Patrones de Seguridad Software en el Contexto de la Arquitectura Multicapa para la Plataforma J2EE

Proyecto Fin de Grado en Ingeniería del Software

Grado en Ingeniería del Software

Facultad de Informática

Universidad Complutense de Madrid

Autores: Sergio Martín Gómez y Eduardo Romero Palencia

Director: Antonio Navarro Martín

Curso 2018/2019

Resumen

Durante las últimas décadas la dependencia de los sistemas informáticos en todos los ámbitos de la sociedad ha crecido de forma exponencial. Hoy en día cualquier empresa depende de un sistema informático para almacenar información, gestionar su actividad comercial o tomar decisiones de negocio. Además, cualquier persona en su vida diaria realiza gestiones mediante sistemas informáticos, como la gestión de sus cuentas bancarias, el pago de los impuestos, compras en internet, etc. Debido a la rápida adopción de tecnologías en red han aparecido nuevas amenazas tales como interrupciones de servicio, accesos no autorizados, suplantación o robo de identidad, robo de datos confidenciales y muchos más riesgos de seguridad. Estas amenazas exponen la importancia de la seguridad e imponen a las empresas una responsabilidad legal y ética de almacenar de forma segura la información. Forzar los mecanismos de seguridad a todos los niveles asegura que la información es procesada, almacenada o transmitida de forma segura.

En este proyecto hemos dotado de seguridad a todos los niveles a una aplicación en el contexto de la arquitectura multicapa para la plataforma J2EE. Mediante este proyecto no sólo hemos protegido a la aplicación de ataques como inyección SQL o acceso no autorizado, sino que también se han registrado todas las operaciones realizadas de forma segura, con el propósito de conocer al autor, la fecha y la operación realizada. Los mecanismos de seguridad implementados también incluyen, entre otros, la transmisión de datos de forma segura a través de la red mediante WS-Security y HTTPS, la monitorización de los servicios web y la intercepción de los mensajes enviados a través de la red para su autorización en función de distintos parámetros como la IP.

Palabras clave

Servicios Web, J2EE, Patrones de seguridad, SAML, WS-Security, UML, Apache CXF, HTTPS, STS

Abstract

During the last decades, the dependency on computer systems in all areas of society has grown exponentially. Nowadays, any business depends on a computer system to store information, manage its commercial activities or make business decisions. In addition, any person in his daily life manages arrangements using a computer system, such as managing his bank accounts, paying taxes, shopping on the internet, and so on. Due to the quick adoption of network technologies new threats have appeared such as service interruption, unauthorized access, impersonation or identity theft, confidential information theft and many more security risks. These threats expose the importance of security and impose to companies a legal and ethical responsibility to store this information securely. Enforcing the security mechanisms at all levels ensures that information is processes, stored or transmitted securely.

In this project, we have provided security at all levels to an application in the multi-tier architecture context for the J2EE platform. Not only have we protected the application against attacks like SQL injection or unauthorized access, but we have also registered every operation securely, with the purpose of knowing the author, the date and the operation performed. The security mechanisms implemented also include, among others, the transmission of data securely through the network using WS-Security and HTTPS, the monitoring of web services and the intercepting of messages sent through the network for its authorization based on different parameters such as the IP of the request.

Keywords

Web Services, J2EE, Security Patterns, SAML, WS-Security, UML, Apache CXF, HTTPS, STS

Índice

Índice de figuras

1.Introducción

1.1. Antecedentes

Debido a la creciente importancia de los sistemas informáticos, de sus datos, y de su transmisión, la seguridad informática se ha vuelto un factor muy importante en el desarrollo de software. Los principales riesgos que deben ser tratados son el acceso a datos sin autorización, siendo estos confidenciales o secretos, la modificación de datos sin autorización, que puede dar lugar a inconsistencias o datos erróneos, e incluso a la eliminación de los datos. Otro riesgo importante es la denegación de servicio, en la que usuarios u otros sistemas pueden evitar que los usuarios legítimos puedan usar su sistema con normalidad. También hay que considerar como riesgo la falta de asociación de los usuarios con las acciones que llevan a cabo, ya que estos no deberían ser capaces de negar lo que hayan hecho en el sistema (principio de no repudio) (Fernandez-Buglioni, 2013; Steel et al., 2005).

Uno de los ejemplos más claros que ilustra la necesidad de utilizar patrones y protocolos de seguridad en las aplicaciones informáticas es el *Caso Enron* (Monk & Wagner, 2012), sucedido en octubre de 2001, y que llevó a la quiebra de la empresa. En este caso, un equipo de ejecutivos pudo ocultar grandes cantidades de dólares en deudas y perdidas por proyectos fallidos, engañando tanto a la directiva como al comité auditor, debido a la falta de registros de estos movimientos. Para evitar que estos hechos pudieran repetirse en el futuro, surgió la *Ley Sarbanes-Oxley* (Sarbanes-Oxley Act of 2002, Pub. L. No. 107-204, 116 Stat. 745) que expandió las repercusiones por alterar, destruir o fabricar registros o por tratar de estafar a los accionistas. A partir de la entrada en vigor de esta ley en Estados Unidos, la necesidad de autenticar cambios, mantener la integridad en los datos, llevar registros de cualquier operación y poder auditarlos externamente paso a ser obligatoria.

A pesar de esto, no hay demasiada información disponible sobre las distintas formas de añadir seguridad a las aplicaciones y la que hay disponible suele ser excesivamente compleja o utilizar un lenguaje demasiado técnico, conduciendo a errores o a la imposibilidad de la implementación.

Por ello, el objetivo de este trabajo es implementar distintos patrones de seguridad en el contexto de una aplicación empresarial J2EE (Oracle, 2000) para generar una guía básica con un lenguaje claro y con documentación suficiente, que permita la implantación de seguridad en distintas aplicaciones de una manera eficiente.

En el contexto de las aplicaciones empresariales, la *arquitectura multicapa* es la arquitectura de referencia (Alur et al., 2003; Fowler, 2002). Las capas en las que se divide esta arquitectura son:

- *Capa de Cliente*: es la capa de acceso a la aplicación, como, por ejemplo, personas u otros sistemas.
- *Capa de Presentación*: es la capa que presenta la lógica de la aplicación, comunicando información de los procesos a la capa de cliente y capturando la información proporcionada por el usuario para mandarla a la capa de negocio.
- *Capa de Negocio*: es la capa que contiene la lógica de negocio, es decir, donde están contenidas las reglas que hay que cumplirse. Recibe las peticiones del usuario y envía las respuestas tras el proceso. Se comunica con la capa de presentación para recibir las solicitudes y mostrar las respuestas, y con la capa de integración para solicitar a la capa de recursos almacenar o recuperar datos.
- *Capa de Integración*: capa a través de la cual se accede a los datos almacenados en la capa de recursos. Normalmente, recibe peticiones de almacenamiento o recuperación de la información desde la capa de negocio.
- *Capa de Recursos*: es la capa donde residen los datos, como sistemas de gestión de bases de datos relacionales, u otros sistemas heredados.

Debido a la presión por proteger y evitar el robo de datos y los ataques, las organizaciones necesitan un mayor control de acceso y seguridad en las comunicaciones. Por ello, es necesario proteger todas las capas de una aplicación empresarial, desde el acceso a la interfaz, hasta la escritura en la capa de recursos, pasando por la protección de la lógica de negocio que inicia los distintos procesos.

Un requisito importante es que la seguridad en las aplicaciones empresariales sea ortogonal a su desarrollo. Así, la aplicación debería poder desarrollarse casi sin elementos de seguridad, y estos, deberían poder añadirse de manera modular a la misma. De esta forma, una misma aplicación sería portable a distintos entornos de despliegue, donde los requisitos y las técnicas de seguridad podrían variar (Steel et al., 2005).

Existen diversas fuentes de patrones de seguridad para aplicaciones empresariales tales como (Steel et al., 2005) o (Fernandez-Buglioni, 2013). Un trabajo fin de máster previo de esta facultad había identificado a (Steel et al., 2005) como la fuente de patrones más cercana a la arquitectura multicapa (Parra, 2014), y había hecho un primer intento de fusionar el catálogo de arquitectura multicapa de Alur et al. (2003) con el catálogo de patrones de seguridad de Steel et al., (2005). No obstante, el trabajo del 2014, al ser un trabajo fin de máster, se había quedado a un nivel más teórico, centrado en arquitecturas con independencia de plataformas, sin llegar a implementar una aplicación multicapa de cierta entidad que incluyese los patrones de seguridad en una plataforma concreta de desarrollo empresarial como Oracle J2EE (Oracle, 2000) o Microsoft .NET (Microsoft, 2002).

El trabajo realizado en este trabajo fin de grado (TFG), fusiona en el contexto de la plataforma J2EE el catálogo de patrones de Steel et al. (2005), con el de Alur et al. (2003), ilustrando hasta sus últimas consecuencias la integración de patrones de seguridad en una arquitectura multicapa desarrollada sobre la plataforma J2EE. Es por ello, que esta memoria incluye multitud de *diagramas de despliegue UML* (ISO/IEC 19505-2:2012), fundamentales a la hora de describir el uso de marcos, ficheros de configuración y otros artefactos imprescindibles para el correcto funcionamiento de las aplicaciones multicapa J2EE seguras.

Steel et al. (2005) agrupa los patrones de seguridad en cinco grandes grupos:

- *Web-Tier*: agrupa los patrones que aseguran la parte relacionada con la interfaz de usuario, siendo esta el punto inicial más frecuente para la mayoría de los ataques.
- *Business-Tier*: se refiere a los patrones relacionados con los componentes que implementan la lógica de negocio en la aplicación.

- *Web-Services*: son aquellos patrones utilizados para asegurar la comunicación entre *Web Service Clients* (WSCs) y *Web Service Provider* (WSPs).
- *Identity Management*: agrupa a los patrones usados para tratar la gestión de identidades.
- *Service Provisioning*: contiene a los patrones que centralizan y gestionan el acceso de los usuarios a los sistemas corporativos.

Como se puede observar, las capas de aplicación consideradas por los catálogos de Alur et al. (2003) y de Steel et al. (2005) no coinciden al 100%, por lo que la integración de ambos catálogos no es ni mucho menos una cuestión evidente.

1.2. Objetivos

El objetivo fundamental de este trabajo es implementar el catálogo de patrones de seguridad propuesto por Steel et al. (2015) en el contexto de una aplicación multicapa J2EE[1].

Con tal fin, se parte de una aplicación multicapa desarrollada como trabajo fin de grado por de Miguel (2018), a la que se aplican los patrones de seguridad del catálogo de Steel et al. (2015). Así, en el contexto de la plataforma J2EE, se refinan los diagramas de clases y de secuencia propuestos por Parra (2014), y se proporcionan diagramas de despliegue para cada patrón de seguridad. Finalmente, los patrones de seguridad son implementados acorde a los diseños propuestos.

La aplicación multicapa de la que se parte, es una sencilla aplicación que ilustra el uso de los patrones de diseño de Alur et al. (2003) en el contexto de una aplicación J2EE. Los únicos elementos de seguridad que contiene esta aplicación de partida son

[1] El código de la aplicación cliente puede encontrarse en `https://github.com/sergiomgm/TFG_cliente`, y el código de la aplicación servidor puede encontrarse en `https://github.com/sergiomgm/TFG_servidor`.

autenticación/autorización en la capa de presentación, e invocación de servicios web con autenticación/autorización a través de conexiones HTTPS.

Así, la aplicación de partida considera la gestión de departamentos, empleados y proyectos de una empresa con los siguientes casos de uso:

- Gestión de departamentos. Gestión CRUD (*Create*, *Read*, *Update*, *Delete*) de departamentos, y cálculo de la nómina de los departamentos como suma de las nóminas de sus empleados.
- Gestión de empleados. Gestión CRUD (*Create*, *Read*, *Update*, *Delete*) de empleados.
- Gestión de proyectos. Gestión CRUD (*Create*, *Read*, *Update*, *Delete*) de proyectos. También se encarga de la gestión de asignaciones de empleados a proyectos.

La mayor parte de patrones de seguridad del catálogo de Steel et al. (2005) ha podido ser implementada sobre el TFG de partida. No obstante, ha sido necesaria la inclusión de un nuevo caso de uso de sincronización de contraseñas entre distintas bases de datos para poder implementar algunos patrones de seguridad.

Por tanto, este TFG se plantea como un proyecto de reingeniería, los más complejos en desarrollo de software (Pressman, 2001).

Este trabajo no sólo se centra en desarrollar una aplicación funcional multicapa segura. El desarrollo de modelos UML de diseño y despliegue que favorezca la comprensión de la integración de patrones de seguridad en el contexto de una arquitectura multicapa es un objetivo clave de este trabajo, y núcleo de esta memoria.

Debido a que el TFG parte de un proyecto ya realizado es necesario distinguir entre las clases diseñadas e implementadas previamente, las diseñadas e implementadas en este TFG y, finalmente, las clases de existentes que han tenido que ser modificadas en este proyecto. Para tal fin, usaremos en los diagramas de clases el siguiente código de colores:

- Color caqui: clases existentes en el trabajo previo no modificadas. Figura 1.1.

Figura 1.1. Clase previa no modificada.

- Color verde: clases de existentes en el trabajo previo que han sido modificadas en este TFG para añadir patrones de seguridad. Figura 1.2.

Figura 1.2 Clase previa modificada en este TFG.

- Color azul: clases desarrolladas en este proyecto, o clases Java, o de marcos reutilizados. Figura 1.3.

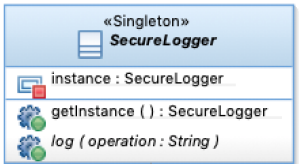

Figura 1.3. Clase desarrollada en este TFG.

Cabe destacar que, desde un punto de vista docente, este trabajo nos sitúa ante una dimensión fundamental de las aplicaciones empresariales que sólo es tratado de forma tangencial en el Grado de Ingeniería del Software. Por tanto, otro objetivo importante del trabajo es mejorar la formación recibida como alumnos, con el fin de subir un escalón importante en la formación recibida en el grado.

Merece la pena resaltar que este trabajo se centra en los patrones de seguridad software que pueden garantizar la seguridad de una aplicación multicapa, y no tiene en cuenta otro tipo de seguridad más cercana a sistemas, encargada de repeler ataques como la denegación de servicio (Fernandez-Buglioni, 2013).

1.3. Plan de trabajo

Para desarrollar el trabajo se ha utilizado una metodología ágil, concretamente se ha utilizado una metodología basada en XP (Beck & Andrés, 2004). Se ha elegido esta metodología debido se disponía de un grupo pequeño, con una fácil comunicación con el cliente (el profesor en este caso). Así, por cada grupo de patrones de seguridad, se procedía a una reunión con el director del TFG para presentar los patrones de seguridad y acordar un diseño previo para cada patrón. Al desconocer totalmente las tecnologías de desarrollo, este diseño previo no era excesivamente detallado, y tras la codificación y prueba del mismo, se procedía a levantar acta del código con modelos de diseño y despliegue detallados, tal y como sugiere XP.

Para utilizar este modelo de proceso se ha recurrido a una planificación que se podía revisar continuamente, al uso de versiones pequeñas que ofrecen código útil y funcional, y se ha integrado al director en el grupo de trabajo para que todas las personas del equipo estuvieran relacionadas de forma directa con el proyecto.

Para la implementación en el proyecto se ha utilizado la plataforma J2EE, ya que se partía de un proyecto anterior implementado en esta plataforma y es una de las más importantes para el desarrollo de aplicaciones empresariales. Por el mismo motivo de reingeniería de proyectos se ha utilizado Apache Tomcat 9.0 (Apache Software Foundation, 2018) como contenedor de *servlets* y MySQL Server 8.0 (Oracle, 2018) como sistema de gestión de bases de datos relacional.

Como entorno de modelado se ha utilizado IBM Rational Software Architect Designer 9.6 (IBM RSA, 2018) y como entorno de desarrollo Eclipse Photon (IBM Eclipse, 2018).

Para la gestión de los repositorios se ha utilizado GitHub para el código (GitHub Inc., 2008) y SVN (Apache, 2000) para los diagramas realizados en IBM Rational Software Architect Designer.

Un aspecto fundamental para el desarrollo del trabajo ha sido la formación necesaria para acometer el desarrollo de este, empezando por entender las tecnologías utilizadas en el TFG

anterior y las que se iban a tener que utilizar en el propio desarrollo del proyecto. Estas tecnologías son:

- *JavaServer Faces* (JSF) (Geary, 2006) para poder modificar la capa de presentación del proyecto.
- *JavaServerPages* (JSP) (Hall, 2003) ya que es la tecnología que permite el despliegue de las paginas JSF.
- *Java API for RESTful Web Services* (JAX-RS) (Burke, 2013) para el despliegue de servicios web REST (*Representational State Transfer*) (Fielding, 2000).
- *Java API for XML Web Services* (JAX-WS) (Hansen, 2007) para el despliegue de servicios SOAP (*Simple Object Access Protocol*) (W3C, 2000).
- *Java Authentication and Authorization Service* (JAAS) (Coté, 2009) para realizar el control de autenticación y acceso en la aplicación.

Para realizar una formación efectiva se procedió a realizar un prototipo simple en los primeros meses utilizando estas tecnologías, aplicando también *WS-Security* (OASIS, 2002) como tecnología que se iba a implantar en primer lugar en el proyecto existente.

Para aplicar los patrones de seguridad ha sido necesaria la formación en SAML (*Security Assertion Markup Language*) (OASIS, 2005) para comprender la estructura de estos mensajes y en *WS-Policy* (W3C, 2006) para poder utilizar archivos que definan políticas de acceso y utilizarlos con mensajes SAML. También ha sido necesario analizar el estándar XACML (*eXtensible Access Control Markup Language*) (OASIS, 2013) para el uso de políticas de control de acceso.

Por último, ha sido necesario asimilar los 22 patrones de seguridad del catálogo de Steel et al. (2015), así como algunos patrones de arquitectura multicapa que no habían sido vistos, o sólo de manera tangencial, en el grado.

1.4. Estructura del documento

Debido a la cantidad de información que incluye esta memoria hemos optado por no hacer un resumen de las principales tecnologías utilizadas en el proyecto, siendo las referencias

incluidas en esta introducción muy valiosas a la hora de tener una visión sobre ellas. Por tanto, la memoria se centra en describir la implementación de los patrones de Steel et al. (2015) en el contexto de una arquitectura multicapa J2EE. Así, la sección 2 se centra en el uso de WS-Security en una aplicación construida sobre Apache CXF (Apache Software Foundation, 2008). Las secciones 3-7 describen la implementación de los patrones de Steel et al. (2015) acorde a la agrupación que realizan los autores en cinco categorías. Finalmente, la sección 8 recoge las conclusiones y trabajo futuro.

1.Introduction

1.1. Background

Due to the increasing importance of the computer systems, their data, and their transmission, the computer security has become a very important factory in software development. The main risk that must be treated are the data access without authorization, being confidential or secret, the data modification without authorization, that can lead to inconsistences or wrong data, and even to the elimination of the data. Another important risk is the denial of service, by which users or another system prevents legitimate users to use their system normally. Another factor to consider as a risk is the lack of association of the users with their actions, because they shouldn't be able to deny what they have done in the system (principle of non-repudiation) (Fernandez-Buglioni, 2013; Steel et al., 2005).

One of the main examples, that shows the need of the use of security patterns and protocols in software applications is the *Enron Case* (Monk & Wagner, 2012), that happened in October 2001, and led the company to bankruptcy. In this case, a team of executives could hide large amounts of debts and losses in failed projects, cheating the directive and the auditors, due to the lack of logs of these movements. To avoid this to happen in the future, the Sarbanes Oxley Law (Sarbanes-Oxley Act of 2002, Pub. L. No. 107-204, 116 Stat. 745) was created, expanding the consequences for modifying, destroying or creating registers or trying to cheat the shareholders. From the entrance of this law in the United States, the need of authenticate changes, keep data integrity, maintain logs of every operation and be able to audit them externally become obligatory.

Despite this, there are not a lot of information available about the different ways to add security to applications and, the information available is very complex or use a very technical language, leading to mistakes or to the impossibility of implementation.

Therefore, the objective of this work is the implementation of different security patterns in the context of a J2EE (Oracle, 2000) business application to make a basic guide with a clear language and enough documentation, that allows the security implantation in different applications in an efficient way.

In the context of business applications, the *multitier architecture* is the main architecture (Alur et al., 2003; Fowler, 2002). The tiers of this architecture are:

- *Client Tier*: is the tier that provides the access to the application, such as users or other systems.
- *Presentation Tier*: is the tier that presents the application logic, sharing information with the client tier and capturing the information given by the user to send it to the business tier.
- *Business Tier*: is the tier that contents the business logic, where the rules the must be met are. It receives the user requests and sends the answers after the process. It is communicated with the presentation tier to receive the information and to show the answers, and with the integration tier to get data from the resources tier.
- *Integration Tier*: tier that allow the access to the data stored in the resources tier. Normally, it receives storage or recovery requests from the business tier.
- *Resources Tier*: is the tier where the data resides, like database management systems, or another inherited systems.

Due to the pressure to protect and avoid the data theft, the companies need a better access control and security in the communications. Thus, is necessary to protect all the tiers of a business application, from the access to the interface, to the write in the resources tier, through the protection of the business logic that starts the different processes.

An important requirement is that the business application security must be orthogonal to their development. This way, the application can be developed without security elements, and these, can be added in a modular way. This allows applications to be portable among different deployment environments, where the requirements and the security technics could change (Steel et al., 2005).

There are various sources of security patterns to business application as (Steel et al., 2005) o (Fernandez-Buglioni, 2013). A previous master's thesis wrote in this faculty identified (Steel et al., 2005) as the closest source to the multitier architecture (Parra, 2014), and made a first try to merge the multitier architecture catalog (Alur et al., 2003) with the security

patterns catalog (Steel et al., 2005). However, that work, as master's thesis, was written at a more theoretical level, focused on architectures independently of platforms, without implementing a multitier application that includes the security patterns in a concrete business development platform as Oracle J2EE (Oracle, 2000) or Microsoft .NET (Microsoft, 2002).

The work made in this bachelor thesis, merges Steel et al. patterns catalog (2005), with Alur et al. catalog (2003) in the J2EE platform context, illustrating in detail the integration of security patterns in a multitier architecture developed under J2EE platform. Therefore, this document includes a significant number of *UML deployment diagrams* (ISO/IEC 19505-2:2012), that are fundamental for describing the framework use, configuration archives and other indispensable artefacts for the correct working of the secure J2EE multitier applications.

Steel et al. (2005) group security patterns in five groups:

- *Web-Tier*: groups patterns that secure the part associated with the user interface, being this the more frequent initial point for almost every attack.
- *Business-Tier*: is referred to the patterns associated with those components that implements applications business logic.
- *Web-Services*: those patterns used to secure the communication between *Web Service Clients* (WSCs) and *Web Service Provider* (WSPs).
- *Identity Management*: groups the patterns used to treat the identity management.
- *Service Provisioning*: it contents the patterns that centralizes and manages the user access to the corporative systems.

As it can be observed, the application tiers considered by the Alur et al. (2003) and Steel et al. (2005) catalogs aren´t 100% the same, so the integration of both catalogs isn´t obvious.

1.2. Goals

The main goal of this project is the implementation of the security patterns catalog of Steel et al. (2015) in the context of a J2EE multitier application[2].

With this goal in mind, this project considers a multitier application developed as bachelor thesis written by de Miguel (2018), and expands it to include the security patterns identified by Steel et al. (2015). This way, in the J2EE platform context, the class and sequence diagrams suggested by Parra (2014) are refined, and deployment diagrams are given by every security pattern. Finally, the security patterns are implemented according to the proposed designs.

The original multitier application is a simple application that illustrates the using of the Alur et al. (2003) patterns in the context of an J2EE application. The only security elements that are contained in this application was authorization/authentication in the presentation tier, and invocation of web services with authentication/authorization with HTTPS connections.

This way, the original application considers the management of departments, employees and projects of a business with the following use cases:

- Department Management. CRUD management (Create, Read, Update, Delete) of departments, and department payroll calculation as the sum of the payrolls of its employees.
- Employee Management. CRUD management (Create, Read, Update, Delete) of employees.

[2] The source code of the client application can be found in `https://github.com/sergiomgm/TFG_cliente`, and the source code of the server application can be found in `https://github.com/sergiomgm/TFG_servidor`.

- Project Management. CRUD management (Create, Read, Update, Delete) of projects. Is also manages assignation of employees to projects.

Most security patterns of the Steel et al. (2005) catalog have been be implemented updating the original project. However, it was necessary the inclusion of a password synchronization use case among different databases to implement some security patterns.

Thus, this project is conceived as a reengineering project, the most complex one in software development (Pressman, 2001).

This work isn´t just focused on the development of a functional secure multitier application. The development of UML design and deployment models, that favor the comprehension of the security patterns integration in the multitier architecture context, is a key goal in this work, and the core of this document.

Because this project is based on an existing project, it is essential to difference between the classes that were previously designed and implemented and the ones that have been designed and implemented in this project. For this, we use the next color code in the class diagrams:

- Khaki color: existent classes in the previous project that have not been modified. Figure 1.1.

Figure 1.1. Previous non-modified class.

- Green color: existing classes in the previous project that have been modified in this project to add security patterns. Figure 1.2.

Figure 1.2 Previous class modified in this project.

- Blue color: new classes developed in this project, Java classes or from reused frameworks. Figure 1.3.

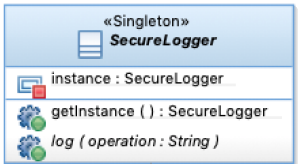

Figure 1.3. New class developed in this project

Is important to point that, from an educational point of view, this project put us against a fundamental dimension of business applications that is only is tangentially presented in the Software Engineering Degree. Therefore, another important goal is to improve the training given received as students, in order to climb an important step in the training received in the grade.

It is worth mentioning that this work is focused on the security software patterns that guarantee the security of multitier applications, and doesn´t take into account another security types nearer to systems, that are in charge to repel attacks such as the denial of service (Fernandez-Buglioni, 2013).

1.3. Workplan

In order to develop this work an agile methodology was used, in concrete an XP-based model (Beck & Andres 2004). This methodology was chosen since a small group was available, with an easy communication with the client (in this case the professor). This way, for every group of security patterns, a meeting with the project director to present the security patterns and agree in a previous design for every pattern was made. Due to the ignorance of the development technologies, this previous design was not very detailed, and after the implementation and testing, the code was documented with detailed design and deployment models, as XP suggests.

For using this process model, a schedule that could be revised continuously was used and small versions that offer functional and useful code were scheduled. The director was integrated in the work group to enhance communication.

In the implementation of the project, the J2EE platform was used, because the project was the continuation of a previous project implemented in that platform and because is one of the most important in the business applications development. For the same reason, Apache Tomcat 9.0 (Apache Software Foundation, 2018) as *servlets* container and MySQL Server 8.0 (Oracle, 2018) as relational database management system were used.

Rational Software Architect Designer 9.6 (IBM RSA, 2018) was used as modeling CASE tool and Eclipse Photon (IBM Eclipse, 2018) was used as IDE.

To manage the repositories, GitHub was used for storing the code (GitHub Inc., 2008) and SVN (Apache, 2000) was used for storing the diagrams made with IBM Rational Software Architect Designer.

Additional knowledge was needed in order to afford the development of this project. Thus, the initial work focused on understanding the technologies used in the previous project and those that were going to be used in the project development. These technologies are:

- *JavaServer Faces* (JSF) (Geary, 2006) for being able to modify the presentation tier in the project.
- *JavaServerPages* (JSP) (Hall, 2003) because is the technology that allow the development of JSF pages.
- *Java API for RESTful Web Services* (JAX-RS) (Burke, 2013) for deploying REST (*Representational State Transfer*) web services (Fielding, 2000).
- *Java API for XML Web Services* (JAX-WS) (Hansen, 2007 for deploying SOAP (*Simple Object Access Protocol*) web services (W3C, 2000).
- *Java Authentication and Authorization Service* (JAAS) (Coté, 2009) to make the authentication and authorization control in the application.

In order to make an effective training, a simple prototype was made in the first months using these technologies, as well as WS-Security (OASIS, 2002) a key technology in the project.

For implementing the security patterns, training in SAML (*Security Assertion Markup Language*) (OASIS, 2005) was needed in order to understand the message structure, and in *WS-Policy* (W3C, 2006) for being able to use archives that defines access policies and use them with SAML messages. It also was necessary to analyze the XACML (*eXtensible Access Control Markup Language*) standard (OASIS, 2013) for using access policies.

At last, it was necessary to assimilate the 22 security patterns of Steel et al. (2015) catalog, and some multitier architecture patterns that hadn´t been seen, or only tangentially, in the grade.

1.4. Document Structure

Due to the information quantity in this document, we choose not to do a resume of the main technologies used in the project, being the references included very valuable in order to have a vision over them. Thus, this document is focused on the implementation of Steel et al. (2015) patterns in the context of J2EE multitier architecture. Section 2 is focused on the use of WS-Security in an Apache CXF-based application (Apache Software Foundation, 2008). Sections 3-7 show the implementation of the security patterns of Steel et al. (2015) according to the five categories described by the authors. Finally, Section 8 presents conclusions and future work.

2. WS-Security sobre Apache CXF

WS-Security (OASIS, 2002) es una extensión de SOAP utilizada para aplicar seguridad sobre estos servicios web. Provee confidencialidad e integridad a estos mensajes, así como el no repudio, y asegura la seguridad extremo a extremo, y no punto a punto. La primera garantiza la seguridad del mensaje desde el emisor al receptor, con independencia de las conexiones intermedias. La segunda, sólo garantiza la seguridad en el contexto de una conexión. (Steel et al., 2005).

En este proyecto se ha utilizado WS-Security en los siguientes casos de uso:

- Todas las operaciones de Empleado
- Todas las operaciones de Departamento
- Patrón *secure service proxy* en Proyecto
- Todas las operaciones SOAP implicadas en el patrón *secure message router*

El uso de WS-Security necesita de la presencia de certificados X.509 (ITU-T, 1988). Así, en el proyecto, en primer lugar, se tienen que crear los certificados correspondientes al WSC y al WSP, (`clienttestkey` y `testkey`, respectivamente) en los *keystores* del WSC y del WSP (`clientkeystore` y `keystore`, respectivamente). La herramienta keytool de Java permite hacerlo con los siguientes comandos:

```
keytool -genkey -alias testkey -keystore keystore.jks -storepass storepass -
dname "cn=testkey" -keyalg RSA

keytool -genkey -alias clienttestkey -keystore clientkeystore.jks -storepass
clientstorepass -dname "cn=testkey" -keyalg RSA
```

A continuación, se han de exportar las claves públicas de ambos certificados e importarlas de forma cruzada en los *keystores* que no contienen los certificados. La herramienta keytool de Java permite hacerlo con los siguientes comandos:

```
keytool -export -alias testkey -file publickey.rsa -keystore keystore.jks -
storepass storepass
```

```
keytool -export -alias clienttestkey -file clientpublickey.rsa -keystore
clientkeystore.jks -storepass clientstorepass

keytool -import -alias testkey -file publickey.rsa -keystore
clientkeystore.jks -storepass clientstorepass

keytool -import -alias clienttestkey -file clientpublickey.rsa -keystore
keystore.jks -storepass storepass
```

Para poder acceder a los *keystores* desde la implementación se ha indicado la configuración necesaria en diferentes archivos *properties*. El archivo *properties* del cliente en el que se indica el archivo `clientkeystore.jks` y la contraseña `clientstorepass`.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypt
o.Merlin
org.apache.ws.security.crypto.merlin.keystore.file=clientkeystore.jks
org.apache.ws.security.crypto.merlin.keystore.password=clientstorepass
org.apache.ws.security.crypto.merlin.keystore.type=jks
```

Y el archivo *properties* del servidor en el que se indica el archivo `keystore.jks` y la contraseña `storepass`.

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypt
o.Merlin
org.apache.ws.security.crypto.merlin.keystore.file=keystore.jks
org.apache.ws.security.crypto.merlin.keystore.password=storepass
org.apache.ws.security.crypto.merlin.keystore.type=jks
```

El uso del protocolo WS-Security es transparente a la aplicación, y los cambios en los mensajes SOAP se llevan a cabo mediante los *interceptores* de Apache CXF (Apache Software Foundation, 2008).

En el delegado del negocio del WSC, donde se realiza la conexión con el servidor, se carga el *bean* correspondiente del archivo `cxf.xml`. En este archivo se han establecido los interceptores CXF y se ha indicado que el WSC tiene que encriptar el paquete SOAP con

la clave del usuario `testkey` (que es la clave pública del WSP almacenada en el *keystore* del WSC), y también que tiene que desencriptar los paquetes con la clave del usuario `clienttestkey` (que la clave privada del WSC almacenada en el *keystore* del WSC).

En el archivo `cxf-servlet.xml` del WSP se crean también los interceptores CXF indicando que se tienen que desencriptar los paquetes con la clave del usuario `testkey` (que es la clave privada del WSP almacenada en el *keystore* del WSP), y que encriptar con la clave del usuario `clienttestkey` (que es la clave pública del WSC almacenada en el *keystore* del WSP).

De esta forma conseguiremos encriptar el mensaje SOAP enviado del WSC al WSP con la clave pública del WSP, desencriptarlo en el WSP, y encriptar el mensaje SOAP enviado del WSP al WSC con la clave pública del WSC, y desencriptarlo en el WSC.

La figura 2.1 muestra el diagrama de despliegue que permite usar WS-Security con Apache CXF.

Una vez realizadas estas configuraciones, se puede comprobar como los mensajes en la consola de comandos van encriptados tanto en la comunicación WSC a WSP, como en las respuestas WSP a WSC. En el caso de este TFG se han encriptado bajo WS-Security todos los servicios web SOAP.

Por último, hay que destacar que la aproximación actual de WS-Security sobre CXF reduce el rendimiento de la aplicación debido a las operaciones de firma y encriptado según los estándares XML, debido en parte a que requieren un paso llamado *canonicalization* que convierte el XML a una forma canónica antes de que el valor de la firma sea calculado (IBM, 2018).

La figura 2.1 incluye el diagrama de despliegue que describe los artefactos, nodos y entornos de ejecución necesarios para aplicar WS-Security.

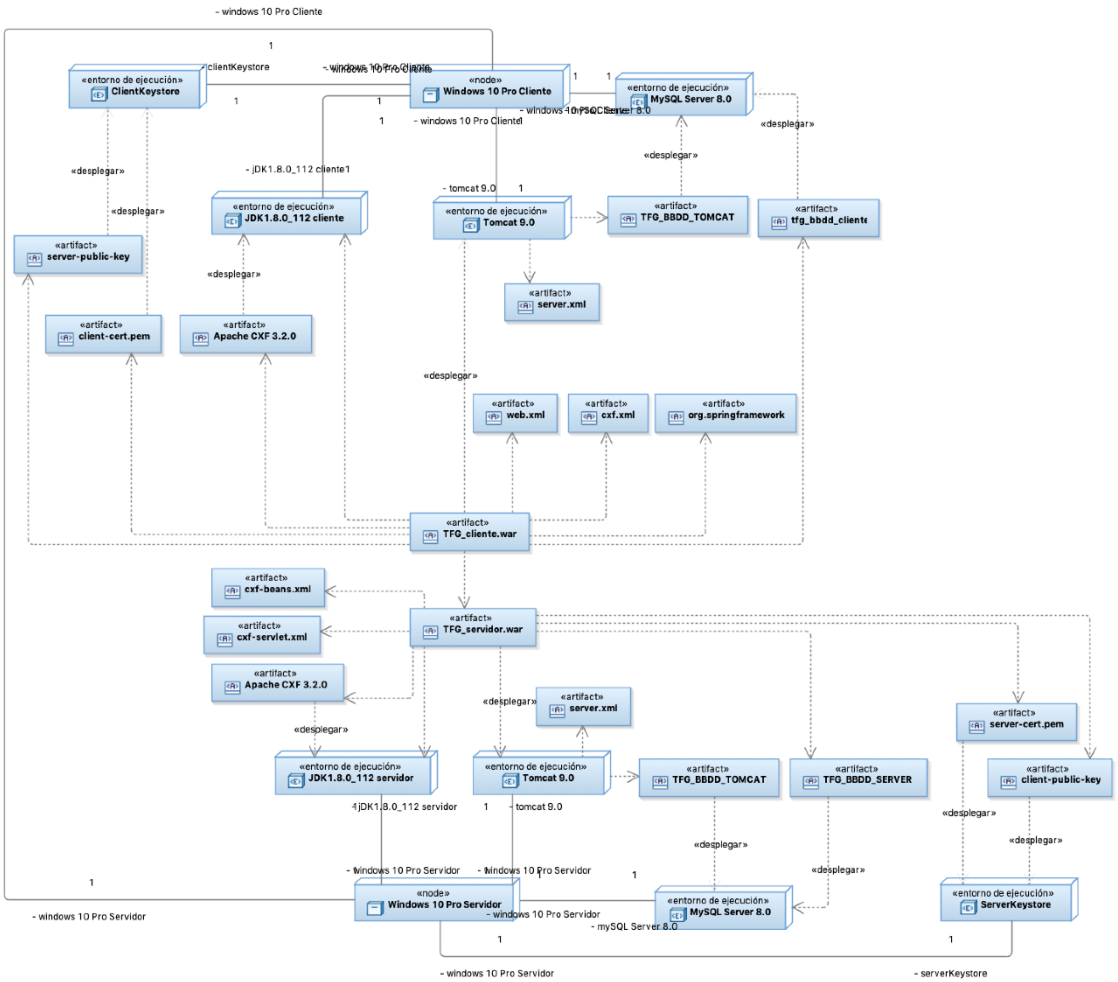

Figura 2.1 - Diagrama de despliegue de WS-Security

3. Patrones Web-Tier

3.1. Authentication y Authorization Enforcer

Estos dos patrones se encargan de autenticar a los usuarios para restringir el acceso a la aplicación, y de proveer una autorización para definir los diferentes niveles de acceso que tiene el usuario.

El *authentication enforcer* (Steel et al., 2005) asume la responsabilidad de autenticar y verificar al usuario, centralizando la autenticación y encapsulando los mecanismos utilizados para realizarla, desacoplando estas responsabilidades de la capa de negocio y de presentación.

El *authorization enforcer* (Steel et al., 2005) provee un control de acceso de grano fino a través de la restricción de acceso a determinadas URL a distintos roles, definidos mediante `Principals` (Oracle, 2019). Centraliza la autorización y encapsula los mecanismos utilizados.

En el caso de este trabajo se ha utilizado JAAS, que implementa ambos patrones, y que ya estaba presente en el TFG de partida. Con JAAS sólo es necesario crear la entidad `Usuario`, las clases `UserPrincipal` y `RolePrincipal` y la clase `LoginModule` que asigna los `Principal` al usuario.

Las figuras 3.1, 3.2, 3.3 y 3.4 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

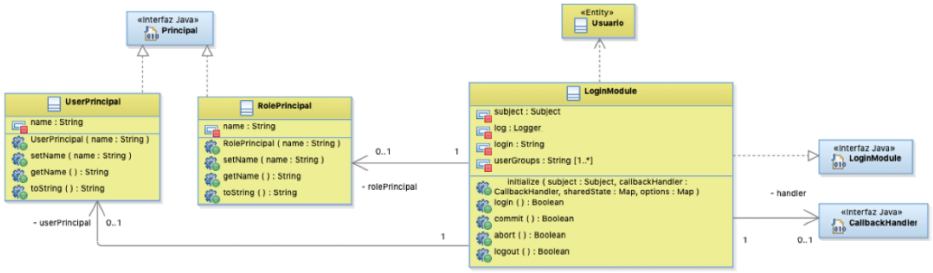

Figura 3.1. Diagrama de clases del patrón *authentication/authorization enforcer*

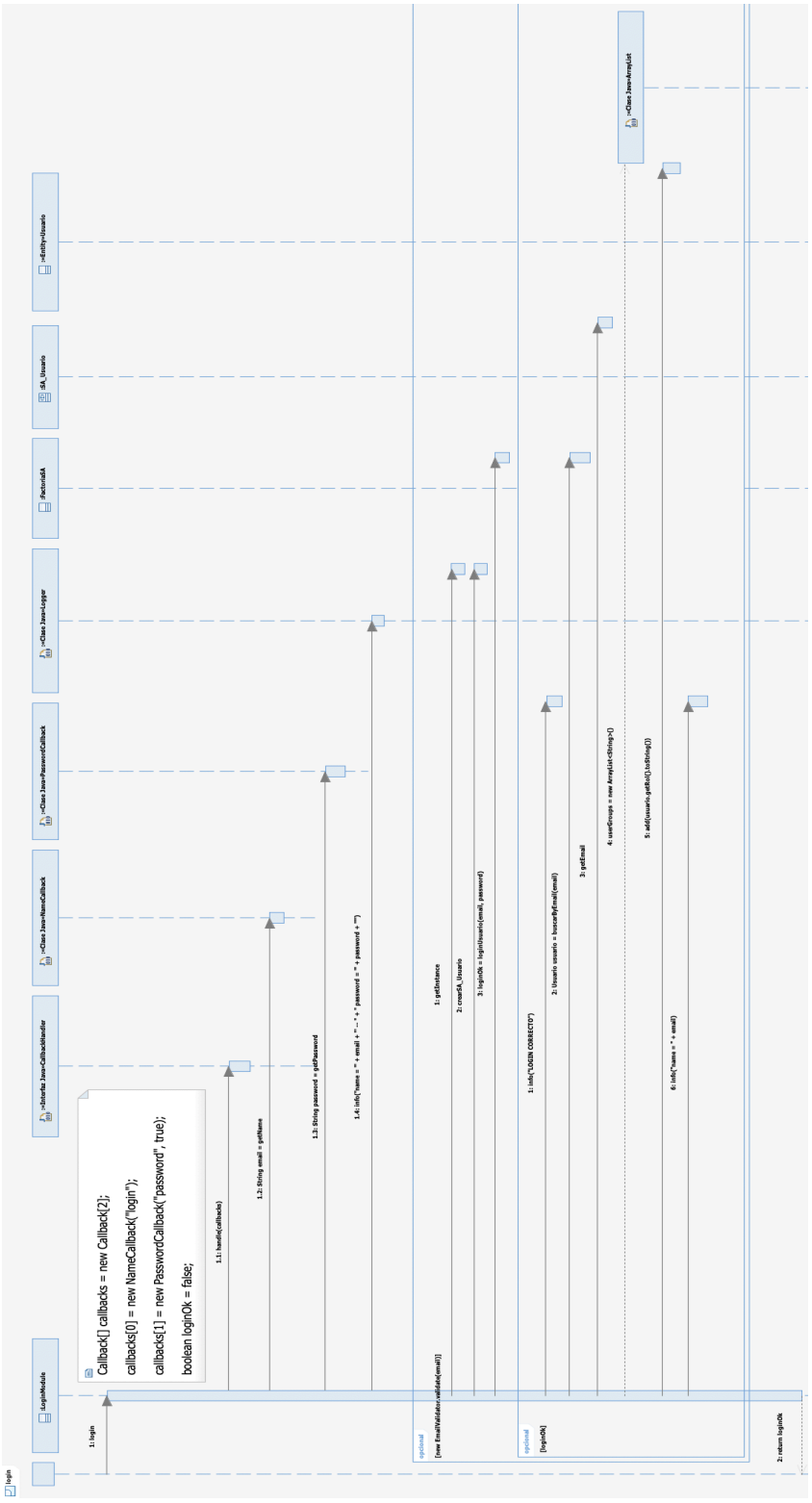

Figura 3.2 – Diagrama de secuencia del patrón *authentication/authorization enforcer* para el método *login* de la clase *LoginModule*

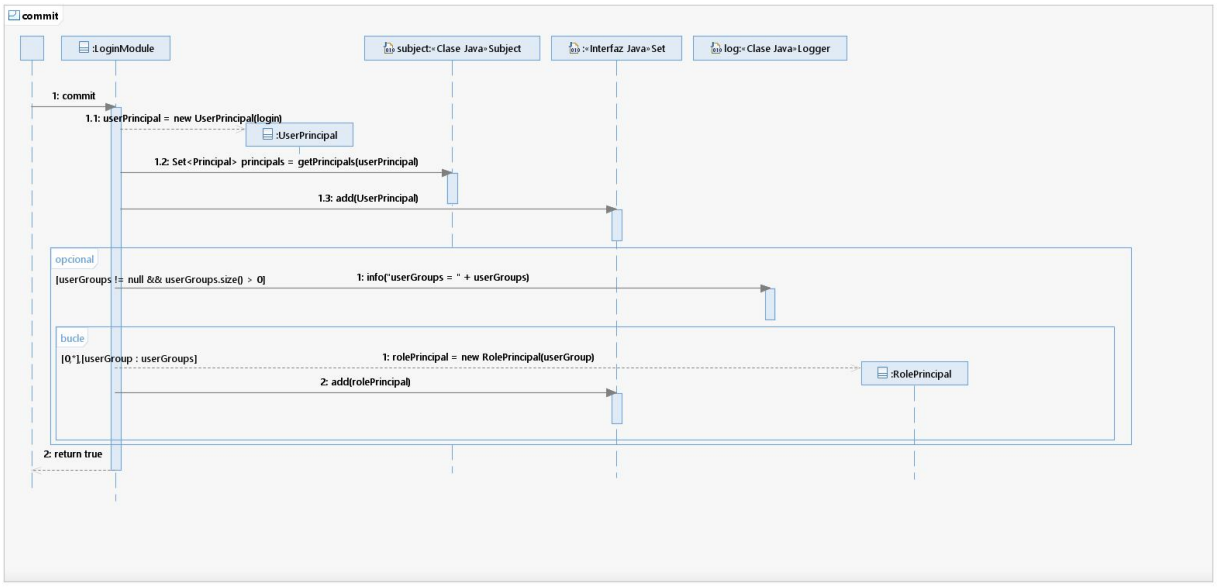

Figura 3.3 – Diagrama de secuencia del patrón *authentication/authorization enforcer* para el método *commit* de la clase *LoginModule*

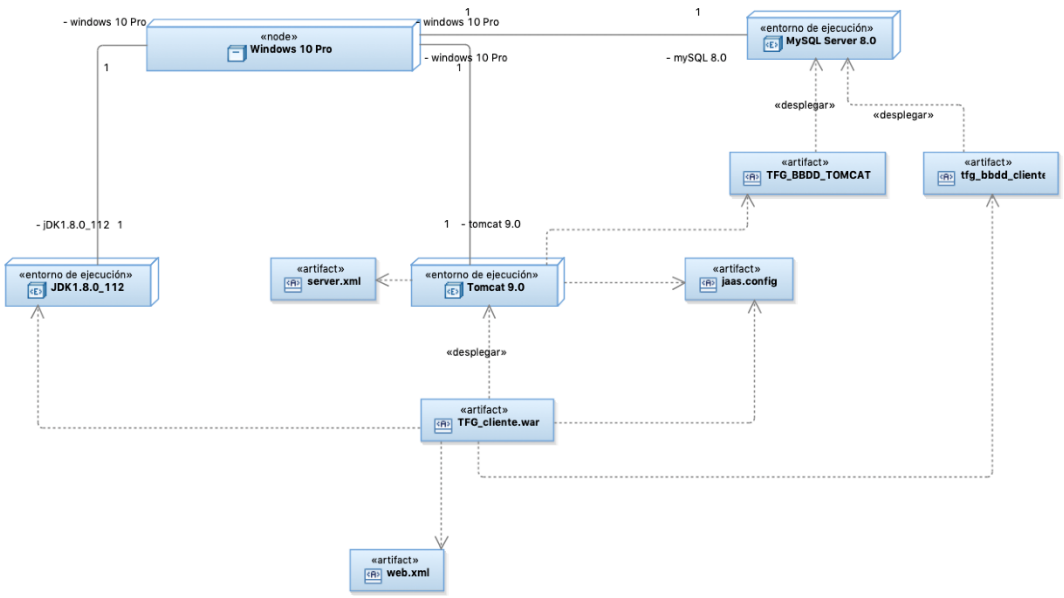

Figura 3.4 - Diagrama de despliegue del patrón *authentication/authorization enforcer*

3.2. Intercepting Filter

El objetivo del *intercepting filter* (Steel et al., 2005) es evitar ataques que puedan comprometer el sistema mediante el envío de datos inválidos o mediante la inyección de código malicioso (scripts, sentencias SQL, contenido XML…). Para ello, verifica todas las entradas mediante filtros que contienen una lógica de validación.

Estas verificaciones deben realizarse en el lado del servidor, ya que, si se realizan mediante JavaScript en el lado web, pueden ser obviadas fácilmente.

Para su implementación en este proyecto se han utilizado los filtros *servlet* (Oracle, 2017), creando las clases `SQLFilter` (que filtra la introducción de código SQL) y `ValidatorFilter` (que valida las entradas del usuario para evitar caracteres extraños en los formularios). Ambas clases implementan la interfaz `Filter` de los *servlets* Java.

Las figuras 3.5, 3.6, 3.7, 3.8 y 3.9 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

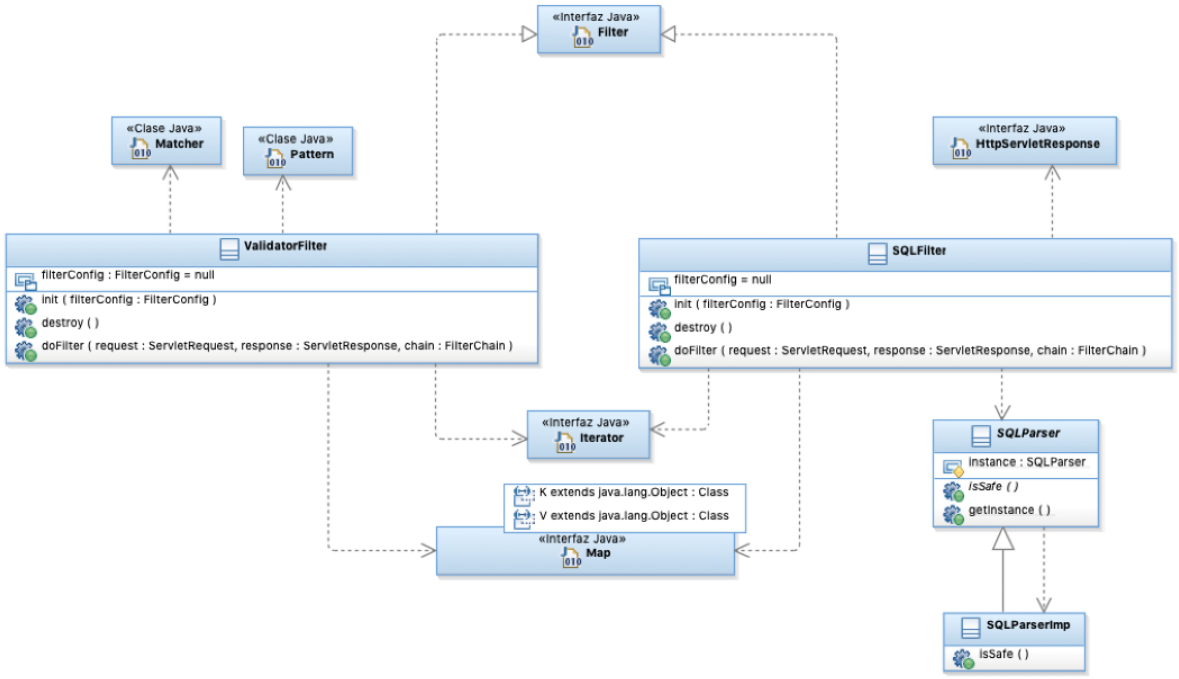

Figura 3.5 - Diagrama de clases del patrón *intercepting filter*

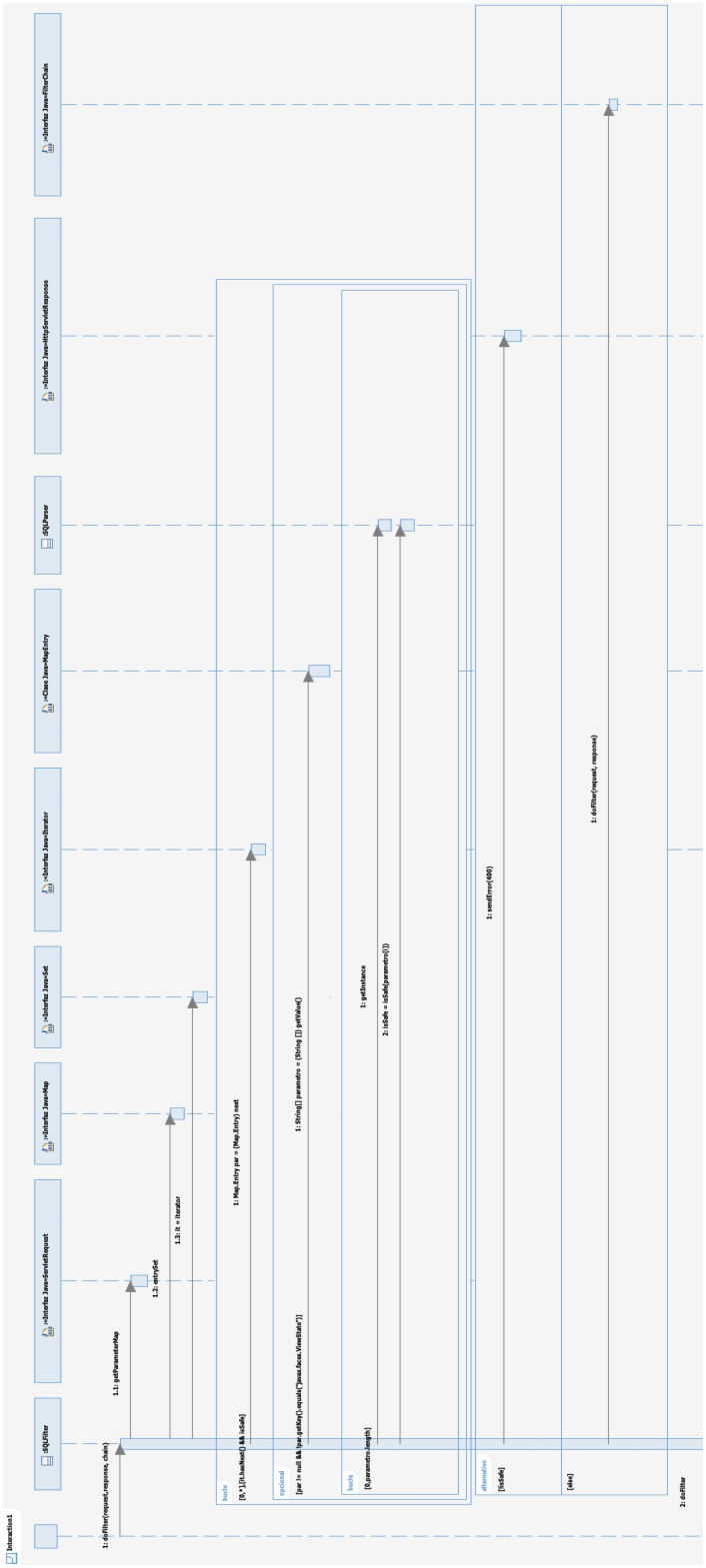

Figura 3.6 – Diagrama de secuencia del patrón *intercepting filter* para el método *doFilter* de la clase *SQLFilter*

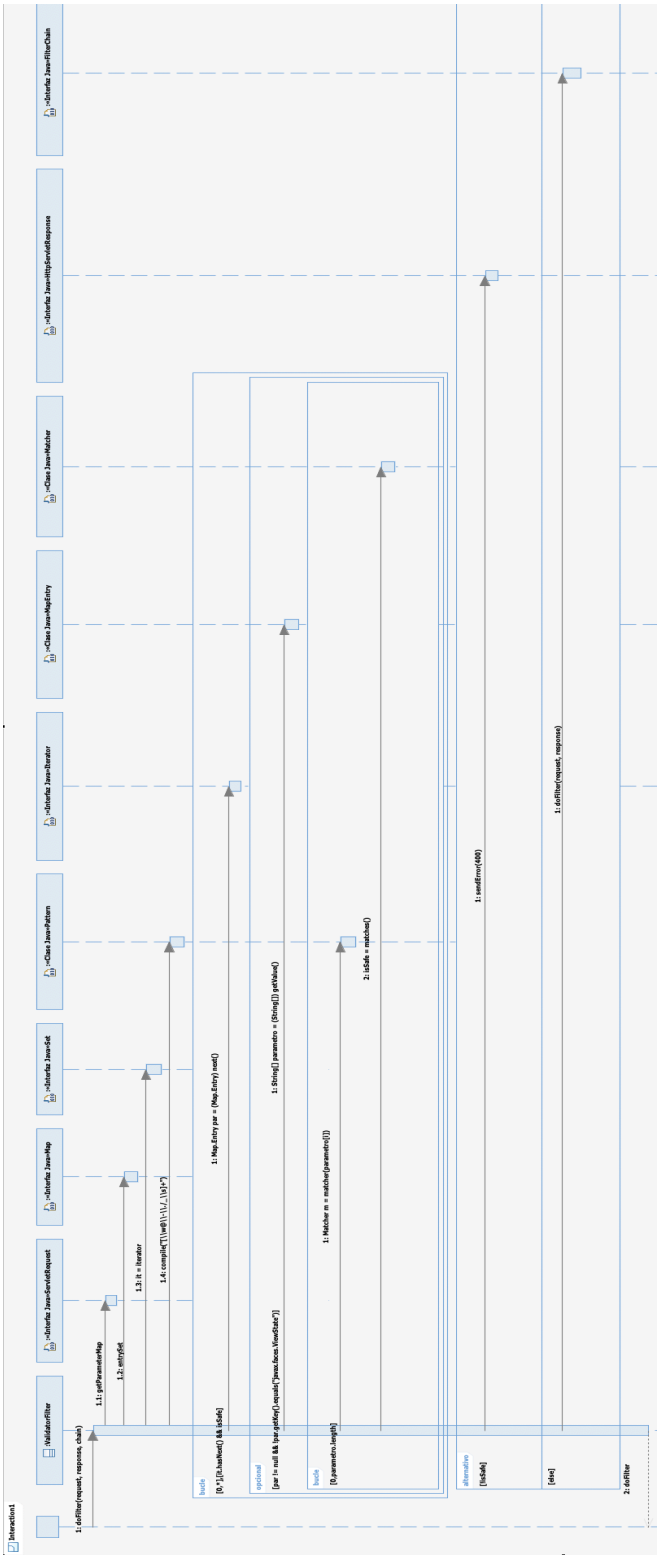

Figura 3.7 – Diagrama de secuencia del patrón *intercepting filter* para el método *doFilter* de la clase *ValidatorFilter*

Figura 3.8 – Diagrama de secuencia del patrón *intercepting filter* para el método *isSafe* de la clase *SQLParserImp*

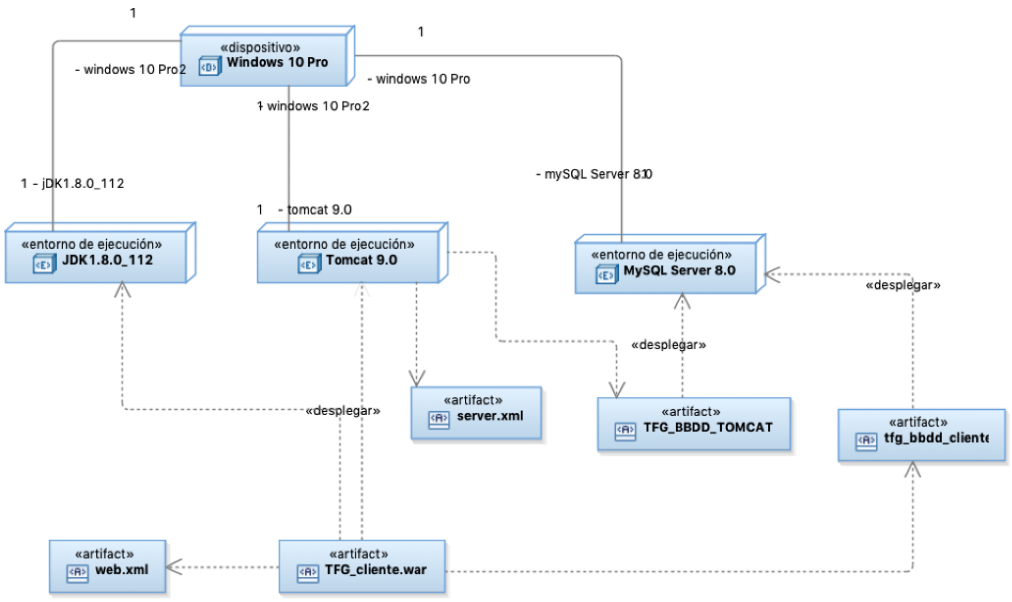

Figura 3.9 - Diagrama de despliegue del patrón *intercepting filter*

3.3. Secure Base Action

El objetivo del *secure base action* (Steel et al., 2005) es unificar la invocación de todos los componentes de seguridad, para que haya un punto de acceso central para administrar las funcionalidades de seguridad en la capa de presentación.

En este proyecto esta función la desempeñan las *managed be*ans JSF (Geary, 2006), ya que es lo más cercano a una *acción/comando* (Gamma et al., 1994) en el marco JSF. Por tanto, son el punto por donde tienen que pasar todas las invocaciones para ejecutar servicios y recibir las respuestas, por lo que es el punto donde se realizan las invocaciones relacionadas con la seguridad.

Como muchas de las actividades de seguridad que la *secure base action* llevaba a cabo en el texto del libro se han delegado en otros patrones, en la práctica, la principal acción que lleva a cabo la *secure base action* implementada es el *log* en la capa de presentación.

Las figuras 3.10, 3.11 y 3.12 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

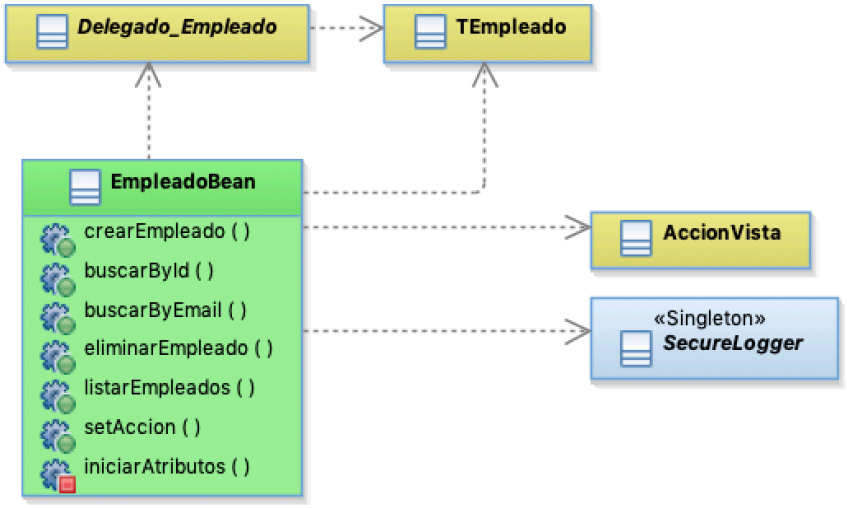

Figura 3.10 - Diagrama de clases del patrón *secure base action*

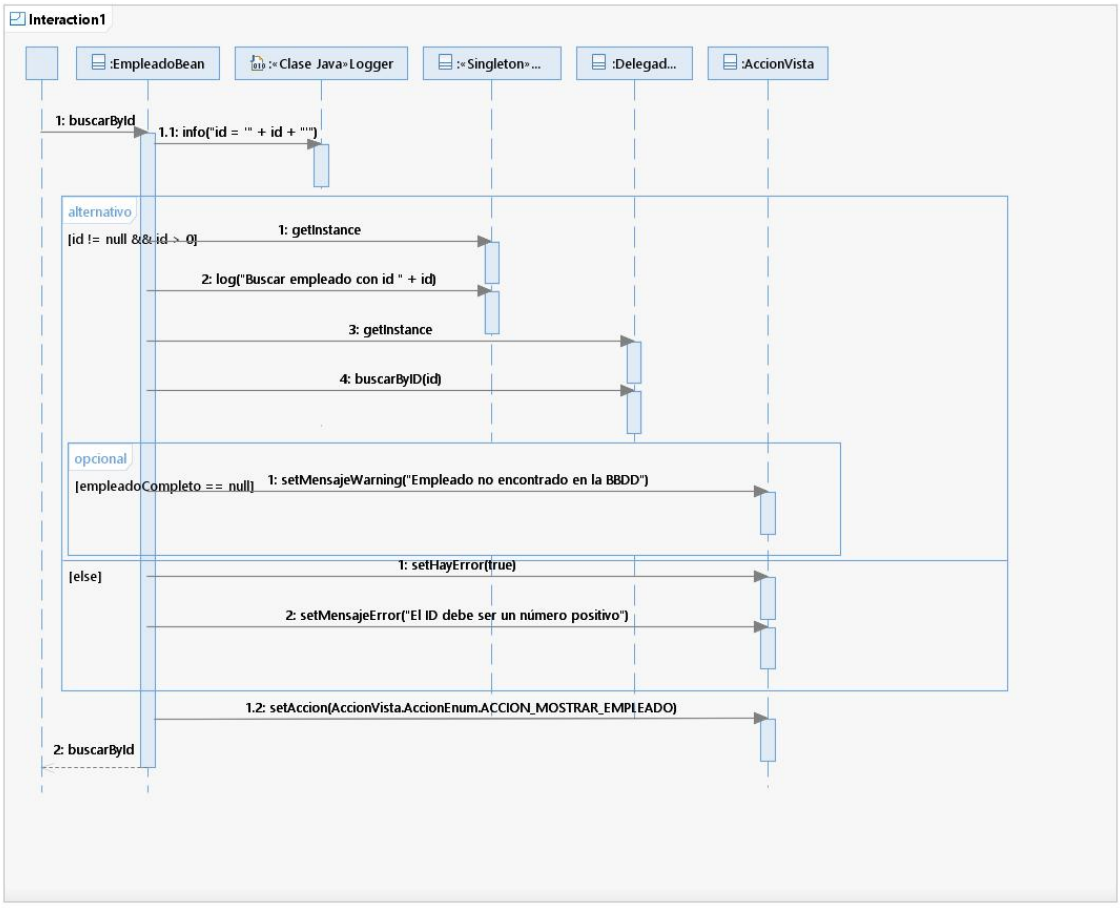

Figura 3.11 - Diagrama de secuencia del patrón *secure base action* para el método *buscarById* de la clase *EmpleadoBean*

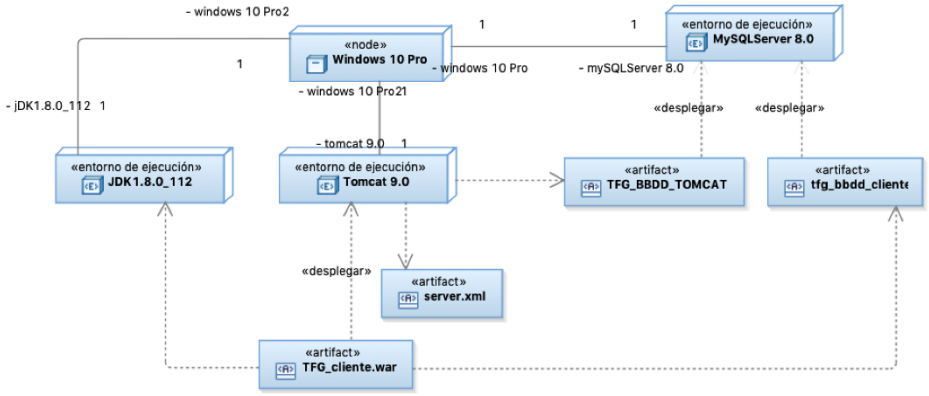

Figura 3.12 - Diagrama de despliegue del patrón *secure base action*

3.4. Secure Logger

El cometido del patrón *secure logger* (Steel et al., 2005) es registrar todas las invocaciones que se realicen en la capa de presentación.

Para ello se vale de la clase `SecureLogger`, cuyo parámetro es la operación que se ha invocado. Este a su vez llama al `LogManager`, que, con esta operación, el rol y el nombre del usuario que ha realizado la invocación crea un `SecureLog` que es persistido mediante JPA (Oracle, 2013).

La conexión que establece con la base de datos se realiza a través de SSL mediante el patrón *secure pipe* (Stee et al., 2005), asegurándose así de que los *logs* no son modificados antes de llegar a la base de datos.

Las figuras 3.13, 3.14, 3.15, 3.16 y 3.17 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

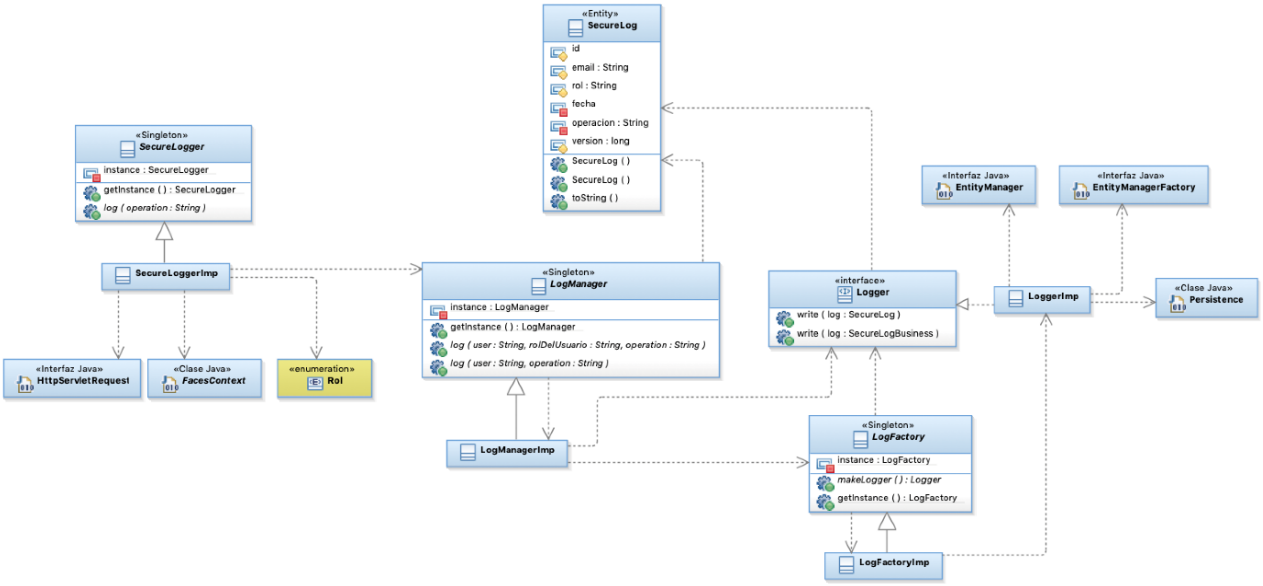

Figura 3.13 - Diagrama de clases del patrón *secure logger*

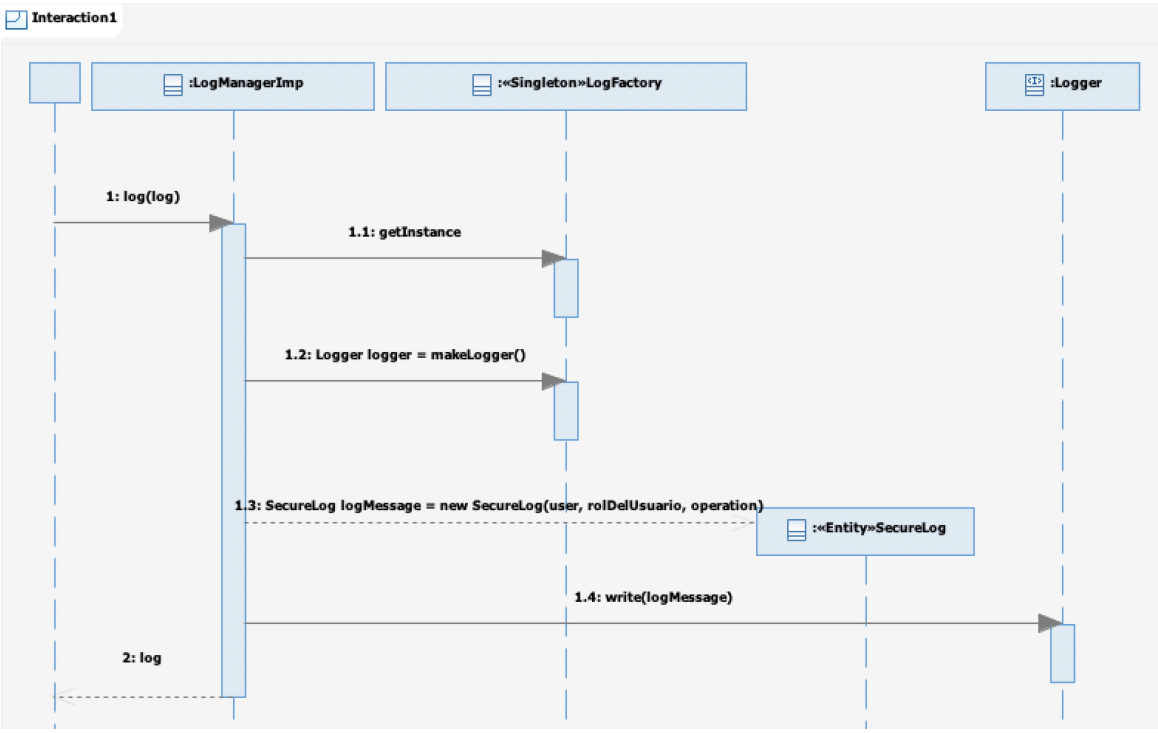

Figura 3.14 - Diagrama de secuencia del patrón *secure logger* para el método *log* de la clase *LogManagerImp*

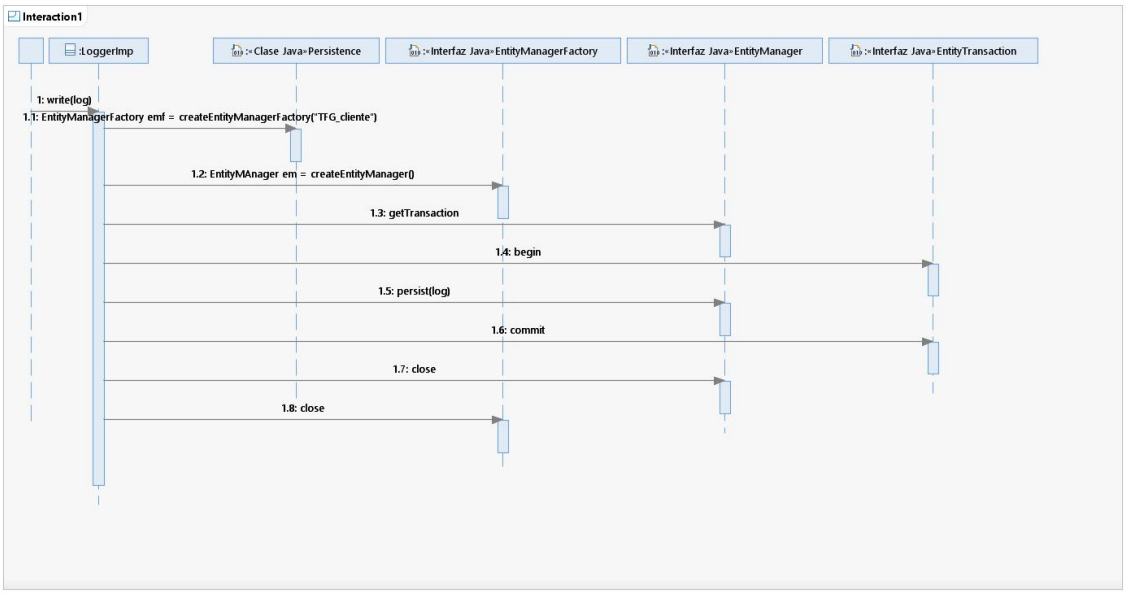

Figura 3.15 - Diagrama de secuencia del patrón *secure logger* para el método *write* de la clase *LoggerImp*

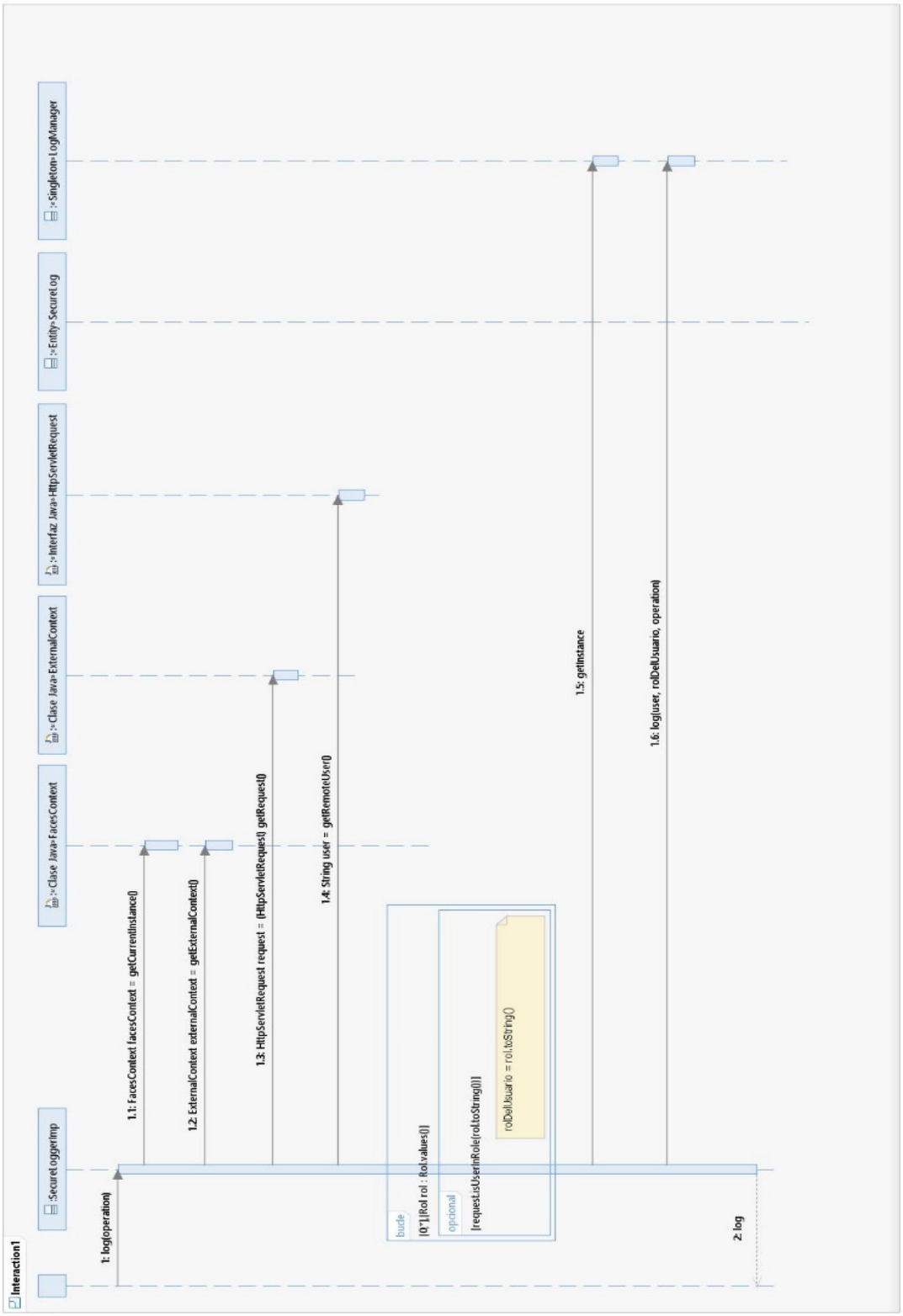

Figura 3.16 - Diagrama de secuencia del patrón *secure logger* para el método *log* de la clase *SecureLoggerImp*

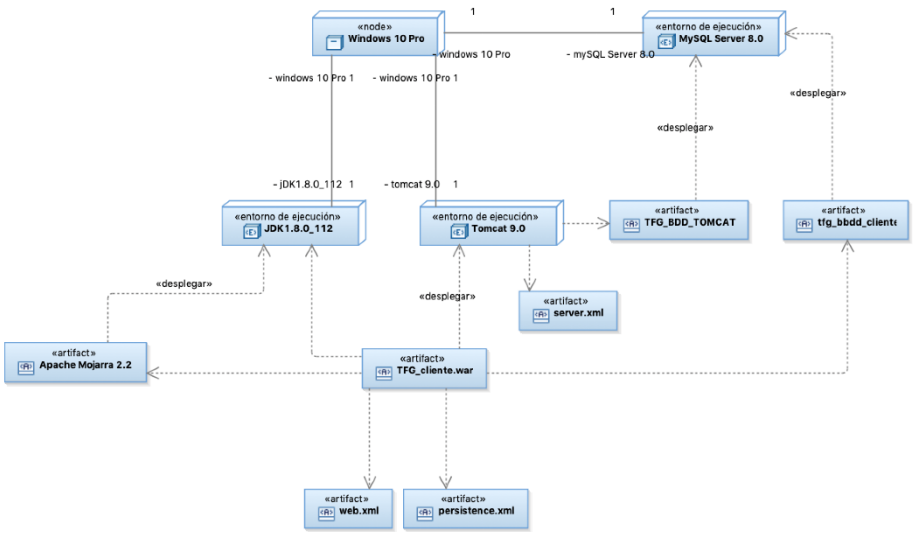

Figura 3.17 - Diagrama de despliegue del patrón *secure logger*

3.5. Secure Pipe

El patrón *secure pipe* (Steel et al., 2005) es el encargado de garantizar la integridad de los datos que se envían a través de una conexión remota. La idea fundamental tras el patrón es establecer conexiones HTTPS entre los puntos a conectar.

Secure pipe puede utilizarse para enviar datos sensibles (como usuario/contraseña) en la invocación de servicios web, o para hacer *logs* seguros en un sistema de gestión de bases de datos relacionales. En esta sección, describimos en detalle este último caso, al ser menos frecuente en la bibliografía. Se puede consultar (Java67, 2012) para apreciar las diferencias entre *truststores* y *keystores*.

Para ello en primer lugar hay que generar unos certificados X.509 de la CA (*Certificate Authority*), de servidor y de cliente mediante los siguientes comandos:

```
openssl genrsa 2048 > ca-key.pem
```

```
openssl req -new -x509 -nodes -days 3600 -key ca-key.pem -out ca.pem

openssl req -newkey rsa:2048 -days 3600 -nodes -keyout server-key.pem -out
server-req.pem

openssl rsa -in server-key.pem -out server-key.pem

openssl x509 -req -in server-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -
set_serial 01 -out server-cert.pem

openssl req -newkey rsa:2048 -days 3600 -nodes -keyout client-key.pem -out
client-req.pem

openssl rsa -in client-key.pem -out client-key.pem

openssl x509 -req -in client-req.pem -days 3600 -CA ca.pem -CAkey ca-key.pem -
set_serial 01 -out client-cert.pem
```

Con los archivos de los certificados, modificamos el archivo `my.ini` de MySQL Server y modificamos las siguientes líneas:

```
ssl-ca = "D:/Universidad/TFG/Almacenes/MySQL/ca.pem"
ssl-cert = "D:/Universidad/TFG/Almacenes/MySQL/server-cert.pem"
ssl-key = "D:/Universidad/TFG/Almacenes/MySQL/server-key.pem"
```

Una vez hecho esto, MySQL Server ya permite conexiones mediante SSL y se puede utilizar el patrón *secure pipe*. Para ello hay que añadir a un *truststore* el certificado de la CA, que ha firmado el certificado del servidor. Para esto ejecutamos:

```
keytool -importcert -alias MySQLCACert -file ca.pem -keystore
path_to_truststore_file -storepass mypassword
```

A continuación, necesitamos que Java pueda localizar el *truststore* para lo que tenemos dos métodos:

- En la *run configuration* del servidor:

```
-Djavax.net.ssl.trustStore=path_to_truststore_file

-Djavax.net.ssl.trustStorePassword=mypassword
```

- En el código:

```
System.setProperty("javax.net.ssl.trustStore","path_to_truststore_file");

System.setProperty("javax.net.ssl.trustStorePassword","mypassword");
```

Por último, añadimos a un *keystore* el certificado del cliente para que se autentique con el servidor. En primer lugar, la convertimos al formato pkcs12 y a continuación la importamos:

```
openssl pkcs12 -export -in client-cert.pem -inkey client-key.pem -name
"mysqlclient" -passout pass:changeit -out client-keystore.p12

keytool -importkeystore -srckeystore client-keystore.p12 -srcstoretype pkcs12 -
srcstorepass changeit -destkeystore ../server.keystore -deststoretype JKS -
deststorepass changeit
```

Y para que Java pueda localizar el *keystore* tenemos dos métodos:

- En la *run configuration* del servidor

```
-Djavax.net.ssl.keyStore=path_to_keystore_file

-Djavax.net.ssl.keyStorePassword=mypassword
```

- En el código:

```
System.setProperty("javax.net.ssl.keyStore","path_to_keystore_file");
System.setProperty("javax.net.ssl.keyStorePassword","mypassword");
```

La figura 3.18 incluye el diagrama de despliegue, que describe la implementación del patrón en el contexto de una aplicación multicapa J2EE.

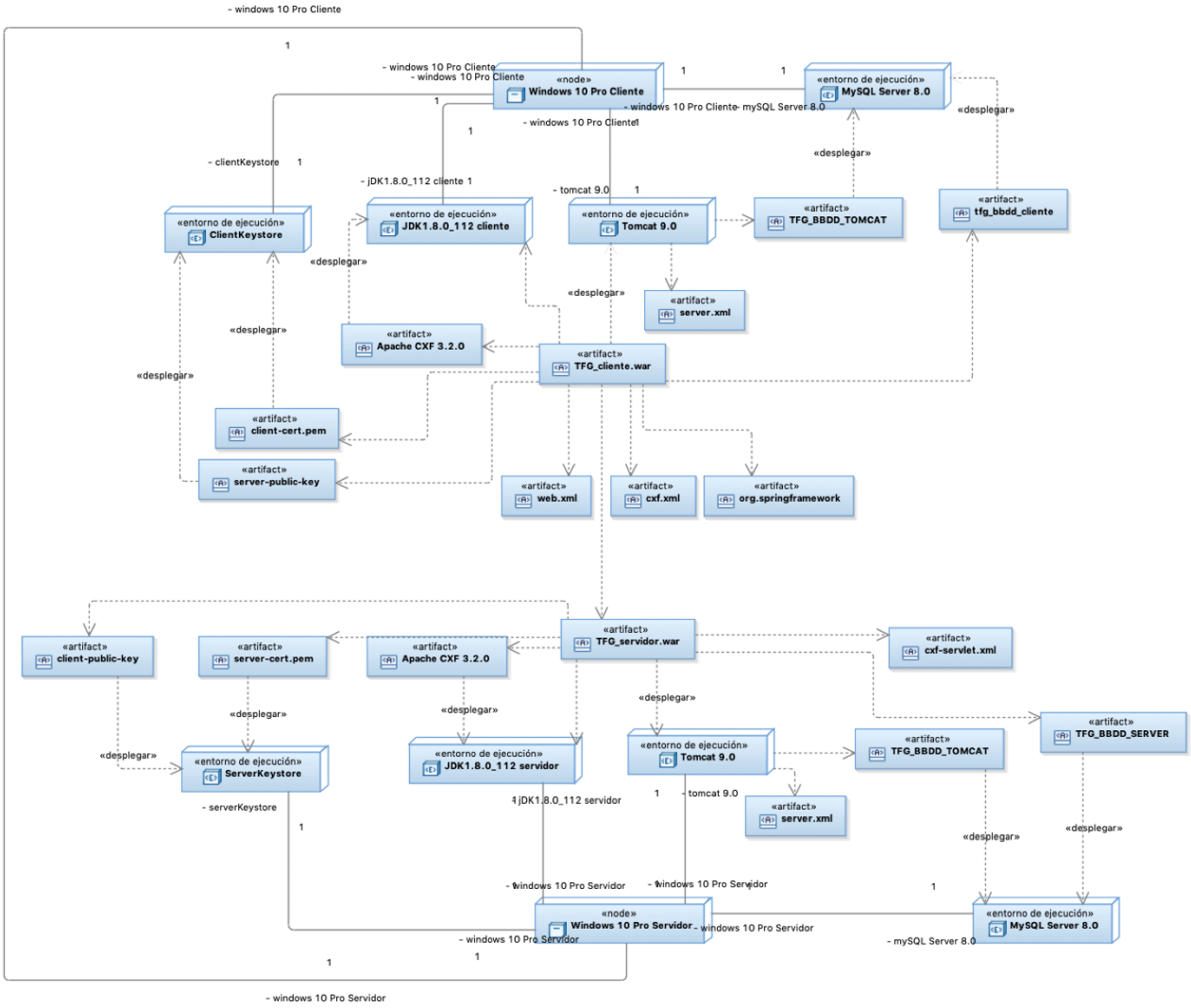

Figura 3.18 - Diagrama de despliegue del patrón *secure pipe*

3.6. Secure Service Proxy

El objetivo del patrón *secure service proxy* (SSP) (Steel et al., 2005) es la reutilización de servicios mediante la integración de nuevos protocolos en una aplicación existente sin modificar los servicios originales.

Para ello, en este proyecto se ha realizado la invocación de un servicio SOAP que sirve como SSP para ejecutar un servicio REST ya existente añadiéndole todas las posibilidades

del protocolo SOAP. Además, abstrae al usuario del tipo de servicio que está usando, ya que este cambio es transparente para él.

Aunque en este proyecto se ha implementado el patrón desde cero, cabe destacar la existencia de diferentes herramientas desarrolladas para este propósito (Da Silva, 2013). Hay dos enfoques principales:

- *Service Gateways*: son interceptores de mensaje software (Steel, 2005) que han evolucionado asumiendo el papel desempeñado por los buses de servicios empresariales (ESB) (Ryan, 2011). Algunos son Vordel Policy Studio (Vordel, 2011) y SOA Gateway (System, 2008).

- *Enterprise Service Bus* (ESB) (Ryan, 2011): son sistemas intermediarios que traducen el mensaje recibido y lo envían al consumidor correcto. Algunos son Oracle Service Bus (Oracle, 2011), IBM WebSphere Enterprise Service Bus o Apache ServiceMix (Apache, 2011).

Las figuras 3.19, 3.20, 3.21, 3.22 y 3.23 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

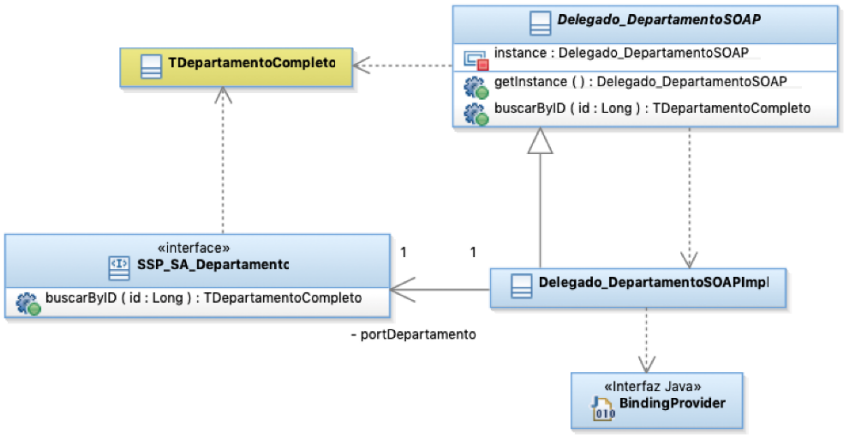

Figura 3.19 - Diagrama de clases de la aplicación cliente del patrón *secure service proxy* en el WSC

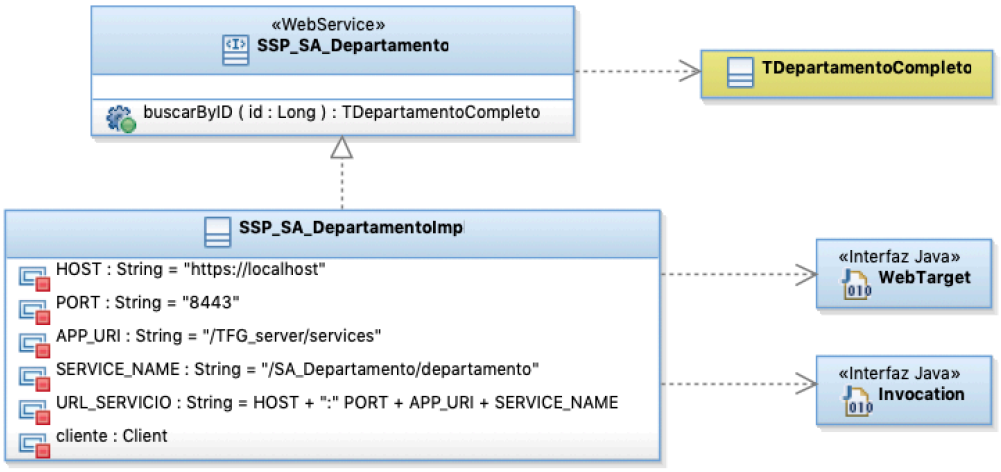

Figura 3.20 - Diagrama de clases de la aplicación servidor del patrón *secure service proxy* en el WSP

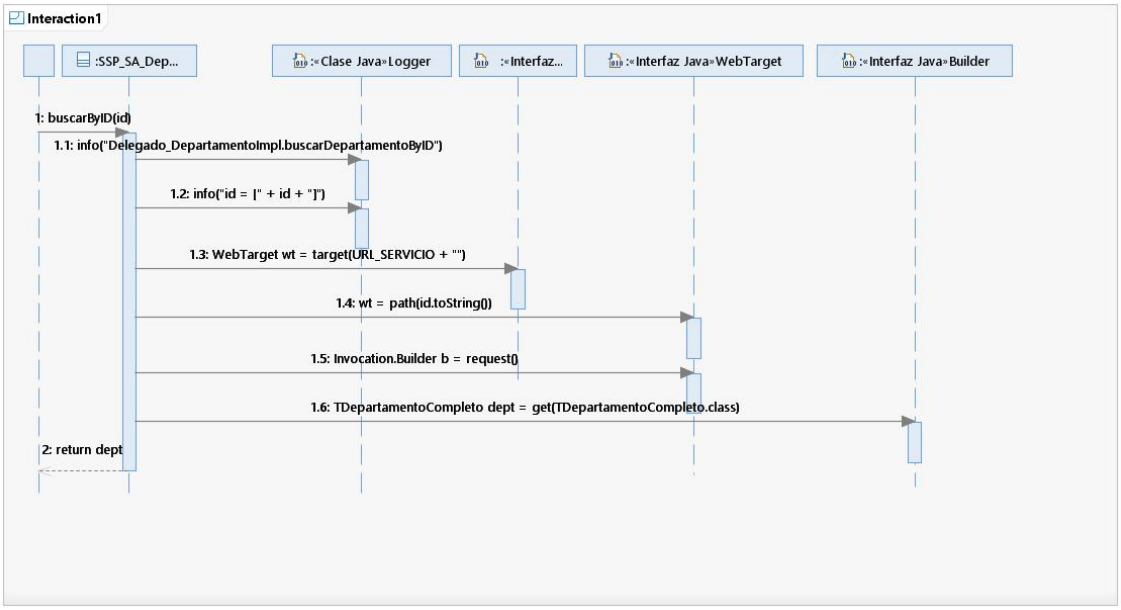

Figura 3.21 - Diagrama de secuencia del patrón *secure service proxy* para el método *buscarById* de la clase *SSP_SA_DepartamentoImpl*

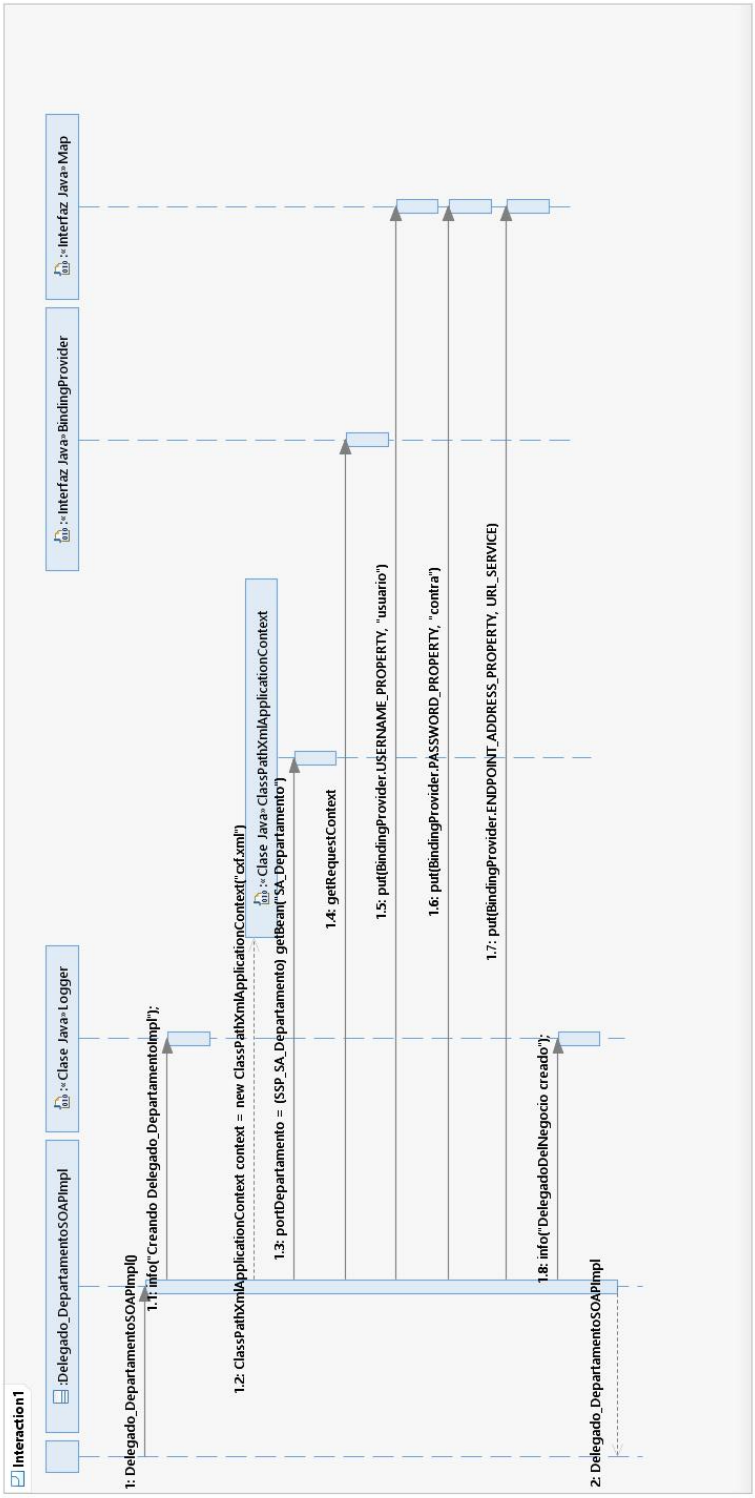

Figura 3.22 - Diagrama de secuencia del patrón *secure service proxy* para el constructor de la clase *Delegado_DepartamentoSOAPImpl*

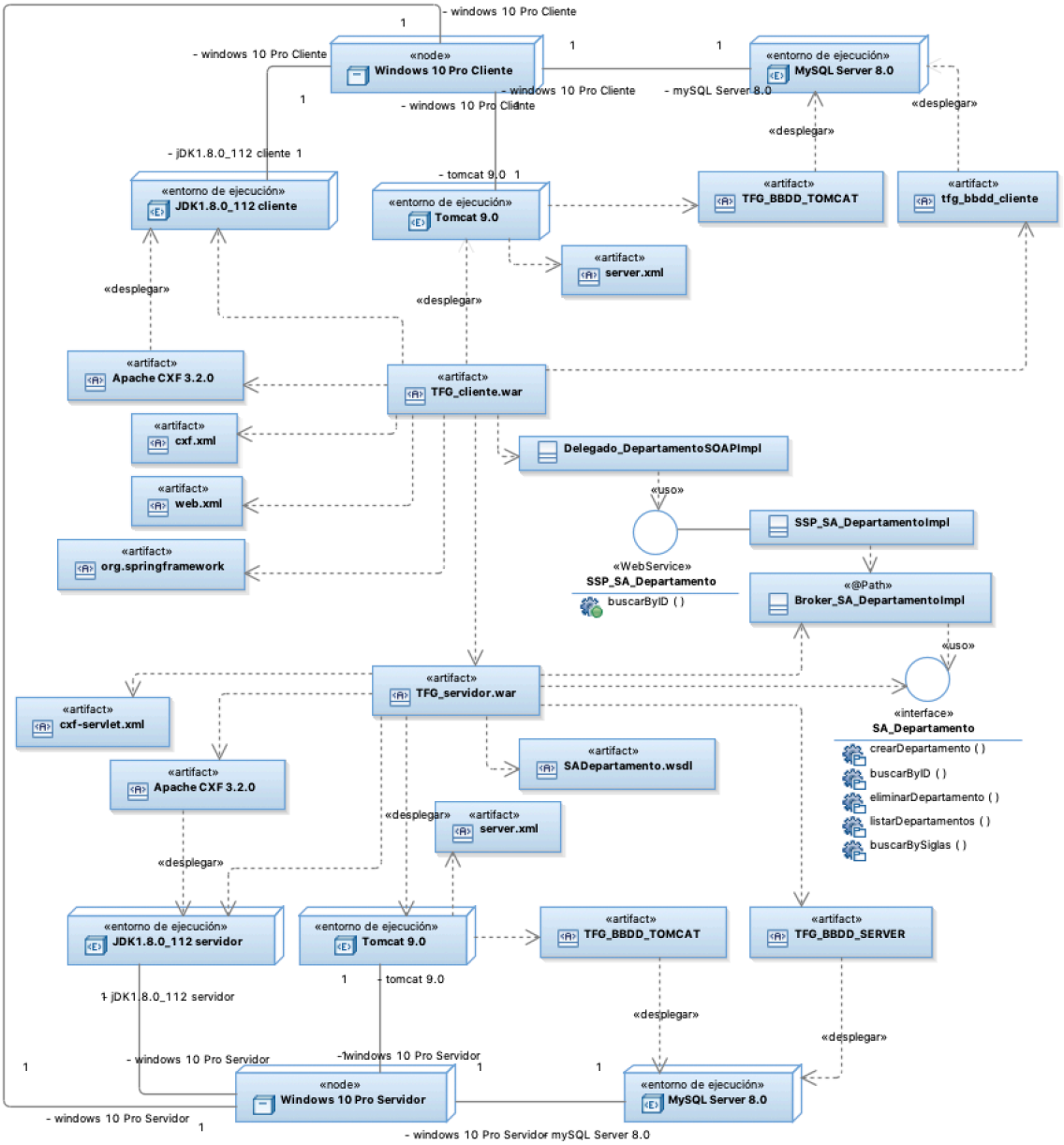

Figura 3.23 - Diagrama de despliegue de la aplicación cliente del patrón *secure service proxy*

3.7. Intercepting Web Agent

El cometido del *intercepting web agent* (Steel et al., 2005) es proveer de autenticación y autorización externa a la aplicación mediante la interceptación de peticiones para no tener que modificar una aplicación ya implementada, lo cual sería muy costoso y difícil de realizar.

La implementación de este patrón se ha en Apache Tomcat ya que utiliza sus interceptores para realizar la autenticación sobre HTTPS antes de realizar cada una de las invocaciones a los servicios web. En el proyecto se ha implementado el patrón tanto para WSP SOAP como REST.

Las figuras 3.24, 3.25, 3.26, 3.27, 3.28, 3.29 y 3.30 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

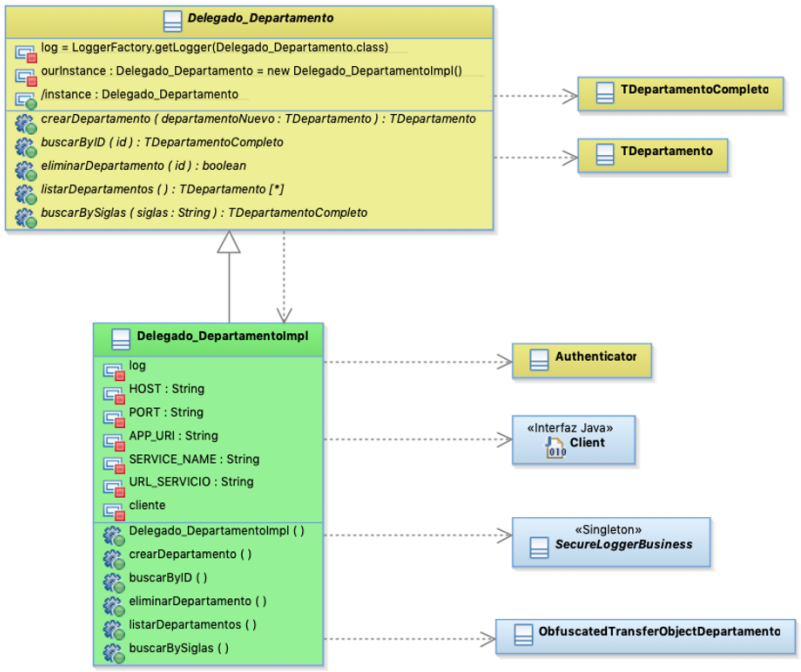

Figura 3.24 - Diagrama de clases de un delegado del negocio de un servicio web REST en el WSC

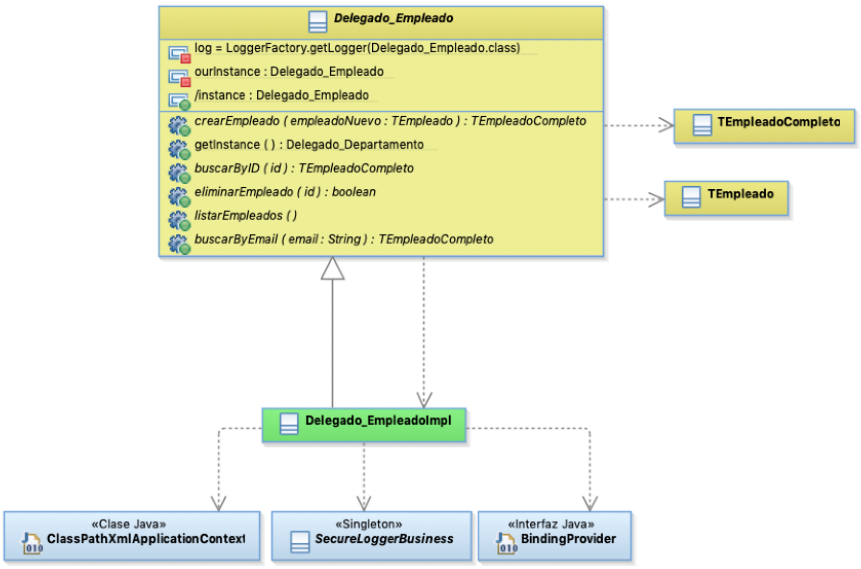

Figura 3.25 - Diagrama de clases de un delegado del negocio de un servicio web SOAP en el WSC

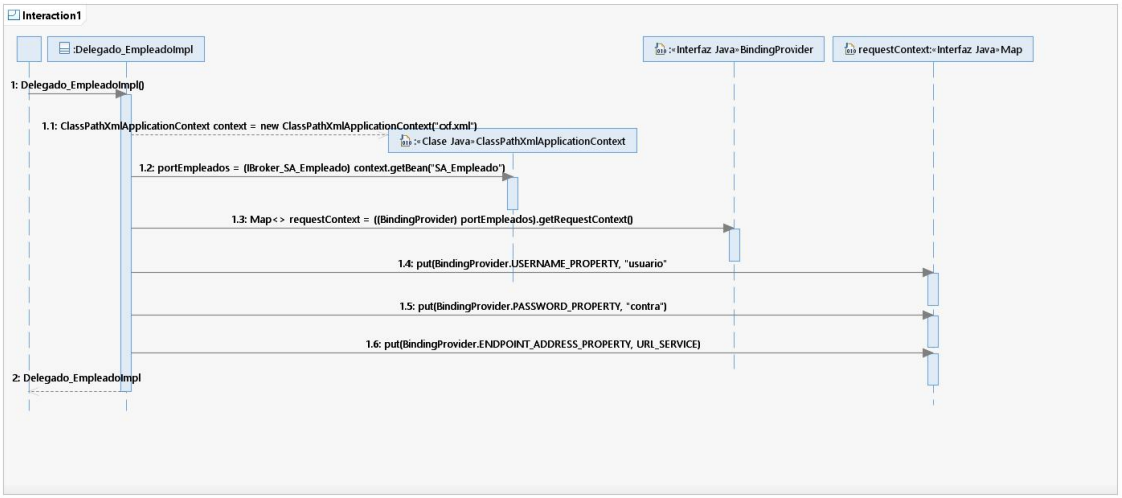

Figura 3.26 - Diagrama de secuencia del patrón *intercepting web agent* para el constructor de clase *DelegadoEmpleadoImpl (SOAP)*

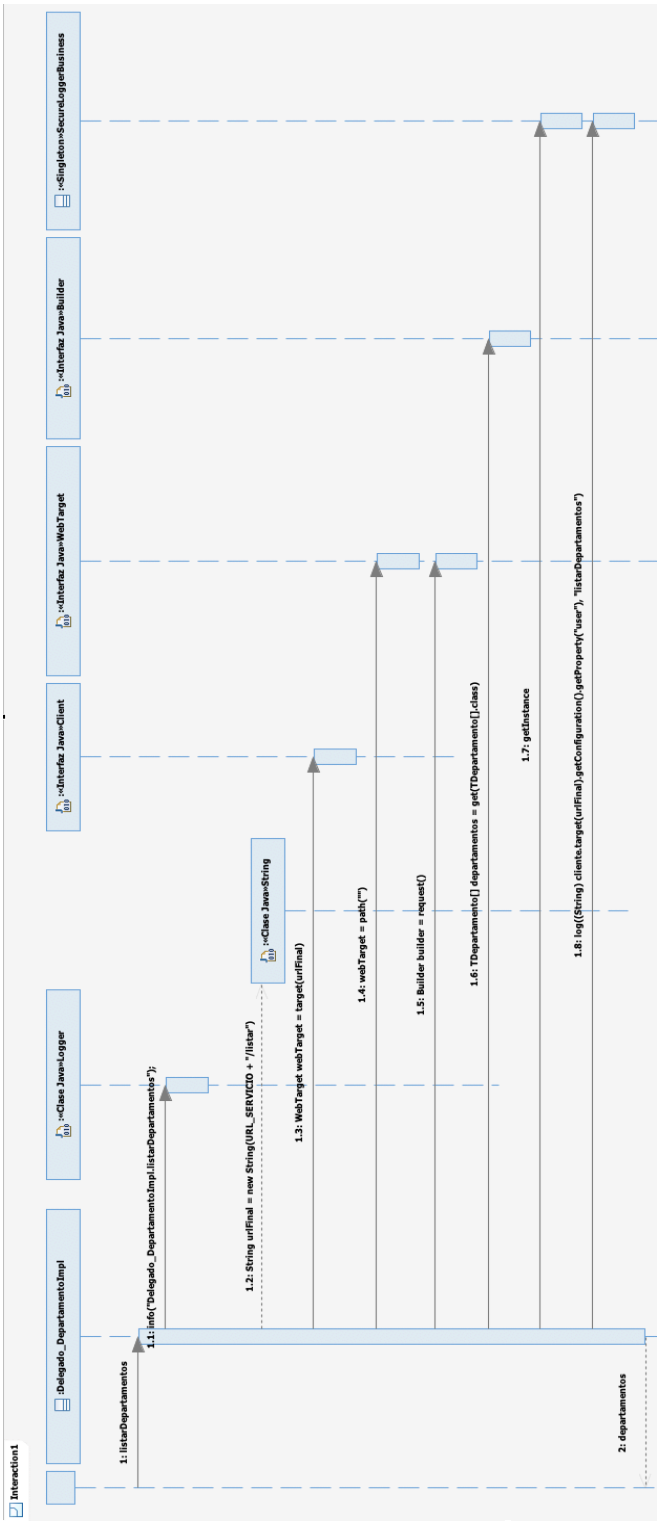

Figura 3.27 - Diagrama de secuencia del patrón *intercepting web agent* para el método *listarDepartamentos* de la clase *DelegadoNegocioDepartamentoImpl (REST)*

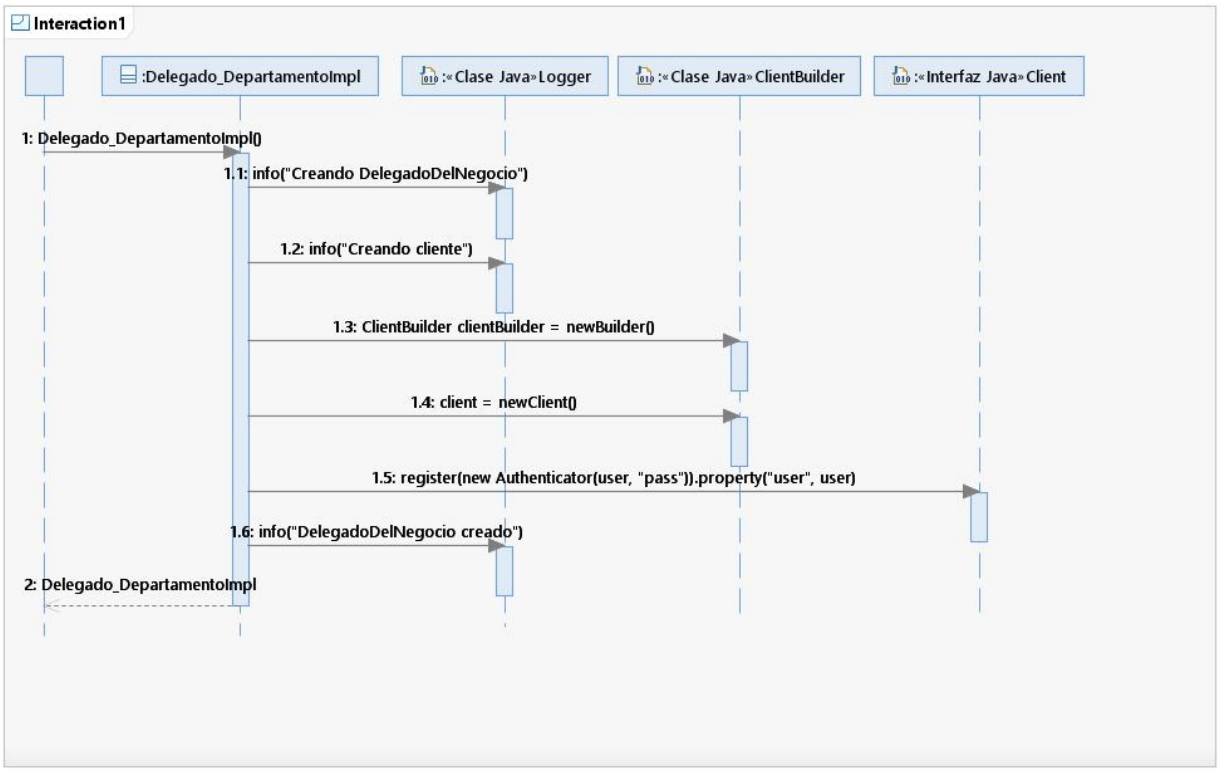

Figura 3.28 - Diagrama de secuencia del patrón *intercepting web agent* para el constructor de *DelegadoDepartamentoImpl (REST)*

Figura 3.29 - Diagrama de secuencia del patrón *intercepting web agent* para el método *crearEmpleado* de la clase *DelegadoEmpleadoImpl (SOAP)*

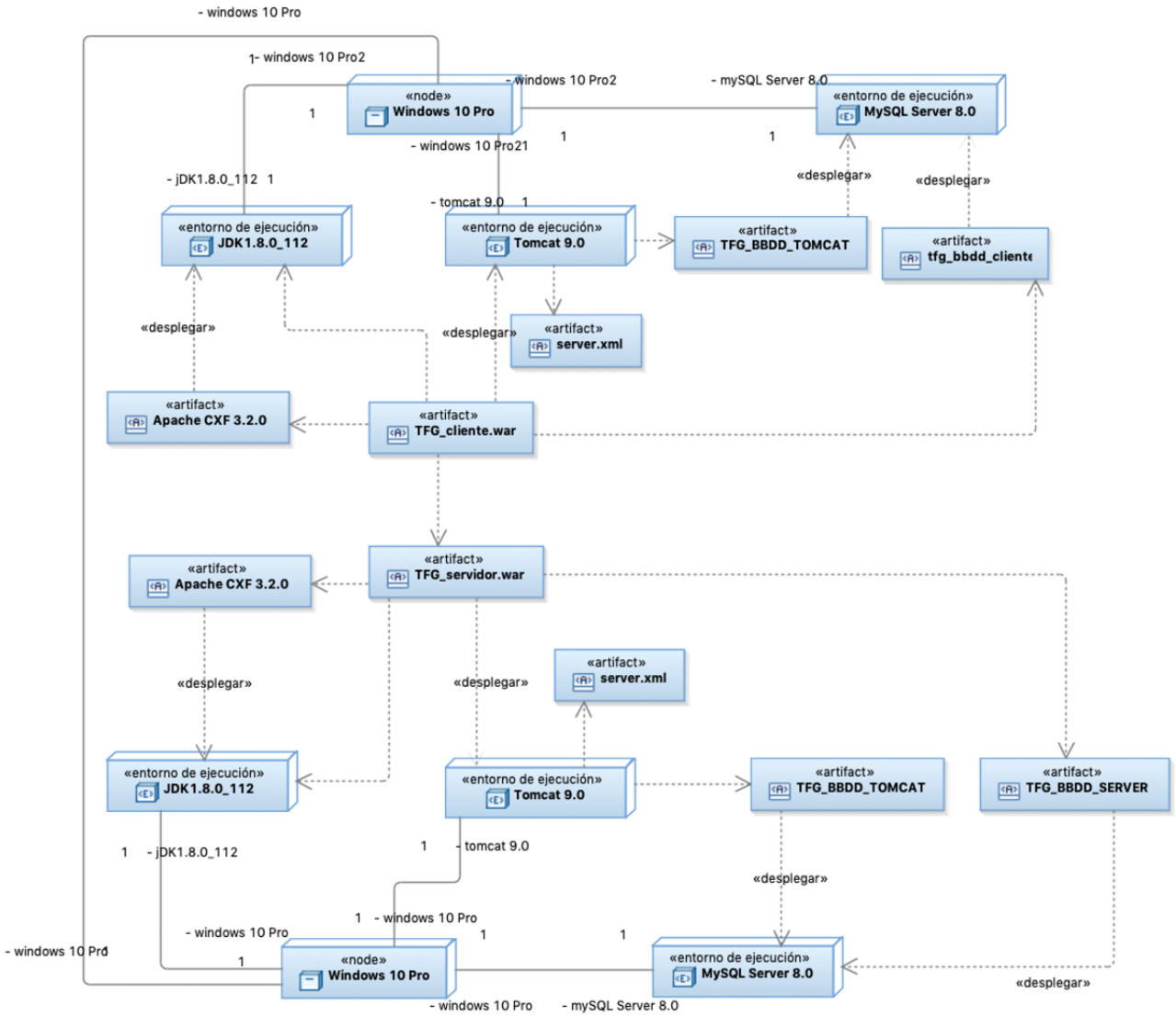

Figura 3.30 - Diagrama de despliegue del patrón *intercepting web agent*

4. Patrones Business-Tier

4.1. Audit Interceptor

El patrón *audit interceptor* (Steel et al., 2005) se encarga de centralizar la funcionalidad de la auditoría de las peticiones y respuestas de la capa de negocio. Centralizando la funcionalidad de la auditoría se elimina la carga de su implementación a los desarrolladores de las componentes de negocio de la aplicación. Por tanto, se reduce la duplicación de código y se favorece su reutilización.

A menudo es necesario auditar en la capa de servicios además de en la capa de negocio. Por esta razón, en este trabajo se ha implementado como un *message inspector* que hace *logs* de la IP, del usuario y la contraseña de la petición y del servicio invocado. Para tal fin, el *audit interceptor* ha sido implementado en este trabajo como un `AbstractPhaseInterceptor` Apache CXF, es decir, como un interceptor Apache CXF.

Las figuras 4.1, 4.2 y 4.3 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

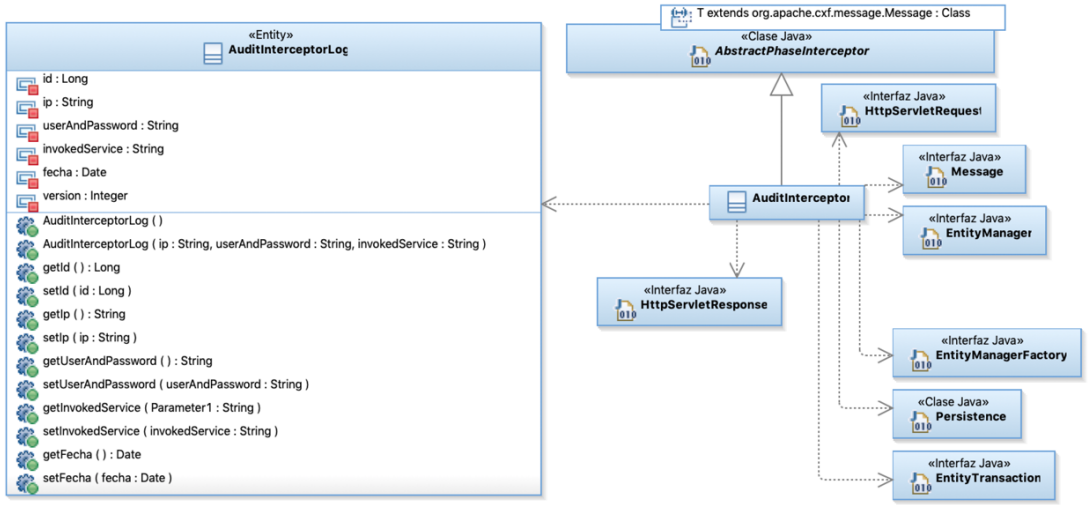

Figura 4.1 - Diagrama de clases del patrón *audit interceptor*

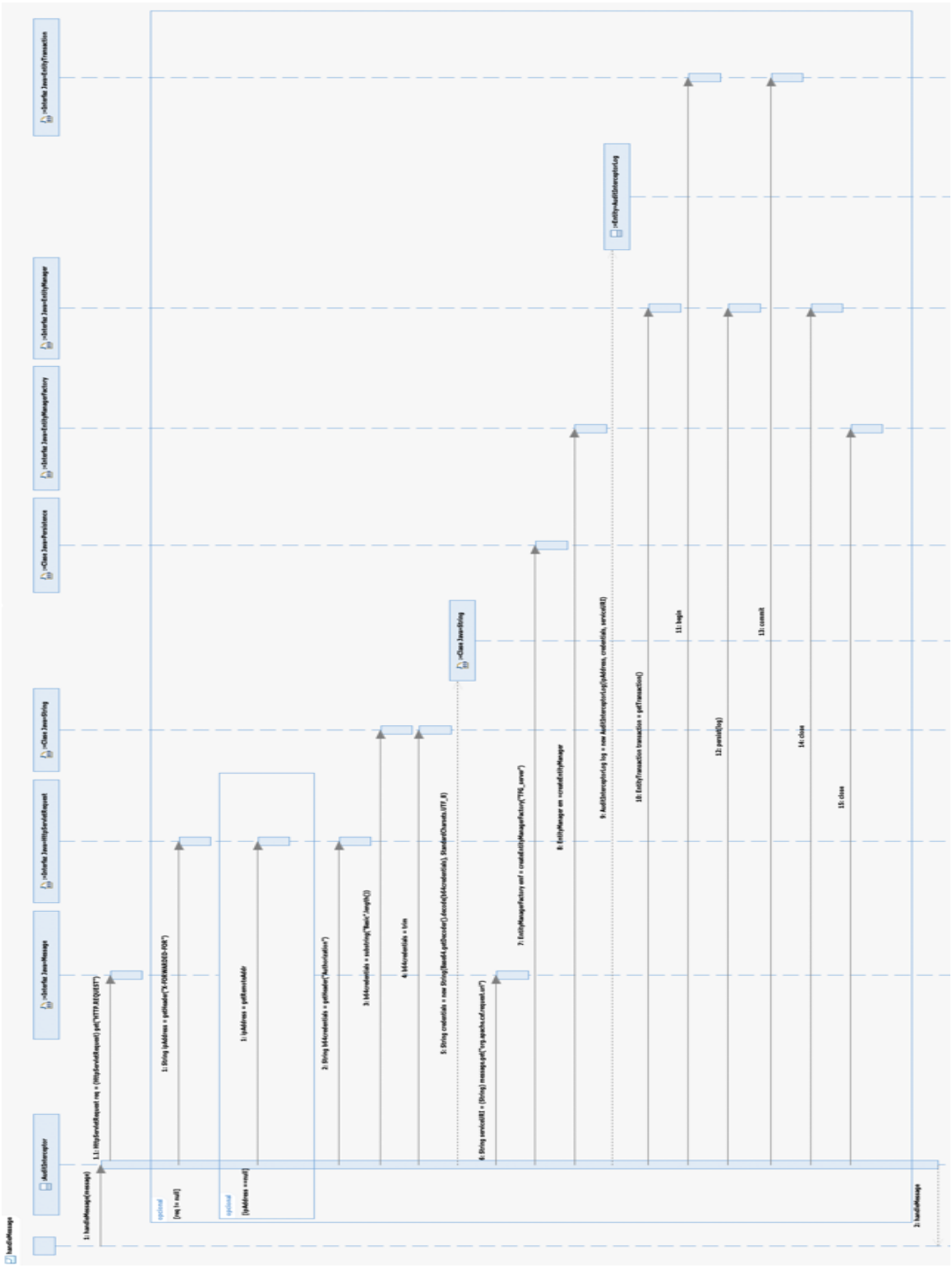

Figura 4.2 - Diagrama de secuencia del patrón *audit interceptor* para el método *handleMessage* de la clase *AuditInterceptor*

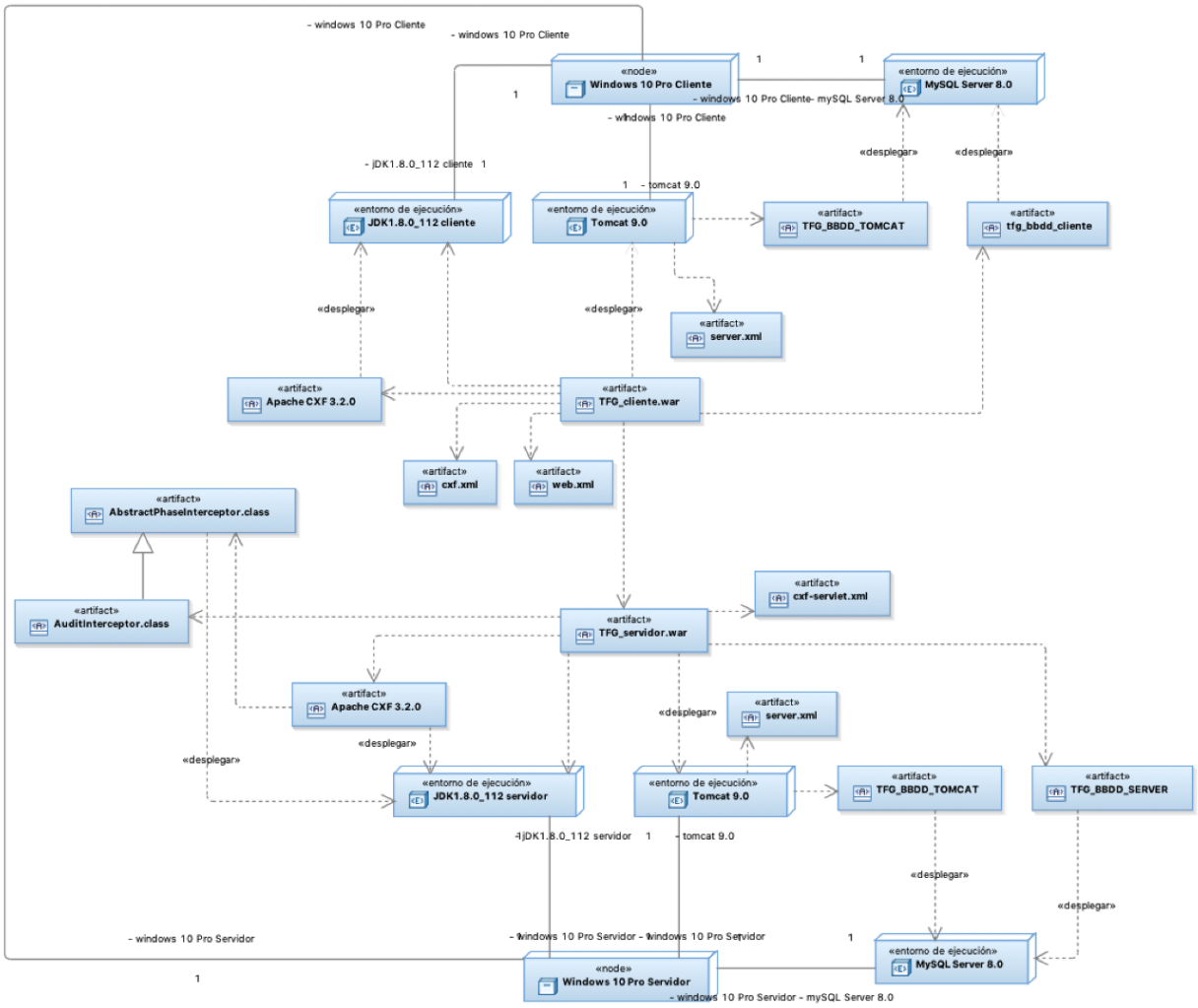

Figura 4.3 - Diagrama de despliegue del patrón *audit interceptor*

4.2. Container Managed Security

El patrón *container managed security* (Steel et al., 2005) define roles a nivel de aplicación en tiempo de desarrollo y mapea usuarios a roles en tiempo de despliegue o tras el despliegue. Este patrón es una manera simple y estandarizada de implementar autenticación y autorización en aplicaciones J2EE. Usando este patrón el contenedor, en nuestro caso Apache Tomcat, realiza la autenticación y autorización del usuario, quitando la carga al desarrollador de introducir las directivas de seguridad en el código.

La implementación de este patrón está delegada en Apache Tomcat mediante el *realm JDBC*. De esta forma, el usuario y contraseña recibidos para invocar un WSP son automáticamente contrastados por Tomcat contra una base de datos relacional. Para utilizarlo hay que modificar el fichero `server.xml` añadiendo lo siguiente:

```
<Realm className=”org.apache.catalina.realm.JDBCRealm”
      connectionURL=”jdbc:mysql://localhost/BBDD_NAME?user=bbdd_user;password=
bbdd_password”  driverName=com.mysql.jdbc.Driver” roleNameCol=”role_name”
      useSSL=”true” userCredCol=”user_pass” userNameCol=”user_name”
      userRoleTable=”user_roles” userTable=”users” />
```

Donde *BBDD_NAME* es el nombre de la base de datos que contiene los usuarios y roles, *bbdd_user* es el nombre del usuario de la base de datos y *bbdd_password* es la contraseña de acceso a la base de datos.

La figura 4.4 incluye el diagrama de despliegue que describe la implementación del patrón en el contexto de una aplicación multicapa J2EE.

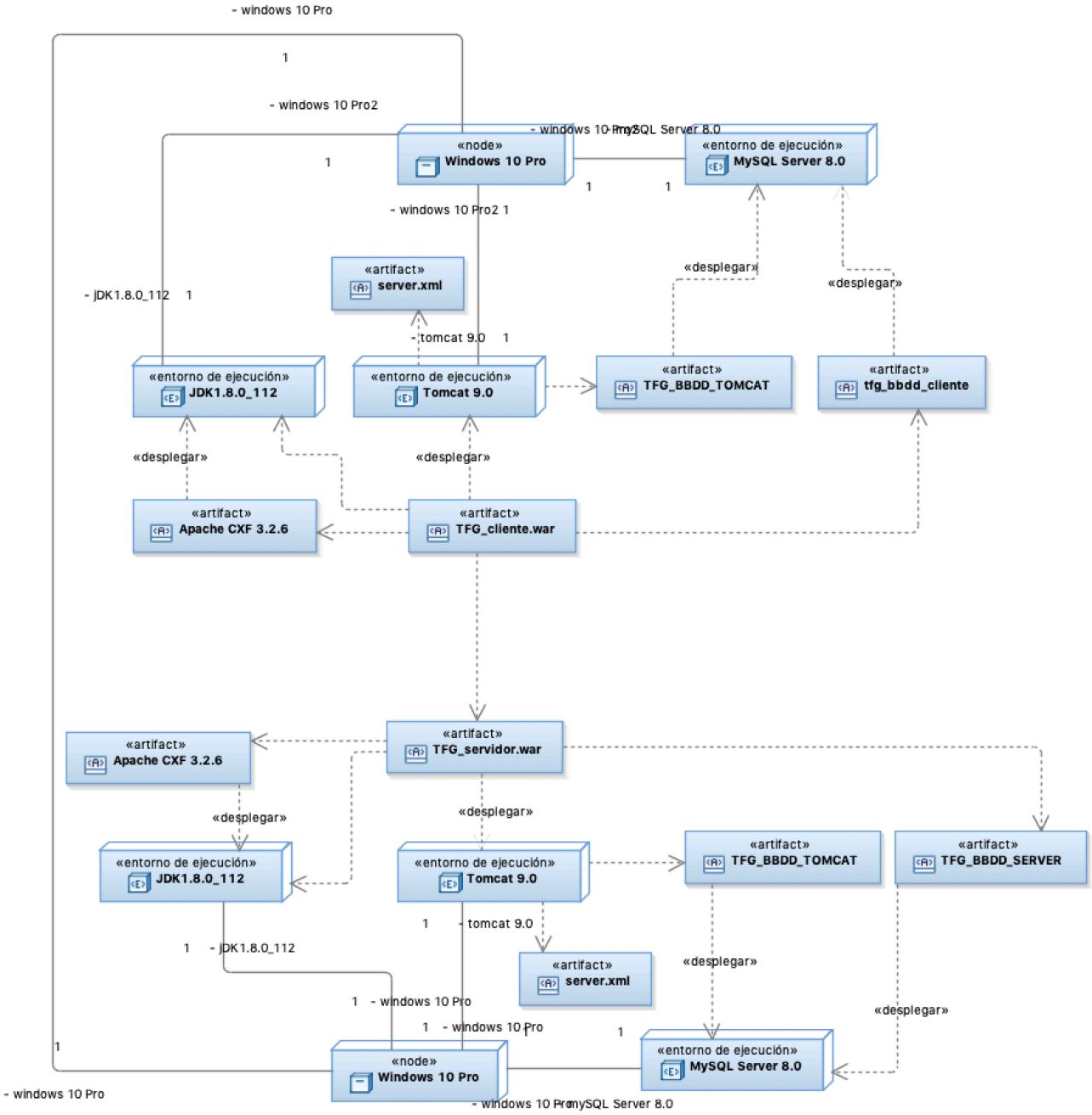

Figura 4.4 - Diagrama de despliegue del patrón *container managed security*

4.3. Dynamic Service Management

El patrón *dynamic service management* (Steel et al., 2005) permite la instrumentación dinámica de componentes de grano fino para gestionar y monitorizar la aplicación al nivel de detalle necesario. El uso de una herramienta de monitorización es un importante aspecto en la seguridad de una aplicación, pues permite a los administradores prevenir intrusiones o ataques maliciosos proactivamente.

El desarrollo de una herramienta de monitorización es muy complejo y requeriría mucho tiempo de desarrollo, por tanto, en el trabajo hemos usado la tecnología *Java Management Extension* (*JMX*) (Oracle, 2005). Para poder utilizar *JMX* es necesario modificar la *run configuration* del servidor añadiendo lo siguiente:

```
-Dcom.sun.management.jmxremote.port=8008
-Dcom.sun.management.jmxremote.authenticate=true
-Dcom.sun.management.jmxremote.ssl=true
-Dcom.sun.management.jmxremote.ssl.need.client.auth=false
-Dcom.sun.management.jmxremote.password.file=/ruta/jmxremote.password
```

El archivo `jmxremote.password` es una copia del archivo `jmxremote.password.template` que incluye los roles presentes en el directorio `jre/lib/management` del *JDK*. No es necesaria la autenticación del cliente porque están configurados los *keystores* y se iniciará el servicio de monitorización indicando la *truststore.* Una vez configurado sólo es necesario ejecutar el comando `jconsole` desde un terminal en el que el *JDK* esté en el *path* indicando la *truststore*:

```
jconsole localhost:8008 -J-Djavax.net.ssl.trustStore=”/ruta/server.keystore” -
J-Djavax.net.ssl.trustStorePassword=”password”
```

En este trabajo se ha decidido monitorizar los servicios de aplicación de las entidades departamento, empleado y proyecto. En particular, los métodos *buscar* de cada servicio de aplicación. De esta manera se tiene acceso al número de veces que se ha consultado una entidad y cuál ha sido la última entidad consultada.

Las figuras 4.5, 4.6, 4.7 y 4.8 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

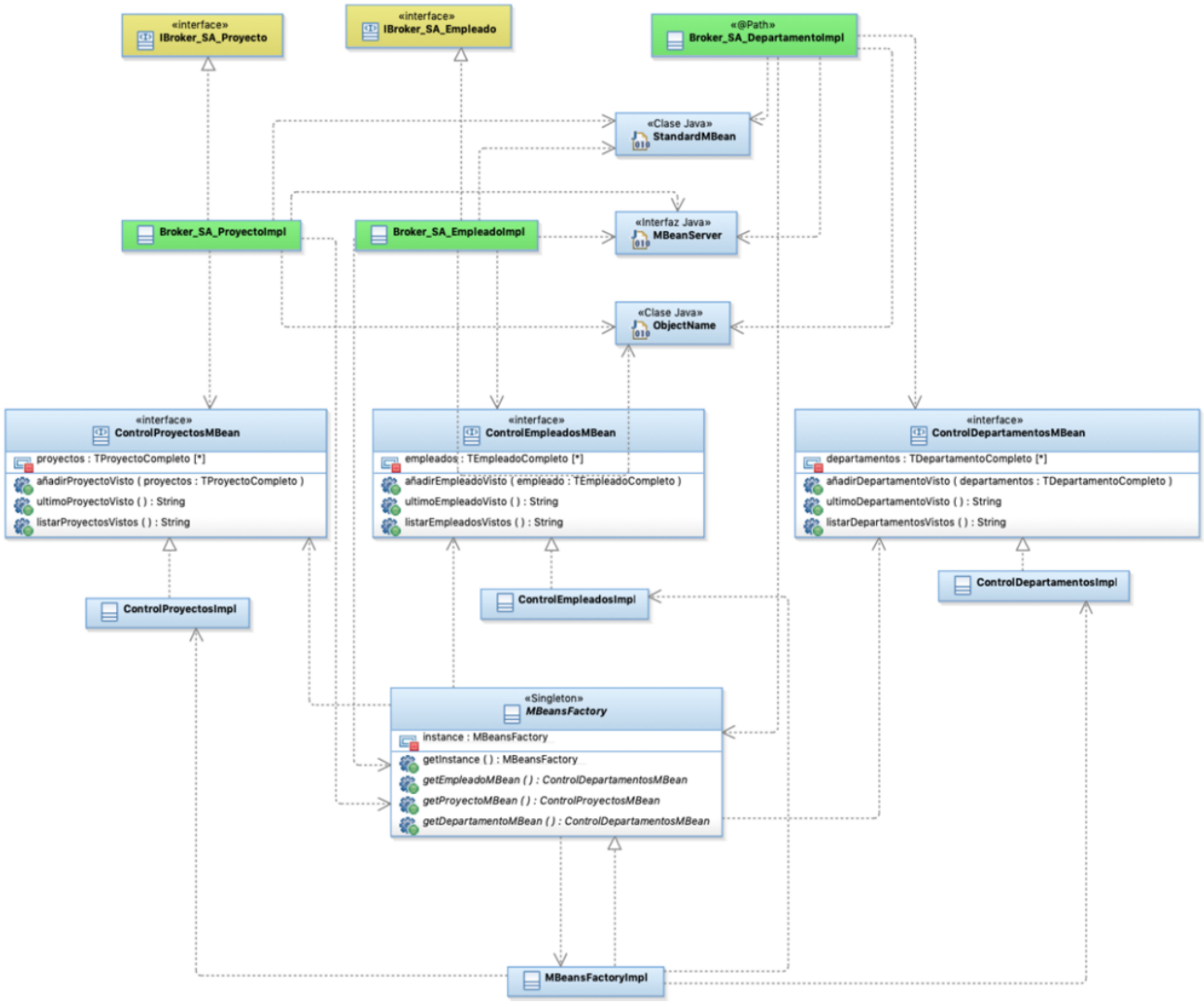

Figura 4.5 - Diagrama de clases del patrón *dynamic service management*

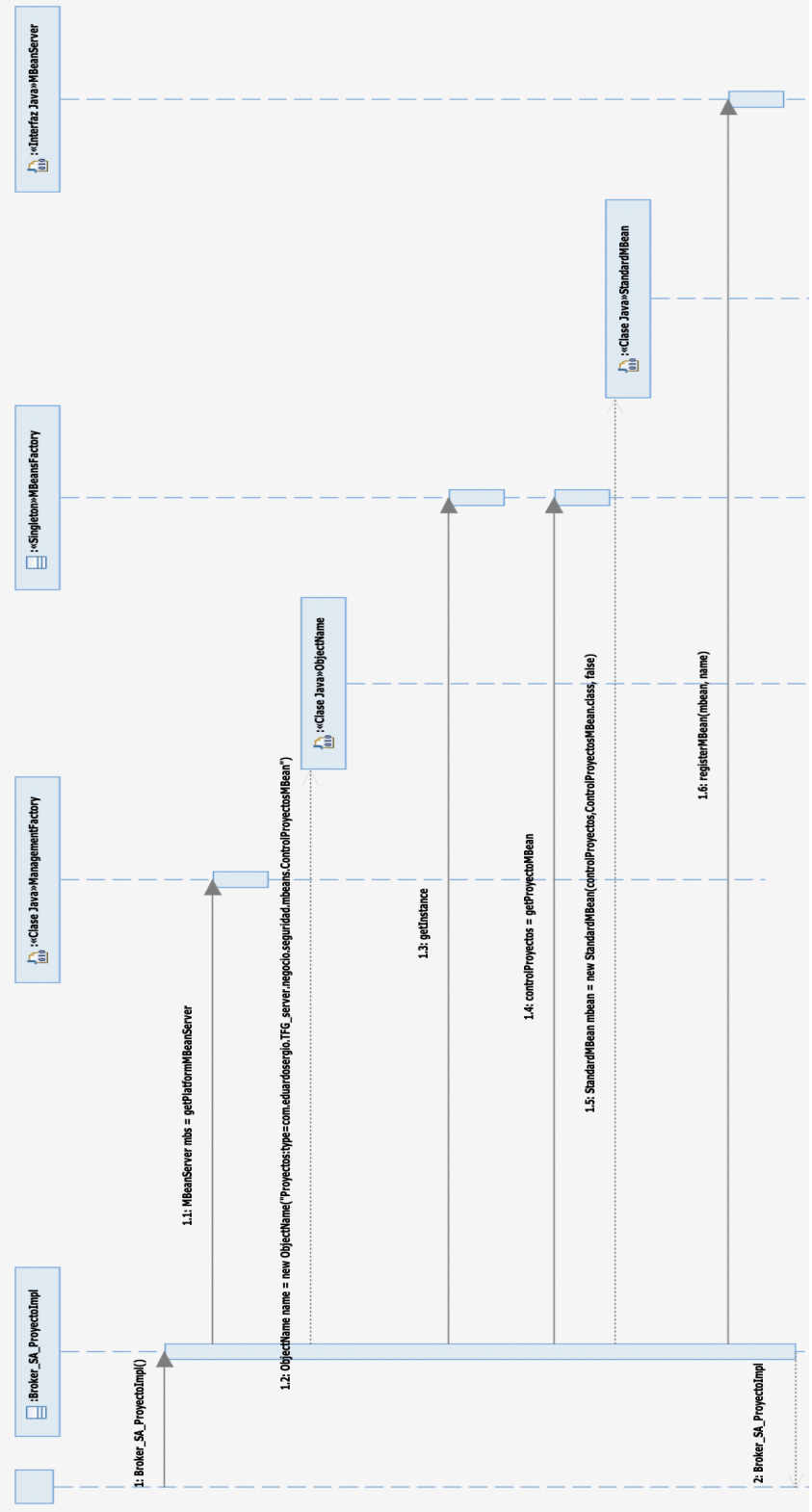

Figura 4.6 - Diagrama de secuencia del patrón *dynamic service management* para la constructora de la clase *Broker_SA_ProyectoImpl*

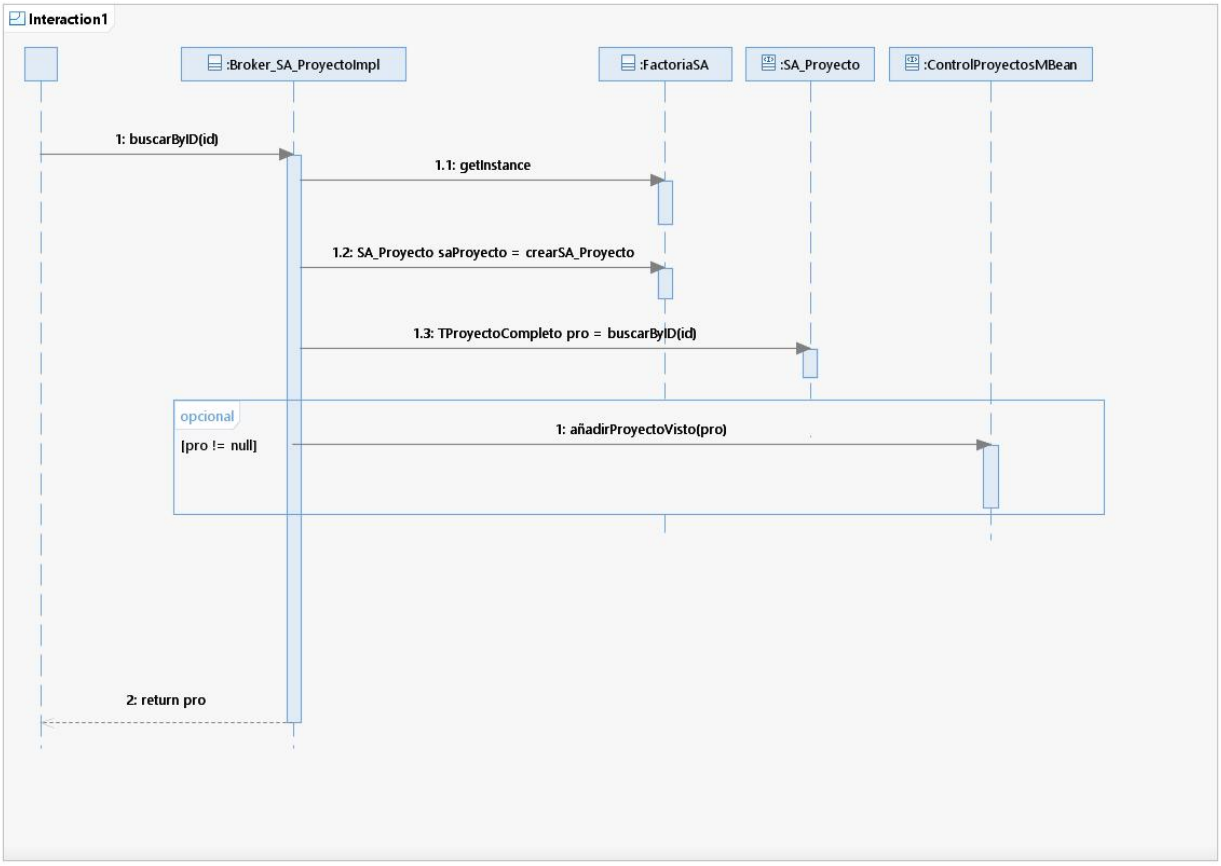

Figura 4.7 - Diagrama de secuencia del patrón *dynamic service management* para el método *buscarById* de la clase *Broker_SA_ProyectoImpl*

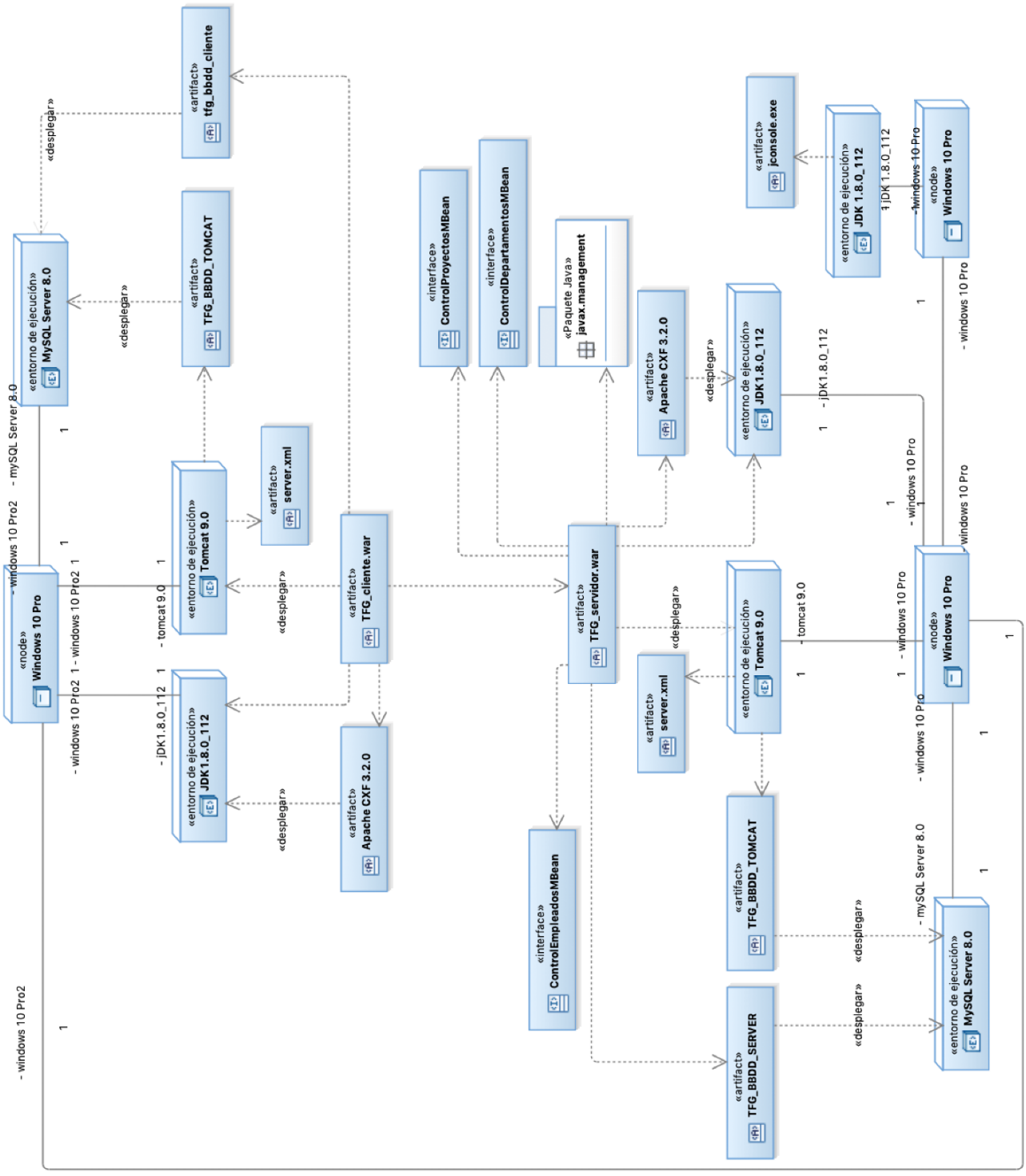

Figura 4.8 - Diagrama de despliegue del patrón *dynamic service management*

4.4. Obfuscated Transfer Object

El patrón *obfuscated transfer object* (Steel et al., 2005) protege datos críticos cuando la información es pasada a través de la aplicación y entre capas. Los *transfer objects* son un mecanismo para transportar elementos con datos a través de capas y componentes de manera eficiente (Alur et al., 2003). Sin embargo, esto conlleva un problema de seguridad, pues se expone información innecesariamente a componentes que no la requieren o no deberían tener acceso a ella. Al utilizar el patrón *obfuscated transfer object* esta información crítica se protege permitiendo el acceso a ella únicamente al consumidor de la información.

Las figuras 4.9, 4.10, 4.11 y 4.12 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

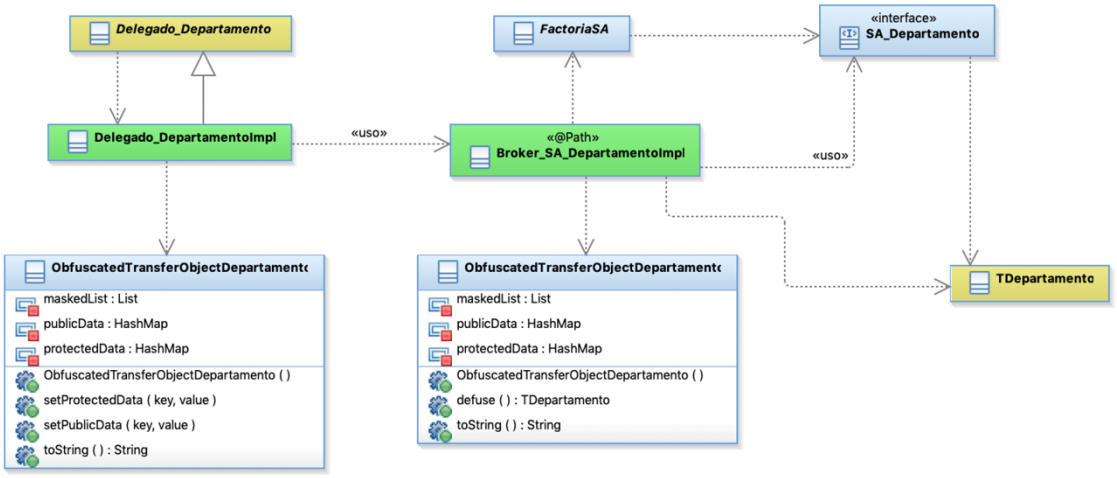

Figura 4.9 - Diagrama de clases del patrón *obfuscated transfer object*. En este caso, se ofusca el transfer con la información de un departamento

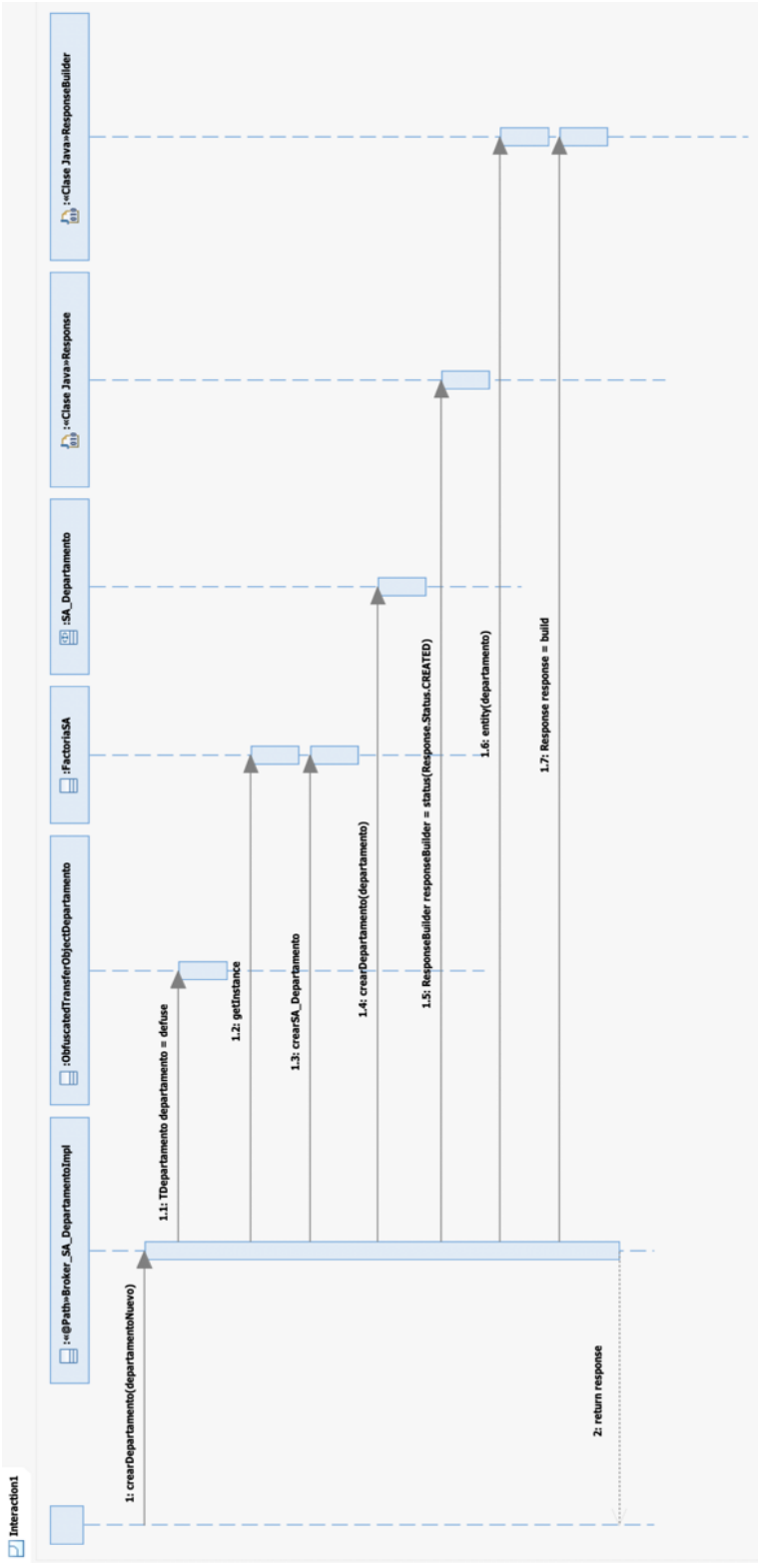

Figura 4.10 - Diagrama de secuencia del patrón *obfuscated transfer object* para el método *crearDepartamento* de la clase *Broker_SA_DepartamentoImpl*

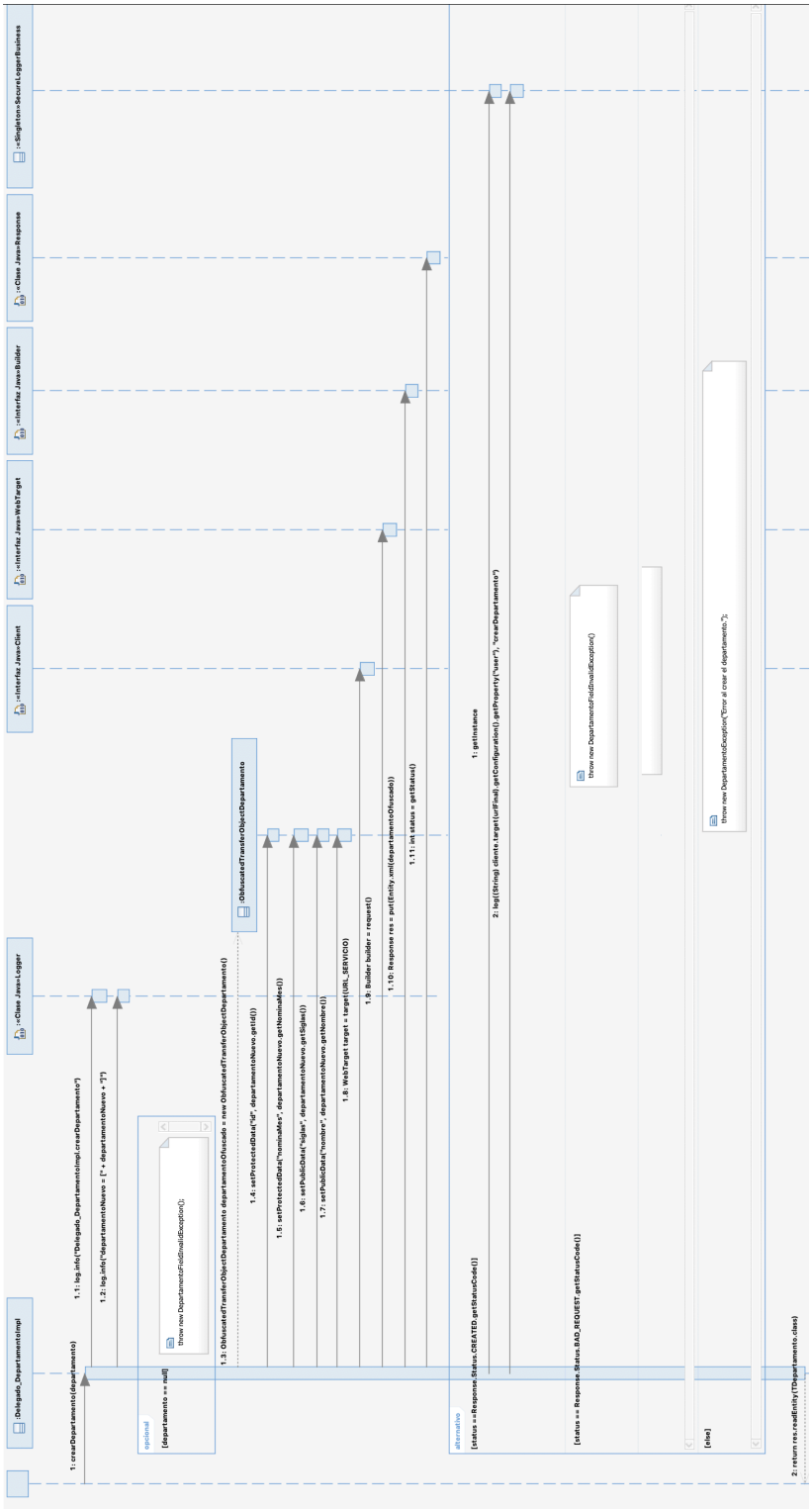

Figura 4.11 - Diagrama de secuencia del patrón *obfuscated transfer object* para el método *crearDepartamento* de la clase *Delegado_DepartamentoImpl*

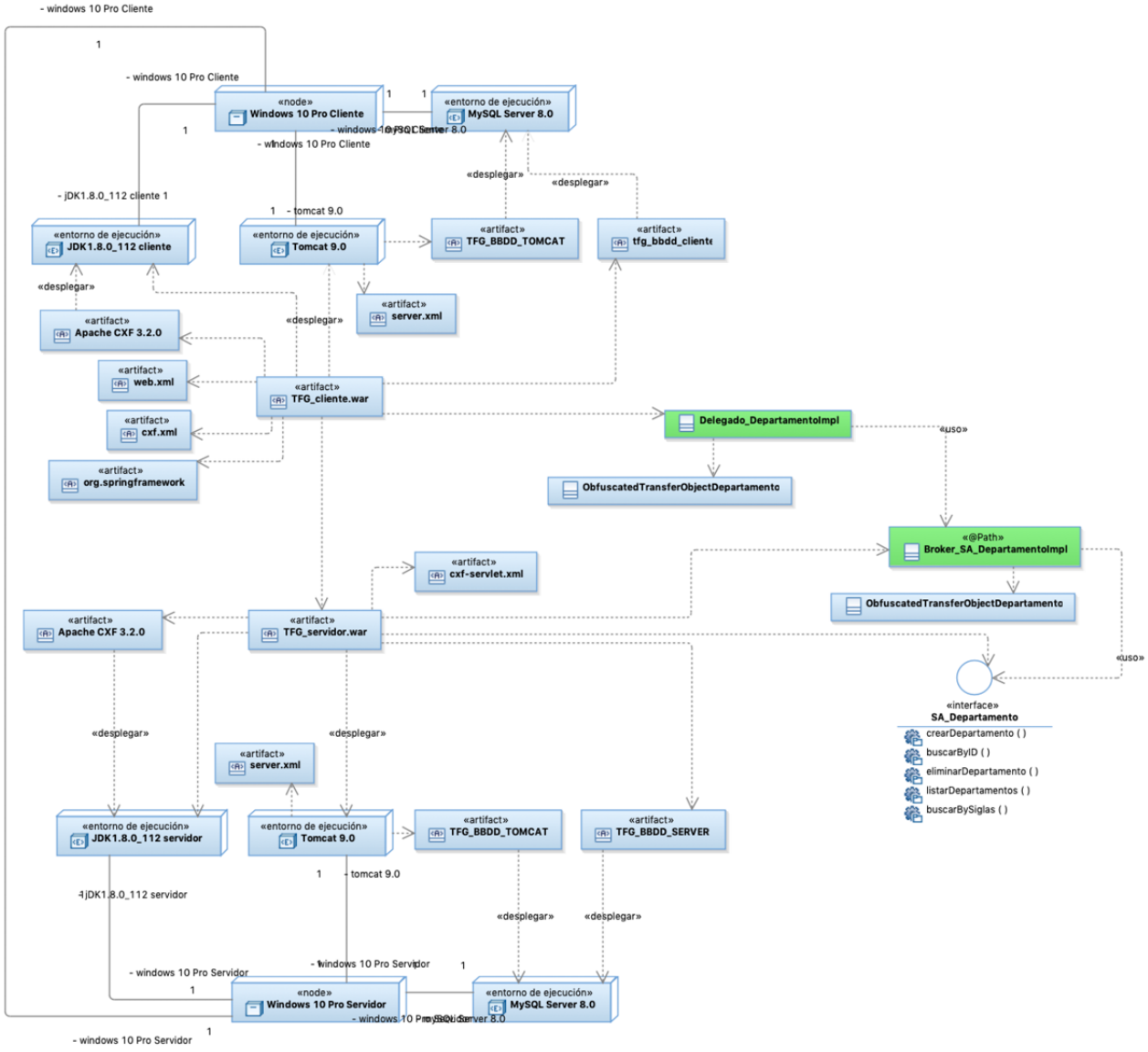

Figura 4.12 - Diagrama de despliegue del patrón *obfuscated transfer object*

4.5. Policy Delegate

El patrón *policy delegate* (Steel et al., 2005) encapsula los detalles de invocación de los servicios de seguridad y controla las interacciones del WSC interceptando y administrando las políticas en las peticiones del cliente. De esta manera se desacopla el código del cliente de la implementación específica de invocación al WSP.

En nuestro proyecto este patrón está implementado mediante los delegados del negocio en la aplicación cliente, que fueron desarrollados en el TFG previo. Por tanto, para ampliar la funcionalidad de este patrón hemos implementado un *logger* con la operación invocada y como usuario el cliente del servicio web.

Las figuras 4.13, 4.14 y 4.15 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

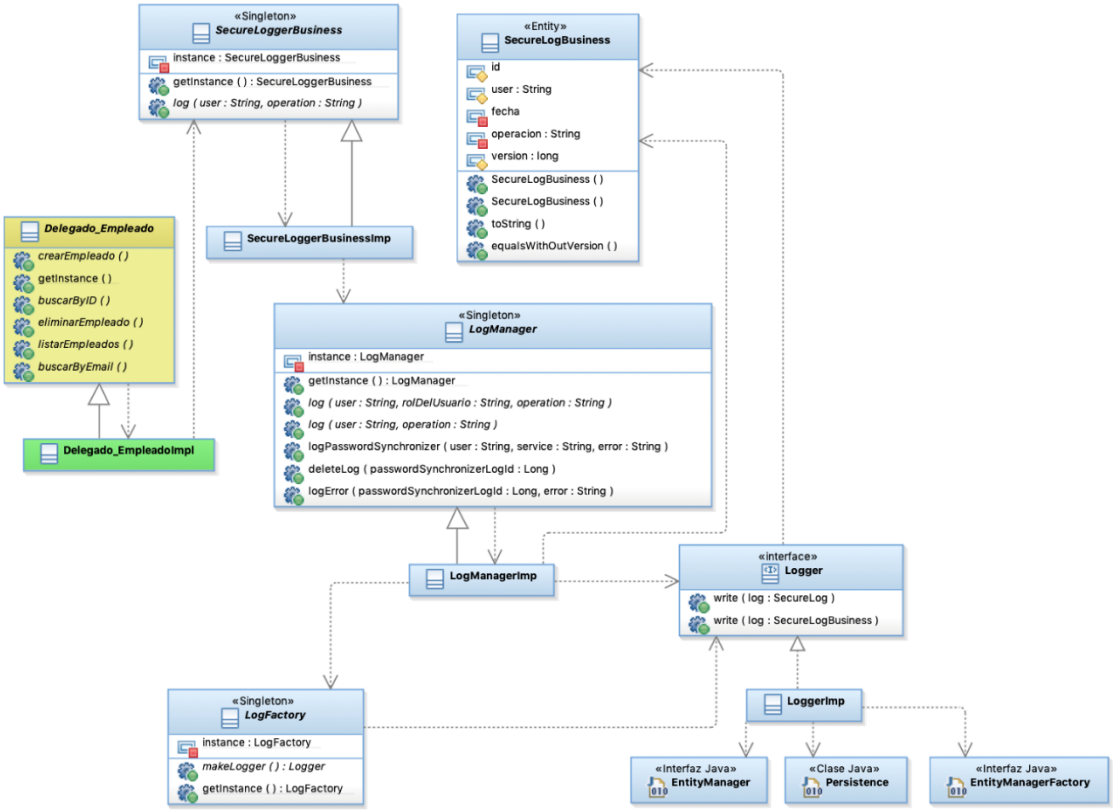

Figura 4.13 - Diagrama de clases del patrón *policy delegate*

Figura 4.14 - Diagrama de secuencia del patrón *policy delegate* para el método *crearEmpleado* de la clase *Delegado_EmpleadoImpl*

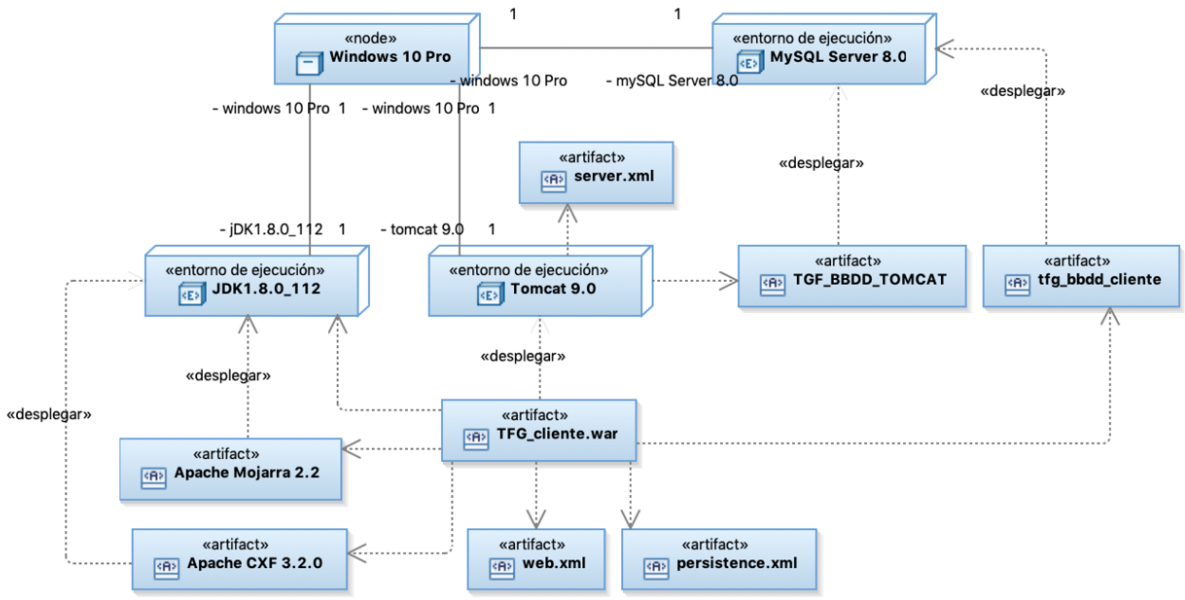

Figura 4.15 - Diagrama de despliegue del patrón *policy delegate*

4.6. Secure Service Façade

El patrón *secure service façade* (Steel et al., 2005) integra implementaciones de grano fino independientes de la seguridad y ofrece una interfaz unificada a los clientes. Este patrón actúa como una puerta de acceso donde las peticiones del cliente son validadas y enviadas a los servicios.

En nuestro trabajo la validación del contexto de seguridad realizada por este patrón no es necesaria porque, tal y como sugieren Steel et al. (2005), en este proyecto la autenticación y el control de acceso al servicio están delegados en Apache Tomcat que implementa el patrón *container managed security*. Además, en nuestro trabajo no tenemos fachadas de sesión, propias de implementaciones basadas en EJBs (Oracle, 2003). Por todo esto, hemos decidido no implementar el patrón *secure service façade* en nuestro proyecto.

4.7. Secure Session Object

El patrón *secure session object* (Steel et al., 2005) provee una manera de encapsular la información de autenticación y autorización tal como credenciales, roles y privilegios, y los transporta de forma segura. Esto permite que las componentes de distintas capas puedan verificar que el transmisor de la petición está autenticado y autorizado para ese servicio en particular.

En nuestro trabajo, no es necesario implementar explícitamente este patrón, ya que la información sobre roles y privilegios está delegada en Apache Tomcat y Apache CXF a través del estándar SAML, utilizando la estrategia de interceptores. El cliente se autentica en el servidor y el cliente recupera una referencia a la información de seguridad (que debería estar guardada en el *secure session object*) mediante un interceptor Apache CXF. En la implementación de Apache Tomcat el *secure session object* es serializado mediante un *token* SAML.

5. Patrones Web-Services

5.1. Message Interceptor Gateway y Message Inspector

El patrón *message interceptor gateway* (Steel et al., 2005) es un proxy que provee un punto de entrada centralizado encapsulando el acceso a todos los *endpoints* de un *WSP*. Actúa como un controlador que garantiza los mecanismos de seguridad para todo el tráfico XML entrante y saliente.

El patrón *message inspector* (Steel et al., 2005) es un componente modular que se integra con los componentes de los servicios para manejar pre-procesado y post-procesado de mensajes SOAP o XML entrantes y salientes. Actúa como un filtro para analizar y validar las peticiones a los servicios web. Puede haber tantos filtros como sea necesario, estos filtros se encadenan y son invocados por el patrón *message interceptor gateway.*

En nuestro proyecto el patrón *message interceptor gateway* está implementado por Apache Tomcat, y el patrón *message inspector* por interceptores Apache CXF, ya que era el mecanismo más sencillo de implementación. Para filtrar las peticiones, tanto a servicios web SOAP como a servicios web REST, es necesario definir los interceptores JAX-WS y JAX-RS en el fichero `cxf-servlet.xml` del WSP. Se han implementado dos *message inspectors*: uno obtiene la IP de la petición y deniega el acceso si la IP es 127.0.0.1 y el otro comprueba que la petición está autorizada, es decir, que las credenciales de la petición coinciden con alguna de las almacenadas en la base datos de usuarios de *Tomcat* (duplicando así la funcionalidad implementada con el *container managed security*).

Las figuras 5.1, 5.2 y 5.3 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón *message inspector* en el contexto de una aplicación multicapa J2EE. El patrón *message interceptor gateway* no tiene diagramas UML al estar delegado en Apache Tomcat.

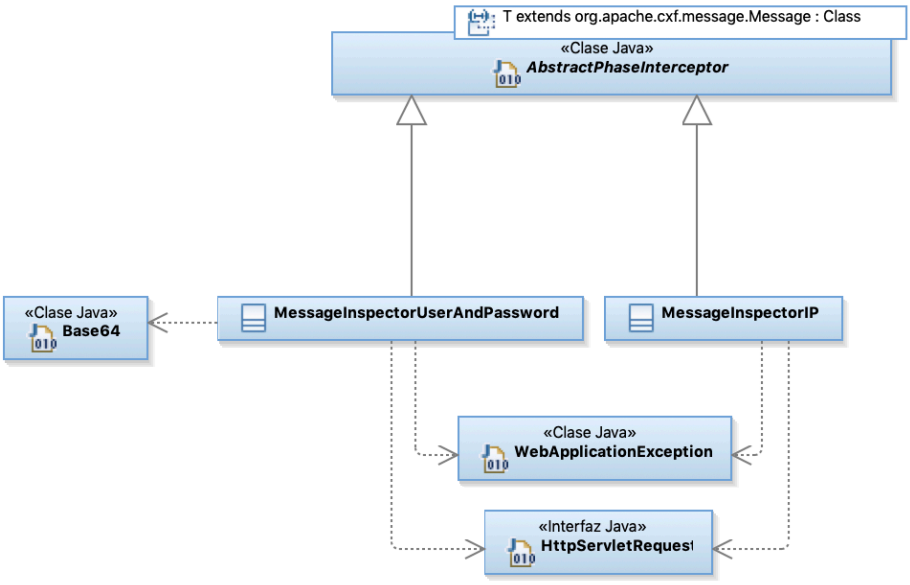

Figura 5.1 - Diagrama de clases del patrón *message inspector*

Figura 5.2 - Diagrama de secuencia del patrón *message inspector* para el método *handleMessage* de la clase *MessageInspectorIP*

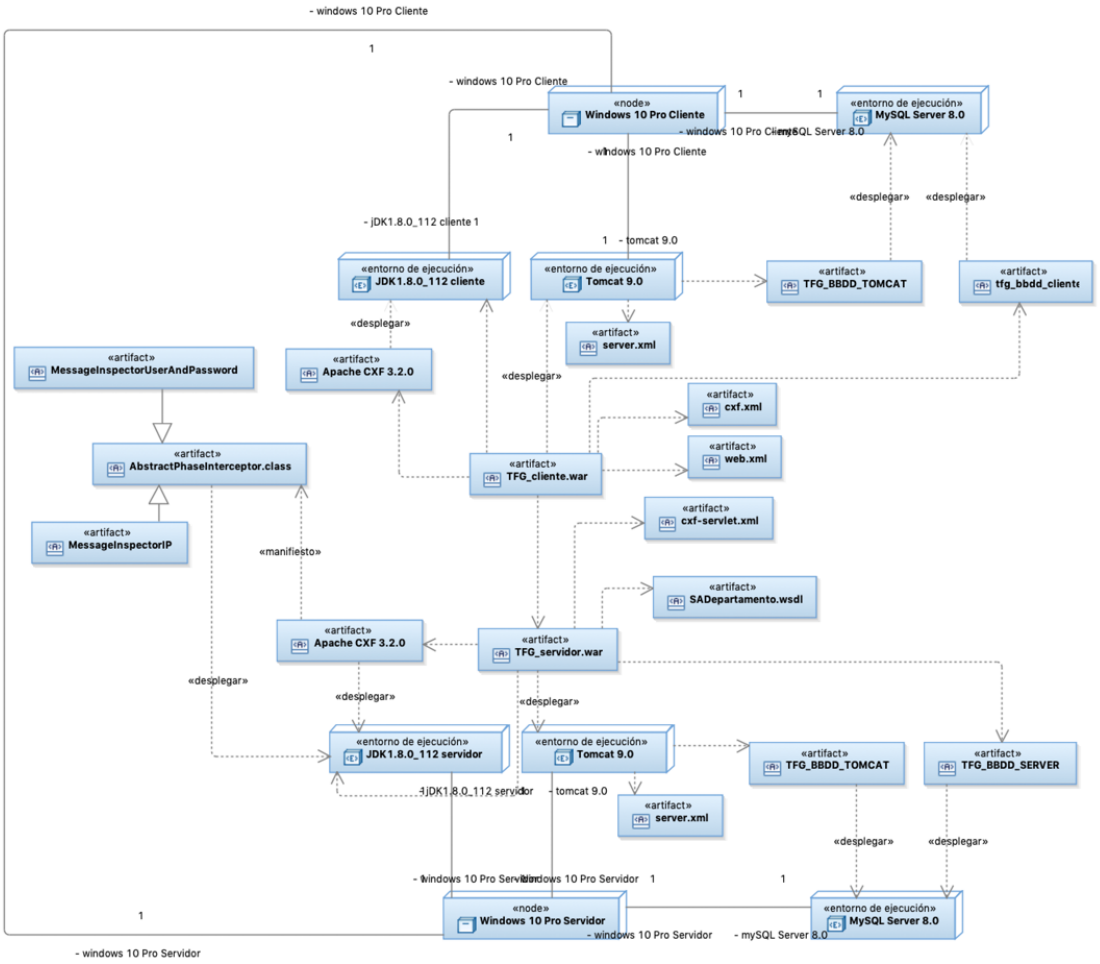

Figura 5.3 - Diagrama de despliegue del patrón *message inspector*

5.2.Secure Message Router

El patrón *secure message router* (SMR) (Steel et al., 2005) establece una infraestructura de seguridad intermediaria que agrega acceso a múltiples WSP desde un mismo WSC. El propósito principal de este patrón es que al acceder a distintos WSP evitando que el WSC tenga que hacer *login* explícito en cada uno de ellos, facilitando así el *single sign-on.* El mecanismo utilizado para identificar al usuario es un proveedor de identidad, al que debe acceder el WSC para validar al usuario mediante *tokens* SAML, que son transmitidos a los WSP.

En nuestro trabajo el proveedor de identidad utilizado es de la empresa Talend (Talend, 2019), que provee un *security token service* (STS) para autenticar y autorizar a los usuarios de servicios web mediante *tokens* SAML. Para la implementación mediante el STS de Talend se deben modificar tanto el WSC como el WSP. El WSC debe solicitar un *token* SAML al STS antes de la invocación a un WSP, y el WSP debe validar el *token* SAML recibido con el proveedor de identidad para verificar que el cliente del servicio web está autenticado y autorizado (Mazza, 2017; Wulff 2011; Wulff 2012). El uso del STS coincide al 100% con la filosofía de uso del SMR, pero, paradójicamente, difiere en gran manera con el *identity provider* sugerido en Steel et al. (2005), propio de aplicaciones web y cuyo uso en el SMR era inviable.

SAML define distintos perfiles, dependiendo del tipo de uso que se necesite (OASIS Standard, 2005). En el caso del STS utilizado en este proyecto hemos usado el perfil *holder of key*. Mediante este método, el STS crea un *token* SAML que contiene la clave pública del WSC y firma el token SAML con su propia clave privada. A continuación, el WSC incluye en la petición el *token* SAML y firma la petición SOAP al WSP con su clave privada. Por último, el WSP valida el mensaje SOAP y el *token* SAML.

En la aplicación cliente la única modificación que hay que realizar para obtener el *token* SAML es añadir una propiedad de JAX-WS al fichero `cxf.xml`. El siguiente bloque muestra la propiedad que se debe añadir al WSC que comprueba la contraseña obteniendo un *token* de un STS:

```
<jaxws:properties>
    <entry key=”ws-security.signature.properties”
value=”cliKeystore.properties” />
    <entry key=”ws-security.signature.username” value=”myclientkey” />
    <entry key=”es-security.callback-handler”
value=”com.eduardosergio.TFG_cliente.presentacion.seguridad.callback.ClientCal
lbackHandler” />
    <entry key=”ws-security.encryption.properties”
value=”cliKeystore.properties” />
    <entry key=”ws-security.encryption.username” value=”myservicekey” />
    <entry key=”es-security.sts.client”>
```

```
        <bean class=”org.apache.cxf.ws.security.trust.STSClient”>
            <constructor-arg ref=”cxf”>
            <property name=”wsdlLocation”
value=”https://localhost:8443/TFG_STS/STS?wsdl” />
            <property name=”serviceName” value=”{http://docs.oasis-
open.org/ws-sx/ws-trust/200512/}SecurityTokenService” />
            <property name=”endpointName” value=”{https://docs.oasis-
open.org/ws-sx/ws-trust/200512/}STS_Port” />
            <property name=”properties” >
                <map>
                    <entry key=”ws-security.username” value=”alice” />
                    <entry key=”ws-security.callback-handler”
value=”com.eduardosergio.TFG_cliente.presentacion.seguridad.callback.ClientCal
lbackHandler” />
                    <entry key=”ws-security.encryption.properties”
value=”cliKeystore.properties” />
                    <entry key=”ws-security.encryption.username”
value=”mystskey” />
                </ map>
            </property>
        </bean>
    </entry>
</jaxws:properties>
```

Además, se debe crear la clase `ClientCallbackHandler` que validará el usuario y contraseña del *token* recibido del STS.

En el WSP las modificaciones que hay que realizar son únicamente en los servicios web. Se deben añadir anotaciones que indican los ficheros en los que están las políticas de seguridad, es decir, la ubicación del STS para validar el *token* recibido y permitir el acceso al servicio web. Además, se debe añadir la clase `ServiceKeystorePasswordCallback` que almacena el usuario y contraseña y lo valida con las credenciales del *token* SAML recibido. Las anotaciones correspondientes a las políticas de acceso para el STS se muestran

en la figura 5.4, que muestra una captura de uno de los servicios web que validan el *token* recibido con el STS:

```
@WebService
public interface PasswordSynchronizerSTS1 {

    @Policies({
        @Policy(uri="policy.xml",
                placement=Policy.Placement.BINDING_OPERATION,
                includeInWSDL=true),
        @Policy(uri="inputPolicy.xml",
                placement=Policy.Placement.BINDING_OPERATION_INPUT,
                includeInWSDL=true),
        @Policy(uri="outputPolicy.xml",
                placement=Policy.Placement.BINDING_OPERATION_OUTPUT,
                includeInWSDL=true)
    })

    @WebMethod
    public Integer synchronize(String user, String pass);
}
```

Figura 5.4 – Captura de la interfaz *PasswordSynchronizerSTS1*

En este proyecto, el SMR delega la invocación del WSC a los WSP en un *single sign-on delegator* (SSOD) (Steel et al., 2005), descrito en la Sección 6. Además, el SMR juega el papel de *password synchronizer* (Steel et al., 2005), descrito en la Sección 7. Así, el SMR delega en el SSOD la invocación de dos servicios web SOAP y un servicio web REST que sincronizan la contraseña del usuario en tres bases de datos distintas. Los WSP SOAP validan su usuario y contraseña mediante *tokens* SAML obtenidos de dos STSs distintos, mientras que WSP REST valida su usuario y contraseña utilizando *container managed security* (es decir, delegando en Apache Tomcat). Cabe destacar que el uso del SMR necesitaría la aplicación de transacciones distribuidas (Little, 2004), aspecto este que queda totalmente fuera de los objetivos de este trabajo. Para simular el comportamiento de las transacciones distribuidas, el SMR lleva a cabo un proceso de log para facilitar la compensación transaccional, si fuera necesario.

En esta sección se incluyen los diagramas de clase y secuencia relacionados SMR, dejando para las secciones correspondientes los diagramas de clase y secuencia vinculados al SSOD y al *password synchronizer*.

Las figuras 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13 y 5.14 incluyen los diagramas de clase, secuencia y despliegue, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE. En este trabajo el SMR ha sido implementado como un *singleton*, pero nada impediría implementarlo como una clase oculta tras un interfaz y generada con una factoría.

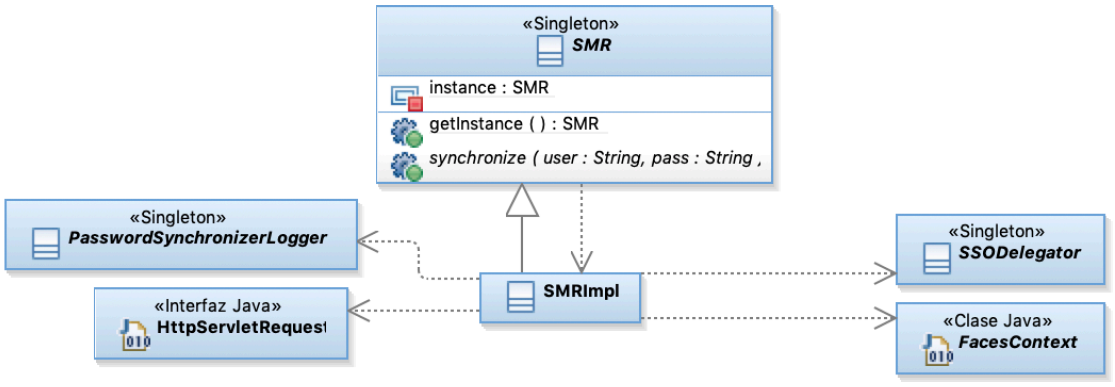

Figura 5.5 - Diagrama de clases del patrón *secure message router*

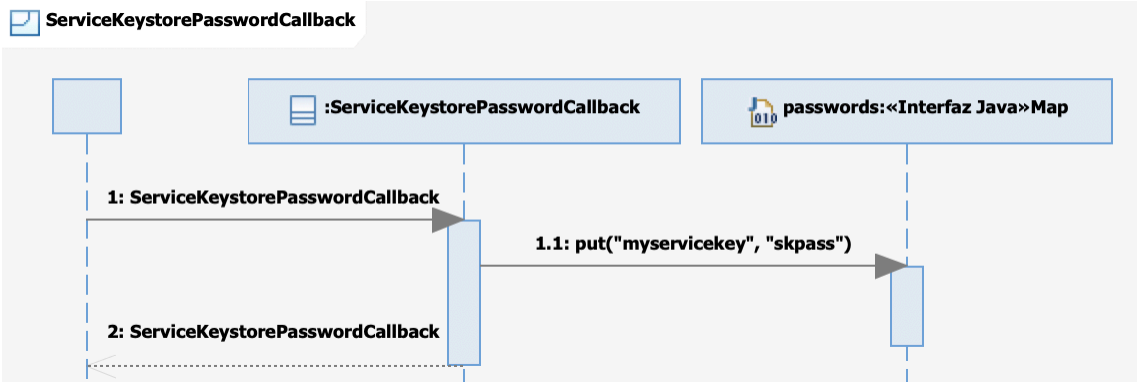

Figura 5.6 - Diagrama de secuencia del patrón *secure message router* para la constructora de la clase *ServiceKeystoreCallbackHandler*

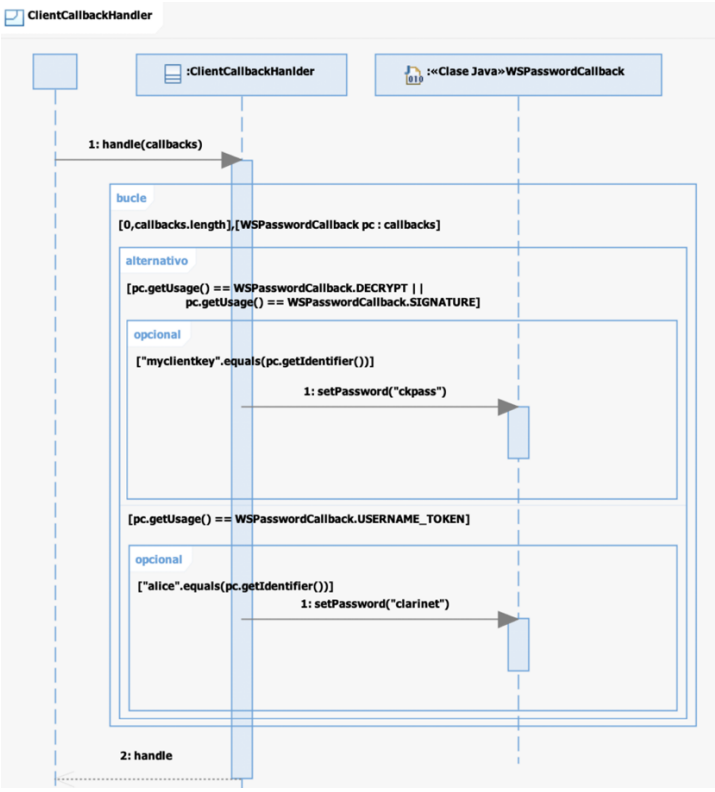

Figura 5.7 - Diagrama de secuencia del patrón *secure message router* para el método *handle* de la clase *ClientCallbackHandler*

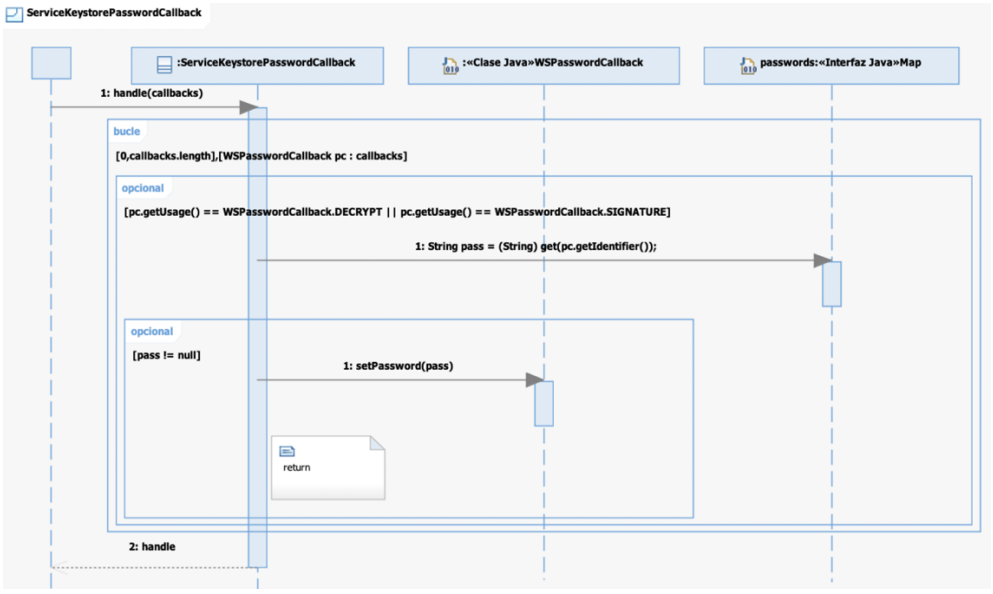

Figura 5.8 - Diagrama de secuencia del patrón *secure message router* para el método *handle* de la clase *ServiceKeystoreCallbackHandler*

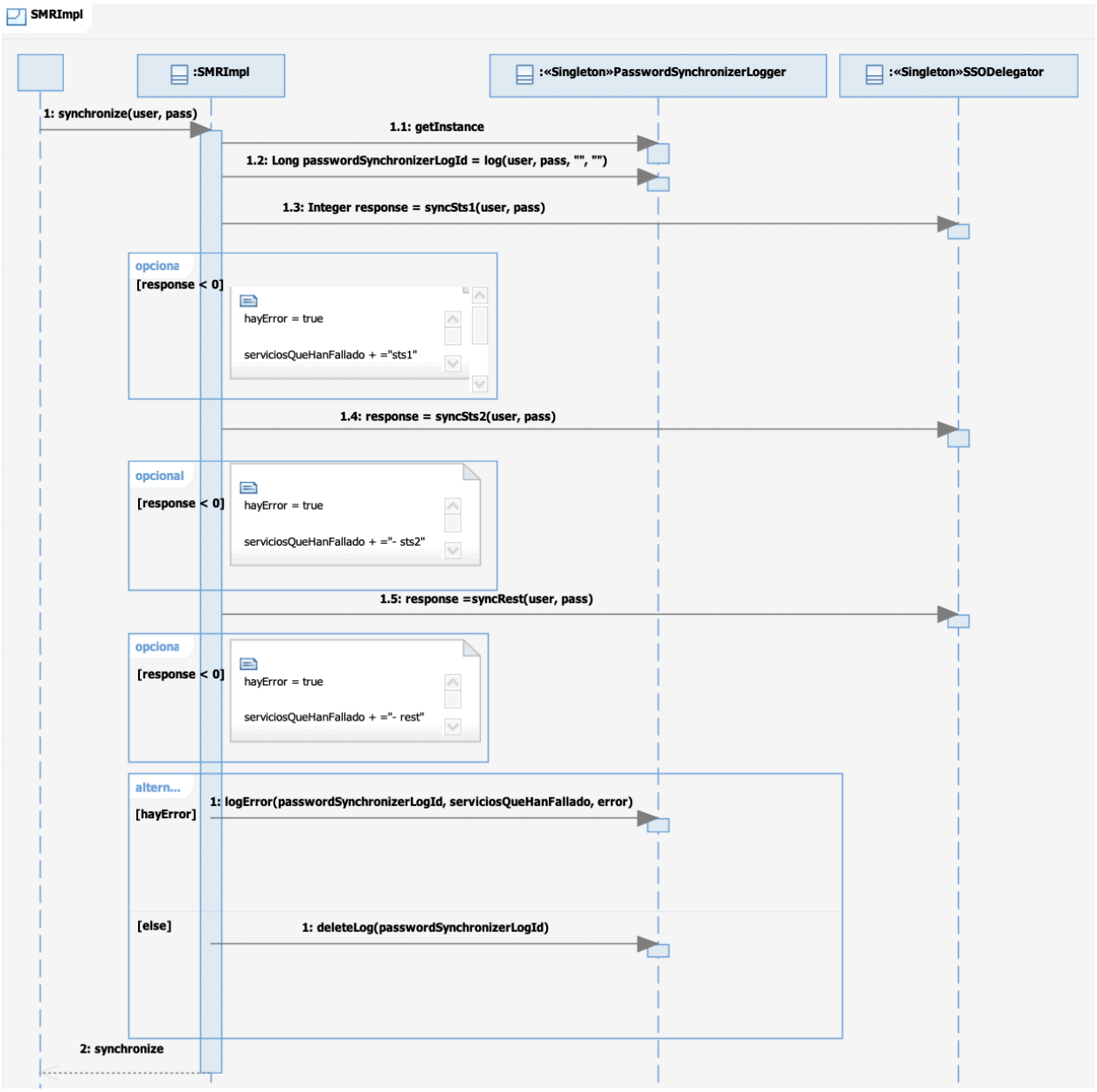

Figura 5.9 - Diagrama de secuencia del patrón *secure message router* para el método *synchronize* de la clase *SMRImpl*

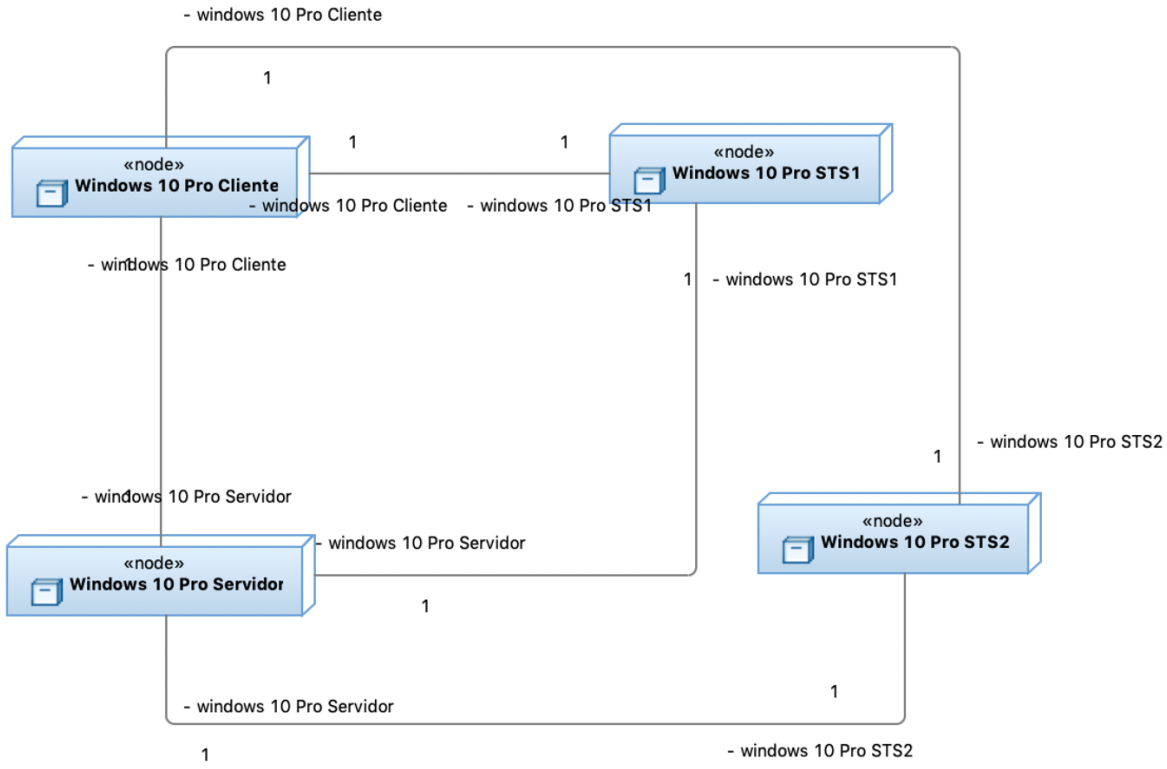

Figura 5.10 – Nodos del diagrama de despliegue del patrón *secure message router*

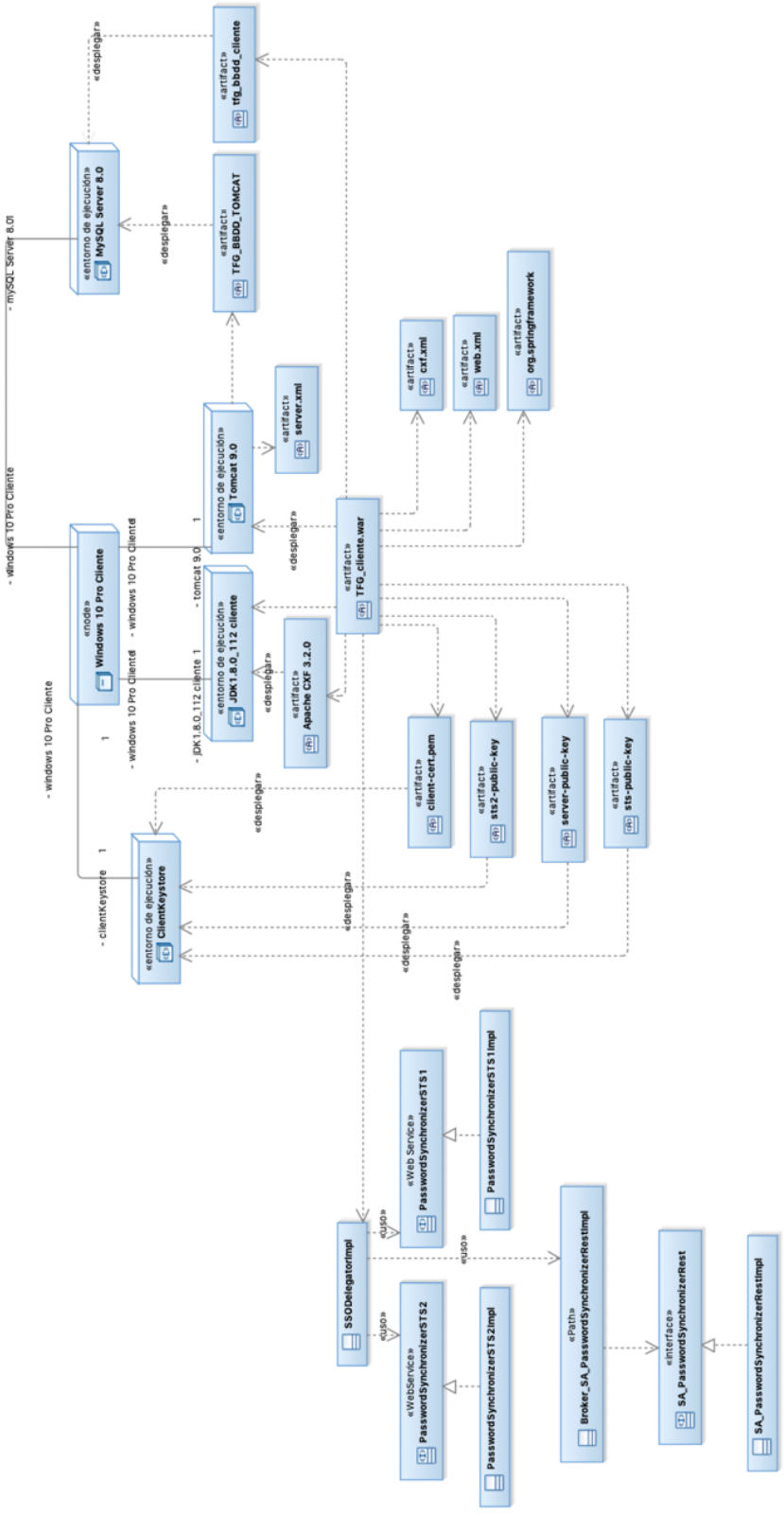

Figura 5.11 – Diagrama de despliegue del WSC del patrón *secure message router*

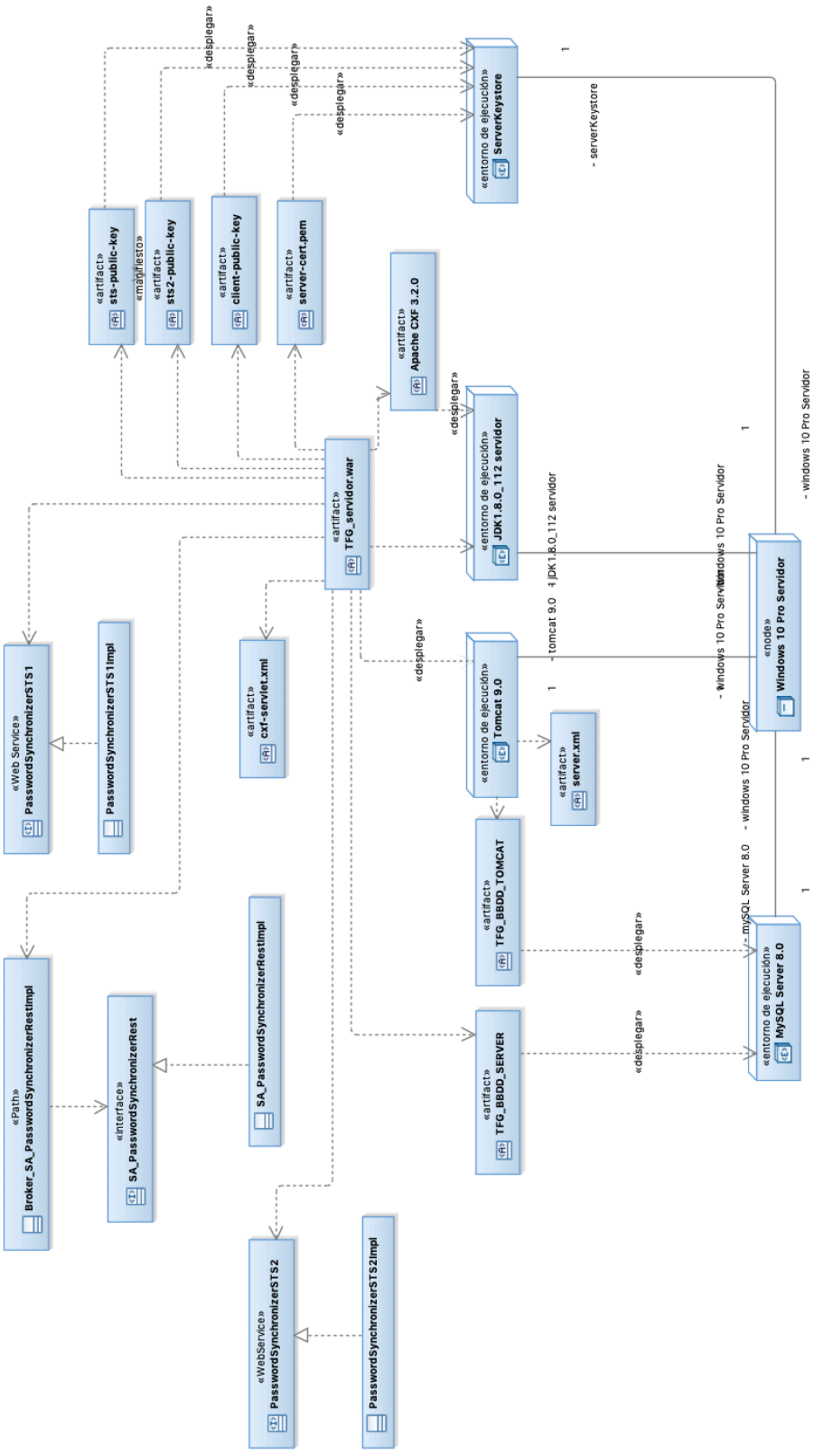

Figura 5.12 – Diagrama de despliegue del WSP del patrón *secure message router*

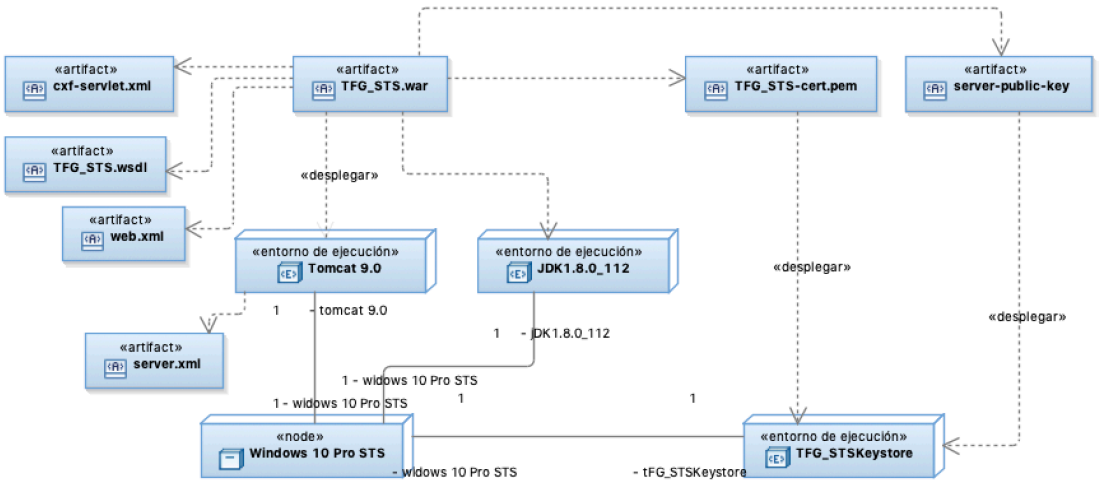

Figura 5.13 – Diagrama de despliegue del primer *Security Token Service* del patrón *secure message router*

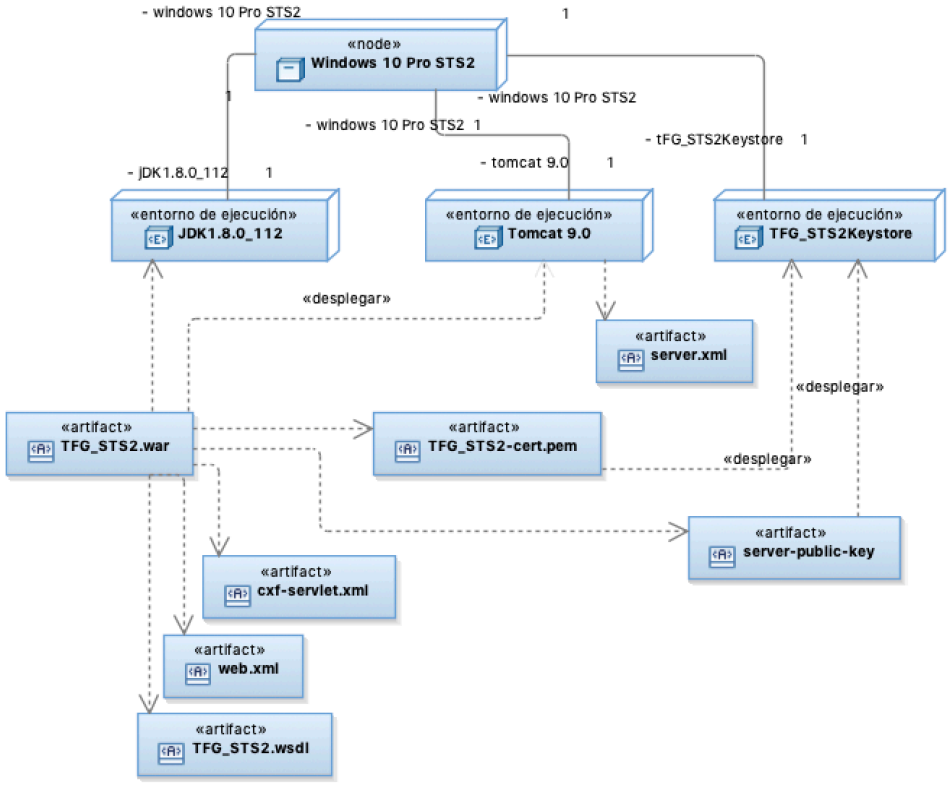

Figura 5.14 – Diagrama de despliegue del segundo *Security Token Service* del patrón *secure message router*

6. Patrones Identity Management

6.1. Assertion Builder

El patrón *assertion builder* (Steel et al., 2005) encapsula la lógica de procesamiento para crear aserciones SAML de autorización y autenticación y lo expone como un servicio.

Tal y como sugieren Steel et al. (2005), en vez de desarrollarlo desde cero, en nuestro trabajo hemos usado un proveedor de identidad implementa un *assertion builder* encargado de crear los *token* SAML. El proveedor utilizado es el STS de Talend (Talend, 2019), que actúa como proveedor de identidad, generando un *token* SAML cuando recibe una petición de un cliente y validándolo posteriormente con el servicio web al que el cliente intenta acceder.

No obstante, Apache WSS4J (Apache Software Foundation, 2008) proporciona diversas clases para el manejo de *tokens* SAML en su paquete `org.apache.wss4j.common.saml`.

6.2. Credential Tokenizer

El patrón *credential tokenizer* (Steel et al., 2015) encapsula diferentes tipos de credenciales de usuario como un *token* de seguridad que puede ser reusado entre diferentes proveedores de seguridad. Es una API de seguridad que crea y recupera la información de identidad del usuario. Mediante este patrón se pretende crear una componente reutilizable que ayuda a extraer la lógica de procesamiento y la gestión de *tokens* de seguridad de la lógica de negocio.

De esta forma, el delegado del negocio ya no guarda la información de usuario y contraseña, sino que lo hace el *credential tokenizer*, tal y como hace la implementación del SSOD en el siguiente apartado de este trabajo.

Aunque un *credential tokenizer* puede guardar la información de usuario y contraseña de diversas formas (por ejemplo, un certificado X.509), En nuestro trabajo el *token* de seguridad incluye el usuario y la contraseña validados por el contenedor de *servlets* Apache Tomcat, lo que en el catálogo de Steel et al. (2015) denominan *user token*.

Las figuras 6.1 y 6.2 incluyen los diagramas de clase y secuencia, que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE.

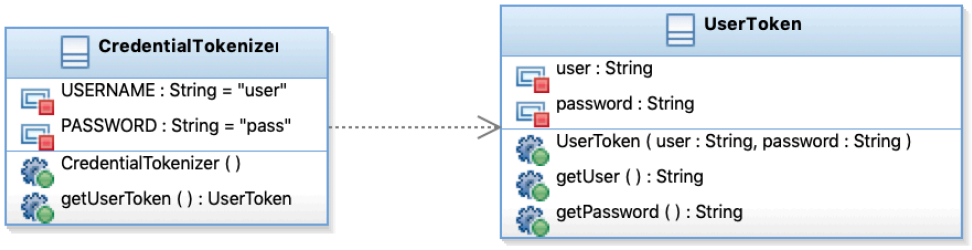

Figura 6.1 - Diagrama de clases del patrón *credential tokenizer*

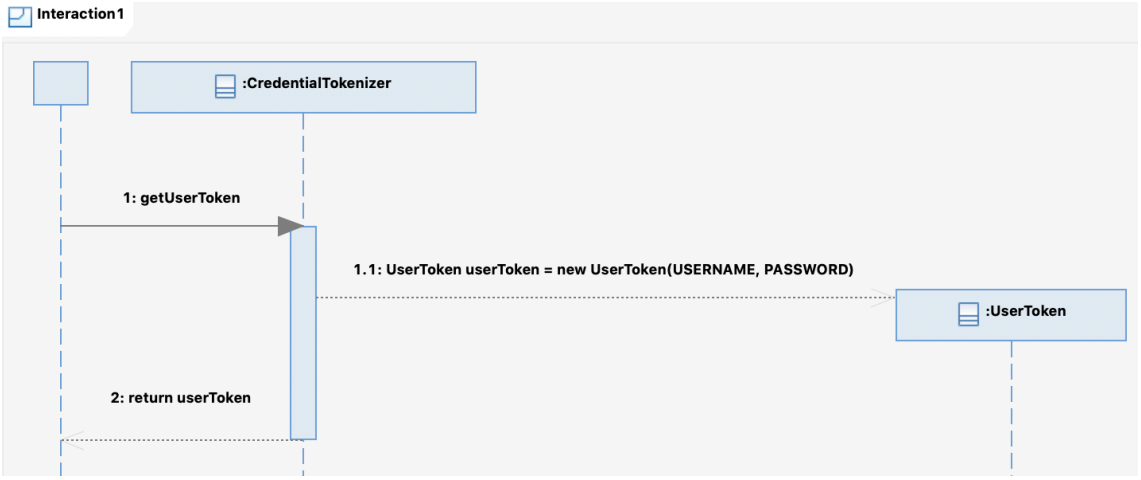

Figura 6.2 - Diagrama de secuencia del patrón *credential tokenizer* para el método *getUserToken* de la clase *CredentialTokenizer*

6.3. SSO Delegator

El patrón *single sign-on delegator* (SSOD) (Steel et al., 2005) encapsula el acceso a la gestión de identidad y a las funcionalidades de *single sign-on*, permitiendo una evolución independiente de servicios de gestión de identidad débilmente acoplados de su invocación por el WSC. Este patrón reside en la capa intermedia entre los clientes y los componentes del servicio de gestión de identidad. Delega la petición del servicio a las componentes remotas del servicio. Oculta al WSC los detalles de invocación de servicio, la recuperación de la configuración de seguridad y el procesamiento de los *tokens* de credenciales. En pocas palabras, un SSOD es un delegado del negocio usado por el WSC que encapsula la inclusión de credenciales para la autenticación y autorización de la invocación de los WSP.

Tal y como describíamos en el patrón SMR, el SSOD desarrollado en este proyecto incluye tres operaciones para la implementación de un *password synchronizer* mediante dos servicios web SOAP que reciben la información de autenticación/autorización de dos STSs distintos, y un tercer servicio web REST que reciben dicha información en la petición HTTP. Es más, en el delegado del SSOD, el usuario y contraseña ya no residen en él, sino en un *credential tokenizer*.

Las figuras 6.3, 6.4, 6.5 y 6.6 incluyen los diagramas de clases, secuencia y despliegue que describen la implementación del patrón en el contexto de una aplicación multicapa J2EE. El diagrama de despliegue coincide con el del SMR descritos en las figuras 5.10, 5.11, 5.12 y 5.14 .

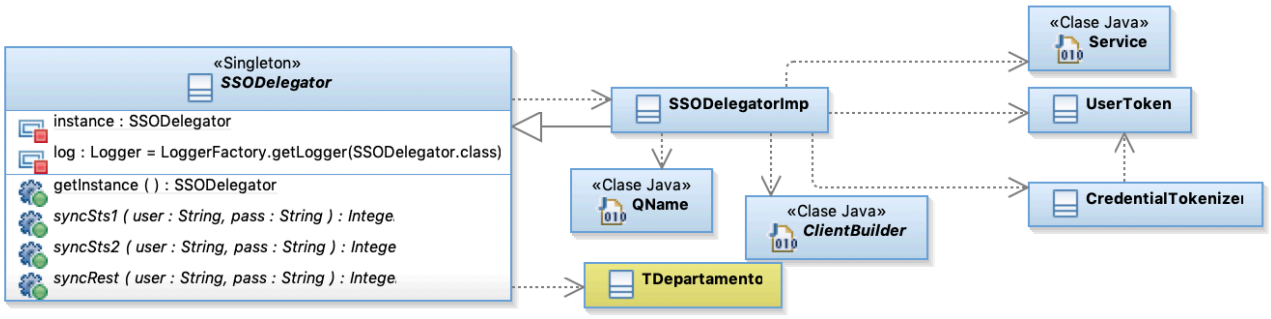

Figura 6.3 - Diagrama de clases del patrón *single sign-on delegator*

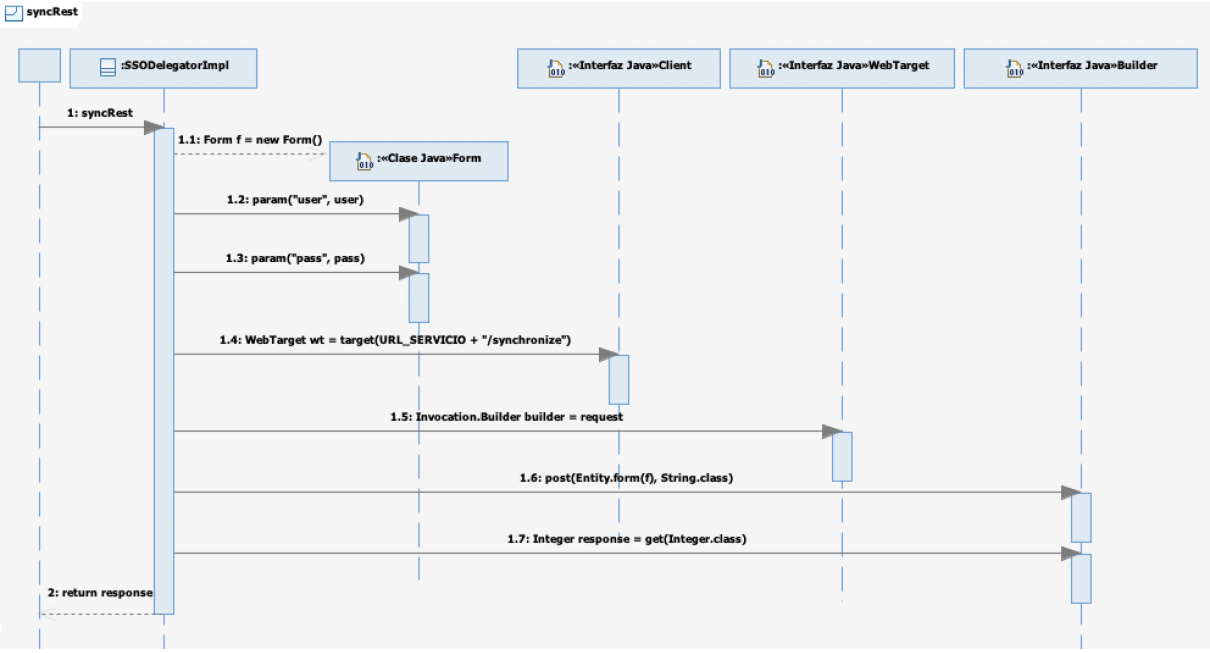

Figura 6.4 - Diagrama de secuencia del patrón *single sign-on delegator* para el método *syncRest* de la clase *SSODelegatorImpl*

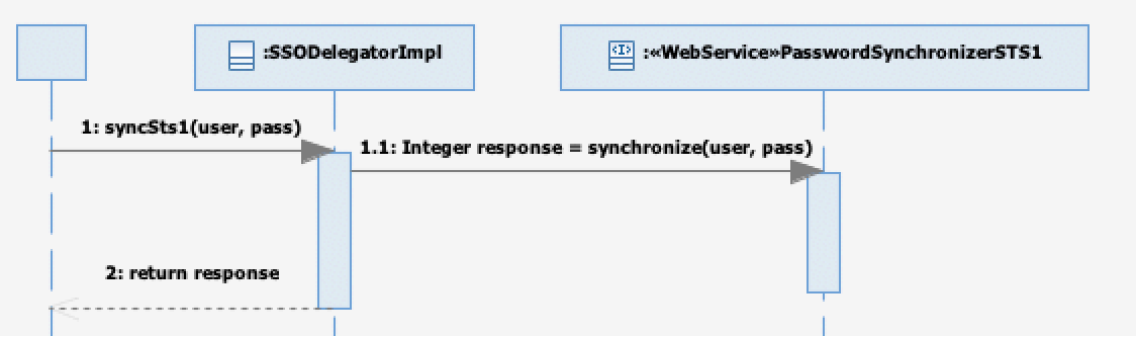

Figura 6.5 - Diagrama de secuencia del patrón *single sign-on delegator* para el método *syncSts1* de la clase *SSODelegatorImpl*

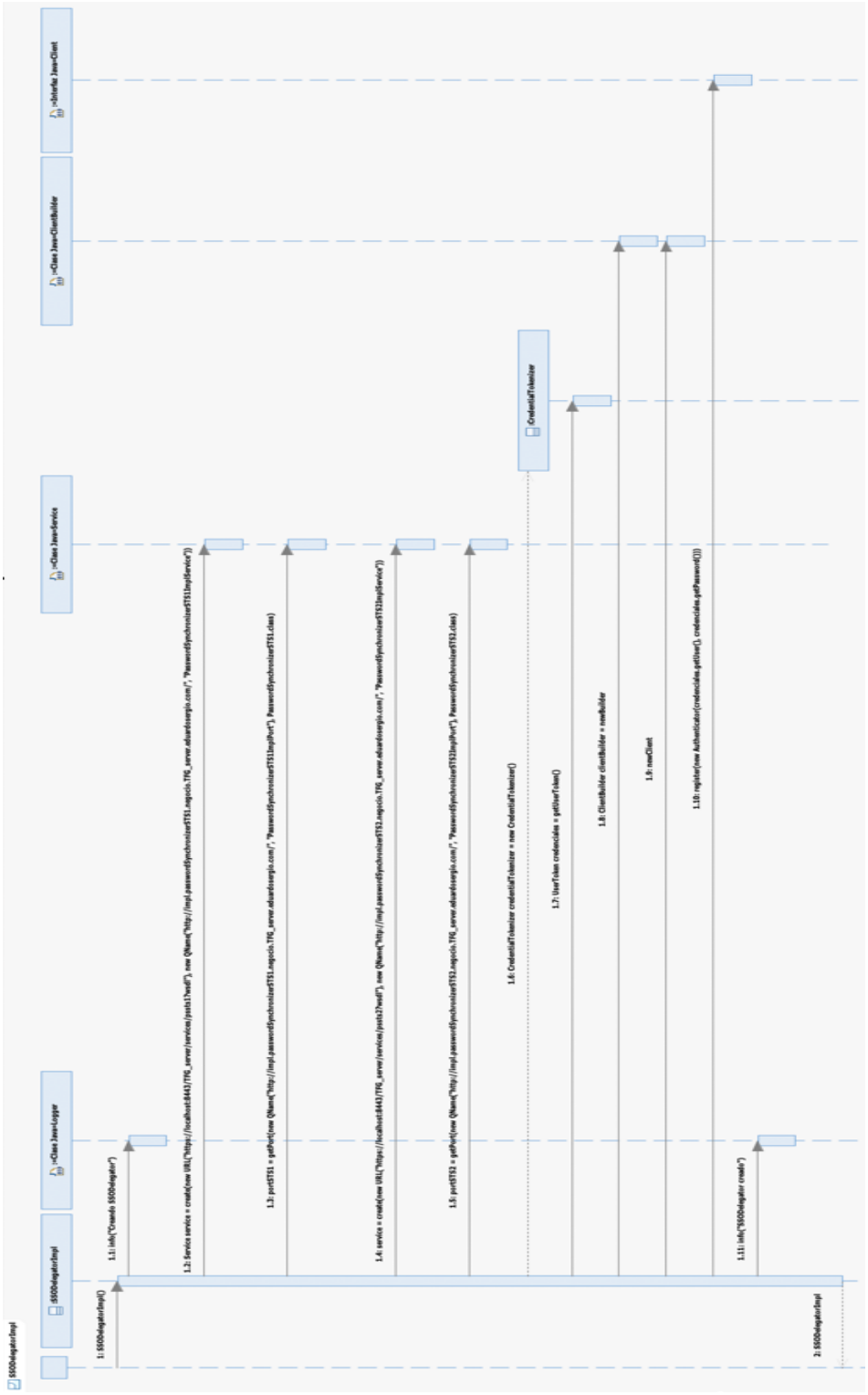

Figura 6.6 - Diagrama de secuencia del patrón *single sign-on delegator* para la constructora de la clase *SSODelegatorImpl*

7. Patrones Service Provisioning

7.1. Password Synchronizer

El patrón *password synchronizer* (Steel et al., 2015) centraliza la gestión de sincronización de credenciales de usuario entre diferentes aplicaciones a través de interfaces programáticas. El propósito de este patrón es que un administrador de sistemas pueda modificar las credenciales de un usuario en varios sistemas sin tener que modificarlo en cada uno individualmente. Por lo tanto, este patrón está más orientado a administración de sistemas que a seguridad.

Esta sincronización debería llevarse a cabo mediante una transacción distribuida (Little, 2004) en el WSC. Así, en el caso de que uno el sistema fallase al actualizar la contraseña, se haría un *rollback* de todas las modificaciones en todas las bases de datos modificadas, o al menos, se llevarían a cabo acciones de compensación transaccional. Sin embargo, debido a la complejidad de implementarlo con transacciones distribuidas, en este trabajo la transacción distribuida se simula con un *log* de error que se crea antes de invocar a los WSPs (que modifican la contraseña en una base de datos) y se borra después, en caso de que no haya habido ningún fallo. Este es un mecanismo muy sencillo, pero suficiente para garantizar la compensación transaccional de forma manual.

Tal y como hemos mencionado antes, el WSC que invoca las operaciones de los distintos WSP que manejan las cuentas de usuarios en distintos sistemas, es un es un SMR, que a su vez delega en un SSOD. Así, el SMR delega en el SSOD la invocación de dos servicios web SOAP y un servicio web REST que sincronizan la contraseña del usuario en tres bases de datos distintas. Los WSP SOAP validan su usuario y contraseña mediante *tokens* SAML obtenidos de dos STSs distintos, mientras que WSP REST valida su usuario y contraseña utilizando *container managed security* (es decir, delegando en Apache Tomcat).

También cabe destacar que, aunque la información de manejo de las cuentas de usuarios podría haberse hecho utilizando *Service Provisioning Markup Languaje* (SPML) (OASIS, 2006), en aras de una mayor simplicidad, hemos optado por no usarlo en este trabajo.

Las figuras 7.1, 7.2, 7.3, 7.4, 7.5 y 7.6 incluyen los diagramas de clase y secuencia, que describen la implementación del patrón en WSP. Los diagramas de clase y secuencia de las figuras 5.5, 5.6, 5.7, 5.8 y 5.9, del patrón SMR describen la implementación del WSC en términos de un SMR. El diagrama de despliegue de este patrón puede verse en los diagramas de despliegue del SMR, figuras 5.10, 5.11, 5.12, 5.13 y 5.14.

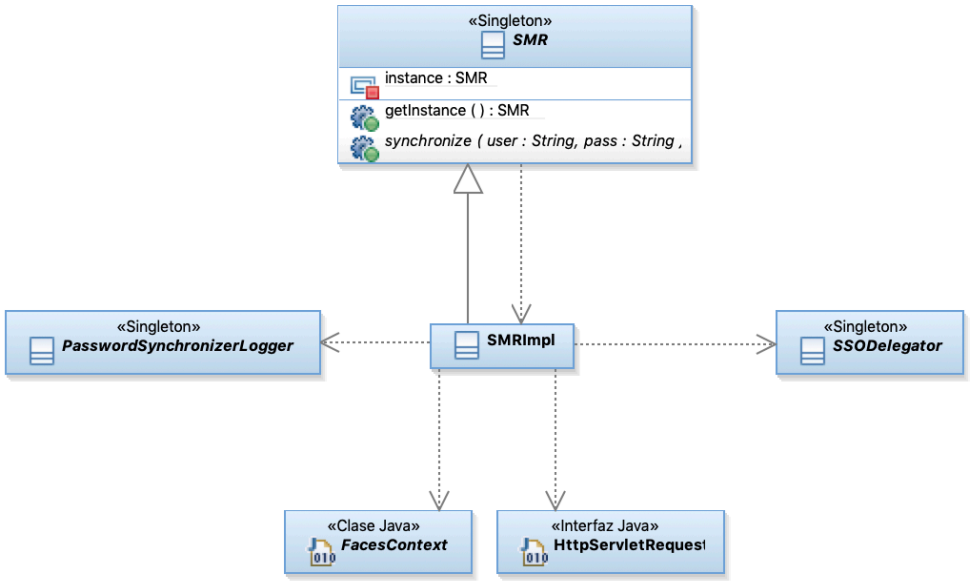

Figura 7.1 - Diagrama de clases de la aplicación cliente del patrón *password synchronizer*

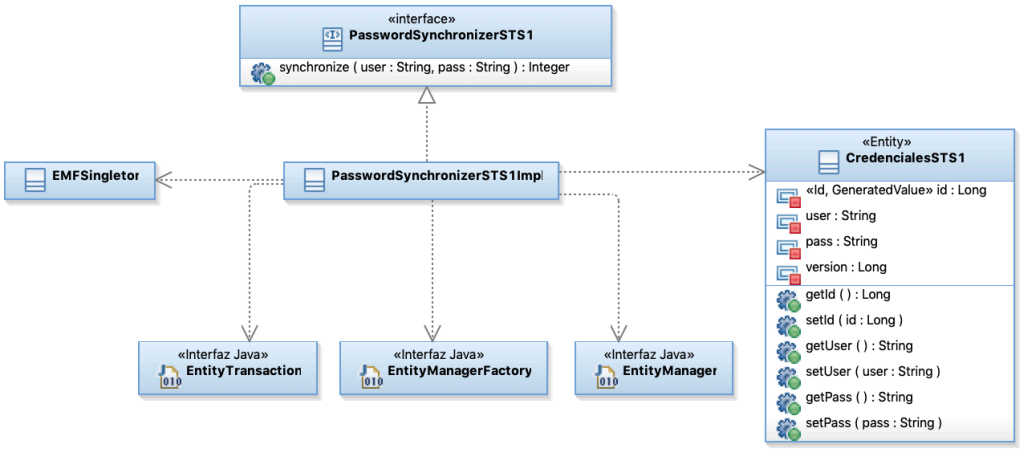

Figura 7.2 - Diagrama de clases de la aplicación servidor para el servicio web SOAP *PasswordSynchronizerSTS1* del patrón *password synchronizer*

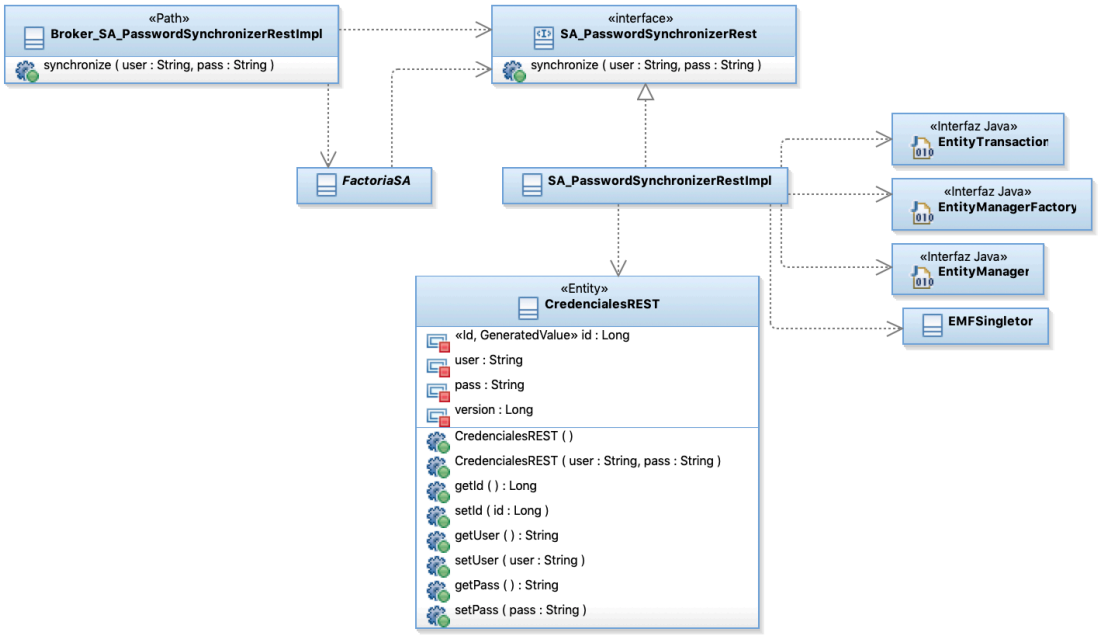

Figura 7.3 - Diagrama de clases de la aplicación servidor para el servicio web REST *SA_PasswordSynchronizerRest* del patrón *password synchronizer*

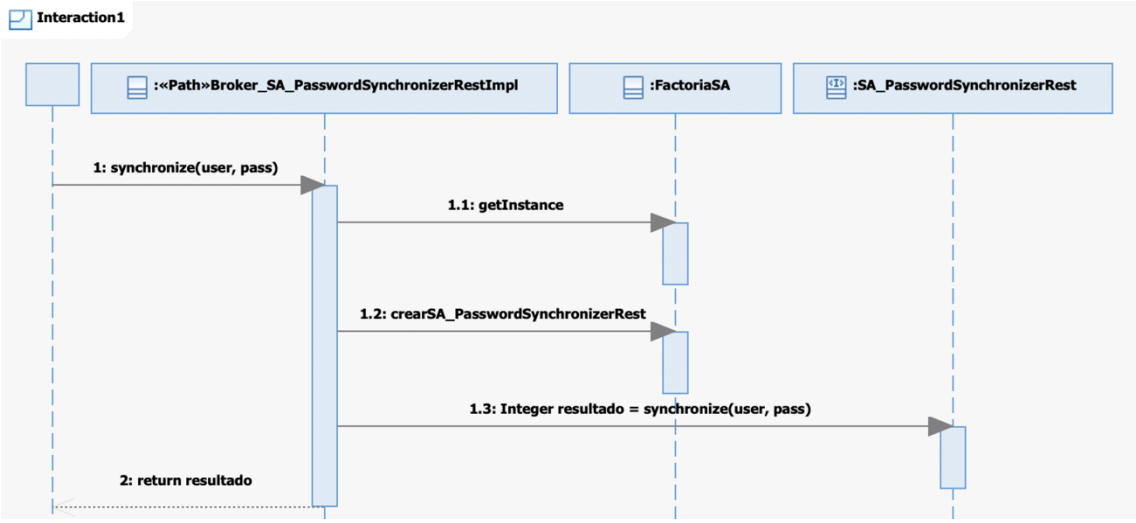

Figura 7.4 - Diagrama de secuencia del patrón *password synchronizer* para el método *synchronize* de la clase *Broker_SA_PasswordSynchronizerRestImpl*

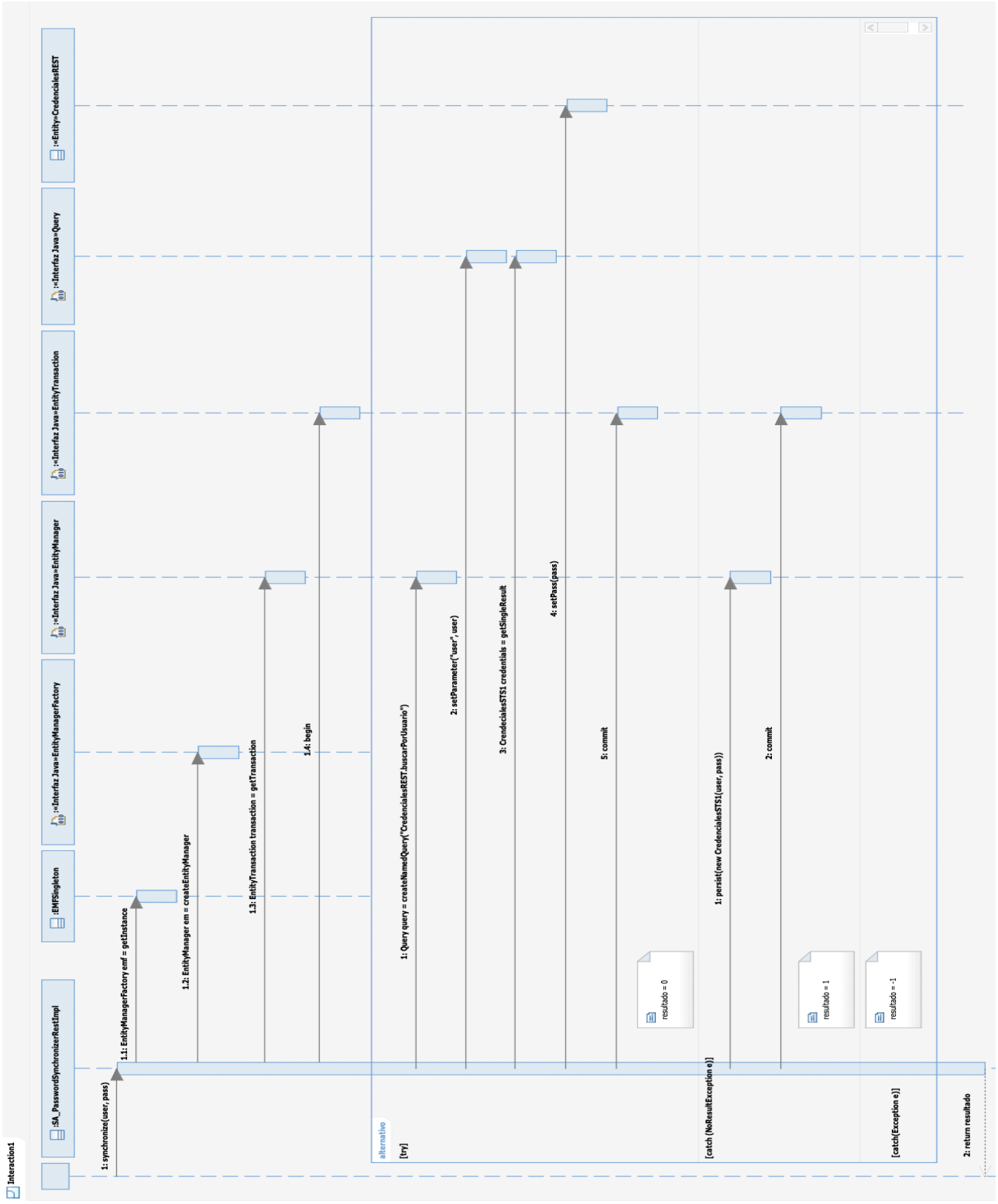

Figura 7.5 - Diagrama de secuencia del patrón *password synchronizer* para el método *synchronize* de la clase *SA_PasswordSynchronizerRestImpl*

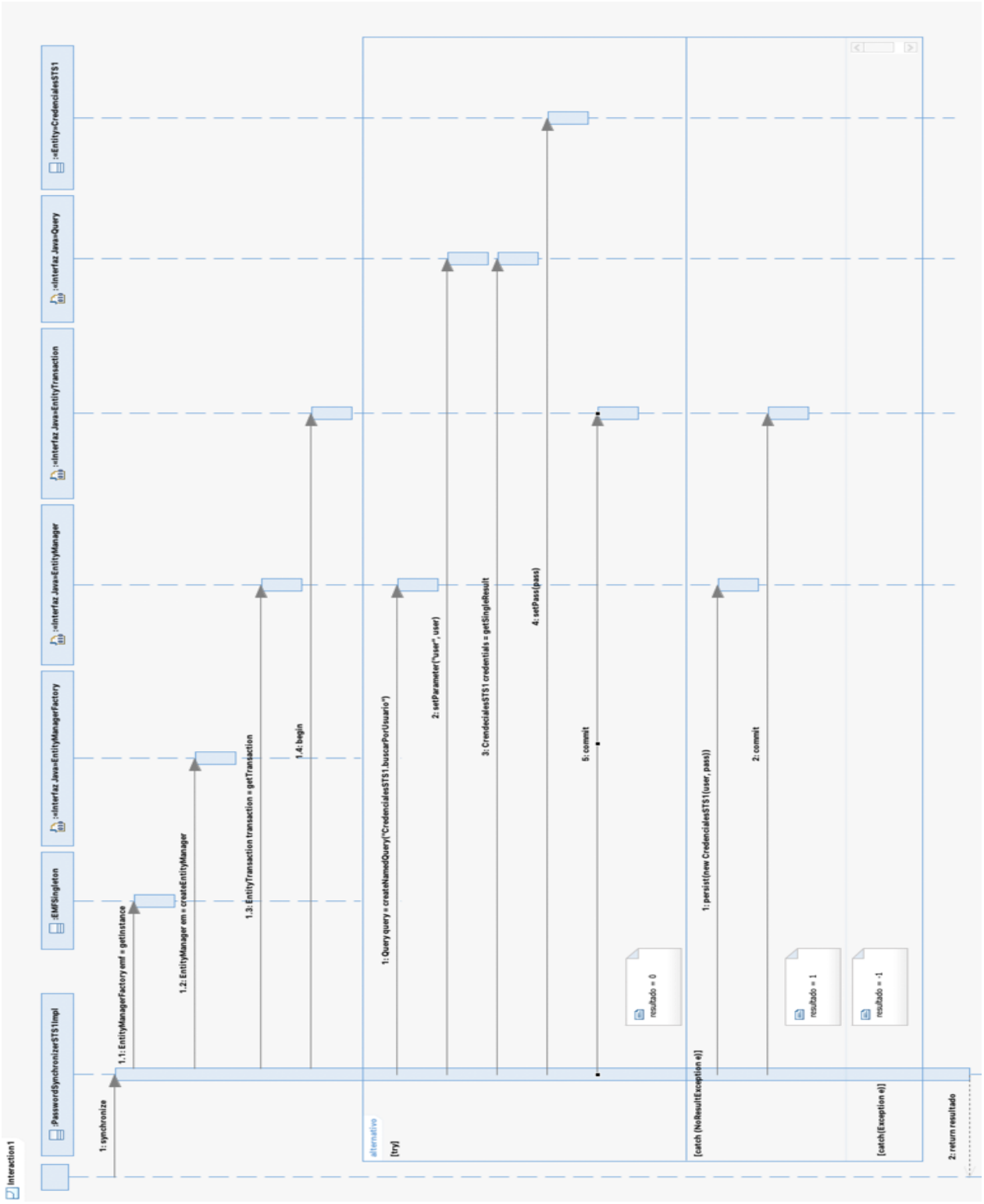

Figura 7.6 - Diagrama de secuencia del patrón *password synchronizer* para el método *synchronize* de la clase *PasswordSynchronizerSTS1Impl*

8. Conclusiones y trabajo futuro

8.1. Conclusiones

El desarrollo de este proyecto ha sido un gran complemento a la formación recibida en el grado, ya que no hay ninguna asignatura en el Grado en Ingeniería del Software en el que se forme a los alumnos en detalle sobre servicios web o seguridad en aplicaciones empresariales.

El desarrollo de este TFG ha supuesto un gran trabajo de investigación, tanto para la aplicación de patrones de seguridad, como para el desarrollo de aplicaciones basadas en servicios web. Cabe destacar el esfuerzo para el uso del protocolo HTTPS, WS-Security, y una multitud de tecnologías como JavaServer Faces (JSF), JavaServer Pages (JSP) o Java Authentication and Authorization Service (JAAS) entre otras. Para entender estas tecnologías, antes de iniciar la implementación sobre el proyecto original, se procedió a realizar un prototipo incremental, aplicando estas tecnologías a partir de manuales y seminarios realizados con el director del TFG, y documentando lo necesario para poder utilizarlo posteriormente en el proyecto.

La exigencia en el proyecto ha estado presente desde el principio, debido a que se trataba de un proyecto de reingeniería. Por suerte, el proceso de reingeniería se aplicó a un proyecto que contaba con multitud de patrones arquitectónicos vistos en el grado, lo que favoreció la comprensión y modificación del mismo. Además, al ser la seguridad un elemento ortogonal a la funcionalidad del proyecto, la inclusión de patrones de seguridad en un proyecto existente ha sido factible respetando en gran medida el código del proyecto original.

De esta forma, se partió de un proyecto cuyos únicos elementos de seguridad eran la autenticación/autorización en la capa de presentación del WSC con JAAS, y la invocación de servicios web mediante usuario y contraseña a través de conexiones HTTPS, y se ha acabado con una aplicación que implementa todos los patrones del catálogo de Steel et al. (2015). Esta implementación, la mayor parte de las veces, ha necesitado el desarrollo de

código específico, y sólo en algunos casos se ha delegado en código existente en marcos y/o plataformas.

Con la realización de este trabajo, se ha podido comprobar que uno de los puntos importantes de la implantación de seguridad en aplicaciones multicapa de tipo empresarial, es la inclusión de registros de actividad (*logs*) en todas las capas descritas en la arquitectura multicapa: las capas de presentación y negocio del WSC y la capa de negocio del WSP. Casos como el de Enron, descrito en el apartado 1.1, ponen de relieve la importancia de guardar un registro de cada operación realizada en todas las capas de la aplicación. Además, es de primordial importancia que la transmisión de estos datos entre la aplicación y el sistema de gestión de base de datos que almacena los registros sea seguro para que estos no sean modificados sin autorización en la conexión. Evidentemente, también es importante almacenarlos de forma segura para que no puedan ser modificado o eliminados sin autorización.

En cuanto a los patrones introducidos en cada capa, se aprecia que los aplicados a la capa de presentación están asociados a la autenticación y autorización para acceder a las diferentes vistas y a la validación de los datos introducidos por el usuario. Los pertenecientes a la capa de negocio se orientan al proceso de aseguración de los distintos servicios web frente a invocaciones no autorizadas, a la transmisión de datos entre los diferentes niveles de la aplicación y la validación e intercepción en el envío de mensajes. En cuanto a la capa de integración, los patrones se centran en asegurar la transmisión de datos a la capa de recursos y en la gestión de su integridad.

Con respecto a las categorías de patrones identificadas por Steel et al. (2005), los comentarios para los patrones de *web* y *business-tier* son asimilables a los de la capa de presentación y de negocio multicapa. Los patrones de la *web service-tier* se centran o bien en garantizar que los mensajes incluidos en los servicios web no contengan información maliciosa (*message interceptor gateway* y *message inspector*), o bien en aumentar las capacidades de autenticación/autorización en la invocación de servicios web más allá del uso de una conexión HTTPS (*secure message router* (SMR)). En este sentido, el patrón SMR ha sido con mucho el patrón más complejo de implementar en este trabajo, a pesar, o precisamente debido a, la presencia al STS de Talend. El STS ha facilitado la

implementación de la autenticación, autorización y generación de *tokens* SAML, pero su uso y puesta en funcionamiento ha sido muy compleja. Además, la implementación del patrón ha incluido a los patrones *single-sign on delegator* (SSOD) y *password synchronizer* de otras capas. Con respecto a los patrones de *identity management*, unos se centran en la gestión de credenciales (*assertion builder* y *credential tokenizer*) y otro en el uso de estas credenciales para la invocación de servicios web en un entorno de *single sign-on* (SSO). Esto explica el porqué de implementar juntos los patrones SMR y SSOD. Finalmente, los patrones de *service provisioning* son los menos numerosos, incluyendo un único patrón centrado en la sincronización de credenciales, que hemos aprovechado para implementar en el contexto de un SMR-SSOD.

Otro de los puntos importantes que se pueden extraer tras la realización de este proyecto es la dependencia de diferentes proveedores en tareas de seguridad. Así, Apache CXF, cuya seguridad depende a su vez del proyecto Apache WSS4J, es uno de los elementos más utilizados en el trabajo por la posibilidad de usar sus interceptores, sus implementaciones de seguridad en servicios web, o para la utilización de WSS4J para extraer la información incluida en los diversos *logs* realizados. También es importante la dependencia de la implementación del STS implementado por Talend. Con independencia de este software, gran parte de la seguridad recae también en el uso de conexiones HTTPS, y en las validaciones de usuario/contraseña enviadas a través de este protocolo, lo cual nos hace dependientes tanto de las facilidades Java para el manejo de certificados X.509, como de las validaciones y encriptaciones llevadas a cabo por Apache Tomcat. Finalmente, los *logs* seguros están basados en conexiones HTTPS a sistemas de gestión de bases de datos relacionales. Por tanto, las empresas deben elegir cuidadosamente a los proveedores de software sobre los que van a construir su infraestructura de seguridad, ya que la seguridad desarrollada es totalmente dependiente de ellos. No obstante, este riesgo y problema real debe ser asumido por las empresas, ya que la implementación de estos servicios desde cero seria extremadamente costoso e inviable.

J2EE es una plataforma de desarrollo abierta, con múltiples implementaciones para su despliegue, lo que es sin duda alguna una ventaja. No obstante, tal y como hemos comentado, al depender la seguridad de las aplicaciones de las implementaciones J2EE

concreta donde son desplegadas, hace que dichas aplicaciones sean dependientes de las implementaciones concretas J2EE, lo que puede condicionar la portabilidad del código J2EE, entre implementaciones concretas. Por tanto, la portabilidad del código J2EE entre distintas implementaciones de la plataforma está muy limitada en el código de seguridad de la aplicación. Así, en el apartado seguridad J2EE, una vez elegidas implementaciones concretas de despliegue como Apache CXF o Apache Tomcat, tiende a volverse una especie de plataforma cerrada como Microsoft .NET.

8.2. Trabajo Futuro

Como trabajo futuro, al ser una aplicación centrada en la seguridad, sería interesante hacer una prueba intensiva sometiendo la aplicación desarrollada a diferentes ataques para comprobar la efectividad de los diferentes patrones.

Puede ser también una tarea futura la resolución de los problemas de descenso del rendimiento generados por el uso de WS-Security, mediante el uso de otras implementaciones o alternativas a esta tecnología.

Otra posibilidad es hacer migraciones a diferentes implementaciones de la plataforma J2EE para comprobar el esfuerzo necesario para adaptar el código relativo a seguridad. Un extremo de esta migración sería traducir toda la aplicación a la plataforma .NET para comprobar las facilidades que proporciona dicha plataforma en el apartado seguridad.

También sería interesante el uso de distintos estándares de seguridad para analizar sus ventajas en inconvenientes frente a los aquí utilizados.

8.3. Trabajo Individual Eduardo Romero Palencia

En primer lugar, hubo que realizar un prototipo para aprender a utilizar las distintas tecnologías utilizadas en el TFG previo puesto que era nuestro proyecto base. Para tal fin, Sergio y yo trabajamos juntos para realizarlo, realizando implementaciones por duplicado para aprender ambos las tecnologías necesarias.

En este prototipo debíamos aprender a utilizar las tecnologías utilizadas por la parte de interfaz de usuario como son JavaServer Faces y JavaServer Pages, por lo que el primer

paso fue leer manuales y distintas páginas web para poder desarrollar la primera parte del prototipo.

A continuación, debíamos aprender a utilizar servicios web, dado que no los habíamos utilizado en ninguna asignatura del grado. Para tal fin, desarrollé dos variantes del prototipo inicial, una con un servicio SOAP y otra con un servicio REST. Cuando estas variantes fueron totalmente funcionales, se utilizó JAAS para asegurar la autenticación y la autorización.

Finalmente, como uno de los primeros objetivos era insertar WS-Security en el proyecto, se procedió a implementarlo en el prototipo.

Cuando el prototipo estuvo terminado, empecé a analizar el TFG previo para poder desplegar su aplicación, lo cual fue complicado, pero fue posible gracias a la documentación. A continuación, con las notas obtenidas por la realización del prototipo, procedí a la implantación en el proyecto de WS-Security, aprendiendo en el proceso como modificar más en profundidad la configuración de ApacheTomcat y cómo utilizar Apache CXF en un proyecto ya implementado.

A continuación, procedimos a implementar el conjunto de patrones *Web Tier*. En primer lugar, desarrolle el filtro SQL del patrón *intercepting filter* para evitar la inyección de sentencias SQL en campos de formularios, para luego realizar la configuración necesaria para desplegarlos como filtros Apache CXF. A continuación, creé los certificados y realicé las modificaciones necesarias en MySQL Server para que permitiese conexiones por SSL/TLS poder utilizar el patrón *secure pipe*. Por último, en este primer paquete, implementé el *secure service proxy*, para el cual tuve que crear desde cero un nuevo servicio web SOAP que actuaría como *secure service proxy*, lanzando un servicio REST ya existente sin variar su configuración.

A continuación, pasamos a implementar la parte de patrones de la categoría *business tier*, en la que en primer lugar desarrolle parte del *audit interceptor*, que mediante interceptores capturaba el mensaje recibido y guardaba un registro de diversos datos. Después pasé a implementar el patrón *dynamic service management*, para lo cual tuve primero que leer diversa documentación de Java para aprender a utilizar su herramienta JConsole, y como

registrar las *beans* para que se pudieran ver en dicha herramienta. Por último, cree el log utilizado en el *policy delegate* que registra el servicio invocado.

En siguiente lugar, procedimos a insertar en el proyecto los patrones de la categoría *web services*. En primer lugar, implementé el *message inspector*, con la funcionalidad correcta, pero con un concepto erróneo del patrón, por lo que después hubo que modificarlo. Después continúe con la implementación del patrón *secure message router* (SMR), que era el más importante en cuanto a dificultad. En primer lugar, comenzamos investigando la posibilidad de implementarlo con un *identity provider*, por lo que comencé a investigar e intentar implementarlo utilizando OneLogin (OneLogin, 2019) con este propósito. A las tres semanas, y ante la imposibilidad de implementarlo a pesar de modificar configuraciones tanto de Apache Tomcat, como de Apache CXF, nos dimos cuenta de que la idea de usar un *identity provider* web no era viable, por lo que procedí a estudiar cómo implementarlo con un *Security Token Service* (STS).

Para implementar el SMR con un STS en primer lugar procedí a crear un nuevo servicio web SOAP, usando anotaciones en lugar de generar el archivo WSDL a mano para aprender a crearlo de una manera alternativa. A continuación, empecé a leer documentación sobre XACML para entender los archivos de políticas que hacían que el servicio requiriese el *token* para permitir el acceso, haciendo así funcionar el patrón. Por último, generé los certificados necesarios para la autenticación entre los diferentes servicios y modifiqué la configuración de los STS para que funcionaran correctamente.

Por último, pase a implementar los patrones las categorías *identity management* y *service provisioning*. En primer lugar, implemente el *credential tokenizer*, para después utilizarlo dentro de *single sign-on delegator*, para el cual cree un servicio web SOAP y otro REST, y configure otro STS. Estos nuevos servicios web pasaron a ser el patrón *password synchronizer*, el cual realizaba una actualización en cascada de unas tablas de contraseñas de una base de datos.

En la parte final del proyecto, Sergio y yo acordamos que él realizaría los diagramas de clases, secuencia y despliegue, por lo que únicamente he servido de apoyo cuando tenía dudas sobre implementaciones o sobre los conceptos de los patrones. En última instancia,

en la memoria, he hecho la introducción y las conclusiones en ambos idiomas, el primer grupo de patrones de seguridad, y la búsqueda y anotación de referencias.

8.4. Trabajo Individual Sergio Martín Gómez

La primera parte del proyecto consistía en aprender a utilizar tecnologías que no habíamos aprendido durante el grado, por lo tanto, en esta primera parte trabajamos en paralelo haciendo los dos miembros del equipo el trabajo por duplicado para estar los dos familiarizados con todas las tecnologías.

El primer paso para el desarrollo de este proyecto fue aprender todas las tecnologías utilizadas en el TFG previo para poder modificarlo posteriormente. Además, debido a la falta de formación en servicios web en el grado, también fue necesario aprender a crear servicios web con Java. Para el aprendizaje de todas las tecnologías desarrollamos un prototipo simple que tenía los dos tipos de servicios web que íbamos a necesitar, uno SOAP y uno REST, además de usar las tecnologías de la capa de presentación: JAAS para la autenticación y autorización y JSF para la vista. También fue necesario aprender WS-Security y el estándar SAML, necesarios para poder implementar algunos patrones de nuestro proyecto. La función de los dos servicios web era sumar dos números, no necesitábamos que realizasen una tarea más compleja porque el único propósito del prototipo era aprender a utilizar los marcos que usamos durante todo el proyecto.

Tras tener el prototipo funcionando comenzó la fase de reingeniería, que consistió en hacer funcionar el código de del TFG previo con toda la información que nos proporcionaba sobre cómo desplegar la aplicación. Disponíamos de los diagramas de clases, de secuencia y despliegue de la aplicación previa incluidos en la memoria del TFG, que no cubrían a la aplicación en su totalidad. Sin embargo, por ser una aplicación realizada según la arquitectura multicapa, pudimos empezar a modificarla con facilidad. Para poder ejecutarla utilizamos toda la información que proporcionada en la memoria del TFG y en la documentación del repositorio GitHub del proyecto. En esta fase también tuvimos que aprender a utilizar un contenedor de *servlets* y desplegar aplicaciones en él, en nuestro caso Apache Tomcat.

Tras estas dos primeras fases de aprendizaje comenzó nuestro proyecto, en el que empezamos a implementar los patrones de seguridad. En esta fase la división del trabajo no fue acordada, utilizamos Trello (Atlassian, 2011) para organizar las tareas del trabajo y cada miembro del equipo realizaba las tareas que había pendientes a su ritmo. El libro de patrones tiene un catálogo de patrones que se divide en varias capas, a continuación, describo los patrones que he implementado en cada capa.

De la capa *web tier* he implementado uno de los filtros del *patrón intercepting filter*, que consistía en validar que ninguno de los parámetros de la petición realizada del cliente al servidor contenga caracteres extraños tales como *, # u otros caracteres que pudieran comprometer la seguridad del servidor. De esta capa también he implementado el patrón *secure logger*, que consiste en hacer un log de la operación realizada y de quién la hizo desde los *beans* de presentación. El resto de los patrones de esta capa fueron implementados por Eduardo o están delegados en los marcos que hemos utilizado.

La siguiente capa del catálogo de patrones es *business tier*. En esta capa he implementado parte del patrón *audit interceptor*, que consiste en un interceptor que guarda un registro de la IP, el usuario, la contraseña y el servicio invocado. Además, también he implementado el patrón *obfuscated transfer object*, que consiste en enviar al servidor la información de un transfer con varios campos encriptados, para que en el caso de que la petición sea interceptada no se pueda ver información confidencial. De nuevo, el resto de los patrones de esta capa han sido implementados por Eduardo o están implementados por los marcos utilizados.

El siguiente grupo de patrones son los patrones *web services*. De este grupo Eduardo implementó el patrón *message inspector*. Sin embargo, Eduardo implementó los dos interceptores en una sola clase, y la idea era tener dos interceptores distintos, uno para comprobar la IP y otro para comprobar el usuario y la contraseña. Yo me encargué de la modificación del patrón para dividirlo en dos interceptores distintos. El siguiente patrón es el *secure message router*, cuya mayor dificultad era tener un proveedor de identidad funcionando para validar los *tokens* SAML. Tras investigar el uso de OneLogin durante tres semanas y darnos cuenta de que no era lo que necesitábamos, utilizamos el *security token service* de Talend. La mayor parte del trabajo de este patrón la realizó Eduardo, mi

aportación a este patrón fue crear la clase que invocase a todos los servicios web que involucraban a este servicio web.

Debido a que Eduardo realizó un esfuerzo mucho mayor durante el desarrollo del código, especialmente en el patrón *secure message router*, que ha sido el más complejo y el que más tiempo ha llevado, acordamos que yo realizaría todos los diagramas del trabajo. Por lo tanto, una vez acabado el código, empecé a realizar los diagramas de clases, secuencia y despliegue de cada patrón del proyecto. En esta parte del proyecto ha sido necesario aprender a realizar diagramas de despliegue para servicios web, algo que no habíamos visto durante el grado. Durante el grado, en las asignaturas de Ingeniería del Software y Modelado de Software, pusieron el foco en la realización de diagramas de clases y de secuencia, prestando menor atención a los diagramas de despliegue, por lo que una parte importante de esta fase del proyecto ha sido el aprendizaje de los diagramas de despliegue para aplicaciones empresariales tan complejas como las que usan un SMR.

Para el desarrollo de la memoria Eduardo y yo nos dividimos el trabajo por catálogos de patrones. Además de hacer el resumen de la memoria, he descrito los catálogos de patrones *business tier*, *web services*, *identity management* y *service provisioning*.

8. Conclusions

8.1. Conclusions

The development of this project was a great complement to the training received in the bachelor, because there is no any subject in the Software Engineering grade in which the students were formed in detail about web services or security in business applications.

The development have had also a significant research work, for the application of the security patterns, and for the development of web service-based applications. It is worth mentioning the effort made for using the HTTPS protocol, WS-Security, and a lot of technologies such as JavaServer Faces (JSF), JavaServer Pages (JSP) or Java Authentication and Authorization Service (JAAS) among others. For understanding these technologies, before starting the implementation over the original project, an incremental prototype was made, applying these technologies from manuals and seminaries realized with the project director, and documenting everything that was needed to use later in the project.

The project's thoroughness was present in the beginning, because it was a reengineering project. Luckily, the reengineering process was applied to a project that had lots of architectonical patterns learnt in the grade, which favored its comprehension. In addition, because security is an orthogonal element to projects functionality, the inclusion of security patterns into an existent project was feasible using the original project code largely.

This way, the original project only had user authentication/authorization in the presentation tier of the WSC using JAAS, and web service invocation with user and password under HTTPS, while the resulting project implements all the patterns of the Steel et al. (2015) catalog. This implementation usually required the development of new code, and only in a couple of cases the functionality was delegated in existing frameworks or platforms.

With the realization of this work, it had been probed that one of the important points of the security in multitier business applications, is the inclusion of logs in every tier of the multitier architecture: the presentation and business tiers of the WSC and the business tier of the WSP. Cases as the Enron´s one, described in Section 1.1, shows the importance of

saving logs of every operation made in every tier of the application. Secure data transmission between the application and the database management system is a key issue also. Of course, to store information in a secure way for protecting it for being modified or destroyed without authorization is a key issue also.

Regarding to the patterns introduced in each tier, it can be appreciated that the ones applied to the presentation tier are associated to the authentication and the authorization to access to the different views and to the validation of the data introduced by the users. The ones associated to the business tier are oriented to prevent different web services against non-authorized invocations, to secure data transmission between the different levels of the application and to validate validation and interception in the message exchange. Finally, in the integration tier, patterns are focused on securing data transmission to the resources tier and on the integrity management.

Regarding to the pattern categories identified by Steel et al. (2005), the comments for the web and business tier patterns are applicable to those of the presentation and business tier from the multitier architecture. The patterns of the web service tier are focused on filtering malicious information included in the service invocation (*message interceptor gateway* and *message inspector*), or on enhancing the authorization/authentication capabilities in the web service invocation besides the use of a HTTPS connection (*secure message router* (SMR)). In this matter, the SMR pattern was the most complex one of the ones implemented in this project, even the presence of the Talend´s STS. The STS made the implementation of authentication, authorization and SAML token generation easier, but its use and deployment were very complex. In addition, the implementation of the SMR pattern has included the *single sign-on delegator* (SSOD) and *password synchronizer* patterns belonging to other tiers. In the identity management category, some patterns were focused on credential management (*assertion builder* and *credential tokenizer*) and others on the use of these credentials for the web services invocation in a *single sign-on* (SSO). This explains why the SMR and SSOD were implemented together. Finally, the service provisioning pattern are the least numerous, including just one pattern focused on the credential synchronizer, that was used to be implemented in an SMR-SSOD context.

Another important consequence obtained after the realization of this project is the dependency of different providers in security matters. This way, Apache CXF, which security depends of the Apache WSS4J (Apache Software Foundation, 2008) project, is one of the more used element in the project for the possibility of using interceptors, security implementations in web services, or for the use of WSS4J to extract the information included in different logs. Also is important the dependency the implementation of the STS made by Talend. Regardless of this software, great part of the security depends on the use of HTTPS connections, and on the user/password validation sent by that protocol, that create a dependency on the Java facilities that manage X.509 certificates, and on validations and encryptions made by Apache Tomcat. Finally, the secure logs are based on HTTPS connections to relational databases management systems. Thus, companies must choose carefully the software providers on which are going to base their security infrastructure, because the security developed is highly dependent on them. However, this risk must be accepted by the companies, because the implementation of this services from the scratch would be very expensive and non-affordable.

J2EE is an open development platform, with multiple implementations for the deployment, which is an advantage. However, the applications' security dependency on concrete J2EE implementations, could condition the J2EE code portability between concrete implementations. Thus, the portability of the J2EE code is very limited regarding security aspects of the application. This way, regarding security, J2EE platform, once deployment implementations such as Apache CXF or Apache Tomcat are chosen, tends to turn into a kind of closed platform like Microsoft .NET.

8.2. Future Work

As future work, it would be interesting to perform different attacks against the developed application in order to prove the effectiveness of the implemented patterns.

The resolution of the performance decrease problems generated due to the use WS-Security using other implementations or alternatives to this technology could be also very interesting.

The application transfer to different implementations of the J2EE platform, in order to test the effort needed to adapt the code relative to security, is another interesting issue. An extreme transfer would be translating all the application to the .NET platform to test the facilities that gives that platform in the security subject.

Finally, it would be interesting using different security patterns to analyze the advantages against which are used in this project.

8.3. Individual Work Done by Eduardo Romero Palencia

In first place, a prototype had to be made to learn to use the different technologies used in the previous project because it was our base project. For this, Sergio and I worked together to make it, doing separated implementation to learn the necessary technologies.

In that prototype we had to learn to use the technologies used in the user interface part as JavaServer Faces and JavaServer Pages, so the first step was reading manuals and web pages to start developing the first part of the prototype.

Next, we must learn to use web services, because we have not used them in any subject in the grade. For that, I developed two variants of the initial prototype, one with a SOAP service and the other with a REST service. Once those variants were totally functional, JAAS was used to secure authentication and authorization.

Finally, as one of the first objectives was insert WS-Security in the project, this technology was used in the prototype.

When the prototype was finished, I started to analyze the previous project for being able to deploy the applications, which wasn´t easy, but it was possible thanks to the documentation. Next, with the notes of the prototype, I proceed to the implantation of WS-Security in the project, learning in the process how to modify more deeply the Apache Tomcat configuration and how to use Apache CXF in an already implemented project.

Then, we proceeded to implement the Web-Tier patterns group. In first place, I have developed the SQL filter of the *intercepting filter* pattern to avoid the SQL injection in forms, and then I have made the necessary configuration to deploy them as Apache CXF filters. Next, I have created the certificates and made the necessary modifications in

MySQL Server to allow connections by SSL/TLS to use the *secure pipe* pattern. At last, in this first package, I have implemented the *secure service proxy*, in which I have made a new SOAP web service that acts as the *secure service proxy*, launching an existing REST service without changing the configuration.

Following this, we started to implement the patterns of the Business Tier category, in which in first place I have developed part of the *audit interceptor*, which captures received messages and stores logs with different data. Then, I started to develop the *dynamic service management* pattern. Thus, I started to read Java documentation to learn to use the JConsole tool, and how to register *beans* to be able to watch them with that tool. Finally, I have created the log used in the *policy delegate* that register the invocated service.

Then, we proceeded to insert the Web Services group patterns. In first place, I have implemented the *message inspector*, with the correct functionality, but with a wrong concept of the pattern, so later we had to modify it. Then I have continued with the implementation of the *secure message router*, which was the most difficult of them. In first place, we started investigating the possibility of implementing it with an *identity provider*. Thus, I tried to implement it using OneLogin (OneLogin, 2019). Three weeks later and realizing the impossibility of implementing it even modifying configurations from Apache Tomcat and from Apache CXF, we accepted that the use an *identity provider* wasn´t viable, so I began to study how to implement the pattern with a *Security Token Service* (STS).

To implement the SMR with an STS I started creating a new SOAP web service, using annotations instead of generating the WSDL archive to learning an alternate way of creating. Next, I started to read documentation about XACML to understand the policies files that make the service require the token to allow the access, making the pattern working. Finally, I have generated the necessary certificates to the authentication between the different services and I have modified the STS configuration to make it work.

Finally, I have implemented the patterns of the Identity Management and Service Provisioning. In first place, I have implemented the *credential tokenizer*, for using it later inside the *single sign-on delegator*, for which I have created a SOAP web service and a

REST web service. These new web services were the *password synchronizer* pattern, which makes an update in cascade of some password in different databases.

In the final part of the project, Sergio and I agreed that he would make the class, sequence and deployment diagrams, so I have just been supporting him when he has doubts about implementations or about the pattern concepts. Finally, in this document, I have made the introduction, and the conclusions in both languages, the first group of security patterns, and the references section.

8.4. Individual Work Done by Sergio Martín Gómez

The first part of the project was focused on learning to use technologies that we had not learnt during the grade. Thus, in this first part we have worked in parallel making both members of the team duplicate work for being both familiar with all the technologies.

The first step for developing this project was learning all the technologies used in the previous project to be able to modify it later. Due to the lack of training in web services in the grade, it was also necessary to learn to create Java web services. In order to learn all the technologies that we needed, we developed a simple prototype that included both types of web services that we were going to need, one SOAP and one REST, as well as the technologies of the presentation tier: JAAS for the authentication and authorization and JSF for the view. It was also necessary to learn WS-Security and SAML, needed to implement some patterns in our project. The function of the web services was adding two numbers, we didn't need a more complex work because the one purpose of the prototype was learnt to use the frameworks that we have use during all the project.

Before having the prototype working the reengineering phase started, that was about making work the code of the previous project with all the information about how deploy the application. We had the class, sequence and deployment diagrams of the previous application included in this document, but they didn´t cover all the application. However, thanks to the multitier architecture we could start to modify it easily. To be able to execute the original project we used all the information given in its document and in the GitHub documentation of the project. In this phase, we also had to learn to use a servlet container and to deploy application on it, in our case Apache Tomcat.

After these two first learning phases, our project started. In this new phase, the work division wasn´t scheduled, and we used Trello (Atlassian, 2011) to organize the task and each member of the team realized the pending tasks.

The pattern book has a pattern catalog divided in multiple tiers. Therefore, I am going to describe the patterns that I have implemented in each tier.

In the web tier I have implemented one of the filters of the *intercepting filter* pattern, that validates that none of the parameters of the request made to the server contains strange characters such as *, # or other characters that could compromise the security. In this tier I have also implemented the *secure logger* pattern, which makes a log of the operation and the user who have executed it from the presentation beans. The rest of the patterns were implemented by Eduardo or were delegated in the used frameworks.

The next tier of the catalog is the business tier. In this tier I have implemented part of the *audit interceptor* patter, that consist in an interceptor that stores an IP, user, password and invoiced service log. I have also implemented the *obfuscated transfer object* that sends to the server information of a transfer with a couple of encrypted fields. Thus if the request is intercepted, the confidential information couldn´t be seen. Again, the rest of the patterns of this tier was implemented by Eduardo or were implemented in the used frameworks.

The next pattern group are the web service ones. In this group, Eduardo implemented the *message inspector* pattern. However, Eduardo implemented both interceptors in one class, and the idea was to have two different interceptors, one to test the IP and the other to test the user and password. I oversaw the modification of this pattern. The next pattern was the *secure massage router*, which biggest difficult was to have an identity provider working to validate the SAML tokens. After doing a research about the use of OneLogin for three week and realizing that it wasn´t what we needed, we used the Security Token Service from Talend. The biggest part of the job in this pattern was made by Eduardo. My contribution to this pattern was the creation of the class that orchestrates all the web services used in this use case.

Because Eduardo had made a bigger effort during the code development, especially in the *secure message router* patter, which was the most complex, we agreed that I was going to

do all the diagrams of the project. Thus, when the code was over, I started to make the class, sequence and deployment diagrams for each pattern in the project. In this phase, it was necessary to learn to do deployment diagrams for web services, which was something unseen in the grade. During the grade, the subjects of Software Engineering and Software Modeling put the focus on UML class and sequence diagrams, paying less attention to the deployment diagrams. Thus, an important part of this phase was to learn how to use them for business application as complex as the SMR ones.

For the development of this document, Eduardo and I divided the work between the pattern catalogs. I have done the abstract, and I have described the business tier, web service, identity management and services provisioning groups of patterns.

9. Referencias

Alur et al., 2003 - Deepak Alur, John Crupi, Dan Malks. *Core J2EE Patterns. Best Practices and Design Strategies*. Prentice Hall, 2003.

Apache Software Foundation, 2008 - Apache CXF, 2008. http://cxf.apache.org/

Apache Software Foundation, 2018 - Apache Tomcat, 2018. https://tomcat.apache.org/

Apache, 2000 – Apache Subversion, 2000. https://subversion.apache.org/

Apache, 2011 – Apache ServiceMix, 2011. http://servicemix.apache.org/

Atlassian, 2011 – Trello, 2011. https://trello.com/

Beck & Andres, 2014 - Kent Beck, Cynthia Andres. *Extreme Programming Explained: Embrace Change. Second Edition*. Addison-Wesley, 2004.

Burke, 2013 - Bill Burke. *RESTful Java with JAX-RS 2.0: Designing and Developing Distributed Web Services*. O'Reilly, 2013.

Coté, 2009 - Michael Coté. *JAAS in Action*. 2009.

Da Silva, 2013 - Anayansi Da Silva de la Cruz, Antonio Navarro Martín. *Una Aproximación MDA para la Conversión entre Servicios Web SOAP y RESTful*. Universidad Complutense de Madrid, 2013.

De Miguel, 2018 - Antonio Navarro Martín, Rodrigo de Miguel González. *Atravesando las Capas de una Aplicación Empresarial: Demostrador Tecnológico J2EE*. Universidad Complutense de Madrid, 2018.

Fernandez-Buglioni, 2013 - Eduardo Fernández-Buglioni. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley, 2013.

Fielding, 2000 – Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, 2000.

Fowler, 2002 - Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

Gamma, Helm, Johnson, Vlissides, 1994 - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Geary, 2006 - David Geary, Cay S. Horstmann. *Core JavaServer Faces*. Prentice Hall, 2006.

GitHub Inc., 2008 – GitHub, 2008. https://github.com/

Hall, 2003 - Marty Hall. *Core Servlets and JavaServer Pages*. Prentice Hall, 2003.

Hansen, 2007 - Mark D. Hansen. *SOA Using Java Web Services*. Prentice Hall, 2007.

IBM Eclipse, 2018 - IBM Eclipse Photon, 2018. https://www.eclipse.org/photon/

IBM, 2018 - The high cost of (WS-)Security, 2009. https://www.ibm.com/developerworks/library/j-jws6/index.html

IBM RSA, 2018 - IBM Rational Software Architect Designer, v9.6.0 product documentation, 2018. https://www.ibm.com/support/knowledgecenter/SS8PJ7_9.6.0/com.ibm.xtools.rsa_base.legal.doc/helpindex_rsa_base.html

ITU-T, 1988 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, 1988. https://tools.ietf.org/html/rfc5280

Java67, 2012 - Difference between trustStore vs keyStore in Java SSL, 2012, http://www.java67.com/2012/12/difference-between-truststore-vs.html

Little, 2004 – Mark Little, Jon Maron, Greg Pavlik. *Java Transaction Processing: Design and implementation*. Prentice Hall, 2004

Mazza, 2017 - Deploying and Using a CXF Security Token Service (STS), 2017. https://glenmazza.net/blog/entry/cxf-sts-tutorial

Microsoft, 2002 – Microsoft .NET, 2002. https://dotnet.microsoft.com/

Monk & Wagner, 2012 – Ellen Monk, Bret Wagner. *Concepts in Enterprise Resource Planning*. Cengage, 2012

OASIS, 2002 - OASIS Web Services Security (WSS) TC, 2002. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss

OASIS, 2003 – Service Provisioning Markup Language (SPML) v1.0, [OASIS 200306], 2003. https://www.oasis-open.org/standards#spmlv2.0

OASIS, 2005 - OASIS Security Services (SAML) TC, 2005. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security

OASIS, 2013 - eXtensible Access Control Markup Language (XACML)Version 3.0, 2013. https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs-01-en.pdf

OASIS Standard, 2005 - Profiles for the OASIS Security Assertion Markup Language (SAML)V2.0, 2005. https://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf

OneLogin, 2019 - OneLogin, 2019. https://developers.onelogin.com/

Oracle, 2000 - Java™ EE at a Glance, 2000. https://www.oracle.com/technetwork/java/javaee/overview/index.html

Oracle, 2003 – Enterprise JavaBeans, 2003. https://www.oracle.com/technetwork/java/docs-135218.html

Oracle, 2005 - Java Management Extensions (JMX) Technology, 2005. https://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

Oracle, 2011 - Oracle Service Bus Architecture, 2011. http://docs.oracle.com/cd/E14571_01/doc.1111/e15020/architecture_overview.htm

Oracle, 2013 – Java(TM) EE 7 Specification APIs (javax.persistence), 2013. https://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html

Oracle, 2017 - JSR 369: JavaTM Servlet 4.0 Specification, 2017. https://jcp.org/en/jsr/detail?id=369

Oracle, 2018 - MySQL Server 8.0, 2018. https://dev.mysql.com/

Oracle, 2019 - Java™ Platform, Standard Edition 7 API Specification (java.security.Principal), 2019. https://docs.oracle.com/javase/7/docs/api/java/security/Principal.html

Parra, 2004 - Wilmer Eduardo Parra Valdés, Antonio Navarro Martín. *Integración de Patrones de Seguridad y Patrones de Diseño J2EE.* Universidad Complutense de Madrid, 2004.

Pressman, 2001 - Roger S. Pressman, Bruce Maxim. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2001.

Ryan, 2011 - Rethinking the ESB: Building a simple, secure, scalable Service Bus with an SOA Gateway, 2011. http://www.computerworld.com/s/article/9219205/Rethinking_the_ESB_Building_a_simple_secure_scalable_Service_Bus_with_an_SOA_Gateway

Steel et al., 2015 - Christopher Steel, Ramesh Nagappan, Ray Lai. *Core Security Patterns; Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall, 2006.

System, 2008 - Best Practices in Deploying XML SOA Gateways, 2008. http://es.scribd.com/doc/43981774/Best-Practices-in-DeployingXML-SOAGateways#

Vordel, 2011 – Vordel, 2011. http://www.vordel.com/research/

W3C, 2000 - SOAP Version 1.2, 2000. https://www.w3.org/TR/soap/

Wikipedia WS-Security, 2019 - https://en.wikipedia.org/wiki/WS-Security

Wulff, 2011 - Configure and deploy CXF 2.5 STS - Part I, 2011. http://owulff.blogspot.com/2011/10/configure-and-deploy-cxf-25-sts-part-i.html

Wulff, 2012 - SAML tokens and WS-Trust Security Token Service (STS), 2012 http://owulff.blogspot.com/2012/02/saml-tokens-and-ws-trust-security-token.html

Talend, 2019 – Talend, 2019. https://www.talend.com