ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Desarrollo de Aplicaciones Seguras: Un Patrón Arquitectónico Siguiendo un Enfoque Orientado a Aspectos

AUTOR: Angel David Melero Sánchez
DIRECTOR: Pedro Sánchez Palma
Enero / 2006

Autor	Angel David Melero Sánchez
E-mail del Autor	admelero@hotmail.com
Director(es)	Pedro Sánchez Palma
E-mail del Director	pedro.sanchez@upct.es
Codirector(es)	
Título del PFC	Desarrollo de Aplicaciones Seguras: Un Patrón Arquitectónico Siguiendo un Enfoque Orientado a Aspectos
Descriptores	Seguridad, Programación Orientada a Aspectos.

Resumen

El presente proyecto se basa en la implementación de un paquete de seguridad capaz de aportar los requisitos que el contexto actual demanda a través de la utilización de las técnicas de la programación orientada a aspectos.

Para su consecución ha sido necesaria la elaboración de una filosofía y patrón de reusabilidad capaz de aportar transparencia y portabilidad a la implementación desarrollada. Así mismo se ha dispuesto de un ejemplo de utilización a partir de la generación de una aplicación con las más comunes necesidades de seguridad.

El trabajo realizado puede servir para extraer distintos criterios acerca del desarrollo de software así como también como base de futuros trabajos tanto a un nivel específico como a nivel más general o filosófico.

La implementación ha sido llevada a cabo en Java (JDK 1.5) a partir de sus APIs de seguridad y registro y en su extensión orientada a aspectos AspectJ (versión 1.5)

Titulación	Ingeniería Técnica de Telecomunicación, esp. Telemática
Intensificación	
Departamento	TIC
Fecha de Presentación	Enero - 2006

Agradecimientos

- A Pedro Sánchez Palma por sus consejos y por su paciencia y flexibilidad para mis constantes retrasos.

- A mis padres por la libertad que me ofrecen.

- A mis ‘sobrinos’ Gerardo, Jesús, Alberto y al ‘tío’ Jose Antonio por su apoyo anímico eventual.

Índice

Introducción

El contexto del desarrollo de software en la actualidad está cambiando constantemente, cada vez los medios son más complejos y los requerimientos mayores, el declinar exponencial del coste de la computación y la comunicación y el cada vez más dinámico entorno de desarrollo de sistemas de larga vida conducen a los desarrolladores a buscar métodos más efectivos en los que la eficiencia comienza a perder importancia frente a cualidades como la escalabilidad y reusabilidad.

La gran mayoría de las técnicas utilizadas en hoy en día se basan en el paradigma de la programación orientada a objetos, el cual ha demostrado ser capaz de aportar capacidad para afrontar de manera satisfactoria muchos de los requerimientos exigidos por la demanda actual mediante su capacidad para modelar funcionalidades comunes en módulos. Sin embargo dicho paradigma no ha sido capaz aún de afrontar el correcto modelado de las funcionalidades denominadas 'transversales', las cuales se implementan de manera desmodularizada a través del código principal de las aplicaciones, causando una gran problemática en su desarrollo, evolución y mantenimiento.

En respuesta a la necesidad de solucionar las carencias de la orientación a objetos y mejorar la capacidad para afrontar de manera más eficiente el desarrollo y postdesarrollo de los sistemas ha nacido el paradigma de la programación orientada a aspectos, a partir del cual han aparecido varias técnicas capaces de incrementar el potencial para afrontar de manera efectiva la implementación de funcionalidades transversales.

Uno de los objetivos más usuales e importantes de estas técnicas es el campo de la seguridad, el cual supone un gran problema a la hora de cumplir los requisitos actuales de desarrollo debido a su inherente transversalidad. La seguridad tiene una arquitectura compleja y aunque es posible la encapsulación de su funcionamiento no es posible hacer lo propio con el momento y la importancia del contexto de ese funcionamiento.

Debido a las circunstancias la seguridad representa un desafío para la evolución de las técnicas de implementación actuales habiéndose convertido en un clásico y cada vez más popular ejemplo de concern transversal y por tanto en uno de los candidatos más legítimos para la validación de las técnicas AOP.

Este proyecto ha tenido como objetivo principal la elaboración de una implementación de seguridad genérica basada en la programación orientada a aspectos, capaz de aportar su funcionalidad de manera absolutamente modularizada y reutilizable. Dicha implementación consistirá en un paquete realizado mediante la combinación de código Java y código realizado con la implementación más madura y utilizada del contexto de la programación orientada a aspectos: AspectJ.

La utilización de la programación orientada a aspectos permitirá la modularización no intrusiva del código elaborado de modo que el código orientado a objetos al que se aplique no se vea modificado, sin embargo será necesario usar algún tipo de metodología para conseguir una reusabilidad total del motor en el que reside la funcionalidad de seguridad. Se diseñará para ello un patrón de redefinición capaz de aportar una metodología clara y efectiva para la reutilización del código en AspectJ, el cual ha servido como base para la confección del paquete de seguridad. Dicho patrón consistirá fundamentalmente en la diferenciación clara de aquella parte del código aspectual con la que el desarrollador deba

interactuar de aquella parte que le sea posible ignorar, o lo que es lo mismo, la división en la parte ‘funcional’ y la parte de ‘acoplamiento’.

Para la implementación relativa a la parte funcional de los aspectos se abordará la arquitectura de seguridad de Java la cual aporta de por sí modularidad y flexibilidad en el código motor mediante el polimórfico y configurable diseño de sus APIs. Dichos APIs utilizados serán JCE, JSSE, JAAS y adicionalmente el API de registro Logging.

Así mismo se elaborará un caso de utilización en el que se aplicará el paquete generado a una aplicación con las distintas necesidades de seguridad genéricas con el objetivo de mostrar a modo de ejemplo el funcionamiento del código implementado y la capacidad de las técnicas y metodologías utilizadas.

A partir de lo realizado en este proyecto se podrá valorar no solo la capacidad del paquete implementado si no también la importancia de una filosofía de utilización directa y portable para la implementación de funcionalidades transversales a partir del estado actual de las técnicas de programación orientada a aspectos.

La Programación Orientada a Aspectos y AspectJ

2.1 El Desarrollo de Software en la Actualidad

El entorno actual donde se desenvuelve la producción de software es un mundo complejo, de cambio constante y de gran competitividad en el cual los objetivos del diseño de software han de centrarse principalmente en la calidad, los costes de producción, el mantenimiento y la evolución.

La metodología de la orientación a objetos (OOP)[1], incluyendo análisis, diseño y programación, trajo a las ciencias de la computación la capacidad de diseñar software de manera más acorde a como los sistemas pueden verse en el mundo real usando como herramienta primaria el objeto, el cual pretende ser la representación de un componente primario en el contexto del problema o lo que es lo mismo la modularización de una funcionalidad o requisito de la aplicación. Como tecnología la OOP ha pasado con el tiempo a una fase de madurez y su eficacia ha sido probada de manera relativamente satisfactoria tanto en pequeños como grandes proyectos, aportando principalmente modularidad y capacidad de reutilización y por tanto menor complejidad de implementación y costes reducidos de mantenimiento y evolución.

El problema fundamental de las técnicas de desarrollo de software basadas en OOP reside en que los sistemas de la actualidad deben de ser vistos como sistemas divididos en funcionalidades o *concerns*[2], pero sin embargo la naturaleza de muchas de estas es *transversal*, o lo que es lo mismo, su código habría de extenderse en varios módulos que no estarían asociados directamente a cada concern y que por tanto violarían la ley de modularización o encapsulación propia de la OOP, dificultando así mismo la comprensibilidad, reusabilidad y la capacidad de evolución y mantenimiento del código.

Visto desde otra perspectiva, las lógicas de los requerimientos en los que puede ser descompuesto un sistema pertenecen a varias dimensiones[2], las cuales puede entenderse que son ortogonales o trasversales, sin embargo la OOP tiende a implementar esas lógicas en una sola dimensión principal teniendo como resultado varias dimensiones en los requerimientos y una dimensión en la implementación, de manera que dicha ortogonalidad desaparece.

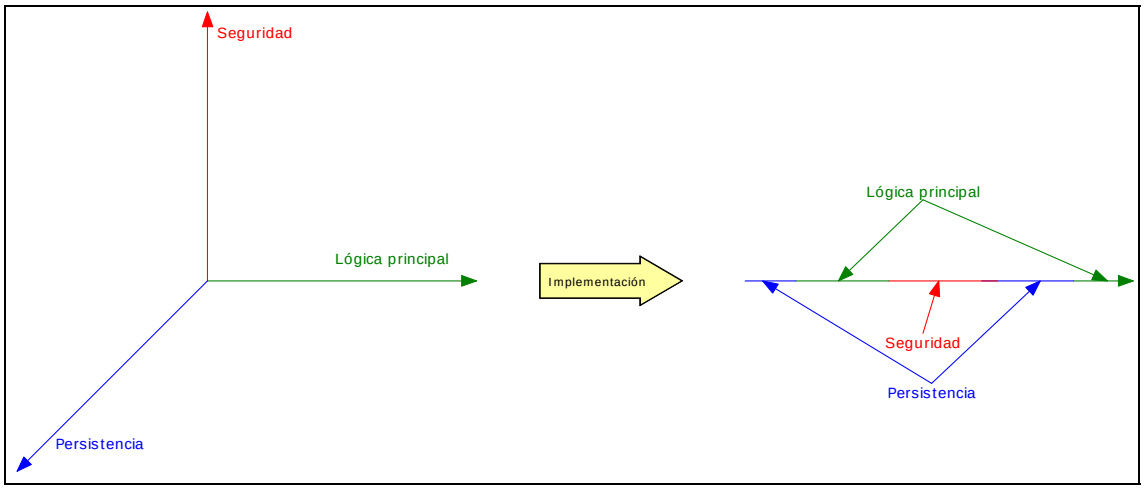

Figura 1 - Problemática vista desde la perspectiva de las dimensiones

El otro gran problema de las técnicas basadas en OOP reside en el hecho de que añadir o cambiar un concern en un sistema fuera de la fase de diseño suele ir asociado a cambios engorrosos que reducen la calidad de la implementación. Un buen diseño inicial puede intentar tener en cuenta los requerimientos presentes y futuros pero la precognición de éstos no siempre es posible sino sobrediseñando el sistema para contemplar posibles futuros requerimientos de baja probabilidad, lo cual tendría como consecuencia un sistema sobrecargado más complejo y difícil de entender e implementar de lo deseable.

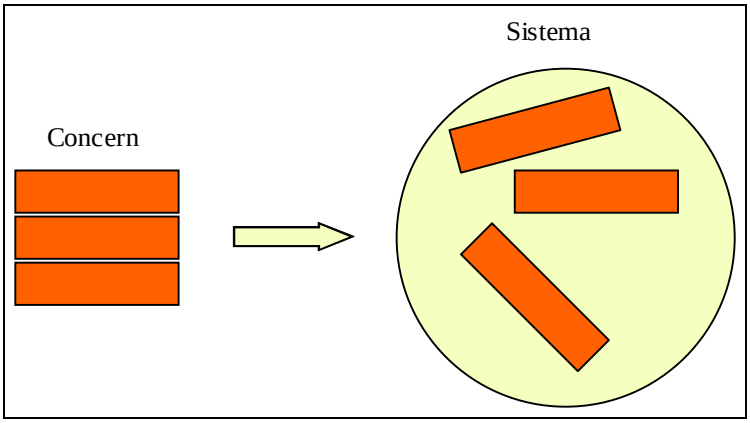

Figura 2 - Adición de un concern en la fase de postdiseño

La consecuencia principal de esta falta de flexibilidad en la implementación consiste en la aparición de código que se disemina en las distintas clases aumentando la complejidad y comprensibilidad de la implementación. Este código es denominado *código intrusivo* y su síntoma se representa principalmente como *enredo de código[2]*, el cual aparece cuando un módulo contiene código perteneciente a varios concerns distintos y dificulta la modificación, reutilización y testeo del módulo.

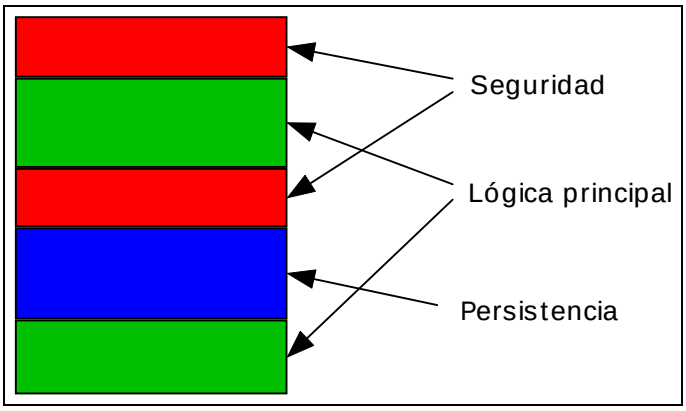

Figura 3 - Enredo de código

Así mismo, como consecuencia de la falta de focalización surge la *dispersión de código*[2] debido a que parte del código de los concerns está distribuido en varios módulos. Este fenómeno puede clasificarse en dos categorías: la duplicación de bloques de código y la complementariedad de bloques de código. La duplicación aparece cuando se repiten bloques de código con la misma funcionalidad a lo largo del código del programa y la complementariedad cuando varios módulos implementan partes complementarias de una funcionalidad.

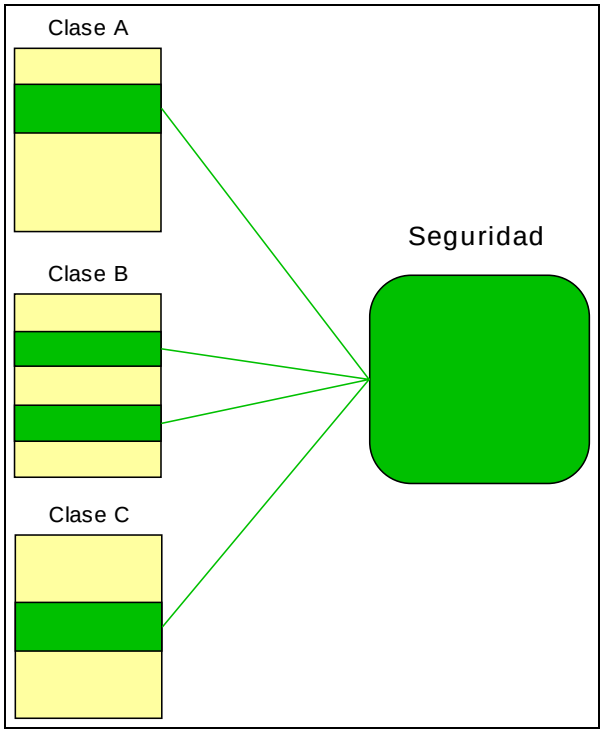

Figura 4 - Dispersión de código

Las consecuencias finales de los síntomas negativos de la utilización de la OOP son:

- Baja productividad y mala calidad: la implementación simultánea de varios concerns desenfoca y entorpece la implementación del concern principal y dificulta la supervisión de éste.
- Mala comprensibilidad: la implementación simultánea de varios concerns dificulta la visión de la correspondencia entre el código y los concerns a los que pertenece.
- Baja reusabilidad: la clave para la reusabilidad se encuentra en la reutilización de módulos que gestionen un concern de manera clara y delimitada.
- Dificultad para evolucionar: los concerns no pueden ser adecuadamente focalizados para su evolución.

2.2 La Programación Orientada a Aspectos

Como se expuso anteriormente el desarrollo de software de hoy día obliga a entender los sistemas como un conjunto de concerns. Estos representan una funcionalidad, objetivo, concepto o área de interés en un sistema en el cual pueden pertenecer a varias dimensiones.

La metodología predominante para abordar esta visión multidimensional de los sistemas consiste en el paradigma de la programación orientada a aspectos (AOP) el cual afronta la complejidad del sistema generalmente atendiendo a la implementación de los distintos concerns en dos dimensiones principales: la dimensión principal y la dimensión perteneciente a aquellas funcionalidades que atravesarían esa dimensión principal. Aquellos concerns clasificados en la dimensión transversal son denominados *aspectos*[3].

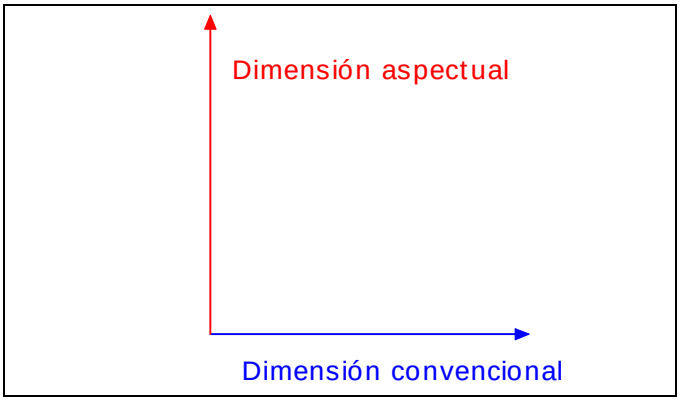

Figura 5 - Dimensiones en la AOP

La programación orientada a aspectos (AOP) permite capturar los diferentes concerns que componen una aplicación en entidades bien definidas, eliminando las dependencias innecesarias entre los módulos del sistema, consiguiendo de esta forma evitar los problemas de la OOP relativos al enredo y dispersión de código y en consecuencia

mejorando la calidad del desarrollo de software. Así mismo aporta la solución idónea para el problema relativo a la falta de flexibilidad de la OOP para añadir funcionalidades en fase de postdiseño, ofreciendo la posibilidad de añadir futuros requerimientos sin alterar la arquitectura básica del sistema al no generar código intrusivo.

Sus principales beneficios técnicos son:

- Capacidad para modular concerns transversales.
- Mejora de la capacidad de evolución.
- Mayor reusabilidad del código.
- Capacidad de postergar decisiones sobre el diseño del sistema.
- Mayor desarrollo en paralelo y más especializado.
- Mayor comprensibilidad del código y menor probabilidad de fallos.

La utilización de AOP no supone una solución para la falta de diseño así como tampoco significa el reemplazo del paradigma OOP si no la extensión del mismo. Sin embargo el uso de AOP si que supone un cambio en la metodología de desarrollo, la cual está comúnmente asociada con las siguientes fases:

1. Descomposición aspectual: se determinan cuales son los concern principales y cuales los transversales.
2. Implementación de concerns: cada concern se implementa independientemente, utilizando comúnmente técnicas OOP para los concerns principales y técnicas AOP para aquellos que son transversales.
3. Recomposición aspectual: se genera el sistema final mediante la recomposición de los dos tipos de concerns implementados independientemente.

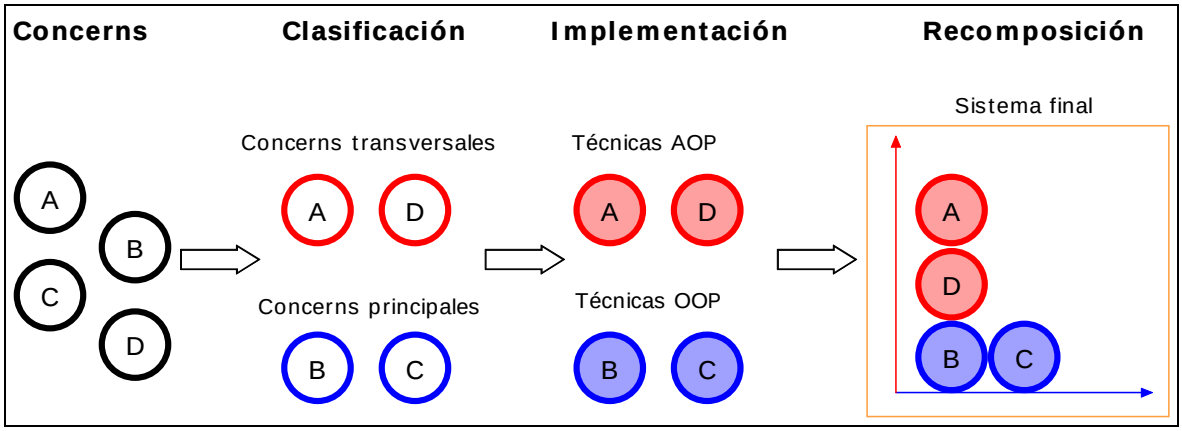

Figura 6 - Fases del desarrollo en AOP

Cada técnica basada en AOP aporta generalmente dos lenguajes: el lenguaje base y el lenguaje aspectual. El primero es aquel que define el funcionamiento principal y generalmente suele ser un lenguaje de uso general tal como Java o C++, el lenguaje aspectual sin embargo será el que defina el funcionamiento de los aspectos.

El encargado de generar el resultado final es el tejedor o *weaver* el cual convertirá el lenguaje base relativo a cada *concern* en código ejecutable que será combinado apropiadamente mediante las reglas aportadas por el lenguaje aspectual. Cada técnica de AOP puede implementar el tejedor de varias maneras, las más usuales son dos:

- Preprocesamiento del código fuente con los aspectos para producir código fuente tejido que será convertido en código ejecutable por el compilador del lenguaje base.
- Tejiendo a nivel de código de bytes con una metodología similar a la llevada con el código fuente. Es posible en este caso utilizar una implementación de máquina virtual capaz de manejar aspectos (just-in-time aspect weaver).

2.3 AspectJ

AspectJ[4] es una extensión orientada a aspectos de propósito general para JAVA de Xerox PARC. AspectJ usa Java como lenguaje base para la implementación de los concerns aspectuales y especifica extensiones para Java para las reglas de tejido.

El tejedor de AspectJ funciona mediante el preprocesamiento de código creando un sistema constituido por código de bytes de Java, el cual puede ser ejecutado en cualquier JVM[5]. Además también posee herramientas como un depurador y una integración de IDE seleccionada.

Las reglas de tejido de AspectJ están especificadas en términos de pointcuts, joinpoints, advices, declarations y clases aspecto:

- Clase aspecto: es la unidad modular del mismo modo que las clases convencionales lo son en Java. Pueden contener pointcuts, advices y declaraciones y adicionalmente puede contener variables, métodos y clases anidadas como cualquier clase Java.
- Joinpoints: son aquellos puntos o lugares de la ejecución de un programa que pueden ser identificados. Por ejemplo pueden ser las llamadas a métodos o la asignación a un miembro.
- Pointcuts: son selecciones de uno o varios joinpoints de los que pueden obtener adicionalmente sus argumentos.
- Advices: son las piezas de la implementación de un aspecto a ser ejecutadas en los pointcuts. Los advices pueden ser ejecutados antes, después o alrededor de la ejecución de un joinpoint seleccionado por un pointcut.
- Declarations: son declaraciones que permiten:

- La introducción de cambios estáticos en las clases, interfaces o aspectos del sistema como por ejemplo la adición nuevos miembros o métodos (declaraciones de miembros inter-tipo).
- Implementación de interfaces y declaración de herencia en clases.
- Generación de errores y avisos en tiempo de compilación.
- Envolvimiento de excepciones no chequeadas.
- Asignación de precedencias entre aspectos.

2.3.1 Sintaxis

En este apartado se expondrá de manera resumida la sintaxis para AspectJ 1.5 relacionada con el desarrollo realizado en este proyecto.

Clases aspecto

Las clases aspecto son similares a las clases convencionales Java salvo por la principal diferencia de que no pueden ser instanciadas explícitamente sino que lo hacen de forma automática según su cláusula, comúnmente se instancian cuando van a ser utilizados por primera vez.

Forma general

[**privileged**] [*Modificadores*] **aspect** *Id*
[**extends** *Tipo*] [**implements** *Interfaces*] [*PerClause*] { *Cuerpo* }

A diferencia también de las clases convencionales pueden tener el modificador ‘privileged’ el cual designa que el cuerpo de la clase aspecto podrá acceder a los elementos private de otras clases.

Pointcuts

Los pointcuts tienen las características propias de un elemento de java tal como la firma, los modificadores y la capacidad de ser abstracto.

La definición de un pointcut consta de dos partes separadas por el carácter “:”. La primera es la firma y la segunda los designadores.

Definición de un pointcut

[*Modifiers*] **pointcut** *Id* (*Formals*) : *Pointcut* *(designadores)*

Definición de un pointcut abstracto

abstract [*Modifiers*] **pointcut** *Id* (*Formals*) ;

Los argumentos de la firma son aquellos que el pointcut debe obtener del joinpoint como por ejemplo los argumentos de un método invocado o la referencia del objeto invocador u objetivo.

Los designadores tienen como función definir los joinpoints mediante la descripción sintáctica de los puntos de ejecución que han de ser seleccionados así como también los argumentos que deben obtenerse. Los designadores pueden combinarse con la referencia a otros pointcuts

Los designadores de pointcut son los siguientes:

- call (firma): selecciona las llamadas a métodos o constructores.
- execution (firma): selecciona la ejecución de métodos y contructores.
- initialization (firma): selecciona la inicialización de objetos.
- preinitialization (firma): selecciona la preinicialización de objetos dada antes de la llamada al superconstructor.
- staticinitialization (tipo): selecciona la inicialización de un tipo después de ser cargado.
- get (firma): selecciona la lectura de una variable.
- set (firma): selecciona la adjudicación de una variable.
- cflow (pointcut): condiciona que la selección se encuentre en el flujo de ejecución de un pointcut.
- cflowbelow (pointcut): condiciona que la selección se encuentre en el flujo de ejecución de un pointcut sin incluir al mismo.
- within (tipo): condiciona que la selección se encuentre en el cuerpo de una clase determinado.
- withincode (firma): condiciona que la selección se encuentre dentro de un método determinado.
- this (tipo u objeto): condiciona la selección a que el jointpoint sea invocado desde un objeto cuya referencia podrá obtenerse.
- target (tipo u objeto): condiciona la selección a que el joinpoint se encuentre dentro de un objeto o tipo de objeto cuya referencia podrá obtenerse.
- args (tipos u objetos): condiciona la selección a que el joinpoint se encuentre dentro de un objeto o tipo de objeto cuya referencia podrá obtenerse.

- if (condición): condiciona la selección a que la condición se cumpla.
- handler (tipo Exception): selecciona el momento en que una excepción manejada por un bloque catch.

Los símbolos que sirven para condicionar o combinar los designadores son:

- caracteres “!”,“&&” y “||”: determinan negación, ‘and’ y ‘or’.
- carácter “+”: como último carácter en el nombre de un tipo indica que se hace referencia a dicho tipo y a sus subtipos.
- carácter “*”: sirve comodín de tipos o bien como comodín para de cualquier secuencia de caracteres sin incluir “.” para la formación de nombres.
- carácter “..”: sirve como comodín de cualquier secuencia de caracteres que empiecen y terminen por “.”.

Advices

La definición de un advice, al igual que en el caso de los pointcuts, consta de dos partes separadas por el carácter “:”, sin embargo carece de identificador.

Definición de un advice

[**argumento devuelto**] *AdviceSpec* [**throws** *Exception*] : *Pointcut* { *Cuerpo* }

La primera es la especificación del advice cuya firma define los argumentos que debe obtener y a los cuales se les podrá hacer referencia dentro del cuerpo. Así mismo define cuando será ejecutado el cuerpo pudiéndose dar cinco casos distintos:

- after: determina que el cuerpo del advice deberá ejecutarse inmediatamente después de la ejecución del joinpoint independientemente de las circunstancias.
- after returning: determina que el cuerpo del advice deberá ejecutarse inmediatamente después de la ejecución del joinpoint en el caso de que haya transcurrido con normalidad y no haya lanzado ninguna excepción.
- after returning: determina que el cuerpo del advice deberá ejecutarse inmediatamente después de la ejecución del joinpoint en el caso de que lance una excepción.
- before: determina que el cuerpo del advice deberá ejecutarse justo antes de la ejecución del joinpoint.
- around: determina que el cuerpo del advice deberá ejecutarse suplantando la ejecución del joinpoint debiendo devolver un argumento determinado en el caso en el que el joinpoint lo haga y debiendo indicar la propagación de excepciones que

propaga dicho joinpoint. Dentro del cuerpo del advice es posible ejecutar el joinpoint original mediante la invocación del método ‘proceed’.

La segunda parte de su definición consiste en la combinación de referencias de pointcut y designadores de pointcut creados específicamente para el advice. Los argumentos obtenidos en esta parte deben coincidir con los exigidos por la firma.

Clase JoinPoint

La clase JoinPoint provee acceso al estado de un joinpoint y su información estática. La referencia a la instancia del JoinPoint actual en ejecución puede obtenerse mediante el uso del identificador thisJoinPoint dentro del cuerpo de un advice.

Sus métodos principales son:

getArgs

```
Object[] getArgs()
```

Obtiene utilizados en el joinpoint.

getTarget

```
Object getTarget()
```

Obtiene el objeto objetivo del joinpoint.

getThis

```
Object getThis()
```

Obtiene el objeto desde donde se invocó el joinpoint.

Seguridad y Java

3.1 El Modelo de Seguridad de Java

La seguridad fue tenida en cuenta en la creación de Java como uno de los pilares fundamentales de su arquitectura por lo que se encuentra integrada de manera inherente[6].

La máquina virtual de Java (JVM) se encarga de la ejecución de los programas, la cual consiste en la transformación del código de bytes de éstos en código del sistema que es ejecutado. Este proceso es realizado en un entorno seguro proporcionado fundamentalmente por las características del lenguaje Java y por tres componentes principales: el verificador de código de bytes, el cargador de clases y el gestor de seguridad.

El modelo de seguridad de Java ha pasado por tres fases diferenciables en la evolución de la plataforma Java desde el modelo original hasta el modelo actual de Java 2[7]:

- Modelo original de seguridad o 'modelo de sandbox' JDK 1.0:
    - Aplica un entorno muy restringido para la ejecución de código proveniente de la red.
    - Existen dos niveles de acceso: acceso total para el código local y muy restringido para el código remoto.
    - La seguridad es gestionada por el cargador de clases, el verificador de clases y el gestor de seguridad.
    - Los programas remotos apenas pueden ser funcionales.
- Modelo de seguridad JDK 1.1
    - Inclusión del concepto de código firmado que permite que código obtenido del exterior pueda ser ejecutado fuera del sandbox.
    - Inclusión de Herramientas jar y javakey posibilitando un almacén compacto y firma para clases.
    - API de seguridad con paquetes de clases que proporcionan funciones criptográficas permitiendo su uso estandarizado en el desarrollo de aplicaciones. Es extensible e incluye herramientas de firmado digital, resumen, gestión de claves y uso de listas de control de accesos.
- Modelo de seguridad en Java 2

- Control de acceso de grano fino. Dejan de existir los dos niveles de acceso (acceso total y sandbox) y se introduce un sistema de control de permisos flexible. Este control de acceso es ahora aplicado a todo el código, es decir ahora la firma de código es aplicable tanto al código local como al código remoto.
- Ajuste sencillo de los permisos de acceso basado en el fichero de políticas (policy file) en el que se definen los permisos de acceso para el código.
- Inclusión de las herramientas jar (nueva versión), keytool, jarsigner y policytool.
- El API de seguridad se amplía con subpaquetes que mejoran el soporte de certificados X.509 y permiten crear Listas de Revocación de Certificados (CRLs) y Peticiones de Firmado de Certificados (CSRss). java.security.cert y java.security.spec.

La seguridad de la plataforma Java consta de cuatro fronteras primarias[8]: las características de diseño del lenguaje, el verificador de códigos de bytes, el cargador de clases y el gestor de seguridad.

3.1.1 Características de Diseño del Lenguaje Java

Tienen como objetivo evitar errores de memoria y el acceso al sistema operativo subyacente:

- Ausencia de punteros: protección contra la imitación de objetos, la violación de la encapsulación y el acceso incontrolado a áreas de memoria.
- Gestión de memoria: la memoria no se gestiona de manera directa disminuyendo así la interacción del programador con la memoria y evitando las grietas de memoria y los punteros a posiciones vacías. No se reserva memoria si no que se instancian objetos, una vez éstos pasen a ser desechados por la aplicación el recolector de basura libera la memoria ocupada por los objetos.
- Comprobación de rangos en el acceso a vectores: se controla los accesos a los vectores y se lanza una excepción cuando el acceso esta fuera de rango.
- Tipado fuerte: se impiden los castings que no son legítimos para impedir accesos de memoria erróneos.
- Control de acceso a métodos y atributos basado en niveles.
- Existencia del modificador ‘final’ que impide que se puedan redefinir atributos o que se puedan definir subclases de una clase con dicho modificador.

3.1.2 El Verificador de Código de Bytes

Su función es verificar los códigos de bytes de las clases que no se consideran seguras (normalmente las que no son clases del sistema o no están validadas por el usuario). Forma parte de la máquina virtual de Java y lleva a cabo cuatro fases de verificación:

1. Validación del formato del fichero.
2. Comprobación de cláusula final para clases.
3. Verificación del código de bytes.
4. Finalización del proceso de verificación con los últimos tests.

Una vez un fichero es verificado se le atribuyen la garantía de que el código tiene el formato correcto y no viola restricciones de consistencia de tipos y que las pilas internas no puedan desbordarse.

3.1.3 El Cargador de Clases

Su función es separar las clases que carga, definiendo para ello tres grupos de clases dependiendo de su procedencia: las clases del sistema, las clases asociadas a la ruta de extensión y las clases asociadas a la ruta de la aplicación. El procedimiento que sigue para la carga es buscar la clase por el orden anterior hasta dar con ella.

Su funcionalidad reside en la clase ClassLoader la cual abstracta y puede ser extendida para definir cargadores especiales.

3.1.4 El Gestor de Seguridad

Su función es comprobar el acceso en tiempo de ejecución y su funcionalidad reside en la clase SecurityManager la cual puede ser extendida.

3.2 Criptografía

3.2.1 Criptografía

La criptografía es el arte o ciencia de cifrar y descifrar información utilizando diferentes técnicas. En la actualidad existen numerosas metodologías para cubrir las distintas necesidades existentes en el mundo de la computación y la informática.

3.2.1.1 Algoritmos de Cifrado Simétrico

Los algoritmos de cifrado simétrico convierten un texto cualquiera (denominado usualmente texto plano) en un texto cifrado del mismo tamaño que el original utilizando una clave y son empleados normalmente para encriptar grandes cantidades de datos debido

a su coste computacional bajo. Pueden distinguirse principalmente por la cantidad de datos que manejan y por el tipo de cifrado (por bloques o por flujo).

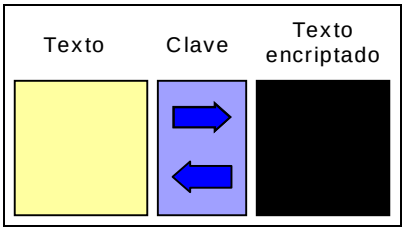

Figura 7 - cifrado simétrico

El cifrado por bloques produce bloques de cifrado de tamaño fijo a partir de bloques de entrada de generalmente el mismo tamaño y el cifrado de flujo de datos produce los datos encriptados a partir del procesamiento de los datos de entrada operando sobre sus bits o bytes con una secuencia cifrante generada.

Algunos algoritmos de cifrado simétrico más utilizados son:

- DES: (Estándar de Encriptación de Datos) o también llamado DEA (Algoritmo de Encriptación de Datos) es un algoritmo de cifrado por bloques que emplea una clave 64 bits de los cuales 56 son efectivos y 8 son de paridad. Hoy en día no se considera un algoritmo de mucha fuerza.
- Triple DES: consiste en la encriptación triple mediante DES. Existen varios tipos de triple DES dependiendo de la secuencia y claves utilizadas.
- AES: (Advanced Encryption Standard) es un algoritmo de cifrado por bloques destinado a ser el sustituto de DES.
- IDEA: Algoritmo iterativo de cifrado de bloques de 64 bits y clave de que se basa en 8 rotaciones complejas 128 bits.
- Blowfish: algoritmo de cifrado por bloques de clave variable.
- RC2: algoritmo de cifrado por bloques de 64 bits con tamaño de clave variable y con un procesamiento mediante software más económico que el de DES.
- RC4: algoritmo de cifrado de flujo a nivel de byte de clave de tamaño variable.
- RC5: algoritmo de cifrado por bloques parametrizable.
- RC6: algoritmo de cifrado por bloques derivado de RC5.

3.2.1.2 Algoritmos de Cifrado Asimétrico

Los algoritmos de cifrado simétrico convierten un texto cualquiera en un texto cifrado del mismo tamaño que el original utilizando dos claves relacionadas matemáticamente: la

clave pública y la clave privada. Su coste computacional es alto y a menudo sirven para intercambiar claves simétricas con las que el cifrado es mucho más económico. Pueden ser distinguirse fundamentalmente tres tipos de algoritmo:

- Algoritmo reversible: el mensaje encriptado con la clave privada puede ser desencriptado con la clave pública y viceversa.
- Algoritmo irreversible: el mensaje encriptado con la clave pública puede ser desencriptado con la clave privada pero no a la inversa.
- Algoritmo de intercambio de claves: permite la negociación segura de una clave secreta entre los dos extremos.

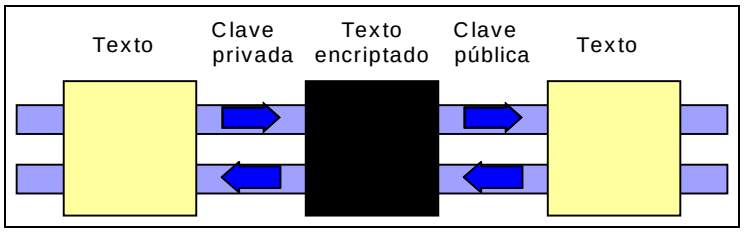

Figura 8 - Cifrado asimétrico

Los algoritmos de clave pública más comunes son RSA y Diffie-Hellman:

- RSA: es un algoritmo de clave variable y de tamaño de bloque ajustable siempre por debajo del tamaño de la clave; a mayor tamaño de clave genera mayor cantidad de texto cifrado y menor es su eficiencia. Es el algoritmo de clave pública más utilizado y puede ser usado para la encriptación de comunicaciones, intercambio de claves y firmas digitales.
- Diffie-Hellman: es un algoritmo orientado principalmente al intercambio de claves secreta compartida entre varios extremos. Puede ser utilizado para encriptar comunicaciones y firmas digitales.

3.2.1.3 Algoritmos de Dispersión o de Resumen

Produce un resumen de longitud fija de un mensaje de longitud variable teniendo en cuenta que ha de ser criptográficamente seguro. Para ello debe de responder a los siguientes criterios:

- El mensaje original no puede ser obtenido a partir del resumen.
- Dos mensajes distintos no pueden producir el mismo resumen.
- Dado un resumen no puede ser obtenido un mensaje generador.

Los algoritmos de dispersión más comunes son:

- MD2, MD4 y MD5: generan un resumen de 128 bits a partir de un mensaje de longitud arbitraria.
- SHA: genera resúmenes de 160 bits a partir de mensajes de tamaño inferior a 2^{64} bits.
- SHA-1: versión mejorada de SHA. Es considerado el sucesor de MD5.

3.2.1.4 Código de autenticación de mensajes

El código de autenticación de mensajes o MAC es un bloque de datos de tamaño fijo que se adjunta a un mensaje para identificar su origen e integridad.

Generalmente suele emplearse generando un MAC a partir del mensaje y un código secreto compartido de manera que el receptor puede verificar el MAC creando otro del mismo modo y comparándolos.

3.2.1.5 Firma digital

La firma digital es similar a un código de autentificación de mensajes con la salvedad de que no requiere de una clave secreta compartida sino que hace uso de un algoritmo de clave pública de manera que puede ser verificado por aquellos que posean la clave pública y producido por el único poseedor de la clave privada.

El proceso consiste en la generación de un resumen del mensaje, el cual es encriptado con la clave privada del emisor. El receptor obtiene otro resumen del mensaje y lo compara al mensaje recibido descifrado con la clave pública del emisor.

El algoritmo explícito para firma digital principal más común es DSA (Algoritmo Estándar de Firmado) el cual emplea claves de 1024 bits y su verificación de firmas es relativamente lenta.

3.2.1.6 Certificados digitales

Son el punto de unión entre la clave pública de una entidad y sus atributos de referencia de manera que se garantice que la clave pública pertenece a dicha entidad la cual ha de poseer la correspondiente clave privada. Para que sean útiles deben de ser validadas por alguna autoridad certificadora la cual ha de haberlo firmado digitalmente como garantía de dicha validación.

Sus principales aplicaciones residen en la autenticación y en la distribución claves públicas.

3.2.2 JCA y JCE

El soporte de criptografía en JAVA se divide en dos bloques JCA Y JCE; el primero aporta las bases del soporte criptográfico de Java y el segundo provee los algoritmos criptográficos. La separación existente entre JCA y JCE se debe principalmente a las leyes de exportación de EEUU para la encriptación que impidieron incluir en las versiones de JDK anteriores a la 1.4 las clases motor de implementación de criptografía de clave simétrica y generación y manipulación de las claves empleadas[9].

JCA (Java Cryptography Architecture) es el marco para acceder y desarrollar la funcionalidad criptográfica de Java e incluye el API relativo a la criptografía y las especificaciones para la creación de extensiones criptográficas por terceros. Su filosofía está basada en la 'arquitectura de proveedores' de manera que los proveedores (que serán identificados en la configuración de seguridad de java o bien añadidos dinámicamente) se encargarán de aportar los algoritmos específicos. Estos algoritmos podrán ser utilizados de manera transparente para el desarrollador respondiendo a los siguientes principios:

- Independencia e interoperabilidad de las implementaciones: a través de la base compuesta por interfaces y clases abstractas que forman la arquitectura de criptografía y que permite que los proveedores puedan desarrollar implementaciones concretas de manera independiente y con la certeza de que serán compatibles con el funcionamiento de otras.
- Independencia y extensibilidad de los algoritmos: existen tipos de servicios criptográficos con sus respectivas clases también llamadas clases motor que pueden ser extendidas para incorporar algoritmos que respondan al mecanismo abstracto que representan, de esta manera nuevos algoritmos pueden ser añadidos de manera independiente.

El concepto de proveedor se define en el JCA mediante la clase Provider del paquete java.security la cual es abstracta y contiene los métodos para acceder al nombre del proveedor, el número de versión y otra información sobre las implementaciones de los algoritmos para la generación, conversión y gestión de claves y la generación de firmas y resúmenes.

Las clases e interfaces principales de JCA son:

- Security: gestiona los proveedores instalados y las propiedades de seguridad.
- MessageDigest: es la clase motor que provee la funcionalidad de la creación de resúmenes criptográficamente seguros de mensajes.
- Signature: es la clase motor que provee la funcionalidad del algoritmo de firma digital.
- AlgorithmParameters: es la clase motor que gestiona los parámetros de un algoritmo incluyendo codificación y decodificación.
- Key: interfaz que representa una clave.

- PublicKey y PrivateKey: son interfaces que derivan de Key y representan una clave pública y una clave privada respectivamente.
- KeySpec: interfaz que representa una especificación de clave.
- KeyFactory: es la clase motor que provee la conversión entre claves y especificaciones de claves.
- KeyPair: es una clase contenedora de un par de claves (pública y privada).
- KeyPairGenerator: es la clase motor que permite la generación de un par de claves pública y privada.
- KeyStore: es la clase motor que funciona como interfaz para acceder y modificar los almacenes de claves.
- SecureRandom: es la clase motor que provee la funcionalidad de un generador de números aleatorios seguro.

JCE (Java Cryptography Extensión) provee implementaciones para encriptación, generación de clave, acuerdo de claves y algoritmos MAC. El soporte para encriptación incluye cifrado simétrico, asimétrico, por bloques y por flujo.

El uso de JCE por tanto conlleva usar un grupo consistente de clases básicas para realizar encriptación y desencriptación de las cuales la principal es la clase Cipher que encapsula la funcionalidad del cifrado criptográfico. Un objeto Cipher es creado para un tipo específico de transformación y puede singularmente encriptar datos, desencriptar datos, envolver claves o desenvolver claves.

Las clases e interfaces principales de JCE son:

- Cipher: representa un cifrador para encriptación y desencriptación.
- CipherInputStream y CipherOutputStream: proporcionan la funcionalidad de ‘streams’ de encriptación de entrada y salida respectivamente mediante el uso de un objeto Cipher.
- SecretKey: interfaz que representa una clave simétrica.
- KeyGenerator: es la clase motor que permite la generación de claves para algoritmos simétricos.
- SecretKeyFactory: es la clase motor que tiene la funcionalidad de factoría de claves simétricas.
- KeyAgreement: proporciona la funcionalidad de un protocolo de acuerdo de claves.
- Mac: proporciona la funcionalidad de autenticación de mensajes.
- SealedObject: es la clase motor que permite crear objetos encriptados.

3.3 Sockets Seguros

3.3.1 SSL y TLS

El protocolo SSL o Secure Socket Layer fue originalmente diseñado para establecer comunicaciones seguras para protocolos como HTTP y FTP, TLS o Transport Security Layer es la cuarta versión y más actual de éste protocolo; en este proyecto será referenciado indistintamente como SSL.

SSL es un protocolo que permite conexiones seguras y autenticadas basándose en la criptografía y atendiendo a los siguientes parámetros:

- Interoperabilidad: los extremos deben de poder interoperar de manera transparente.
- Extensibilidad: el protocolo debe de ser capaz de incorporar nuevos algoritmos criptográficos sin que suponga un cambio en el mismo.
- Eficiencia: debe de considerar el coste de procesamiento de los algoritmos criptográficos del mismo modo que su fortaleza.

Su funcionalidad reside fundamentalmente en una comunicación cifrada mediante una clave simétrica acordada y autenticación de las partes mediante claves públicas y certificados digitales. Esto supone que la conexión es privada puesto que los datos son encriptados con una clave que solo pueden conocer las partes de la comunicación y fiable puesto que además esta comunicación incluye verificación de integridad.

El protocolo SSL está a su vez constituido por varios protocolos:

- Protocolo de registro: se encarga de la transmisión y recepción de datos y consta de varias capas en cada cual se incluyen los campos de tamaño, descripción y contenido. El proceso en el emisor sigue los siguientes pasos:

  1- División del mensaje en bloques.
  2- Compresión opcional de los datos.
  3- Encriptación de los datos.
  4- Adición de un MAC
  5- Transmisión

  El proceso en el receptor es el inverso al del transmisor.

- Protocolo de mutuo acuerdo o ‘handshake’: opera sobre el protocolo de registro y se encarga del acuerdo de la autenticación, el acuerdo de parámetros de seguridad y la gestión de errores en la comunicación. Este proceso está constituido por las siguientes fases:

    - Intercambio de mensajes de inicialización para el acuerdo sobre los algoritmos y versión del protocolo a emplear, intercambio de valores aleatorios y verificación de sesión reanudad.
    - Intercambio opcional de certificados de autentificación.
    - Generación de la clave secreta.
    - Aportación de los parámetros de seguridad a la capa de registro.
    - Verificación de que el proceso transcurrió correctamente en los extremos.

- Protocolo de datos de aplicación: se encarga de la fragmentación compresión y cifrado de los datos de manera que los mensajes sean manejados de manera transparente por el protocolo de registro.
- Protocolo de cambio de parámetros criptográficos: gestiona la transición de los parámetros criptográficos utilizados en la conexión.
- Protocolo de alerta: gestiona la existencia de errores o cierres inesperados en la conexión.

3.3.2 JSSE

JSSE es un API de Java que permite conexiones seguras con SSL sobre TCP/IP, este API adjunta los servicios principales de red y criptografía suministrando clases socket extendidas, gestores de certificados, clases contexto SSL y un entorno de trabajo de factorías de socket para encapsular la funcionalidad de creación de sockets. De este modo provee un esqueleto y una implementación para una versión de Java de los protocolos SSL y TLS e incluye funcionalidades para encriptación de datos, autenticación de servidor o de cliente e integridad de mensajes.

Del mismo modo que JCE su diseño se basa en la arquitectura de proveedores de manera que aporta independencia de implementación y cuando es posible independencia de algoritmo. Sus principales aportaciones son:

- Soporte de API para SSL 2.0 y 3.0 y una implementación de SSL 3.0.
- Soporte de API y una implementación para TLS 1.0.
- Clases que pueden ser instanciadas para crear canales seguros.
- Soporte para la negociación de suites de cifrado como parte del proceso handshake de SSL utilizado para inicializar o verificar las comunicaciones seguras.

- Soporte para autenticación de cliente y servidor como parte del proceso handshake de SSL.
- Soporte para HTTP encapsulado en el protocolo SSL (HTTPS).
- APIs de gestión de sesión de servidor para gestionar sesiones SSL existentes.

El API provee soporte para varios algoritmos criptográficos comúnmente usados en suites de cifrado:

Algoritmo	**Proceso criptográfico**	**Longitud de la clave**
RSA	Autenticación e intercambio de claves	2048 (autenticación) 2048 (intercambio de claves) 512 (intercambio de claves)
RC4	Envolvimiento	128 128 (40 efectivos)
DES	Envolvimiento	64 (56 efectivos) 64 (40 efectivos)
Triple DES	Envolvimiento	192 (112 efectivos)
AES	Envolvimiento	256 128
Diffie-Hellman	Envolvimiento	1024 512
DSA	Autenticación	1024

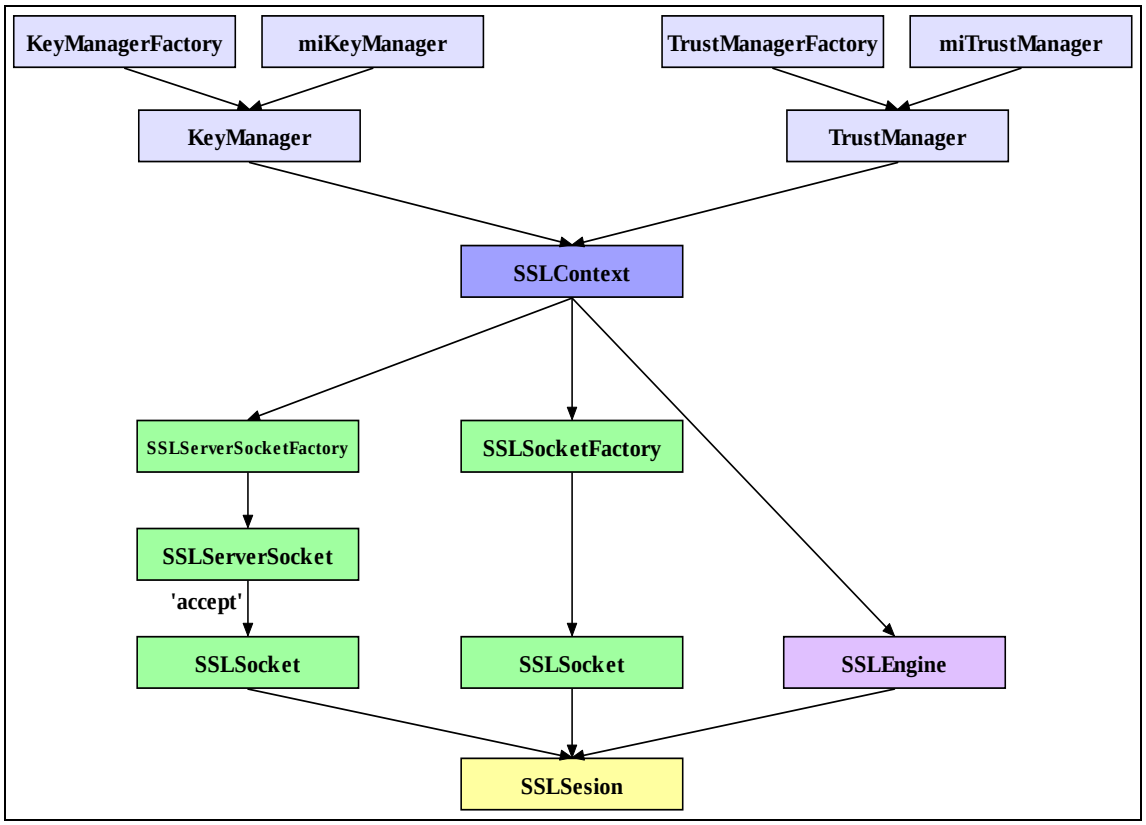

Figura 9 - Esquema de clases de JSSE

Las clases e interfaces principales del API son:

- SSLSocket: es una subclase de Socket cuya funcionalidad es la de un socket seguro. Soporta todos los métodos de la clase Socket y añade métodos adicionales relacionados con la seguridad.
- SSLSocketFactory y SSLServerSocketFactory: actúan como factorías para la creación de sockets seguros (SSLSocket) y servidores de sockets seguros (SSLServerSocket) respectivamente.
- SSLContext: es la clase motor para una implementación del protocolo de sockets seguro cuya instancia actúa como factoría de objetos SSLSocketFactory y SSLEngine.
- TrustManager: se encarga de determinar si se debe confiar en ciertos credenciales de autenticación.
- TrustManagerFactory: es la clase motor que actúa como factoría de uno o más tipos de objetos TrustManager.
- KeyManager: se encarga de seleccionar los credenciales de autenticación que serán enviados eventualmente al otro extremo de la comunicación.
- KeyManagerFactory: es la clase motor que actúa como factoría de uno más tipos de objetos KeyManager.

- SSLSesion: representa un contexto de seguridad negociado entre las dos partes de la conexión.
- SSLEngine: encapsula una máquina de estados SSL/TLS que opera con los buffers de bytes de entrada y de salida con el fin de que pueda existir un control explícito de la comunicación por parte de aplicación.

3.4 Autenticación

Dentro del contexto de la tecnología autenticación es el proceso mediante el cual una entidad que interfiere con un sistema garantiza su identidad.

3.4.1 JAAS

El paquete JAAS (Java Authentication and Authorization Services) facilita un API que funciona dentro de la arquitectura de seguridad de Java para aportar servicios de autenticación y autorización; autenticación para determinar con seguridad quien esta ejecutando el código y autorización para estar seguros de que quien lo hace tiene los permisos correctos para realizar las acciones.

JAAS esta diseñado para ser 'conectable' de manera que las partes de una implementación de seguridad basada en este API pueden ser intercambiadas sin requerir cambios en el código, de este modo los implementadores pueden aplicar cambios a las aplicaciones de manera transparente.

Usando las interfaces del paquete JAAS pueden crearse componentes para validar los intentos de autenticación o para interactuar con el usuario para obtener los datos de dicho intento, estos componentes pueden ser integrados dentro de una aplicación simplemente cambiando el archivo de configuración JAAS.

Las clases comunes de JAAS para autorización y autenticación son:

- Subject: representa la entidad fuente de una petición. Una vez es autenticado se rellena con varias identidades asociadas (objetos Principal).
- Principal: representa una identidad de una entidad.

Las clases e interfaces principales de JAAS relacionadas con la autenticación son:

- LoginContext: es la clase que se encarga de gestionar el proceso de autenticación.
- LoginModule: se encarga de validar los datos en un proceso de autenticación.
- CallbackHandler: se encarga de la interacción con el usuario para obtener los datos de autenticación.

- Callback: representa uno de los datos comunicados por el usuario en el proceso de autenticación (nombre, password, etc).

El algoritmo básico de autenticación con JAAS consta de los siguientes pasos:

1. Creación de una instancia de LoginContext, uno o más LoginModule son cargados basándose en el archivo de configuración de JAAS.
2. La instanciación de cada LoginModule es opcionalmente proveído con un CallbackHandler que gestionará el proceso de comunicación con el usuario para obtener los datos con los que éste tratará de autenticarse.
3. Invocación del método 'login' del LoginContext el cual invocará el método 'login' del LoginModule
4. Los datos del usuario son obtenidos por medio del uso del CallbackHandler.
5. El LoginModule evalúa los datos introducidos por el usuario y resuelve su validación. Si dicha validación tiene éxito se insertan los Principals adecuados al Subject que representa a la entidad autenticada.

La autorización se basa en la concesión de permisos de control de acceso basándose en qué código esta siendo ejecutado y quién lo ejecuta. Para que la autorización tenga lugar deben de darse las siguientes circunstancias:

- El usuario debe de estar autenticado
- El 'Subject' autenticado debe de estar asociado con un contexto de control.
- Los 'Principals' del 'Subject' deben de estar configurados en la póliza de seguridad de Java.

Las clases principales de JAAS relacionadas con la autorización son:

- Policy: representa la póliza de control de acceso.
- AuthPermission: encapsula los permisos básicos requeridos para JAAS.
- PrivateCredentialPermission: protege el acceso a las credenciales privadas de un Subject.

3.5 Registro de Sucesos Críticos

El registro de sucesos críticos en el contexto del software consiste fundamentalmente en la obtención, clasificación y almacenamiento de la información de los sucesos importantes que transcurren en una aplicación en ejecución.

No se trata de una funcionalidad que suela entenderse directamente integrada en el grupo de la seguridad debido a que es incapaz de solucionar problemas en tiempo real sin embargo es un elemento imprescindible para el mantenimiento y supervisión de la seguridad a largo plazo debido a que permite conocer las anomalías producidas tanto por las deficiencias del sistema como por la acción de un individuo malintencionado.

3.5.1 Logging

Logging es un API de java orientado a facilitar el registro de sucesos de las aplicaciones que permite controlar su funcionamiento tanto a través del código como a través de la configuración externa.

Las clases principales del API son:

- Logger: es la entidad principal en la cual las aplicaciones realizan invocaciones de registro. Un objeto Logger suele ser utilizado para registrar mensajes para un sistema específico o componente de aplicación.

- LogRecord: es utilizado para pasar peticiones de registro entre el framework de registro y los manejadores de registro individuales.

- Handler: exporta los objetos LogRecord a distintos destinos como pueden ser la memoria, streams de salida, consolas, archivos y sockets dependiendo de cada subclase de Handler la cuales pueden ser implementadas por terceros. El paquete incluye las siguientes extensiones de Handler:

    - StreamHandler: escribe las entradas de registro formateadas en un OutputStream.
    - ConsoleHandler: escribe las entradas de registro formateadas en System.err.
    - FileHandler: escribe las entradas de registro formateadas en un fichero o en un conjunto rotacional de ficheros.
    - SocketHandler: escribe las entradas de registro formateadas estableciendo una conexión TCP con un puerto remoto.
    - MemoryHandler: almacena las entradas de registro en memoria.

- Level: define un grupo de niveles de registro standard que pueden ser usados para controlar la salida de registro. Las aplicaciones pueden ser configuradas para registrar unos niveles mientras que otros se ignoren mediante la configuración externa.

- Filter: provee control de acceso de ‘grano fino’ sobre lo que debe de ser registrado después del control de niveles. El API provee un mecanismo de filtro de propósito

general que permite al código de la aplicación adosar filtros arbitrarios al control de salida de registro.

- Formatter: encapsula la funcionalidad de formateo de los datos registrados. El paquete java.util.logging incluye dos extensiones de Formatter, SimpleFormatter y XMLFormatter para formato en texto plano y en XML respectivamente.

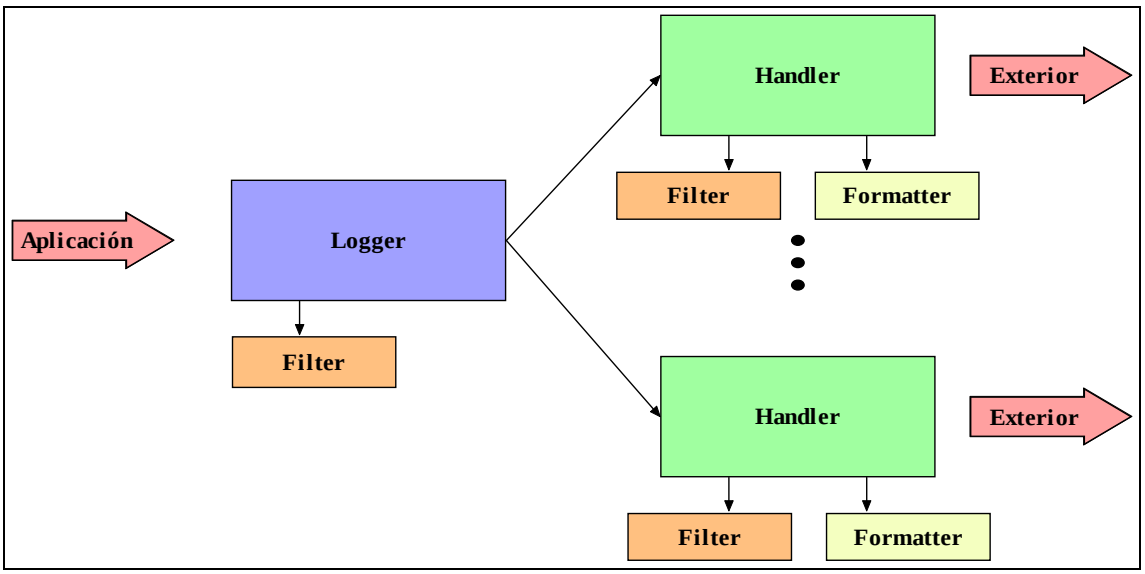

Figura 10 - Esquema de funcionamiento de Logging

Su funcionalidad consiste en las llamadas de registro que el código realiza a objetos Logger los cuales destinan objetos LogRecord que son pasados a uno o varios objetos Handler para la publicación. Dichos objetos Logger y Handler pueden usar niveles de registro y filtros (Level y Filter) para la decisión sobre cada LogRecord. Cuando un logRecord ha de ser publicado externamente cada objeto Handler puede opcionalmente usar un objeto Formatter para localizar y formatear el mensaje.

Paquete de seguridad en Java y AspectJ

4.1 Diseño

4.1.1 El Aspecto Seguridad

La implementación de las funcionalidades o *concerns* relativos a la seguridad en sistemas de media a alta complejidad supone una tarea de enorme problemática debido a los dos problemas inherentes a las técnicas de desarrollo actuales: la falta de modularización para *concerns* transversales y la tiranía del diseño inicial del sistema. Mientras la madurez de las técnicas basadas en el paradigma OOP ha permitido con el tiempo modularizar la implementación de los mecanismos de seguridad, la encapsulación de el 'donde' y el 'cuando' deben funcionar dichos mecanismos sigue siendo un problema que el paradigma no puede afrontar. Así mismo la inclusión de la seguridad en la fase inicial del desarrollo tiene como consecuencia una aplicación notablemente más compleja y por tanto cara, teniendo además que tener en cuenta que aún en los sistemas bien dotados inicialmente pueden aparecer nuevos requerimientos de seguridad no anticipados debido a cambios en el entorno de funcionamiento.

La principal causa de la intrusividad de la seguridad es la diferencia estructural entre la lógica de aplicación y la lógica de seguridad. Esta naturaleza transversal o intrusiva de la seguridad implica no solo a la diversidad de lugares específicos donde aplicar mecanismos de seguridad si no también a la importancia del contexto en el que se ejecuta esos mecanismos. Esto unido al hecho de que la seguridad abarque un muy amplio espectro de funciones hace que se pueda entender a ésta como un concern especialmente intrusivo, lo cual desemboca no solo en una reducción de la calidad del resto del código sino también en una mayor dificultad para detectar agujeros de seguridad[11].

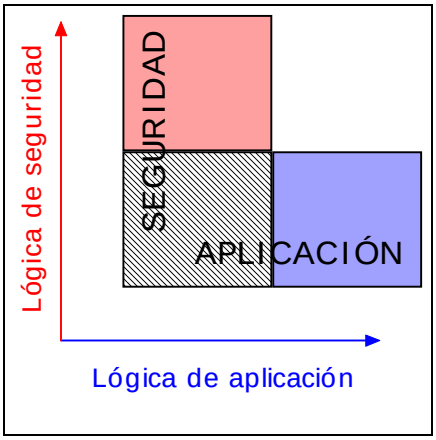

Figura 11 - Transversalidad de la seguridad

Por otro lado la implementación de seguridad en sistemas de poca complejidad puede suponer en muchas ocasiones una carga de trabajo desproporcionada en relación al resto del desarrollo debido al propio peso de la implementación de dicho concern y a la necesidad de conocimientos especializados por parte del desarrollador o el pequeño grupo de desarrolladores.

Debido a estas circunstancias la seguridad representa un desafío para la evolución de las técnicas de implementación actuales habiéndose convertido en un clásico y cada vez más popular ejemplo de concern transversal y por tanto en uno de los candidatos más legítimos para la validación de las técnicas AOP.

4.1.2 Patrón de Reusabilidad

Los requisitos fundamentales que se pretenden satisfacer en la implementación del prototipo expuesto en este proyecto son principalmente una modularización carente absolutamente de intrusividad y una capacidad de reutilización total y transparente para el desarrollador.

La utilización de AspectJ para dicha implementación implicará de manera directa la modularización del código relativo a la seguridad que con las técnicas basadas en OOP resultaría mezclado y dispersado por el resto del código, sin embargo la disponibilidad del código para su reutilización o su evolución sería relativamente mala en comparación con lo posible y deseable, limitándose a ser características derivadas únicamente de dicha modularización implícita[9]; por lo general la reutilización se basa en el reaprovechamiento de métodos motor de superclases abstractas sin hacer lo propio con el motor implementado dentro de advices.

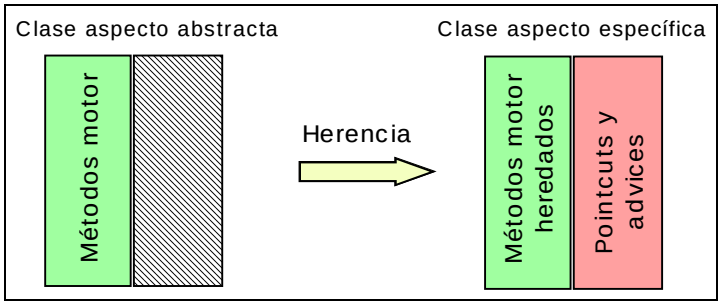

Figura 12 - Reutilización común en AspectJ

Para potenciar la reusabilidad del código de seguridad implementado se ha desarrollado para este proyecto una filosofía de *redefinición* adecuada para lenguajes de estructura similar a AspectJ. Con esta metodología el código aspectual residirá en clases aspecto abstractas que podrán ser heredadas por otras clases aspecto específicas, las cuales redefinirán una parte de los elementos heredados para poder adaptarse a la función específica requerida

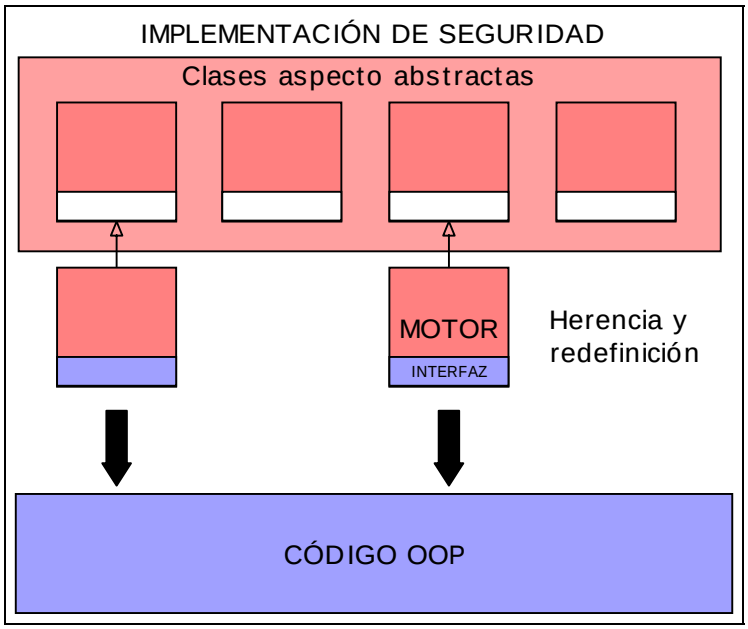

Figura 13 - Filosofía de redefinición

La filosofía de redefinición aplicada se basa fundamentalmente en la división del código de los aspectos en dos partes fundamentales: la parte redefinible llamada *interfaz de reutilización* y la parte de funcionalidad principal del aspecto llamada *núcleo de ejecución*.

La interfaz de reutilización esta constituida por los elementos de las clases aspectos abstractas capaces de o bien definir los puntos donde debe darse el comportamiento aspectual o bien encapsular funciones útiles para el implementador. Estos elementos pueden estar definidos por defecto y en su redefinición pueden bien volver a ser definidos o bien ser anulados en el caso de no desearse su interacción.

Los elementos de la interfaz de reutilización son:

- Pointcuts redefinibles: son los elementos redefinibles de mayor importancia, al redefinirse en las subclases deben capturar los puntos de unión del código OOP concreta al que se añadirá el *concern*.
- Variables reasignables: pueden contener parámetros como el nombre de un algoritmo que ha de ser utilizado o la ruta de un fichero. La reasignación de estas variables puede ser vista como la configuración del funcionamiento del aspecto y se realizará de manera dinámica dentro del método 'init' de cada clase aspecto, el cual será invocado en el constructor.
- Variables reutilizables: contienen referencias a objetos funcionales para el implementador cuya inicialización se supone llevada acabo por la propia inicialización de la clase aspecto.
- Métodos redefinibles: pueden ser redefinidos para llevar a cabo algún proceso que solo puede definirse adecuadamente en el caso específico de aplicación como por ejemplo una respuesta específica a una anomalía. El código del núcleo de ejecución es el encargado de invocarlos.

- Métodos reutilizables: permiten que las subclases puedan utilizar la funcionalidad que encapsulan.

Es necesario denotar que las variables reasignables y reutilizables pueden ser encapsuladas dentro de métodos con el fin de conseguir un uso más formal, sin embargo en este no se tendrá en cuenta con el fin de simplificar la metodología.

El núcleo de ejecución esta constituido por aquellos componentes del código de los aspectos que debe de ser invisible al implementador de las subclases y que a su vez constituyen el motor de la funcionalidad que aporta el aspecto, estos componentes son el constructor, los advices y el resto elementos privados de las clases aspecto. Los advices son la parte fundamental del núcleo de ejecución y deberán ser capaces de autoadaptar el funcionamiento que aportan a las distintas posibilidades que podrían capturar los pointcuts de la interfaz de reutilización.

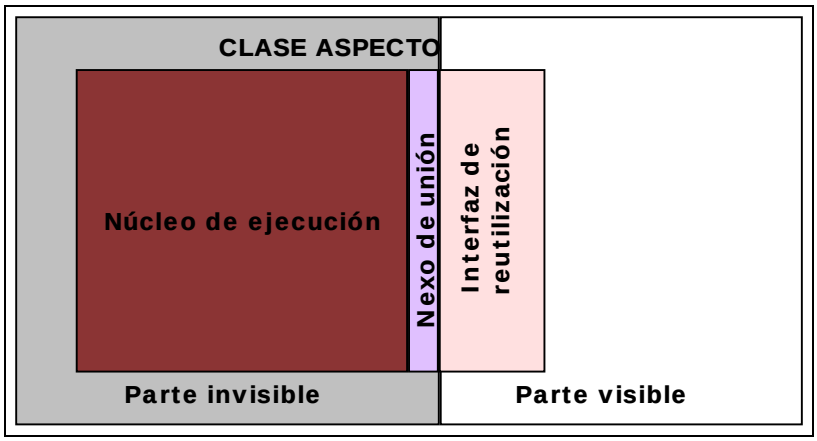

Figura 14 - Esquema de una clase aspecto con el patrón de reutilización

El *nexo de unión* mediante el cual ambas partes se enlazan consiste principalmente en la asociación de los advices del núcleo de ejecución con los pointcuts redefinibles de la interfaz de reutilización. Como se expone en el apartado 2.3.2 los pointcuts están formados por una firma que especifica los argumentos a obtener y por una captura de argumentos que responde a la firma, de manera que encajen con la firma esperada por el advice; es aquí donde surgen los entresijos más críticos debido a que AspectJ no fue diseñado para tener en cuenta la redefinición como una cuestión primaria. En consecuencia es necesario diferenciar en dos tipos de asociaciones: asociaciones con firma indeterminada y asociaciones con firma determinada.

Cuando la firma de un pointcut redefinible debe de ser indeterminada o no especifica argumentos, éstos en el caso de ser necesarios pueden ser obtenidos por los advices mediante el método ‘getArgs’ de JoinPoint. Un pointcut con una firma sin argumentos supone por tanto que la parte de advices es suficientemente flexible como para soportar las distintas posibilidades de captura de los pointcuts en el momento de redefinición de estos. Cuando las asociaciones pointcuts-advices deban diseñarse para capturar joinpoints pertenecientes a elementos de código conocido de antemano como podría ser el API de Java, puede obtenerse flexibilidad total al ser viable conocer todas las coyunturas posibles y tener el lenguaje la posibilidad de acceder a los argumentos.

En el caso de que las asociaciones pointcuts-advices deban diseñarse para capturar joinpoints relacionados con código desconocido a priori como puede ser el perteneciente a una futura aplicación, las posibilidades a tener en cuenta divergen y por tanto la flexibidad disminuye; por ejemplo la captura de un método podría significar tener en cuenta todas sus posibles firmas o lo que es lo mismo todos los números posibles de argumentos, y todos los tipos posibles de dichos argumentos y de su argumento devuelto, por lo que es necesario determinar la firma.

Cuando la firma de un pointcut redefinible debe de ser determinada o especifica argumentos concretos a capturar se supone que los advices están limitados a funcionar con estos argumentos debido a que el lenguaje no aporta la suficiente flexibilidad para la asociación de pointcuts con advices del tipo 'around' que usan el método 'proceed' (el cual permite continuar la acción original con nuevos argumentos) ya que para ello es necesario que los argumentos y excepciones de la firma del pointcut sean los mismos que los definidos en el joinpoint original.

Esto significaría que en el caso de ser factible preparar los advices para una gran variedad de casos posibles en el caso de que estos fueran de tipo around debería de construirse uno para cada firma posible. En este caso por tanto debe imponerse una firma concreta en los pointcuts y en el caso de que el código objetivo no se ajuste a ella (si hubiera sido desconocido en el momento de la implementación) es posible que sea necesario tomar medidas indirectas como reformar los joinpoints necesarios del código objetivo o crear puentes entre firmas.

En cualquiera de los casos expuestos es posible que o bien los designadores de los pointcuts no se definan (pointcut abstracto) o bien se definan por defecto (pudiendo ser anulados o redefinidos al ser heredados).

4.1.3 Guía de Implementación y Utilización

La implementación de un paquete de código AOP mediante el patrón de reutilización expuesto ha de seguir por lo general los siguientes pasos:

1. Definición de los concerns (aspectos) y de los requerimientos internos del paquete (gestión de excepciones etc).
2. Creación de distintas clases aspecto abstractas funcionales que encapsulen los concerns anteriores así como de las clases necesarias para la herencia interna, control de excepciones etc.
3. Definición de la interfaz de reutilización de las clases aspecto funcionales mediante la definición de variables y métodos públicos (o protected) y la generación de las firmas de los advices y pointcuts públicos del nexo de unión, pudiendo diferenciar en ello:

    - Firmas indeterminadas para el caso de la captura de elementos conocidos a priori donde la parte de advices se encargará de soportar un conjunto

limitado de posibilidades (joinpoints) que puede contener el pointcut de argumentos indefinidos.

- Firmas determinadas en advices y joinpoints para el caso de la captura de elementos desconocidos a priori en donde se impondrá la utilización de dicha firma para la aplicación del paquete.

4. Elaboración del núcleo de ejecución las clases aspecto funcionales mediante la implementación del cuerpo de los advices y los métodos privados así como también de las clases OOP necesarias para éstos.

Así mismo será necesaria la generación de la especificación del paquete incluyendo información detallada acerca de la función de los elementos de la interfaz de redefinición. La utilización de un paquete generado de la manera expuesta requerirá que los desarrolladores que vayan a hacer uso de él conozcan dicha especificación de modo que no necesiten conocer acerca del núcleo de ejecución.

La aplicación de un paquete implementado del modo expuesto constará de los siguientes pasos:

1. Análisis de las necesidades funcionales del código objetivo.
2. Generación de las clases aspecto específicas que heredarán de las clases aspecto funcionales que sean necesarias. Podrán crearse varias subclases de la misma clase aspecto funcional.
3. Redefinición de los elementos requeridos de cada una de las interfaces de reutilización de las clases aspecto específicas.
4. Extensión de la funcionalidad adquirida en los casos en los que sea necesario mediante la ampliación del código aspectual de las clases aspecto específicas o la creación de nuevas subclases.
5. Adición de las clases aspecto específicas a la lista de compilación.

4.1.4 Implementación del Paquete de Seguridad

Se dan dos posiciones extremas posibles a la hora de escoger la orientación de la aplicación de un paquete destinado a ser reutilizado. Una es el desarrollo orientado a una metodología concreta de desarrollo de software, como por ejemplo la utilizada por un grupo desarrolladores, de modo que este código reutilizable pueda ser más específico y por tanto eficiente para esa metodología pero generalmente poco eficiente para cualquier otra. La otra alternativa es la orientación a una filosofía de seguridad más genérica de manera que el código pueda ser menos eficiente o incompleto pero suponer una solución práctica y escalable para desarrollos en los que los requerimientos de seguridad no sean muy exigentes o bien contemplen la posibilidad de extender las funcionalidades genéricas aportadas para conseguir una aplicación más específica o compleja. La implementación

realizada en este proyecto se encuentra a medio camino de estas dos posiciones si bien más cerca de una posición genérica.

El prototipo diseñado para este proyecto consiste en un paquete implementado en Java y AspectJ constituido por clases convencionales y clases aspecto. La parte funcional de este paquete residirá en clases aspecto abstractas que podrán ser heredadas por los implementadores para redefinir aquellos elementos pertenecientes a sus interfaces de redefinición, donde cada una de esas clases aspectos encapsulará uno de los aspectos de seguridad que constituyen la funcionalidad total de la implementación. El paquete consiste así mismo en una arquitectura íntegra capaz de contener funcionalidades compartidas por sus elementos tales como la estructura de gestión y manejo de excepciones.

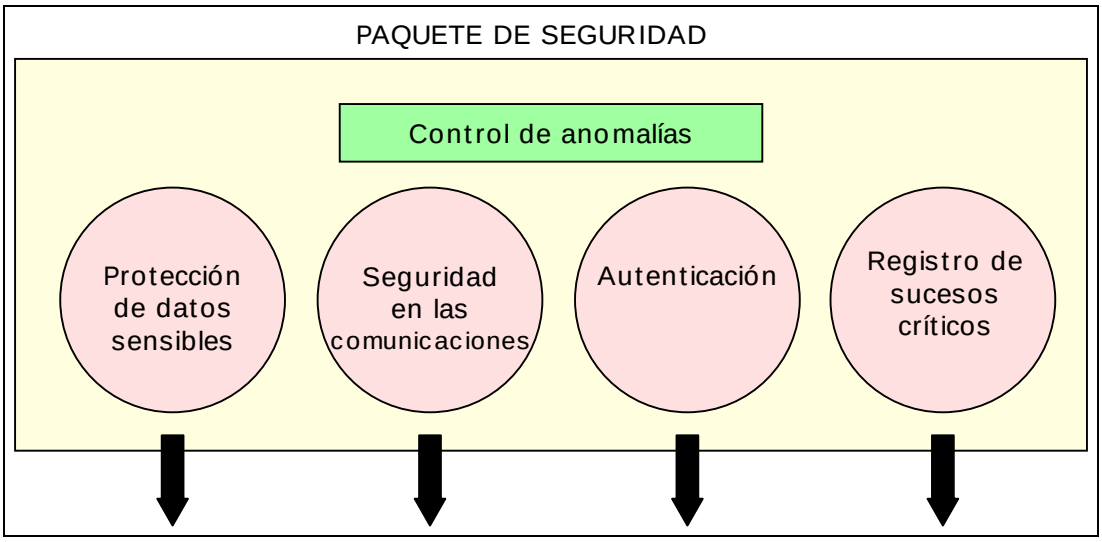

Figura 15 - Paquete genérico de seguridad aspectual

Los aspectos que aporta el paquete se corresponden con los cuatro requerimientos de seguridad más representativos del campo de la seguridad:

- Protección de datos sensibles: abarca el procesamiento basado en encriptación de los datos que viajan hacia o desde una región insegura como por ejemplo una base de datos o un fichero local.
- Seguridad en las comunicaciones: abarca la seguridad en las comunicaciones establecidas por conexiones de sockets mediante código.
- Autenticación: abarca el control sobre la identidad de las entidades que manejan o interfieren en una aplicación.
- Registro de sucesos críticos: abarca el registro de sucesos críticos que ocurren en la ejecución de las aplicaciones.

Estos aspectos han sido implementados de manera prototipal aportando los requerimientos más representativos del espectro particular que cada uno abarca.

4.2 Morfología

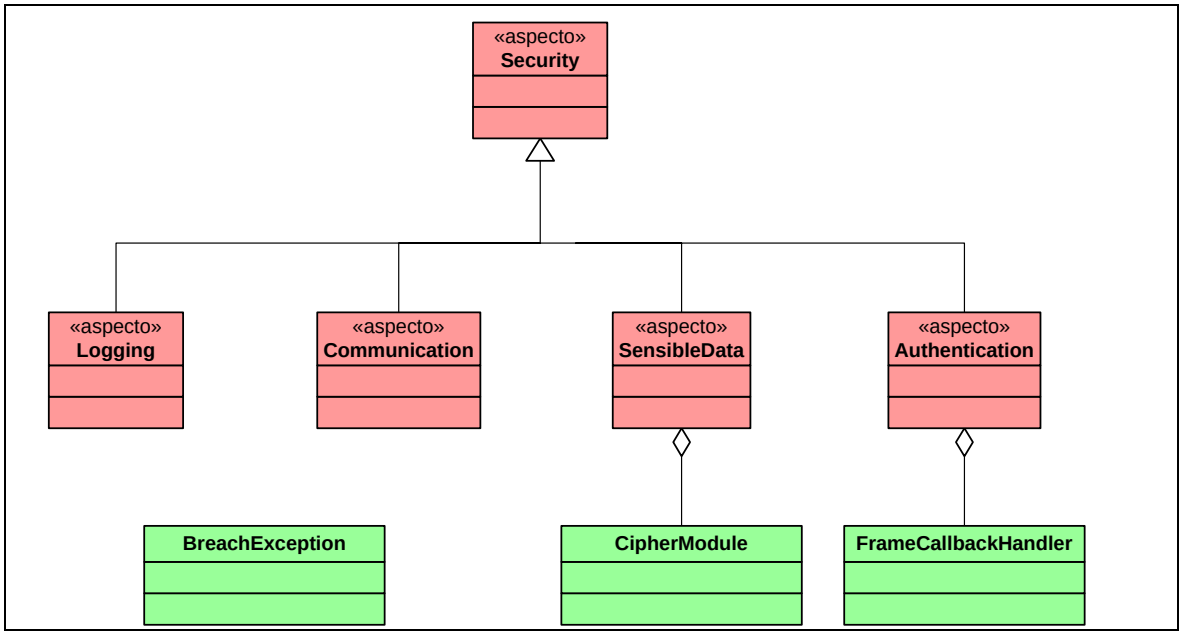

Figura 16 - Esquema del paquete de seguridad

El paquete de seguridad implementado consiste en un sistema formado por un conjunto dependiente de clases aspecto abstractas y clases convencionales cuya parte funcional consiste en la extensión de dichas clases aspecto en clases aspecto específicas para un código objetivo determinado.

Las clases aspecto funcionales son SensibleData, Communication, Authentication y Logging y encapsulan respectivamente cada uno de los aspectos de seguridad que aporta el paquete: protección de datos sensibles, seguridad en las comunicaciones, autenticación y registro de sucesos. Estas clases aspecto descienden de la clase aspecto Security y son abstractas.

Las clases convencionales son BreachException, CipherModule y FrameCallbackHandler y representan funcionalidades comunes de la programación orientada a objetos que son necesarias para las clases aspecto.

Cuando la ejecución de un proceso incluido en el paquete de seguridad detecta una anomalía relativa a alguna circunstancia de seguridad ésta es transmitida en forma de excepción de modo que pueda propagarse a través de la pila de llamadas para que el proceso ‘raíz’ pueda hacerse cargo de ella; este tipo de anomalías son representadas por la clase BreachException.

Cuando una excepción BreachException es generada queda configurada con el tipo de anomalía que originó el problema clasificándola en tres niveles posibles:

- Posible transgresión de seguridad: indica que la anomalía pudo ser originada por alguna acción externa de carácter malicioso, como por ejemplo la alteración de un dato crítico en una base de datos.
- Fallo del sistema: indica que la anomalía pudo ser debida a un fallo del sistema de seguridad por ejemplo debido a una mala configuración o a un fallo relativo a un almacén de claves.
- Anomalía desconocida: indica aquellas anomalías que pudieran tener una naturaleza distinta a las anteriores o bien desconocida.

Una vez el proceso 'raíz' toma cuenta de la anomalía puede notificarla gracias a los métodos heredados de la clase aspecto Security.

4.2.1 Clase Aspecto Security

Es la clase padre de todas las clases aspecto del paquete a las cuales aporta el método 'init' para la reasignación variables y los métodos que hacen posible la notificación de anomalías al usuario.

4.2.1.1 Métodos

init

```
protected void init()
```

Es invocado en las subclases de Security al comienzo de la ejecución de su constructor. Su funcionalidad principal que las subclases de dichas subclases puedan reasignar las variables reasignables de éstas dentro de la redefinición de ese método.

noticeBreach

```
protected static void noticeBreach(JoinPoint joinPoint, BreachException bex,
                    String message)
```

Es invocado tras producirse una anomalía de seguridad y cumple dos funciones fundamentales:

- Notifica al administrador que se produjo una anomalía en la seguridad del sistema donde la variable NOTICE_BREACHS de cada aspecto condicionará si dicha notificación será producida finalmente.

- Sirve como punto de corte para el registro de anomalías por parte de las clases aspecto herederas de Logging, es decir la filosofía de registro de anomalías se basa en que estas son definidas en el momento de la invocación del método en las clases aspecto del paquete.

getStackTrace

```
protected static String getStackTrace()
```

getJoinPointInfo

```
protected static String getJoinPointInfo(JoinPoint joinPoint)
```

getArgsString

```
protected static String getArgsString(Object[] argsObject)
```

Los métodos 'getStackTrace', 'getJoinPointInfo' y 'getArgsString' son utilizados para obtener cadenas de texto explicativas con propósito informativo, sus funciones particulares son:

- getStackTrace devuelve una cadena con la pila de llamadas actual.
- getJoinPointInfo de devuelve una cadena informativa sobre un objeto JoinPoint.
- getArgsString devuelve una cadena con un extracto de un array de objetos (los cuales serán argumentos de un punto de corte).

4.2.2 Clase BreachException

Es la clase derivada de Exception que representa una excepción basada en una anomalía de seguridad producida en el paquete de seguridad.

En el momento de su instanciación ha de ser proveída con el tipo de anomalía que se prudujo entre los tipos UNKNOWN, TRANSGRESSION Y SYSTEM. Así mismo puede ser proveída por el mensaje con un mensaje descriptivo de la causa y un objeto Throwable.

Dispone el siguiente método para dar a conocer el tipo de anomalía que encapsula:

getKind

```
public int getKind()
```

4.3 Aspecto de Protección de Datos Sensibles

La función del aspecto de protección de datos sensibles consiste en blindar y desblindar los datos críticos utilizados en tiempo de ejecución entendiendo el sistema como una región segura y una región insegura (por ejemplo una base de datos) donde en ésta última los datos han de estar blindados. Para el blindado de estos datos es precisa la encriptación con algoritmos y claves de fuerza suficiente sin perder de vista su coste de procesamiento.

En este prototipo el blindado de datos considerará el cifrado simétrico de los datos para evitar su comprensión y adicionalmente la comprensión de relaciones de similitud entre dos datos (dos cadenas iguales cifradas con la misma clave tienen la misma cadena como resultado) y la sustitución o cambio de datos por otros cifrados con la misma clave añadiendo bytes aleatorios y un resumen del dato sin cifrar respectivamente[12].

Las distintos procesos técnicos necesarios para los dos procesos principales quedan delegados en el módulo de cifrado (clase CipherModule) de la cual la clase aspecto SensibleData en su inicialización obtendrá una instancia con la configuración apropiada.

4.3.1 Clase CipherModule

La clase CipherModule utiliza JCE para proveer los métodos necesarios para los distintos procesos genéricos de encriptación tales como el cifrado o el resumen con el objetivo de servir como herramienta para los distintos aspectos presentes o futuros del paquete. Los algoritmos que puede soportar en esta versión prototipal son triple DES para el cifrado y SHA-1 para la generación de resúmenes.

4.3.1.1 Funcionamiento

Al invocarse su constructor son aportadas en los argumentos las cadenas de texto que designan los algoritmos de cifrado y resumen y la ruta de la clave con las que quedará configurado el funcionamiento del nuevo objeto.

Todos los métodos públicos propagan excepciones de tipo BreachException las cuales son configuradas con respecto a la valoración de la excepción primaria y el lugar donde se produce dentro de cada proceso.

Para que una instancia de CipherModule se considere preparada para su funcionamiento debe de ser invocado el método ‘init’ que provocará la inicialización interna del objeto pudiendo propagar una excepción BreachException en el caso de que dicha inicialización no sea concluida con normalidad.

4.3.1.2 Métodos Públicos

encode

```
public String encode( String data ) throws BreachException
```

Devuelve una cadena en formato BASE64 resultado de cifrar la cadena argumento con los algoritmos configurados.

decode

```
public String decode( String data ) throws BreachException
```

Devuelve una cadena en formato BASE64 resultado de desencriptar la cadena argumento con los algoritmos configurados.

getResume

```
public String getResume( String data ) throws BreachException
```

Devuelve una cadena resumen de la cadena de texto argumento según el algoritmo de resumen establecido en la instancia.

getSalt

```
public String getSalt( int bytesNumber ) throws BreachException
```

Obtiene una cadena de texto de la longitud especificada formada por caracteres aleatorios.

4.3.2 Clase Aspecto SensibleData

La clase aspecto SensibleData tiene como objetivo blindar los datos sensibles en algún momento crítico (por ejemplo al ser introducidos en una base de datos o enviados al exterior) y del mismo modo al ser accedido un dato que debiera estar cifrado con la misma configuración (con las mismas claves y variables) proceder a su validación y obtener el dato original. En esta implementación base los pointcuts deben capturar/retornar 2 objetos String con el dato principal y un dato referencia (por ejemplo la 'id' del usuario de un banco cuyo saldo se procede a establecer) que serán los datos necesarios para el proceso de blindaje.

El proceso de blindado consta de los siguientes pasos:

1. Generación de una cadena resumen de la cadena más una cadena referencia y codificación de esta a base64:

2. Adición de un número determinado de caracteres aleatorios al comienzo de la cadena con el objetivo de disfrazar el dato una vez encriptado de manera que si fuera encriptado varias veces dato pueda tener codificaciones diferentes y de ese modo no poder ser relacionado. Por ejemplo un individuo malintencionado que entrara en la base de datos no podría encontrar una referencia de cual es el valor ”0”.
3. Encriptación de la cadena formada por la cadena más los caracteres aleatorios mediante un algoritmo simétrico y la clave de cifrado de la aplicación y codificación a base64. Este paso garantizará que el dato solo podrá ser accedido mediante el conocimiento de la clave de encriptación.
4. Adición al dato procesado de un espacio más el resumen generado anteriormente (base 64 no tiene el carácter ‘espacio’ de modo que éste sirve para separar el dato del resumen). Este paso garantizará la autenticación del dato con respecto a una referencia, lo que impedirá que los datos sean intercambiados o suplantados de manera similar a como lo hacen los algoritmos MAC. Por ejemplo un individuo malintencionado no podría intercambiar con éxito dos valores de dos usuarios en una base de datos debido a que estos datos contienen la prueba de que pertenecen a cada usuario.

Ejemplo de dato cifrado:

Dato: “1500”
Referencia: “Ricardo Gómez”
Dato cifrado y resumen:
F8K6w6RuFVY44oCT4oSiUzE1MDAuMA== Qe7v3DYobr3rJt1yg5z6DPkqPO0=

El proceso de inversión del blindado consta de los siguientes pasos:

1. Separación de dato y resumen.
2. Descifrado del dato. Si la validación es fallida se considerará una posible infracción de seguridad.
3. Eliminación de caracteres de disfraz del dato.
4. Obtención de un resumen del dato más su referencia codificado en base 64 y comparación con el resumen que acompañaba el dato. Si la validación es fallida se considerará una posible infracción de seguridad.

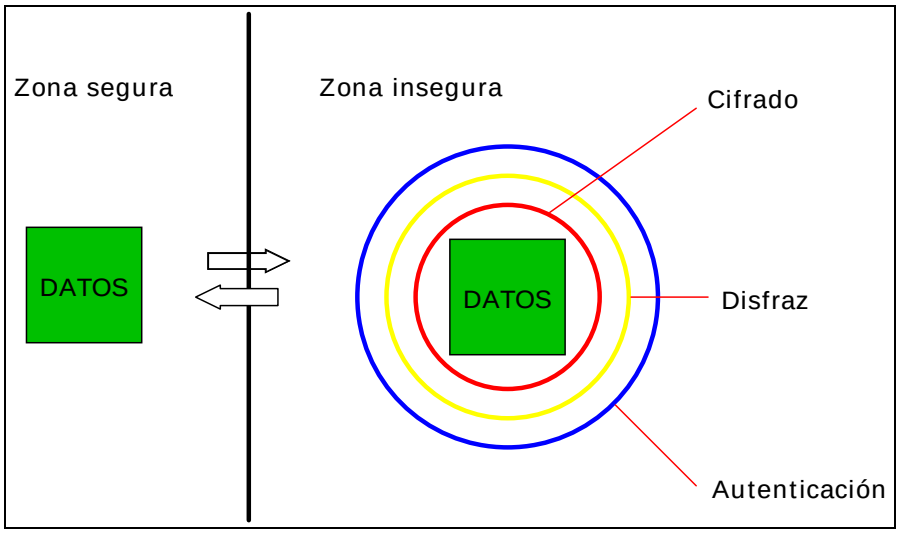

Figura 17 - Proceso de protección de datos

Los algoritmos que puede utilizar esta versión prototipo serán triple DES para el cifrado y SHA-1 para los resúmenes.

4.3.2.1 Interfaz de Reutilización

Pointcuts

settingSensibleData

```
protected pointcut settingSensibleData (String sensibleData, String referral);
```

Debe definir el joinpoint en el que un dato que se considere crítico deba de ser protegido, obteniendo la cadena del dato y la cadena referencia.

accedingSensibleData

```
protected pointcut accedingSensibleData (String referral);
```

Debe definir el joinpoint donde se accede a un dato que haya sido blindado con la misma configuración que este aspecto para que el advice adecuado intermedie validando el dato y devolviendo el dato sin blindar. Debe de obtener la cadena del dato cifrada y la cadena referencia.

Ambos pointcuts imponen una firma específica debido a que pretenden capturar joinpoints desconocidos a priori.

Variables reasignables

```
protected int SALT_BYTES_NUMBER = 10;
protected String ENCRYPTION_ALGORITHM = "DESede";
protected String DIGEST_ALGORITHM = "SHA-1";
protected String KEY_PATH = "C:\\key";
```

ENCRYPTION_ALGORITHM: define el algoritmo utilizado para la encriptación. Solo admite “DESede”.

DIGEST_ALGORITHM: define el algoritmo utilizado para la generación de resumenes. Solo admite “SHA-1”.

KEY_PATH: define la ruta del archivo que contiene la clave de cifrado.

SALT_BYTES_NUMBER: define el numero de caracteres aleatorios que se añadirán/eliminarán en los procesos.

Métodos redefinibles

`dataProcessFailed`

```
protected void dataProcessFailed( JoinPoint jp )
```

Es invocado tras producirse una anomalía en el proceso de blindado de un dato.

`dataCorrupted`

```
protected String dataCorrupted( JoinPoint jp )
```

Es invocado tras producirse una anomalía en el proceso inverso de blindado de un dato y ha de devolver una cadena de texto sustituta en el caso de que la ejecución continúe.

Métodos reutilizables

`process`

```
protected String process( String data, String referral ) throws BreachException
```

Devuelve la cadena resultante del proceso de blindado realizado con las dos cadenas argumentos.

unprocess

```
protected String unProcess( String pData, String referral ) throws
BreachException
```

Devuelve la cadena resultante del proceso inverso de desblindado realizado con las dos cadenas argumentos.

En ambos métodos si no se desea garantizar que el dato no ha sido intercambiado o bien si no se dispone de una cadena referencia para el caso el argumento 'referral' bien puede ser una cadena vacia.

Los dos métodos propagan la excepción BreachException que indicará una anomalía en el proceso ya sea dada por un error de sistema o por la detección de una violación de seguridad y que podrá ser procesada de manera más adecuada desde el código de invocación de los métodos.

4.3.2.2 Núcleo de Ejecución

Métodos motor

encodeData

```
private String encodeData ( String data )throws BreachException
```

Devuelve la cadena resultante de la encriptación de la cadena argumento por el módulo de cifrado.

addSalt

```
private String addSalt ( String data )throws BreachException
```

Devuelve una cadena compuesta por la cadena inicial y el número de caracteres aleatorios indicado por la variable SALT_BYTES_NUMBER añadidos al principio de la cadena que son obtenidos del módulo de cifrado de cifrado.

getResume

```
private String getResume ( String data, String referral )throws BreachException
```

Devuelve una cadena resumen producida mediante el módulo de cifrado a partir de la cadena compuesta por las dos cadenas argumentos.

decodeData

```
private String decodeData ( String data )throws BreachException
```

Devuelve la cadena resultado de desencriptar la cadena argumento mediante el uso del modulo de cifrado.

removeSalt

```
private String removeSalt ( String data )throws BreachException
```

Devuelve la cadena resultado de eliminar SALT_BYTES_NUMBER caracteres de la cadena argumento teniendo en cuenta que estos fueron añadidos por el proceso de 'addSalt' (son eliminados del principio de la cadena).

checkResume

```
private void checkResume ( String resumeData, String mainData, String referral
        )throws BreachException
```

Comprueba que la cadena resumeData sea un resumen válido del dato obteniendo para ello el resumen correcto a partir de la cadena compuesta por las cadenas mainData y referral. En el caso de que la cadena resumen resulte ser inválida (el resumen generado no coincida con el producido a partir de mainData y referral) se producirá una excepción BreachException.

Inicialización

Al ejecutarse el constructor se crea una instancia de CipherModule pasando como argumentos las cadenas de algoritmo de encriptación, algoritmo de resumen y ruta de la clave de encriptación que configurarán su funcionamiento. Seguidamente se invoca el método 'init' del objeto de cifrado el cual puede producir una excepción BreachException en el caso de que haya alguna anomalía en la inicialización del objeto.

Funcionamiento

Advice 1

```
void around(String sensibleData, String referral) :
                        settingSensibleData(sensibleData, referral)
```

Advice 2

String around(String referral) : accedingSensibleData(referral)

El advice 1 intermedia en los puntos de corte de establecimiento de un dato crítico intercambiando la cadena del dato por una cadena procesada obtenida mediante el método 'process'. El método 'process' obtendrá una cadena resultado mediante el uso secuencial de los métodos 'encodeData', 'addSalt' y 'getResume'.

El advice 2 intermedia en los puntos de corte de obtención de un dato crítico intercambiando la cadena del dato obtenido por una cadena procesada obtenida mediante el método unprocess que correspondrá al dato original sin el blindaje correspondiente y con la garantía de que no fue modificado. El método unprocess obtendrá la cadena resultado a partir de las dos cadenas separadas por espacio (en base 64) que componen la cadena blindada argumento mediante el uso secuencial de los siguientes métodos 'decodeData', 'removeSalt' y 'checkResume'.

Todos los métodos pueden propagar excepciones de tipo BreachException producidas por la invocación de los métodos del módulo de cifrado. El método 'process' o 'unprocess' propagará a su vez la excepción de manera que ésta sea gestionada por el proceso 'raíz' que generalmente será el código del advice1 o el advice2 (aunque podría tratarse de la invocación de 'process' o 'unprocess' desde el código de extensión de una subclase). Cuando los advices 1 y 2 gestionan una anomalía invocan finalmente los métodos redefinibles 'dataProcessFailed' y 'dataCorrupted'.

4.3.2.3 Utilización Básica

Los pointcuts 'settingSensibleData' y 'accedingSensibleData' deben de ser redefinidos teniendo en cuenta que debido a que el lenguaje no aporta suficiente flexibilidad el aspecto impone una única firma para cada uno de los pointcuts/advices, de modo tal y como se sugiere en el apartado 4.1.2 acerca de las firmas determinadas y los advices del tipo 'around' para joinpoints desconocidos de antemano. En el caso en el que dicha firma no coincida con la requerida puede alterarse el código OOP para que lo haga o bien puede implementarse un sistema intermediador o puente mediante AspectJ. Como último recurso el desarrollador puede implementar una subclase que simplemente implemente sus propios pointcuts y advices aprovechando los métodos 'process' y 'unprocess'.

La variable KEY_PATH deberá ser reasignada para indicar el fichero que contendrá la clave de cifrado. Opcionalmente ENCRYPTION_ALGORITHM Y DIGEST_ALGORITHM podrían ser reasignadas para configurar los algoritmos de encriptación y resumen a utilizar en el caso en el que la versión básica del aspecto se ampliara para poder funcionar con varios algoritmos. SALT_BYTES_NUMBER puede ser reasignada también para especificar otro número de caracteres aleatorios usados en los procesos.

El método 'dataProcessFailed' puede ser redefinido para la realización de acciones tras la detección de una anomalía en el blindado de un dato.

El método 'dataCorrupted' puede ser redefinido para la realización de acciones tras la detección de una anomalía en el proceso inverso de blindado de un dato. El método debe

devolver un objeto String que será el objeto que obtendrá el punto de ejecución en el código OOP que invocó la acción que fue intervenida (obviamente si se para el flujo de ejecución el objeto String devuelto será irrelevante).

4.4 Aspecto de Seguridad en las Comunicaciones

El aspecto de seguridad en las comunicaciones aporta principalmente privacidad y autenticación a las comunicaciones. En éste prototipo el aspecto se encarga de convertir las comunicaciones por sockets TCP en comunicaciones seguras mediante SSL utilizando JSSE y basando su acción en tres puntos:

- Intervención de la instanciación de objetos Socket y ServerSocket para su sustitución por la instanciación de sus subclases respectivas SSLSocket y SSLServerSocket.
- Forzamiento de una negociación SSL posterior a una nueva conexión de un socket seguro de forma que es ejecutada de manera controlada en el código de la clase aspecto.
- Notificación en los casos en los que el otro extremo de la comunicación no se autentica correctamente o no utiliza SSL en el proceso de negociación SSL.

4.4.1 Clase Aspecto Communication

La clase aspecto Communication captura las llamadas de instanciación de sockets y servidores de sockets no seguros que serán procesadas por los advices preparados para soportar los distintos tipos de constructores posibles y aplicar el método adecuado en relación a cada uno. En la ejecución de los advices se utilizarán las factorías de la clase aspecto para obtener la instancia de socket o servidor de sockets seguros que será devuelta como resultado.

4.4.1.1 Interfaz de Reutilización

Poincuts

newSensibleSocket

```
protected pointcut newSensibleSocket(): call (Socket.new(..)) &&
                    !within(securityAspect.Communication+);
```

newSensibleServerSocket

```
protected pointcut newSensibleServerSocket(): call (ServerSocket.new(..)) &&
                    !within(securityAspect.Communication+);
```

Deben capturar las llamadas al constructor de clases Socket y ServerSocket respectivamente cuya comunicación se suponga sensible a intromisiones. Se supone para ello que si el código obtiene dichas instancias de alguna clase factoría o con algún método similar es porque las características de esos objetos están controladas en el código OOP y su seguridad no debe de tenerse en cuenta por lo que solo ha de capturarse la instanciación directa de las clases Socket y ServerSocket por medio de una llamada directa a su constructor.

No especifican argumentos a recoger puesto que se relega en los advices la característica de soportar las distintas posibilidades relativas a cada uno de los constructores de las clases Socket y ServerSocket.

Por defecto están definido para capturar los intentos de creación de cualquier objeto Socket o ServerSocket que no se den en éste mismo aspecto o en sus subclases.

secureSocketAccept

```
protected pointcut secureSocketAccept(): call (Socket ServerSocket+.accept());
```

secureSocketConnection

```
protected pointcut secureSocketConnection(): call (void Socket+.connect(..));
```

Deben capturar las llamadas tras las cuales se establece una conexión de un objeto Socket que se suponga seguro exceptuando las que son consecuencias de la creación directa de un objeto Socket por medio de los constructores que devuelven un Socket conectado. Su funcionalidad principal es la de permitir la ejecución de un proceso de 'handshake' posterior a la conexión.

secureSocketAccept debe seleccionar las recepciones de conexiones de los objetos servidores de sockets que sean seguros. Por defecto está definido para capturar las llamadas al método 'accept' de cualquier instancia de ServerSocket* o sus subclases.

secureSocketConection debe seleccionar las nuevas conexiones de sockets seguros. Por defecto capturará las llamadas al método 'connect' de cualquier instancia de Socket* o sus subclases.

**Es importante denotar que en el código OOP los objetos SSLServerSocket y SSLSocket usurpadores por consecuencia de la acción de esta clase aspecto tendrán por lo general la forma de sus superclases (ServerSocket y Socket) y por tanto el código AO accederá a ellos como tales, siendo labor de los advices adivinar el verdader tipo de la instancia.*

secureSocketHandshaking

```
protected pointcut secureSocketHandshaking(): call (void
SSLSocket.startHandshake());
```

Debe capturar la ejecución explícita mediante código de los 'handshake' de SSL con el fin de que pueda controlarse el suceso mediante el código AOP.

Por defecto está definido para capturar la llamada al método startHandshake de las instancias de SSLSocket, para ello se ha tenido en cuenta que dicho proceso de negociación no es realizado automáticamente salvo en el momento de la primera lectura/escritura o la primera llamada al método getSession().

Variables reasignables

```
protected String SELF_KEYS_PATH = "C:\\keystore";
protected String TRUST_KEYS_PATH = "C:\\keystore";
protected String TRUST_KEYSTORE_PASSWORD = "password";
protected String SELF_KEYSTORE_PASSWORD = "password";
protected String SELF_KEYS_PASSWORD = "password"
protected boolean NEED_HOST_AUTH = false;
protected boolean NEED_SERVER_AUTH = false;
```

TRUST_KEYS_PATH: define la localización del almacén de claves que contiene las credenciales en las que confía la aplicación.

SELF_KEYS_PATH: define la localización del almacén de claves que contiene las credenciales de la aplicación.

TRUST_KEYSTORE_PASSWORD: define el password del almacén de claves que contiene las credenciales en las que confía la aplicación.

SELF_KEYSTORE_PASSWORD: define el password del almacén de claves que contiene las credenciales de la aplicación.

SELF_KEYS_PASSWORD: define el password de las credenciales contenidas en el almacén de claves propias que serán utilizadas.

NEED_HOST_AUTH: define si para establecer una sesión SSL entrante se ha de exigir autenticación del otro extremo. Por defecto su valor es 'false'.

NEED_SERVER_AUTH: define si para establecer una sesión SSL saliente se ha de exigir autenticación del otro extremo. Por defecto su valor es 'false'.

Variables reutilizables

sslSocketFactory

```
private SocketFactory sslSocketFactory;
```

Contiene la referencia a la instancia de SSLSocketFactory que se inicializó en el constructor del aspecto y que será utilizada para obtener objetos SSLSocket.

sslServerSocketFactory

```
private ServerSocketFactory sslServerSocketFactory;
```

Contiene la referencia a la instancia de SSLServerSocketFactory que se inicializó en el constructor del aspecto y que será utilizada para obtener objetos SSLServerSocket.

4.4.1.2 Núcleo de Ejecución

Inicialización

La ejecución del constructor del aspecto invoca el método 'initFactories' con el cual asigna a las variables sslServerSocketFactory y sslSocketFactory las instancias de SSLServerSocketFactory y SSLSocketFactory. Para ello se crea un objeto SSLContext inicializado con las credenciales propias y las credenciales de confianza obtenidas de los almacenes de claves configurados.

Funcionamiento

Advice 1

```
ServerSocket around () throws IOException : newSensibleServerSocket ()
```

El advice 1 interviene en las llamadas al constructor de ServerSocket seleccionadas cualesquiera sean sus argumentos utilizados para invocar según dichos argumentos el método adecuado para crear una instancia de SSLServerSocket a partir de la factoría referenciada por sslServerSocketFactory. El objeto SSLServerSocket será configurado para requerir o no autenticación del cliente y será devuelto como una instancia de ServerSocket. Del mismo modo que los constructores de ServerSocket declara la propagación de excepciones de tipo IOException.

Advice 2

```
Socket around ()  : newSensibleSocket() && args()
```

Advice 3

```
Socket around () throws IOException : newSensibleSocket() &&
                    args(Object, Object, ..)
```

Los advices 2 y 3 intervienen en las llamadas al constructor de Socket seleccionadas dependiendo de sus argumentos, devolviendo una instancia de SSLSocket obtenida de la factoría referenciada por sslSocketFactory que será configurada para requerir o no autenticación del servidor.

El advice 2 interviene en la invocación del constructor sin argumentos de Socket creando de manera homóloga un objeto SSLServerSocket sin conectar. En el caso de producirse una excepción IOException en el método de la factoría ésta no se propagará puesto que el constructor original no lo hace y el código OOP intervenido no la espera, de manera que el objeto devuelto por el constructor será null;

El advice 3 interviene en la invocación de los constructores de dos o más argumentos de Socket y del mismo modo que estos declara la propagación de excepciones de tipo IOException. Posteriormente a la creación y configuración del objeto SSLSocket puesto que se tratará de un socket conectado se procederá a la ejecución del proceso de 'handshake'.

El único constructor público de Socket de un argumento (una instancia de Proxy) queda fuera del rango de acción del aspecto.

Advice 4

```
after() returning (Socket socket) throws IOException : secureSocketAccept()
```

Advice 5

```
after() throws IOException : secureSocketConnection()
```

Los advices 4 y 5 ejecutarán el proceso de 'handshake' tras la conexión de un socket SSLSocket. La ejecución del 'handshake' para las nuevas instancias de SSLSocket ya conectadas se produce dentro del advice 3.

Puesto que dentro del rango del código OOP estas instancias serán entendidas como objetos Socket normales los pointcuts deberán estar definidos para capturar las acciones en clases Socket o hijas, por lo tanto antes de ejecutar el proceso confirmarán que se tratan de instancias SSLSocket y no de cualquier otro tipo.

El advice 4 lo hará únicamente cuando el método original haya devuelto un objeto Socket (y no haya lanzado una excepción). Tanto el advice 4 como el 5 declaran la propagación de excepciones de tipo IOException.

Advice 6

```
void around() throws IOException: secureSocketHandshaking()
```

El advice 6 intermediará y continuará en la llamada a los procesos de handshake seleccionados (generalmente todos los dados en éste mismo aspecto) para controlar las excepciones que pudieran ser producidas en él para procesarlas según la metodología del paquete para el tratamiento de anomalías de seguridad.

4.4.1.3 Utilización Básica

Los pointcuts están configurados a priori para convertir todas las comunicaciones implementadas en el código OOP objetivo en comunicaciones seguras por lo que es suficiente con que se añada a la compilación un aspecto subclase de Communications con las adecuadas reasignaciones de variables.

Para una aplicación más específica los pointcuts pueden ser redefinidos (por lo general 'newSensibleSocket' y 'newSensibleServerSocket') de manera que se acote el funcionamiento para algunos de los casos. Varias subclases distintas de Communication podrían utilizarse también para aplicar configuraciones distintas a distintos casos.

Las variables SELF_KEYS_PATH, TRUST_KEYS_PATH TRUST_KEYSTORE_PASSWORD, SELF_KEYSTORE_PASSWORD y SELF_KEYS_PASSWORD, deberán ser reasignadas para configurar las rutas de los almacenes de claves y los password de certificados y almacenes.

Opcionalmente pueden reasignarse las variables NEED_HOST_AUTH y NEED_SERVER_AUTH para configurar si es necesaria la autenticación del otro extremo en el caso de conexiones entrantes y conexiones salientes respectivamente.

4.5 Aspecto de Autenticación

El aspecto de autenticación tiene como principal objetivo provocar y gestionar requerimientos de autenticación basándose en JAAS.

El caso previsto para esta versión prototipo constaría de la utilización de un LoginModule del tipo KeyStoreLoginModule el cual valida los datos a partir de la información de un almacén de claves para lo que requiere tres Callbacks que deberán aportar:

1. El alias de una clave almacenada en almacén de caves.
2. El password del almacén de claves.
3. El password de la clave asociada con el alias.

4.5.1 Clase FrameCallbackHandler

FrameCallbackHandler provee la funcionalidad de CallbackHandler requerida para posibilitar los intentos de autenticación y obtiene los datos mediante la entrada manual del usuario a través de su interfaz visual.

Los datos que proporciona son identificador de usuario, password de grupo y password de usuario.

4.5.2 Clase Aspecto Authentication

La clase aspecto Authentication tiene como función provocar un proceso de autenticación en los puntos de ejecución en los que sea requerido de manera que si esta transcurre con éxito el usuario vigente será el autenticado y la ejecución del programa continuará normalmente y en el caso de que se produzcan el máximo número de intentos y por tanto la autenticación sea fallida el usuario vigente será nulo y se ejecutará el proceso pertinente.

La clase concreta de LoginModule requerida para validar los datos introducidos por los usuarios dependerá de la configuración de JAAS aunque como se expone anteriormente el único caso contemplado y garantizado en esta versión prototipo es el de la utilización del tipo KeyStoreLoginModule.

4.5.2.1 Interfaz de Reutilización

Pointcuts

doUserLogin

```
protected pointcut doUserLogin();
```

Debe definir aquellos puntos de ejecución en los que se requiera una autenticación de usuario.

doUserLogout

```
protected pointcut doUserLogout();
```

Debe definir aquellos puntos de ejecución en los que el usuario vigente deba pasar a ser nulo.

Variable reasignable

MAX_LOGIN_TRIES

```
protected int MAX_LOGIN_TRIES = 3;
```

Define el número máximo de intentos que se permiten en un proceso de autenticación antes de que dicha autenticación se considere fallida.

Variable reutilizable

adminSubject

```
protected Subject adminSubject = null;
```

Contiene el objeto 'Subject' del usuario autenticado vigente, en el caso de que sea 'null' no habrá usuario vigente.

Método redefinible

loginFailed

```
protected void loginFailed( JoinPoint jp )
```

Es invocado tras producirse un fracaso en la autenticación.

Método reutilizable

login

```
protected synchronized void login() throws BreachException{
```

Es la parte principal de la clase aspecto Authentication. Ejecuta el proceso de autenticación lanzando la excepción BreachException en el caso en el que se de una anomalía en el proceso o bien el usuario falle el número máximo intentos de autenticación.

4.5.2.2 Núcleo de Ejecución

El funcionamiento de la clase aspecto Authentication se reduce al proceso ejecutado por los dos advices:

Advice 1

```
before(): doUserLogin()
```

Ejecuta el proceso de autenticación antes de los puntos de unión determinados por ‘doUserLogin’ invocando el método ‘login’. Si el método login no produce ninguna excepción la variable adminSubject se actualiza obteniendo el nuevo ‘Subject’ vigente de la última instancia de loginContext utilizada. En el caso en el que se produzca una excepción ésta es notificada y el método ‘loginFailed’ es invocado.

Advice 2

```
after(): doUserLogout()
```

Asigna ‘null’ a la variable adminSubject posteriormente a los puntos de unión determinados por ‘doUserLogout’.

4.5.2.3 Utilización Básica

Los poincuts ‘doUserLogin’ y ‘doUserLogout’ deben de ser redefinidos adecuadamente y opcionalmente la variable MAX_LOGIN_TRIES puede ser reasignada.

Para configurar el funcionamiento de JAAS es necesario añadir en el fichero java.security una línea que indique la ruta del archivo JAAS.config que contiene la configuración del módulo de login:

```
login.config.url.1=file:RUTA
```

Para la utilización de un módulo de login de la clase KeyStoreLoginModule el archivo JAAS.config deberá contener una entrada semejante a ésta:

```
PFC {
 com.sun.security.auth.module.KeyStoreLoginModule required
 keyStoreURL="file:C:keystore"
 keyStoreType="JKS"
};
```

donde:

- ‘PFC’ es el nombre con el que se referenciará mediante el código y que por tanto no debe de ser modificado.
- com.sun.security.auth.module.KeyStoreLoginModule indica la clase que hará la función de modulo de login.
- ‘required’ es el valor de ‘flag’ que indica la categoría de la validación realizada por el módulo de login.
- keyStoreURL indica la dirección del almacén de claves a utilizar para la validación.
- keyStoreType indica el tipo de almacén de claves a utilizar para la validación.

4.6 Aspecto de Registro de Sucesos Críticos

Registra los sucesos destacables en la ejecución del programa mediante el uso de JAAS ya sean producidos éstos en la aplicación base o en el propio código del paquete de seguridad. Estos sucesos se clasifican en los siguientes grupos:

- Acciones de usuarios y administradores:
    - Engloba las acciones provocadas por la acción de un usuario externo o de un usuario administrador.
    - Se registran con un nivel de importancia FINE incluyendo diversa información acerca del punto de corte.
- Establecimiento y acceso de datos críticos:
    - Engloba los establecimientos y accesos de datos críticos en el código.
    - Se registran con un nivel de importancia INFO incluyendo información sobre los datos involucrados.
- Conexiones:
    - Engloba las conexiones entrantes y salientes que realiza la aplicación.
    - Se registran con un nivel de importancia INFO incluyendo información sobre la conexión.
- Autenticación

- Engloba los intentos de autenticación y los éxitos finales de estos (los fallos finales pertenecen al grupo de 'anomalías de seguridad').
- Se registran los intentos de autenticación se registran con un nivel de importancia INFO incluyendo información acerca de la 'id' de usuario que se intentó validar.
- Se registran las procesos de autenticación finalizados con éxito con un nivel de importancia INFO incluyendo información sobre el sujeto autenticado y sobre el punto de corte en el caso en el que la autenticación sea gestionada por el aspecto Authenticaton del paquete.

- Anomalías de seguridad

  - Engobla las llamadas al método 'noticeBreach' de las subclases de la clase aspecto Security las cuales implican la notificación final de una anomalía producida en el contexto de la seguridad de la aplicación.
  - Se registran con un nivel de importancia WARNING.

4.6.1 Clase Aspecto Logging

La clase aspecto Logging registra los sucesos determinados por sus pointcuts mediante el uso de una instancia de Logger.

El objeto objeto Logger funcionará con dos Handlers del tipo ConsoleHandler y FileHandler. El primero es añadido por defecto en la creación de la instancia de Logger (aunque puede ser eliminado) y representa la información registrada en la consola del sistema, el segundo y más importante representará la información en un fichero de texto plano.

4.6.1.1 Interfaz de reutilización

Pointcuts

userAction

```
protected pointcut userAction();
```

Captura las acciones destacables producidas por la acción de un usuario común.

adminAction

```
protected pointcut adminAction();
```

Captura las acciones destacables producidas por la acción de un administrador.

settingSensibleData

```
protected pointcut settingSensibleData();
```

Define los puntos de corte en los que se establece un dato que se considere crítico.

accedingSensibleData

```
protected pointcut accedingSensibleData();
```

Define los puntos de corte en los que se accede a un dato que se considere crítico.

newConnectionFrom

```
protected pointcut newConnectionFrom(): call (Socket ServerSocket+.accept());
```

Captura las llamadas al método 'accept' de las instancias ServerSocket (o subclase) que se consideren estimables. Por defecto captura cualquier llamada a dicho método 'accept'.

newConnectionTo

```
protected pointcut newConnectionTo(): call (void Socket+.connect(..));
```

Captura las llamadas al método 'connect' de Socket (o subclase) que se consideren estimables. Por defecto captura cualquier llamada a dicho método 'connect()'.

newSocketConnectedTo

```
protected pointcut newSocketConnectedTo(): call(Socket+.new(..)) ||
                    call(Socket SocketFactory+.createSocket(..));
```

Captura la instanciación de objetos Socket (o subclase) conectados que se consideren estimables con objetivo de que dicha conexión sea registrada.

Por defecto captura cualquier llamada a un constructor de la clase Socket (o subclase) o al método createSocket de una instancia de SocketFactory (o subclase).

authenticationTry

```
protected pointcut authenticationTry(String loginId):
        within (FrameCallBackHandler) &&
        call (void javax.security.auth.callback.NameCallback.setName(String))
        && args (loginId);
```

Selecciona los puntos de ejecución que son referencia de un intento de autenticación y cptura como argumento la cadena que contiene la identificación de usuario introducida en el intento de autenticación.

Está especialmente diseñado capturar la llamada al método 'setName' de la clase NameCallback dentro del código de la clase NameCallBackHandler. Por lo general no debe de ser redefinido salvo para ser anulado.

authenticationProcess

```
protected pointcut authenticationProcess(): call (void Authentication+.login());
```

Selecciona los puntos de ejecución donde comienza un proceso de autenticación.

Está especialmente diseñado para capturar la llamada al método 'login' de las subclases de la clase aspecto Authentication. Por lo general no debe de ser redefinido salvo para ser anulado.

4.6.1.2 Núcleo de Ejecución

Advice 1

```
before(): userAction()
```

Registra las acciones de usuario que van a producirse añadiendo información sobre el punto de unión.

Advice 2

```
before(): adminAction()
```

Registra las acciones de administrador que van a producirse añadiendo información sobre el punto de unión.

Advice 3

```
after(): settingSensibleData()
```

Registra las adjudicaciones de un dato crítico aportando los argumentos de del punto de unión.

Advice 4

```
after(): accedingSensibleData()
```

Registra las obtenciones de un dato crítico aportando los argumentos del punto de unión.

Advice 5

```
after() returning (Socket socket): newConnectionFrom()
```

Registra las conexiones entrantes que se han producido correctamente aportando información sobre la conexión.

Advice 6

```
after() returning () : newConnectionTo()
```

Registra las conexiones salientes explícitas (método 'connect' de Socket) que se han producido correctamente aportando información sobre la conexión.

Advice 7

```
after() returning (Socket socket): newSocketConnectedTo()
```

Registra las conexiones salientes que se han producido por consecuencia de la creación de un Socket conectado aportando información sobre la conexión. Para ello comprobará que se ha utilizado un constructor de Socket de dos argumentos o más.

Advice 8

```
after(String loginId): authenticationTry(loginId){
```

Registra los intentos de autenticación aportando la identificación de usuario utilizada que aporta los argumentos recogidos por el pointcut authenticationTry() .

Advice 9

```
after() returning() : authenticationProcess()
```

Registra la finalización correcta de los procesos de autenticación gestionado por el aspecto de autenticación del paquete. Aporta los datos disponibles del sujeto autenticado e información sobre el punto de unión y la pila de llamadas.

Advice 10

```
after() throwing(BreachException bex) : authenticationProcess()
```

Registra la finalización fallida de los procesos de autenticación gestionados por el aspecto de autenticación del paquete. Aporta información sobre el punto de unión y la pila de llamadas.

Advice 11

```
before(JoinPoint joinPoint, BreachException bex, String message):
        noticingBreach( joinPoint, bex, message)
```

Registra la ocurrencia de anomalías de seguridad producidas en el paquete aportando información sobre el punto de unión y la excepción BreachException que las representa.

4.6.1.3 Utilización Básica

Los pointcuts ‘settingSensibleData’ y ‘accedingSensibleData’ deben de ser redefinidos para capturar las acciones de adjudicación y obtención de datos sensibles. Generalmente harán referencia a los mismos puntos de unión que las subclases SensibleData implementadas de con la diferencia en que sus firma no necesitará obtener argumentos.

Los pointcuts ‘newConnectionFrom’, ‘newConnectionTo’, ‘newSocketConnectedTo’ y ‘noticingBreach’ pueden ser redefinidos para acotar el conjunto de acciones que debén de ser registradas o bien para ser anulados.

Los pointcuts authenticationProcess y authenticationTry tienen como objetivo el código asociado a la autenticación dentro del paquete de seguridad por lo que por lo general no deben de ser redefinidos salvo para ser anulados.

Los pointcuts adminAction y userAction pueden ser redefinidos para adicionalmente registrar las acciones relevantes producidas por usuarios y administradores asda.

Los parámetros de configuración de los Handler utilizados pueden ser configurados en el archivo logging.properties de la instalación de Java. Los parámetros para el Handler de ficheros de texto ‘FileHandler’ son:

- java.util.logging.FileHandler.level: que indicará el mínimo nivel de importancia que ha de tener una entrada de registro para que sea finalmente procesada por ese Handler. Por defecto ‘Level.ALL’.
- java.util.logging.FileHandler.formatter: indica la clase Formatter que dará formato a la información registrada. Por defecto ‘java.util.logging.XMLFormatter’.
- java.util.logging.FileHandler.filter: especifica el nombre de la clase Filter a utilizar. Por defecto no será utilizada ninguna.
- java.util.logging.FileHandler.encoding: especifica el nombre del juego de caracteres utilizados. Por efecto será utilizado la codificación por defecto de la plataforma.
- java.util.logging.FileHandler.limit: especifica el número límite de bytes que puede tener el archivo de registro, si el valor es 0 entonces no habrá límite. Por defecto no tendrá límite.
- java.util.logging.FileHandler.count: especifica el número de ficheros utilizados para el proceso cíclico. Por defecto será uno.
- java.util.logging.FileHandler.pattern: especifica la ruta del archivo de registro pudiendo utilizar caracteres para formar un patrón. Por defecto ‘%h/java%u.log’.
- java.util.logging.FileHandler.append: especifica si ha de encadenarse los datos de registro con los habidos anteriormente o si por el contrario ha de comenzar el registro con un fichero blanco. Por defecto ‘false’.

4.7 Plantilla

4.7.1 Clase Aspecto SensibleData

Interfaz de reutilización

Pointcuts

```
protected pointcut settingSensibleData (String sensibleData, String referral);

protected pointcut accedingSensibleData (String referral);
```

Variables reasignables

```
protected int SALT_BYTES_NUMBER = 10;
protected String ENCRYPTION_ALGORITHM = "DESede";
protected String DIGEST_ALGORITHM = "SHA-1";
protected String KEY_PATH = "C:\\key";
```

Métodos redefinibles

```
protected void dataProcessFailed( JoinPoint jp )
```

Métodos reutilizables

```
protected String process( String data, String referral ) throws BreachException

protected String unProcess( String pData, String referral )
        throws BreachException
```

Elementos externos

Fichero con clave simétrica de tipo DES EDE.

4.7.2 Clase Aspecto Communication

Interfaz de reutilización

Pointcuts

```
protected pointcut newSensibleSocket(): call (Socket.new(..)) &&
                !within(securityAspect.Communication+);

protected pointcut newSensibleServerSocket(): call (ServerSocket.new(..)) &&
                !within(securityAspect.Communication+);

protected pointcut secureSocketAccept(): call (Socket ServerSocket+.accept());

protected pointcut secureSocketConnection(): call (void Socket+.connect(..));

protected pointcut secureSocketHandshaking():
                call (void SSLSocket.startHandshake());
```

Variables reasignables

```
protected String SELF_KEYS_PATH = "C:\\keystore";
protected String TRUST_KEYS_PATH = "C:\\keystore";
protected String TRUST_KEYSTORE_PASSWORD = "password";
protected String SELF_KEYSTORE_PASSWORD = "password";
protected String SELF_KEYS_PASSWORD = "password"
protected boolean NEED_HOST_AUTH = false;
protected boolean NEED_SERVER_AUTH = false;
```

Variables reutilizables

```
private SocketFactory sslSocketFactory;

private ServerSocketFactory sslServerSocketFactory;
```

Elementos externos

- Almacén de claves con credenciales propias.

- Almacén de claves con credenciales en las que se confía.

4.7.3 Clase Aspecto Authentication

Interfaz de redefinición

Pointcuts

protected pointcut doUserLogin();

protected pointcut doUserLogout();

Variable reasignable

protected int MAX_LOGIN_TRIES = 3;

Variable reutilizable

protected Subject adminSubject = null;

Método redefinible

protected void loginFailed(JoinPoint jp)

Método reutilizable

protected synchronized void login() throws BreachException

Elementos externos

- Fichero de configuración JAAS con una entrada de nombre '**PFC'**

- Configuración del fichero **java.security**: login.config.url.[n]=file:[ruta conf JAAS]

4.7.4 Clase Aspecto Logging

Interfaz de redefinición

Pointcuts

```
protected pointcut userAction();

protected pointcut adminAction();

protected pointcut settingSensibleData();

protected pointcut accedingSensibleData();

protected pointcut newConnectionFrom(): call (Socket ServerSocket+.accept());

protected pointcut newConnectionTo(): call (void Socket+.connect(..));

protected pointcut newSocketConnectedTo(): call(Socket+.new(..)) ||
                    call(Socket SocketFactory+.createSocket(..));

protected pointcut authenticationTry(String loginId):
          within (FrameCallBackHandler) &&
          call (void javax.security.auth.callback.NameCallback.setName(String))
          && args (loginId);

protected pointcut authenticationProcess(): call (void Authentication+.login());
```

Métodos reutilizables

```
protected void log( Level level, String data )
```

Elementos externos

Configuración del fichero **logging.properties**

Fichero de registro.

Ejemplo de Uso

5.1 La Aplicación Banco

La aplicación banco es un sistema que provee la gestión por parte de usuarios y administradores de una base de datos con cuentas bancarias. Los usuarios lo harán remotamente mediante una comunicación por sockets TCP y mensajes y podrán iniciar una conexión/sesión con su password para poder realizar transacciones con su cuenta o bien consultar su valor. Los administradores utilizarán la interfaz gráfica de la aplicación y podrán realizar transacciones, obtener valores y crear nuevas cuentas.

Las necesidades de la aplicación son las siguientes:

- Seguridad en las comunicaciones: debe de existir la certeza de que los medios remotos a través de los cuales los usuarios interactúan con el banco son de confianza y de que la comunicación establecida para dicha actividad no puede ser intervenida ni percibida por terceros. Las actividades relacionadas con una conexión deben de pertenecer a una sesión iniciada al comienzo de dichas actividades en la cual el usuario comunicará el password de su cuenta bancaria. Debido a que se trata de un punto muy específico y unido a la lógica de aplicación ha sido implementado convencionalmente dentro del código Java.
- Seguridad en los datos persistentes sin tener en cuenta que la base de datos posea mecanismos propios de seguridad debe de garantizarse que esos datos no pueden ser entendidos, modificados ni relacionados.
- Seguridad en la administración del sistema: se debe de garantizar que las acciones de administración local estén realizadas por un usuario autorizado para ello. No requiere distintos niveles de autorización por lo que basta con que haya un grupo de usuarios autorizados; el usuario deberá de autenticarse como uno de ellos.
- Registro de sucesos: se deben registrar todos los sucesos relevantes como transacciones, acciones de administrador o anomalías en la seguridad del sistema.

5.1.1 Morfología y Funcionamiento

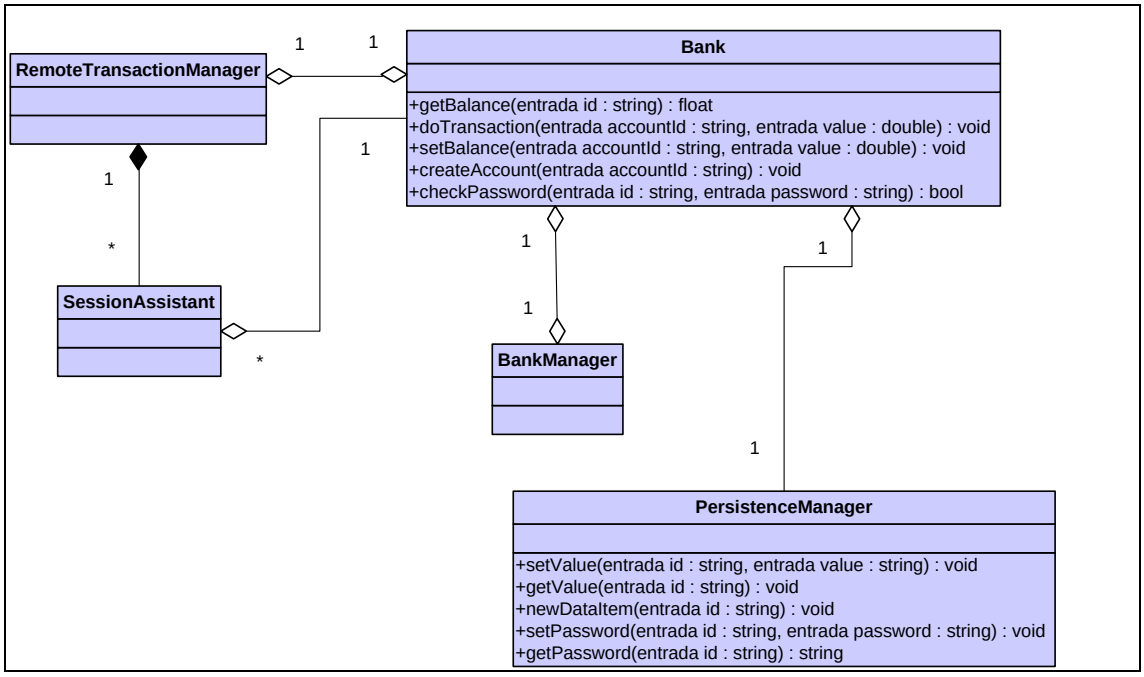

Figura 18 - Esquema de la aplicación Banco

5.1.1.1 Clase Bank

Es el objeto central del sistema en funcionamiento y se encarga de aportar las acciones básicas para la gestión del banco mediante sus métodos públicos que se servirán a su vez de los métodos de la instancia de PersistenceManager.

En su inicialización creará las instancias de BankManager, RemoteTransactionManager y PersistenceManager pasando a los constructores de los dos primeros una referencia a la propia instancia de Bank.

Sus métodos son:

- getBalance: obtiene el saldo de una cuenta.
- setBalace: establece el saldo de una cuenta.
- doTransaction: modifica el saldo de una cuenta.
- createAccount: crea una nueva cuenta con su password.
- checkPassword: comprueba que el password de una cuenta determinada sea igual al password chequeado.

5.1.1.2 Clase BankManager

Permite a un usuario administrador realizar acciones de gestión mediante una interfaz gráfica. Dichas acciones serán realizadas mediante el inicio de un nuevo hilo de ejecución con el que se encargará de hacer el requerimiento apropiado a la instancia de Bank. Estas acciones son:

- Obtener el saldo de una cuenta.
- Establecer el saldo de una cuenta.
- Crear una cuenta nueva y con identificación y password determinados.

5.1.1.3 Clase RemoteTransactionManager

Se encarga de establecer las sesiones de gestión de cuentas para usuarios remotos para lo que generará un nuevo hilo de ejecución en el que pondrá un servidor de sockets en espera de recibir conexiones entrantes. Por cada conexión establecida creará una instancia de SessionAssitant que gestionará la sesión iniciada sirviéndose de los argumentos aportados a su constructor: el nuevo socket de dicha conexión y la instancia de Bank.

5.1.1.4 Clase SessionAssistant

Es una clase privada de la clase RemoteTransactionManager y se encarga de inicializar y gestionar una sesión de cuenta con un usuario remoto según el protocolo establecido. Para ello creará un nuevo hilo de ejecución que se encargará de atender las comunicaciones del socket de conexión con el usuario.

Las acciones solicitadas por el usuario serán procesadas mediante la utilización de los métodos públicos de la instancia de Bank.

5.1.1.5 Clase PersistenceManager

Se encarga de gestionar la persistencia de datos de la aplicación mediante la utilización de una base de datos de tipo Access. Para ello aportará los siguientes métodos públicos:

- setValue: establece el saldo de una cuenta.
- getValue: obtiene el saldo de una cuenta.
- newDataItem: crea una nueva cuenta.
- setPassword: establece el password de una cuenta.
- getPassword: obtiene el password de una cuenta.

5.1.1.6 Base de Datos

La base de datos consta de una tabla donde almacena las cuentas bancarias. Cada cuenta bancaria consta de las columnas:

- ‘id’: identificación de la cuenta.
- ‘value’: valor del saldo de la cuenta.
- ‘password’: password de la cuenta.

5.1.1.7 Protocolo de Sesión Remota

El protocolo de sesión remota se basa en el envío de cadenas de texto cuyos parámetros van separados por el carácter ‘#’. El primero de los parámetros indica la acción solicitada y los siguientes los datos necesarios para dicho proceso. Las dos fases del proceso son:

1. Inicialización de la sesión de usuario:

a. 1. El usuario envía la acción ‘SESSION’ y los datos de identificación cuenta y password.

```
SESSION#JUAN#MIPASSWORD
```

b. El gestor de sesión responderá con las cadenas ‘OK’ O ‘FAILED’ dependiendo respectivamente de si los datos de identificación cuenta y password son correctos o si por el contrario no lo son.

2 Solicitud de acciones de gestión bancaria por parte del usuario: el usuario puede realizar las siguientes tres acciones:

a. Realizar una transacción:

El usuario envía el parámetro de acción ‘TRANSACTION’ y el parámetro del valor de la transacción.

El gestor de sesión confirmará que la acción se realizó correctamente con la cadena ‘OK’.

TRANSACTION#-325

b. Obtener el saldo de la cuenta

El usuario envía el parámentro de acción ‘GET_BALANCE’.

El servidor responderá con el valor del saldo de la cuenta del usuario.

c. Acabar la sesión

El usuario envía el parámetro de acción ‘END_SESSION’.

5.2 Aplicación del Paquete Seguridad

Como se indica en la seccion 4.1.2 para realizar una implementación específica de seguridad mediante el uso de un paquete basado en el patrón de redefinición expuesto es necesaria la creación de las clases aspectos adecuadas que extiendan cada una de las clases aspectos abstractos principales y posteriormente redefinir los elementos de la interfaz de reutilización.

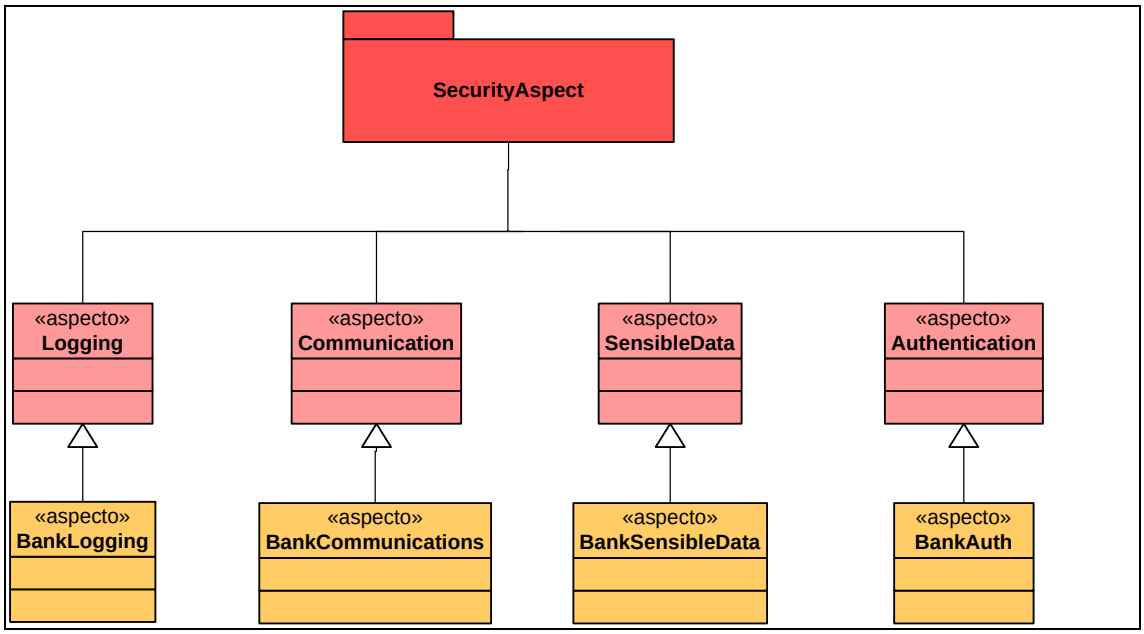

Figura 19 - Extensión en clases específicas

5.2.1 Aspecto de Conexiones Autenticadas y Protegidas

La clase aspecto BankCommunications que se encargará de la seguridad en las comunicaciones de la aplicación banco heredará de la clase aspecto Communication.

Para que las comunicaciones de la aplicación sean seguras es necesario sustituir la creación de la instancia ServerSocket para atender las conexiones de los clientes por una instancia de SSLServerSocket en su mismo puerto, esta habrá de requerir la autenticación del otro extremo de la comunicación a partir de las credenciales de confianza almacenadas en un almacén de claves local y debe de ser capaz de autenticarse en sentido contrario con sus propias credenciales. El proceso de 'handshake' deberá ser realizado de manera controlada de forma inmediata a generación del nuevo socket consecuente al inicio de una nueva conexión.

Puesto que la clase aspecto Communication está definida por defecto para funcionar del modo requerido en este caso, es suficiente con redefinir las variables que indican la ruta de los almacenes de claves y los passwords de éstos así como el password de las claves propias a utilizar en el proceso de handshake.

```
public aspect BankCommunications extends Communication {

protected void init(){
                        SELFKEYS_PATH = "C:\\PFC\\selfkeystore";
                        TRUSTKEYS_PATH = "C:\\PFC\\trustkeystore";
                        TRUST_KEYSTORE_PASSWORD = "storePassword";
                        SELF_KEYSTORE_PASSWORD = "storePassword";
                        SELF_KEYS_PASSWORD = "keyPassword";
        }
}
```

5.2.2 Aspecto de Seguridad de Datos Persistentes en la Aplicación

La clase aspecto que se encargará de la seguridad en los datos críticos persistentes de la aplicación banco heredará de la clase aspecto SensibleData.

Los datos críticos que usa la aplicación son el valor y el password de las cuentas de usuario por lo que estos datos habrán de ser procesados a la hora de ser introducidos en la base de datos y al extraerlos de ella habrán de ser sintetizados con el proceso inverso. Este proceso usará la clave de cifrado de un archivo local.

Los algoritmos que serán utilizados para el proceso de blindaje serán aquellos definidos por defecto (triple DES para el cifrado y SHA-1 para el resumen) y el valor de número de bytes aleatorios añadidos también será el definido por defecto.

El pointcut ‘settingSensibleData’ será definido para capturar las llamadas a los métodos ‘setValue’ y ‘setPassword’ de la clase PersistenceManager y obtener los dos argumentos. Así mismo el pointcut ‘accedingSensibleData’ será definido para capturar las llamadas a los métodos ‘getValue’ y ‘getPassword’ obteniendo el argumento.

La variable KEY_PATH será redefinida a la ruta del archivo donde se almacena la clave de cifrado para utilizar con el algoritmo triple DES.

Los métodos settingSensibleData y accedingSensibleData serán redefinidos para que en el caso en el que ocurra alguna anomalía en el proceso de blindado o de inversión del blindado del dato el hilo que produjo la acción será eliminado para impedir errores de consistencia en el sistema.

```
public aspect BankSensibleData extends SensibleData {

	protected void init(){
		KEY_PATH = "C:\\PFC\\cipherKey";
	}

	public pointcut settingSensibleData (String sensibleData, String referral):
		(call ( void PersistenceManager.setValue( String, String ) )
		|| call ( void PersistenceManager.setPassword( String, String ) ))
				&& args(referral, sensibleData);

	public pointcut accedingSensibleData (String referral):
		(call ( String PersistenceManager.getValue( String ) )
		|| call ( void PersistenceManager.getPassword( String ) ) )
				&& args (referral);

	protected void dataProcessFailed( JoinPoint jp ){
		Thread.currentThread().stop();
	}

	protected String dataCorrupted( JoinPoint jp ){
		Thread.currentThread().stop();
		return "";
	}

}
```

5.2.3 Aspecto de Autenticación en la Aplicación.

La clase aspecto que se encargará de la autenticación de usuarios administradores será bankAuth y heredará de la clase aspecto Authentication. Deberá encargarse de obligar la autenticación del usuario administrador de la aplicación siempre que éste intente ejecutar una de las acciones críticas; éstas son la creación de una cuenta nueva y la introducción de un nuevo valor para una cuenta. En el caso en el que la autenticación del usuario falle la acción deberá ser cancelada.

El proceso requerirá un almacén de claves para la validación del usuario mediante dos passwords el primero de los cuales será general o de grupo y coincidirá con el password del almacén de claves y el segundo será el de usuario y deberá ser el password de la clave cuyo alias sea igual al nombre de usuario.

El pointcut 'doUserLogin' deberá ser definido por tanto para capturar las llamadas realizadas desde la clase BankManager a los métodos 'setBalance' y 'createAccount' de la clase Bank.

El método 'loginFailed' deberá ser definido para parar el thread de la acción, (el cual se originó para la acción producida) cancelando de éste modo dicha acción.

```
public aspect BankAuth extends Authentication {

public pointcut doUserLogin() : within (BankManager)
                                && ( call (void Bank.setBalance(String, String))
                                || call (void Bank.createAccount(String, String)));

protected void loginFailed( JoinPoint jp ){
                        Thread.currentThread().stop();
        }

}
```

Para realizar la configuración de JAAS será necesario añadir en el archivo java.security una línea que indique la ruta del fichero JAAS.config que contendrá la configuración del módulo de login.

```
login.config.url.1=file:C:/PFC/JAAS.config
```

La funcionalidad de LoginModule requerida para validar los datos introducidos por los usuarios residirá en la clase KeyStoreLoginModule del paquete com.sun.security.auth que utilizará las claves de un almacén de claves para dicha validación. El fichero JAAS.config deberá contener la siguiente entrada:

```
PFC {
 com.sun.security.auth.module.KeyStoreLoginModule required
 keyStoreURL="file:C:/PFC/mainkeystore"
```

```
 keyStoreType="JKS"
};
```

El directorio del almacén de claves utilizado será “C:/PFC/” y su nombre ‘mainKeyStore’. Se definirá un password para el almacén que será el password de grupo y se añadirán tantas claves como usuarios autorizados haya con alias igual al nombre de usuario y password igual al password de usuario.

5.2.4 Aspecto de Registro de Sucesos Relevantes en la Aplicación

La clase aspecto BankLogging se encargará del registro de sucesos y heredará de la clase aspecto Logging. Deberá de registrar los siguientes sucesos:

- Las anomalías de seguridad.
- Las conexiones entrantes y salientes.
- Los intentos de autenticación y las autenticaciones realizadas con éxito.
- Las adjudicaciones y accesos de datos críticos.
- Las acciones realizadas por los usuarios remotos o por el administrador local.

Los pointcuts relativos a los tres primeros puntos están definidos de modo genérico en la clase aspecto Logging de manera que sirven adecuadamente para este caso. Los otros pointcuts habrán de ser definidos del siguiente modo:

Los pointcuts ‘settingSensibleData’ y ‘accedingSensibleData’ se definirán del mismo modo que los puntos de corte ya definidos para el aspecto de seguridad en los datos persistentes de la aplicación, en este caso sin capturar los argumentos.

El pointcut ‘userAction’ que deberá capturar las acciones de los usuarios se definirá para capturar todas las invocaciones de los métodos ‘getBalance’ y ‘doTransaction’ de la clase Bank realizadas desde el código de la clase RemoteTransactionManager.

El pointcut ‘adminAction’ que deberá capturar las acciones de los usuarios administradores de la aplicación se definirá para capturar todas las llamadas a métodos de la clase Bank desde la clase BankManager.

```
public aspect BankLogging extends Logging {

        public pointcut adminAction(): call(* Bank.*(..)) && within( BankManager );

        public pointcut userAction(): (call(String Bank.getBalance())
                                        || call(String Bank.doTransaction()) )
                                        && !within( TransactionManagerRmi );

        public pointcut settingSensibleData ():
                call ( void PersistenceManager.setValue( String, String ) )
```

```
                || call ( void PersistenceManager.setPassword( String, String ) );

        public pointcut accedingSensibleData ():
                call ( String PersistenceManager.getValue( String ) )
                || call ( void PersistenceManager.getPassword( String ) );

}
```

Puede ser necesario modificar la configuración global del funcionamiento del API de Logging del modo expuesto en el apartado 4.6.1.3 para seleccionar parámeros como la ruta del fichero de registro o el nivel mínimo de los sucesos que han de ser registrados.

Conclusiones

El objetivo primario llevado a cabo en este proyecto ha sido la elaboración de una implementación de seguridad en Java cuya filosofía se ajustara a las necesidades del entorno de desarrollo actual, cumpliendo para ello los requisitos de modularidad absoluta (ausencia de código intrusivo) y reusabilidad. Para ello se ha dispuesto del paradigma de la programación orientada a aspectos mediante la utilización del lenguaje AspectJ, lo cual ha permitido encapsular el código motor en un paquete indepediente formado por una combinación de clases aspecto y clases convencionales. Dicho motor ha sido desarrollado mediante la utilización de las APIs de seguridad y registro del JDK, cuyas arquitecturas han sumado sus virtudes a las del diseño expuesto aportando modularidad intrínseca en su implementación y permitiendo una mejor capacidad del paquete para ampliar su funcionalidad en el futuro.

Para aportar la independencia y transparencia necesarias para poder hablar propiamente de reusabilidad se ha elaborado un patrón basado en la redefinición y reutilización de elementos de las clases aspecto en el cual se diferencia la parte que debe ser conocida por el desarrollador de la parte que no tiene porque conocer. El resultado de la combinación de AspectJ y este patrón ha tenido como consecuencia la modularización absoluta del concern de seguridad en un paquete capaz de ser utilizado del mismo modo que puede ser utilizado un paquete común basado en OOP, salvo por la diferencia de que su uso no implica ningún tipo de intrusividad a pesar de tratarse de una funcionalidad claramente transversal.

Como conclusión el paquete resulta ser código capaz de ser utilizado de manera transparente y sencilla por un desarrollador de software, para bien cubrir las necesidades genéricas de seguridad de una aplicación o bien servir como base ampliable de una aplicación con requerimientos más específicos o numerosos. La potencia de esto se puede entrever en el caso de ejemplo expuesto en el que mediante la implementación de unas pocas líneas de código se ha podido dotar a una aplicación de un sistema íntegro de seguridad sin ningún efecto sobre su código, para lo cual el desarrollador no hubiera tenido que conocer los entresijos del funcionamiento del código de seguridad sino únicamente aquello relativo a la configuración externa de los APIs de Java utilizados y sus recursos (claves de cifrado, almacenes de claves, etc).

Sin embargo la utilización del patrón expuesto se ha visto enfrentada a las limitaciones congénitas del lenguaje y paradigma cuya problemática se representa dentro del denominado nexo de unión. Esto es en parte consecuencia de la falta de madurez de la AOP que a pesar de sus años de vida aún parece encontrarse lejos de encontrar su forma correcta.

Finalmente la aplicación de las técnicas empleadas ha tenido distintos efectos particulares en cada una de las clases aspecto funcionales del paquete:

- La clase aspecto Communication tiene un uso relativamente sencillo y directo de manera que es posible incluso su utilización por defecto sin necesidad de redefinir pointcuts como se vió en el caso de ejemplo, esto es debido a que el tipo de elementos a intervenir pertenecen a un paquete conocido de antemano (se supone el uso del API de red del JDK) y por tanto a un conjunto limitado de posibilidades a las que el código aspectual debe de adaptarse.

- La clase aspecto SensibleData a diferencia de la anterior tiene la necesidad de imponer una firma específica para los pointcuts redefinibles debido al desconocimiento de los elementos intervenidos (se supone que no pertenecen a ningún API conocido) y a la falta de capacidad del lenguaje para aportar la flexibilidad necesaria para adaptarse a distintos elementos posibles de manera sencilla. No obstante las firmas de los pointcuts podrían estar relacionadas con un API o arquitectura de implementación relativos a la adjudicación y establecimiento de datos usualmente utilizados por un grupo de desarrollo de manera que podría ser aplicada directamente, este es el caso de la aplicación banco que utiliza el tipo de firmas esperado para la adjudicación y obtención de datos de su clase de gestión de persistencia.
- La clase aspecto Authentication tiene una interfaz de redefinición muy sencilla debido a que solo se encarga de gestionar el momento de la autenticación y fácilmente sus subclases podrían abstenerse de utilizar la redefinición de pointcuts para únicamente aprovechar su método de 'login'. Para una funcionalidad más amplia y específica que gestionara por ejemplo la existencia de una o varias sesiones de autenticación o el control de autorización de Java sería posible ampliar fácilmente la clase aspecto implementada en sus subclases.
- La clase aspecto Logging tiene una interfaz de redefinición sencilla en la que la mayoría de los pointcuts únicamente deben de capturar el joinpoint preciso que ha de ser registrado. Así mismo varios de estos pointcuts están apropiadamente definidos por defecto debido a que capturan joinponts contenidos en las propias clases del paquete.

Si bien no en todos los casos la utilización del patrón de redefinición supone el mismo beneficio su utilización como filosofía de implementación ha aportado una línea de división entre aquello que debe de ser transparente y aquello que debe de ser conocido por el desarrollador, en definitiva, una metodología de reutilización clara. Dicha metodología ha estado enfocada en el lenguaje AspectJ, el cual a pesar de ser la referencia principal de las técnicas AOP y servir como modelo de varios de los otros lenguajes no puede entenderse como representante absoluto del paradigma. Sin embargo como filosofía a modo genérico y abstracto si que supone un concepto a tener muy en cuenta en la evolución de dicho paradigma y sus técnicas.

A partir de lo desarrollado en este proyecto surgen cuatro posibles líneas futuras:

1. Extensión del paquete de seguridad implementado.

El código de las clases desarrolladas puede ser extendido para ampliar su funcionalidad y flexibilidad como tal sería el caso de la adición del soporte para mayor número de algoritmos de cifrado, mayor configuración de registro o la inclusión de seguridad de sockets en RMI. Así mismo podría orientarse la funcionalidad aportada para hacer frente a casos de mayor detalle añadiendo nuevos aspectos o ampliando los existentes con el fin de imponer metodologías o arquitecturas particulares para un grupo de desarrolladores como por ejemplo para el control de sesiones de autenticación y autorización.

2. Elaboración de herramientas para la implementación y aplicación de código mediante el patrón de reutilización expuesto.

Estas herramientas podrían facilitar la implementación del código mediante la metodología expuesta enfocando especialmente las distintas vicisitudes que pueden darse en el nexo de unión indicando las circunstancias y posibilidades que tendría cada caso. Así mismo abordarían de una manera sencilla y directa la utilización del código implementado mediante la automatización del proceso de generación de las clases herederas específicas (y su adición a la lista de compilación), la indicación de la definición por defecto que poseerían los elementos de la interfaz de reutilización y la especificación textual escrita por el implementador.

3. Una extensión o modificación para el lenguaje AspectJ.

La técnica final utilizada debería de ser capaz de aportar gran capacidad de adaptación al núcleo de ejecución de manera que se eliminara la mayoría de los problemas del nexo de unión. Para ello debería de poder hacer frente a tres problemas fundamentales:

- El problema de los advices de tipo around, en los cuales la firma del joinpoint debe de coincidir con las propias de los advices si se requiere la continuación de la acción incluyendo los argumentos y también las excepciones propagadas.
- El problema en la obtención de argumentos de un joinpoint en tiempo de ejecución, el cual consiste en la conversión de tipos primitivos en tipos de envoltura para que el conjunto de argumentos pueda incluirse dentro de un array de objetos. Esto implica el desconocimiento de la verdadera naturaleza de aquellos argumentos cuya naturaleza sea la de un tipo de envoltura de tipos primitivos.
- El problema de la incapacidad de precognición acerca de los joinpoints basados en elementos desconocidos a priori. Para resolverlo sería necesario aumentar la capacidad del API de AspectJ para aportar la flexibilidad requerida. Es posible que dicha resolución tuviera que poder tener que adivinar la naturaleza funcional de cada argumento lo cual se asocia con la siguiente posible línea de evolución.

4. Un estudio del paradigma de la AOP orientado a la utilización transparente.

Dicho estudio podría suponer una integración mayor entre el paradigma OOP y AOP con el fin de que el nexo de unión de la filosofía expuesta en este proyecto desapareciera (o al menos su problemática) y así mismo quizás también la interfaz de reutilización.

Esto significaría que el código OOP aportara información sobre la naturaleza funcional de sus elementos permitiendo por ejemplo conocer la función de un argumento o una variable.

Para ello esa información explícita podría ir incluida en el propio nombre o bien en algún tipo de anotación a modo de metadatos[13].

Así mismo el núcleo de ejecución habría de considerarse totalmente capaz de adaptarse al código OOP objetivo por lo que las técnicas empleadas deberían de tener tanto reglas para definir el comportamiento aspectual como para autoadaptar dicho comportamiento aspectual al caso específico del mencionado código objetivo. Esto podría tener como consecuencia un trabajo en una línea meta-aspectual.

Anexo A: Código del Paquete de Seguridad

Clase aspecto Security

```
package securityAspect;


import org.aspectj.lang.*;

//Aspecto padre de seguridad
public abstract aspect Security {


	//VARIABLES


	//Flag de notificacion de anomalías al usuario
	protected static boolean NOTICE_BREACHS = true;


	//METODOS


	protected void init(){

	}

	//Notifica al usuario una anomalía de seguridad producida aportando el
punto de union y un mensaje adicional
	protected static void noticeBreach(JoinPoint joinPoint, BreachException
bex, String message){

		if (!NOTICE_BREACHS) return;

		System.out.println("-----------------------------------------------
----");

		if (bex.getKind() == BreachException.TRANSGRESSION)
System.out.println("ATENCIÓN, POSIBLE INTENTO DE INFRACCIÓN - " + message + " - :
" + bex.getMessage());
		if (bex.getKind() == BreachException.SYSTEM)
System.out.println("ERROR DEL SISTEMA DE SEGURIDAD - " + message + " - : " +
bex.getMessage());
		if (bex.getKind() == BreachException.UNKNOWN)
System.out.println("ERROR DESCONOCIDO - " + message + " - : " +
bex.getMessage());

		if (joinPoint!=null) System.out.println("PUNTO DE CORTE: \n" +
getJoinPointInfo(joinPoint));
		else System.out.println("Pila de llamadas : " + getStackTrace());

		bex.printStackTrace();

		System.out.println("-----------------------------------------------
----");
```

```
	}

	//Devuelve una cadena con la pila de llamadas
	protected static String getStackTrace(){
		StackTraceElement[] stes = Thread.currentThread().getStackTrace();
		String stackTrace = stes[0].getClassName();

		for (int i = 1; i<stes.length; i++)
			stackTrace = stackTrace + " <- " + stes[i].getClassName();
		return stackTrace;
	}

	//Devuelve una cadena informativa sobre un punto de unión
	protected static String getJoinPointInfo(JoinPoint joinPoint){
		return "Clase: " + joinPoint.getThis().getClass().toString() +
"\nAcción " + joinPoint.getSignature().toLongString() + "\nArgumentos dinámicos:
" + getArgsString(joinPoint.getArgs()) + "\nPila de llamadas: " +
getStackTrace();
	}

	//Devuelve una cadena que contiene un extracto de un array de objetos
	protected static String getArgsString(Object[] argsObject){

		if (argsObject == null ) return "-no argumentos-";
		if (argsObject.length == 0 ) return "-no argumentos-";
		String argsString = "' ";
		for (int i=0; i < argsObject.length; i++){
			if (argsObject[i]==null) argsString = " ' NULL ' ";
			else argsString = argsString + argsObject[i].toString() + " '
";
		}
		return argsString;

	}

}
```

Clase aspecto SensibleData

```
package securityAspect;

import org.aspectj.lang.*;

//Aspecto de proteccion de datos sensibles
public abstract aspect SensibleData extends Security {

	public SensibleData(){

		init();
		cipherModule = new CipherModule(ENCRYPTION_ALGORITHM,
DIGEST_ALGORITHM, KEY_PATH);

		try{
		cipherModule.init();
```

```
            }catch(BreachException bex){noticeBreach(null, bex, "No fue posible
la inicialización del módulo de cifrado para datos críticos");}

      }


      //VARIABLES


      //Numero de bytes aleatorios anadidos
      protected int SALT_BYTES_NUMBER = 10;

      //Algoritmo de encriptacion utilizado
      protected String ENCRYPTION_ALGORITHM = "DESede";

      //Algoritmo de resumen utilizado
      protected String DIGEST_ALGORITHM = "SHA-1";

      //Ruta de la clave de cifrado
      protected String KEY_PATH = "C:\\key";

      //instancia del modulo de cifrado utilizado
      private CipherModule cipherModule;


      //POINTCUTS


      //adjudicacion de un dato sensible
      protected pointcut settingSensibleData (String sensibleData, String
referral);

      //acceso a un dato sensible
      protected pointcut accedingSensibleData (String referral);


      //ADVICES


      //Advice 1
      //Intermedia blindando datos sensibles que van a ser adjudicados
      void around(String sensibleData, String referral) :
settingSensibleData(sensibleData, referral) {

            try{
            proceed(process(sensibleData, referral), referral);
            }catch(BreachException bex){noticeBreach(thisJoinPoint, bex, "Fallo
procesando blindaje de dato");dataProcessFailed( thisJoinPoint );}

      }

      //Advice 2
      //Intermedia desblindando datos sensibles que van a ser accedidos
      String around(String referral) : accedingSensibleData(referral){

            String sensibleData = proceed( referral );

            try{
```

```
            return unProcess(sensibleData, referral);
            }catch(BreachException bex){noticeBreach(thisJoinPoint, bex, "Fallo
desprocesando blindaje de dato");return dataCorrupted( thisJoinPoint );}

      }


      //METODOS


      //proceso de blindado de datos
      protected String process( String data, String referral ) throws
BreachException{

            String pData = "";
            pData = addSalt(data);
            pData = encodeData(pData);

            pData = pData + " " + getResume(data, referral);

            return pData;

      }

      //proceso de desblindado
      protected String unProcess( String pData, String referral ) throws
BreachException{

            String mainData;
            String resumeData;
            String[] sa = pData.split(" ");

            mainData = sa[0];
            resumeData = sa[1];

            mainData = decodeData(mainData);
            mainData = removeSalt(mainData);

            checkResume(resumeData, mainData, referral);

            return mainData;

      }

      //Cifrado de la cadena
      private String encodeData ( String data )throws BreachException{
            return cipherModule.encode(data);
      }

      //Adicion de caracteres aleatorios
      private String addSalt ( String data )throws BreachException{
            return cipherModule.getSalt(SALT_BYTES_NUMBER) + data;
      }

      //Obtencion de un resumen de la cadena
      private String getResume ( String data, String referral )throws
BreachException{
            return cipherModule.getResume(data + referral);
```

```
	}

	//Decodificacion de la cadena
	private String decodeData ( String data )throws BreachException{
		return cipherModule.decode(data);
	}

	//Eliminacion de los caracteres aleatorios de a cadena
	private String removeSalt ( String data )throws BreachException{
		return data.substring(SALT_BYTES_NUMBER - 1);
	}

	//Verificacion de resumen
	private void checkResume ( String resumeData, String mainData, String
referral )throws BreachException{

		if (!getResume(mainData, referral).equals(resumeData))
			throw new BreachException(BreachException.TRANSGRESSION,
"Dato suplantado");

	}

	//Proceso posterior a un fallo de procesamiento de datos
	protected void dataProcessFailed( JoinPoint jp ){

	}

	//Proceso posterior a un fallo posiblemente derivado de datos corruptos
	protected String dataCorrupted( JoinPoint jp ){
		return null;
	}

}
```

Clase aspecto Communication

```
package securityAspect;

import java.net.*;
import javax.net.*;
import javax.net.ssl.*;
import java.security.KeyStore;
import java.io.*;

//Aspecto de seguridad en las comunicaciones
public abstract aspect Communication extends Security{

	public Communication(){

		init();
		initFactories();

	}


	//VARIABLES
```

```
	//Ruta del almacen de claves con los certificados propios
	protected String SELF_KEYS_PATH = "C:\\keystore";

	//Ruta del almacen de claves con los certificados en los que se confia
	protected String TRUST_KEYS_PATH = "C:\\keystore";

	//Password de los almacenes de claves
	protected String TRUST_KEYSTORE_PASSWORD = "password";
	protected String SELF_KEYSTORE_PASSWORD = "password";

	//Password de los certificados propios
	protected String SELF_KEYS_PASSWORD = "password";

	//Requerimiento de autenticacion de host en el proceso de handshake
	protected boolean NEED_HOST_AUTH = false;

	//Requerimiento de autenticacion de servidor en el proceso de handshake
	protected boolean NEED_SERVER_AUTH = false;

	//Instancia de SSLSocketFactory utilizada
	private SocketFactory sslSocketFactory;

	//Instancia de SSLServerSocketFactory utilizada
	private ServerSocketFactory sslServerSocketFactory;


	//POINTCUTS


	//Creacion de un socket sensible
	protected pointcut newSensibleSocket(): call (Socket.new(..)) &&
!within(securityAspect.Communication+);

	//Creación de un servidor de sockets sensible
	protected pointcut newSensibleServerSocket(): call (ServerSocket.new(..))
&& !within(securityAspect.Communication+);

	//Invocacion del metodo accept de instancias de SSLServerSocket.
	protected pointcut secureSocketAccept(): call (Socket
ServerSocket+.accept());

	//Conexion explicita mediante codigo de conexiones en instancias de
SSLSocket
	protected pointcut secureSocketConnection(): call (void
Socket+.connect(..));

	//Ejecucion explicita mediante de un proceso de handshake
	protected pointcut secureSocketHandshaking(): call (void
SSLSocket.startHandshake());


	//ADVICES


	//Advice 1
	//Intermedia en la creacion de instancias de ServerSocket sensibles con
```

```
	//cualquiera de sus constructores creando en su lugar instancias de
SSLServerSocket.
	ServerSocket around () throws IOException : newSensibleServerSocket () {

		Object arguments[] = thisJoinPoint.getArgs();
		SSLServerSocket serverSocket = null;
		switch (arguments.length){
			case 0: serverSocket =
(SSLServerSocket)sslServerSocketFactory.createServerSocket();break;
			case 1: serverSocket =
(SSLServerSocket)sslServerSocketFactory.createServerSocket(((Integer)arguments[0]
).intValue());break;
			case 2: serverSocket =
(SSLServerSocket)sslServerSocketFactory.createServerSocket(((Integer)arguments[0]
).intValue(), ((Integer)arguments[1]).intValue());break;
			case 3: serverSocket =
(SSLServerSocket)sslServerSocketFactory.createServerSocket(((Integer)arguments[0]
).intValue(), ((Integer)arguments[1]).intValue(), (InetAddress)
arguments[2]);break;
		}
		serverSocket.setNeedClientAuth(NEED_HOST_AUTH);
		return serverSocket;

	}

	//Advice 2
	//Intermedia en la creacion de instancias de Socket con su constructor sin
	//argumentos creando en su lugar instancias de SSLSocket
	Socket around ()  : newSensibleSocket() && args() {

		SSLSocket socket;

		try{

		socket = (SSLSocket) sslSocketFactory.createSocket();

		}catch (IOException ioex){return null;}

		socket.setNeedClientAuth(NEED_SERVER_AUTH);
		return socket;

	}

	//Advice 3
	//Intermedia en la creacion de instancias de Socket con sus constructores
de
	//más de dos argumentos creando en su lugar instancias de SSLSocket
	Socket around () throws IOException : newSensibleSocket() && args(Object,
Object, ..){

		Object arguments[] = thisJoinPoint.getArgs();
		SSLSocket socket = null;

		switch (arguments.length){

			case 2: if (String.class.isInstance(arguments[0])) socket =
(SSLSocket)
```

```
sslSocketFactory.createSocket((String)arguments[0],((Integer)arguments[1]).intVal
ue());
                                  else   socket = (SSLSocket)
sslSocketFactory.createSocket((InetAddress)arguments[0],((Integer)arguments[1]).i
ntValue());break;
                     case 3: if (String.class.isInstance(arguments[0])) socket =
(SSLSocket)
sslSocketFactory.createSocket((String)arguments[0],((Integer)arguments[1]).intVal
ue());
                                  else   socket = (SSLSocket)
sslSocketFactory.createSocket((InetAddress)arguments[0],((Integer)arguments[1]).i
ntValue());break;
                     case 4: if (String.class.isInstance(arguments[0])) socket =
(SSLSocket)
sslSocketFactory.createSocket((String)arguments[0],((Integer)arguments[1]).intVal
ue(), (InetAddress)arguments[0],((Integer)arguments[1]).intValue());
                                  else socket = (SSLSocket)
sslSocketFactory.createSocket((InetAddress)arguments[0],((Integer)arguments[1]).i
ntValue(), (InetAddress)arguments[0],((Integer)arguments[1]).intValue());break;

             }

             socket.setNeedClientAuth(NEED_SERVER_AUTH); //Esto es importante.

             socket.startHandshake();
             return (Socket) socket;

      }

      //Advice 4
      //Ejecuta un proceso de handshake tras la creación de nuevas instancias de
      //SSLSocket producidas por una instancia de SSLServerSocket.
      after() returning (Socket socket) throws IOException :
secureSocketAccept(){

             if (SSLSocket.class.isInstance(socket))
((SSLSocket)socket).startHandshake();

      }

      //Advice 5
      //Ejecuta un proceso de handshake tras conexiones de instancias de
SSLSocket
      //producidas por una instancia de SSLServerSocket.
      after() throws IOException : secureSocketConnection(){

             Object socket = thisJoinPoint.getTarget();
             if
(SSLSocket.class.isInstance(socket))((SSLSocket)socket).startHandshake();

      }

      //Advice 6
      //Ejecuta los procesos de handshake controlando las excepciones producidas
en él
      void around() throws IOException: secureSocketHandshaking(){

             try{
```

```
            proceed();

            }catch(SSLHandshakeException sslhex){noticeBreach(thisJoinPoint, new
BreachException(BreachException.TRANSGRESSION, "Intento de conexión de extremo no
autenticado", sslhex), "Error en handshake SSL"); throw sslhex;}
            catch(SSLException sslex){noticeBreach(thisJoinPoint, new
BreachException(BreachException.TRANSGRESSION, "Intento de conexión de socket
externo no seguro", sslex), "Error en handshake SSL");}

      }


      //METODOS


      //inicializacion de las factorías de sockets y servidores de socket
      private void initFactories(){

            char[] keysPassword;
            char[] trustKeyStorePassword;
            char[] selfKeyStorePassword;
            KeyStore selfKeyStore;
            KeyStore trustKeyStore;
            TrustManagerFactory trustManagerFactory;
            TrustManager[] trustManagers;
            SSLContext context;
            KeyManagerFactory keyManagerFactory;
            KeyManager[] keyManagers;

            try{

            keysPassword = SELF_KEYS_PASSWORD.toCharArray();

            trustKeyStorePassword = TRUST_KEYSTORE_PASSWORD.toCharArray();
            selfKeyStorePassword = SELF_KEYSTORE_PASSWORD.toCharArray();

            trustKeyStore = KeyStore.getInstance( "JKS" );
            trustKeyStore.load( new FileInputStream(TRUST_KEYS_PATH),
trustKeyStorePassword );
            trustManagerFactory = TrustManagerFactory.getInstance(
TrustManagerFactory.getDefaultAlgorithm() );
            trustManagerFactory.init( trustKeyStore );
            trustManagers = trustManagerFactory.getTrustManagers();

            selfKeyStore = KeyStore.getInstance("JKS");
            selfKeyStore.load( new FileInputStream(SELF_KEYS_PATH),
selfKeyStorePassword);
            keyManagerFactory = KeyManagerFactory.getInstance(
KeyManagerFactory.getDefaultAlgorithm() );
            keyManagerFactory.init(selfKeyStore, keysPassword);
            keyManagers = keyManagerFactory.getKeyManagers();

            context = SSLContext.getInstance("TLS");
            context.init( keyManagers, trustManagers, null );
            sslSocketFactory = context.getSocketFactory();
            sslServerSocketFactory = context.getServerSocketFactory();
```

```
            }catch ( Exception ex ) {
                  noticeBreach(null, new
BreachException(BreachException.SYSTEM, ex), "Error en la inicialización de
SSL");
                  Thread.currentThread().stop();}

      }
}
```

Clase aspecto Authentication

```
package securityAspect;

import org.aspectj.lang.*;
import javax.security.auth.*;
import javax.security.auth.login.*;

//Aspecto de autenticacion
public abstract privileged aspect Authentication extends Security{

      public Authentication(){

            init();

      }


      //VARIABLES


      //Maximo numero de intentos fallidos de introduccion de datos de
autenticación.
      protected int MAX_LOGIN_TRIES = 3;

      //Ultimo LoginContext utilizado para autenticación
      private LoginContext loginContext;

      //Subject de la actual entidad autenticada
      protected Subject adminSubject = null;


      //POINTCUTS


      //Establece cuando ha de ejecutarse un proceso de autenticacion
      protected pointcut doUserLogin();

      //Establece cuando un usuario deja de estar autenticado.
      protected pointcut doUserLogout();


      //ADVICES


      //Advice 1
```

```
	//Autenticación del usuario
	before(): doUserLogin(){

		adminSubject = null;
		try{

		login();
		adminSubject = loginContext.getSubject();

		}catch(BreachException bex){
		noticeBreach(thisJoinPoint, bex, "");
		loginFailed(thisJoinPoint);}

	}

	//Advice 2
	//El usuario deja de estar autenticado
	after(): doUserLogout(){

				adminSubject = null;

	}


	//METODOS


	//realiza el proceso de autenticacion mediante JAAS
	protected synchronized void login() throws BreachException{

		try{
		loginContext = new LoginContext("PFC", new FrameCallBackHandler());
		}catch(LoginException lex){
			throw new BreachException(BreachException.SYSTEM, "Error
creando LoginContext", lex);}

		int loginTries;

		for (loginTries = 0; loginTries < MAX_LOGIN_TRIES; loginTries++){

			try{
			loginContext.login();
			}catch(LoginException lex){
				continue;}

			break;

		}

		if (loginTries >= MAX_LOGIN_TRIES)
			throw new BreachException(BreachException.TRANSGRESSION, "" +
loginTries + " intentos fallidos de autenticación de administrador");

	}

	//Proceso posterior a una autenticacion fallida.
	protected void loginFailed( JoinPoint jp ){
```

```
	}

}
```

Clase aspecto Logging

```
package securityAspect;

import java.net.*;
import javax.net.*;
import java.util.logging.*;
import java.util.*;
import java.security.*;
import javax.security.auth.*;
import org.aspectj.lang.*;

//Aspecto de registro de sucesos
public abstract privileged aspect Logging extends Security{

	public Logging(){

		init();
		initLogger();

	}


	//VARIABLES


	//Instancia de Logger utilizada
	private Logger logger;


	//POINTCUTS


	//Accion producida por un usuario
	protected pointcut userAction();

	//Accion producida por un administrador
	protected pointcut adminAction();

	//Adjudicacion de un dato sensible
	protected pointcut settingSensibleData();

	//Acceso a un dato sensible
	protected pointcut accedingSensibleData();

	//Nueva conexión entrante
	protected pointcut newConnectionFrom(): call (Socket
ServerSocket+.accept());

	//Nueva conexión saliente
	protected pointcut newConnectionTo(): call (void Socket+.connect(..));
```

```
	//Creacion de un socket conectado
	protected pointcut newSocketConnectedTo(): call(Socket+.new(..)) ||
call(Socket SocketFactory+.createSocket(..));

	//Intento de autenticacion
	protected pointcut authenticationTry(String loginId): within
(FrameCallBackHandler) && call (void
javax.security.auth.callback.NameCallback.setName(String)) && args (loginId);

	//Ejecucion del proceso de autenticacion
	protected pointcut authenticationProcess(): call (void
Authentication+.login());

	//Notificacion de falla de seguridad
	protected pointcut noticingBreach(JoinPoint joinPoint, BreachException
bex, String message): call (void Security+.noticeBreach(JoinPoint,
BreachException, String)) && args(joinPoint, bex, message);


	//ADVICES


	//Advice 1
	//Registro de acciones de usuario
	before(): userAction(){

		log(Level.FINE, "-Acción de administrador- Punto de corte: " +
getJoinPointInfo(thisJoinPoint));

	}

	//Advice 2
	//Registro de acciones de administrador
	before(): adminAction(){

		log(Level.FINE, "-Acción de usuario- Punto de corte : " +
getJoinPointInfo(thisJoinPoint));

	}

	//Advice 3
	//Registro de adjudicacion de datos sensibles
	after(): settingSensibleData(){

		Object[] argsObject = thisJoinPoint.getArgs();
		log(Level.INFO, "Adjudicación de un valor crítico: " +
getArgsString(argsObject));

	}

	//Advice 4
	//Registro de acceso a datos sensibles
	after(): accedingSensibleData(){

		Object[] argsObject = thisJoinPoint.getArgs();
		log(Level.INFO, "Acceso a un valor crítico: " +
getArgsString(argsObject));
```

```
	}

	//Advice 5
	//Registro de conexiones entrantes
	after() returning (Socket socket): newConnectionFrom(){

		String remotePort = new Integer(socket.getPort()).toString();
		log(Level.INFO, "Nueva conexión entrante: " +
getSocketInfo(socket));

	}

	//Advice 6
	//Registro de conexiones salientes
	after() returning () : newConnectionTo(){

		try{
			log(Level.INFO, "Nueva conexión saliente: " +
getSocketInfo((Socket)thisJoinPoint.getTarget()));
		}catch(ClassCastException cex){}

	}

	//Advice 7
	//Registro de conexiones salientes producidas por la creacion de un nuevo
socket conectado
	after() returning (Socket socket): newSocketConnectedTo(){

		Object[] as = thisJoinPoint.getArgs();
		if (as.length < 2) return;
		log(Level.INFO, "Nueva conexión saliente: " +
getSocketInfo(socket));

	}

	//Advice 8
	//Registro de intento de autenticacion
	after(String loginId): authenticationTry(loginId){

			log(Level.INFO, "Intento de autenticación con nombre de
usuario: " + loginId);

	}

	//Advice 9
	//Registro de un proceso de autenticacion correcta
	after() returning() : authenticationProcess(){

		String principalsString = "";
		Subject sb;

		try{
		sb =
((Authentication)thisJoinPoint.getThis()).loginContext.getSubject();
		}catch(ClassCastException cex){return;}
		Set s = sb.getPrincipals();
		Iterator i = s.iterator();
		while (i.hasNext()){
```

```
                    principalsString = principalsString +
((Principal)i.next()).toString();
             }

                    log (Level.INFO, "Autenticación correcta\nDatos del sujeto: "
+ principalsString + "\nPila de llamadas: " + getStackTrace() + "\nPUNTO DE
UNION: " + getJoinPointInfo(thisJoinPoint));

      }

      //Advice 10
      //Registro de un proceso de autenticacon fallida
      after() throwing(BreachException bex) : authenticationProcess(){ //el
método lanza una excepción BreachException luego la autenticación fue erronea

                    log (Level.WARNING, "Autenticación incorrecta" + "\nPila de
llamadas: " + getStackTrace() + "\nPUNTO DE UNION: " +
getJoinPointInfo(thisJoinPoint) );

      }

      //Advice 11
      //Registro de anomalias de seguridad
      before(JoinPoint joinPoint, BreachException bex, String message):
noticingBreach( joinPoint, bex, message){

             log(Level.WARNING, "ANOMALÍA DE SEGURIDAD \n" + message + "\n" +
bex.getMessage() + "\n" + getJoinPointInfo(thisJoinPoint));

      }


      //METODOS


      //Inicializa el objeto Logger a utilizar
      private void initLogger(){
             logger = Logger.getLogger("Main");
             try{
             logger.addHandler(new FileHandler());
             }catch(java.io.IOException ex){noticeBreach(null , new
BreachException(BreachException.SYSTEM, "Error de acceso al fichero de registro
", ex), "Error inicializando motor de registro");}
      }

      //Obtiene una cadena informativa sobre una instancia de Socket
      private String getSocketInfo( Socket socket ){

             String socketInfo = "";

             try{
                    String localAddress = socket.getLocalAddress().toString();
                    String localPort = new
Integer(socket.getLocalPort()).toString();
                    String remoteAddress = socket.getInetAddress().toString();
                    String remotePort = new Integer(socket.getPort()).toString();
                    socketInfo = "\nDirección local: " + localAddress + ":" +
localPort + "\nDirección remota: " + remoteAddress + ":" + remotePort ;
```

```
            }catch(Exception ex){}
            return socketInfo;

      }

      //Registra una cadena de datos con un nivel
      protected void log( Level level, String data ){

            logger.log(level, data);

      }

}
```

Clase CipherModule

```
package securityAspect;

import java.security.*;
import java.util.Random;
import sun.misc.*;
import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.*;

//Modulo ayudante de cifrado
public class CipherModule {

      public CipherModule(String ENCRYPTION_ALGORITHM, String DIGEST_ALGORITHM,
String KEY_PATH){

            this.ENCRYPTION_ALGORITHM = ENCRYPTION_ALGORITHM;
            this.DECRYPTION_ALGORITHM = ENCRYPTION_ALGORITHM;
            this.DIGEST_ALGORITHM = DIGEST_ALGORITHM;
            this.KEY_PATH = KEY_PATH;

      }

      //Algoritmos utilizados en la instancia
      private String ENCRYPTION_ALGORITHM;
      private String DECRYPTION_ALGORITHM;
      private String DIGEST_ALGORITHM;

      //Ruta de la clave secreta
      private String KEY_PATH;

      //Clave secreta de la instancia
      private SecretKeySpec cipherKey;

      private Random random;


      //Arranca el modulo de cifrado para poder ser utilizado
      public void init() throws BreachException{

            cipherKey = getCipherKey();
```

```
		random = new Random();

	}

	//Devuelve una cadena en formato BASE64 resultado de cifrar 'data'
	public String encode( String data ) throws BreachException{

		try{
			return new BASE64Encoder().encode(encode(data.getBytes("UTF-
8")));
		}catch(UnsupportedEncodingException ex){
			throw new BreachException(BreachException.UNKNOWN, ex);}

	}

	//Devuelve un array de bytes resultado de cifrar 'data'
	private byte[] encode( byte[] data ) throws BreachException{

		Cipher c;

		try{

		c = Cipher.getInstance(ENCRYPTION_ALGORITHM);

		}catch(NoSuchAlgorithmException ex){
			throw new BreachException(BreachException.SYSTEM, ex);}
		catch(NoSuchPaddingException ex){
			throw new BreachException(BreachException.SYSTEM, ex);}

		try{

		c.init(Cipher.ENCRYPT_MODE, cipherKey);

		}catch(InvalidKeyException ex){
			throw new BreachException(BreachException.SYSTEM, "Clave de
encriptación inválida", ex);}

		try{

		return c.doFinal(data);

		}catch(BadPaddingException ex){
			throw new BreachException(BreachException.SYSTEM, ex);}
		catch(IllegalBlockSizeException ex){
			throw new BreachException(BreachException.SYSTEM, ex);}

	}



	//Devuelve una cadena formada por 'bytesNumber' numeros aleatorios
	public String getSalt( int bytesNumber ) throws BreachException{

		byte[] b = new byte[bytesNumber];
		random.nextBytes(b);
		return new String(b);

	}
```

```
	//Devuelve una cadena en formato BASE64 resultado de aplicar un algoritmo
de resumen a 'data'
	public String getResume( String data ) throws BreachException{

		byte[] dataBytes;

		try{

		dataBytes = data.getBytes("UTF-8");

		}catch(UnsupportedEncodingException ex){
			throw new BreachException(BreachException.UNKNOWN, ex);}

		MessageDigest md;

		try{

		md = MessageDigest.getInstance(DIGEST_ALGORITHM);

		}catch(NoSuchAlgorithmException ex){
			throw new BreachException(BreachException.SYSTEM, ex);}

		md.update(dataBytes);
		byte[] b = md.digest();
		return new BASE64Encoder().encode(b);

	}

	//Devuelve una cadena en formato BASE64 resultado de descifrar 'data'
	public String decode( String data ) throws BreachException{

		try{

		return new String(decode(new BASE64Decoder().decodeBuffer(data)),
"UTF-8").substring(1);

		}catch(UnsupportedEncodingException ex){
			throw new BreachException(BreachException.UNKNOWN, ex);}
		catch(IOException ex){System.out.println("" + ex.toString());
			throw new BreachException(BreachException.UNKNOWN, ex);}

	}

	//Devuelve un array de bytes resultado de descifrar 'data'
	private byte[] decode( byte[] data ) throws BreachException {

		Cipher c;

		try{

		c = Cipher.getInstance(DECRYPTION_ALGORITHM);

		}catch(NoSuchPaddingException ex){
			throw new BreachException(BreachException.SYSTEM, ex);}
		catch(NoSuchAlgorithmException ex){
			throw new BreachException(BreachException.SYSTEM, ex);}
```

```
            try{

            c.init(Cipher.DECRYPT_MODE, cipherKey);

            }catch(InvalidKeyException ex){
                  throw new BreachException(BreachException.SYSTEM, "Clave de
encriptación inválida", ex);}

            try{


            return c.doFinal(data);

            }catch(BadPaddingException ex){
                  throw new BreachException(BreachException.TRANSGRESSION,
"Dato corrupto o incapacidad del sistema para desencriptar el dato", ex);}
            catch(IllegalBlockSizeException ex){
                  throw new BreachException(BreachException.TRANSGRESSION,
"Dato corrupto o incapacidad del sistema para desencriptar el dato", ex);}

      }

      //devuelve la clave obtenida en el path de clave configurado.
      private SecretKeySpec getCipherKey()throws BreachException{

            byte[] b = new byte[24];

            FileInputStream fis;

            try{

            fis = new FileInputStream(KEY_PATH);

            }catch(FileNotFoundException ex){
                  throw new BreachException(BreachException.SYSTEM, "No se
encontró el fichero con la clave de cifrado en la ruta: " + KEY_PATH, ex);}

            try{

            fis.read(b);

            }catch(IOException ex){
                  throw new BreachException(BreachException.SYSTEM, "Fallo al
acceder al fichero de la clave de cifrado", ex);}

            return new SecretKeySpec(b, ENCRYPTION_ALGORITHM);

      }
}
```

Clase FrameCallBackHandler

```
package securityAspect;

import javax.security.auth.callback.*;
import javax.swing.*;
```

```
import java.awt.event.*;


//Clase CallBackHandler que proporciona una interfaz grafica para la
autenticacion de usuarios.
public class FrameCallBackHandler extends JFrame implements CallbackHandler,
ActionListener{

	public FrameCallBackHandler (){

		super("Login");
		initFrame();

	}


	//flag de control de introduccion de datos
	private boolean dataEntered = false;

	//Componentes visuales
	JButton enterButton;

	public JTextField userTextfield;
	public JTextField sPassTextfield;
	public JTextField kPassTextfield;

	JTextArea mainTextarea;

	JLabel userLabel;
	JLabel sPassLabel;
	JLabel kPassLabel;

	JPanel mainPanel;
	JPanel p1, p2, p3, p4, p5;

	//intentos actuales de autenticacion
	int tries = 0;


	//inicializacion de la interfaz grafica de introduccion de datos de
autenticacion
	public void initFrame(){

		this.setSize(300,200);

		enterButton = new JButton("   Login   ");

		enterButton.addActionListener(this);

		userLabel = new JLabel("  ID Cuenta                          ");
		sPassLabel = new JLabel("  Password general       ");
		kPassLabel = new JLabel("  Password de usuario  ");

		userTextfield = new JTextField(10);
		sPassTextfield = new JTextField(10);
		kPassTextfield = new JTextField(10);

		mainTextarea = new JTextArea(4, 25);
```

```
            p1 = new JPanel();
            p1.setLayout(new BoxLayout(p1, BoxLayout.X_AXIS));
            p2 = new JPanel();
            p2.setLayout(new BoxLayout(p2, BoxLayout.X_AXIS));
            p3 = new JPanel();
            p3.setLayout(new BoxLayout(p3, BoxLayout.X_AXIS));
            p4 = new JPanel();
            p5 = new JPanel();

            mainPanel = new JPanel();
            mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.Y_AXIS));

            p1.add(userLabel);
            p1.add(userTextfield);
            p2.add(sPassLabel);
            p2.add(sPassTextfield);
            p3.add(kPassLabel);
            p3.add(kPassTextfield);

            p4.add(enterButton);

            p5.add(mainTextarea);

            mainPanel.add(p1);
            mainPanel.add(p2);
            mainPanel.add(p3);
            mainPanel.add(p4);
            mainPanel.add(p5);

            setContentPane(mainPanel);
      }

      //devuelve la entrada de datos producida por el usuario
      public void handle (Callback[] cs){

            try{

            setVisible(true);
            dataEntered = false;

            if (tries == 0) mainTextarea.setText("Introduzca datos de
autenticación");
            else mainTextarea.setText("\nAutenticación erronea\nIntroduzca de
nuevo los datos");

            while(!dataEntered){
                  Thread.currentThread().sleep(500);
            }

            ((NameCallback)cs[1]).setName(userTextfield.getText());


      ((PasswordCallback)cs[2]).setPassword(sPassTextfield.getText().toCharArray
());
```

```
      ((PasswordCallback)cs[3]).setPassword(kPassTextfield.getText().toCharArray
());

            tries++;

            userTextfield.setText("");
            sPassTextfield.setText("");
            kPassTextfield.setText("");

            setVisible(false);

            }catch(Exception ex){}

      }

      public void actionPerformed(ActionEvent ae){

            dataEntered = true;

      }

}
```

Clase BreachException

```
package securityAspect;

//Excepcion relacionada con un fallo de seguridad
public class BreachException extends Exception {

      public BreachException (int breachKind){

            super();
            kind = breachKind;

      }

      public BreachException( int breachKind, String message ){

            super(message);
            kind = breachKind;

      }

      public BreachException( int breachKind , Throwable cause){

            super(cause);
            kind = breachKind;

      }

      public BreachException( int breachKind , String message, Throwable cause){

            super(message, cause);
```

```
            kind = breachKind;

        }

        //niveles de seguridad posibles
        public static final int UNKNOWN = 0;
        public static final int TRANSGRESSION = 1;
        public static final int SYSTEM = 2;

        //nivel de seguridad por defecto
        private int kind = 0;

        //Obtiene el tipo de excepción de seguridad
        public int getKind(){

            return kind;

        }
}
```

ANEXO B: Código de la aplicación Banco

Clase Bank

```
package bank;

//Clase central de la aplicación banco
public class Bank {

	public Bank(){

		init();

	}

	//Instancia de PersistenceManager utilizada
	private PersistenceManager persistenceManager;
	//Puerto de servidor de conexiones TCP
	private final int SERVER_PORT = 5045;


	//Creacion de las instancias de PersistenceManager, BankManager y
RemoteTransactionManager
	private void init(){

		persistenceManager = new PersistenceManager();
		new BankManager(this);
		new RemoteTransactionManager(this, SERVER_PORT);

	}

	//Obtiene el saldo de la cuenta con identificacion 'accountId'
	public String getBalance( String accountId ){

		String value = persistenceManager.getValue(accountId);
		return value;

	}

	//Suma 'value' al saldo de la cuenta con id 'accountId'
	public void doTransaction( String accountId, String value ){
		setBalance(accountId,
Double.toString(Double.parseDouble(getBalance(accountId)) +
Double.parseDouble(value)));
	}

	//Establece el saldo de la cuenta con identificacion 'accountId' a
'balance'
	public void setBalance( String accountId, String balance ){
		persistenceManager.setValue(accountId, balance);
	}

	//Crea una cuenta nueva con identificacion de cuenta idAccount, password
'password'
	//y saldo 0
```

```
	public void createAccount ( String accountId, String password ) {
		persistenceManager.newDataItem(accountId);
		persistenceManager.setValue(accountId, "0");
		persistenceManager.setPassword(accountId, password);
	}

	//Devuelve true si el password 'password' es el password de la  cuenta
identificada
	//con 'accountId'
	public boolean checkPassword (String accountId, String password){
		return password.equals(persistenceManager.getPassword(accountId));
	}
}
```

Clase BankManager

```
package bank;

import javax.swing.*;
import java.awt.event.*;

//Clase de gestión local del banco mediante interfaz gráfica
public class BankManager extends JFrame implements ActionListener, Runnable {

	JButton createAccountButton;
	JButton getBalanceButton;
	JButton setBalanceButton;

	JTextField idTextfield;
	JTextField valueTextfield;

	JTextArea mainTextarea;

	JLabel idLabel;
	JLabel valueLabel;

	JPanel mainPanel;
	JPanel p1, p2, p3, p4, p5;

	Bank bank;

	ActionEvent event;

	public BankManager ( Bank bank ){

		super("Gestión del banco");
		this.bank = bank;

		initFrame();

	}

	//Inicializa los componentes gráficos del Frame
	public void initFrame(){

		this.setSize(300,200);
```

```
        createAccountButton = new JButton("   Crear cuenta   ");
        getBalanceButton = new JButton("   Obtener saldo   ");
        setBalanceButton = new JButton("  Establecer saldo  ");

        createAccountButton.addActionListener(this);
        getBalanceButton.addActionListener(this);
        setBalanceButton.addActionListener(this);

        idLabel = new JLabel("   ID Cuenta   ");
        valueLabel = new JLabel("       DATO      ");

        idTextfield = new JTextField( 10 );
        valueTextfield = new JTextField( 10 );

        mainTextarea = new JTextArea(8, 25);

        p1 = new JPanel();
        p1.setLayout(new BoxLayout(p1, BoxLayout.X_AXIS));
        p2 = new JPanel();
        p2.setLayout(new BoxLayout(p2, BoxLayout.X_AXIS));
        p3 = new JPanel();

        p4 = new JPanel();
        p5 = new JPanel();

        mainPanel = new JPanel();
        mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.Y_AXIS));

        p1.add(idLabel);
        p1.add(idTextfield);
        p2.add(valueLabel);
        p2.add(valueTextfield);

        p4.add(createAccountButton);
        p3.add(getBalanceButton);
        p3.add(setBalanceButton);
        p5.add(mainTextarea);

        mainPanel.add(p1);
        mainPanel.add(p2);
        mainPanel.add(p3);
        mainPanel.add(p4);
        mainPanel.add(p5);

        setContentPane(mainPanel);

        setVisible(true);

    }

    //Comienzo de un hilo para el procesamiento de una acción
    public void run(){

        String action = Thread.currentThread().getName();
        if (action.equals("createAccount")) createAccount();
        if (action.equals("getBalance")) getBalance();
        if (action.equals("setBalance")) setBalance();
```

```
	}

	//Recoje los eventos producidos en los botones de acción
	public void actionPerformed( ActionEvent ae ){

		if (ae.getSource().equals(this.createAccountButton)){
			new Thread(this, "createAccount").start();
		}
		if (ae.getSource().equals(this.getBalanceButton)){
			new Thread(this, "getBalance").start();
		}
		if (ae.getSource().equals(this.setBalanceButton)){
			new Thread(this, "setBalance").start();
		}
	}

	//Solicita a la instancia de Bank la creación de una cuenta
	private void createAccount(){

		String id = this.idTextfield.getText();
		String password = this.valueTextfield.getText();
		bank.createAccount(id, password);
		this.mainTextarea.setText( "Se creó la cuenta: " + id);

	}

	//Solicita a la instancia de Bank el saldo de una cuenta
	private void getBalance(){

		String id = this.idTextfield.getText();
		String balance = bank.getBalance(id);
		this.mainTextarea.setText( "El saldo de " + id + " es: " + balance);

	}

	//Solicita a la instancia de Bank adjudicar un nuevo valor de saldo a una
cuenta
	private void setBalance(){

		String id = this.idTextfield.getText();
		String value = this.valueTextfield.getText();
		bank.setBalance(id, value);
		this.mainTextarea.setText( "El saldo de " + id + " pasa a ser: " +
value);

	}

}
```

Clase RemoteTransactionManager

```
package bank;

import java.net.*;
import java.util.ArrayList;
```

```
import java.io.*;

//Gestiona transacacciones remotas por conexión TCP
public class RemoteTransactionManager implements Runnable{

	public RemoteTransactionManager ( Bank bank, int serverPort ){
		this.bank = bank;
		this.serverPort = serverPort;
		(new Thread(this)).start();
	}


	//Referencia a la instancia de Bank
	private Bank bank;
	//Puerto de funcionamiento
	int serverPort;



	//El servidor de sockets es inicializado y se mantiene a la espera de
conexiones entrantes
	//en un bucle infinito.
	public void run(){

		ServerSocket ss = null;
		ArrayList socketList = new ArrayList();

		try{
		ss = new ServerSocket(serverPort);
		}catch(Exception ex){
			System.out.println("No se pudo crear ServerSocket en " +
serverPort);
			ex.printStackTrace();
			return;


		}
		System.out.println("Servidor inicializado");

		while(true){
			try{
			Socket s = ss.accept();
			new SessionAssistant( bank, s );
			System.out.println("Se estableció una conexión");

			}catch(Exception ex){
				System.out.println("Fallo accept: " + ex.toString());
				ex.printStackTrace();
				break;
			}

		}

	}

	//Gestiona la sesión asociada a la conexión particular de cada usuario
	private class SessionAssistant implements Runnable{
```

```
		public SessionAssistant( Bank bank, Socket socket ){
			this.bank = bank;
			this.socket = socket;
			init();
		}

		//Referencia a la instanica de Bank
		Bank bank;
		//Socket de conexión con el usuario
		Socket socket;

		BufferedReader input;
		PrintWriter output;

		//id de la cuenta del usuario que inició la sesión.
		String accountId;

		//Inicia el procesamiento de la sesión de usuario
		private void init(){

			new Thread(this).start();

		}

	//Gestión del inicio de conexión y tramitatación de las solicitudes de
acción
	public void run(){

		try{
			input = new BufferedReader (new InputStreamReader
(socket.getInputStream()));
			output = new PrintWriter(socket.getOutputStream(), true);
			System.out.println("Esperando inicio de sesión");
			String initSessionString = input.readLine();
			System.out.println("Se recibió: " + initSessionString);
			String[] s = initSessionString.split("#");

			if (s[0].equals("SESSION") && s.length == 3){
				if (!bank.checkPassword(s[1], s[2])){
					output.write("FAILED" + "\n");
					System.out.println("Se envió FAILED" );
					return;
				}
				else{
					accountId = s[1];
					output.println("OK" + "\n");
					System.out.println("Se envió OK");
				}
			}
			}catch(Exception ex){ex.printStackTrace();}

		while(true){
			try{
				String remoteActionString = input.readLine();
				System.out.println("Se recibió: " +
remoteActionString);
				String[] s = remoteActionString.split("#");
				if (s[0].equals("TRANSACTION") && s.length == 2){
```

```
                                bank.doTransaction( accountId, s[1]);
                                output.println("OK" + "\n");
                                System.out.println("Se envió OK");
                                continue;
                        }else if (s[0].equals("GET_BALANCE") && s.length == 1){
                                String value = bank.getBalance(accountId);
                                output.println(value + "\n");
                                System.out.println("Se envió" + value);
                                continue;
                        }else if (s[0].equals("END_SESSION") && s.length == 1){
                                System.out.println("Se cerró la sesión");
                                break;
                        }
                        else output.println("UNKNOWN ACTION" + "\n");

                }catch(Exception ex){ex.printStackTrace();}

        }
    }

    }

}
```

Clase PersistenceManager

```
package bank;

import java.sql.*;

//Clase que se encarga de la persistencia de datos por medio del acceso a una
//base de datos
public class PersistenceManager {

    //Conexión con la base de datos
    private Connection conn;

    //Cadena de connexion
    private final String CONNECTION_STRING = "Cadena_de_conexión";

    public PersistenceManager(){

        connect();

    }

    //Realiza la conexión con las base de datos
    public void connect(){

        try{

        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver);
        conn = DriverManager.getConnection(CONNECTION_STRING);

        }catch(ClassNotFoundException cnfex){
        System.out.println("ASDAS: " + cnfex.toString());
```

```
			}catch(SQLException sqlex){
				System.out.println("ASDAS: " + sqlex.toString());
			}

		}

		//Obtiene de la base de datos el saldo de la cuenta con identificación de
cuenta 'id'
		public String getValue(String id){

			try{
				return conn.createStatement().executeQuery("SELECT BALANCE
FROM ACCOUNTS WHERE id='"+id+"'").getString("balance");
			}catch(SQLException sqlex){
				System.out.println("Error al acceder a la base de datos:" +
sqlex.toString());
			}
			return "error";

		}

		//Obtiene de la base de datos el password de la cuenta con identificación
de cuenta 'id'
		public String getPassword(String id){

			try{
				return conn.createStatement().executeQuery("SELECT PASSWORD
FROM ACCOUNTS WHERE id='"+id+"'").getString("password");
			}catch(SQLException sqlex){
				System.out.println("Error al acceder a la base de datos:" +
sqlex.toString());
			}

			return "error";

		}

		//Crea una nueva cuenta con identificación 'id' y password y valor vacios
		public void newDataItem(String id){

			try{
			conn.createStatement().executeUpdate("INSERT INTO ACCOUNTS VALUES
('"+id+"',' ',' ')");
			}catch(SQLException sqlex){
				System.out.println("Error al acceder a la base de datos:" +
sqlex.toString());
			}

		}

		//Establece el valor de la cuenta con identificación 'id' a 'value'
		public void setValue(String id, String value){

			try{
				conn.createStatement().executeQuery("UPDATE ACCOUNTS SET
BALANCE = '"+value+"' WHERE ID LIKE '"+id+"'");
			}catch(SQLException sqlex){
```

```
                        System.out.println("Error al acceder a la base de
datos:" + sqlex.toString() + " valor: " + value);
                  }

      }

      //Establece el password de la cuenta con identifiación 'id' a 'password'
      public void setPassword(String id, String password){

            try{
                  conn.createStatement().executeQuery("UPDATE ACCOUNTS SET
PASSWORD = '"+password+"' WHERE ID LIKE '"+id+"'");
            }catch(SQLException sqlex){
                        System.out.println("Error al acceder a la base de
datos:" + sqlex.toString() + " valor: " + password);
                  }

      }

}
```

ANEXO C: Implementación de Seguridad para la Aplicación Banco

Clase aspecto BankSensibleData

```
package bankSecurity;

import org.aspectj.lang.*;

import bank.*;
import securityAspect.*;

//Aspecto de proteccion de datos sensibles para la aplicacion banco
public aspect BankSensibleData extends SensibleData {

	//Establece la ruta de la clave de cifrado
	protected void init(){
		KEY_PATH = "C:\\PFC\\cipherKey";
	}

	//Captura las llamadas a los métodos de inserción de datos de la clase que
	//que gestiona la base de datos 'PersistenceManager' obteniedo los
argumentos
	public pointcut settingSensibleData (String sensibleData, String
referral):
		(call ( void PersistenceManager.setValue( String, String ) )
				|| call ( void PersistenceManager.setPassword( String,
String ) ))
				&& args(referral, sensibleData);

	//Captura las llamadas a los métodos de obtencion de datos de la clase que
	//que gestiona la base de datos 'PersistenceManager' obteniedo los
argumentos
	public pointcut accedingSensibleData (String referral):
		(call ( String PersistenceManager.getValue( String ) )
				|| call ( void PersistenceManager.getPassword( String )
) )
				&& args (referral);

	//Si ocurre un error en el procesamiento de datos el hilo de ejecucion es
parado
	protected void dataProcessFailed( JoinPoint jp ){
		Thread.currentThread().stop();
	}

	//Si ocurre un error en el procesamiento de datos devido a un posible dato
corrupto
	//el hilo de ejecucion es parado
	protected String dataCorrupted( JoinPoint jp ){
		Thread.currentThread().stop();
		return "";
	}
```

```
}
```

Clase aspecto BankCommunications

```
package bankSecurity;

import securityAspect.*;

//Aspecto de seguridad de comunicaciones para la aplicacion banco
public aspect BankCommunications extends Communication {

	//Establece las rutas de los almacenes de claves y los passwords
convenientes
	protected void init(){

	SELF_KEYS_PATH = "C:\\PFC\\selfkeystore";
	TRUST_KEYS_PATH = "C:\\PFC\\trustkeystore";
	TRUST_KEYSTORE_PASSWORD = "storePassword";
	SELF_KEYSTORE_PASSWORD = "storePassword";
	SELF_KEYS_PASSWORD = "keyPassword";

	}


}
```

Clase aspecto BankAuth

```
package bankSecurity;

import securityAspect.*;
import bank.*;
import org.aspectj.lang.*;

//Aspecto de autenticación para la aplicación Banco
public aspect BankAuth extends Authentication {

	//la precedencia es
	declare precedence: SensibleData, Logging, Communication, Authentication;

	//Será provocada la autenticacion en las llamadas a los metodos setBalance
	//y createAccount por parte de la aplicación de  administrador.
	public pointcut doUserLogin() : within (BankManager)
					&& ( call (void
Bank.setBalance(String, String))
					|| call (void
Bank.createAccount(String, String)));

	//Si el proceso de autenticacion resulta fallido se para el hilo que
llevaba
```

```
	//acabo la accion.
	protected void loginFailed( JoinPoint jp ){
		Thread.currentThread().stop();
	}
```

Clase aspecto BankLogging

```
package bankSecurity;

import bank.*;
import securityAspect.*;

//Aspecto de registro de sucesos para la aplicacion banco
public aspect BankLogging extends Logging {

	protected void init(){

	}

	//Captura las llamadas a los métodos de la clase Bank desde la clase de
gestion
	//de administrador 'BankManager'
	public pointcut adminAction(): call(* Bank.*(..))
							&& within( BankManager
);

	//Captura las llamadas a los metodos getBalance y doTransaction producidas
	//producidas por mensajes de usuarios remotos en la clase
RemoteTransactionManager
	public pointcut userAction(): (call(String Bank.getBalance()) ||
							call(String
Bank.doTransaction()) )
							&& !within(
RemoteTransactionManager );

	//Captura las llamadas a los métodos de inserción de datos de la clase que
	//que gestiona la base de datos 'PersistenceManager'
	public pointcut settingSensibleData ():
			call ( void PersistenceManager.setValue( String, String
) )
			|| call ( void PersistenceManager.setPassword( String,
String ) );

	//Captura las llamadas a los métodos de obtencion de datos de la clase que
	//que gestiona la base de datos 'PersistenceManager'
	public pointcut accedingSensibleData ():
			call ( String PersistenceManager.getValue( String ) )
			|| call ( void PersistenceManager.getPassword( String )
);

}
```

Referencias

[1]H.M Deitel, P.J. Deitel. Java how to program. Prentice Hall. 2002.

[2] Ramnivas Laddad. AspectJ In Action. Manning. 2003

[3]G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, Aspect-Oriented Programming, Xerox Palo Alto Research Center, 1997

[4] http://www.aspectcj.org

[5] Ivan Kiselev. Aspect-Oriented Programming with AspectJ.Sams.2002

[6] Marco Pistotia, Duane F. Reller, Deepak Gupta, Milind Nagnur, Ashok K. ramani. Java 2 Network Security. IBM. 1999

[7] http://java.sun.com/docs/books/tutorial/security1.2/overview

[8] http://www.iec.csic.es/criptonomicon/java/

[9] http://www.uv.es/sto/cursos/seguridad.java/html/sjava.html

[10] Minhuan Huang, Chunlei Wang, Lufeng Zhang. Toward a Reusable and Generic Security Aspect Library. AOSDSEC 2004

[11] Bart Vanhauten, Bart de win. AOP, Security and Generity.
http://www.cs.kuleuven.ac.be/~bartd/publications.html

[12] Art Tylor, Brian Buege, Randy Laiman. Hacking Exposed J2EE & Java. Mc Graw Hill. 2002

[13] http://www-128.ibm.com/developerworks/java/library/j-aopwork3/