INTEGRACIÓN DE PATRONES DE SEGURIDAD Y PATRONES DE DISEÑO J2EE

WILMER EDUARDO PARRA VALDÉS

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería del Software e Inteligencia Artificial

Director: Antonio Navarro

Calificación obtenida: Notable (8.5)

Madrid, 23 de junio de 2014

A las personas que me dieron la vida, Julio Ernesto y Emilia

Y a mis hermanos Diego, Fernando y Laura

Autorización de difusión y utilización

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "INTEGRACIÓN DE PATRONES DE SEGURIDAD Y PATRONES DE DISEÑO J2EE", realizado durante el curso académico 2013-2014 bajo la dirección de Antonio Navarro, en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Wilmer Eduardo Parra Valdés
Madrid, 23 de junio de 2014

Agradecimientos

Agradezco a la Facultad de Informática de la Universidad Complutense de Madrid por haberme acogido para realizar este máster en investigación, por haberme abierto sus aulas y poner a mi disposición un magnifico grupo de profesores y recursos necesarios para poder cumplir mis objetivos y crecer como profesional, y además por permitirme sentirme participe al aportar un granito de arena a este gran trabajo de la investigación.

Quiero dar un especial agradecimiento al profesor Antonio Navarro. Considero que ha sido un buen director de proyecto, pero también el mejor compañero de equipo para realizar este trabajo de investigación. Gracias por su dedicación en cada una de las tutorías y seguimiento, por los consejos y correcciones en la metodología y también por su aportación profesional y experiencia.

A todas las personas que durante este máster han sido participes de este trabajo, profesores y compañeros de clase, grupos de investigación, amigos y compañeros de trabajo, que han colaborado para que este nuevo estudio sea una experiencia enriquecedora en lo personal y profesional.

Agradezco también a mi familia por la comprensión y apoyo que han demostrado todo el tiempo que he dedicado a este máster, sé que en ciertos momentos no hemos podido compartir todo el tiempo que nos gustaría.

Índice de contenidos

Índice de figuras

Lista de acrónimos

CSP	Core Security Patterns
CJP	Core J2EE Pattern
J2EE	Java Enterprise Edition
JSF	Java Server Faces
API	Application Programming Interface
JAX-WS	Java API for XML Web Services
JAX- RS	Java API for RESTful Web Services
JPA	Java Persistence API
ACM	Association for Computing Machinery
IEEE CS	IEEE Computer Society
OOPSLA	Object-Oriented Programming, Systems, Languages & Applications
SOA	Service Oriented Architecture
JSP	Java Server Pages
EJB	Enterprise Java Beans
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JDBC	Java Database Connectivity
LOB	Large Objects
BLOB	Binary Large Objects
CLOB	Character Large Object
XML	eXtensible Markup Language
SAML	Security Assertion Markup Language
SOAP	Simple Object Access Protocol
TLS	Transport Layer Security
VPN	Virtual Private Network
IPSec	Internet Protocol security
IDS	Intrusion Detection System
XACML	eXtensible Access Control Markup Language

W3C	World Wide Web Consortium
VoIP	Voice over IP
UML	Unified Modeling Language
JAAS	Java Authentication and Authorization Service

Resumen en castellano

El presente trabajo analiza dos de los catálogos de patrones de diseño de software más populares que provienen de la experiencia de Sun Microsystems (actualmente Oracle Corporation), define las relaciones existentes entre ellos para proporcionar un uso coherente de ambos catálogos y presenta el modelado de dos casos prácticos de aplicaciones que integran patrones de los dos catálogos analizados.

Los patrones pertenecientes al Catálogo Core Security Patterns (CSP) (Steel et al. 2005) se han analizado de forma individual y se han interpretado en el contexto Core J2EE Pattern Catalogue (CJP) (Alur et al. 2003). En este estudio se ha considerado: (i) la interpretación de los patrones de CSP en términos de arquitectura multicapa (es decir, el catálogo de patrones CJP); (ii) la información adicional incluida en el catálogo del CSP para facilitar su comprensión; (iii) los requisitos previos CJP para entender los patrones de CSP; y (iv) los requisitos previos CSP para la comprensión del patrón CSP analizado.

Los resultados de este trabajo tienen una doble aplicabilidad. Desde un punto de vista industrial, este trabajo facilita el uso de patrones de seguridad en el contexto de una arquitectura multicapa. Desde un punto de vista educativo, este trabajo establece una relación de precedencia entre los patrones multicapa y los de seguridad, y define además un subconjunto de patrones multicapa como el mínimo necesario para comprender los patrones de seguridad. De esta forma se facilita la docencia de los patrones CSP.

Palabras clave

Patrones de diseño, patrones de seguridad, lenguajes de patrones, J2EE, arquitectura de software, arquitectura multicapa.

Resumen en inglés

This paper briefly analyses two of the most popular software design pattern catalogues based on experience at Sun Microsystems (currently Oracle Corporation), draws relations between them to provide a cohesive use of both catalogues and implements two case studies of applications that integrate patterns of the two catalogues analysed.

Patterns belonging to the Core Security Patterns Catalogue (CSP) have been individually analysed and interpreted in the context of the Core J2EE Pattern Catalogue (CJP), highlighting: (i) the interpretation of CSP patterns in terms of multitier architecture (i.e. CJP patterns); (ii) the additional information included in the CSP catalogue; (iii) CJP prerequisites for understanding CSP patterns; and (iv) CSP prerequisites for understanding the CSP analysed pattern.

The results apply to two areas. From an industrial point of view, the use of security patterns is made easier because their integration in a multitier architecture is facilitated by the work presented in this paper. From an educational point of view, a relationship of precedence between multitier and security patterns is established, and a subset of multitier patterns is proposed as the minimum needed to understand security patterns.

Keywords

Design patterns, patterns Security, pattern language, J2EE, software architecture, multitier architecture.

1 Introducción

El desarrollo de una arquitectura empresarial se ha convertido en una cuestión compleja en la actualidad (Fowler et al. 2003). Una de las cuestiones más importantes que los desarrolladores de software tiene que enfrentar es elegir la arquitectura de software correcta. Con la llegada de arquitecturas empresariales distribuidas, que hacen uso de los servicios expuestos a través de la web, la arquitectura multicapa parece ser una de la más comúnmente aceptada por la industria (Alur et al. 2003, Fowler et al. 2003).

Esta arquitectura separa las aplicaciones de software en cinco niveles:

- La capa cliente, que identifica a los actores utilizando la aplicación.
- La capa de presentación, que proporciona una interfaz de usuario para los actores.
- La capa de lógica de negocio, que pone en práctica las normas de los procesos de negocio del dominio.
- La capa de integración, que proporciona acceso a los recursos externos de la aplicación.
- La capa de recursos, bases de datos y/u otros sistemas que almacenan datos persistentes de las aplicaciones.

Estos niveles se basan en un conjunto de patrones de software de diseño arquitectónico que definen los elementos de software pertenecientes a cada nivel y la forma en que estos elementos están relacionados con elementos de otras capas. Estos patrones son complejos, pero permiten el desarrollo sistemático de las aplicaciones de software. Por lo tanto, los marcos de software pueden ser definidos con el fin de facilitar la aplicación de los patrones definidos en cada nivel. Por ejemplo, en la plataforma J2EE (Goncalves 2013), marcos como Java Server Faces (JSF) (Geary y Horstmann 2010), la API de Java para servicios web XML (JAX-WS) (Hansen 2007), la API de Java para servicios web RESTful (JAX- RS) (Burke 2013), y la API de persistencia de Java (JPA) (Keith y Schincariol 2013) se han definido en base a los patrones de arquitectura multicapa para el desarrollo de las capas de presentación, negocio e integración.

Sin embargo, todos estos patrones y marcos mencionados no tienen en cuenta uno de los principales problemas en la implementación de software: la seguridad. Las aplicaciones empresariales suelen almacenar miles de datos de usuario que tienen que acceder de forma segura. El problema de seguridad es tan complejo que merece sus propios catálogos de patrones. Uno de estos catálogos es Core Security Patterns (CSP) (Steel et al. 2005), que está estrechamente relacionado con el catálogo de patrones de arquitectura multicapa Core J2EE Patterns (CJP) (Alur et al. 2003). Sin embargo, aunque en teoría el catálogo CSP incluye referencias al catálogo CJP, el uso coordinado de los dos no es una tarea fácil. Por ejemplo, el catálogo CSP considera cuatro niveles para la clasificación de patrones: capa web, capa de negocio, servicios web y estrategias de diseño para identidad. Sin embargo, los patrones y estrategias de seguridad de identidad, y, en menor medida, los patrones de seguridad de servicios web no están relacionados directamente con la clasificación por capas de CJP, por lo que es difícil relacionar los dos catálogos.

Esta falta de relación directa plantea dos problemas, uno práctico y uno pedagógico. Desde un punto de vista práctico, es difícil integrar patrones de seguridad en una aplicación de arquitectura multicapa, ya que, por ejemplo, no está claro si un patrón *Audit Interceptor* perteneciente a la capa de negocio (Steel et al. 2005), es un *Application Service* (Alur et al. 2003) o debe formar parte de un *Application Service*. Si se consulta el catálogo CSP, sorprendentemente aparece en relación con un *Intercepting Filter* de CJP, que es parte de la capa de presentación en el catálogo CJP. Por lo tanto: (i) o el *Audit Interceptor* es erróneamente catalogado como un patrón en la capa de negocio y debe ser considerada como un patrón capa de presentación; (ii) o el *Audit Interceptor* no está estrechamente relacionado con el patrón de *Intercepting Filter*. Esperamos aclarar cuestiones de este tipo en nuestro trabajo.

Desde un punto de vista pedagógico surge un segundo problema: ¿cuál es el número mínimo de patrones de arquitectura multicapa que necesitan ser explicados para permitirle a alguien comprender los patrones de seguridad? El catálogo CJP incluye veintiún patrones, algunos de ellos más básicos que los demás. Por ejemplo, con el fin de diseñar una aplicación de arquitectura multicapa básica y sin ningún tipo de mecanismo de mapeo objeto-relacional (en general un mecanismo de asignación de nivel de objeto de los recursos) sólo *Front y Application*

Controller, *Application Service*, *Transfer Object* y *DAO* con algún tipo de gestión transaccional inspirado en el patrón Domain *Store* tienen que ser utilizados (Navarro et al. 2012). Sin embargo, si se proporciona un marco de mapeo objeto-relacional como JPA, entonces los patrones antes mencionados se enriquecen con un *Domain Store*, y un *Business Object* y *Composite Entity* (Navarro et al. 2012). Así, estos patrones de diseño pueden ser considerados como el núcleo de CJP. Pero, ¿es este núcleo suficiente para entender los patrones de CSP? Si los catálogos de patrones se enseñan en el contexto de los programas de grado en Ingeniería de Software, como ACM / IEEE CS Software Engineering 2004 (SE 2004) (ACM/IEEE 2004), ambos catálogos se pueden enseñar, empezando por CJP y continuando con CSP. Sin embargo, en los programas informáticos genéricos como ACM / IEEE CS Computer Science de 2008 (ACM/IEEE 2013), el número de créditos dedicados a la ingeniería de software es mucho más restringido. Por lo tanto, en estos programas de grado una relación de precedencia entre CJP y CSP puede ser muy útil para la enseñanza de los patrones fundamentales de ambos catálogos. Como consecuencia de este problema, surge una nueva pregunta: ¿existe un conjunto de patrones fundamentales en el catálogo CSP?

1.1 Objetivos

El objetivo principal de este trabajo es analizar el catálogo de patrones CSP desde la perspectiva del catálogo CJP, para explicar cada uno los patrones de CSP en términos de patrones CJP y proporcionar un uso coherente de ambos catálogos. En este análisis se deben considerar cuatro aspectos:

i. La interpretación de los patrones de CSP en términos de arquitectura multicapa (es decir, la relación de los patrones CJP con el catálogo de patrones CJP).
ii. La información adicional que necesita el catálogo CSP para facilitar su comprensión.
iii. Los requisitos previos CJP para entender los patrones del catálogo CSP.
iv. Los requisitos previos CSP para la comprensión del patrón CSP analizado.

Los objetivos secundarios de este trabajo es dar respuestas a las preguntas planteadas en la introducción:

i. ¿Cuál es el número mínimo de patrones de arquitectura multicapa que necesitan ser explicados para permitirle a alguien comprender los patrones de seguridad?
ii. ¿Existe un conjunto de patrones fundamentales en el catálogo CSP?

1.2 Motivación

En la actualidad el uso de patrones de diseño es ampliamente aceptado en la comunidad de ingeniería de software. Sin embargo, como el número de catálogos de patrones aumenta, se hace más difícil de usarlos de una manera integrada. Esto no sería un inconveniente importante a menos que los catálogos de patrones se centren en aspectos complementarios de una aplicación software, tales como la arquitectura de software y la seguridad.

Aunque hay varios catálogos patrones arquitectónicos y de patrones de seguridad, y en cierta medida los catálogos mismos definen las relaciones entre los diferentes patrones, no hemos encontrado un trabajo específico relacionado con la interpretación y relación de los patrones de seguridad con los patrones de arquitectura clásicos por capas. Esto hace que sea difícil utilizar patrones de seguridad desde el comienzo de un proyecto, y aún más difícil si se intenta incluir patrones de seguridad en una aplicación existente. Es por esto que nuestra motivación se centra en analizar y profundizar en las relaciones entre estos catálogos para permitir a profesores, estudiantes y profesionales conocer las relaciones, uso y configuración de estos patrones para que puedan utilizarlos desde el inicio del proyecto.

1.3 Esquema del trabajo

En este primer capítulo se expone los objetivos, la motivación y el porqué de este trabajo, además se introduce al lector en el contexto de los temas a desarrollar, definiciones de conceptos básicos sobre patrones y el ámbito de contribución.

El segundo capítulo es un estado del arte donde se exponen los diferentes catálogos de patrones relacionados con nuestro trabajo; es decir, los catálogos de patrones de diseño convencionales y los catálogos de patrones de seguridad.

El tercer capítulo es el núcleo del trabajo, contiene el análisis de los dos catálogos de patrones escogidos. En este capítulo se analiza el catálogo de patrones CSP desde la perspectiva del catálogo CJP, definiendo explícitamente la relación entre cada uno de los patrones y haciendo la distribución en capas.

El cuarto capítulo contiene el modelado de dos ejemplos prácticos que implementan los patrones de los dos catálogos analizados.

Finalmente el quinto capítulo describe las conclusiones y el trabajo futuro, resaltando el aporte en este trabajo y los aspectos que se pueden mejorar.

1.4 Contexto

1.4.1 Historia de los patrones

La historia de los patrones de diseño se remonta al año 1977, año en el que el arquitecto civil Christopher Alexander escribió la siguiente definición de patrón arquitectónico: “Cada patrón describe un problema que ocurre una y otra vez en nuestro medio, y luego describe el núcleo de la solución a ese problema, de tal manera que se puede utilizar esta solución un millón de veces, sin tener que hacerlo de la misma manera dos veces” (Alexander 1977).

Según Alexander el lenguaje de patrones es aplicable a cualquier tarea de ingeniería compleja, de hecho ha sido aplicada en varias de ellas. Por ejemplo ha sido bastante influyente en el campo de la ingeniería de software, por esto Alexander ha sido reconocido como el padre del movimiento "Patterns language" en ciencias de la computación.

Después, en el contexto de OOPSLA 87 (Object-Oriented Programming, Systems, Languages & Applications), reunión anual que celebran grupos de la Association for Computing Machinery (ACM), Kent Beck y Ward Cunningham empezaron a aplicar el concepto de patrones en la informática. Se preocuparon por capturar las buenas ideas para luego traspasarlas a los

nuevos programadores, fue entonces cuando decidieron publicar un artículo titulado "Using Pattern Languages for OO Programs" (Beck et al. 1987).

En 1991 Gang of Four (GoF) o la banda de los cuatro, como se les conoce, compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides trabajaron en su primer catálogo de patrones. Este catálogo se popularizó en el año 1994, tras la publicación del libro "Design Patterns" (Gamma et al. 1994), en el que se recogían veintitrés patrones de diseño comunes.

En 2003 Martin Fowler, David Rice, Mateo Foemmel, Edward Hieatt, Robert Mee, y Randy Stafford publicaron el libro "Patterns of Enterprise Application Architecture" (Fowler et al. 2003). Este libro divide la aplicación empresarial en diferentes capas y enfatiza en los patrones relacionados con mapeo entre objetos y bases de datos relacionales. El libro recoge una serie de buenas prácticas utilizadas por Martin Fowler y otros arquitectos de software para el desarrollo de diversos sistemas.

En 2003 Deepak Alur, John Crupi y Dan Malks publicaron la segunda edición de su libro, publicado originalmente en 2001, enfocado a las buenas prácticas y estrategias de diseño para aplicaciones empresariales, "Core J2EE Patterns" (Alur et al. 2003). En este trabajo se estudian veintiún patrones de arquitectura multicapa con ejemplos de código y puntos clave para la refactorización de este tipo de aplicaciones utilizando patrones de diseño. El libro recoge una serie de buenas prácticas utilizadas en los desarrollos llevados a cabo por Sun Microsystems.

También en 2003, William Crawford y Jonathan Kaplan hicieron público su trabajo "J2EE Design Patterns" (Crawford 2003). Este libro está enfocado en el estudio de patrones de diseño para aplicaciones empresariales y abarca temas críticos como extensibilidad y mantenimiento, escalabilidad, modelado de datos, transacciones e interoperabilidad.

1.4.2 ¿Qué es un patrón de software?

Hay muchas definiciones de patrones, por ejemplo, en la página web de la comunidad The Hillside Group[1] (Hillside Group Web 2014), que es un gran repositorio de información en relación a patrones del software y lenguajes de patrones, dos miembros del grupo definen los patrones así:

James Coplien escribe:

> "Los patrones son una disciplina de resolución de problemas de ingeniería de software que reciente surge de la comunidad orientada a objetos. Los patrones tienen sus raíces en muchas disciplinas, incluyendo la programación literaria, y sobre todo en la obra de Alexander en la planificación urbana y la arquitectura del edificio (Alexander 1977). Pero los patrones han sido utilizados para dominios tan diversos como la organización de desarrollo y el proceso, la exposición y la enseñanza, y la arquitectura de software" (James Copien y Richard Gabriel, 2014).

Dick Gabriel hace un acertado comentario sobre lo que pudo haber escrito el padre del concepto de patrón:

> "Alexander pudo haber escrito una definición, de una frase de qué es un patrón, o un ensayo, pero en lugar de eso, escribió un libro de 550 páginas para hacerlo, ya que el concepto de patrón es complejo".

Pero también nos proporciona su propia definición de lo que para él es un patrón:

> "Cada patrón es una regla de tres partes que expresa una relación entre un cierto contexto, un cierto sistema de fuerzas que se produce repetidamente en ese contexto, y una cierta configuración de software que permite a estas fuerzas resolverse a sí mismas. (James Copien y Richard Gabriel, 2014).

[1] The Hillside Group con página web www.hillside.net, es una comunidad que promueve, registra y analiza el uso de patrones y lenguajes de patones para mejorar el software y su desarrollo.

James Coplien, sostiene que un buen patrón tiene que cumplir las siguientes características (James Copien y Richard Gabriel, 2014):

- *Soluciona un problema*. Un patrón captura soluciones, no sólo los principios abstractos o estrategias.
- *Se trata de un concepto probado*. Los patrones son soluciones probadas por la industria, no teorías o especulaciones.
- *La solución no es obvia*. Muchas de las técnicas de resolución de problemas tratan de obtener soluciones a partir de primeros principios. Los mejores patrones generan una solución a un problema indirectamente, un enfoque necesario para los problemas más difíciles de diseño.
- *En él se describe una relación*. Los patrones no sólo describen los módulos, describen también las estructuras y mecanismos del sistema más profundas.
- *El patrón tiene un componente humano significativo*. Todo software sirve a la comodidad o la calidad de la vida humana. Los mejores patrones facilitan el uso y mantenimiento del software a las personas.

1.4.3 Tipos de patrones

En la bibliografía de patrones de software podemos encontrar los patrones clasificados en diversas categorías según el rango de abstracción. En particular los autores de la serie de libros "Pattern-Oriented Software Architecture" (Buschmann et al. 1996; Schmidt et al. 2002; Kircher y Jain 2004; Buschmann et al. 2007; Buschmann et al. 2007) definen tres tipos de patrones:

- *Patrones arquitectónicos*: Un patrón arquitectónico expresa una organización estructural fundamental o un esquema para sistemas de software. Proporciona un conjunto de subsistemas predefinidos, especifica sus responsabilidades, e incluye reglas y directrices para la organización de las relaciones entre ellos.

- *Patrones de diseño*: Son patrones de escala media que proporcionan un esquema para refinar los subsistemas o componentes de un sistema de software, o las relaciones

entre ellos. En él se describe la estructura comúnmente recurrente de comunicar componentes que resuelve un problema de diseño general en un contexto particular.

Tipos de patrones de diseño:

- *Patrones creacionales*: Tratan sobre cómo crear instancias de objetos y sobre cómo hacer los programas más flexibles y generales abstrayendo el proceso de creación de instancias.
- *Patrones estructurales*: Describen cómo las clases y los objetos pueden ser combinados para formar estructuras mayores.
- *Patrones de comportamiento*: Son patrones que tratan de forma más específica con aspectos relacionados con la comunicación entre objetos.

- *Idiomas*: Son patrones de bajo nivel que ayudan a implementar problemas específicos para un determinado lenguaje de programación.

Además, también es importante reseñar el concepto de "anti patrón de diseño", que con forma semejante a la de un patrón, intenta prevenir contra errores comunes de diseño en el software.

1.4.4 Catálogos y lenguajes de patrones

Un catálogo de patrones es una colección de patrones relacionados (tal vez sólo vagamente o informalmente relacionados). Típicamente subdivide los patrones en al menos un pequeño número de categorías amplias y puede incluir una cierta cantidad de referencias cruzadas entre los patrones.

Un lenguaje de patrones es un conjunto coherente de patrones relacionados que trabajan juntos para apoyar la construcción y evolución de las arquitecturas integrales. No sólo está organizado en grupos y subgrupos relacionados en varios niveles de granularidad, sino que además describe las numerosas interrelaciones entre los patrones y sus agrupaciones y cómo pueden ser combinados y compuestos para resolver problemas más complejos. Los patrones en un lenguaje de patrones deben ser descritos en un estilo coherente y uniforme, y tienen que

cubrir una parte suficientemente amplia de los problemas y las soluciones para que una parte significativa de las arquitecturas completas sean construidas.

Un catálogo, añade un poco de estructura y organización a una colección de patrones, pero no suele ir mucho más allá de mostrar sólo la estructura y las relaciones más exteriormente visible (si de hecho se nota cualquiera de ellos). Un lenguaje de patrones agrega estructura profunda, interacciones entre patrones, y la uniformidad de un catálogo de patrones. Ambos, sistemas de patrones y lenguajes de patrones forman conjuntos coherentes de patrones fuertemente entretejidos para describir y resolver problemas en un dominio particular. Sin embargo, un lenguaje de patrones añade robustez, amplitud y plenitud a un sistema de patrones. La principal diferencia es que, idealmente, los lenguajes de patrones son computacionalmente completos, muestran todas las posibles combinaciones de los patrones y sus variaciones para producir arquitecturas completas.

1.4.5 Historia de los patrones de seguridad

Un patrón de seguridad describe una solución al problema de controlar (detener o mitigar) un conjunto de amenazas específicas a través de algún mecanismo de seguridad, que se define en un contexto dado (Schumacher et al. 2006b).

Un patrón de seguridad no está directamente relacionado con una vulnerabilidad, pero está directamente relacionado con una amenaza. La amenaza específica puede ser el resultado de uno o más vulnerabilidades, pero el patrón no está destinado a reparar la vulnerabilidad, sino a detener o mitigar la amenaza.

En el libro de Eduardo Fernandez, "Security Patterns in Practice: Designing Secure Architectures Using Software Patterns" encontramos el siguiente resumen sobre la historia de los patrones de seguridad (Fernandez 2013):

"Yoder y Barcalow escribieron el primer artículo sobre patrones de seguridad que hemos encontrado en la literatura (Yoder y Barcalow 1997). Incluyen una variedad de patrones útiles en diferentes aspectos de la seguridad. Antes de ellos, al menos tres documentos (Fernandez 1993a; Fernandez 1994a; Essmayr et al. 1997) habían mostrado los patrones orientados a objetos de sistemas seguros sin llamarlos "patrones", o el uso de una de las plantillas de patrón estándar. En

1998, dos patrones más aparecieron, un patrón para la criptografía (Braga et al. 98) y un patrón para el control de acceso (Das y Garrido 1998). Después de eso, han aparecido diversos catálogos, y en la actualidad hay hasta tres libros sobre el tema (Blakley et al. 2004; Schumacher et al. 2006b; Steel et al. 2005), uno de los cuales (Schumacher et al. 2006b) fue el primero en tratar de clasificar y unificar una variedad de patrones de seguridad".

Los catálogos de patrones de seguridad más conocidos son:

- *Core Security Patterns* (Steel et al. 2005). Incluye veintitrés patrones de seguridad. Fue publicado en octubre de 2005.
- *Enterprise Security and Risk Management Patterns* (Schumacher et al. 2006b). Incluye cuarenta y seis patrones de seguridad de los dominios de seguridad de la empresa y gestión de riesgos, identificación y autenticación, control de acceso, la contabilidad, la arquitectura de firewall, y aplicaciones de internet seguras.
- *Core Web Service Security Patterns* (Hogg et al. 2006). Incluye dieciocho patrones que abordan la autenticación, la seguridad de mensaje, capa de transporte y el mensaje, acceso a los recursos, la protección del límite de servicio y la implementación del servicio. Todos estos patrones se describen desde la perspectiva de la tecnología Microsoft.

También existen otras agrupaciones de patrones como guías y/o repositorios, tales como *Security design patterns technical guide* (Blakley et al. 2004), *Secure Design Patterns* (Dougherty et al. 2009) y (Hafiz et al. 2012; Kienzle et al. 2002; Yskout et al. 2006). No son estrictamente catálogos de patrones, son repositorios con el inventario de los patrones donde se organizan de acuerdo a alguna clasificación, pero no proporcionan descripción de patrón.

2 Trabajo relacionado

En este capítulo se analizan algunos de los catálogos de patrones que existen y se explica la razón de haber elegido dos de ellos para el análisis de este trabajo.

"Design Patterns" (Gamma et al. 1994), es uno de los catálogos de patrones de diseño más exitosos. Este catálogo analiza algunas de las APIs más prominentes de su tiempo y abstrae soluciones recurrentes en patrones de diseño. Aunque este catálogo es muy útil y sus patrones se utilizan en otros catálogos de patrones, no define una forma de organizar la arquitectura de la aplicación desde la interfaz de usuario hasta la capa de recursos. Por lo tanto, en nuestra opinión, no puede ser considerado como un catálogo de patrones de arquitectura completo.

"Core J2EE Design Patterns", CJP, (Alur et al. 2003) fue uno de los primeros catálogos de patrones de diseño, publicado originalmente en 2001 (Alur et al. 2001). Estos patrones reúnen el conocimiento y experiencia de los ingenieros de Sun Microsystems / Oracle Corporación en materia de desarrollo de aplicaciones web. Aunque parece que se refiere específicamente a la plataforma J2EE, sus patrones se pueden utilizar en otras plataformas.

Los patrones del libro de Fowler "Patterns of Enterprise Application Architecture" (Fowler et al. 2003) es otro interesante catálogo de aplicaciones de arquitectura multicapa. La mayor parte de sus patrones están directamente relacionados con CJP. Sin embargo, el catálogo de Fowler proporciona un tratamiento en profundidad de los problemas de concurrencia y persistencia. De hecho, el patrón Domain Store de CJP es una mezcla de nueve de los patrones de Fowler.

"J2EE Design Patterns" (Crawford 2003) es un catálogo muy similar a CJP. De hecho la mayoría de los patrones incluidos en este catálogo se pueden asignar directamente a los patrones descritos en CJP. Sin embargo, este catálogo ofrece una visión más detallada de los problemas de concurrencia y persistencia (como el catálogo de Fowler), y también incluye pautas adicionales para la mensajería.

La Serie libros de la editorial Wiley, “Pattern-Oriented Software Architecture” (Buschmann et al. 1996; Schmidt et al. 2002; Kircher y Jain 2004; Buschmann et al. 2007; Buschmann et al. 2007) incluye cinco volúmenes que consideran patrones similares a los definidos en los catálogos de patrones antes mencionados y dos volúmenes centrados específicamente en los lenguajes de patrones.

“Core Security Patterns” (Steel et al. 2005) es uno de los primeros catálogos con un enfoque centrado en los patrones de seguridad. Escrito por ingenieros de Sun Microsystems / Oracle Corporation, se ocupa de cuestiones interesantes respecto a la seguridad del software y define la forma de aplicar estas cuestiones en términos de patrones de seguridad. Aunque este catálogo está estrechamente relacionado con el catálogo CJP, en la práctica esta relación es insuficiente o en algunos casos incluso erróneos si se desea integrar los patrones de CSP con una arquitectura multicapa que se define en términos de patrones CJP.

Por último, “Security Patterns in Practice” (Fernandez 2013) es un catálogo completo de los patrones de seguridad. Este catálogo no pone la misma atención a las cuestiones de seguridad básicas que el catálogo CSP y define directamente el catálogo de patrones. Además, este catálogo considera algunos tipos de patrón no considerados en CSP, como los dedicados a asegurar la gestión de procesos, gestión de ejecución y archivos de forma segura, arquitectura segura del sistema operativo y la administración, o la computación en la nube.

Por supuesto, hay muchos otros catálogos de patrones de software interesantes, como los dedicados a los patrones SOA, como por ejemplo (Erl 2009). Sin embargo, estos catálogos no se analizan porque son catálogos que no son específicamente arquitectónicos ni de seguridad.

En cuanto a esta serie de catálogos de patrones, no es una tarea fácil seleccionar un catálogo de arquitectura multicapa y otro de patrones de seguridad con el fin de relacionarlos. Hemos elegido los catálogos producidos por los ingenieros de Sun Microsystems / Oracle Corporación porque ambos catálogos son escritos por ingenieros pertenecientes a la misma empresa y para la misma plataforma (aunque son generalizables a diferentes plataformas) y esto proporciona un punto de partida común para los dos.

2.1 Catálogo de patrones

2.1.1 Core J2EE Patterns

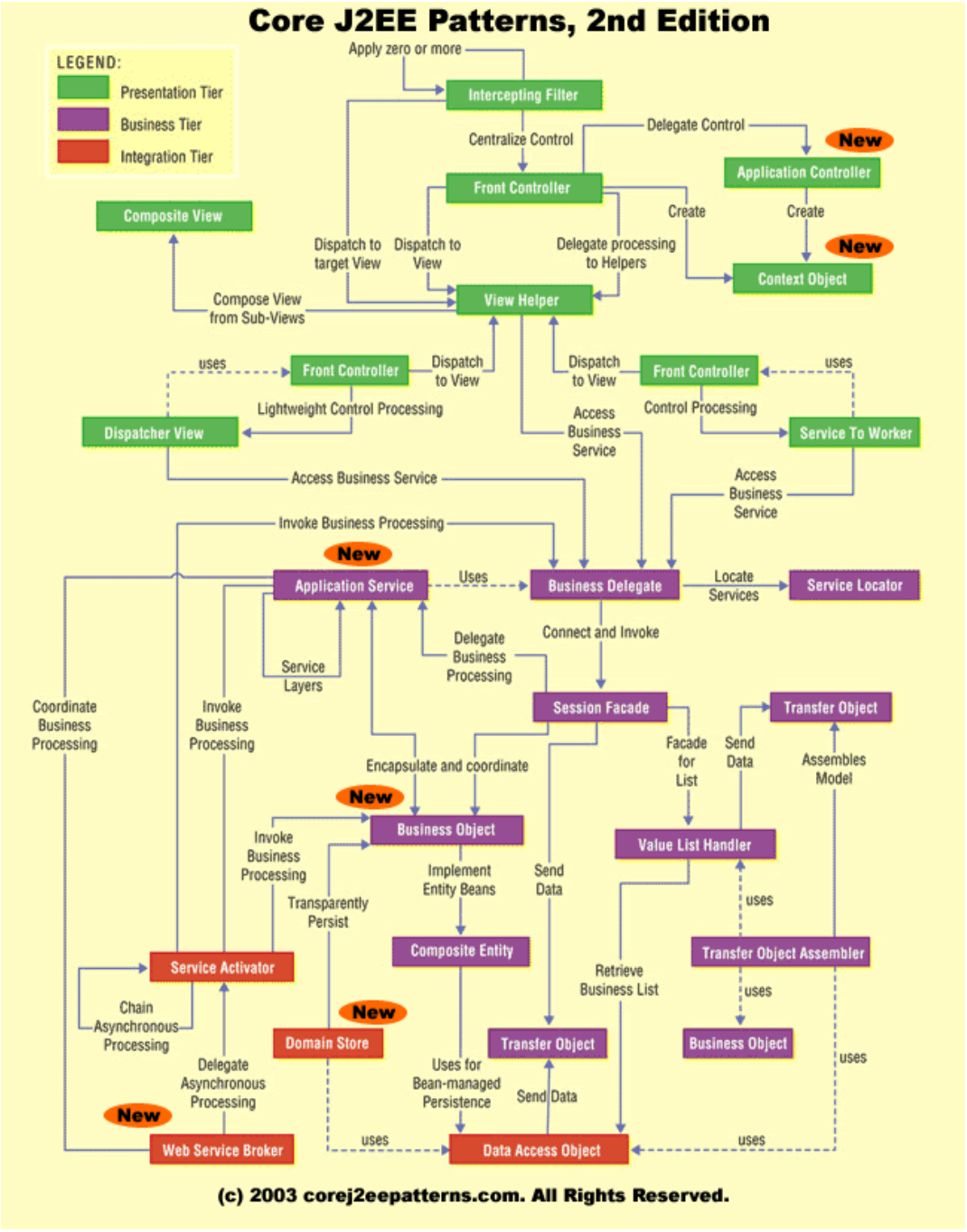


Figura 2.1 Catálogo de Patrones Core J2EE

El libro Core J2EE Patterns (CJP) (Alur et al. 2003), presenta una colección de patrones fruto de la experiencia de los arquitectos de Java Sun Center. Trata sobre los patrones de la plataforma Java 2, Enterprise Edition (J2EE). El libro se centra en las siguientes cuatro tecnologías J2EE: Servlets, JSP, los componentes EJB y JMS. Los patrones están clasificados en tres capas: presentación, negocio e integración, como se observa en la Figura 2.1

2.1.1.1 Capa de presentación

Esta capa tiene ocho patrones que se refieren al uso de Servlets, JSP, JavaBeans y etiquetas personalizadas para diseñar aplicaciones Web. Los patrones describen numerosas estrategias de implementación y hacen frente a los problemas comunes, tales como el control de solicitudes, la partición de la aplicación, y la generación de pantallas compuestas. Esos patrones son:

- *Intercepting Filter*. Intercepta las peticiones entrantes y las respuestas salientes y aplica un filtro. Estos filtros se pueden añadir y eliminar de una manera declarativa, permitiendo aplicarlos discretamente en una variedad de combinaciones. Después de este procesamiento previo y/o post-procesamiento, el filtro final dirige la petición al recurso destino para una solicitud entrante. Dicho recurso es a menudo un *Front Controller*, pero puede ser una vista si es saliente.

- *Front Controller*. Patrón que trabaja como un contenedor para almacenar la lógica de procesamiento común que se produce en el nivel de presentación y que de otra manera podría ser colocado por error en una vista. Un controlador gestiona y centraliza las solicitudes. Además gestiona la recuperación de contenidos, seguridad, administración de la vista, y la navegación, delegando en un componente *Dispatcher* para enviar la solicitud a una vista.

- *Application Controller*. Este patrón centraliza el control, la recuperación, y la invocación de la vista y el procesamiento de comandos. Mientras que un *Front Controller* actúa como un punto de acceso centralizado y control para las solicitudes de entrada,

el *Application Controller* se encarga de identificar e invocar comandos, y de la redirección a las vistas.

- *Context Object*. Patrón que encapsula el estado de forma independiente del protocolo para ser compartido a través de su aplicación. El uso del contexto del objeto hace que las pruebas sean más fáciles, lo que facilita un entorno de prueba más genérico con una menor dependencia en un contenedor específico.

- *View Helper*. Patrón que fomenta la separación de código relacionado con el formato desde otra lógica de negocio. Se sugiere el uso de componentes auxiliares para encapsular la lógica en relación con el inicio de recuperación de contenido, validación y adaptación y el formateo del modelo. El componente *View Helper* se deja entonces para encapsular el formato de presentación. Componentes auxiliares normalmente delegan en los servicios de negocio a través de un *Business Delegate* o un *Application Service*, mientras que un *View Helper* puede estar compuesto de varias sub-componentes para crear su plantilla.

- *Composite View*. Patrón que sugiere componer una vista desde numerosas piezas atómicas. Múltiples *View Helper* más pequeñas, tanto estáticos como dinámicos, se unen para crear una única plantilla.

- *Service to Worker*. Es la combinación de un *Front Controller, Application Controller* y *View Helper*. Utilizado principalmente cuando se desea centralizar el control de solicitudes e invocar la lógica de negocio antes de pasar el control a una vista que además presenta un comportamiento dinámico dependiendo de la invocación.

- *Dispatcher View*. En este patrón no se utilizan controladores y las propias vistas invocan al negocio y redirigen a las vistas. Más utilizado para vistas estáticas que para vistas generadas a partir de un modelo de presentación existente. En general, se recomienda usar cuando las vistas son independientes de cualquier respuesta del negocio.

2.1.1.2 Capa de negocio

Esta capa presenta nueve patrones que se refieren a la utilización especialmente de la tecnología EJB para diseñar componentes de negocio. Los patrones en este capítulo proporcionan mejores prácticas para el uso de la tecnología EJB y JMS. Estos patrones incluyen alguna discusión sobre otras tecnologías, tales como JNDI y JDBC:

- *Business Delegate*. Este patrón reduce el acoplamiento entre las capas remotas y proporciona un punto de entrada para el acceso a los servicios a distancia en el nivel de negocio. Un *Business Delegate* podría también cachear datos cuando sea necesario mejorar el rendimiento. Un *Business Delegate* encapsula una *Session Façade* y mantiene una relación de uno a uno con una *Session Façade*. Un patrón *Application Service* utiliza un *Business Delegate* para invocar una *Session Façade*.

- *Service Locator*. Es un patrón que encapsula los mecanismos de búsqueda de componentes de servicios de negocios. Un *Business Delegate* utiliza un localizador de servicios para conectarse a una *Session Façade*. Otros clientes que necesitan localizar y conectarse a la *Session Façade*, otros servicios de la capa de negocio, y los servicios Web pueden utilizar un *Service Locator*.

- *Session Façade*. Es un patrón que proporciona servicios de grano grueso a los clientes, al ocultar la complejidad de las interacciones del servicio de negocio. Una *Session Façade* podría invocar varias implementaciones de *Application Service* o *Business Objects*. Una *Session Façade* también puede encapsular un *Value List Handler*.

- *Application Service*. Centraliza y agrega comportamiento para proporcionar una capa uniforme de servicio a los servicios de la capa de negocio. Un *Application Service* podría interactuar con otros servicios o *Business Objects*. Un *Application Service* puede invocar otros *Application Services* y así crear una capa de servicios en su aplicación.

- *Business Object*. Implementa el modelo de dominio conceptual utilizando un modelo de objetos. *Business Object* separa datos de negocio y lógica en una capa independiente de

la aplicación. Los *Business Objects* suelen representar objetos persistentes y su persistencia puede ser transparente usando un patrón *Domain Store*.

- *Composite Entity*. Este patrón es útil cuando se usan beans de entidad para implementar un modelo de dominio conceptual y para evitar la sobrecarga en la red en las relaciones entre beans remotos. Un *Composite Entity* agrega un conjunto de *Business Objects* relacionados dentro de beans de entidad de grano grueso.

- *Transfer Object*. Proporciona las mejores técnicas y estrategias de intercambio de datos entre las capas para reducir la sobrecarga de la red, reduciendo al mínimo el número de llamadas para obtener los datos de otras capas.

- *Transfer Object Assembler*. Construye un compuesto de *Transfer Objects* a partir de diversas fuentes de datos. Estas fuentes podrían ser componentes EJB, *Data Access Objects*, u otros objetos de Java arbitrarios. Este patrón es más útil cuando el cliente necesita obtener datos para el modelo de aplicación o parte del modelo.

- *Value List Handler*. Utiliza el patrón *Iterador* de GoF para proporcionar ejecución de consultas y procesar servicios. El *Value List Handler* almacena en caché los resultados de la ejecución de la consulta y devuelve subconjuntos del resultado a los clientes según lo solicitado. Mediante el uso de este patrón, es posible evitar los gastos generales asociados con la búsqueda de un gran número de beans de entidad. El *Value List Handler* utiliza un *Data Access Object* para ejecutar una consulta y recuperar los resultados de un almacén persistente.

2.1.1.3 Capa de integración

Esta capa presenta cuatro patrones que se refieren a la integración de aplicaciones J2EE con el nivel de recursos y sistemas externos. Los patrones se relacionan con el uso de JDBC y JMS para permitir la integración entre los componentes del nivel de la capa de negocio y de recursos.

- *Data Access Object*. Este patrón permite el acoplamiento débil entre la capa de negocio y la capa de recursos. *Data Access Object* encapsula toda la lógica de acceso a datos para crear, recuperar, eliminar y actualizar datos de un almacén persistente. *Data Access Object* utiliza un *Transfer Object* para enviar y recibir datos.

- *Service Activator*. Permite el procesamiento asíncrono en las aplicaciones empresariales utilizando JMS. Un *Service Activator* puede invocar un *Application Service*, *Session Facade* o *Business Objects*. También se pueden utilizar varios *Service Activator* para proporcionar el procesamiento asíncrono paralelo para tareas complejas.

- *Domain Store*. Proporciona un poderoso mecanismo para implementar la persistencia transparente para su modelo de objetos. Combina y conecta varios otros patrones incluyendo *Data Access Object*. Es la base de marcos de persistencia como JPA.

- *Web Service Broker*. Expone y trabaja como bróker en uno o más servicios de su aplicación para clientes externos como un servicio web utilizando XML y protocolos web estándar. Un *Web Service Broker* puede interactuar con *Application Service* y *Session Facade*. Un *Web Service Broker* utiliza uno o varios *Service Activator* para llevar a cabo el procesamiento asincrónico de una solicitud.

2.1.2 Patterns of Enterprise Application Architecture - Martín Fowler

Este catálogo de patrones (Fowler et al. 2003) centra su discusión en torno a una arquitectura de tres capas principales:

- Presentación: Prestación de servicios, la visualización de información, las solicitudes HTTP, las invocaciones de línea de comandos.
- Dominio o lógica de negocio: Lógica que implementa los procesos de negocio en el sistema.
- Fuente de datos: Comunicación con las bases de datos, sistemas de mensajería, gestores de transacciones, etc.

Sin embargo, como el mismo autor cita en el libro, algunos de sus patones pueden ser llamados arquitectónicos, pero otros son más patrones de diseño ya que ayudan más a realizar una arquitectura.

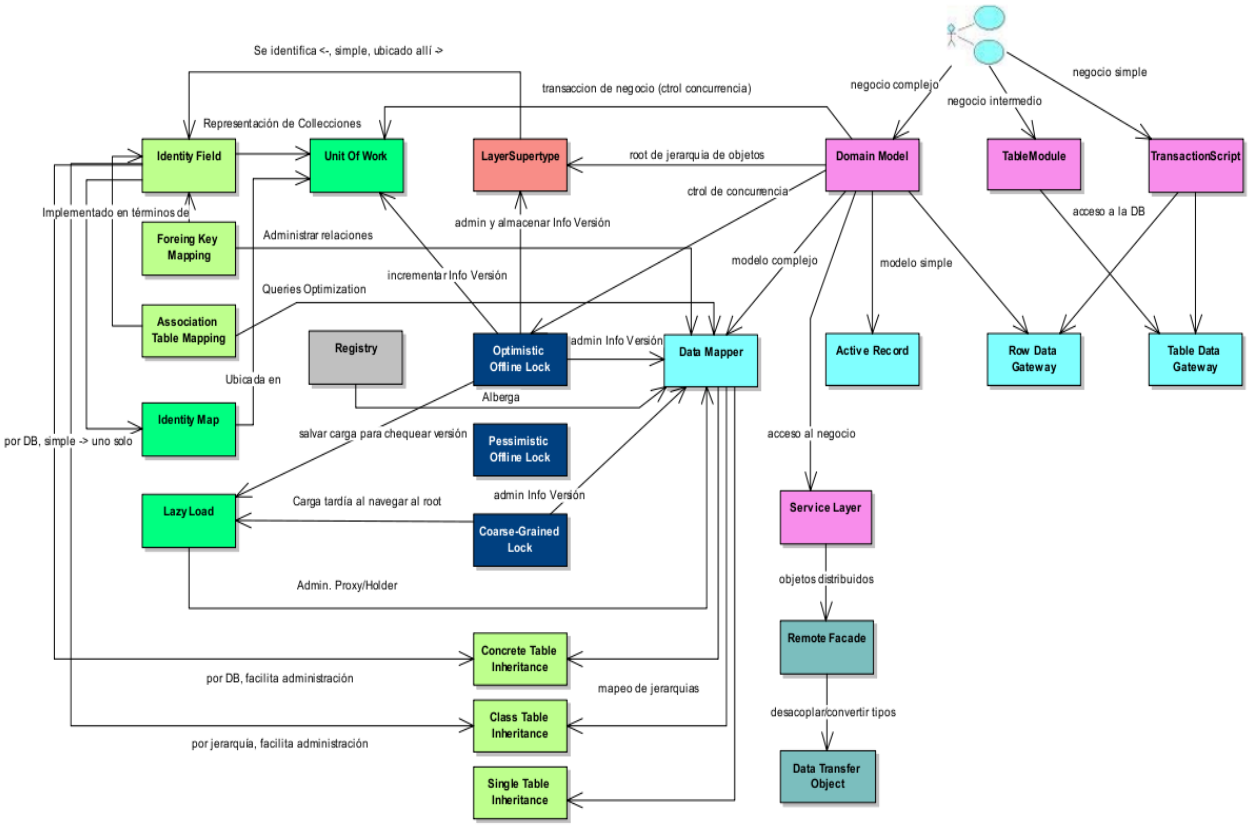


Figura 2.2 Relación entre patrones del catálogo de Martin Fowler[2]

2.1.2.1 Patrones de lógica de dominio

- *Transaction Script*. Es esencialmente un procedimiento que se lleva a la entrada de la presentación, la procesa con validaciones y cálculos o invoca las operaciones de otros sistemas y almacena los datos en la base de datos. Si se sabe que la aplicación será simple y que no evolucionará con el tiempo, no tiene sentido invertir en la construcción de un modelo de dominio. Para eso existe el patrón *Transaction Script*.

[2] Figura tomada de la página web: http://elblogdelfrasco.blogspot.com.es

- *Table Module*. Si el dominio es un poco más complejo, es mejor usar un patrón *Table Module*. Éste organiza la lógica de dominio con una clase por tabla en la base de datos, y una sola instancia de una clase contiene los diversos procedimientos que actuará sobre los datos. Para aplicaciones que ameritan tener capa de negocio y persistencia, aunque no tengan un modelo del dominio puro. *Table Module* está diseñado para trabajar con *Record Set*. La capa de persistencia podría ser un conjunto de *Gateways* que ejecutan consultas a la base de datos y devuelvan los resultados en Record Sets.

- *Domain Model*. Es un patrón que representa objetos del modelo de dominio y que incorpora tanto datos como comportamiento. Este ya es un patrón con los beneficios de la orientación a objetos. Un modelo de dominio crea una red de objetos interconectados, donde cada objeto representa un individuo significativo, ya sea tan grande como una corporación o tan pequeño como una sola línea en un formulario de pedido. Habrá una representación en memoria para cada instancia de cada objeto.
  Implementar un Domain Model implica un costo altísimo de diseño y construcción de código de infraestructura que sólo es rentable si la aplicación es lo suficientemente grande y compleja, y/o se prevé que deberá escalarse en el futuro.

- *Service Layer*. Las aplicaciones empresariales suelen requerir diferentes tipos de interfaces a los datos que almacenan y la lógica que aplican: cargadores de datos, interfaces de usuario, y pasarelas de integración, entre otros. Un patrón *Service Layer* define límites de una solicitud y su conjunto de operaciones disponibles desde la perspectiva de una capa de cliente interactiva. Encapsula la lógica de negocio de la aplicación, el control de las transacciones y las respuestas de coordinación en la ejecución de sus operaciones.

2.1.2.2 Patrones de arquitectura de fuente de datos

Es conveniente separar el acceso SQL a partir de la lógica de negocio y colocarlo en clases separadas. Una buena manera de organizar estas clases es basarlas en la estructura de la tabla de la base de datos para tener una clase por tabla de base de datos. *Row Data Gateway* y *Table Data Gateway* se basan ambos en el patrón básico *Gateway*, una clase en memoria que mapea una tabla de la base de datos y tiene un campo por columna de la base de datos.

- *Table Data Gateway*. Es un patrón que centraliza el uso de SQL, actuando como un Gateway a una tabla de base de datos. Contiene toda la sintaxis SQL para acceder a una sola tabla o vista: seleccionar, insertar, actualizar y eliminar. Otro código llama a sus métodos de toda la interacción con la base de datos. Es un ejemplo que funciona a la perfección con el patrón de dominio *Table Module*.

- *Row Data Gateway*. Un patrón que implementa un objeto que actúa como una puerta de enlace a un único registro de un origen de datos. Hay una instancia por cada fila.

- *Active Record*. Un objeto que contiene una fila de una tabla de base de datos o vista. Muchos de los datos son persistentes y tienen que ser almacenados en una base de datos. Si la aplicación tiene un modelo de dominio que se parece mucho al modelo de la base de datos, entones es mejor considerar *Active Record*, que combina la pasarela y el objeto del dominio en una sola clase que incluye tanto código de acceso a la base de datos. El acoplamiento con la base de datos es muy grande, ya que no hay capa de persistencia y cada objeto de negocio sabe cómo persistirse.

- *Data Mapper*. Es una capa de software que separa los objetos en memoria de la base de datos. Su función es transferir los datos entre los dos y también aislarlos unos de otros. Con *Data Mapper* los objetos en memoria no tienen por qué saber siquiera que hay una base de datos actual, no necesitan ningún código de interfaz SQL, y ciertamente ningún conocimiento del esquema de base de datos. Usado en frameworks, como Hibernate o Top Link, que transforman de forma transparente los objetos del dominio en tablas y entidades del mundo relacional de las bases de datos.

2.1.2.3 Patrones de comportamiento objeto relacional

Los patrones de comportamiento resuelven problemas de infraestructura relacionados directamente con funcionalidades de soporte para el dominio:

- *Unit of Work*. Este patrón sirve para la gestión de concurrencia. Mantiene un registro de todos los datos accedidos durante una transacción comercial. Una vez terminada la transacción, actúa de manera conveniente sobre la base de datos.

- *Identity Map*. Este patrón sirve para evitar tener en memoria dos representaciones distintas del mismo objeto en una transacción de negocio. Funciona como una caché de objetos de negocio y minimiza las lecturas a la base de datos. El ciclo de vida de un *Identity Map* es una transacción de negocio.

- *Lazy Load*. Un objeto que no contiene todos los datos que necesita, pero sabe cómo conseguirlo. Este patrón resuelve problemas de carga desmesurada y dependencias circulares. El ejemplo clásico es una relación uno a muchos, donde un objeto de negocio posee una colección de otros objetos. Cuando el objeto A es solicitado, se trae a memoria el objeto A y sólo el objeto A, con la colección de objetos B vacía. Los objetos B serán leídos cuando realmente se necesiten.

2.1.2.4 Patrones estructurales objeto relacional

Fowler define tres tipos de patrones de Persistencia:

- *Patrones de Mapeo (Mapeo de Relaciones y Mapeo de Jerarquías)*
- *Patrones de Comportamiento*
- *Patrones de Arquitectura*

Los patrones de Mapeo de Relaciones centran su atención en la forma en que los objetos se relacionan en memoria y la forma en que las tablas se relacionan en la base de datos. Estos patrones son:

- *Identity Field*. Este patrón se utiliza para las referencias entre los objetos que se convierten en las claves externas. *Identity Field* guarda un campo ID de base de datos en un objeto para mantener la identidad entre un objeto en memoria y una fila de base de datos. Todo lo que hace es almacenar la clave principal de la tabla de base de datos relacional en los campos del objeto.

- *Foreign Key Mapping*. Este patrón asigna una asociación entre los objetos a una referencia de clave externa entre las tablas. Sirve para mapear relaciones de uno a muchos.

- *Association Table Mapping*. Guarda una asociación como una tabla con claves externas a las tablas que están vinculadas por la asociación. Sirve para mapear relaciones de muchos a muchos. En una base de datos esta relación se resuelve introduciendo una tercera tabla que contenga las claves primarias de ambas entidades involucradas en la relación.

- *Dependent Mapping.* Este patrón realiza en una clase el mapeo de base de datos de clases hijas. La idea básica detrás de *Dependent Mapping* es que una clase (la dependiente) se basa en alguna otra clase (el propietario) para su persistencia en la de base de datos. Cada dependiente puede tener un solo propietario y debe tener un dueño.

- *Embedded Value*. Asigna los valores de un objeto en campos en el registro del propietario del objeto. *Embedded Value* sólo se puede utilizar para dependientes bastante simples. Un solo dependiente, o unos pocos dependientes separados, funcionan bien. *Serialized LOB* trabaja con estructuras más complejas.

- *Serialized LOB*. LOB significa "objeto grande," que pueden ser binarios (BLOB) o textual (CLOB-Character Large Object). Estructuras jerárquicas, tales como organigramas y listas de materiales son donde un LOB serializado puede ahorrar un montón de idas y vueltas de bases de datos. Este patrón guarda un grafo de objetos

serializados en un único objeto grande (LOB), que almacena en un campo de base de datos.

Los patrones de Mapeo de Jerarquías, como su nombre lo indica, se dedican a resolver la representación de la herencia de clases/objetos en disco. Los cuatro patrones sugieren distintas formas de llevar a cabo esta proyección del dominio en la base de datos:

- *Single Table Inheritance*. Representa una jerarquía de herencia de clases como una sola tabla que tiene columnas para todos los campos de las distintas clases. Este patrón mapea toda la jerarquía en una única tabla, duplicando muchos datos, pero facilitando las búsquedas.

- *Class Table Inheritance*. Representa una jerarquía de herencia de clases con una tabla para cada clase. Mapea cada clase de la jerarquía en una tabla individual, sin ninguna duplicación de campos.

- *Concrete Table Inheritance*. Representa una jerarquía de herencia de clases con una tabla por clase concreta de la jerarquía. Es una solución intermedia, en la que sólo se mapean las clases concretas y las abstractas simplemente duplican campos en sus clases hijas.

- *Inheritance Mappers*. Una estructura para organizar la base de datos que manejan mapeo de jerarquías de herencia. Al asignar de una jerarquía de herencia orientada a objetos en memoria a una base de datos relacional que tiene que minimizar la cantidad de código necesario para guardar y cargar los datos a la base de datos. También quiere proporcionar tanto el comportamiento de asignación abstracta y concreta que le permite guardar o cargar una superclase o subclase.

2.1.2.5 Patrones de asignación de metadatos a objetos relacionales

- *Metadata Mapping*. Contiene detalles de mapeo objeto-relacional en los metadatos. Un mapeo de metadatos permite a los desarrolladores definir las asignaciones en

forma de tabla simple, que luego pueden ser procesados por código genérico para llevar a cabo los detalles de la lectura, inserción y actualización de los datos.

- *Query Object*. Un objeto que representa una consulta de base de datos. Un Query Object es un intérprete (GoF), es decir, una estructura de objetos que se pueden formar en sí en una consulta SQL. Puede crear esta consulta para las clases y los campos en lugar de tablas y columnas. De esta manera, los que escriben las consultas pueden hacerlo independientemente del esquema de base de datos y cambios en el esquema puede ser localizada en un solo lugar.

- *Repository*. Conceptualmente un repositorio encapsula el conjunto de objetos persistentes en un almacén de datos y las operaciones que se realizan sobre ellos, proporcionando una visión más orientada a objetos de la capa de persistencia. Repositorio también es compatible con el objetivo de lograr una separación limpia y de un solo sentido de dependencia entre las capas de dominio y asignación de datos.

2.1.2.6 Patrones presentación web

- *Model View Controller*. Divide la interacción de la interfaz de usuario en tres roles distintos. El modelo es un objeto que representa una cierta información sobre el dominio. La vista representa la visualización del modelo en la interfaz de usuario. El controlador toma la entrada del usuario, manipula el modelo, y hace que el fin de actualizar apropiadamente. De esta manera la interfaz de usuario es una combinación de la vista y el controlador. La primera, y más importante, razón de aplicar *Model View Controller* es asegurar que los modelos están completamente separados de la presentación.

- *Page Controller*. Un objeto que controla una solicitud para una página específica o acción en un sitio Web. El controlador de la página tiene un controlador de entrada para cada página lógica del sitio web. Ese controlador puede ser la página en sí, ya

que a menudo es en entornos de página de servidor, o puede ser un objeto independiente que corresponde a esa página.

- *Front Controller*. Un controlador que se encarga de todas las solicitudes de un sitio Web. Este patrón consolida toda la administración de solicitudes mediante la canalización de solicitudes a través de un único objeto de controlador. Este objeto se puede llevar a cabo un comportamiento común, que se puede modificar en tiempo de ejecución con decoradores. El controlador entonces envía a mandar objetos para comportamiento en particular a una solicitud.

- *Template View*. Representa la información en HTML mediante la incorporación de marcadores en una página HTML. Un buen número de plataformas populares se basan en este modelo, Muchas de las cuales son las tecnologías de páginas de servidor (ASP, JSP, PHP) que le permiten usar un lenguaje de programación en la página.

- *Transform View*. Una vista que procesa los datos de dominio elemento a elemento y lo transforma en HTML. Al emitir las solicitudes de datos a las capas de dominio y desde fuentes de datos, se obtiene toda la información que necesita para cumplir con ellos, pero sin el formato que se necesita para hacer una página web adecuada. El papel de la vista en el *Modelo Vista Controlador* es hacer que estos datos se muestren en una página Web. Transformar significa pensar en esto como una transformación en la que con los datos del modelo como entrada y la página HTML como salida.

- *Two-Step View*. Convierte los datos del dominio en HTML en dos pasos. El primero transforma los datos del modelo en una presentación lógica sin ningún formato específico. El segundo convierte esa presentación lógica al formato real necesario. De esta manera se puede hacer un cambio global en la alteración de la segunda etapa, o se puede soportar múltiples vistas de salida y se siente con una sola segunda etapa cada uno.

- *Application Controller*. Actúa como un punto centralizado para el manejo de la navegación de la pantalla y el flujo de una aplicación. Algunas aplicaciones contienen una cantidad significativa de la lógica acerca de las pantallas a usar en diferentes puntos, lo que puede implicar la invocación de ciertas pantallas en ciertos momentos en una aplicación. Cuando hay muchas decisiones la aplicación se vuelve más compleja y esto puede conducir a código duplicado en varios controladores para diferentes pantallas y tienen que saber qué hacer en una situación determinada. Puede eliminar esta duplicación mediante la colocación de toda la lógica de flujo en un *Application Controller*. El *Application Controller* selecciona los comandos apropiados para su ejecución contra el modelo y selecciona la vista correcta a utilizar dependiendo del contexto de aplicación.

2.1.2.7 Patrones de distribución

- *Remote Facade*. Proporciona una fachada de grano grueso que recubre a los objetos de grano fino para mejorar la eficiencia través de una red. Una *Remote Facade* es una fachada de grano grueso (GoF) sobre una red de objetos de grano fino. Ninguno de los objetos de grano fino tiene una interfaz remota, y la *Remote Facade* no contiene ninguna lógica de dominio. Toda lo que hace la *Remote Facade* es traducir los métodos de grano grueso en los objetos de grano fino subyacentes.

- *Data Transfer Object (DTO)*. Un objeto que transporta datos entre los procesos con el fin de reducir el número de llamadas de método. Cuando se trabaja con una interfaz remota, como *Remote Facade*, cada llamada es costosa. Por tanto, es necesario reducir el número de llamadas, y eso significa que se necesita transferir menos datos en cada llamada. La solución es crear un objeto de transferencia de datos que puede almacenar todos los datos de la llamada. Tiene que ser serializable para ir al otro lado de la conexión. Por lo general, un ensamblador se utiliza en el lado del servidor para transferir datos entre el *DTO* y todos los objetos de dominio.

2.1.2.8 Patrones de concurrencia offline

- *Optimistic Offline Lock*. Supone que la posibilidad de un conflicto de sesión es alta y por lo tanto limita la concurrencia del sistema. *Optimistic Offline Lock* asume que la probabilidad de conflicto es baja. Este bloqueo no evita conflictos, pero sí los detecta, contrastando la versión en memoria del dato a modificar con la versión almacenada en la base de datos.

- *Pessimistic Offline Lock*. Evita los conflictos. Obliga a una transacción de negocios a adquirir un bloqueo sobre una pieza de información antes de empezar a usarla, por lo que, la mayor parte del tiempo, una vez que comience una transacción comercial, dicha transacción poseerá la información bloqueando su acceso a otras transacciones.

- *Coarse Grained Lock*. Bloquea un conjunto de objetos relacionados con un candado. Un bloqueo de grano grueso es un candado que cubre muchos objetos. No sólo simplifica la acción de bloqueo en sí, sino también le libera de tener que cargar todos los miembros de un grupo con el fin de bloquearlos.

- *Implicit Lock*. Permite a un framework o capa de Layer Supertype adquirir un bloqueo offline. El hecho de que la mayoría de las aplicaciones empresariales hagan uso de una combinación de frameworks y Layer Supertypes nos ofrece una gran oportunidad para facilitar el bloqueo implícito.

2.1.2.9 Patrones de estado de sesión

- *Client Session State*. Almacena los datos en el cliente. Hay varias maneras de hacer esto: codificar los datos en una URL para una presentación Web, utilizar cookies, la serialización de los datos en algún campo oculto de un formulario Web, y mantener los datos en los objetos en un cliente. Se aconseja su uso cuando la cantidad de datos del estado de sesión que se necesita guardar es bastante pequeña.

- *Server Session State*. Puede ser tan simple como mantener los datos en la memoria entre las peticiones. Sin embargo, no hay un mecanismo para almacenar el estado de sesión

en algún lugar más duradera que un objeto serializado. El objeto puede ser almacenado en el sistema de archivos local del servidor de aplicaciones, o se puede colocar en un origen de datos compartido. Esto puede ser una tabla de base de datos simple con un ID de sesión como una clave y un objeto serializado como un valor.

- *Database Session State*. Es también almacenamiento en el servidor, pero se trata de dividir los datos en tablas y campos, y almacenarlo en la base de. Este patrón obliga a utilizar la base de datos para conseguir los datos de sesión (y tal vez hacer un poco de transformación también). Esto implica que cada enfoque puede tener diferentes efectos en la capacidad de respuesta del sistema. El tamaño y la complejidad de los datos tendrán un efecto en la aplicación que hay que tener en cuenta.

2.1.2.10 Patrones base

Estos nos son patrones arquitectónicos, sino patrones genéricos que son especializados en el catálogo de Fowler en otros patrones más específicos.

- *Gateway*. Este patrón envuelve un API de comunicación de recursos externos, en una clase cuya interfaz se parece a un objeto de regular, y crea esa puerta de enlace o *Gateway*, que traduce las llamadas a métodos simples en el API especializado adecuado.

- *Mapper*. Es un patrón que implementa un objeto que establece una comunicación entre dos objetos independientes. A veces es necesario configurar las comunicaciones entre los dos subsistemas que aún deben permanecer ignorante uno del otro. Esencialmente un *Mapper* desacopla diferentes partes de un sistema. En general, habrá que elegir entre *Mapper* y *Gateway*. En aplicaciones empresariales encontramos principalmente *Mapper*.

- *Layer Supertype*. Un tipo que actúa como el súper tipo para todos los tipos en su capa. No es raro que para todos los objetos de una capa tenga métodos que no desea tener duplicado en todo el sistema. Por ejemplo, un dominio de la superclase del objeto para todos los objetos de dominio de un *Domain Model*.

- *Separated Interface*. Este patrón define una interfaz en un paquete separado de su implementación. A medida que desarrolla un sistema, se puede mejorar la calidad de su diseño al reducir el acoplamiento entre las partes del sistema. Una buena manera de hacer esto es agrupar las clases en paquetes y controlar las dependencias entre ellos. *Separated Interface* se utiliza para definir una interfaz en un paquete, y ponerlo en práctica en otro. Por ejemplo cuando es necesario llamar a las funciones desarrolladas por otro grupo de desarrollo, pero no quieren una dependencia en sus APIs.

- *Registry*. Un objeto bien conocido que otros objetos pueden utilizar para encontrar objetos y servicios comunes. Un registro es esencialmente un objeto global, o al menos eso parece para algunos. Una lista de todos los estados en los Estados Unidos hace es buen candidato para un proceso de ámbito de *Registry*. Estos datos se pueden cargar cuando un proceso se inicia y no necesitan cambiar, o ser actualizados raramente con algún tipo de alarma de proceso.

- *Value Object*. Representa un simple objeto, como el dinero o un rango de fechas, valido para el uso de objetos cuya igualdad no se basa en la identidad. Es útil distinguir entre objetos de referencia y objetos de valor. La diferencia clave entre referencia y de valor los objetos radica en cómo se ocupan de la igualdad. Una referencia de objeto utiliza la identidad como base de la igualdad. Un valor de objeto basa su noción de igualdad en campo de los valores dentro de la clase.

- *Money*. Representa un valor monetario. Una gran proporción de las computadoras en este mundo manipulan dinero, La falta de un tipo moneda en los lenguajes de programación causa problemas, por lo menos para las monedas más utilizadas. Si todos los cálculos se hacen en una sola moneda, esto no es un gran problema, pero una vez que involucras varias monedas se desea evitar la adición de dólares a yen sin tomar las diferencias monetarias en cuenta. El problema es más sutil con el redondeo. Cálculos monetarios a menudo redondean a la unidad monetaria más pequeña. Al hacer esto, es fácil perder centavos de una de las monedas (o su equivalente en moneda local), debido a

errores de redondeo. Lo bueno de la programación orientada a objetos es que se puede solucionar estos problemas mediante la creación de una clase dinero que los maneja.

- *Special Case*. Es una subclase que proporciona el comportamiento especial para casos particulares. Si es posible que una variable a devuelva un nulo, hay que recordar verificar el código en caso de ser null y a menudo, esto en muchos contextos, por lo que se acaba por escribir código similar en muchos lugares y cometer el error de la duplicación de código. En lugar de devolver null, o algún valor impar, usamos este patrón para devolver un caso especial que tiene la misma interfaz que lo que se espera de la llamada.

- *Plugin*. Proporciona configuración de ejecución centralizada para casos en los que se quiere utilizar un entorno de pruebas o un entorno de producción con parámetros de configuración diferentes.

- *Service Stub*. Elimina la dependencia de los servicios problemáticos durante la prueba. Este patrón resuelve el problema de la dependencia de resultados de servicios externos y que durante el desarrollo no se pueden probar o retrasan el test. La sustitución del servicio durante las pruebas con un *Service Stub* que se ejecuta localmente, rápido, y en memoria mejora la experiencia de desarrollo.

- *Record Set*. Una representación en memoria de los datos de la tabla. La idea del Record Set es proporcionar una estructura en memoria que se ve exactamente como el resultado de una consulta SQL, pero puede ser generado y manipulado por otras partes del sistema.

2.1.3 Design patterns: Elements of reusable object oriented software (GoF)

El libro “Design Patterns: Elements of Reusable Object Oriented Software” (Gamma et al. 1994) describe soluciones simples y elegantes a problemas específicos en el diseño de software orientado a objetos. Presenta un catálogo de veintitrés patrones. Esta obra ha sido escrita por un grupo de cuatro autores a los cuales se conoce como “Gang of Four” (GoF) o en

español “La banda de los cuatro”. En este libro los patrones se clasificarían según el propósito para el que han sido definidos:

2.1.3.1 Patrones de creación

Patrones de diseño software que solucionan problemas de creación de instancias. Ayudan a encapsular y abstraer dicha creación.

- *Abstract Factory*. Proporciona una interfaz para crear familias de objetos o que dependen entre sí, sin especificar sus clases concretas.

- *Builder*. Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

- *Factory Method*. Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar. Permite que una clase delegue en sus subclases la creación de objetos.

- *Prototype*. Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crear nuevos objetos copiando este prototipo.

- *Singleton*. Garantiza que una clase sólo tenga una instancia, y proporciona un punto de acceso global a ella.

2.1.3.2 Patrones estructurales

Son los patrones de diseño software que solucionan problemas de composición (agregación) de clases y objetos.

- *Adapter*. Convierte la interfaz de una clase en otra distinta que es la que esperan los clientes. Permiten que cooperen clases que de otra manera no podrían por tener interfaces incompatibles.

- *Bridge.* Desvincula una abstracción de su implementación, de manera que ambas puedan variar de forma independiente.

- *Composite*. Combina objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.

- *Decorator.* Añade dinámicamente nuevas responsabilidades a un objeto, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.

- *Facade.* Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

- *Flyweight.* Usa el compartimiento para permitir un gran número de objetos de grano fino de forma eficiente.

- *Proxy*. Proporciona un sustituto o representante de otro objeto para controlar el acceso a éste.

2.1.3.3 Patrones de comportamiento

Son patrones de diseño software que ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan.

- *Chain of Responsibility*. Evita acoplar el emisor de una petición a su receptor, al dar a más de un objeto la posibilidad de responder a la petición. Crea una cadena con los objetos receptores y pasa la petición a través de la cadena hasta que esta sea tratada por algún objeto.

- *Command*. Encapsula una petición en un objeto, permitiendo así declarar los parámetros de clientes con distintas peticiones, encolar o llevar un registro de las peticiones y poder deshacer las operaciones.

- *Interpreter*. Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar las sentencias del lenguaje.

- *Iterator*. Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.

- *Mediator*. Define un objeto que encapsula cómo interactúan un conjunto de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.

- *Memento*. Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste puede volver a dicho estado más tarde.

- *Observer*. Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambia de estado se notifica y actualizan automáticamente todos los objetos.

- *State*. Permite que un objeto modifique su comportamiento cada vez que cambia su estado interno. Parecerá que cambia la clase del objeto.
- *Strategy*. Define una familia de algoritmos, encapsula uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.

- *Template Method*. Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos del algoritmo sin cambiar su estructura.

- *Visitor*. Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.

2.1.4 Otros

Existe alguna otra bibliografía relacionada con los patrones de diseño que mencionaremos sin detallar cada uno de sus patrones, pero si mencionando muy a groso modo la forma en que los clasifican.

2.1.4.1 Pattern - Oriented software architecture. A system of patterns

Este libro "Pattern-Oriented Software Architecture Volume 1" (Buschmann et al. 1996) representa la progresión y la evolución del enfoque de patrones en un sistema capaz de describir y documentar aplicaciones a gran escala.

Contiene 17 patrones generales de uso arquitectónicos, patrones de diseño y modismos clasificados así:

- *Structural Decomposition:* Layers, Blackboard, Pipes and Filters, and Whole Part
- *Distributed Systems:* Broker, Forwarder- Receiver, and Client-Dispatcher-Server
- *Interactive Systems:* Model-View-Controller and Presentation-Abstraction-Control
- *Adaptive Systems:* Microkernel, Reflection
- *Organization of Work:* Master Slave
- *Service Access:* Proxy
- *Resource Management:* Counted Pointer, Command Processor, and View Handler

Communication: Publisher-Subscriber

2.1.4.2 J2EE Design Patterns

En el libro "J2EE Design Patterns" (Crawford y Kaplan 2003) describe 50 patrones que ofrecen una amplia cobertura de las cinco áreas de problemas que enfrentan los desarrolladores de aplicaciones empresariales:

- *Mantenimiento (extensibilidad)*
- *Rendimiento (escalabilidad del sistema)*
- *Data Modeling (Modelado de Objetos de Negocios)*
- *Transacciones (Modelado de procesos)*
- *Mensajería (interoperabilidad)*

En lugar de presentar otro catálogo de patrones, en este libro los autores discuten diferentes aspectos del desarrollo de J2EE y presentan a los patrones en su contexto. Hasta cierto punto es un catálogo híbrido entre CJP y el catálogo de Martin Fowler.

2.2 Catálogos de patrones de seguridad

Los patrones de seguridad son una abstracción de los problemas de seguridad en aplicaciones empresariales que abordan una variedad de requisitos de seguridad y proporcionan una solución al problema. Estos patrones muestran cómo un problema de seguridad puede resolverse arquitectónicamente, o pueden ser el diseño de estrategias defensivas sobre el cual se puede construir después con código.

2.2.1 Core Security Patterns

El libro "*Core Security Patterns"* (Steel et al. 2005) expone una colección de patrones de diseño probados para garantizar seguridad de extremo a extremo en aplicaciones J2EE, servicios web, gestión de identidades y aprovisionamiento de servicios.

Estos patrones de seguridad abordan los requisitos de seguridad de una aplicación de extremo a extremo mitigando los riesgos de seguridad en el nivel funcional y despliegue, asegurando los objetos de negocio y datos a través de las capas lógicas, asegurando las comunicaciones y protegiendo la aplicación de amenazas y vulnerabilidades internas y externas no autorizadas.

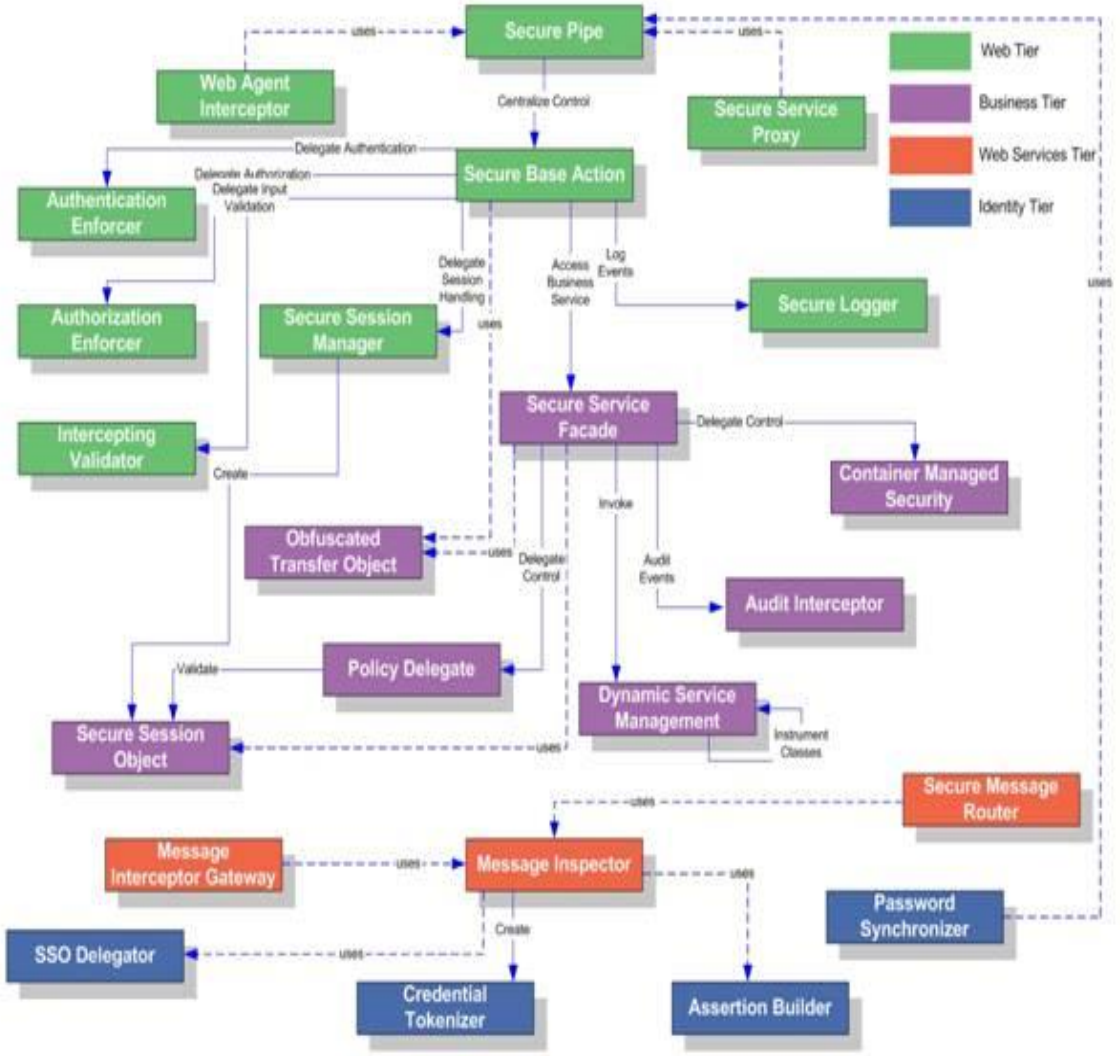


Figura 2.3 Catalogo Core Security Patterns

2.2.1.1 Capa web

La capa web puede ser considerada como sinónimo de la capa de presentación para las aplicaciones web. El catálogo CSP identifica ocho patrones en la capa web:

- *Authentication Enforcer*. Este patrón permite crear una aplicación con autenticación centralizada que realiza la autenticación de usuarios y encapsula los detalles del mecanismo de autenticación a través de todas las acciones en el nivel web.

- *Authorization Enforcer*. Crea una aplicación con un controlador de acceso que realiza comprobaciones de autorización centralizadas y encapsula los detalles de dichos mecanismos, mediante la definición de una forma estandarizada para controlar el acceso a aplicaciones basadas en Web.

- *Intercepting Validator*. Implementa un mecanismo sencillo que permite analizar y validar los datos transmitidos desde el cliente a la capa web de la aplicación, similar al patrón CJP *Intercepting Filter*. Las reglas en este patrón pueden ser configurables, pero en este caso el filtrado se enfoca fundamentalmente en validar si la petición debe continuar o ser denegada.

- *Intercepting Web Agent*. Este patrón es una solución alternativa cuando se desea proporcionar autenticación y autorización en las peticiones a una aplicación que puede estar en las siguientes condiciones:
  - La aplicación no tiene seguridad alguna y se prefiere usar un proveedor externo que haga esta tarea.
  - La aplicación de construyó anteriormente con una arquitectura de seguridad existente, y se prefiere no modificarla.

- *Secure Base Action*. Centraliza la lógica de seguridad de la capa web. Además, separa la lógica de presentación de la lógica de seguridad, utilizando el mecanismo de seguridad adecuado en cada acción, y reutilizando estos métodos en muchos puntos de la aplicación, proporcionando un punto central para la administración de la seguridad relacionada con la funcionalidad.

- *Secure Logger*. Es un patrón gestor de logs especializado en centralizar y proteger la integridad de datos. Registra eventos u operaciones que requieren de un tratamiento confidencial, permitiendo gestionar a diferentes niveles de detalle los datos confidenciales durante el ciclo de vida de las operaciones de la aplicación.

- *Secure Pipe*. Proporciona una manera simple y estandarizada de proteger los datos enviados en una red de procesos formados por distintas actividades bien definidas.

- *Secure Service Proxy*. Permite agregar seguridad a aplicaciones heredadas o antiguas sin tener que modificar el código fuente. Permite proteger los puntos finales de servicios Web de los intrusos aprovechando productos de seguridad de terceros o desarrollos de infraestructura de seguridad J2EE desarrollada para tal fin.

- *Secure Session Manager*. Este patrón describe cómo crear una sesión segura entre el cliente y el servidor o entre servidores. Es importante para proteger la información de una transacción comercial durante una sesión. Se recomienda usar este patrón en conjunto con el patrón *Secure Pipe*.

2.2.1.2 Capa de negocio

- *Audit interceptor*. Este patrón trabaja en conjunto con el patrón *Secure Logger*. *Audit Interceptor* trabaja en el back- end y *Secure Logger* en el front-end. Este patrón registra y audita tanto peticiones como respuestas de forma centralizada a través de un catálogo de eventos definidos previamente.

- *Container Managed Security*. Este patrón describe cómo declarar la información relacionada con la seguridad en un descriptor de despliegue. Es una manera simple y estándar para aplicar la autenticación y autorización de usuarios a un nivel de granularidad deseado en las aplicaciones J2EE.

- *Dynamic Service Management*. Este patrón proporciona instrumentos ajustables dinámicamente de los componentes de seguridad para el control y la gestión activa de los objetos de negocio.

- *Obfuscated Transfer Object*. Describe la manera de proteger los datos de negocio representados en *Transfer Objects* y transmitidos dentro y entre capas lógicas. No

todos los datos de un objeto deben ser protegidos, pero si los que nos interesan o son confidenciales.

- *Policy Delegate*. Crea, gestiona y administra las políticas de gestión de seguridad que rigen el acceso y direccionamiento desde y hacia objetos EJB. Este modelo es un coordinador de servicios de seguridad en la capa de negocio.

- *Secure Service Façade*. Este patrón ofrece una fachada de sesión que puede contener y centralizar las interacciones complejas entre los componentes de negocio bajo una sesión segura. Proporciona seguridad dinámica y declarativa en el back-end de los objetos de negocio en la fachada del servicio.

- *Secure Session Object*. Este patrón define formas de proteger la información de sesión de EJB que facilitan un acceso distribuido y evitar la propagación de fisuras del contexto de seguridad. Este modelo permite encapsular de manera abstracta las credenciales de autorización y autenticación de un contexto para traspasar fronteras en entornos distribuidos.

2.2.1.3 Capa de servicios web

- *Message Inspector*. Este patrón comprueba y verifica la calidad y mecanismos de seguridad a nivel de mensaje XML, tales como *XML Signature* y *XML Encryption* trabajando en conjunto con un Security Token. El patrón *Message Interceptor* también ayuda en la verificación y validación de los mecanismos de seguridad aplicados a mensajes SOAP y procesados por múltiples intermediarios (actores). Es compatible con una variedad de formatos de firma y tecnologías de cifrado utilizados por estos intermediarios.

- *Message Interceptor Gateway*. Este patrón proporciona un único punto de entrada y permite la centralización de la seguridad de los mensajes entrantes y salientes de la aplicación. Las funciones de seguridad incluyen la creación, modificación y gestión de políticas de seguridad para el envío y recepción de mensajes SOAP. Permite aplicar los

mecanismos de seguridad a nivel de mensaje y a nivel de transporte necesarios para comunicarse con los puntos finales de los *Web Service*s.

- *Secure Message Router*. Este patrón facilita la comunicación segura XML con varios puntos finales asociados que adoptan la seguridad a nivel de mensaje y mecanismos de federación de identidad. Actúa como un elemento intermedio de seguridad que aplica mecanismos de seguridad a nivel de mensaje para enviar mensajes a varios destinatarios asegurando que el receptor previsto pueda tener acceso a sólo una parte necesaria del mensaje y los fragmentos restantes de los mensajes sean confidenciales.

2.2.1.4 Capa de identidad

- *Assertion builder*. El patrón define la forma en que se puede construir una aserción de identidad; por ejemplo, la afirmación de autenticación o autorización afirmación. Encapsula la lógica de control de procesamiento con el fin de crear las instrucciones SAML de autenticación, estados de decisión de autorización y declaración de atributos como un servicio.

- *Credential Tokenizer*. Hay diferentes formas de credenciales de usuario (también conocida como security token), tales como nombre de username/passwords, *Binary Security Tokens* (por ejemplo, certificados X.509v3), tickets Kerberos, tokens SAML, token de tarjetas inteligentes y muestras biométricas. El patrón *Credential Tokenizer* describe cómo puede ser encapsulados, incrustados en un mensaje, direccionados, y procesados los principales tokens de seguridad en una infraestructura flexible que puede ser reutilizada por diferentes proveedores.

- *Single Sign-On Delegator*. Describe cómo construir un agente delegado para el manejo de inicio de sesión único (SSO) en un sistema heredado. Un *Single Sign-On Delegator* reside en el nivel intermedio entre los clientes y la gestión de identidad de componentes de servicio. Delega la solicitud de servicio de los componentes de los servicios remotos.

Oculta los detalles de la invocación de servicios, la recuperación de la configuración de seguridad, o procesamiento del cliente *Credential Tokenizer*.

- *Password Synchronizer*. Este patrón describe una interfaz de programación segura para centralizar la sincronización de las credenciales de usuarios a través de múltiples aplicaciones en las cuales está dado de alta el usuario.

2.2.2 Security patterns in practice: Designing secure architectures using software patterns

Es un libro escrito por Eduardo Fernández-Buglioni (Fernández 2013), en el que se analiza la estructura y el propósito de los patrones de seguridad, con numerosos ejemplos de código y las descripciones en UML. Los patrones que se presentan en este libro se agrupan por la característica de diseño que se quiere asegurar, y dentro de éste, por el nivel de arquitectura en la que se utilizarían. Los niveles de arquitectura y las preocupaciones de seguridad son dos dimensiones posibles, ambos usados en este libro.

2.2.2.1 Patrones de gestión de identidad

Los patrones de este capítulo se enfocan en los modelos de seguridad que las aplicaciones necesitan para brindan a los usuarios un acceso e identificación a un sistema y/o a una gama amplia de aplicaciones que no puede saberse con anticipación, por lo tanto hay una necesidad de creación de confianza dinámica y un intercambio de identidad sobre protocolos que el modelo de seguridad debe ser capaz de soportar. Estos patrones pueden ser fácilmente utilizados en el ciclo de desarrollo de software para ayudar a producir un software más seguro.

- *Circle of Trust*. Permite la formación de relaciones de confianza entre los proveedores de servicios para permitir a sus sujetos acceder a un entorno integrado y más seguro.

- *Identity Provider*. Permite la centralización de la administración de la información de identidad de los sujetos para un dominio de seguridad.

- *Identity Federation*. Permite la formación de una identidad creada dinámicamente dentro de una federación de identidades que consta de varios proveedores de servicios. Por lo tanto, la identidad y la información de seguridad acerca de un tema se pueden transmitir de una manera transparente para el usuario entre los proveedores de servicios de diferentes dominios de seguridad.

- *Liberty Alliance Identity Federation*. Permite la fusión de las identidades a través de múltiples organizaciones bajo una identidad federada y siguiendo un conjunto de reglas comunes.

2.2.2.2 Patrones de autenticación

El capítulo anterior describía patrones para la identificación de los usuarios en un sistema, la mayoría de los sistemas además de necesitar que los usuarios se identifiquen necesita que los usuarios se autentiquen, es decir que demuestre que ellos son quienes dicen son, y para ello se definen los siguientes patrones en este capítulo.

- *Authenticator*. Cuando un sujeto se identifica ante el sistema, el patrón *Authenticator* permite la verificación de que el tema con la intención de acceder al sistema es quién o lo que pretende ser.

- *Remote Authenticator Authorizer*. Proporciona facilidades para la autenticación y autorización al acceder a los recursos compartidos en un sistema distribuido.

- *Credential*. Este patrón proporciona un medio seguro para la autenticación de registro y la información de autorización para su uso en sistemas distribuidos.

2.2.2.3 Patrones de control de acceso

Una vez que se ha concedido el acceso a un sistema, se debe controlar su acceso a recursos específicos. Los patrones de control de acceso ayudan a definir las reglas y políticas que

se utilizan como guía para la asignación de los derechos que los usuarios tienen sobre los recursos.

- *Authorization*. También conocido como *Access Matrix*. El patrón describe quién está autorizado para acceder a determinados recursos en un sistema, en un entorno en el que contamos con recursos cuyo acceso debe ser controlado. El modelo indica, para cada materia activa, que los recursos del sujeto pueden acceder y lo que puede hacer con ellos.

- *Role-Based Access Control*. Describe cómo asignar los derechos sobre la base de las funciones o tareas de los usuarios en un entorno en el que se requiere el control de acceso a los recursos informáticos.

- *Multilevel Security*. En algunos entornos los datos y documentos pueden tener un valor crítico y su divulgación podría traer problemas serios. El patrón de niveles múltiples de seguridad describe cómo clasificar la información sensible y evitar su divulgación. En él se describe cómo asignar clasificaciones (espacios libres) a los usuarios y las clasificaciones (niveles de sensibilidad) a los datos y la forma de separar las diferentes unidades organizativas en categorías. El acceso de los usuarios a los datos se basa en las políticas, mientras que los cambios en las clasificaciones se realizan mediante procesos de confianza que pueden modificar las políticas.

- *Policy-Based Access Control*. Describe cómo decidir si un sujeto está autorizado a acceder a un objeto de acuerdo a las políticas definidas en la política de un repositorio central.

- *Access Control List (ACL)*. Permite acceso controlado a los objetos indicando que los sujetos pueden acceder a un objeto y de qué manera. No es por lo general una ACL asociada con cada objeto.

- *Capability*. Permite el acceso controlado a los objetos, proporcionando una credencial o billete a un sujeto para permitir que se acceda a un objeto de una manera específica.

- *Reified Reference Monitor*. También conocido como *Intercepting Filter*, *Application Controller*. Describe cómo forzar autorizaciones cuando un sujeto solicita un objeto de protección y proporcionar al sujeto con una decisión. En un entorno de computación en el que los usuarios o procesos hacen peticiones de datos o recursos, este patrón describe cómo definir un proceso abstracto que intercepta todas las solicitudes de recursos de los sujetos y los controles del cumplimiento de las autorizaciones.

- *Controlled Access Session*. Describe cómo proporcionar un contexto en el cual un sujeto (usuario del sistema) puede acceder a los recursos con diferentes derechos sin necesidad de volver a autenticarse cada vez que acceda a un nuevo recurso.

- *Session-Based Role-Based Access Control*. Permite el acceso a los recursos basados en el papel del sujeto, y limita los derechos que se pueden aplicar en un momento determinado en base a las funciones definidas por la sesión de acceso.

- *Security Logger and Auditor*. También conocido como *Audit Trail*. El patrón describe cómo realizar un seguimiento de las acciones de los usuarios con el fin de determinar quién hizo, qué y cuándo. Registra todas las acciones sensibles a la seguridad realizadas por los usuarios y proporciona acceso controlado a los registros con fines de auditoría.

2.2.2.4 Patrones de seguridad para gestión de procesos

Los sistemas operativos son fundamentales para la provisión de seguridad a los sistemas, El sistema operativo también debe protegerse a sí mismo, porque compromete el acceso a todas las cuentas de usuario y todos los datos en sus archivos. Este capítulo describe los patrones de seguridad básicos para brindar seguridad sobre los procesos del sistema operativo.

- *Secure Process/Thread*. Describe cómo asegurarse de que el proceso no interfiera con otros procesos o recursos compartidos de uso indebido. Un proceso es un programa en ejecución; un proceso seguro es también una unidad de aislamiento de ejecución, así como el titular de los derechos a acceder a los recursos, y tiene un espacio de direcciones virtual independiente. Un hilo es un proceso ligero. Una variante, una hilo seguro, es un hilo con el acceso controlado a los recursos.

- *Controlled-Process Creator*. Describe cómo definir los derechos que debe darse a los nuevos procesos mediante la definición de los derechos como parte de la creación de procesos.

- *Controlled-Object Factory*. Describe cómo especificar los derechos de los procesos con respecto a un nuevo objeto. Cuando un proceso crea un nuevo objeto a través de una fábrica, la solicitud incluye las características del nuevo objeto. Entre estas características se incluye una lista de derechos de acceso al objeto.

- *Controlled-Object Monitor*. Permite el control de acceso de un sujeto a un objeto, utilizando un monitor de referencia especializado para interceptar las solicitudes de acceso a partir de los procesos. El monitor de referencia comprueba si el proceso tiene el tipo solicitado de acceso al objeto.

- *Protected Entry Points*. Describe cómo forzar una llamada de un proceso a otro para ir sólo a través de puntos de entrada predefinidos donde se comprueba la veracidad de la llamada y pueden aplicarse otras restricciones de acceso.

- *Protection Rings*. Permite el control de cómo los procesos llaman a otros procesos y cómo acceder a los datos. El cruce de anillos se realiza a través de puertas que compruebe los derechos del proceso de cruce. Un proceso de llamar a otro proceso o acceder a los datos en un anillo superior debe pasar por una puerta.

2.2.2.5 Patrones de seguridad para ejecución y administración de archivos

En este apartado se describen las pautas para la ejecución segura de los procesos que tienen que ver con recursos. Estos son representados como objetos, como es común en los sistemas operativos modernos. Por ejemplo, se necesita autenticación para el acceso a archivos y acceso controlado al objeto, un sujeto debe estar autorizado a acceder a un objeto de una manera específica, y se debe asegurar que el solicitante no es un impostor.

- *Virtual Address Space Access Control*. Permite el control de acceso de los procesos a las áreas específicas de su espacio de direcciones virtuales (EAV) de acuerdo con un conjunto de tipos de acceso predefinido.

- *Execution Domain*. Describe cómo definir un entorno de ejecución de los procesos. Indica de forma explícita todos los recursos de un proceso que pueden usarse durante su ejecución, así como el tipo de acceso a los recursos.

- *Controlled Execution Domain*. Permite el control de acceso a todos los recursos del sistema operativo por procesos, basado en usuario, grupo o rol.

- *Virtual Address Space Structure Selection*. Describe cómo seleccionar el espacio de direcciones virtuales para sistemas operativos que tienen necesidades especiales de seguridad. Algunos sistemas enfatizan el aislamiento, el intercambio de información a otros, otros un buen rendimiento. La organización del espacio de direcciones virtuales cada proceso (EVA) está definida por la arquitectura de hardware y tiene un efecto sobre el rendimiento y la seguridad. El patrón permite todas las posibilidades del hardware que deben ser consideradas y seleccionadas de acuerdo con las necesidades.

2.2.2.6 Patrones de seguridad para arquitectura y administración de sistemas operativos

La seguridad de las acciones individuales en tiempo de ejecución, tales como la creación de procesos y la protección de la memoria son muy importantes, y ya hemos visto los patrones de estas funciones en el apartado 2.2.2.4 y 2.2.2.5. Sin embargo, la arquitectura general del sistema

operativo también es muy importante para la capacidad del sistema para proporcionar un entorno de ejecución seguro. Esta categoría incluye los patrones que representan las arquitecturas más utilizadas por los Sistemas operativos:

- *Modular Operating System Architecture*. Describe cómo separar los servicios del sistema operativo en módulos, cada uno representando una función básica o componente. El núcleo central básico sólo tiene los componentes necesarios para empezar a sí mismo y la capacidad de cargar módulos. El núcleo es el módulo de siempre en la memoria. Cuando se requieren los servicios de cualquiera de los módulos adicionales, el cargador de módulos carga el módulo adecuado. Cada módulo realiza una función y puede tomar parámetros.

- *Layered Operating System Architecture*. Permite implementar las características generales y la funcionalidad del sistema operativo que se descomponen y se asignan a las capas jerárquicas. Esto proporciona interfaces claramente definidas entre cada sección del sistema operativo y entre aplicaciones de usuario y las funciones del sistema operativo. La capa i utiliza los servicios de una capa inferior i-1 y no sabe de la existencia de una capa superior, i+1.

- *Microkernel Operating System Architecture*. Describe cómo mover la mayor cantidad de la funcionalidad del sistema operativo como sea posible del núcleo en servidores especializados, coordinados por un micro kernel. El micro kernel en sí tiene un conjunto muy básico de funciones. Componentes y servicios del sistema operativo se implementan como servidores externos e internos.
- *Virtual Machine Operating System Architecture*. Describe cómo proporcionar un conjunto de réplicas de la arquitectura de hardware (máquinas virtuales) que se pueden utilizar para ejecutar sistemas operativos múltiples y posiblemente diferentes con un fuerte aislamiento entre ellos.

- *Administrator Hierarchy*. Muchos de los ataques provienen del poder ilimitado de los administradores. Este patrón permite limitar el poder de los administradores mediante

la definición de una jerarquía de administradores de sistemas con derechos controlados usando un control de acceso basado en roles (RBAC) de modelo, y les asigna derechos de acuerdo a sus funciones.

- *File Access Control*. Permite el control de acceso a archivos en un sistema operativo. Los usuarios autorizados son los únicos que pueden utilizar un archivo de manera específica.

2.2.2.7 Patrones de seguridad para redes

Los siguientes son los patrones de seguridad para definir un modelo de referencia y marcar las pautas de seguridad en cada una de las capas de las redes, dando seguridad a las cuatro capas: aplicación, transporte, Internet, y enlace.

- *Abstract Virtual Private Network*. Describe cómo configurar un canal seguro entre dos puntos finales utilizando un túnel cifrado con autenticación en cada punto final. Un extremo es una interfaz expuesta por una unidad de comunicación (sitio del usuario o de la red).

- *IPSec VPN*. Describe cómo configurar un canal seguro entre dos puntos finales utilizando un túnel cifrado a través de la capa IP, con la autenticación en cada punto final.

- *TLS Virtual Private Network*. También conocido como *SSL VPN*. Este patrón describe cómo configurar un canal seguro entre dos puntos finales utilizando un túnel cifrado a través de la capa de transporte, con la autenticación y autorización en cada punto final.

- *Transport Layer Security*. Describe cómo proporcionar un canal seguro entre un cliente y un servidor por el cual los mensajes de aplicación se comunican a través de

la capa de transporte de la Internet. El cliente y el servidor están mutuamente autenticados y la integridad de sus datos se conserva.

- *Abstract IDS*. Permite monitorizar todo el tráfico que pasa a través de una red, y su análisis permite detectar posibles ataques y desencadenar una respuesta adecuada.

- *Signature-Based IDS*. También conocido como *Rule Based IDS*, *Knowledge-Based IDS*. Este patrón describe cómo comprobar todas las solicitudes de acceso a la red con un conjunto de firmas de ataques existentes, para detectar posibles ataques y desencadenar una respuesta adecuada.

- *Behavior-Based IDS*. También conocido como *Anomaly-Based IDS*. Este patrón describe cómo comprobar todas las solicitudes de acceso contra los patrones de tráfico de la red con el fin de detectar posibles desviaciones del comportamiento normal (anomalías) que pueden indicar un ataque y desencadenar respuestas apropiadas.

2.2.2.8 Patrones de seguridad para servicios web

Esta categoría define una serie de patrones enfocados a proteger aplicaciones que presentan Arquitecturas Orientadas a Servicios (SOA- Service-Oriented Architectures) y Servicios Web. Una organización necesita definir las políticas de seguridad, que son directrices de alto nivel que especifican los estados en los que el sistema es considerado como seguro. Una forma de permitir la interoperabilidad, aplicar la seguridad, y exigir el cumplimiento de la normativa es a través del uso de estándares que definen las arquitecturas para garantizar que todos los participantes sigan las mismas reglas en sus interacciones. Muchos de los siguientes patrones se han identificado en la comunidad de los servicios web, a varios niveles de granularidad.

- *Application Firewall*. También conocido como *Content Firewall*. Este patrón permite el filtrado de llamadas y respuestas a / desde las aplicaciones empresariales, con base en las políticas de control de acceso de una institución.

- *XML Firewall*. Permite el filtrado de mensajes XML a/de las aplicaciones empresariales, con base en las políticas de control de acceso de las empresas y el contenido del mensaje.

- *XACML Authorization*. Puede ser utilizado por una organización para representar las reglas de autorización de una manera estándar.

- *XACML Access Control Evaluation*. Describe cómo resolverá si la solicitud está autorizada a acceder a un recurso de acuerdo a las políticas definidas por el patrón *XACML Authorization*.

- *Web Services Policy Language*. Describe cómo representar las políticas de control de acceso para los servicios web de la organización de una manera estándar, y para permitir a un consumidor de servicios de Internet para expresar sus necesidades de una manera estándar.

- *WS-Policy*. Describe cómo definir un conjunto base de afirmaciones que se pueden usar y se extienden por otras especificaciones de servicios web para describir una amplia gama de requisitos y capacidades de servicio, incluida la seguridad, fiabilidad y demás. Este modelo también proporciona una manera de comprobar las solicitudes formuladas por los solicitantes a fin de verificar que cumplen sus afirmaciones y sus condiciones antes de interactuar con un servicio web.

- *WS-Trust*. Describe cómo definir un servicio de token de seguridad y un motor de confianza que son utilizados por los servicios de Internet para autenticar otros servicios web. Usando las funciones definidas en este patrón, las aplicaciones pueden entablar una comunicación segura después de establecer la confianza.

- *SAML Assertion*. Describe cómo proporcionar una manera de comunicar información de seguridad sobre un tema en particular entre los diferentes dominios de seguridad.

2.2.2.9 Patrones de criptografía de servicios web

Esta sección incluye patrones que sirven para proteger la información con el fin de que ésta no sea capturada y leída durante su transmisión.

- *Symmetric Encryption*. El cifrado protege la confidencialidad del mensaje al hacer un mensaje ilegible para aquellos que no tienen acceso a la clave. El cifrado simétrico utiliza la misma clave para el cifrado y el descifrado.

- *Asymmetric Encryption*. El cifrado asimétrico proporciona la confidencialidad del mensaje por mantener la información secreta de una manera tal que sólo puede ser entendida por los destinatarios pretendidos que tienen el acceso a la clave válida. En el cifrado asimétrico, un par de claves pública/privada se utilizan para el cifrado y el descifrado, respectivamente.

- *Digital Signature with Hashing*. Permite demostrar que un mensaje se originó a partir de una fuente de confianza. También proporciona la integridad del mensaje, mediante la indicación de si un mensaje ha sido alterado durante la transmisión.

- *XML Encryption*. Proporciona confidencialidad al ocultar la información sensible seleccionada en un mensaje utilizando la criptografía.

- *XML Signature*. Permite demostrar que un mensaje se originó a partir de una fuente de confianza. También proporciona la integridad del mensaje mediante la detección de si un mensaje ha sido alterado durante la transmisión. El estándar *XML Signature* (W3C08) describe la sintaxis y el proceso de generación y validación de firmas

digitales para autenticar documentos XML. *XML Signature* también proporciona la integridad del mensaje, y requiere de canonización antes de hash y firma.

- *WS-Security*. El estándar *WS-Security* (OAS2007) describe cómo incrustar los mecanismos de seguridad existentes, tales como *XML Encryption* (W3C2002), *XML-Signature* (W3C2008) y tokens de seguridad digitales XML en los mensajes SOAP para proporcionar confidencialidad del mensaje, integridad, autenticación y no repudio.

2.2.2.10 Patrones de seguridad para middleware

Middleware típicamente incluye un conjunto de funciones que proporcionan servicios a las aplicaciones, incluidos los aspectos distribuidos, como la de brokering, así como servicios específicos, tales como blackboards, pipes y filters, adaptadores y otros. Middleware también puede incluir servicios globales tales como la autenticación, autorización y otros servicios. Esta sección incluye patrones para agregar seguridad a la capa de middleware.

- *Secure Broker*. Extiende el patrón *Broker* para proporcionar interacciones seguras entre componentes distribuidos.

- *Secure Pipes and Filters*. Describe cómo proporcionar un manejo seguro de los flujos de datos. Cada paso del proceso se aplica alguna transformación o el filtrado de datos. Los derechos para aplicar transformaciones a los datos específicos pueden ser controlados. La comunicación de datos entre etapas puede ser también protegida. Las operaciones aplicadas se pueden registrar.

- *Secure Blackboard*. Describe cómo proporcionar un manejo seguro de los datos cuando se accede a una pizarra por clientes conocidos. Cada cliente lee los datos de la pizarra, aplica algún tipo de procesamiento o transformación de datos y actualiza la pizarra. Con el fin de evitar violaciones de integridad y confidencialidad, los derechos a la lectura y actualización de datos se controlan de acuerdo con sus derechos

predefinidos, y sus acciones se registran. Los clientes se autentican antes de poder acceder a la pizarra.

- *Secure Adapter*. También conocido como *Secure Wrapper*. Este patrón describe cómo convertir la interfaz de una clase existente en una interfaz más conveniente, preservando al mismo tiempo la seguridad de la entidad adaptada.

- *Secure Three-Tier Architecture*. Proporciona un medio de estructuración y de descomposición aplicaciones en niveles o capas en la que cada nivel ofrece un nivel diferente de la responsabilidad. Una tiene que ver con el nivel de la parte de presentación del sistema (interfaces de sistema y de usuario), otro se ocupa de la lógica de negocio y el núcleo del sistema, y el último nivel se ocupa del almacenamiento de datos. El patrón extiende el patrón de arquitectura de tres niveles mediante la aplicación de una visión global de la seguridad de las tres capas. En la parte de presentación del sistema, los aspectos de seguridad que se ocupan de la interacción del usuario se aplican, en la lógica de negocio, se aplican restricciones de seguridad a nivel mundial, mientras que el almacenamiento de datos se aplica políticas para restringir el acceso de los usuarios a los datos.

- *Secure Enterprise Service Bus*. Describe cómo proporcionar una infraestructura conveniente para integrar una variedad de servicios distribuidos y componentes relacionados de una manera sencilla y segura.

- *Secure Distributed Publish/Subscribe*. Describe cómo desacoplar los editores de los eventos de los interesados en los acontecimientos (abonados) en un sistema distribuido. Suscripción y publicación se realizan de forma segura.

- *Secure Model-View-Controller*. El patrón describe cómo agregar seguridad a las interacciones de los usuarios con los sistemas configurados con el patrón *Model-View-Controller*.

2.2.2.11 Patrones de uso indebido

Un patrón de uso indebido describe, desde el punto de vista del atacante, como se lleva a cabo un tipo de ataque (las unidades que utiliza y cómo), analiza las maneras de detener el ataque mediante la enumeración de posibles patrones de seguridad que se pueden aplicar para este fin, y describe cómo rastrear el ataque una vez que ha ocurrido por la recolección y observación de datos forenses apropiados. También describe precisamente el contexto en el que puede ocurrir el ataque. Los patrones de uso indebido describen ataques específicos en un entorno determinado, tales como VoIP, servicios web, etc.

- *Worm*. Describe cómo un gusano puede propagarse al mayor número de lugares posible (o sistemas específicos), por lo general lo que indica su presencia, y tal vez la realización de algún daño.

- *Denial-of-Service in VoIP*. Un ataque *DoS VoIP* produce una petición masiva de recursos con el fin de interrumpir las operaciones de VoIP, por lo general a través de una gran cantidad de mensajes. Esto conduce a la degradación del tiempo de respuesta, la prevención de los suscriptores de la utilización eficaz del servicio.

- *Spoofing Web Services*. También conocido como *Principal Spoofing in Web Services*. Un servicio web spoofing intenta suplantar la identidad de un usuario para el robo de sus credenciales, y luego hace peticiones a su nombre con estas credenciales con la intención de acceder a un servicio web de la víctima.

2.2.3 Otros catálogos

2.2.3.1 Repositorio de patrones Munawar Hafiz

El artículo “Towards an Organization of Security Patterns” (Hafiz et al. 2007) hace referencia a el sitio web www.patternshare.org, que funciona como un repositorio único para describir todos los tipos de patrones de software. Uno de los objetivos de la comunidad Patternshare es desarrollar un vocabulario uniforme para profesionales mediante la combinación de todos los patrones que se diferencian sólo por el nombre. El repositorio de patrones de

seguridad de patternshare resume los patrones de seguridad de diversas fuentes y elimina los solapamientos. El patternshare en su conjunto es gestionado por el Grupo Hillside. En diciembre de 2006, el repositorio contenía 90 patrones seguridad únicos. En la actualidad, la página www.patternshare.org esta fuera de línea, desde 2007.

Pero encontramos en el sitio web de Munawar Hafiz (Munawarhafiz Web 2014), al parecer Munawar es quien continúa manteniendo el repositorio antes patternshare. En esta página web existe un catálogo de patrones de seguridad, que contiene todos los patrones de seguridad escritos por todos los expertos en seguridad a partir de la primera obra 1997. Actualmente, el catálogo contiene 97 patrones.

3 Relación entre los catálogos CSP y CJP

En este capítulo se analiza el catálogo Core Security Patterns (CSP) (Steel et al. 2005) en el contexto de la arquitectura multicapa. Los patrones pertenecientes al Catálogo CSP se han analizado de forma individual y se han interpretado en el contexto del modelo de Core J2EE Catalogue (CJP) (Alur et al. 2003). En los diagramas de clases que se presentan en este capítulo marcamos las relaciones identificadas entre CJP y CSP, destacando estas con líneas de conexión más gruesas. Además, un cambio general que hemos introducido en los diagramas ha sido la sustitución de las asociaciones navegadas UML de los diagramas originales por las dependencias de UML, que son más apropiadas porque no fuerzan la presencia de atributos.

Existen diferentes plantillas como las definidas en (Hillside Group Web 2014) para la descripción de patrones de software. No obstante, como este trabajo no describe ningún catálogo de patrones, sino que relaciona dos catálogos ya definidos hemos diseñado nuestra propia plantilla de descripción:

- **Nombre**: Nombre que describe el patrón, se corresponde con el nombre en CSP.
- **Descripción**: Describe brevemente el objetivo del patrón CSP.
- **Interpretación en la arquitectura multicapa**: Una descripción de la interpretación del patrón SP en términos de la arquitectura multicapa (es decir en el contexto de patrones CJP). Y un diagrama UML que ilustra la relación del patrón con otros patrones.
- **Información añadida a CSP**: Información adicional que aportamos al catálogo CSP en el caso en que se haya producido.
- **Requisitos previos CJP**: Es la información requerida o patrones previos que se deben conocer para entender los patrones de CSP.
- **Requisitos previos CSP**: Es la información requerida o patrones previos CSP, para la comprensión del patrón CSP analizado.

3.1 Capa web

3.1.1 Authentication Enforcer

Descripción: Este patrón ayuda a construir una aplicación con un mecanismo de autenticación centralizado, proporcionando autenticación de usuario en la capa web.

Interpretación en la arquitectura multicapa: Los contenedores J2EE y los Servlets suelen proporcionar algún tipo de pluggin para proveer servicios de autenticación, por ejemplo, el módulo de registro JAAS (Coté JAAS 2009). Estos *Authentications Enforcers* utilizan *Requests* de tipo *Context Objects* CJP que contienen las credenciales del usuario extraídas del mecanismo de solicitud específica del protocolo.

El cliente (*Front Controller* o *Application Controller*) invoca una acción, que debe ser realizada por un usuario autenticado, para lo cual el patrón *Authentication Enforcer* recupera las credenciales de usuario adecuadas, las valida, y si es correcto, crea y retorna un *Subject*. La Figura 3.1 describe esta interpretación.

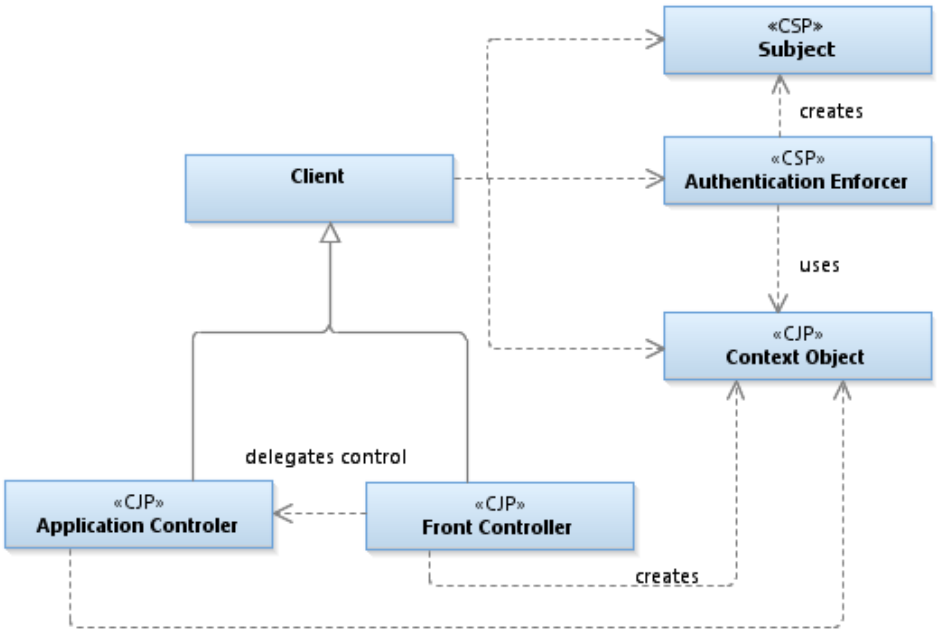


Figura 3.1 Authentication Enforcer interpretado en el contexto multicapa

Información añadida a CSP: Ninguna.

Prerrequisitos CJP: *Front Controller*, *Application Controller*, *Context Object*.

Prerrequisitos CSP: Ninguno.

3.1.2 Authorization Enforcer

Descripción: Este patrón ayuda a construir una aplicación con un mecanismo de autorización centralizado que proporciona autorización a usuarios en la capa web.

Interpretación en la arquitectura multicapa: Los *Front Controller* y *Aplicación Controller* CJP invocan acciones o *Commands*. Estas acciones (las cuales se consideran *Secure Base Action* en CSP) pueden ser invocadas como consecuencia de la autorización prevista por *Authorization Enforcer*. La Figura 3.2 describe esta interpretación.

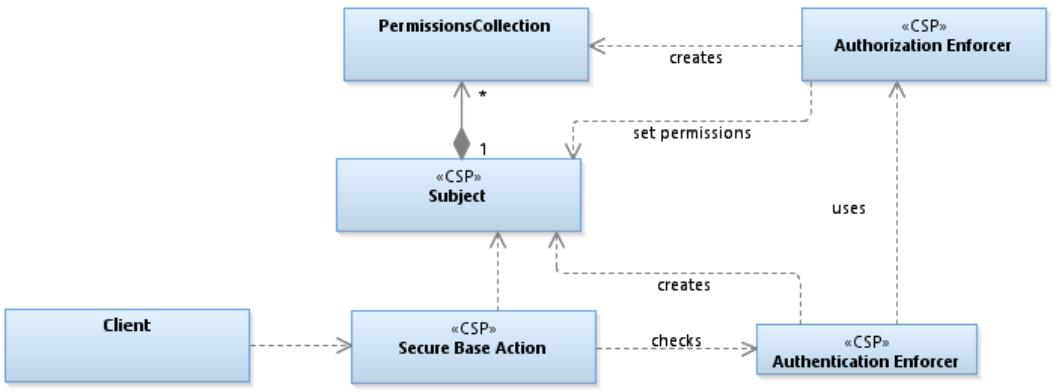


Figura 3.2 Authentication Enforcer interpretado en el contexto multicapa

Información añadida a CSP: Ninguno

Prerrequisitos CJP: *Front Controller* y *Application Controller*.

Prerrequisitos CSP: *Authentication Enforcer*.

3.1.3 Intercepting Validator

Descripción: Este patrón implementa un mecanismo sencillo que permite analizar y validar los datos transmitidos desde el cliente a la capa web de la aplicación. De forma similar al patrón *Intercepting Filter* CJP, las reglas en este patrón pueden ser configurables, pero en este caso el filtrado se enfoca fundamentalmente en validar la petición (validación de falso o verdadero).

Interpretación en la arquitectura multicapa: Hemos definido el patrón *Intercepting Validator*, como una versión especializada del patrón *Intercepting Filter* CJP, encargado de aplicar filtros de validación a los datos entrados por el cliente, para mejorar la seguridad en la capa web. La Figura 3.3 describe esta interpretación.

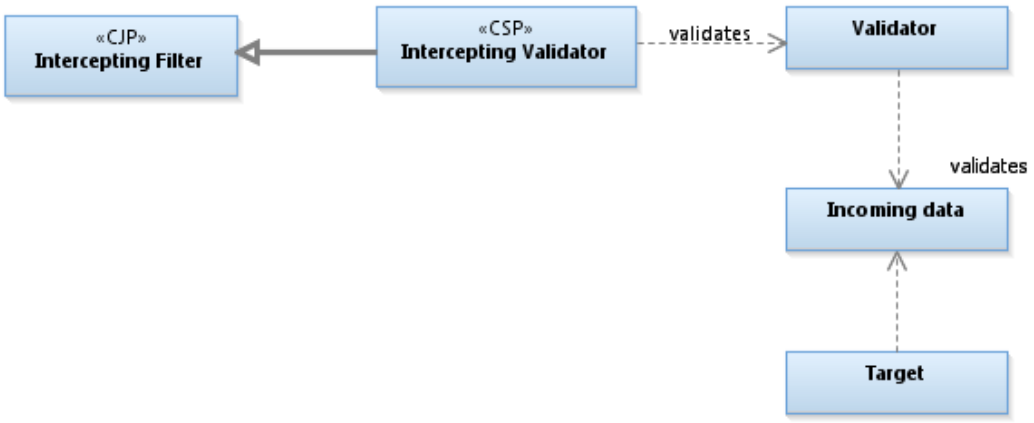


Figura 3.3 Intercepting Validator interpretado en el contexto multicapa

Información añadida a CSP: Este diagrama ha sufrido una modificación con respecto al propuesto por el libro CSP, ya que no creemos necesaria la invocación desde un *SecureBaseAction*, sino que se dibuja en el diagrama como un tipo de *Intercepting Filter* CJP.

Prerrequisitos CJP: *Intercepting Filter*.

Prerrequisitos CSP: *Secure Base Action*.

3.1.4 Secure Base Action

Descripción: Este patrón representa el punto de acceso para la invocación de la lógica de negocio. En este punto se pueden realizar las tareas de seguridad adicionales.

Interpretación en la arquitectura multicapa: Un *Secure Base Action* es un *Command* o acción que incluye tareas de seguridad, invocado por un *Front Controller* o *Application Controller* CJP. La Figura 3.4 describe esta interpretación.

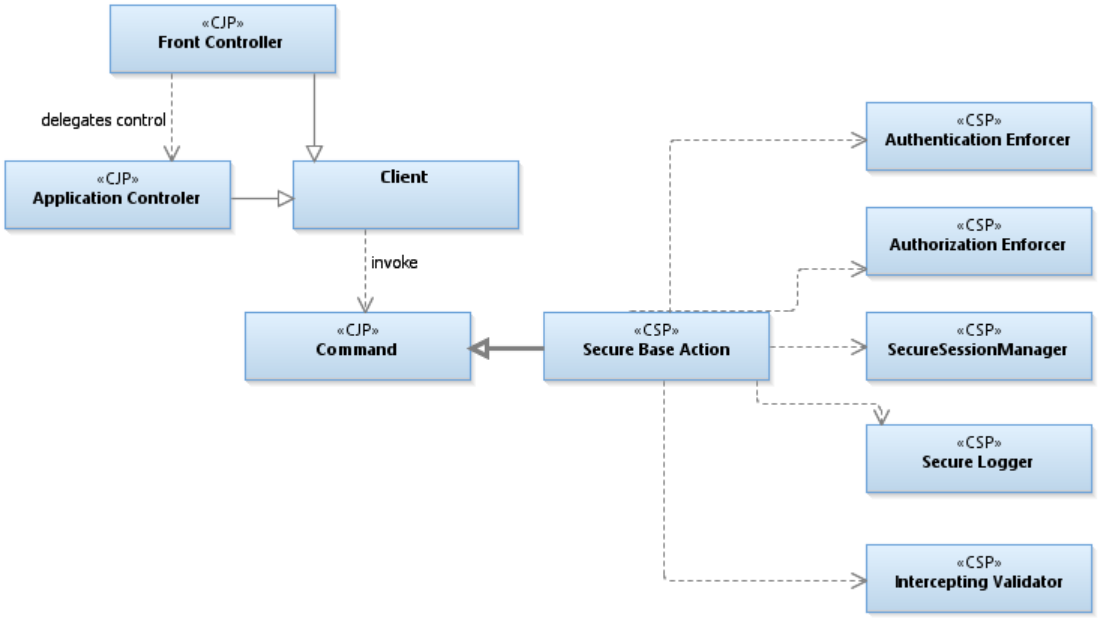


Figura 3.4 Secure Base Action interpretado en el contexto multicapa

Información añadida a CSP: Identificación explícita de un *Secure Base Action* como un Command, y la inclusión de los controladores de CJP.

Requisitos previos CJP: *Front Controller* y *Application Controller*.

Requisitos previos CSP: Ninguno.

3.1.5 Secure Logger

Descripción: Es un patrón gestor de logs, especializado en centralizar y proteger la integridad de datos. Registra eventos u operaciones que requieren de un tratamiento confidencial.

Interpretación en la arquitectura multicapa: El cliente de un *Secure Logger* puede ser, tanto un patrón *Command* genérico, como una *Secure Base Action* CSP. Los componentes de cualquier capa, en particular, la capa de presentación, podrían hacer uso de un *Secure Logger* a efectos de registro de datos confidenciales. La Figura 3.5 describe esta interpretación.

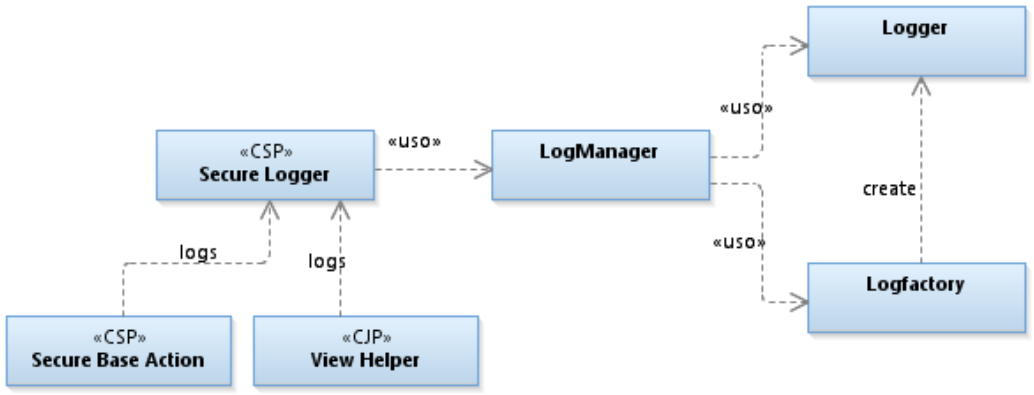


Figura 3.5 Secure Logger interpretado en el contexto multicapa

Información añadida al CSP: Los patrones *View Helper* CJP y *Secure Base Action* son identificados como clientes de un *Secure Logger*.
Requisitos previos CJP: *View Helper*.
Requisitos previos CSP: *Secure Base Action*.

3.1.6 Secure Pipe

Descripción: Este patrón proporciona una manera simple y estandarizada de proteger los datos enviados en una red. Proporciona una conexión segura para evitar el espionaje o cualquier otro tipo de ataque.

Interpretación en la arquitectura multicapa: Este patrón puede ser invocado y creado por cualquier componente de la aplicación que necesite crear un canal de comunicación seguro. *Secure Pipe* está fuera de la aplicación de software en la medida en que una tubería secura define un tipo de conexión encriptada segura para las comunicaciones de red. Una conexión HTTPS es un buen ejemplo de *Secure Pipe*. La Figura 3.6 describe esta interpretación.

Figura 3.6 Secure Pipe interpretado en el contexto multicapa

Información añadida a CSP: Interpretación de *Secure Pipe* como un elemento externo a la aplicación de software, convirtiéndose así en una cuestión de despliegue.
Requisitos previos CJP: Ninguno.
Requisitos previos CSP: Ninguno.

3.1.7 Secure Service Proxy

Descripción: Este patrón permite agregar seguridad a aplicaciones heredadas o antiguas sin tener que modificar el código fuente. Permite proteger los puntos finales de servicios Web de intrusos aprovechando productos de seguridad de terceros o desarrollos de infraestructura de seguridad J2EE creados para tal fin. Por tanto, proporciona validaciones adicionales de seguridad para los servicios pre-existentes.

Interpretación en la arquitectura multicapa: En el catálogo CJP los servicios están expuestos a clientes externos utilizando *Service Activators* CJP y *Web Service Brokers* CJP. Por tanto, el *Secure Service Proxy* es una especialización de este patrón. Vale la pena señalar que el catálogo CJP identifica *Service Activators* y Web *Service Brokers* como elementos que pertenecen a la capa de integración por lo que el *Secure Service Proxy* tal vez debería trasladarse a ese nivel. La Figura 3.7 ilustra esta interpretación.

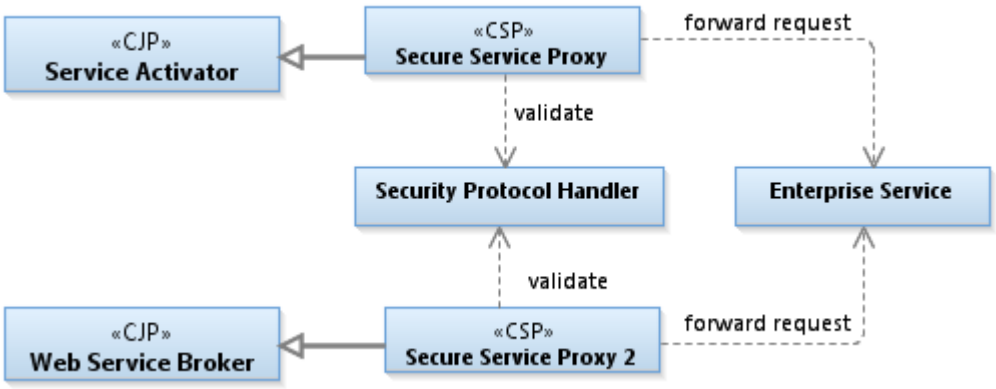


Figura 3.7 Secure Service Proxy interpretado en el contexto multicapa

Información añadida a CSP: Los patrones *Service Activator* CJP y W*eb Service Broker* CJP se muestran como súper tipos del patrón CSP *Secure Service Proxy*, usados dependiendo del tipo de mecanismo de invocación. Este patrón se mueve a la capa de integración.

Requisitos previos CJP: *Service Activator, Web Service Broker*.

Requisitos previos CSP: Ninguno.

3.1.8 Intercepting Web Agent

Descripción: Este patrón es una solución alternativa cuando se desea proporcionar autenticación y autorización en las peticiones a una aplicación que puede estar en las siguientes condiciones:

- La aplicación no tiene seguridad alguna y se prefiere usar un proveedor externo que haga esta tarea.
- La aplicación de construyó anteriormente con una arquitectura de seguridad existente, y se prefiere no modificarla.

El patrón Intercepting Web Agent controla el acceso al contenido de servidores web y servidores proxy.

Interpretación en la arquitectura multicapa: En cierta medida los *Agentes Web* (Oracle 2013) son similares a los filtros CJP, pero, a diferencia de éstos, son dispositivos invocados por los servidores Web y de aplicaciones para fines de seguridad. Por lo tanto, no hay homólogos CJP para agentes web de CSP. La Figura 3.8 describe esta interpretación.

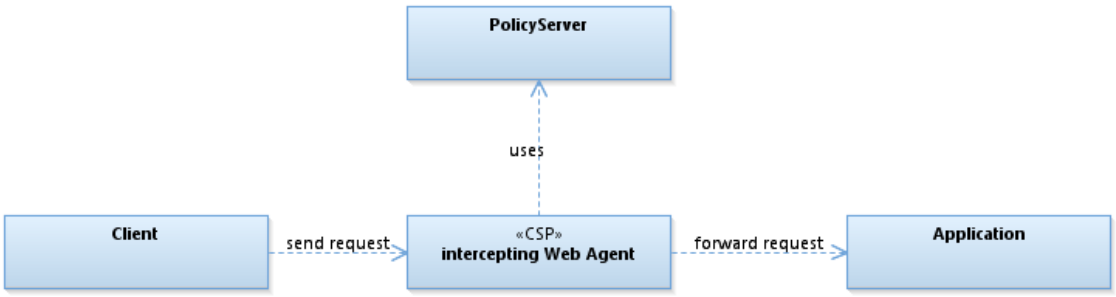


Figura 3.8 Web Agent Interceptor interpretado en el contexto multicapa

Información añadida a CSP: Ninguno.

Requisitos previos CJP: Ninguno.

Requisitos previos CSP: Ninguno.

3.2 Capa de negocio

3.2.1 Audit Interceptor

Descripción: El patrón *Audit Interceptor* trabaja en conjunto con el patrón CSP *Secure Logger*, uno en el back-end y el otro en el front-end. Este patrón registra y audita tanto peticiones como respuestas de forma centralizada a través de un catálogo de eventos definidos previamente.

Interpretación en la arquitectura multicapa: Este patrón es similar a un *Application Service* CJP, con la diferencia que este se especializa en auditar las invocaciones de servicios. La Figura 3.9 describe esta interpretación.

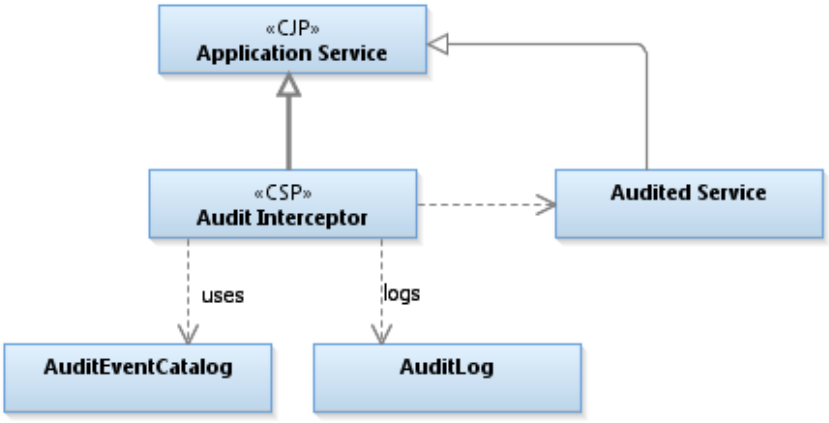


Figura 3.9 Audit Interceptor interpretado en el contexto multicapa

Información añadida a CSP: El *Audit Interceptor* se identifica como un *Application Service* CJP dedicado a cuestiones de seguridad. El *Audit Interceptor* podría ser un tipo de proxy que extiende el servicio auditado para que pudiera ser utilizado como este servicio a pesar de que los clientes no son conscientes de esto.
Requisitos previos CJP: *Application Service*.
Requisitos previos CSP: Ninguno.

3.2.2 Obfuscated Transfer Object

Descripción: Este patrón ofusca y cifra la información de los *Transfer Object*.

Interpretación en la arquitectura multicapa: los objetos *Transfer Object* de CJP encapsulan y mueven datos a través de las diferentes capas. El objeto de transferencia ofuscado esconde y encripta los datos. La Figura 3.10 describe esta interpretación.

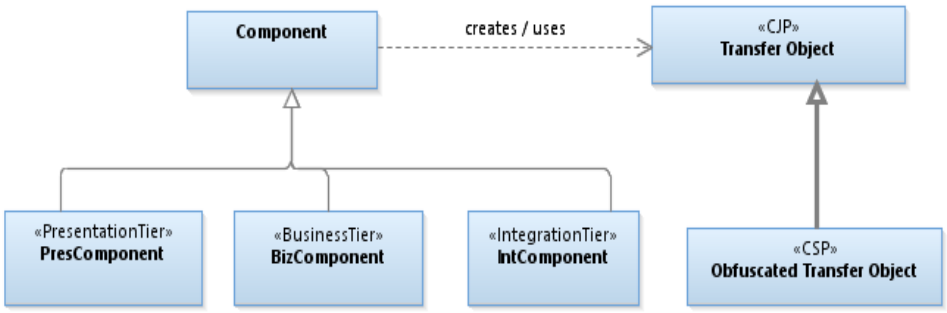


Figura 3.10 Obfuscated Transfer Object interpretado en el contexto multicapa

Información añadida a CSP: *Obfuscated Transfer Object* CSP es un tipo especial de *Transfer Object* CJP.
Requisitos previos CJP: *Transfer Object*.
Requisitos previos CSP: Ninguno.

3.2.3 Policy Delegate

Descripción: Este patrón crea, gestiona y administra las políticas de gestión de seguridad que rigen el acceso y direccionamiento desde y hacia objetos EJB. Ubica servicios seguros para clientes remotos.

Interpretación en la arquitectura multicapa: Un *Policy Delegate* es un *Business Delegate* CJP que localiza e invoca los servicios remotos de forma segura. La Figura 3.11 ilustra esta interpretación.

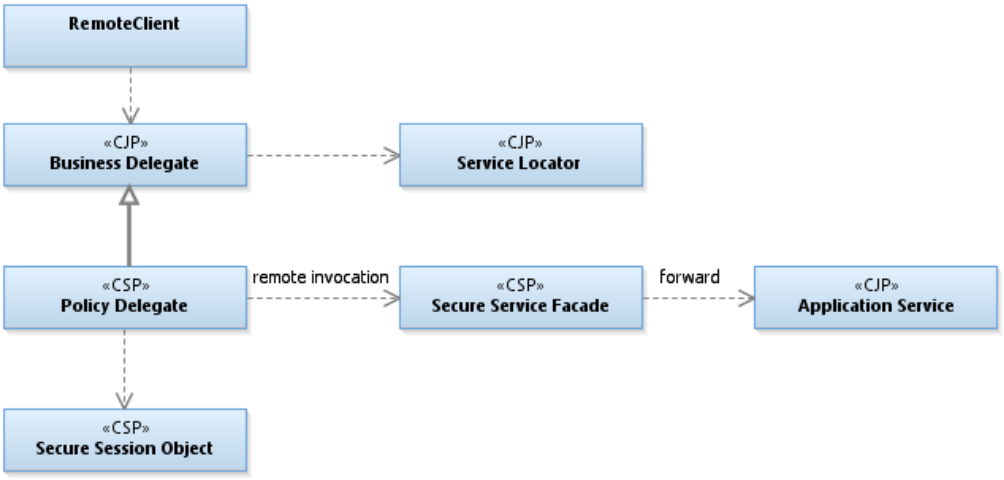


Figura 3.11 Policy Delegate interpretado en el contexto multicapa

Información añadida a CSP: El *Policy Delegate* CSP se identifica como un *Business Delegate* CJP.

Requisitos previos CJP: *Business Delegate*.

Requisitos previos CSP: *Secure Session Object*.

3.2.4 Secure Service Facade

Descripción: Este patrón ofrece una fachada de sesión que puede contener y centralizar las interacciones complejas entre los componentes de negocio bajo una sesión segura. Proporciona seguridad dinámica y declarativa en el back-end de los objetos de negocio en la fachada del servicio. Protege de invocaciones de entidades externas y llamadas de servicios ilegales o no autorizados.

Interpretación en la arquitectura multicapa: Un CSP *Secure Service Facade* es una CJP *Session Facade* que realiza tareas de seguridad adicionales usando un objeto de sesión seguro. La Figura 3.12 describe esta interpretación.

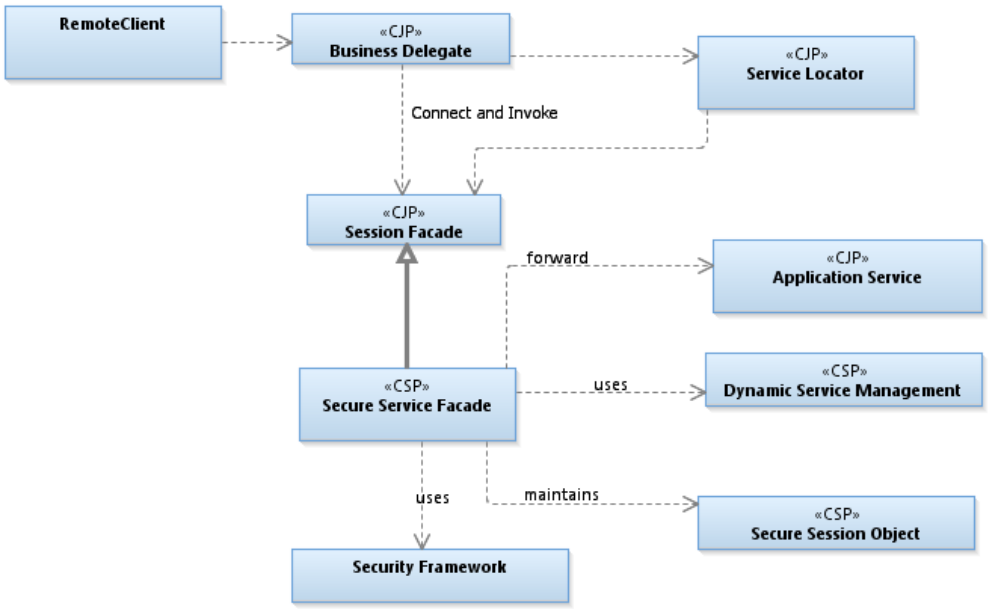


Figura 3.12 Secure Service Facade interpretado en el contexto multicapa

Información añadida a CSP: Ninguno.

Requisitos previos CJP: *Session Facade*, *Business Delegate*, *Application Service*.

Requisitos previos CSP: *Secure Session Object*.

3.2.5 Secure Session Object

Descripción: Este patrón define formas de proteger la información de sesión de EJB que facilitan un acceso distribuido y evitar la propagación de fisuras del contexto de seguridad. Este modelo permite encapsular de manera abstracta las credenciales de autorización y autenticación de un contexto para traspasar fronteras en entornos distribuidos. La Figura 3.13 describe esta interpretación.

Interpretación en la arquitectura multicapa: El patrón *Secure Session Object* es explícitamente identificado como un *Transfer Object* CJP.

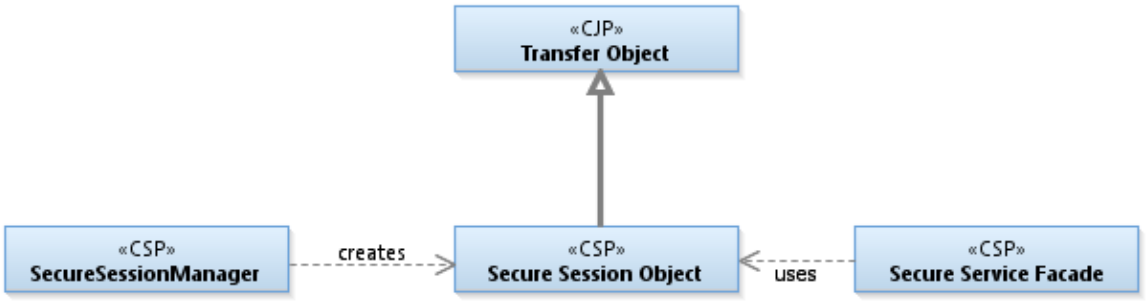


Figura 3.13 Secure Session Object interpretado en el contexto multicapa

Información añadida a CSP: El *Secure Session Object* CSP es considerado un *Transfer Object* CJP.

Requisitos previos CJP: *Transfer Object*.

Requisitos previos CSP: *Secure Service Façade*, *Secure Session Manager*.

3.2.6 Container Managed Security

Descripción: Este patrón describe cómo declarar la información relacionada con la seguridad en un descriptor de despliegue. Es una manera simple y estándar de aplicar la autenticación y autorización de usuarios a un nivel de granularidad deseado en las aplicaciones J2EE.

Interpretación en la arquitectura multicapa: Esta característica reside en un contenedor externo a la aplicación de software y asegura el acceso autenticado a los *Application Services* CSP expuestos como EJBs. La Figura 3.14 ilustra esta interpretación.

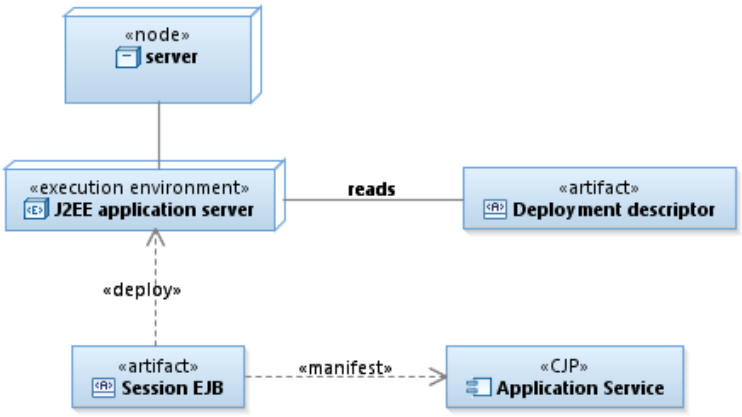


Figura 3.14 Container Managed Security interpretado en el contexto multicapa

Información añadida a CSP: *Container Managed Security* se identifica como un patrón externo a la aplicación de software. Además, se identifican explícitamente en el diagrama la relación con el patrón *Application Service* CJP.

Requisitos previos CJP: *Application Service*.

Requisitos previos CSP: Ninguno.

3.2.7 Dynamic Service Management

Descripción: Este patrón lo podemos resumir como un objeto Java Management Extension (JMX) destinado a fines de vigilancia.

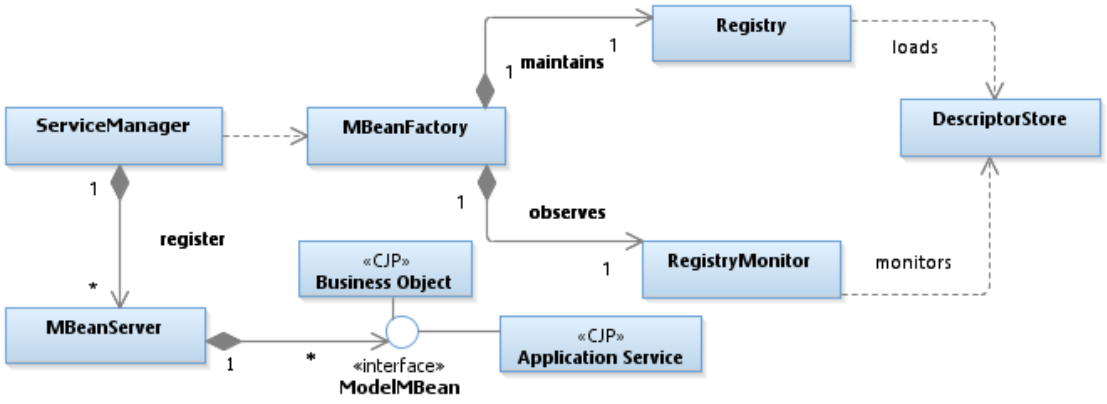


Figura 3.15 Dynamic Service Management interpretado en el contexto multicapa

Interpretación en la arquitectura multicapa: El *Application Service* CJP y *Business Object* CJP, descritos en el diagrama, pueden ser considerados como beans manejados JMX con fines de instrumentación. La Figura 3.15 describe esta interpretación.

Información añadida a CSP: *Application Service* CJP y *Business Object* CJP se identifican explícitamente como objetos para ser monitoreados por el patrón *Dynamic Service Managemen*t.
Requisitos previos CJP: *Application Service*, *Business Object*.
Requisitos previos CSP: Ninguno.

3.3 Capa de servicios web

3.3.1 Message Inspector

Descripción: Este patrón comprueba y verifica la calidad y mecanismos de seguridad a nivel de mensaje XML, tales como XML Signature y XML Encryption, trabajando en conjunto con un *Security Token*. El patrón *Message Interceptor* también ayuda en la verificación y validación de los mecanismos de seguridad aplicadas en mensajes SOAP procesados por múltiples intermediarios (actores). Es compatible con una variedad de formatos de firma y tecnologías de cifrado utilizados por estos intermediarios.

Interpretación en la arquitectura multicapa: Antes de la invocación de servicio web el *Message Interceptor Gateway* CSP invoca al *Message Inspector* CSP para la comprobación de la seguridad de los mensajes XML. Si se aprueba, se da acceso al *Web Service Broker* CJP. La Figura 3.16 describe esta interpretación.

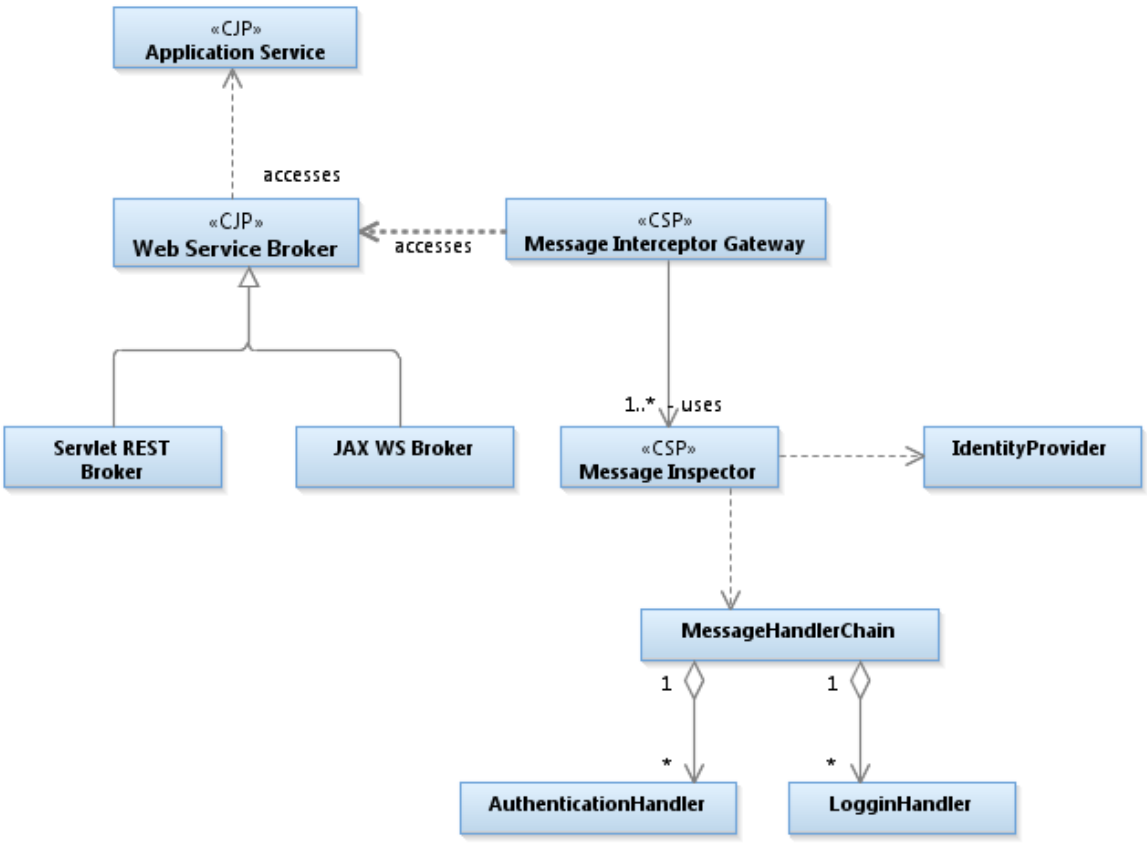


Figura 3.16 Message Inspector interpretado en el contexto multicapa

Información añadida a CSP: El punto final del servicio web es identificado como un *Web Service Broker* CJP.

Requisitos previos CJP: *Web Service Broker*, *Application Service*.

Requisitos previos CSP: *Message Interceptor Gateway*.

3.3.2 Message Interceptor Gateway

Descripción: Este patrón proporciona un único punto de entrada y permite la centralización de la seguridad de los mensajes entrantes y salientes de la aplicación. Las funciones de seguridad incluyen la creación, modificación y gestión de políticas de seguridad para el envío y recepción de mensajes. Permite aplicar los mecanismos de

seguridad a nivel de mensaje y a nivel de transporte necesarios para comunicarse con seguridad con los puntos finales de los servicios web.

Interpretación en la arquitectura multicapa: Antes de la invocación de servicio web, se invoca *Message Interceptor Gateway* (por un firewall XML o por el contenedor J2EE) para la comprobación de la seguridad de grano grueso (por ejemplo, certificado X.509) y para la seguridad de grano fino se comprueba mediante un *Message Inspecto*r CSP. Tras la verificación, se invoca el *Web Service Broker* CJP. La Figura 3.17 describe esta interpretación.

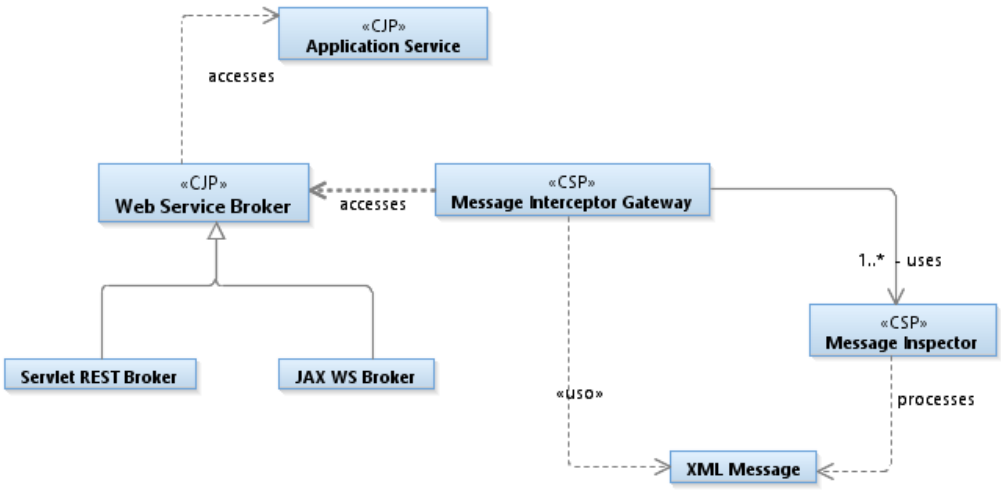


Figura 3.17 Message Interceptor Gateway interpretado en el contexto multicapa

Información añadida a CSP: El punto final del servicio se identifica explícitamente como un *Web Service Broker* CJP.
Requisitos previos CJP: *Web Service Broker*.
Requisitos previos CSP: Ninguno.

3.3.3 Secure Message Router

Descripción: Este patrón facilita la comunicación segura XML con varios puntos finales asociados que adoptan la seguridad a nivel de mensaje y mecanismos de federación de identidad. Actúa como un elemento intermedio de seguridad que aplica mecanismos de seguridad a nivel de mensaje para enviar mensajes a varios destinatarios de tal forma que el receptor pueda tener acceso a sólo una parte necesaria del mensaje y los fragmentos restantes de los mensajes sean confidenciales.

Interpretación en la arquitectura multicapa: Hay ocasiones en que la invocación de servicios y la orquestación tiene que cumplir con las restricciones de seguridad (por ejemplo, autenticación y autorización). El patrón *Secure Message Router* es un tipo de *Web Service Broker* CJP que cumple con todas las exigencias de seguridad de los servicios invocados. La Figura 3.18 describe esta interpretación.

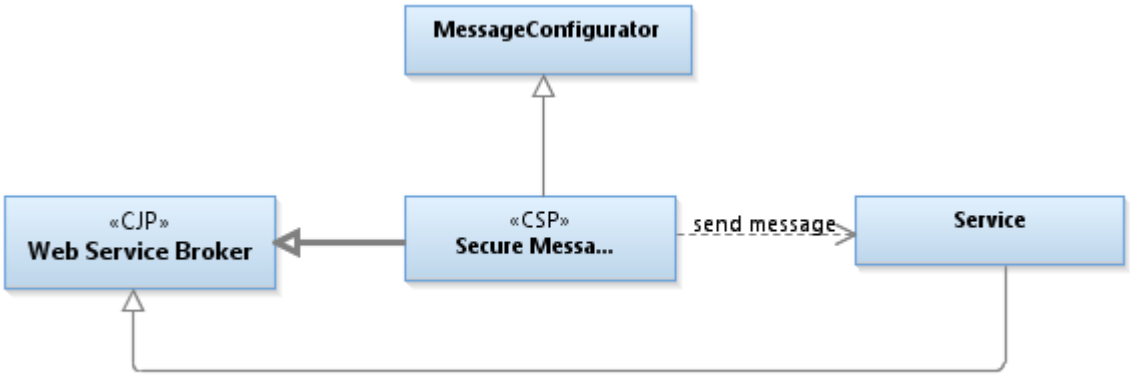


Figura 3.18 Secure Message Router Interpretado en el contexto multicapa

Información añadida al CSP: El *Secure Message Router* es un *Web Service Broker* CJP.
Requisitos previos CJP: *Web Service Broker*
Requisitos previos CSP: Ninguno.

3.4 Capa de identidad

3.4.1 Assertion Builder

Descripción: Este patrón define la forma en que se puede construir una aserción de identidad. Encapsula la lógica de control de procesamiento con el fin de crear las instrucciones SAML de autenticación, estados de decisión de autorización y declaración de atributos como un servicio.

Interpretación en la arquitectura multicapa: En el contexto de un escenario Single Sign-On las aserciones SAML tienen que ser compartidas entre los diferentes sistemas. La Figura 3.19 describe esta interpretación.

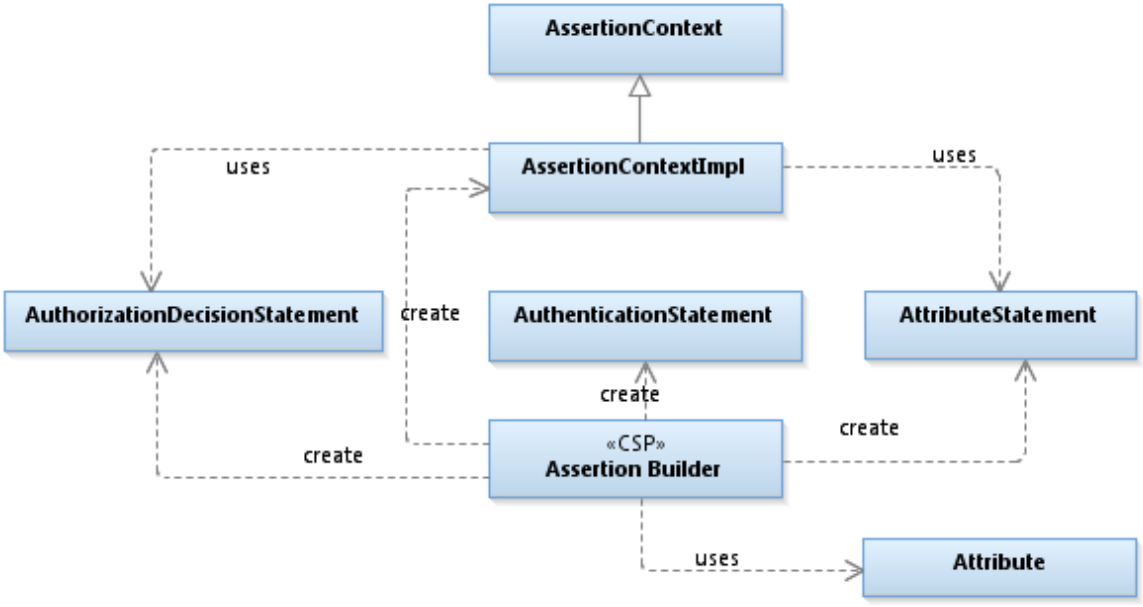


Figura 3.19 Assertion Builder interpretado en el contexto multicapa

Información añadida al CSP: Ninguno.

Requisitos previos CJP: Ninguno.

Requisitos previos CSP: Ninguno.

3.4.2 Credential Tokenizer

Descripción: Hay diferentes formas de credenciales de usuario (también conocida como *Security Token*), tales como nombre de username/passwords, *Binary Security Tokens* (por ejemplo, certificados X.509v3), tickets Kerberos, tokens SAML, token de tarjetas inteligentes y muestras biométricas. El patrón *Credential Tokenizer* describe cómo puede ser encapsulados, incrustados en un mensaje, direccionados, y procesados los principales security tokens. Este patrón tiene sentido en una infraestructura flexible que puede ser reutilizada por diferentes proveedores.

Interpretación en la arquitectura multicapa: En el contexto de un escenario *Single Sign-On*, los diferentes tipos de credenciales de un usuario se puede encapsular como tokens de seguridad que sean reutilizables a través de los diferentes proveedores de seguridad. La Figura 3.20 describe esta interpretación.

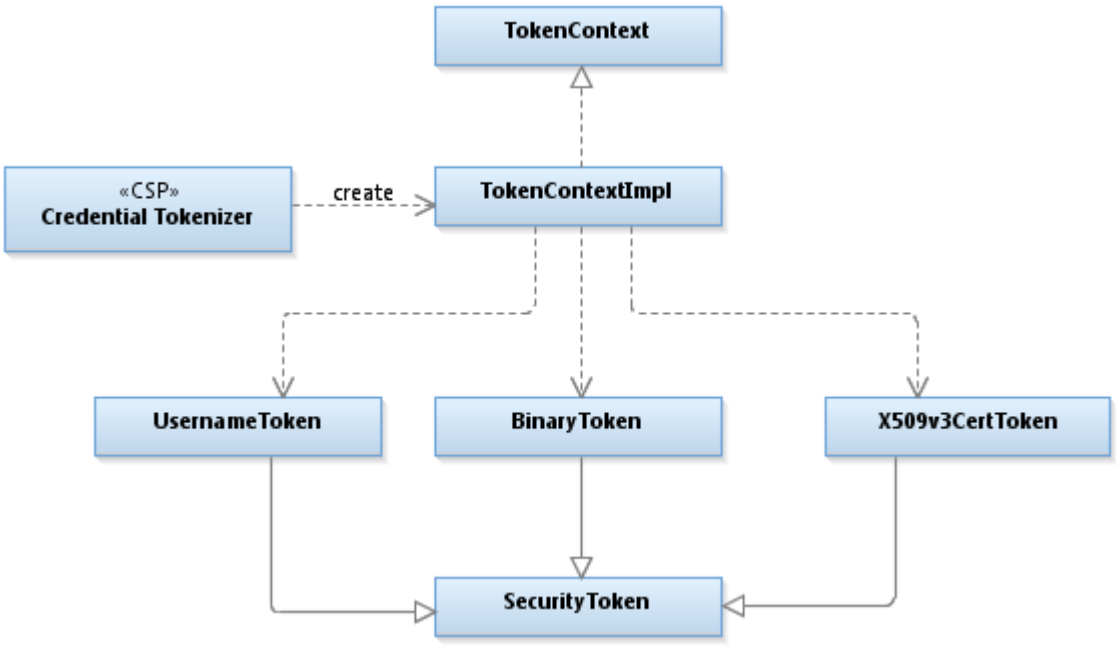


Figura 3.20 Credential Tokenizer interpretado en el contexto multicapa

Información añadida al CSP: Ninguno.

Requisitos previos CJP: Ninguno.

Requisitos previos CSP: Ninguno.

3.4.3 SSO Delegator

Descripción: El patrón describe cómo construir un agente delegado para el manejo de inicio de sesión único (SSO) en un sistema heredado. *Single Sign-on Delegator* reside en el nivel intermedio entre los clientes y la gestión de identidad de componentes de servicio. Oculta los detalles de la invocación de servicios, la recuperación de la configuración de seguridad, o procesamiento del *Credential Tokenizer* del cliente.

Interpretación en la arquitectura multicapa: A veces, un sistema de software está compuesto por diferentes aplicaciones, y se desea tener una sesión única. Un *SSO Delegator* proporciona tales facilidades. *SSO Delegator* proporciona un servicio específico de seguridad: *Sing-On*. Por lo tanto, se ha considerado como un caso especial de un *Application Service* CJP. La Figura 3.21 describe esta interpretación.

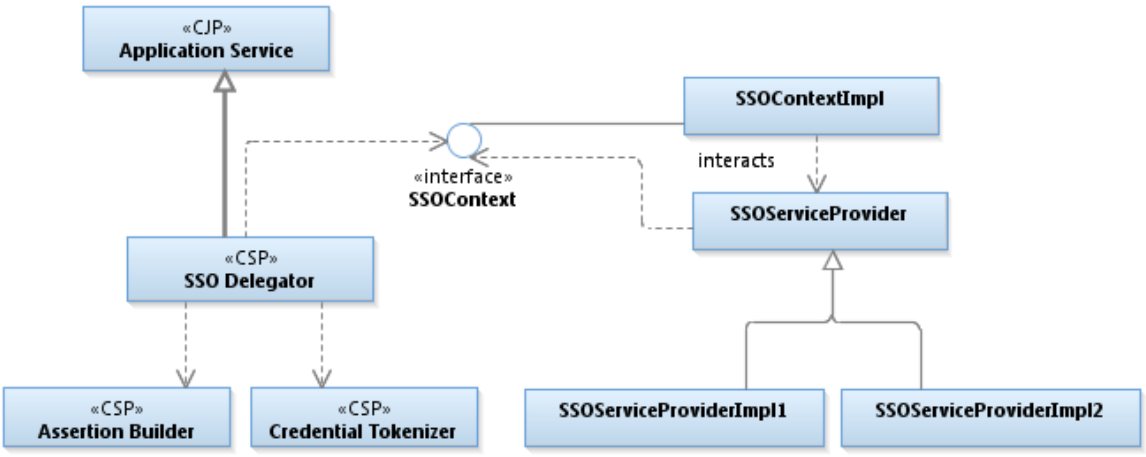


Figura 3.21 SSO Delegator interpretado en el contexto multicapa

Información añadida al CSP: El patrón *SSO Delegator* se identifica como un patrón *Application Service* CJP y se relaciona explícitamente con el *Credential Tokenizer* CSP.

En este patrón hemos eliminado la información incorrecta acerca de la creación de un solo inicio de sesión utilizando fábricas abstractas.

Requisitos previos CJP: *Application Service*.

Requisitos previos CSP: *Assertion Builder* y *Credential Tokenizer*.

3.4.4 Password Synchronizer

Descripción: Este patrón describe una interfaz de programación segura para centralizar la sincronización de las credenciales de usuarios a través de múltiples aplicaciones en las cuales está dado de alta el usuario. A pesar de no estar incluido en esta capa dentro del catálogo CSP, lo hemos incluido aquí por diversos motivos: (i) el catálogo CSP no lo define dentro de ninguna capa concreta; (ii) nos parecía poco razonable definir una capa con un solo patrón; y (iii) está vinculado en cierta forma con la gestión de la identidad de los usuarios.

Interpretación en la arquitectura multicapa: Este patrón proporciona un mecanismo para homogenizar contraseñas a través de distintas aplicaciones. Proporciona un servicio de seguridad específico: sincronizar contraseñas. Por tanto, lo hemos considerado un caso especial de *Application Service* CJP. La Figura 3.22 describe esta interpretación.

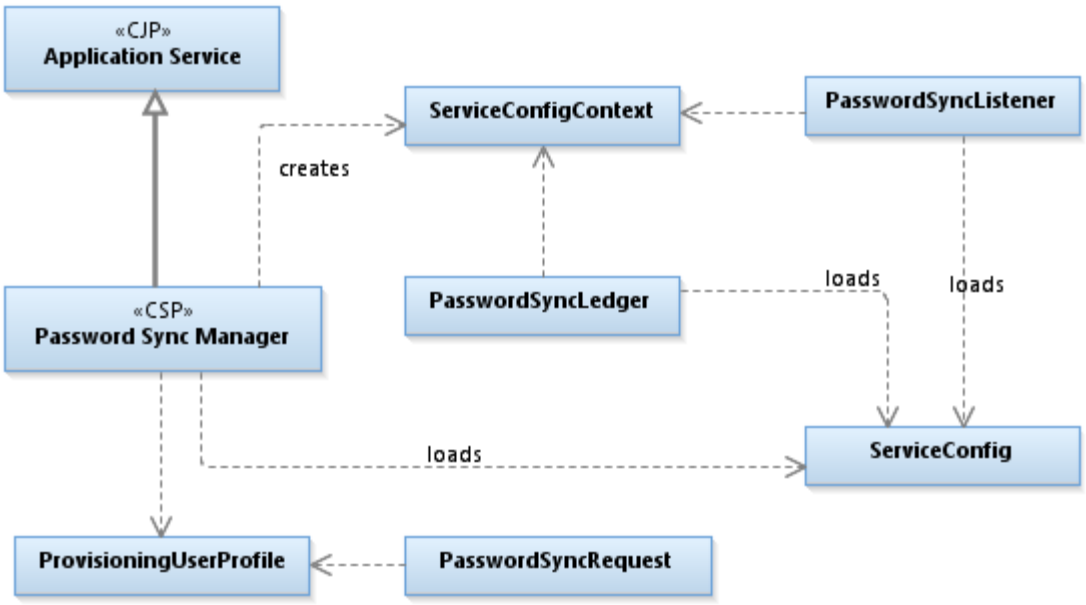


Figura 3.22 Password Synchronizer interpretado en el contexto multicapa

Información añadida al CSP: Ninguno.

Requisitos previos CJP: *Application Service*.

Requisitos previos CSP: Ninguno.

3.5 Resumen de las relaciones entre los catálogos CJP y CSP

Uno de nuestros objetivos principales desde el inicio de este trabajo era conseguir un diagrama UML como el que se muestra en la Figura 3.23 .Dicha figura sintetiza en un diagrama de clases las relaciones entre los dos catálogos CSP Y CJP que hemos analizado e identificado para cada patrón en esta sección.

Los patrones de seguridad CSP están diseñados para la arquitectura de varios niveles y están, por tanto, estrechamente relacionado con los patrones de CJP (de hecho, los patrones de CSP (Steel et al. 2005) mencionan los patrones (CJP) (Alur et al. 2003) en su descripción). Por lo tanto, en teoría, debería ser fácil de integrar los patrones de CSP en aplicaciones arquitectura multicapa. Sin embargo, esto no es cierto en la práctica. El catálogo de patrones CSP no vincula adecuadamente los patrones CSP y los patrones de CJP, lo que hace que su comprensión

unificada sea difícil. Por esto, con este objetivo cumplido en este trabajo deseamos cerrar la brecha entre ambos catálogos, estableciendo explícitamente la relación entre ellos, como se muestra en la Figura 3.23.

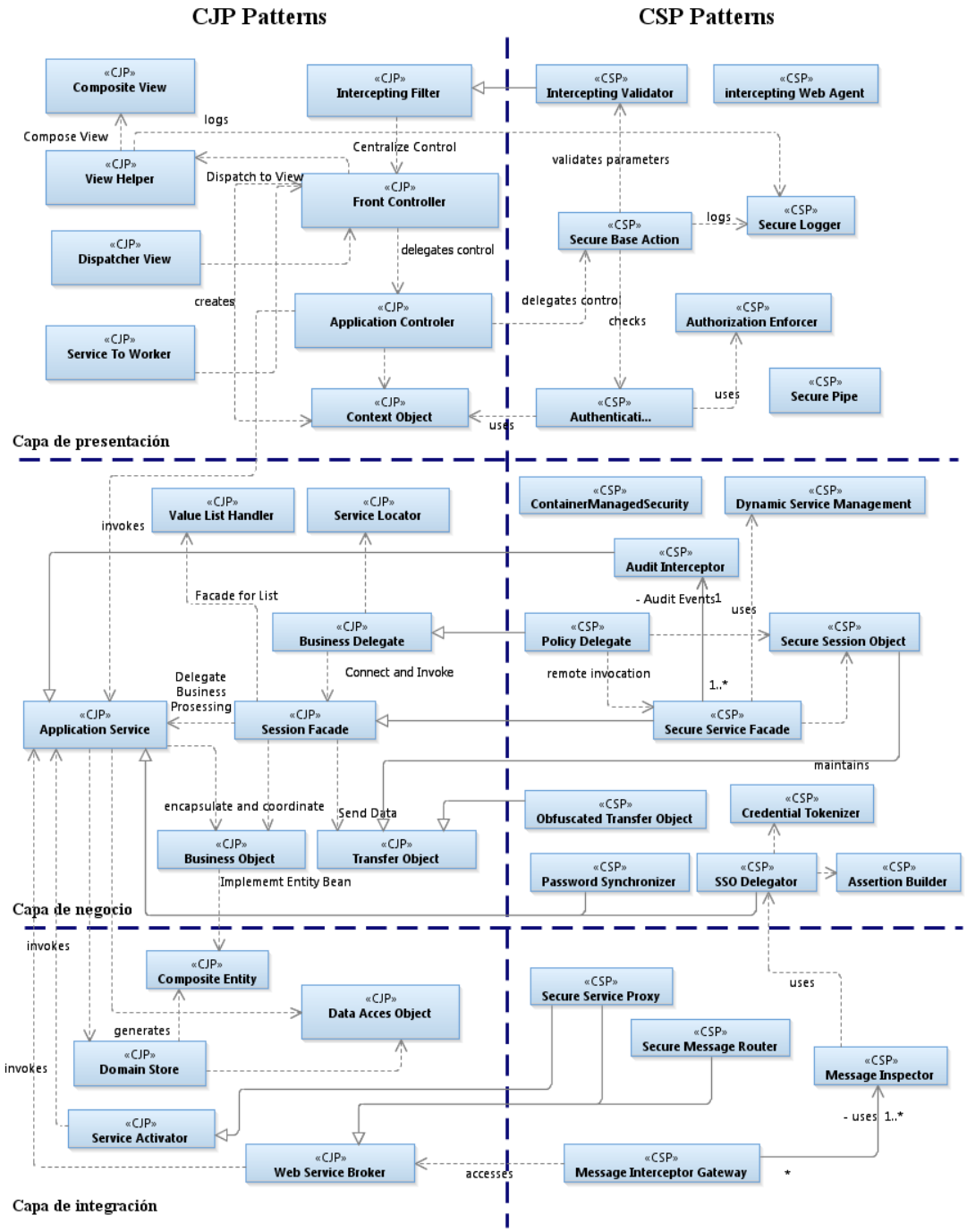


Figura 3.23 Relación entre los catálogos de patrones CJP y CSP

Como hemos visto en el capítulo 3, el catálogo CSP está clasificado por niveles que no coinciden con los niveles del catálogo CSP. La capa La capa web CSP puede asimilarse a la capa de presentación CJP (aunque *Secure Service Proxy* debe ser incluido en la capa de integración) y los niveles de negocio en ambos catálogos podríamos decir que son los que mejor se corresponden. Los patrones de nivel de servicios web del CSP podrían ubicarse en el nivel de integración CJP debido a que el CJP *Web Service Broker* de se encuentra en este nivel y los patrones CSP de este nivel están estrechamente relacionados con este patrón. Los patrones de gestión de identidad del catálogo CSP pueden ser ubicados en la capa de negocio, porque los patrones Single *Sign-On Delegator* y *Password Synchronizer* pueden concebirse como servicios de aplicación CJP que se han dedicado a cuestiones de seguridad.

4 Integración de los patrones CJP y CSP en un caso práctico

Con el objetivo de unir los catálogos revisados en este trabajo, y materializar el análisis realizado sobre los patrones, este capítulo incluye diseños en UML que describen cómo integrar los patrones de ambos catálogos en un caso práctico.

La sección 4.1, describe un sencillo caso de uso que permite poner en práctica los principales patrones arquitectónicos del catálogo CJP.

La Sección 4.2, incluye los patrones del catálogo CSP en el diseño previo que sólo incluía patrones CJP. El objetivo es llevar a la práctica la integración de ambos catálogos de patrones. Esta integración considera dos tipos de aplicaciones, una aplicación multicapa sin servicios web, y otra con servicios web.

4.1 Aplicación original con patrones CJP

Vamos a implementar el diseño de un caso de uso de la aplicación "Nominas", esta funcionalidad "sumar nominas" recibe dos identificadores que se corresponde al id de dos empleados, la lógica de negocio comprueba su existencia, accede a los datos en la capa de recursos, suma el valor de las nóminas de cada empleado y muestra el resultado. Como se puede ver, es un caso de uso un poco peculiar, pero que permite utilizar los principales patrones de arquitectura multicapa (Navarro et al. 2012):

- *Capa de presentación*: Front Controller.
- *Capa de Negocio*: Application Service, Transfer Object.
- *Capa de integración*: Data Access Object.

4.1.1 Diagrama de caso de uso

Este diagrama UML nos permite conocer la funcionalidad de la aplicación "Nominas" mostrando como los usuarios interactúan con ella. Nuestra aplicación de ejemplo tiene el siguiente caso de uso:

- *Sumar nóminas*: Funcionalidad que permite a un usuario de la aplicación sumar la nómina de dos empleados, proporcionando como entrada los identificadores de los empleados.

Figura 4.1 Diagrama de caso de uso

4.1.2 Diagrama de actividad

El diagrama UML de la Figura 4.2 describe paso a paso cada una de las acciones realizadas y el flujo de trabajo de la aplicación, desde que un usuario ingresa los identificadores de los empleados, hasta conseguir el resultado final, que es mostrar al usuario el resultado de la suma de las dos nóminas.

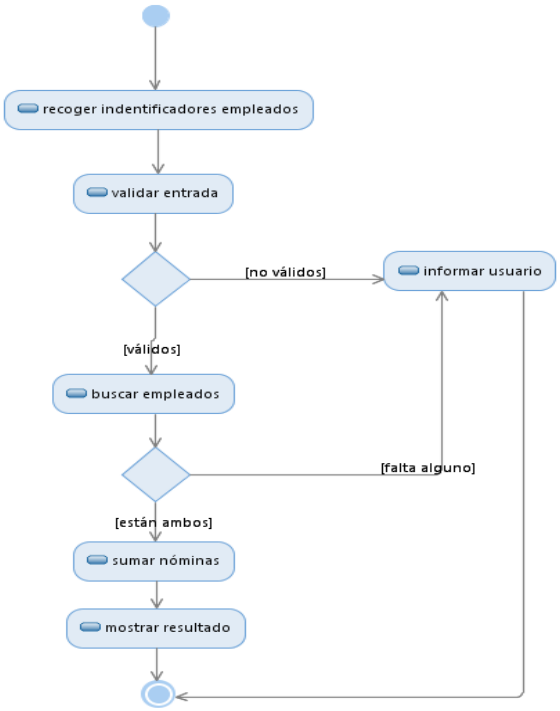


Figura 4.2 Diagrama de Actividad sumar nóminas

4.1.3 Diagramas de clases

El modelado de clases es un tipo especializado para crear la estructura general de un sistema, se usa para implementar los requisitos del sistema y determinar como un sistema satisface a sus necesidades, centrándose en los elementos que lo componen y sus relaciones (Si Alhir 2003). La aplicación que vamos diseñar se basa una arquitectura multicapa, tal y como describe el catálogo CJP.

En un modelado orientado a objetos las clases, objetos y sus relaciones son los principales elementos del diseño, por eso a continuación describimos cada una de las capas de esta arquitectura con diagramas de clases que describen los componentes y relaciones entre ellos.

De los veintiún patrones descritos en el catálogo CJP vamos a utilizar aquellos patrones básicos que determinan una verdadera arquitectura multicapa (Navarro et al. 2012).

4.1.3.1 Capa de presentación

La Figura 4.3 muestra un diagrama de clases UML de la capa de presentación, y a continuación detallamos los patrones CJP usados utilizados en esta capa

- *Front Controller:* La clase *ControladorImp* extiende a la clase *Controlador*, encargada de recibir y gestionar las peticiones desde y hacia la capa de presentación. Tal como describe el patrón *FronController* en CJP, esta clase centraliza la lógica que es común a todas las llamadas entrantes (*GUINominas*) y/o salientes (*GUIResultadoImpl*) y así evitamos la duplicación del código. Además como vemos en el diagrama esta clase es el *ControladorImpl* el que se encarga de gestionar las peticiones hacia la capa de negocio, haciendo la llamada a *SANominas*.

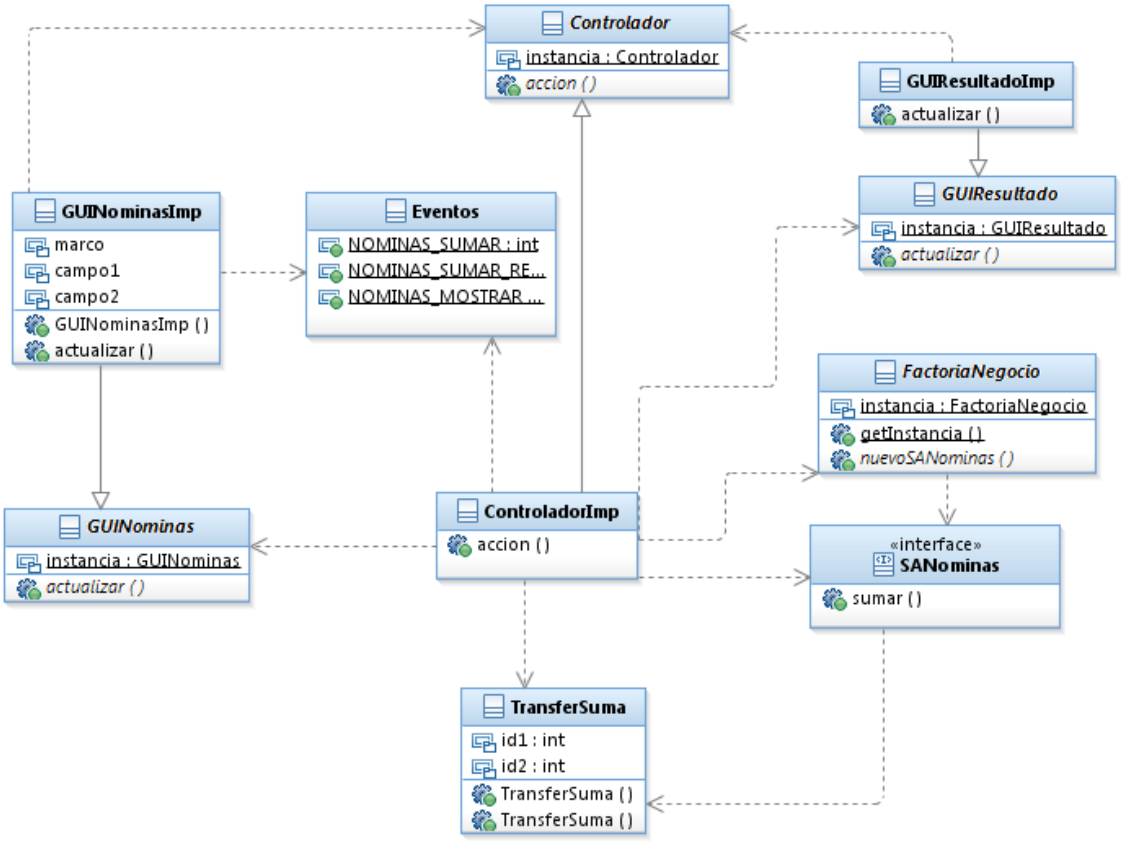


Figura 4.3 Diagrama de clases capa de presentación

4.1.3.2 Capa de negocio

En el diagrama de clases de la capa de negocio representado en la Figura 4.4, podemos ver el siguiente patrón CJP perteneciente a la capa de negocio:

- *Application Service:* Para centralizar la lógica de negocio entre los componentes y servicios de la capa de negocio, y agregar un entorno uniforme de acceso de grano grueso a la capa de negocio, implementamos este patrón con la interface *SANominas* y su implementación *SANominasImp*. Esta interface se encarga de centralizar las acciones de la aplicación (sumar) que llegan de la capa de presentación, hace las invocaciones a los objetos de la capa de integración como es la *FactoriaIntegracion*, *DAOEmpleados*, *TransferEmpleado* y retorna a la capa de presentación el objeto *TrasnferSuma*.

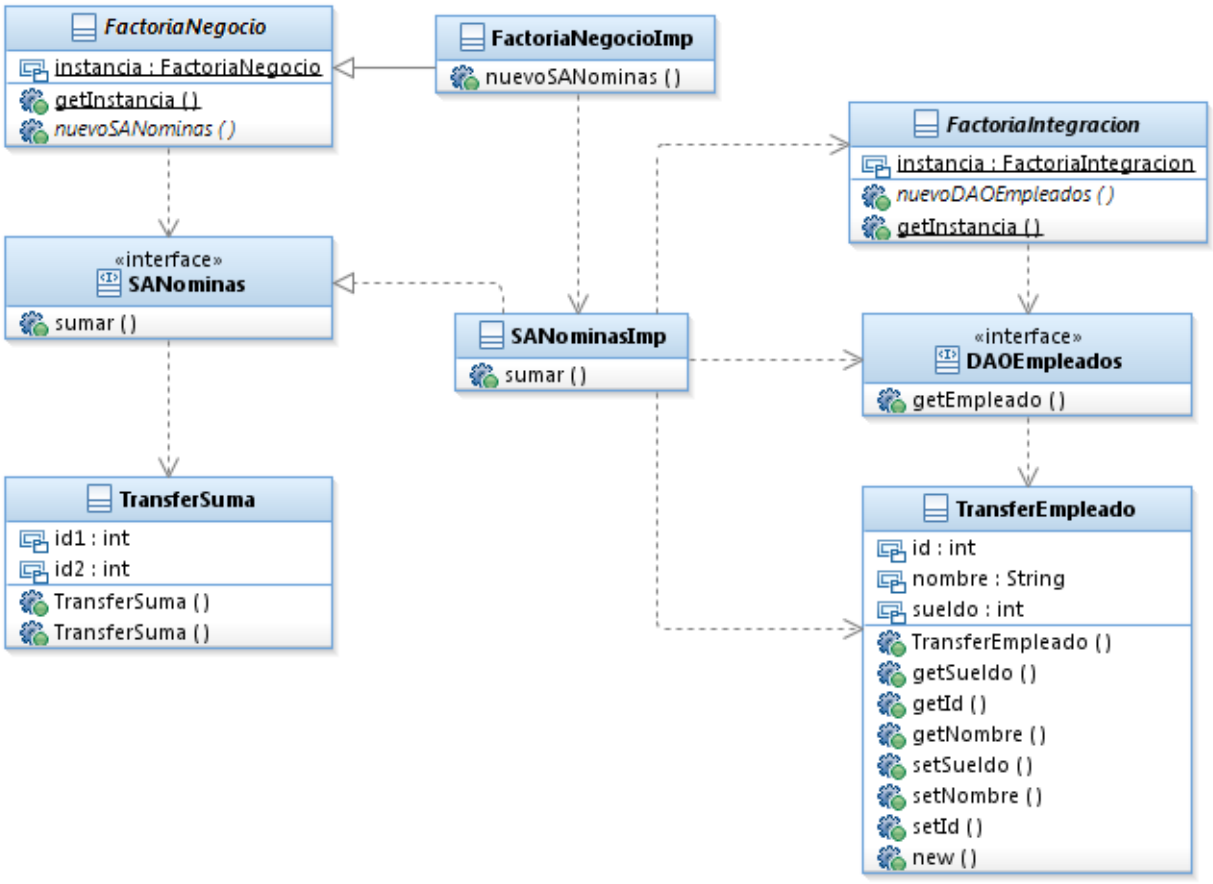


Figura 4.4 Diagrama de clases capa de negocio

- *Transfer Object*: La clase *TransferEmpleado*, y *TransferSuma* son implementaciones de este patrón. Estas clases son utilizadas para transferir todos los elementos de una llamada en un solo objeto:
  - *TransferEmpleado* transfiere datos entre la capa de Integración y la capa de negocio. En este ejemplo, en la capa de negocio se necesita la información de la nómina de los dos empleados para realizar la función suma, por lo que se invoca al *DAOEmpleados*, y este nos devuelve un *TransferEmpleado* con la información requerida.
  - *TransferSuma* se utiliza para transferir datos entre la capa de negocio y la capa de presentación.

4.1.3.3 Capa de integración

En el diagrama de clases UML de la capa de integración representado en la Figura 4.5, podemos ver el siguiente patrón CJP perteneciente la capa de Negocio:

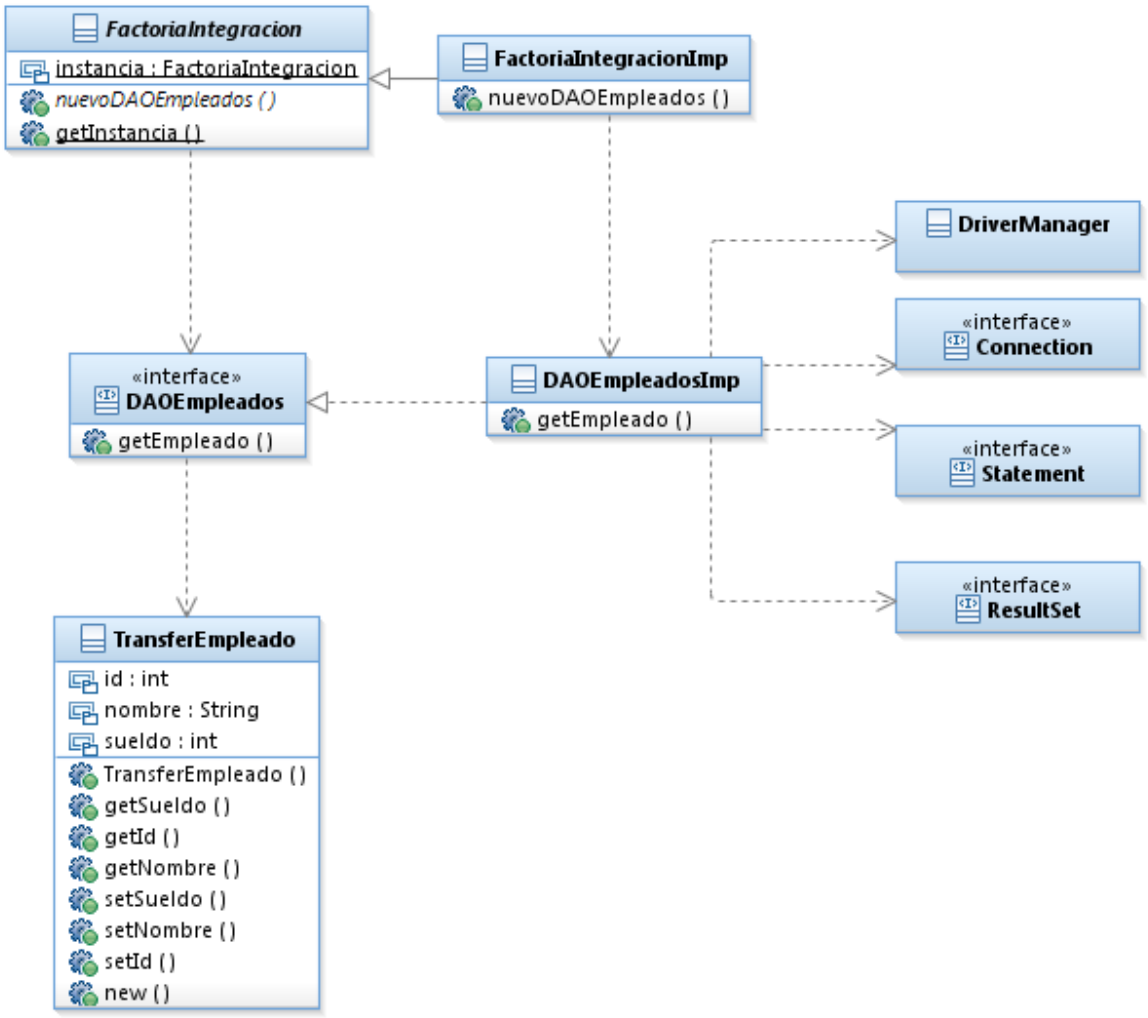


Figura 4.5 Diagrama de clases capa de integración

- *Data Access Object (DAO):* Para desacoplar la implementación del almacenamiento persistente del resto de la aplicación, y para organizar la lógica de acceso a datos y encapsular todos los accesos al almacén persistente, implementamos este patrón con la interface *DAOEmpleados* que es implementada por *DAOEmpleadosImpl*. *DAOEmpleados* es la interfaz que abstrae la implementación subyacente de acceso a

datos (en este caso empleado) para devolver de forma transparente un *Transfer Object* a la capa de negocio.

4.1.4 Diagrama de despliegue

Como se detalla en la Figura 4.6, el diagrama UML de despliegue describe los componentes software y los nodos de ejecución, que forman parte de esta aplicación. Además de dar una visión global de la plataforma y la distribución sobre la que se desea desplegar la aplicación, en este diagrama vemos los componentes hardware y software en el sistema final, su configuración y relación con otros componentes.

En este caso el contenedor de servlets Tomcat es suficiente para desplegar y ejecutar la aplicación que estará empaquetada en un artefacto llamado "nominas.war".

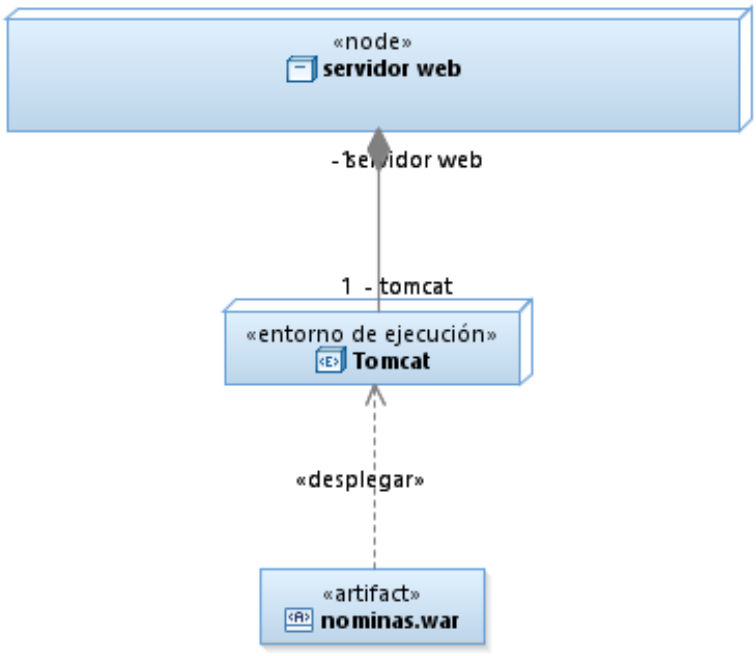


Figura 4.6 Diagrama de despliegue

4.2 Aplicaciones con seguridad

4.2.1 Aplicación web utilizando patrones CJP y CSP

En este ejemplo retomamos la aplicación descrita en la sección anterior, una aplicación web J2EE, y además de implementar los patrones CJP también implementamos los patrones CSP. Describimos el diseño de esta aplicación mediante diagramas UML.

4.2.1.1 Diagrama de despliegue

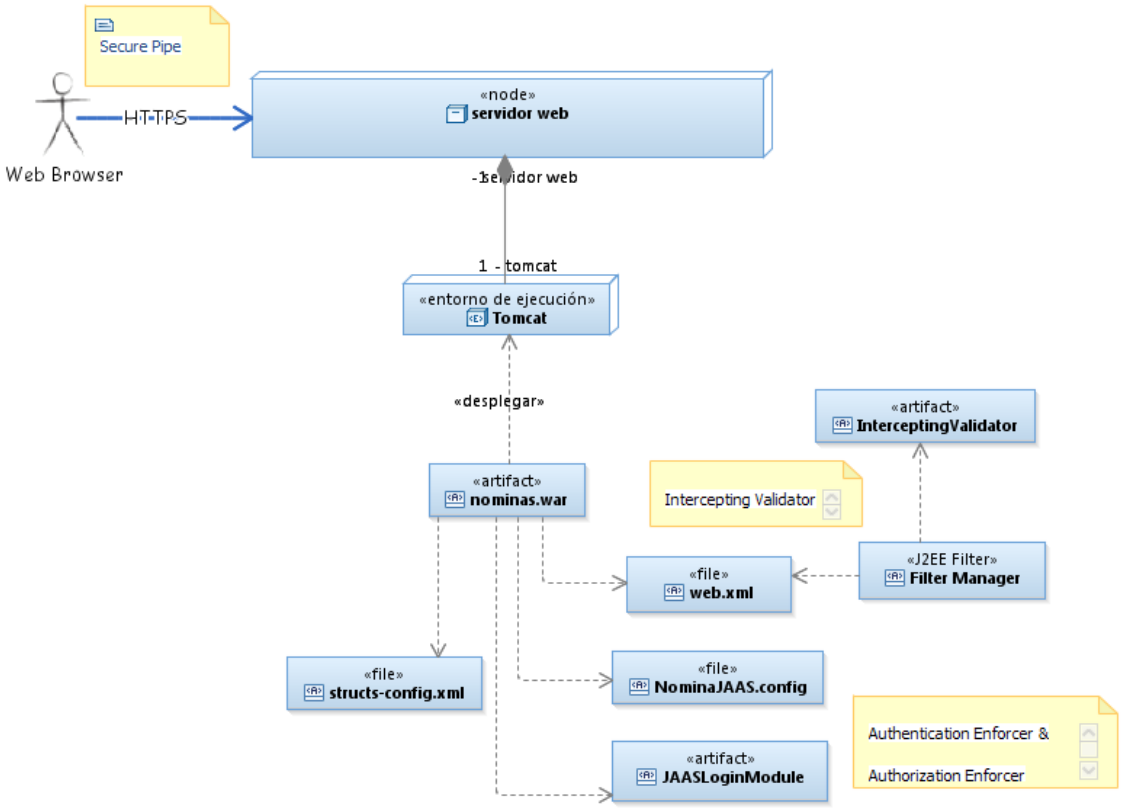


Figura 4.7 Diagrama de despliegue de la aplicación segura

En el diagrama de despliegue de la Figura 4.7 describimos la configuración de los elementos en tiempo de ejecución y los componentes software de seguridad que se ejecutan en ellos. Ahora con la implementación de la seguridad, como se puede ver en el diagrama UML, hay nuevos componentes, en su mayoría los patrones CSP de la capa de presentación. Este

diagrama nos permite observar la ubicación y relación entre los componentes y patrones implementados. Los patrones CSP implementados son:

- *Secure Pipe:* Las transacciones basadas en la Web a menudo están expuestos a la escucha, grabación, y los ataques de suplantación de identidad. Cada vez que una solicitud se pasa a través de una red insegura, los datos pueden ser interceptados o expuestos por usuarios no autorizados. Este patrón no requiere lógica de la capa de aplicación y por lo tanto reduce la complejidad de la implementación. Según se analizó en el capítulo 3, sección 3.1.6, interpretamos el patrón *Secure Pipe* como un elemento externo a la aplicación de software, convirtiéndose así en una cuestión de despliegue. Es por esto que lo implementamos exponiéndolo como un canal de comunicación HTTPS cifrado que proporciona privacidad e integridad de datos.

- *Intercepting Validator:* Existen varias estrategias para implementar el patrón que se encarga de validar el código malicioso o con formato incorrecto que proviene del cliente, la que hemos elegido se trata de un artefacto que se configura en el descriptor de despliegue de la aplicación (web.xml). En este caso, hemos decidido implementarlo como un filtro J2EE. Gracias a la especificación de Servlets 2.3, se incluye la posibilidad de usar este patrón, implementando los métodos init, destroy y doFilter de la interfaz *javax.servlet.Filter.*

- *Authentication Enforcer y Authorization Enforcer:* Muchos componentes tienen que verificar que cada solicitud esté debidamente autenticada y autorizada. Una estrategia para implementar estos patrones es como lo presentamos en el diagrama de despliegue de la Figura 4.7, representados con un artefacto que se implementa a través de un módulo de login JAAS y se despliega en el contenedor. JAAS utiliza un medio declarativo de mapeo entre permisos y recursos. Los desarrolladores pueden asignar permisos a los recursos y roles a los permisos de forma declarativa en tiempo de despliegue.

4.2.1.2 Diagramas de clases

4.2.1.2.1 Capa de presentación

Patrones CSP utilizados en esta capa son:

- *Secure Logger:* Para registrar todos eventos de la aplicación y datos relacionados con seguridad para propósitos forenses y de depuración. Como vemos en el diagrama de clases la invocación de *SecureLogger* se hacen desde la clase *SecureBaseAction* con una acción segura.

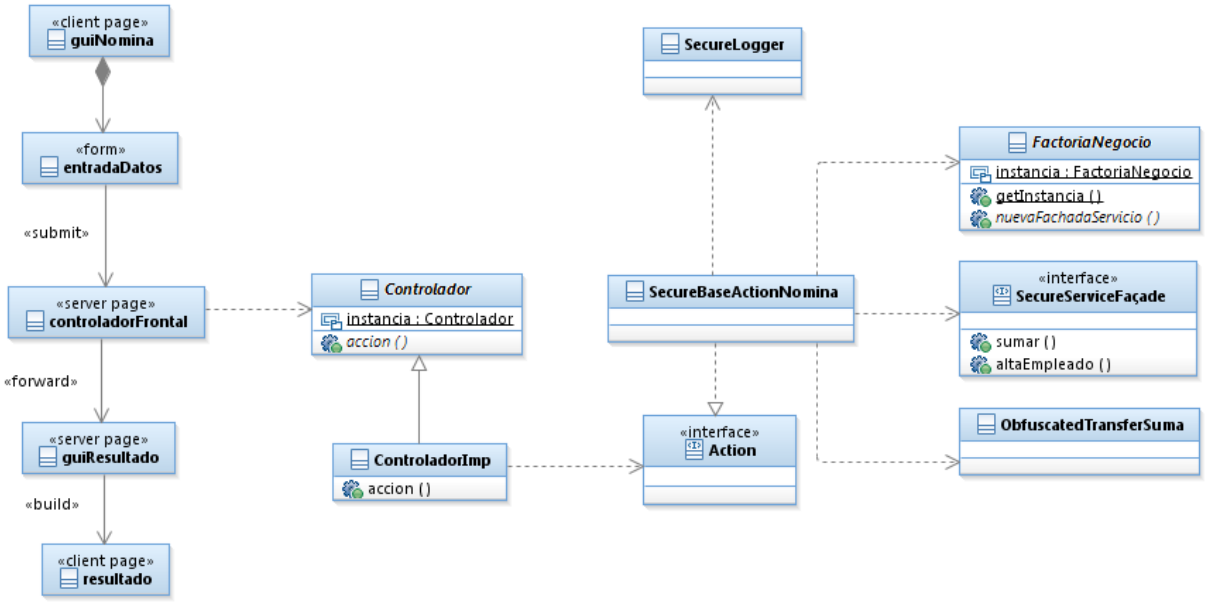


Figura 4.8 Diagrama de clases capa de presentación de la aplicación segura

- *Secure Base Action*: Es el patrón que consolida la interacción de todos los componentes relacionados con la seguridad en la capa web en un único punto de entrada. Lo utilizamos creando la clase *SecureBaseActionNomina*, que implementa la interface *Action*, ejecutando el método *action()* para cada acción que requiere la invocación de la lógica de negocio segura. Por ejemplo la invocación a *SecureLogger*. En caso de no utilizar la autorización y autenticación y validación por contenedor es este el punto donde se centraliza y se encapsula la gestión de estos componentes de seguridad. En este ejemplo hemos sustituido el *Front Controller* original por otro que utiliza comandos para cargar las acciones del controlador.

4.2.1.2.2 Capa de negocio

Un aspecto importante a tener en cuenta en la capa de negocio cuando se inicia el diseño de la aplicación es saber si la capa de negocios será o no expuesta a través de servicios web. Esto puede afectar la decisión sobre los patrones a utilizar y la sobrecarga en relación a esos patrones. En nuestro primer caso práctico consideraremos que la capa de negocio no se expone a través de servicios web.

Los patrones CSP implementados en esta capa son:

- *Obfuscated Transfer Object:* Para proteger los datos a medida que entran y salen de la aplicación y viajan entre niveles, implementamos el patrón *Obfuscated Transfer Object* en dos ocasiones, la primera es con la clase *ObfuscatedTransferSuma*, para ofuscar los datos sensibles de los empleados entre la capa de negocio y la capa de integración y la segunda es con la clase *ObfuscatedTransferSuma*, para pasar los datos de los dos empleados sumados entre la capa de negocio y la capa de presentación.

- *Secure Service Facade:* Para tener una fachada de acceso seguro a los componentes de la capa de negocio, donde se exponga de una forma uniforme una interfaz de grano grueso sobre los servicios de negocio de grano fino, hemos utilizado este patrón con la interface *SecureServiceFacade* y su implementación. Como se puede ver en el diagrama de clases de la figura 4.9, esta interface encapsula el acceso a los servicios de aplicación del negocio (*SANominas* y *SAEmpleados*), haciendo que estos no estén expuestos directamente a los clientes.

- *Audit Interceptor:* Este patrón permite interceptar y auditar las peticiones y respuestas hacia y desde la capa de negocio, de forma declarativa y desacoplada de la lógica de negocio. Lo implementamos con la clase *AuditInterceptorNominas*, quien se encarga de centralizar la auditoria de las peticiones entrantes y salientes a los recursos de la aplicación.

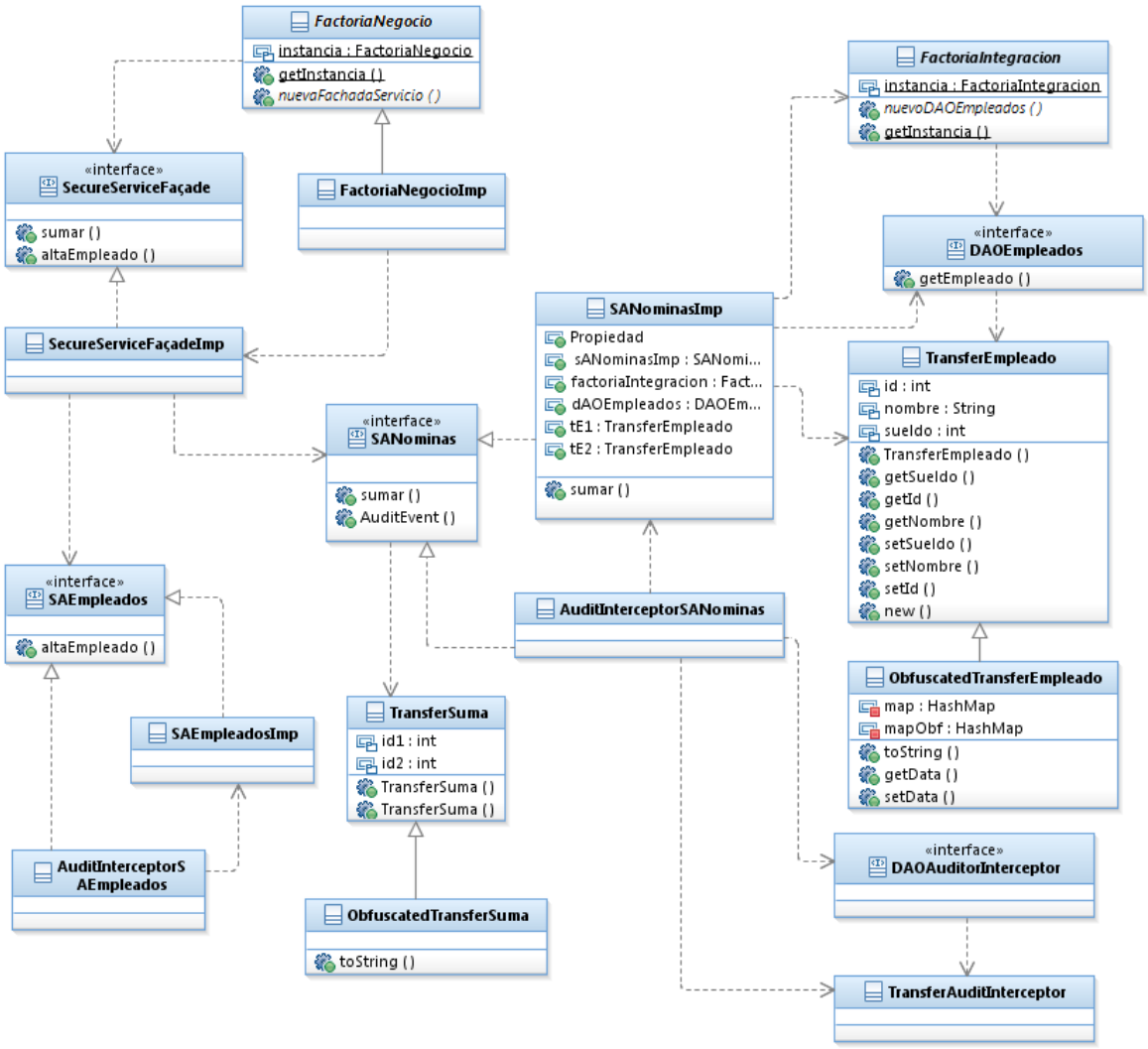


Figura 4.9 Diagrama de clases capa de negocio de la aplicación segura

Como hemos descrito en el capítulo 3, más exactamente en las secciones 3.3 y 3.4, el catálogo CSP expone una serie de patrones para las capas *Web Services* e *Identity*. Estos patrones no están pensados para ser implementados en aplicaciones como la descrita en el ejemplo anterior, que no tiene *Web Services*. Para poder describir la implementación de estos patrones del catálogo CSP en la siguiente sección hemos descrito el diseño de una aplicación que mantiene la misma funcionalidad que la anterior, pero que cuenta con capa de servicios web y gestión de la identidad.

4.2.2 Aplicación con servicios web utilizando patrones CJP y CSP

4.2.2.1 Diagrama de despliegue

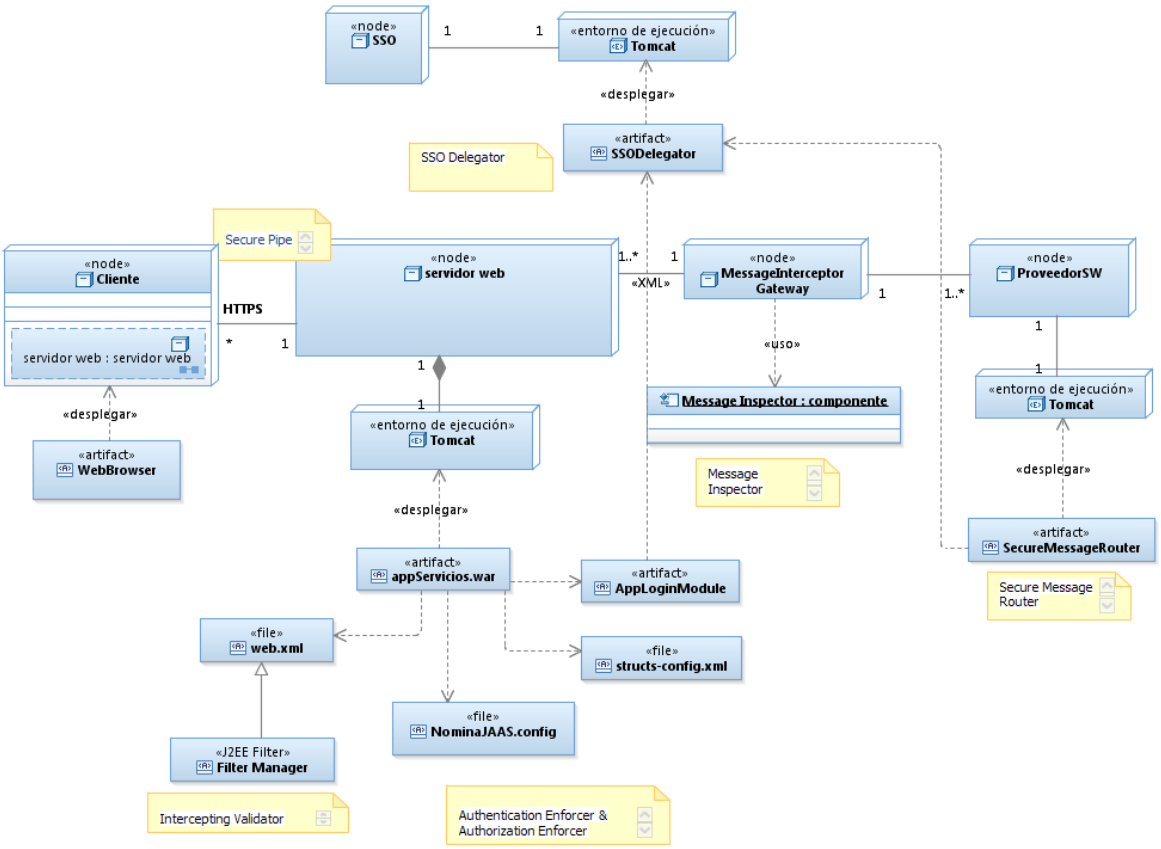


Figura 4.10 Diagrama de despliegue aplicación segura con servicios web

En el diagrama de despliegue de la Figura 4.10 describimos la configuración de los elementos en tiempo de ejecución y los componentes software que se ejecutan en ellos. Como se puede ver en el diagrama UML, en comparación con los anteriores diagramas de despliegue, se han considerado nuevos componentes de seguridad, tanto en el lado del cliente como en el lado del servidor. En este caso se trata de los patrones CSP de la capa *Web Services* e *Identity Management*. Este diagrama nos permite observar de forma global la ubicación y la relación entre los componentes, y los nuevos patrones de seguridad implementados.

Los patrones de la capa de presentación CSP: *Secure Pipe*, *Intercepting Validator*, *Authentication Enforcer*, *Authorization Enforcer*, que han sido descritos en los diagramas de

despliegue de la sección 4.2.1.1. Siguen implementados en este diagrama, y realizando la misma funcionalidad.

Los nuevos patrones CSP implementados son:

- *Message Interceptor Gateway (Web Service Tier):* Este patrón proporciona servicios de infraestructura que pueden interceptar las peticiones entrantes y las respuestas salientes para garantizar la seguridad de capa de transporte, la integridad del mensaje y la confidencialidad, cumplimiento de estándares. Como se muestra en el diagrama UML de la *Figura 4.10*, es un nodo al que llegan todas las peticiones y que se encarga de comprobar que cumplen con las especificaciones de seguridad, revisar el contenido XML usando el *MessageInspector* y dejarlos acceder a los puntos finales del *ProveedorWS* si cumplen las validaciones exigidas.

- *Message Inspector (Web Service Tier):* Este patrón comprueba y verifica la calidad de los mecanismos de seguridad a nivel de mensajes XML, valida los mensajes en el nivel de elemento para identificar la manipulación de parámetros y los ataques de inyección de mensajes a través de XPath y expresiones XQuery. Como se ve en la Figura 4.10 del diagrama de despliegue el *MessageInspector* es usado por el *MessageInspectorGateway* para proporcionar un procesamiento de seguridad a nivel de mensaje.

- *Secure Message Router (Web Service Tier):* Es un patrón utilizado cuando se requiere comunicarse de forma segura con varios puntos finales asociados utilizando la seguridad a nivel de mensaje y los mecanismos de la identidad de la federación. En nuestra aplicación este patrón funciona como un punto de aplicación de seguridad antes de invocar un servicio web. Así, cuando se recibe una petición a *ProveedorWS*, *SecureMessageRouter* recibe un mensaje XML, interactúa con un proveedor de identidad, utilizando el *SSODelegator*, aplica la seguridad a nivel de mensaje, y, en su caso, permite el acceso al servicio web invocado.

- *SSO Delegator (Identity Tier):* El proveedor de identidad facilita la gestión de identidades, inicio de sesión único (SSO) y federación de identidades para diferentes aplicaciones de participantes, proveedores de servicios y los solicitantes de servicios. El *SSODelegator* como vemos en el diagrama de despliegue de la *Figura 4.10*, reside en un nivel intermedio ente los clientes y los componentes de los servicios de gestión de identidades, encapsulando el acceso a la administración de identidades. Así los clientes no interactúan directamente con las interfaces de servicios de gestión de identidades. Un beneficio de implementar este patrón es el frustrar los robos de sesión, un fallo de seguridad muy crítico. El *SSODelegator* crea el inicio de sesión único y lo suministra a las peticiones del servicio de los servicios de seguridad pertinentes, con lo cual las peticiones del cliente tienen que estar autenticadas contra un proveedor de identidad antes de poder establecer un inicio de sesión único (SSO).

4.2.2.2 Diagramas de clases – Cliente

4.2.2.2.1 Capa de negocio

En este caso la capa cliente no tiene la implementación de los servicios de aplicación, los cuales son accedidos por una fachada de servicio segura, tal y como expone la Figura 4.11. Nótese que en este caso, la fachada segura también juega el rol de *Policy Delegate*, prefiriendo exponerla a través de un interfaz, en vez de implementarla como un *Singleton*.

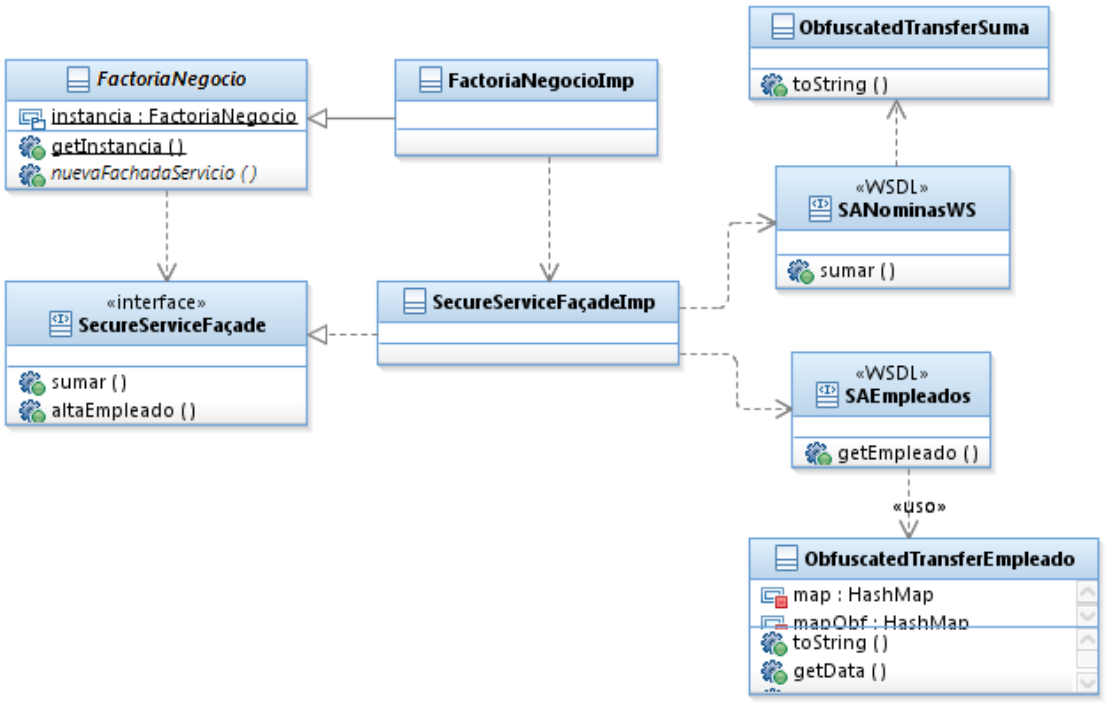


Figura 4.11 Diagrama de clases capa de negocio aplicación segura con servicios web - Cliente

4.2.2.3 Diagramas de clases –Proveedor de servicios

4.2.2.3.1 Capa de negocio

El diagrama de clases de la capa de negocio en el lado del proveedor de servicios de la Figura 4.12, mantiene la implementación del servicio de aplicación, expuesto como un servicio web, *SANominasWS* al cual se accede a través del proxy seguro *SANominaSSProxy* que intercepta el trafico XML para garantizar seguridad a nivel de mensaje XML. El otro componente importante es el *SecureMessageRouterNominaProxy* que hace de intermediario sobre los mensajes entrantes para garantizar seguridad en múltiples extremos de la aplicación, encaminando los mensajes al destino final, el proxy seguro en este caso.

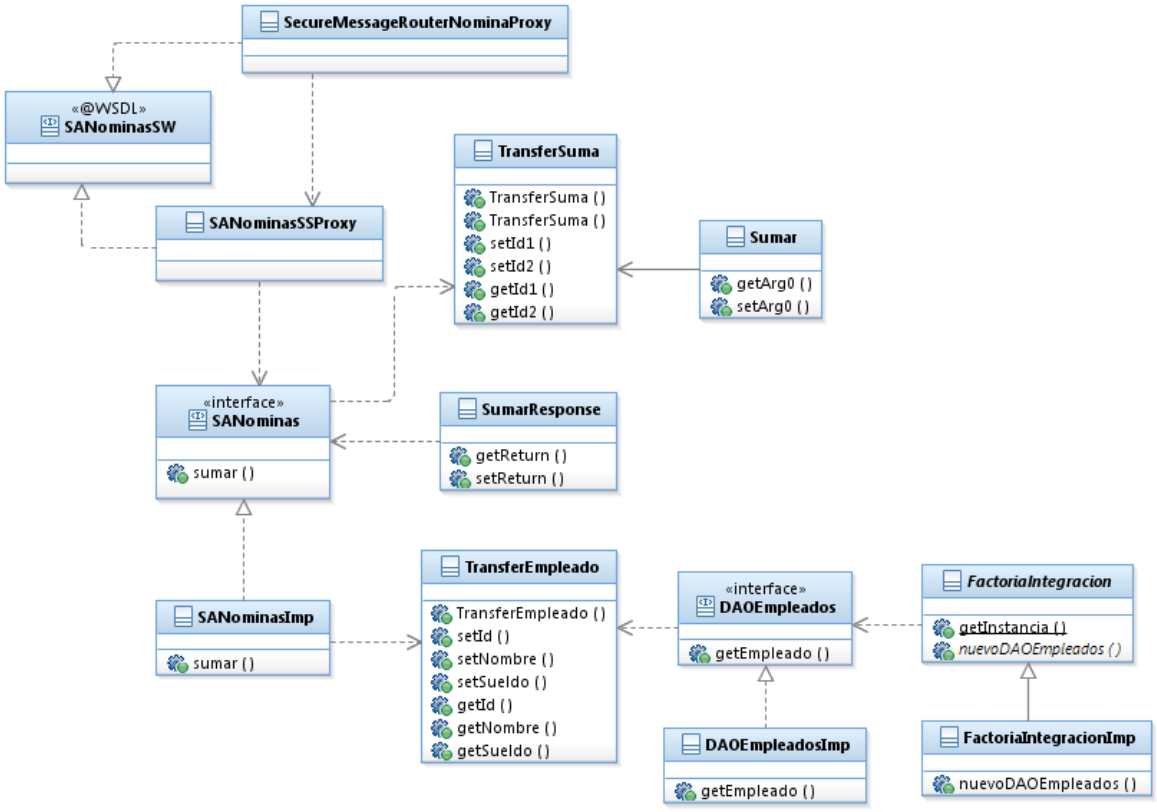


Figura 4.12 Diagrama de clases capa de negocio aplicación segura con servicios web – Servidor

5 Conclusiones y Trabajo futuro

Tal y como hemos comentado en la introducción, los resultados de este trabajo son aplicables tanto en la industria como en la academia. Desde un punto de vista industrial, este trabajo facilita el uso de patrones de seguridad en el contexto de una aplicación de arquitectura multicapa. Desde un punto de vista educativo, este trabajo establece una relación de precedencia entre los patrones de arquitectura multicapa y los patrones de seguridad. Además, propone un subconjunto de patrones de arquitectura multicapa como el mínimo necesario para comprender los patrones de seguridad. Ambas aportaciones facilitan la docencia de los patrones CSP.

Como se ha visto en este trabajo, los patrones de seguridad CSP en la arquitectura multicapa están estrechamente relacionados con los patrones de CJP. Sin embargo, en la práctica de este trabajo nos dimos cuenta que el catálogo CSP no vincula adecuadamente sus patrones de seguridad y los patrones de CJP, lo que hace difícil su comprensión unificada. Por eso en este trabajo, establecimos explícitamente la relación entre ellos, siendo una síntesis del mismo el diagrama de clases descrito en la Figura 3.23 del capítulo 3, que relaciona ambos catálogos. Esperamos que este diagrama de relaciones unificado sea un gran aporte para quienes desean utilizar patrones multicapa y patrones de seguridad.

Como hemos encontrado que uno de los problemas de los patrones del catálogo CSP es que están clasificados por capas que no coinciden con las utilizadas en el catálogo CJP. Después del análisis realizado en este trabajo, proponemos la siguiente relación entre capas CJP y capas CSP. La capa web CSP puede asimilarse a la capa de presentación (aunque el patrón *Secure Service Proxy* debe ser incluido en la capa de integración) y las capas de negocio en ambos catálogos se corresponden. Los patrones de la capa de servicios web CSP podrían ubicarse en la capa de integración debido a que el Web *Service Broker* CJP se encuentra en esta capa y los patrones de CSP están estrechamente relacionados con este patrón. Los patrones de gestión de identidad CSP podrían ser localizados en la capa de negocio, porque los patrones Single *Sign-On Delegator* y *Password Synchronizer* pueden concebirse como servicios de aplicación que se han dedicado a cuestiones de seguridad.

A diferencia de los patrones de CJP, es difícil definir un conjunto mínimo de patrones de CSP. Sin embargo en base a este trabajo podremos decir que en la capa de presentación, posiblemente, *Authentication*, *Authorization Enforcer* y el *Intercepting Validator* deben estar presentes en cualquier aplicación para la autenticación y autorización de usuarios y la validación de las entradas. En la capa de negocio *Obfuscated Transfer* puede ser el único elemento necesario. Si las capas de presentación y negocio son independientes, se crearía la necesidad de un *Policy Delegate* y *Secure Service Façade* para la invocación segura de lógica remota. En la capa de integración, el patrón *Message Interceptor Gateway y Message* Inspector podría ser necesario si se definen servicios Web. Es decir, que con una base del 34 % de los patrones del catálogo CSP tendríamos una aplicación con seguridad de extremo a extremo con arquitectura multicapa y que incluye servicios web.

Por tanto, en base al análisis llevado a cabo en este trabajo podemos afirmar que los patrones de CJP mínimos para explicar y comprender los anteriores patrones de CSP son: *Front Controller* y *Application Controller*, *Context Object*, *Intercepting Filter*, *Transfer Object*, *Business Delegate*, *Session Façade* y *Web Service Broker*, que representan el 40% de los patrones de CJP.

Así, en el contexto de un curso de grado, si sólo pueden explicarse un subconjunto de los patrones CJP junto con los principales patrones CSP, sólo estos patrones CJP necesitan ser explicados previamente. Si se va a presentar todo el catálogo de CSP, además de los patrones CJP previamente identificados también sería necesario explicar los siguientes patrones CJP: *Web Service Activator*, *Application Service* y *Business Object*. En total representan el 52% de los patrones de CJP.

En cualquier caso, los patrones *Application Service* y *Business Object* son parte de los patrones básicos del catálogo CJP. Por lo tanto, los patrones básicos de arquitectura multicapa y los patrones relacionados con la invocación de lógica de negocio remota son elementos esenciales que se necesitan para comprender todo el catálogo de CSP. Los cursos dedicados a los patrones de seguridad podrían así aprovecharse de este hecho, explicando los patrones CJP básicos identificados en este documento y omitiendo la mitad del catálogo CJP.

Además, este trabajo ha identificado la relación de precedencia entre los patrones de CSP, que puede ser muy útil cuando se trata de enseñarlos. La experiencia nos dice que uno de los problemas en la explicación de un catálogo de patrones es la definición de una secuencia coherente, a diferencia de la presentación alfabética que presentan los catálogos. Así, por ejemplo, el patrón *Message Interceptor Gateway* debe ser explicado antes del *Message Interceptor*, a pesar de que se presentan en orden inverso en el catálogo de CSP.

El uso de patrones convencionales y patrones de seguridad en la construcción y arquitecturas y diseño de aplicaciones empresariales, cada vez es más utilizado por marcos y metodologías de desarrollo. Sin embargo es importante tener en cuenta que las aplicaciones empresariales son todas diferentes, y que los diferentes problemas conducen a diferentes formas de hacer las cosas. Por tanto, el desafío en el diseño de una aplicación está en saber elegir entre las diferentes alternativas, juzgar las ventajas y desventajas de la utilización de una u otra alternativa y saber personalizarla para el problema en concreto. Precisamente, este trabajo intenta ser una herramienta más para ayudar a tomar este tipo de decisiones.

Por último, dentro del trabajo futuro, se debería proceder a la implementación de los diseños realizados en este trabajo. También sería interesante analizar la relación del catálogo CSP con otros catálogos de seguridad, y la relación de estos catálogos con CJP. En este sentido, el trabajo desarrollado en esta memoria facilitará dicho análisis y relación. Finalmente, consideramos interesante realizar un análisis de la aplicabilidad del desarrollo dirigido por modelos para facilitar la inclusión automática de patrones arquitectónicos y de seguridad en base a modelos de diseño de alto nivel.

Bibliografía

(ACM/IEEE 2004) ACM/IEEE Computer Society, *Software Engineering 2004*, 2004, [En línea], [fecha de consulta: 15 Diciembre 2013]. Available: http://sites.computer.org/ccse/SE2004Volume.pdf

(ACM/IEEE 2013) ACM/IEEE Computer Society, *Computer Science Curriculum 2008, An Interim Revision of CS 2001*, 2008, [en línea], [fecha de consulta: 15 Diciembre 2013]. Available: http://www.acm.org//education/curricula/ComputerScience2008.pdf

(Alexander 1977) C. Alexander. *A Pattern Language*. Oxford University Press, New York, (1977).

(Alur et al. 2001) D. Alur, J. Crupi, D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2001.

(Alur et al. 2003) D. Alur, J. Crupi, D. Malks. *Core J2EE Patterns: Best practices and design strategies*. Sun Microsystems, Prentice Hall (2003).

(Beck el al. 1987) K. Beck, Apple Computer, Inc., W. Cunningham, Tektronix, Inc., *Using Pattern Languages for Object-Oriented Programs*. Technical Report No. CR-87-43, September 17 (1987), [en línea], [fecha de consulta: 11 Enero 2014] Available: http://c2.com/doc/oopsla87.html

(Blakley et al. 2004) B. Blakley, C. Heath and Members of the Open Group Security Forum, *Technical Guide: Security Design Patterns*, 2014, [en línea], [fecha de consulta: 10 Noviembre 2013]. Available: http://www.opengroup.org/bookstore/catalog/g031.htm

(Burke 2013) B. Burke, *RESTful Java with JAX-RS 2.0*. 2nd Edition.O'Reilly, 2013.

(Buschmann et al. 2001) F. Buschmann; R. Meunier; H. Rohnert; P. Sommerlad, M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. John Wiley & Sons, 2001.

(Buschmann et al. 2007) F. Buschmann, K. Henney, D.C. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. John Wiley & Sons, 2007.

(Buschmann 2007) F. Buschmann; K. Henney; D. C. Schmidt, *Pattern Oriented Software Architecture Volume 5: On Patterns and Pattern Languages*, John Wiley & Sons, 2007.

(Appleton 2000) B. Appleton, *Patterns and Software: Essential Concepts and Terminology (last modified 02/14/2000)*, [en línea], [fecha de consulta: 20 Noviembre 2013]. Available: http://csis.pace.edu/~grossman/dcs/Patterns%20and%20Software-%20Essential%20Concepts%20and%20Terminology.pdf

(Braga et al. 98) A. Braga, C. Rubira, R. Dahab, *Tropyc: A pattern language for cryptographic object-oriented software*, Chapter 16 in Pattern Languages of Program Design 4 (N. Harrison, B. Foote and H. Rohnert, Eds.). Also in Proceedings of PLoP'98, http://0-jerry.cs.uiuc.edu.cisne.sim.ucm.es/~plop/plop98/final_submissions/

(Coté JAAS 2009) M. Coté, *JAAS in Action*, 2009, [en línea], [fecha de consulta: 22 Noviembre 2013]. Available: http://www.jaasbook.com

(Crawford y Kaplan 2003) W. Crawford, J. Kaplan. *J2EE Design Patterns*. O'Reilly Media, California. 2003.

(Coplien 2013) J. Coplien, *Software Design Patterns: Common Questions and Answers.* AT&T Bell Laboratories, [en línea], [fecha de consulta: 15 Octubre 2013]. Available: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=7621ABA2C54BB4E03C9790A0C3A6E763?doi=10.1.1.38.5376&rep=rep1&type=pdf

(Das y Garrido 1998) F. Das Neves, A. Garrido, 'Bodyguard', *Chapter 13 in Pattern Languages of Program Design Volume 3*. Addison-Wesley 1998.

(Dougherty et al. 2009) C. Dougherty, K. Sayre, RC Seacord, D. Svoboda, K. Togashi, *Secure Design Patterns, Technical Report CMU/SEI-2009-TR-010*, March 2009, [en línea], [fecha de consulta: 19 Enero 2013]. Available: http://resources.sei.cmu.edu/asset_files/technicalreport/2009_005_001_15110.pdf

(Erl 2009) T. Erl, *SOA Design Patterns*. Prentice Hall PTR, 2009.

(Essmayr et al. 1997) W. Essmayr, G. Pernul, A. M. Tjoa, *Access controls by object-oriented concepts*. Proceeedings of 11th IFIP WG 11.3 Working Conference on Database Security, August 1997

(Fernandez 2013) E. B Fernandez-Buglioni, *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley, 2013.

(Fernandez 1993a) E. B Fernandez, M. M. Larrondo-Petrie, E. Gudes, *A method-based authorization model for object-oriented databases*. Proceeedings of the OOPSLA 1993 Workshop on Security in Object-oriented Systems, 70–79.

(Fernandez 1994a) E. B. Fernandez, J. Wu, M. H. Fernandez, *User group structures in object-oriented databases*. Proceeedings of the 8th Annual IFIP W.G.11.3 Working Conference on Database Security, Bad Salzdetfurth, Germany, August 1994.

(Fowler et al. 2003) M. Fowler; D. Rice; M. Foemmel, E. Hieatt, R, Mee; R. Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2003.

(Gamma et al. 1994) E. Gamma; R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

(Geary y Horstmann 2010) D. Geary, C. S. Horstmann, *Core JavaServer Faces*. Third ed., Prentice Hall, 2010.

(Goncalves 2013) A. Goncalves, *Beginning Java EE 7*. Apress, 2013.

(Hafiz 2012) M. Hafiz, P. Adamczyk and R. E. Johnson, *Growing a pattern language (for security)*. Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP), 2012

(Hafiz et al. 2007) M. Hafiz, P. Adamczyk, R. E. Johnson, *Organizing Security Patterns*. IEEE Software, vol. 24, no. 4, pp. 52-60, July-Aug. 2007, doi:10.1109/MS.2007.114.

(Hansen 2007) M. D. Hansen, *SOA Using Java Web Services*. Prentice Hall, 2007.

(Hillside Group Web 2014) The Hillside Group. *Patterns Home Page – Your Patterns Library*, [en línea] [fecha de consulta: 18 de Enero de 2014]. Available: http://www.hillside.net/patterns.

(Hogg et al. 2006) J. Hogg, D. Smith, F. Chong, D. Taylor, L. Wall, and P. Slater. *Web service security: Scenarios, patterns, and implementation guidance for Web Services Enhancements (WSE) 3.0*. Microsoft Press, 2006.

(JAAS Oracle) *Java Authentication and Authorization Service (JAAS) Reference Guide*. Oracle, [en línea], [fecha de consulta: 20 de Enero de 2014]. Available: http://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/JAASRefGuide.html.

(Keith 2013) M. Keith; M. Schincariol, *Pro JPA 2*.Second Edition, Apress. 2013.

(Kienzle 2002) D. M. Kienzle, M. C. Elder, D. Tyree, J. Edwards-Hewitt, *Security Patterns Repository, version 1.0, 2002*, [en línea], [fecha de consulta: 10 de Marzo de 2013]. Available: http://www.scrypt.net/~celer/securitypatterns/repository.pdf http://www-03.ibm.com/security/patterns

(Kircher y Jain 2004) M. Kircher, P. Jain, *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management*. John Wiley & Sons, 2004.

(Munawarhafiz Web) Security Pattern Catalog, [en línea] [fecha de consulta: 20 de marzo de 2014]. Available: http://www.munawarhafiz.com/securitypatterncatalog/index.php

(Miles y Hamilton 2006) Russell Miles, Kim Hamilton. *Learning UML 2.0*. O'Reilly Media, Inc, 2006

(MSD Web 2014) Microsoft Patterns and Practices Development Center, [en línea] [fecha de consulta: 20 de febrero de 2014]. Available: http://msdn.microsoft.com/en-us/library/ff921345.aspx.

(Navarro et al. 2012) Navarro, J. Cristóbal, C. Fernández-Chamizo, A. Fernández-Valmayor, *Architecture of a multiplatform virtual campus.* Software: Practice and Experience 42 (2012) 1229-1246.

(Open Group) Open Group, [en línea] [Fecha de consulta: 23 de enero de 2014]. Available: http://www.opengroup.org/security/l2-jsec.htm

(Perry 2002) S. Perry, *Java Management Extensions*, O'Reilly, 2002.

(Schumacher et al. 2005) M. Schumacher, E. B. Fernandez, D. Hybertson, F. Buschmann and P. Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, Inc., 2006

(Schmidt et al. 2000) C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000.

(SEC Web 2014) The Security Patterns Page, maintained by M. Schumacher, [en línea] [Fecha de consulta: 1 de Febrero de 2014]. Available: http://www.securitypatterns.org

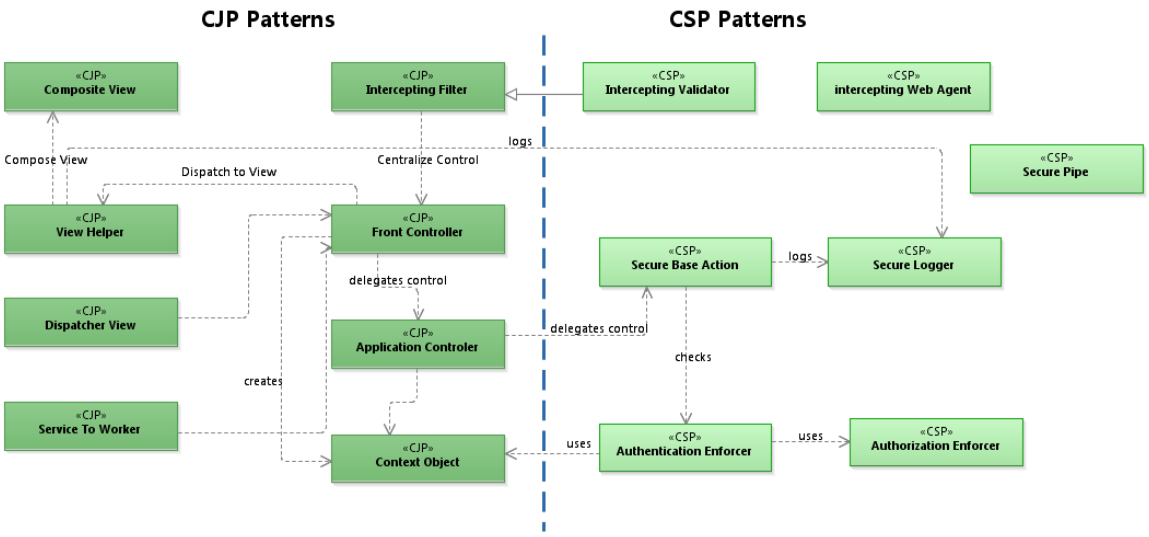
(Si 2003) S. Si Alhir. *Learning UML 2.0*. O'Reilly Media, Inc, 2003

(Steel et al. 2005) C. Steel; R. Nagappan; R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*, Prentice Hall, 2005

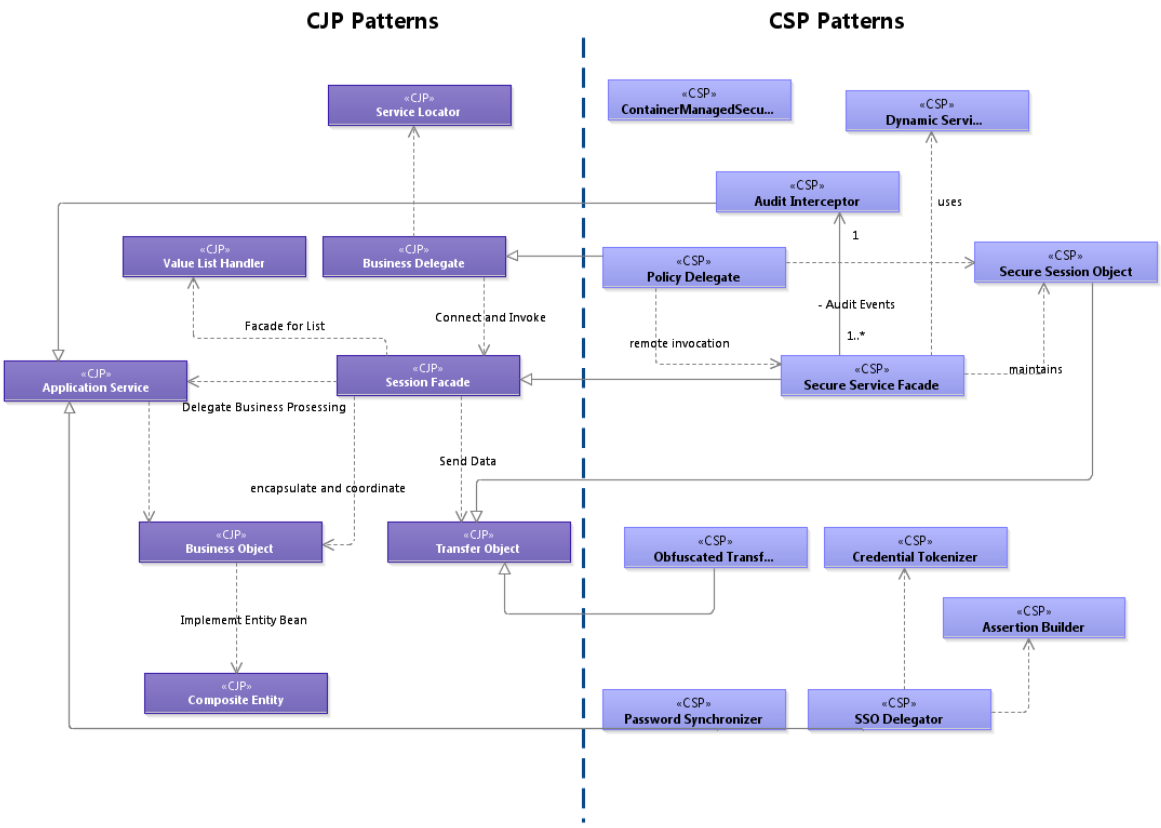
(Yoder y Barcalow 1997) J. Yoder and J. Barcalow, *Architectural Patterns for Enabling Application Security*. Proceedings PLOP'97, http://0-jerry.cs.uiuc.edu.cisne.sim.ucm.es/~plop/plop97 Also Chapter 15 in Pattern Languages of Program Design, vol 4 (N. Harrison, B. Foote and H. Rohnert, eds.), Addison-Wesley, 2000

(Yskout 2006) K. Yskout, T. Heyman, R. Scandariato and W. Joosen, *A System of Security Patterns*. Rept. CW469, Dec. 2006, Dept. of Computer Science, Katholieke Universiteit Leuven, Belgium.

ANEXO 1. Relación entre CJP y CSP en la capa de presentación



ANEXO 2. Relación entre CJP y CSP en la capa de negocio



ANEXO 3. Relación entre CJP y CSP en la capa de integración