

       Integer result = cliente.getContactService().updImportControll(importControll);

       System.out.println("Resultado: " + result.toString());
       Assert.assertTrue(result > 0);
    } catch (Exception e) {
       e.printStackTrace();
    }
  }
}
```

Figura 12: Prueba unitaria generada, para el consumo del método updImportControll expuesto por el WS.

6.5 Ejecución de la Test Suite generada.

Esta operación se habilita una vez es generado el cliente conector para el consumo del WS y también cuando esté generada la Test Suite con las pruebas unitarias. La secuencia asociada a esta operación es la siguiente:

- Obtención de la ruta de la carpeta de trabajo asignada a la generación realizada.
- Haciendo uso del framework Maven, se ejecuta en el servidor la compilación y ejecución del código generado que contiene el banco de pruebas. Este paso ejecutará previa compilación cada uno de los métodos hayan sido generado / implementado dentro de la suite por el generador.
- De nuevo por medio de Maven realizamos la ejecución de la generación de los reportes, con los resultados de la ejecución de las pruebas unitaria. Esta generación se hace por medio del FrameWork Site, que ya se encuentra previamente configurado en la plantilla de la Test Suite que usa el generador para la implementación del código final.
- La salida de consola producto de la compilación, ejecución y generación de los reportes es volcada en un fichero que sera copiado en una carpeta del servidor que tiene acceso permitido para su lectura y visualización.
- Se copian también los reportes en una carpeta de acceso de lectura permitido en el propio servidor.
- Se habilita en la interfaz acceso al usuario para poder visualizar los log's e informes generados.

7 Salidas del prototipo en las generaciones realizadas.

A continuación describiremos como el generador implementa técnicamente las operaciones antes descritas:

7.1 Generación del cliente.

- El usuario indica por medio de la interfaz la URL del WSDL del WS.

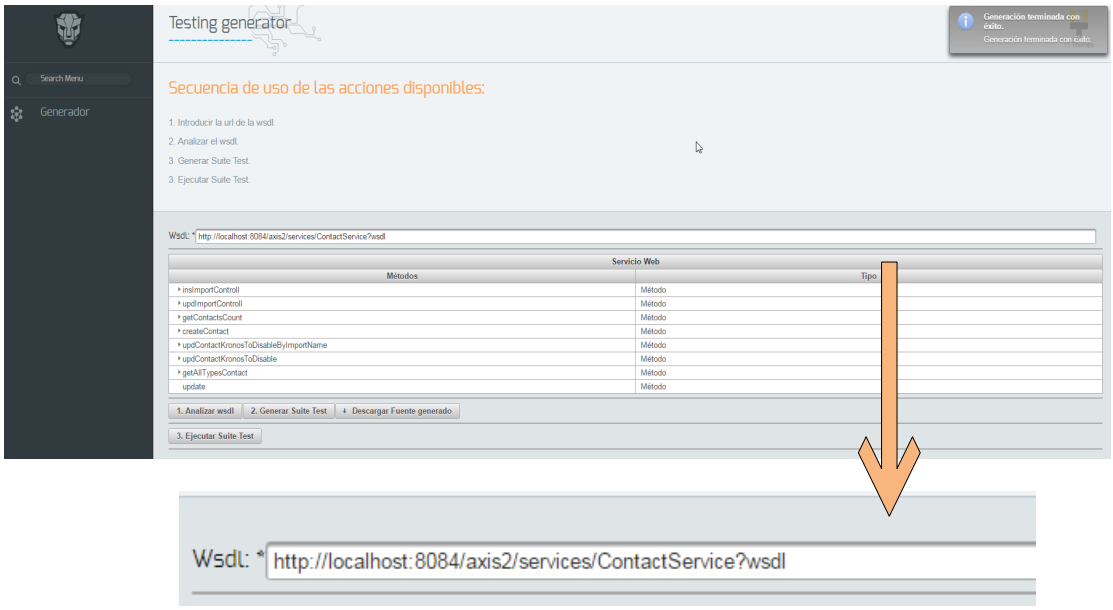


Figura 13: Pantalla donde se especifica la URL al WS.

- El generador toma la plantilla para la generación del cliente.
- El generador debe inyectar la URL indicada (se marca en naranja) en la siguiente plantilla que posteriormente usará el framework Maven.

Ejemplo de inyección de URL del WSDL en el fichero de pom.xml de Maven:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>es.infinitysoft.uned.servicios.clientes</groupId>
    <artifactId>clientes-servicios</artifactId>
    <version>1.0</version>
    <packaging>bundle</packaging>
    <name>Clientes Servicios</name>
    <properties>
        <main.basedir>${project.parent.parent.basedir}</main.basedir>
        <maven-bundle-plugin>2.3.7</maven-bundle-plugin>
        <maven-clean-plugin>2.5</maven-clean-plugin>
        <cxf.codegen>2.4.3</cxf.codegen>
        <java.version>1.6</java.version>
        <pre>buger</pre>
    </properties>
    <build>
        <defaultGoal>clean install</defaultGoal>
        <pluginManagement>
            ...
        </pluginManagement>
        <plugins>
            <plugin>
                <artifactId>maven-clean-plugin</artifactId>
                <version>${maven-clean-plugin}</version>
                <configuration>
                    <filesets>
                        <fileset>
                            <directory>${basedir}/src/main/java</directory>
                            <includes>
                                <include>**/*</include>
                            </includes>
                            <followSymlinks>false</followSymlinks>
                        </fileset>
                    </filesets>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.apache.cxf</groupId>
                <artifactId>cxf-codegen-plugin</artifactId>
                <version>${cxf.codegen}</version>
                <executions>
                    <execution>
                        <id>generate-sources</id>
                        <phase>generate-sources</phase>
                        <configuration>
                            <sourceRoot>${basedir}/src/main/java</sourceRoot>
                         <wsdlOption>
                            <wsdl>http://localhost:8084/axis2/services/ContactService?wsdl</wsdl>
                         </wsdlOptions>
                            </configuration>
                            <goals>
                                <goal>wsdl2java</goal>
                            </goals>
                        </execution>
                </executions>
            </plugin>
           ...
        </plugins>
    </build>
</project>
```

7.2 Generación de la Test Suite.

La salida de la generación corresponderá a una carpeta que contendrá un proyecto Maven preconfigurado para abrirse en cualquier IDE (por ejemplo Eclipse) y que contendrá los conectores y los Test unitarios basados en Junit.

Partiendo del WSDL siguiente se realiza la generación que iremos describiendo en cada uno de los casos:

Figura 14: Imagen del WSDL visualizado en un navegador.

Generará una carpeta denominada test-suite con el siguiente contenido:

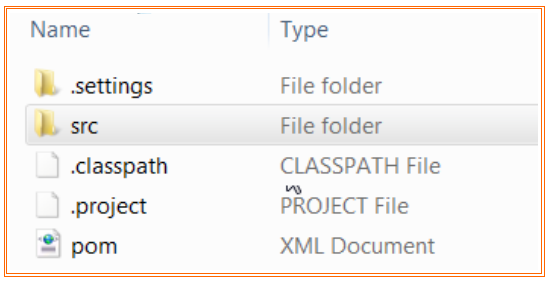


Figura 14: Ejemplo del proyecto generado.

El código fuente se generará en el siguiente paquete del proyecto test-suite: <UUID>\test-suite\src\main\java\es\infinitysoft\uned\test.

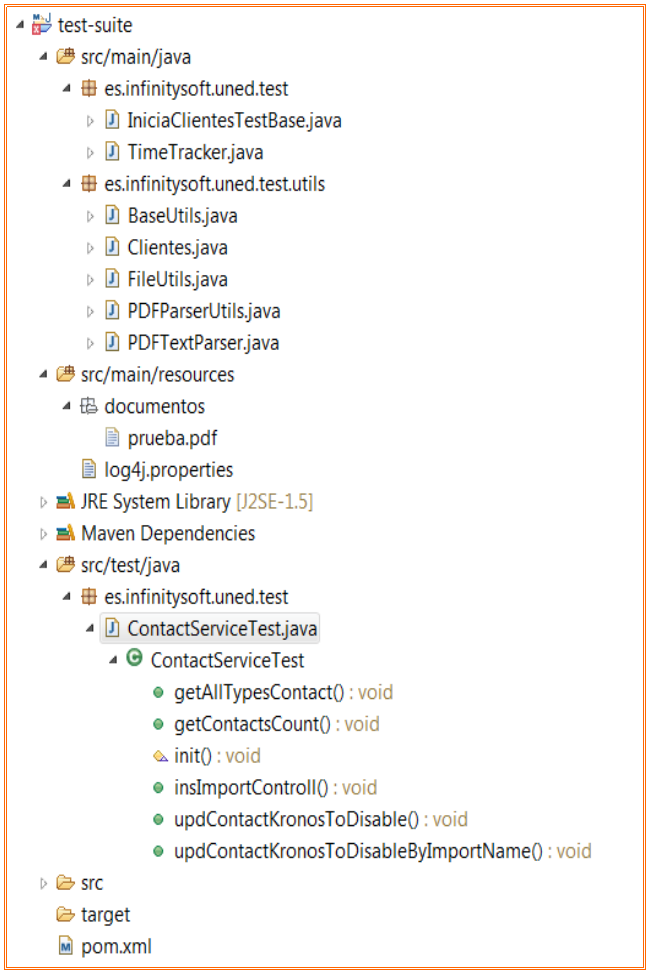


Figura 15: Aspecto del proyecto resultante de la generación y la estructura de paquetes que contiene.

El generador luego de tratar el WSDL que describe un WS llamado ContactService, genera la clase ContactServiceTest que implementa las pruebas unitarias que hará consumo de los métodos del WS. Mostramos la clase acotando solo para un método:

```
package es.infinitysoft.uned.test;

import java.net.URL;
import java.util.List;

import org.junit.Assert;
import org.junit.Test;

import es.infinitysoft.services.*;
import es.infinitysoft.uned.test.utils.BaseUtils;
import es.infinitysoft.uned.test.utils.Clientes;
import es.infinitysoft.uned.test.utils.service.*;
import es.infinitysoft.xsd.GetContactsCountResponse;
import es.infinitysoft.uned.test.IniciaClientesTestBase;
import es.infinitysoft.ws.entity.xsd.*;
import es.infinitysoft.xsd.*;

public class ContactServiceTests extends IniciaClientesTestBase {

	@Override
	protected void init() {
		// TODO Auto-generated method stub
	}

	/** MARK_START_CLIENTS_CODE **/
	@Test
	public void updImportControll() {
	   try {
		ImportControll importControll = new ImportControll();
		importControll.setContactKronos((BaseUtils.getRandomBoolean()));
		importControll.setDateImport((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
		.createImportControllDateImport(BaseUtils.getRandomDate()));
		importControll.setDateImportEnd((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
				.createImportControllDateImportEnd(BaseUtils.getRandomDate()));
		importControll.setDescription((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
				.createImportControllDescription(BaseUtils.getRandomString()));
		importControll.setFileName((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
				.createImportControllFileName(BaseUtils.getRandomString()));
		importControll.setId((BaseUtils.getRandomLong()));
		importControll.setIgnored((BaseUtils.getRandomLong()));
		importControll.setImportName((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
                                                .createImportControllImportName(BaseUtils.getRandomString()));
		importControll.setMimeType((new es.infinitysoft.ws.entity.xsd.ObjectFactory())
				.createImportControllMimeType(BaseUtils.getRandomString()));
		importControll.setProcessed((BaseUtils.getRandomLong()));
		importControll.setStatus((new Short((short) 0)));
		Integer result = cliente.getContactService().updImportControll(importControll);
		System.out.println("Resultado: " + result.toString());
		Assert.assertTrue(result > 0);
	    } catch (Exception e) {
		e.printStackTrace();
	    }
	}
}
```

8 Restricciones.

El prototipo planteado dado su propósito académico tiene una serie de limitaciones o restricciones que enumeraremos a continuación, y que más adelante propondremos lineas de desarrollo e investigación para solventar y ampliar la linea base de desarrollo del generador:

Las restricciones o limitaciones son las siguientes:

- El generador considera solo servicios basados en la arquitectura SOAP.
- Se trata de un prototipo funcional desarrollado y probado sobre servicios desarrollados con el framework Axis2.
- La interfaz no posee implementada un mecanismo para parametrizar los valores de petición sobre los métodos de un WS.
- La interfaz no posee implementada un mecanismo para diseñar un flujo de peticiones entre métodos de un WS.
- Se propone como linea de desarrollo la implementación de mecanismos para especificar formatos de campos específicos.
- Se propone como linea de desarrollo la implementación de mecanismos para especificar criterios de éxito por test.
- Solo está preparado para indicar un WSDL por generación. A pesar de que el generador está preparado para que con pocos cambios pueda manejar varios WSDL's y así generar tantas clases <WSName>Tests.java sean necesarias.

9 Tecnologías usadas.

9.1 Primefaces.

- La interfaz es bajo ambiente web y esta desarrollada bajo el framework Primefaces.

9.2 Apache Maven.

- Uso de Maven para la construcción del generador y de las suite's generadas.

9.3 Junit Framework.

- Es usado el framework Junit para realizar la implementación de las pruebas unitarias.

9.4 Apache Maven surfire report.

- Maven surfire report y Maven site plugin para la generación de los reportes de ejecución de pruebas.

9.5 Simple Logging Facade for Java (SLF4J).

- Slf4j para la generación de logs.

9.6 Apache Commons FileUtils.

- Para uso y manipulación de las plantillas.

9.7 Características de Java.

- Versión 1.7.
- java.util.zip para la generación de los ficheros zip con el código generado.
- java.lang.Runtime para la compilación y ejecución "online"

10 Código Fuente & GitHub.

El código fuente perteneciente al prototipo implementado se encuentra en la herramienta de control de versiones GitHub, cuya ubicación es la siguiente:

https://github.com/joseaperezr/TestGenerator

11 Lineas de investigación y desarrollo.

El actual trabajo pone en evidencia un problema y propone una solución, al igual que se propone un prototipo que ejemplariza el modelo que es propuesto. Evidentemente este trabajo tiene fines académicos, por lo que no se desarrolló enteramente bajo una perspectiva de producto. Es por esto que se implementó una linea base que contiene el núcleo de la propuesta, pero sin haber desarrollado casos que cubran aspectos muy específicos que puedan tener un WS.

Independientemente de que el desarrollo tuviera un mayor o menor espectro de alcance sobre lo que se propone, podemos identificar sobre que aspectos se puede ampliar la funcionalidad del prototipo, que aspectos se deben abordarse para un análisis más profundo y que otros son propicios para abrir lineas de investigación.

11.1 Lineas de desarrollo.

11.1.1. Permitir trabajar con generaciones basadas en varios WS's.

Actualmente el prototipo desarrollado realiza la generación de las pruebas en base a un solo WS. Pero teniendo en cuenta que en multitud de casos los servicios web interaccionan con otros, y considerando que como idea inicial se contempló que adicionalmente a las pruebas unitarias sobre las operaciones expuestas por un WS, es necesario realizar pruebas funcionales donde el ámbito de la prueba pueda abarcar la ejecución de una serie de operaciones en una secuencia determinada y con unas salidas específicas. Estas operaciones pueden pertenecer a distintos WS's, siendo necesario que el generador sea capas de adaptarse y ampliarse a estás situaciones. Y para esto debería cubrir las siguientes características:

- Poder indicarse uno o varios WS's distintos.
- Establecer valores determinados para inyectar a las peticiones o métodos específicos. (punto 10.1.2)
- Poder establecer casos de éxito para los resultados de las pruebas unitarias (punto 10.1.3).
- Poder indicar un flujo de llamadas entre los métodos que expongan los WS indicados (punto 10.1.4).

11.1.2. Permitir parametrización de valores para el consumo de métodos expuestos por un WS.

El prototipo desarrollado realiza una generación automática de las peticiones a realizar y los valores que estás contengan. Y para esta generación del valor a inyectar en la petición, el generador toma como referencia la descripción contenida en el WSDL sobre el tipo de dato que corresponda a los campos que conforman un petición determinada. Y en base a este tipo de campo genera un valor mockup aleatorio.

Evidentemente pueden existir múltiples casos donde sea necesario acotar los posibles valores, ya que un determinado método puede tener de forma encapsulada el uso de los valores de un campo.

Por ejemplo, una operación que se encargue de recibir un campo que represente un prefijo telefónico, y que como respuesta retorne la provincia a la que corresponda dicho prefijo. Ese dato de entrada puede ser definido de un tipo de dato String, pero los posibles valores deberán corresponder a un entero de dos o tres cifras, por lo que no sería coherente inyectar un valor String aleatorio y esperar un respuesta correcta.

Por situaciones de este tipo el generador debería ser capaz de establecer por medio de la interfaz, rangos o conjuntos de valores específicos para peticiones determinadas.

11.1.3. Parametrización de casos de éxitos.

Tal como ocurre con las peticiones, puede suceder con las respuestas de una operación de un WS. Podemos recibir una respuesta de un determinado tipo de dato, pero quizás nos interesa calificar como éxito de la prueba un rango o conjunto de datos a determinar previamente a la generación de las pruebas.

Adicionalmente, la condición de éxito es interesante que sea un valor que pueda establecerse específicamente si se desea. Pudiendo indicar condiciones lógicas predeterminadas o especificas.

Posibles casos que condicionen los casos de éxito (assert's):

- respuesta != null
- respuesta > 0
- respuesta > valor parametrizado. Ej: repuesta == 1

11.1.4. Diseñador de flujo de consumos entre distintos métodos expuestos por un WS.

Básicamente existen dos formas de probar un WS, las cuales no son excluyentes entre sí, una de ellas es consumiendo cada método que exponga el WS. Esta representa una prueba de funcionamiento del método especifico, que nos permite saber su estado de operatividad y la idoneidad de la respuesta. El prototipo desarrollado genera este tipo de pruebas.

El otro tipo de pruebas es la funcional, es decir, la prueba que contempla un flujo de trabajo asociado a una funcionalidad que conglomera el consumo de varias operaciones que pueden pertenecer a uno o varios WS. Para esta finalidad debe dotarse a la interfaz del generador de un mecanismo que permita el diseño de esa secuencia de llamadas entre operaciones que representarán un flujo de trabajo determinado. Esto permite comprobar que una plataforma o sistema cuya arquitectura sea orientada a servicios, tenga un requisito correctamente implementado según el flujo de trabajo que se pretende seguir.

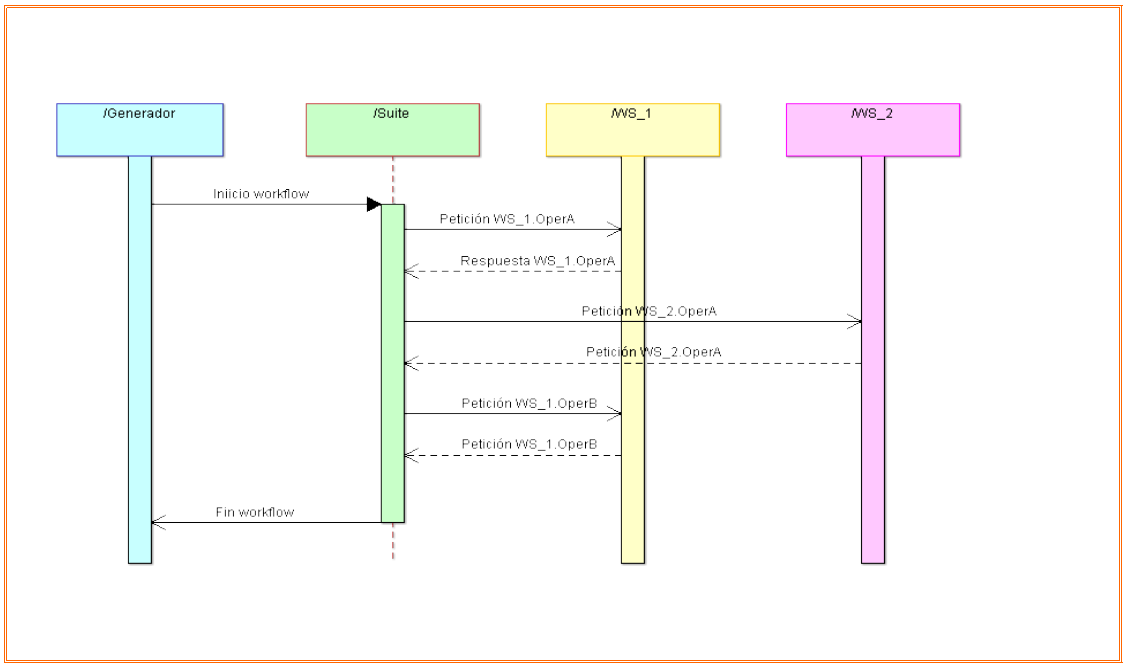


Figura 16: Representación gráfica de una prueba sobre un flujo de trabajo.

11.2 Lineas de investigación.

11.2.1. Manejo de métodos de autenticación.

Una de las linea de desarrollo, ampliación e investigación de la solución propuesta, debe ser el manejo de métodos de autenticación. El prototipo desarrollado se hizo usando WS sin autenticación, pero debe considerarse que los WS pueden requerir autenticaciones y existen muchos mecanismos para asegurar el acceso a un WS. Como pueden ser:

- HTTP Autentication
- WS-Security Tokens
- WS-Signature
- SAML

11.3 Microservicios.

Tras años donde las empresas, departamentos de informática, desarrolladores, etc han desarrollado aplicaciones que no dejan de crecer, exponiendo nuevos servicios, convirtiéndose en aplicaciones densas que pueden convertirse en cuellos de botellas que requieran cada vez más recursos de Hardware. Para evitar tener una enorme piedra "monolítica" se empieza a escuchar hablar sobre microservicios. Estos en concepto exponen pequeños comportamientos y se comunican de forma distribuida entre servicios, sin importar que no estén en la misma aplicación.

La ventajas de una arquitectura basada en microservicios:

- Combinar los servicios según sea el interes. Incluso, reutilizarlos para distintos usos dentro de la empresa. Como piezas de puzzle podemos crear aplicaciones que usen una misma lógica de negocio sin estar cautiva en una aplicación monolítica.

- Escalar a nivel de microservicios. Cada uno de ellos expone una funcionalidad, así podemos distribuirlo según nuestras necesidades pensando en la demanda y el balanceo de carga de cada aplicación. Esto contrapone la idea poco eficiente de replicar aplicaciones enteras cargadas de funcionalidad por unas pocas que representen la mayor carga.

- Simplificamos el mantenimiento. Cumplen a la perfección el principio SOLID. Podemos desechar componentes que no usamos o desplegarlos en otros servidor para engancharlos en nuestra aplicación. Se convierte en una tarea simple tirar algo que no usemos a la basura que mantenerlo como código legacy dentro de una aplicación monolítica.

- Su fallo no arrastra a todo el sistema. Si un componente no funciona correctamente no afecta al resto. Se puede aislar y manejar el error, desplegando por separado.

- Un despliegue puede ser progresivo sin detener todo un entorno enteramente. Implicando esto que ya no se aloja las aplicaciones en un solo contenedor, sino que podrán existir N contenedores.

Esta tendencia debe tenerse en cuenta ya que será una de las inclinaciones en las áreas de desarrollo en arquitectura orientadas a servicios. Y donde cobran más sentido herramientas ágiles para la generación rápida y eficaz de pruebas unitarias, funcionales y de rendimiento.

11.4 Servicios REST.

El actual trabajo esta orientado a la arquitectura SOAP. Estando el generador adaptado a este tipo de arquitectura, sería de interés que el generador cubra las necesidades y funcionalidades descritas y detectadas para la arquitectura Restfull. Esta arquitectura se basa conceptualmente en el mismo propósito que SOAP básico "envió de mensajes entre aplicaciones". Pero de una manera más simple, por medio del protocolo HTTP y menos formal que la arquitectura robusta de SOAP.

12 Conclusiones.

Hemos analizado un problema muy común en distintos tipos de organizaciones, la existencia de innumerables sistemas desarrollados sin prácticamente cobertura de pruebas. Organizaciones privadas o del ámbito público pierden horas haciendo comprobaciones manualmente y sin ningún rigor sistemático sobre sistemas existentes, sin tiempo ni recursos para abordar proyectos que desarrollen pruebas de estos sistemas.

Hemos analizado tres opciones muy conocidas que tratan de resolver estos problemas. Cada una de ellas con un enfoque similar pero con distinto grado de especialización. Destacamos de las tres opciones a la herramienta SOAPUI referente en el mercado para solventar este tipo de problemas, tratando de hacer de una forma sencilla para el usuario, y aportando funcionalidades muy útiles y avanzadas.

No obstante, hemos detectado dentro de un contexto muy limitado algunas posibilidades de aportar nuevos automatismos dentro de la generación de código de cobertura de pruebas. Considerando interesante poder abarcar cada uno de los criterios anteriormente señalados, con el fin de poder desarrollar un generador como el que hemos diseñado conceptualmente en este trabajo, que logre poder cubrir un gran porcentaje de pruebas unitarias y funcionales para un sistema bajo arquitectura SOAP existente.

Dado que el ámbito de este generador es bastante amplio, solo fue posible el desarrollo del prototipo funcional de un software que realiza la generación de test de cobertura sobre los métodos de un WS. Dejando fuera del alcance del prototipo funcionalidades más complejas como la generación de test bajo un flujo de trabajo definido. Pero identificando y proponiendo lineas de desarrollo e investigación orientadas al desarrollo y formalización del generador.

13 Bibliografía.

- Jack Herrington. Code Generation in Action. Manning Publications. 2003.
- Apache Maven Project https://maven.apache.org/
- Apache cxf http://cxf.apache.org/docs/wsdl-to-java.html
- W3Schools http://www.w3schools.com/xml/xml_wsdl.asp
- World Wide Web Consortium https://www.w3.org/TR/wsdl
- Apache Maven Site Plugin https://maven.apache.org/plugins/maven-site-plugin/
- Apache Axis2 http://axis.apache.org/axis2/java/core/
- GitHub https://github.com/
- https://docs.oracle.com/javase/7/docs/api/java/util/zip/package-summary.html
- Apache Commons https://commons.apache.org/proper/commons-io/
- Wikipedia https://es.wikipedia.org/wiki/
- http://www.tutorialspoint.com/restful/restful_introduction.htm
- http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/211
- https://es.wikipedia.org/wiki/Security_Assertion_Markup_Language

14 Listados de Siglas.

- **WS**: Servicio web - tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones.
- **WSDL**: Web Services Description Language - un formato XML que se utiliza para describir WS.
- **URL**: Uniform Resource Locator - es una referencia a un recurso web que especifica su ubicación en una red informática.
- **SOAP**: (S*imple Object Access Protocol*) es un protocolo estándar que define cómo dos objetos en diferentes procesos pueden comunicarse por medio de intercambio de datos XML.
- **BBDD**: Bases de datos.
- **JSP**: Java Server Page - Tecnología para crear páginas web dinámicas basadas en HTML, XML, entre otros tipos de documento.
- **ASP**: Active Server Page - tecnología de Microsoft del tipo "lado del servidor" para páginas web generadas dinamicamente.
- **XML**: e***X****tensible* ***M****arkup* ***L****anguage* - es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos en forma legible.
- **ORM**: Object-Relational mapping - técnica de programación para convertir datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos.
- **IDE**: integrated development environment - es una aplicación de software que proporciona servicios integrales para el desarrollo de software.
- **HTML**: HyperText Markup Language - hace referencia al lenguaje de marcado para la elaboración de páginas web.
- **REST**: Representational State Transfer - es un estilo de arquitectura software para sistemas hipermedia distribuidos como la World Wide Web.

15 Anexos.

A continuación describiremos el flujo entero de uso del generador, incluyendo la importación del fuente generado en el IDE Eclipse para la ejecución de las pruebas por medio de plugin de Junit del propio IDE.

15.1 Pantalla inicial del generador (especificamos la URL del WSDL).

Desde la interfaz inical del generador tenemos el requerimiento de especificar la URL donde el generador puede localizar el WSDL para así extraer toda la información necesaria sobre el WS que se desea analizar.

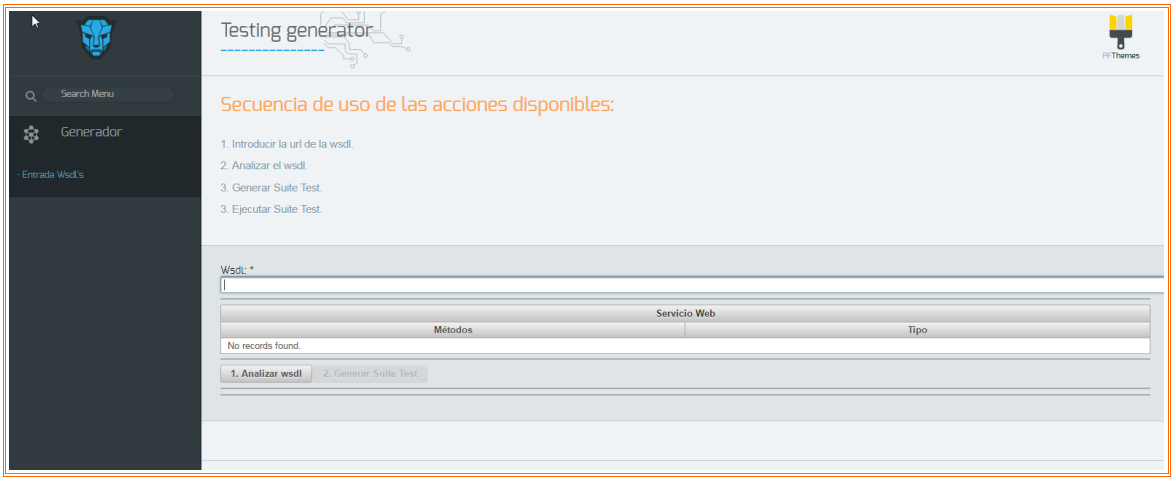


Figura 17: Interfaz inicial.

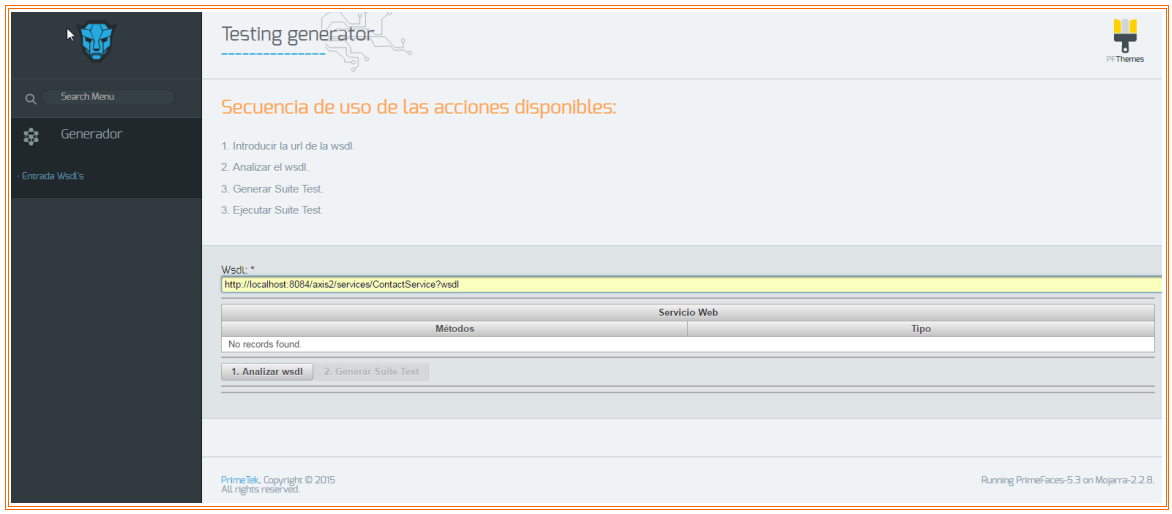


Figura 18: Se especifica la URL del WSDL.

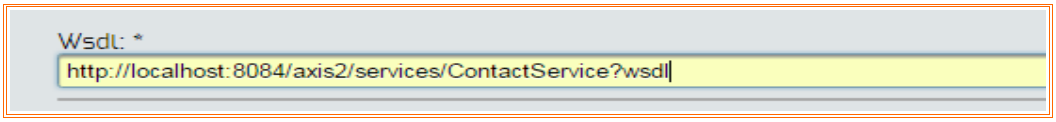


Figura 19: Ruta del WSDL.

15.2 Análisis sobre el WSDL.

- Tras indicar la url donde se encuentra el WS podemos ejecutar la acción de análisis del WSDL.

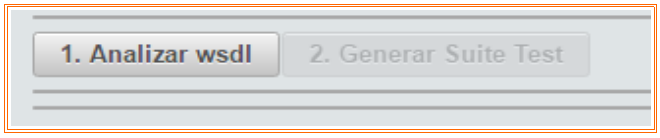


Figura 20: Botón para la ejecución del análisis del WS.

- Tras el análisis la aplicación nos indica que fue realizado con éxito.

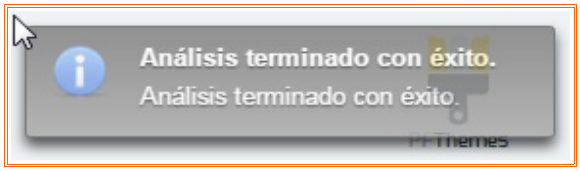


Figura 21: Mensaje de fin de análisis del WSDL.

- Tras la ejecución del análisis del WSDL, se cargará y visualizará toda la información extraída del fichero descriptor y que será usada en la generación del banco de pruebas unitarias.

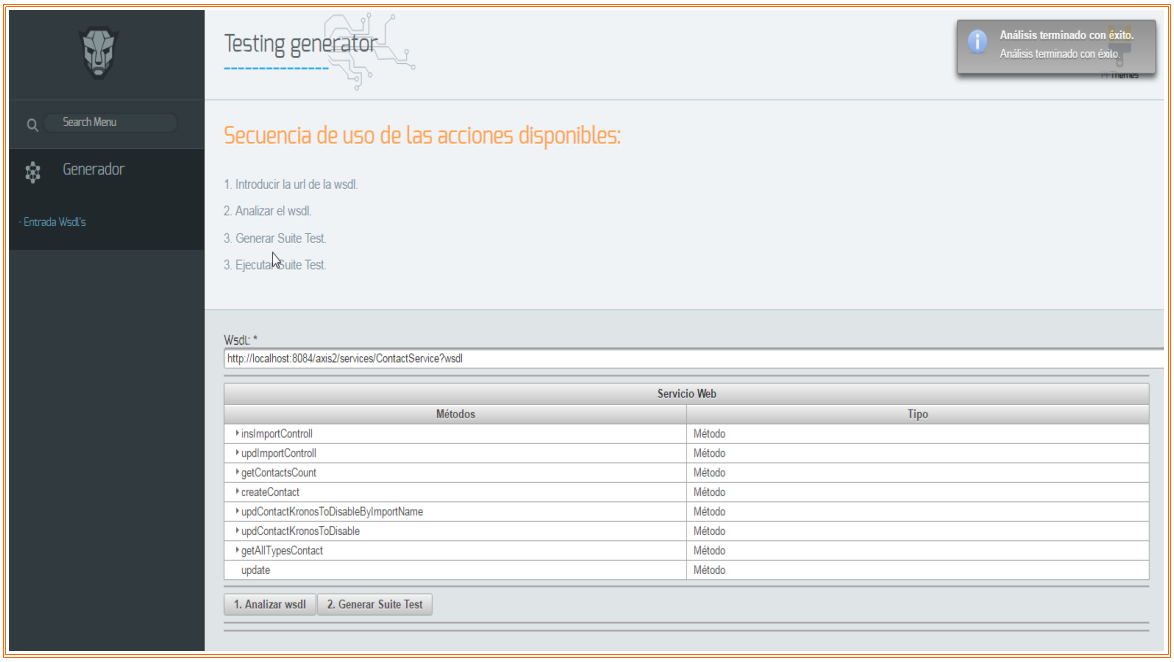
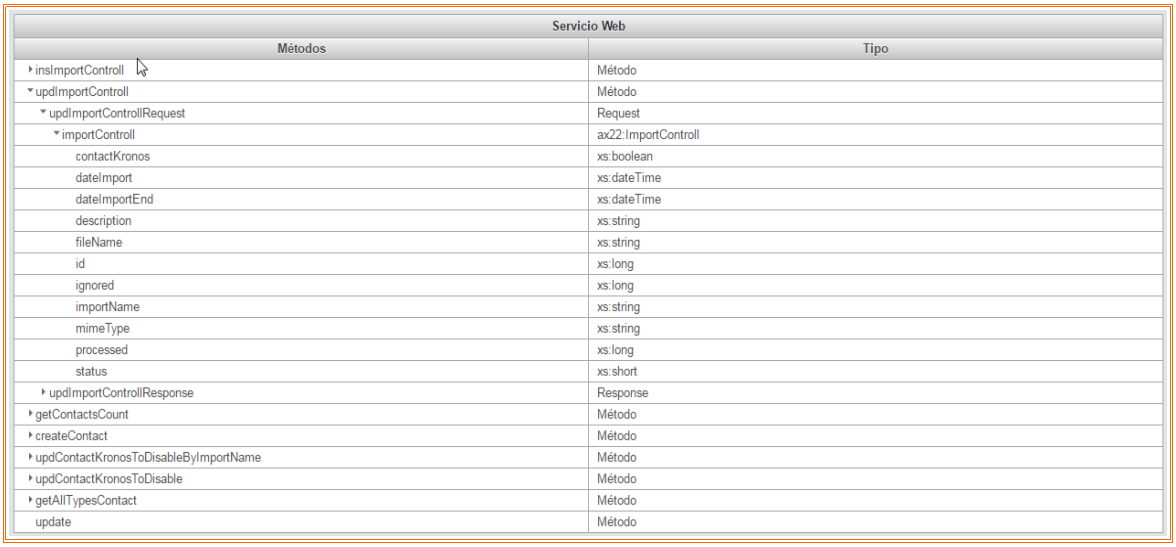


Figura 22: Tras el análisis se visualizan los detalles del WS.

- El nivel de detalle del análisis va desde las operaciones que expone el WS, pasando por los mensajes que componen las peticiones y respuestas, y llegando al nivel de detalle de los parámetros y los tipos de datos que componen estos mensajes.



Servicio Web	
Métodos	Tipo
insImportControll	Método
updImportControll	Método
updImportControllRequest	Request
importControll	ax22:ImportControll
contactKronos	xs:boolean
dateImport	xs:dateTime
dateImportEnd	xs:dateTime
description	xs:string
fileName	xs:string
id	xs:long
ignored	xs:long
importName	xs:string
mimeType	xs:string
processed	xs:long
status	xs:short
updImportControllResponse	Response
getContactsCount	Método
createContact	Método
updContactKronosToDisableByImportName	Método
updContactKronosToDisable	Método
getAllTypesContact	Método
update	Método

Figura 23: Se visualizan detalles de métodos, mensajes de petición y mensajes de respuesta.

15.3 Generación de la TestSuite.

- Una vez realizado el análisis y cargada esta información en el modelo de datos que usará el generador se habilita la operación "Generar Suite Test"

Figura 24: Botón de ejecución para generar la TestSuite.

- Una vez realizada la generación, la aplicación nos indica que ésta fue realizada con éxito.

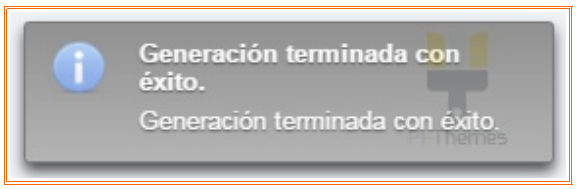


Figura 24: Mensaje de finalización de la generación.

15.4 Descarga del fuente generado.

Tras la generación del proyecto que contiene las pruebas unitarias son habilitadas la siguientes opciones:

- Descargar fuente generado.
- Ejecutar suite test.

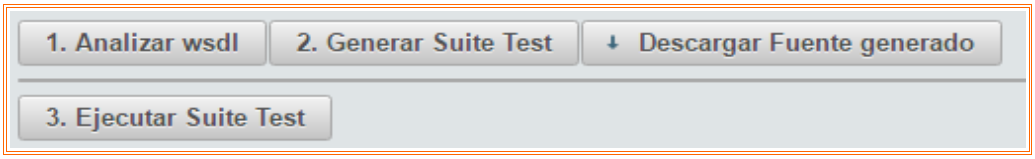


Figura 25: Mensaje de ginalización de la generación.

Al ejecutar la opción de "Descarga de fuente generado" la aplicación nos proporciona un fichero con los fuentes construidos automaticamente por el generador: 

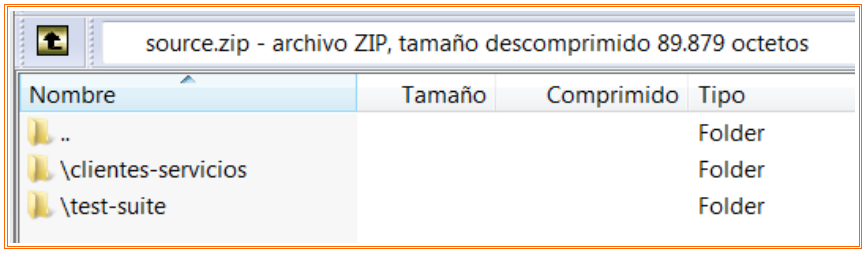


Figura 26: Contenido del fichero comprimido que proporciona la generación.

El fichero comprimido contendrá tanto el cliente o conector como el proyecto que implementa las pruebas unitarias.

- Clientes-servicios. Clientes o conectores usados para el consumo

Figura 27: contenido del proyecto cliente.

- test-suite. Dentro del paquete es.infinitysoft.uned.test se generarán las clases [NombreServicioWeb]Tests.java para cada WS y dentro de ella todos los métodos de test que apuntaran al método correspondiente en el WS.

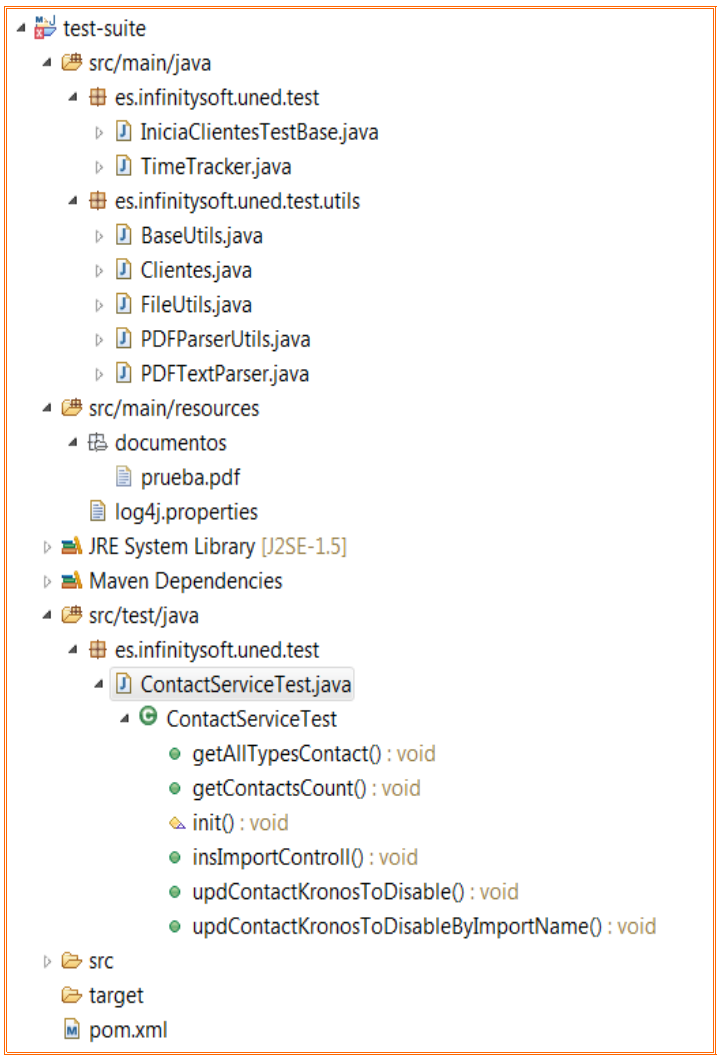


Figura 28: contenido del proyecto Test Suite.

15.5 Ejecución de las pruebas y resultados.

Tras la ejecución de los test's generados se habilitan las acciones que permiten ver los resultados y los log's:

- Resultados.
- Log's

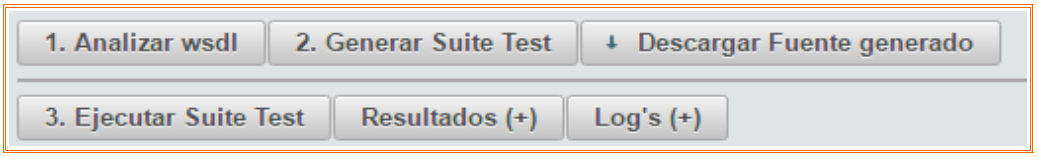


Figura 29: contenido del proyecto Test Suite.

- Si ejecutamos la acción para visualizar los resultados:

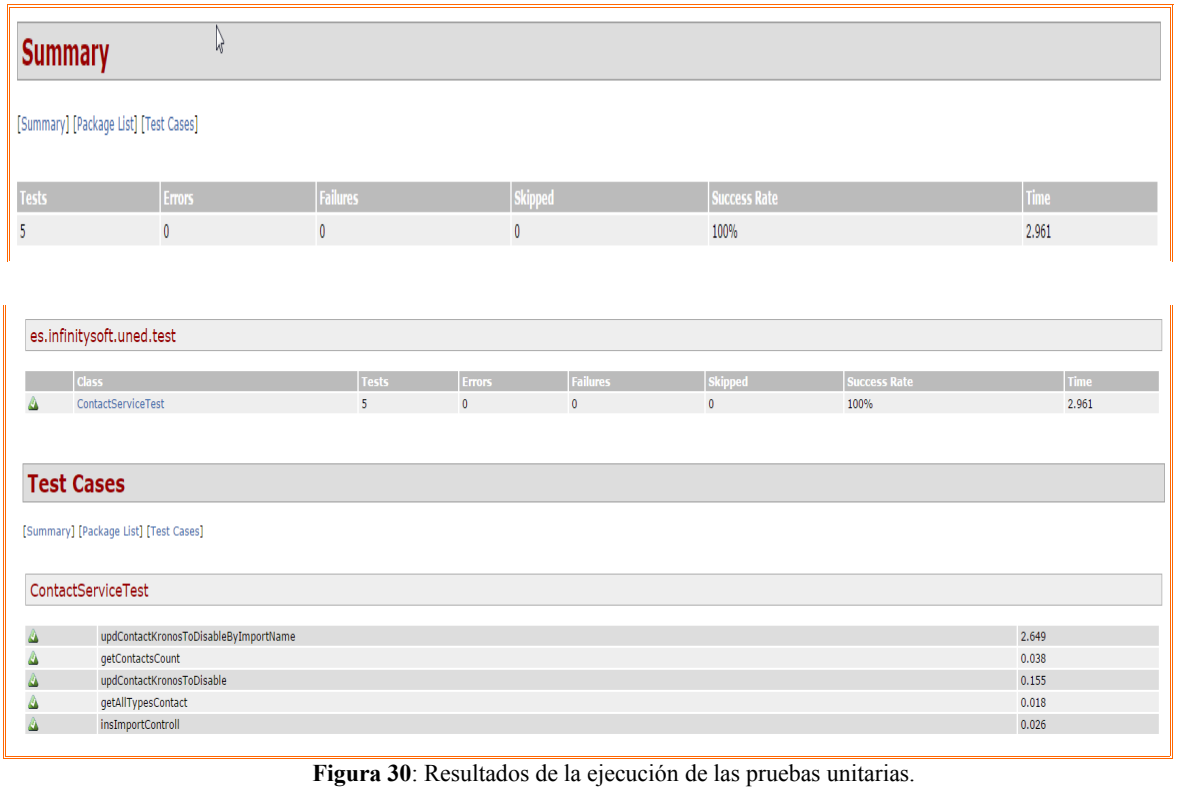


Figura 30: Resultados de la ejecución de las pruebas unitarias.

- Si ejecutamos la opción Log's se puede visualizar la salida de la compilación y ejecución de la suite con los test's por medio de Maven.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running es.infinitysoft.uned.test.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.047 sec
Running es.infinitysoft.uned.test.ContactServiceTest
urlWsdl http://localhost:8084/axis2/services/ContactService?wsdl
[es.infinitysoft.uned.test.utils.BaseUtils] : [URLServicio] http://localhost:8084/axis2/services/ContactService?wsdl
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/C:/Users/tony/.m2/repository/org/ops4j/pax/logging/pax-logging-api/1.6.8/pax-logging-api-1.6.8.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/C:/Users/tony/.m2/repository/org/slf4j/slf4j-log4j12/1.5.2/slf4j-log4j12-1.5.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
may 05, 2016 9:55:12 PM org.apache.cxf.service.factory.ReflectionServiceFactoryBean buildServiceFromWSDL
INFO: Creating Service {http://infinitysoft.es/services/ContactService}ContactService from WSDL: http://localhost:8084/axis2/services/ContactService?wsdl
Resultado: 130634.0
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.469 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] <<< maven-surefire-report-plugin:2.5:report (report:report) @ test-suite <<<
[INFO]
[INFO] >>> maven-surefire-report-plugin:2.5:report-only (report:report-only) @ test-suite >>>
[INFO]
[INFO] <<< maven-surefire-report-plugin:2.5:report-only (report:report-only) @ test-suite <<<
[WARNING] Report plugin org.apache.maven.plugins:maven-project-info-reports-plugin has an empty version.
[WARNING]
[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support building such malformed projects.
[INFO] configuring report plugin org.apache.maven.plugins:maven-project-info-reports-plugin:2.8.1
[WARNING] The artifact org.apache.commons:commons-io:jar:1.3.2 has been relocated to commons-io:commons-io:jar:1.3.2
[WARNING] No project URL defined - decoration links will not be relativized!
[INFO] Rendering site with org.apache.maven.skins:maven-default-skin:jar:1.0 skin.
[INFO] Skipped "Surefire Report" report, file "TestSuite.html" already exists for the English version.
[INFO] Generating "Surefire Report" report    --- maven-surefire-report-plugin:2.5
[WARNING] Unable to locate Test Source XRef to link to - DISABLED
[INFO] Generating "Dependencies" report    --- maven-project-info-reports-plugin:2.8.1
[WARNING] The artifact org.apache.commons:commons-io:jar:1.3.2 has been relocated to commons-io:commons-io:jar:1.3.2
[WARNING] The artifact org.apache.commons:commons-io:jar:1.3.2 has been relocated to commons-io:commons-io:jar:1.3.2
[WARNING] The repository url 'http://snapshots.repository.codehaus.org' is invalid - Repository 'codehaus.snapshots' will be blacklisted.
[WARNING] The repository url 'http://people.apache.org/repo/m2-snapshot-repository' is invalid - Repository 'apache.snapshots' will be blacklisted.
[WARNING] The repository url 'http://people.apache.org/repo/m2-incubating-repository/' is invalid - Repository 'apache-incubator' will be blacklisted.
[WARNING] The repository url 'http://download.java.net/maven/2/' is invalid - Repository 'maven2-repository.dev.java.net' will be blacklisted.
[INFO] Generating "Dependency Convergence" report    --- maven-project-info-reports-plugin:2.8.1
[WARNING] While downloading org.apache.commons:commons-io:1.3.2
  This artifact has been relocated to commons-io:commons-io:1.3.2.
  https://issues.sonatype.org/browse/MVNCENTRAL-244


[INFO] Generating "Dependency Information" report    --- maven-project-info-reports-plugin:2.8.1
[INFO] Generating "About" report    --- maven-project-info-reports-plugin:2.8.1
[INFO] Generating "Plugin Management" report    --- maven-project-info-reports-plugin:2.8.1
[INFO] Generating "Project Plugins" report    --- maven-project-info-reports-plugin:2.8.1
[INFO] Generating "Project Summary" report    --- maven-project-info-reports-plugin:2.8.1
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 33.436s
[INFO] Finished at: Thu May 05 21:55:40 BST 2016
[INFO] Final Memory: 38M/674M
[INFO] ------------------------------------------------------------------------
```

Figura 31: salida de la consola de compilación.

15.6 Importación del Código Generado en Eclipse.

Como hemos mencionado en apartartados anteriores, el generador produce como resultado dos proyectos construidos haciendo uso de Maven y Java. Estos proyectos pueden ser abiertos en cualquier IDE como puede ser Eclipse. A continuación vamos a detallar algunos de los pasos para el montaje de los proyectos, la compilación y la ejecución de las pruebas unitarias en este IDE.

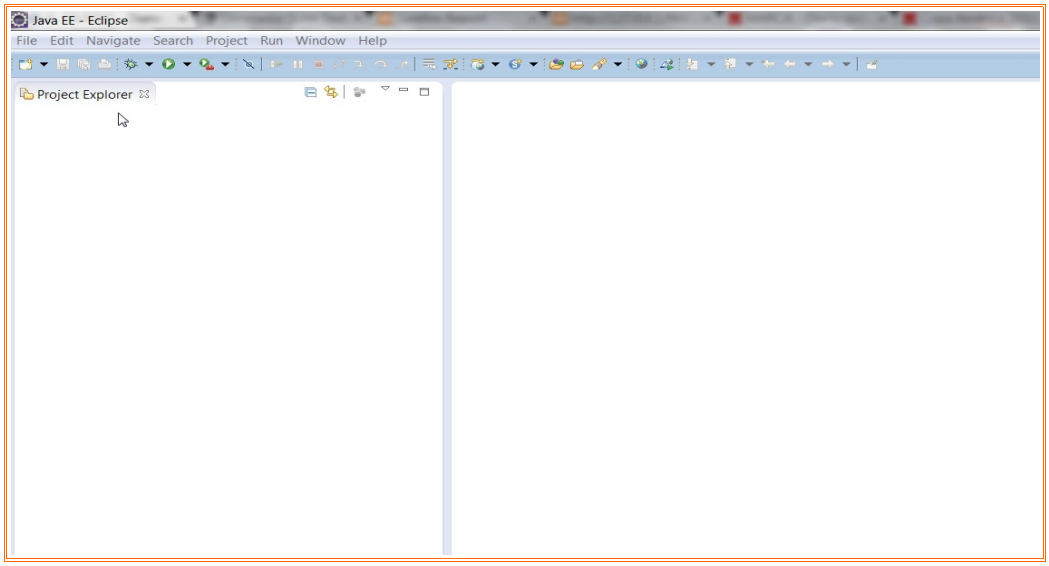


Figura 32: IDE Eclipse.

Podemos hacer la importación del código fuente generado haciendo uso de la opción de importación que contiene Eclipse y que podemos acceder haciendo click con el botón derecho de ratón.

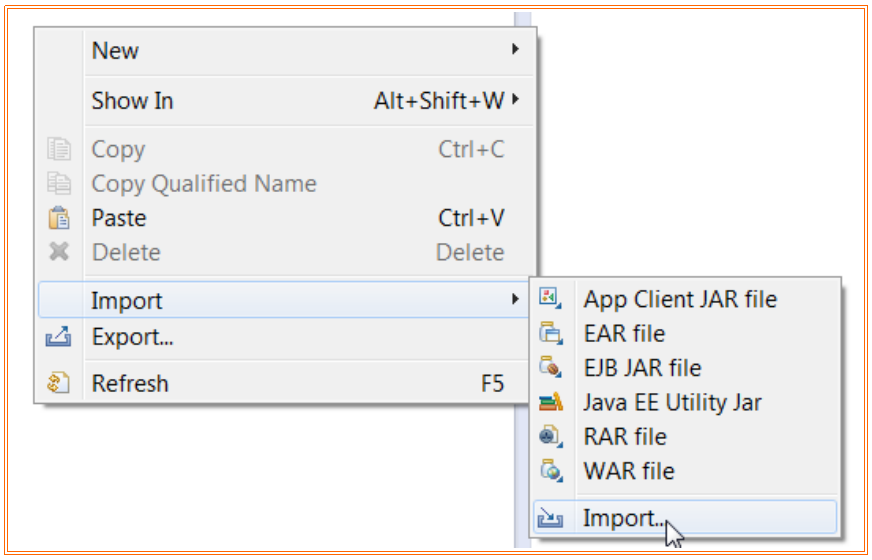


Figura 33: Opción de importación del fuente generado.

Existe multiples formas de importar código existenteen Eclipse, pero nosotros haremos uso de la importación para proyectos generados con Maven. Hacemos la importación del proyecto con los conectores y el proyecto que contiene las pruebas unitarias generadas.

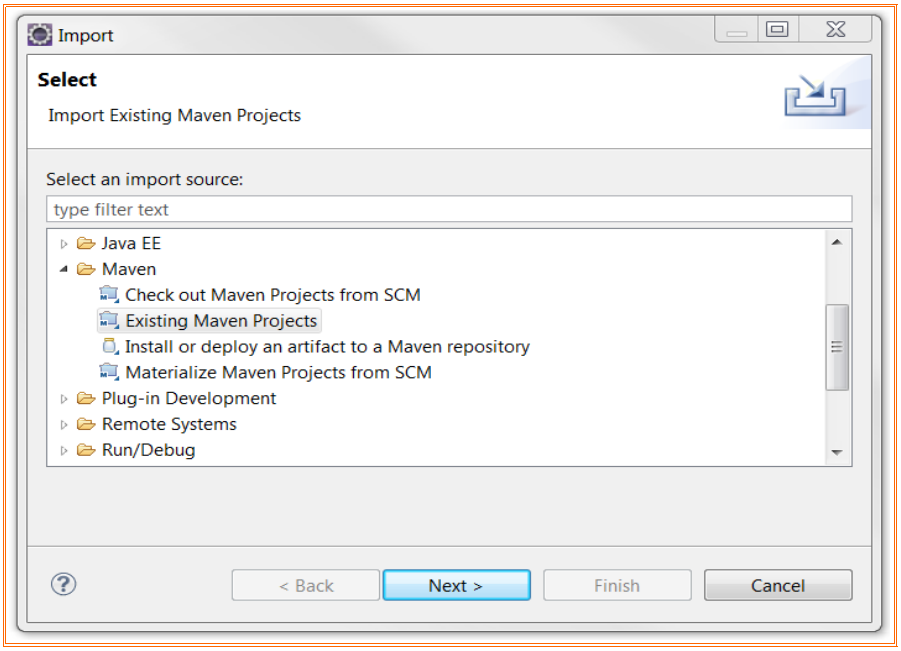


Figura 34: Importación como proyecto maven existente.

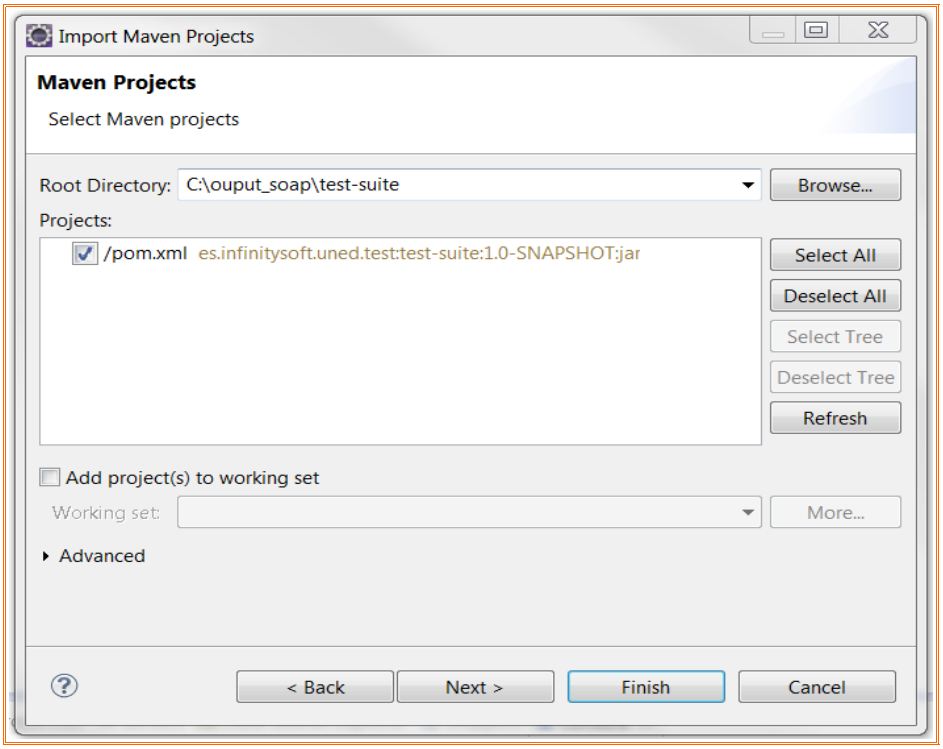


Figura 35: importación del código generado de los clientes del WS para su consumo.

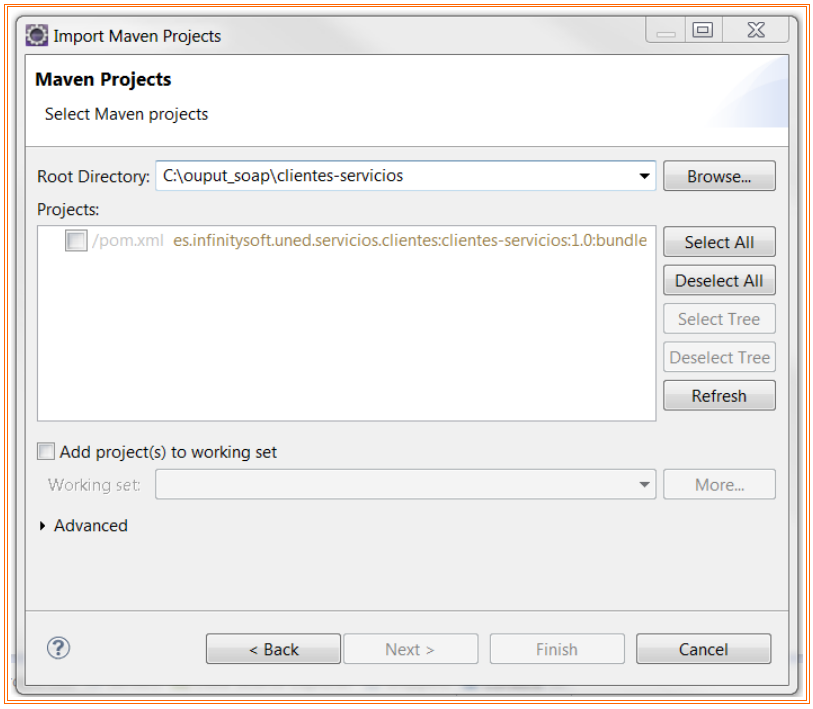


Figura 36: importación del código generado con el proyecto que contiene los Test.

Tras la importación los proyectos se visualizarían en el IDE luciendo de una forma similar a la siguiente:

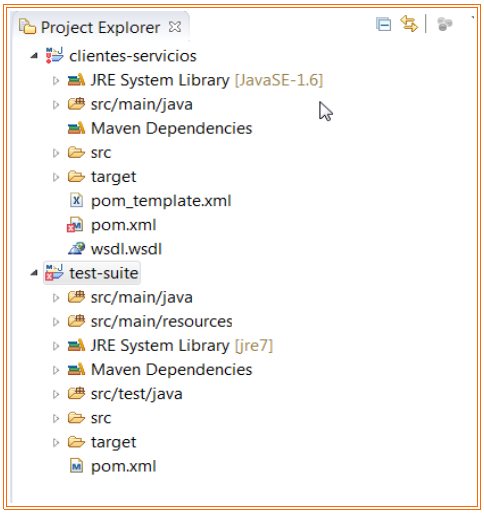


Figura 37: apariencia de los proyectos generados una vez importados en Eclipse IDE.

Realizamos la compilación y ejecución de los Test por medio del plugin de Maven con el que cuenta el IDE Eclipse:

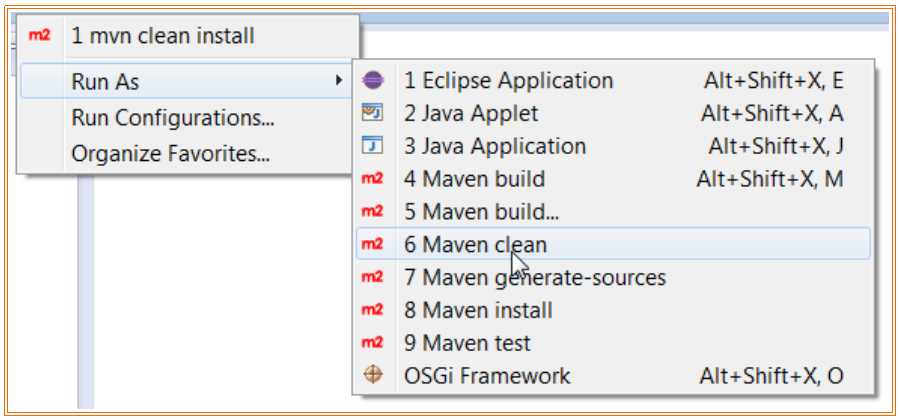


Figura 38: Ejecución la inicialización del proyecto para su compilación.

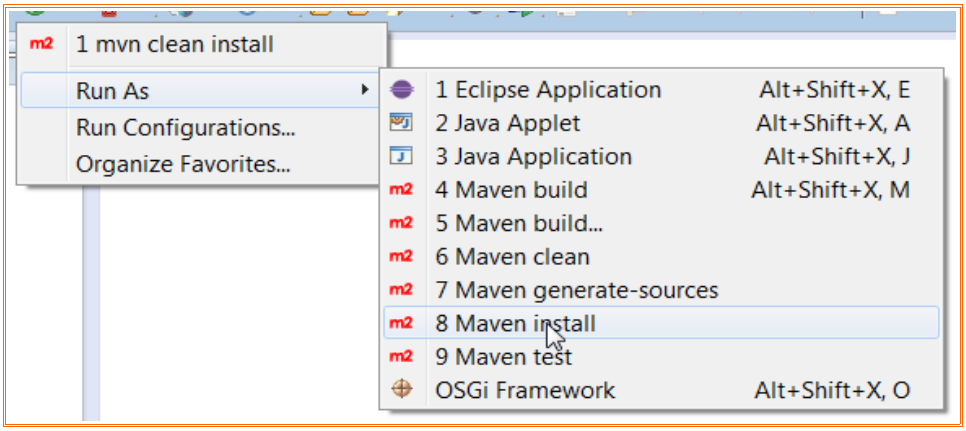


Figura 39: Ejecución de la compilación.

El IDE también cuenta con un plugin Junit, que facilita la ejecución de las pruebas unitarias y la visualización de los resultados:

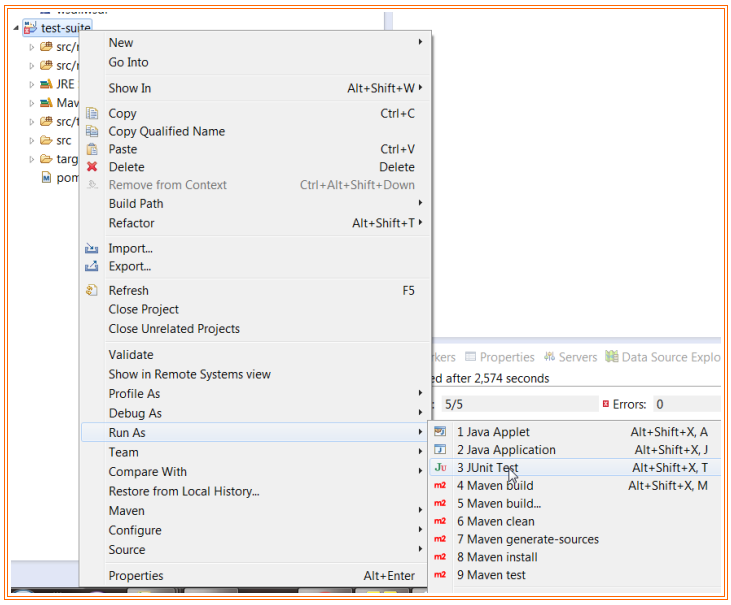


Figura 40: Ejecución de los test.

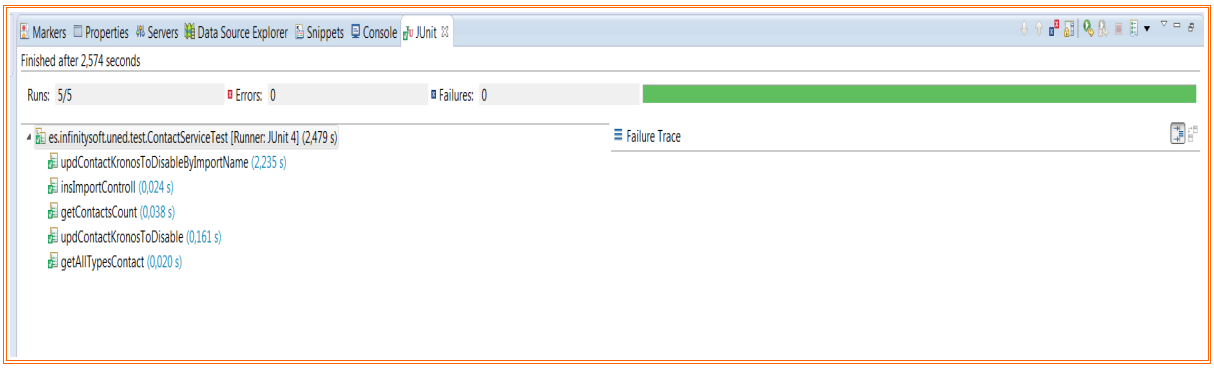


Figura 41: Resultados de la ejecución de las pruebas unitarias por medio del plugin Junit de Eclipse.