UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

REGISTRO DINÁMICO DE BITÁCORAS APLICANDO PROGRAMACIÓN ORIENTADA A ASPECTOS

T E S I S

QUE PARA OBTENER EL TÍTULO DE

INGENIERO EN COMPUTACIÓN

PRESENTA:

LUIS RAMÓN CASILLAS PÉREZ SOTO

DIRECTOR DE TESIS:
ING. ORLANDO ZALDÍVAR ZAMORATEGUI

CIUDAD UNIVERSITARIA NOVIEMBRE DE 2013

Agradecimientos

A mi madre, por haberme formado como persona y siempre estar dispuesta a apoyarme en las buenas y en las malas.

A mi abuelita Josefina, por cuidar de mi. Que aunque ya no esté conmigo, siempre la recuerdo con mucho cariño

A mis hermanas Mariana y Ana, por siempre estar a mi lado apoyándome.

Al Ingeniero Orlando Zaldívar Zamorategui, por su apoyo y consejos, sin los cuales este trabajo no hubiera sido posible.

Al Programa de tecnología en computo de la Facultad de Ingeniería, por haber dado un impulso muy importante a mi carrera y permitirme conocer gente muy valiosa.

Y sobre todo a la Universidad Nacional Autónoma de México y en particular a la Facultad de Ingeniería, siempre estaré en deuda. A ellas les debo lo que soy ahora.

Índice

Introducción **4**

Marco teórico **9**

- **Arquitectura de software** **10**
  - Vistas y estructuras arquitectónicas 11
  - Importancia de la arquitectura 13
- **Patrones de diseño** **18**
  - ¿Qué es un patrón de diseño? 18
  - Problemas que ayudan a resolver los patrones de diseño 19
  - Patrones de diseño creacionales 22
  - Patrones de diseño estructurales 27
  - Patrones de diseño funcionales 32
  - Patrones GRASP 37
- **Programación orientada a aspectos** **43**
  - Definición 44
  - Conceptos 45
  - Implementaciones 47
  - ¿Cómo afecta a una arquitectura el uso de POA? 48
- **Spring Framework** **51**
  - ¿Qué es Spring? 52
  - Módulos 54

Planteamiento del problema **59**

- **Antecedentes** **59**
- **El problema de la implementación del logging dentro de un sistema** **60**

Sistema de bitácoras no invasivo **64**

- **Determinación de objetivos** **65**
- **Análisis de riesgos** **66**
- **Planificación** **72**
- **Desarrollo** **77**
  - Análisis 78
  - Diseño 85
  - Implementación 101
  - Pruebas 115

Resultados **128**

Impacto **136**

Conclusiones **140**

Bibliografía **143**

Mesografía **144**

1

Introducción

Cualquier sistema puede ser visto como un conjunto de conceptos que se implementan y combinan para lograr un objetivo específico. Un sistema típico puede estar compuesto por muchos tipos de conceptos, como pueden ser lógica de negocio, persistencia, presentación, seguridad, gestión de errores, performance, logging, etc.

Dichos conceptos son resultado del análisis de requerimientos, en el cual cada uno es separado para su posterior análisis y diseño. Éstos pueden ser clasificados como conceptos de negocio y de utilidad, estos últimos en general son mutuamente excluyentes, es decir son independientes entre sí y además, son independientes de los conceptos de negocio.

Sin embargo se puede dar el caso de que varios conceptos de negocio dependan directamente de un concepto de utilidad. Este tipo de conceptos que abarcan varios componentes o conceptos, los llamaremos conceptos transversales (Figura 1.1).

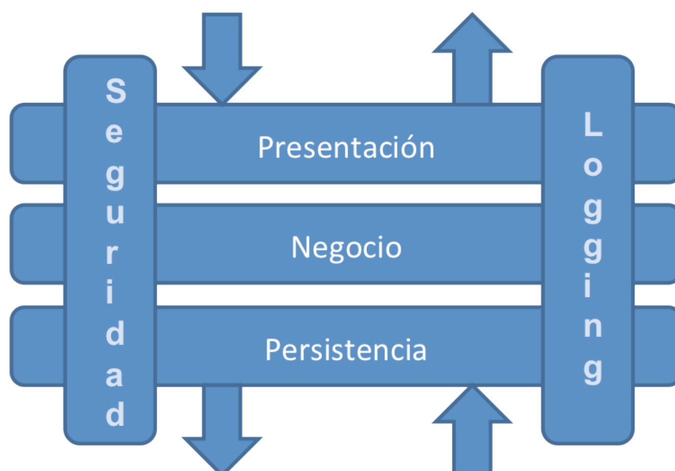

Figura 1.1 Conceptos que se propagan por varios componentes del sistema.

Por ejemplo, un sistema puede tener conceptos transversales como:

- Autenticación.
- Logging.
- Pooling de recursos.
- Administración.
- Transaccionalidad.
- Gestión de errores.

Este tipo de conceptos si no se diseñan adecuadamente pueden causar problemas en la etapa de desarrollo que dificultan el mismo, así como el mantenimiento y evolución del sistema. Los problemas que se pueden presentar son:

- Mezcla de conceptos: Los componentes desarrollados implementan varios conceptos de manera simultánea.
- Dispersión de código: La implementación de un concepto se encuentra dispersa en varios componentes del sistema.

Lo cual provoca:

- Dificultad en el desarrollo debido a que se mezclan conceptos en un mismo componente.
- Menor reutilización de código, ya que es más complicado reutilizar un componente que implementa múltiples conceptos.
- Menor calidad de código.
- Dificultad en el mantenimiento y evolución del componente debido a que en la implementación del mismo no están correctamente separados los conceptos. Implementar requerimientos futuros muchas veces implica rehacer el componente.

Un concepto transversal muy común en un sistema es el logging.

El logging es el registro persistente de eventos. Dichos registros contienen información acerca de los sucesos ocurridos en el sistema, tales como, ejecución exitosa de procesos, entrada de datos, errores, información de depuración, etc. Los registros generados pueden ser gestionados de diversas maneras, como puede ser mediante archivos de texto plano o en una base de datos.

En la actualidad todo sistema debe de complementar sus funciones básicas con la implementación de un subsistema de logging, que le permita llevar un historial confiable y oportuno de los eventos ocurridos en él, en determinado rango de tiempo.

El logging es muy importante por las ventajas que brinda al sistema que lo implementa:

- Permite llevar un historial de eventos del sistema.
- Facilita la labor de desarrollo ya que registra mensajes de depuración que dan una idea acerca del estado que guarda en un cierto momento el sistema.
- Permite detectar y corregir errores oportunamente en un ambiente productivo.
- Se pueden detectar y verificar posibles incidentes de seguridad.
- Útil en tareas de cómputo forense.

Sin embargo su implementación trae consigo algunos problemas:

- Genera una carga extra al sistema, lo cual repercute en un rendimiento menor del mismo.
- Puede producir gran cantidad de datos, los cuales pueden ser difíciles de procesar y mantener.
- Dependiendo de la implementación existe el riesgo de "ensuciar" el código fuente del sistema.

Para poder implementar correctamente el sistema es necesario separar los componentes básicos de los conceptos transversales. Existen varias propuestas de solución:

- Patrones de diseño: Mediante la aplicación de soluciones probadas y documentadas se logran separar los conceptos en un sistema de manera adecuada, con patrones como Delegate, Template, Proxy, etc.
- Soluciones específicas de dominio: Los ambientes en los cuales se ejecutan las aplicaciones, tales como frameworks y servidores de aplicaciones, se encargan de abstraer y modularizar los conceptos transversales. Por ejemplo, los servidores de aplicaciones modularizan conceptos como seguridad, persistencia y administración.

La solución más común y sencilla para implementar un sistema de logging es mediante una biblioteca. Este tipo de biblioteca es un conjunto de clases que brinda servicios que facilitan el registro de logs mediante el patrón de diseño Delegate. Por lo que los módulos, que para completar sus operaciones requieran el registro de logs, solamente tienen que delegar esta tarea a la biblioteca especializada.

Como se mencionó entre los posibles problemas que puede generar un subsistema de logging, está el hecho de que puede "ensuciar" el código, lo cual ocurre muy frecuentemente cuando se utilizan estas bibliotecas. Debido a que el sistema base para crear un registro en la bitácora requiere forzosamente hacer un llamado a alguno de los servicios que ésta brinda para dicho fin.

Por lo anterior es muy común que el código fuente se vaya llenando paulatinamente de un número importante de llamados a los servicios de logging, según se va requiriendo para tareas tales como depuración o detección de errores.

Esto da como resultado que los componentes del sistema se encuentren muy relacionados a la biblioteca de logging. Por lo que se puede decir que dichos componentes no son reutilizables en otro contexto, es decir, se limita su uso en otros sistemas.

En la actualidad el paradigma elegido para el diseño y desarrollo de la mayoría de los proyectos nuevos es el orientado a objetos debido a la facilidad que brinda para modelar conceptos comunes. Sin embargo, este paradigma no cumple cabalmente con los requerimientos cuando se desea modelar una funcionalidad que abarca muchos componentes, lo que se ha definido como conceptos transversales.

A través de este trabajo se hace un análisis de los aspectos más destacados de la arquitectura de un sistema y cómo es que los conceptos transversales la afectan.

En el capítulo 2, Marco teórico, se realiza un análisis de los patrones de diseño más comunes, revisando los problemas que ayudan a solventar y las ventajas y desventajas de su aplicación exitosa. También se toca el tema de la programación orientada a aspectos (POA). Este tema trata otro enfoque que permite implementar un subsistema de logging que tiene la particularidad de que no es invasivo y mantiene un código limpio ya que no afecta en lo absoluto el código fuente de la aplicación.

En esta tesis la programación orientada a aspectos se logra utilizando la implementación provista por Spring. Como se trata en capítulos posteriores, este framework tiene la finalidad de facilitar el desarrollo de aplicaciones empresariales en la plataforma Java.

En el capítulo 3, Planteamiento del problema, se hace un análisis de la complejidad que se introduce en el código de una aplicación al implementar conceptos transversales con el paradigma orientado a objetos.

En el capítulo 4, Sistema de bitácoras no invasivo, utilizando los conceptos expuestos en el marco teórico, se propone la construcción de un sistema de bitácoras no invasivo. Este sistema es desarrollado utilizando la implementación POA de Spring. En este capítulo se detallan los objetivos del proyecto, el análisis

de los riesgos, la planificación y el desarrollo de proyecto. En la sección referente al desarrollo del proyecto se mencionan las siguientes etapas del proyecto:

- Análisis: Se establecen los principales conceptos utilizados durante el proyecto (glosario), se definen y analizan los requerimientos y con base en ellos se establecen los casos de uso del sistema.
- Diseño: Se establece la arquitectura del sistema, se desarrolla el diagrama entidad relación de la base de datos a utilizar, se especifica el diagrama de clases y los diagramas de secuencias.
- Implementación: Se detalla el proceso de construcción del proyecto, las herramientas utilizadas, la estructura de los módulos y paquetes, así como el despliegue del sistema en el servidor de aplicaciones.
- Pruebas: Se diseñan e implementan las pruebas unitarias a realizar a los componentes desarrollados, así como las pruebas de integración del sistema.

En el capítulo 5, Resultados, se hace el análisis del resultado del desarrollo especificado en capítulos anteriores. Se verifican y validan los requerimientos definidos inicialmente.

Posteriormente en el capítulo 6, Impacto, se analiza cómo es que se afecta positivamente la arquitectura de cualquier sistema target al utilizar el sistema de bitácoras desarrollado.

Por último en el capítulo 7, Conclusiones, se mencionan los beneficios que aporta al desarrollo de un sistema, la implementación del Sistema de bitácoras no invasivo.

2

Marco teórico

2.1

Arquitectura de software

La arquitectura de software de un sistema es la estructura o estructuras que conforman dicho sistema, estructuras que se componen de elementos de software, las propiedades externamente visibles de estos elementos y las relaciones entre ellos.

La arquitectura define elementos de software, contiene información acerca de cómo los elementos se relacionan entre ellos, y se omite intencionalmente información que no tiene relación con las interacciones de dichos elementos. Por lo tanto, una arquitectura es principalmente una abstracción de un sistema que omite detalles de los elementos que no afectan el cómo se usan, son usados, se relacionan o interactúan con otros elementos.

En la mayoría de los sistemas actuales los elementos de software interactúan entre ellos a través de interfaces, lo cual separa los detalles del mismo en su parte pública y privada. La arquitectura se enfoca en la parte pública de esta separación. La parte privada está enfocada más a los detalles de implementación.

Un sistema puede estar compuesto por más de una estructura. Cualquier proyecto no trivial está separado en unidades de implementación, a estas unidades se les asignan responsabilidades específicas que, frecuentemente, son la base de la asignación de trabajo a los equipos de desarrollo. Este tipo de estructura es usualmente utilizada para describir un sistema, es estática ya que se enfoca en la manera en que la funcionalidad del sistema es dividida.

Otras estructuras están más enfocadas en la manera cómo los elementos interactúan entre ellos en tiempo de ejecución, con el objetivo de lograr la función del sistema.

La arquitectura consiste en las estructuras mencionadas.

Todo sistema computacional tiene una arquitectura de software de manera inherente, debido a que cada uno puede ser descrito con los elementos que lo componen y las relaciones entre ellos. En el caso más simple un sistema se compondría de un solo elemento, tal vez no muy útil y también difícil de entender, pero finalmente es una arquitectura.

Aún cuando todo sistema tenga una arquitectura, no necesariamente ésta es conocida. Esto puede ser debido a que su documentación no fue desarrollada.

Lo anterior nos indica la diferencia entre arquitectura y su representación, la arquitectura existe independientemente de su descripción o especificación.

El comportamiento de cada elemento es parte de la arquitectura en la medida en que este comportamiento puede ser observado o inferido desde el punto de vista de otro elemento. Dicho comportamiento es lo que permite a los elementos interactuar entre ellos.

Vistas y estructuras arquitectónicas

Puede ser muy complicado entender un sistema por completo utilizando una sola descripción de su arquitectura. Lo anterior debido a la complejidad de los sistemas actuales. Para resolver esta problemática se limitará la atención sobre la arquitectura, a una o un pequeño número de estructuras que conforman el sistema. Para comunicar significativamente una arquitectura se debe tener claro qué estructura o estructuras se están analizando, es decir, qué vista de la arquitectura se está tomando.

Las vistas son representaciones de la totalidad de una arquitectura, que son significativas para uno o más interesados en el sistema. El arquitecto comúnmente elige y desarrolla un conjunto de vistas que permitirán a la arquitectura ser comunicada y entendida por los interesados y les permitirá a éstos verificar que el sistema cumple con los requerimientos.

En general, una arquitectura es representada por uno o más modelos arquitectónicos que juntos proveen una descripción coherente de la arquitectura del sistema. Un único modelo generalmente es muy complicado de ser entendido y comunicado de forma detallada. Normalmente es necesario desarrollar múltiples vistas de una arquitectura.

Una estructura es un conjunto de elementos tal cual existen en software o hardware. Por ejemplo, una estructura de módulos es el conjunto de módulos de un sistema y su organización.

Una vista de módulos es una representación de esa estructura, tal como se documenta y es usada por los interesados en el sistema.

Las estructuras arquitectónicas pueden ser clasificadas en tres grupos dependiendo de la naturaleza de los elementos que muestran:

- Estructuras de módulos: Los elementos de esta estructura son módulos, los cuales son unidades de implementación. Se les asignan responsabilidades funcionales. Permiten responder preguntas como: ¿Cuál es la responsabilidad primaria asignada a cada módulo? ¿Qué otros

elementos tiene permitido usar el módulo? ¿Cómo está estructurado el sistema como un conjunto de módulos? En la Figura 2.1.1 se muestra una estructura de módulos.

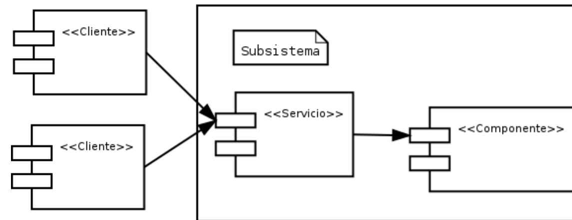

Figura 2.1.1 Estructura de módulos.

- Estructura componente-conector: Los elementos de esta estructura (Figura 2.1.2) son componentes de tiempo de ejecución, y conectores (medios de comunicación entre componentes). Esta estructura permite visualizar:
  - Los principales componentes de ejecución y cómo interactúan.
  - Los principales almacenes de datos compartidos.
  - Las partes del sistema que están replicadas.
  - El flujo de datos dentro del sistema.
  - Las partes del sistema que pueden ejecutarse en paralelo.

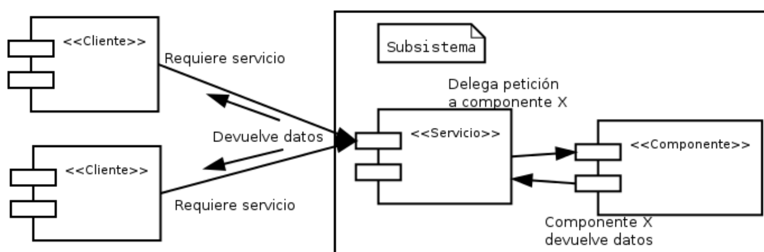

Figura 2.1.2 Estructura componente – conector.

- Estructura de asignación: Muestran la relación entre los elementos de software y otro tipo de elementos en uno o más ambientes externos, en los cuales el software es creado y ejecutado (Figura 2.1.3). Muestra cómo el sistema se relaciona con estructuras que no son software (CPUs, sistemas de archivos, redes, equipos de desarrollo, etc).

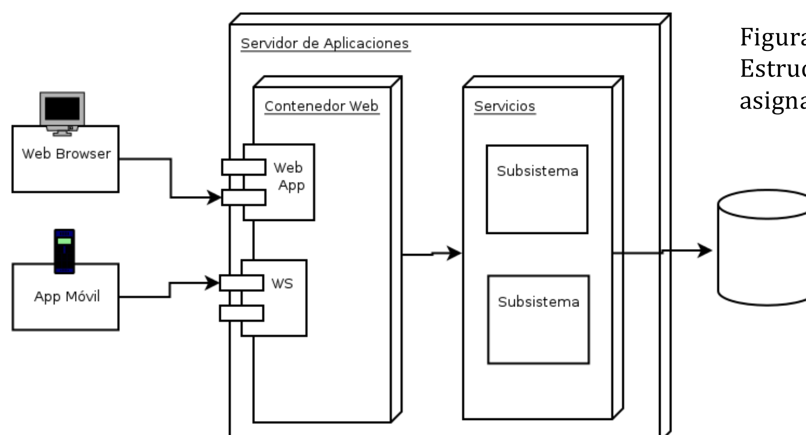

Figura 2.1.3 Estructura de asignación.

Importancia de la arquitectura

Existen tres razones fundamentales por las cuales es importante una arquitectura: Comunicación con interesados en el sistema, decisiones tempranas de diseño y reúso de la abstracción de un sistema.

Comunicación con interesados en el sistema

Cada interesado en un sistema (cliente, usuario, administrador de proyecto, programador, tester, etc.) está enfocado en una característica específica del sistema que es afectada por la arquitectura. Por ejemplo, al usuario le importa que el sistema sea confiable y esté disponible cuando lo necesita, al administrador de proyecto le preocupan los tiempos de entrega y gestionar los equipos de desarrollo con el fin de que puedan trabajar de manera independiente.

El arquitecto se preocupará en desarrollar las estrategias que lleven a conseguir estas metas.

La arquitectura provee un lenguaje común en el cual diferentes aspectos del sistema pueden ser expresados, negociados y resueltos en una etapa del desarrollo adecuada. Sin este lenguaje puede ser muy complicado comprender un sistema lo suficiente como para tomar decisiones que influyan tanto en requerimientos funcionales como no funcionales.

Decisiones tempranas de diseño

La arquitectura de software representa un conjunto de decisiones de diseño. Las decisiones tempranas de diseño son las más difíciles de tomar y las más complicadas de modificar en etapas posteriores del desarrollo.

La arquitectura define restricciones en la implementación
Una implementación manifiesta una arquitectura si ésta cumple con las decisiones de diseño descritas por la arquitectura. Esto quiere decir que la implementación tiene que estar dividida en los elementos establecidos, los cuales deben interactuar entre ellos de la manera establecida y cada uno cumplirá con su responsabilidad tal cual lo dicta la arquitectura.

Estas restricciones permiten una separación de conceptos que favorece un uso óptimo de los recursos humanos y computacionales.

La arquitectura dicta la estructura organizacional
La arquitectura no sólo establece la estructura del sistema en desarrollo, si no que se convierte en la base de la estructura del proyecto de desarrollo y algunas veces, de la organización entera.

La metodología común para dividir las tareas en un sistema de gran tamaño, es asignar a diferentes equipos, diferentes porciones del sistema a construir. Debido a que la arquitectura del sistema incluye la descomposición de alto nivel del mismo, es común que la arquitectura sea usada como la base de la división de tareas.

Los equipos de desarrollo se comunican entre ellos en términos de las interfaces de los elementos principales de la arquitectura.

Sin embargo, este enfoque puede provocar que algunos aspectos de la arquitectura deban permanecer "congelados". Ya que modificar las responsabilidades asignadas a un equipo a través de la división de tareas (basada en la arquitectura), puede ser muy costoso, tanto para la gestión de recursos como para la estructura organizacional.

Ésta es una de las razones por las cuales se debe evaluar detalladamente la arquitectura de software de un sistema antes de congelarla.

La arquitectura inhibe o favorece los requerimientos no funcionales de un sistema
La arquitectura determina si un sistema cumplirá con los requerimientos no funcionales de un sistema deseados o solicitados.

Algunos de los requerimientos no funcionales que puede presentar un sistema son los siguientes:

Requerimientos manifiestos: Son requerimientos que son obvios para el usuario

- Rendimiento: Métrica del tiempo de respuesta desde el punto de vista del usuario.
- Confiabilidad: Métrica de la probabilidad de que los datos sean correctos, tanto en el almacenamiento persistente como en los resultados presentados.
- Disponibilidad: Métrica de la probabilidad de que el sistema está operando en el momento que un usuario lo requiera y que el mismo continúe operando por lo menos hasta que se complete la petición del usuario.
- Usabilidad: Métrica de la facilidad de un usuario para entender y utilizar un sistema para llevar a cabo una tarea en él.

Requerimientos operacionales: Son los requerimientos que son evidentes cuando el sistema está en ejecución, pero no son inmediatamente evidentes para un usuario.

- Throughput: Métrica de la cantidad de trabajo que puede ser ejecutada en un cierto tiempo.
- Capacidad de gestión: Métrica de la cantidad de intervención humana requerida para mantener el sistema operando adecuadamente.
- Seguridad: Métrica de la confiabilidad de la protección de los datos de ser consultados o modificados de manera no autorizada.
- Capacidad de servicio: La métrica de la cantidad de trabajo que es requerido para realizar un mantenimiento fuera de rutina.
- Capacidad de prueba: La métrica de la cantidad de trabajo requerido para identificar y aislar una falla o un error en el sistema.

Requerimientos de desarrollo: Afectan cómo el sistema es construido. Estos requerimientos se enfocan en el proyecto mientras está en la etapa de desarrollo y se vuelven irrelevantes cuando el sistema evoluciona.

- Capacidad de realización: La métrica de la probabilidad de la implementación exitosa de una característica o producto.
- Capacidad de planeación: La métrica de la confianza en un plan y estimaciones de costos.

Requerimientos evolutivos: Requerimientos que se relacionan en cómo el sistema se comporta cuando es alterado o actualizado.

- Escalabilidad: La métrica inversa del trabajo y costo requerido para modificar el sistema para proveer una mayor capacidad de carga (mayor throughput).
- Mantenibilidad: La métrica inversa del trabajo necesario para realizar la corrección de errores.
- Extensibilidad: La métrica inversa del trabajo y costo necesarios para agregar nuevas características al sistema.
- Flexibilidad: La métrica inversa del trabajo y costo necesario para realizar modificaciones a las características actuales del sistema.
- Reusabilidad: La métrica de la probabilidad de que, dado un nuevo proyecto con requerimientos similares, una parte importante del sistema pueda ser integrada exitosamente al nuevo sistema.
- Portabilidad: La métrica inversa del trabajo y costo necesario para migrar componentes a una nueva plataforma.

Las estrategias para alcanzar estos requerimientos son principalmente arquitectónicos, sin embargo una arquitectura por sí sola no puede garantizar funcionalidad o calidad. Un diseño o una implementación pobre puede echar abajo una arquitectura en un principio adecuada. Para asegurar la calidad de un sistema una buena arquitectura es necesaria, pero no suficiente.

La arquitectura puede predecir cualidades del sistema

Es posible realizar predicciones de la calidad del sistema basados únicamente en una evaluación de su arquitectura. De otra forma no habría manera de saber si se

han tomado las decisiones adecuadas en la arquitectura del sistema. Sin esta evaluación la selección de una arquitectura sería complicada y estaría basada en una elección aleatoria.

La arquitectura facilita la gestión de cambios
En promedio el 80% del costo de un sistema ocurre después de su primera liberación. Los sistemas cambian constantemente durante su tiempo de vida y frecuentemente la realización de estos cambios es complicada.

Cada arquitectura clasifica los posibles cambios en tres categorías: locales, no locales y arquitectónicos. Un cambio local implica la modificación de un único elemento. Un cambio no local requiere la modificación de múltiples elementos, pero la arquitectura permanece intacta. Un cambio arquitectónico modifica la manera fundamental en que los elementos interactúan entre ellos y probablemente requerirá cambios en todo el sistema.

Los cambios locales son siempre los más deseables por su facilidad de realización. Una arquitectura efectiva es aquella en la cual los cambios más probables son los más sencillos de realizar.

Además, la arquitectura también ayuda a decidir cuándo una modificación es esencial, qué modificaciones en el sistema tienen menos riesgo, establecer prioridades y a evaluar las consecuencias de los cambios propuestos.

La arquitectura ayuda en la creación de prototipos evolutivos
Una vez que una arquitectura ha sido definida, ésta puede ser analizada y prototipada como el esqueleto del sistema. Esto ayuda al proceso de desarrollo:

- ❖ El sistema puede estar en funcionamiento en etapas más tempranas del ciclo de desarrollo, y conforme avanza el proceso cada uno de los elementos-prototipo van siendo sustituidos por implementaciones intermedias o finales.
- ❖ Tener un prototipo en etapas más tempranas del desarrollo puede ayudar a identificar posibles problemas de rendimiento de manera anticipada.

Lo anterior ayuda a mitigar los riesgos en el proyecto.

La arquitectura facilita la estimación de costos y de tiempos
La estimación de costos y tiempos es de suma importancia para el administrador de un proyecto ya que le permitirá adquirir los recursos necesarios y a identificar cuando el proyecto se encuentre en problemas.

Esta estimación realizada con base al entendimiento de las partes que componen un sistema, siempre será más exacta que una estimación basada en el conocimiento general del mismo.

Como se mencionó anteriormente la estructura organizacional de un proyecto está basada en su arquitectura. Mientras más información se tenga acerca del alcance y la estructura de un sistema, más exactas serán las estimaciones de costos y tiempos.

Reúso de la abstracción de un sistema

El reúso es una técnica muy importante dentro del desarrollo de sistemas. Mientras que el reúso de código puede ser benéfico, el reúso de una arquitectura provee un impulso importante a sistemas con requerimientos similares. Cuando una arquitectura puede ser reutilizada a través de múltiples sistemas todas las consecuencias positivas y negativas son transferidas al nuevo sistema.

Sistemas construidos con elementos externos
El desarrollo basado en una arquitectura generalmente se enfoca en componer o ensamblar elementos que han sido desarrollados muy probablemente por separado. Esta composición es posible debido a que la arquitectura define elementos que pueden ser incorporados al sistema, define posibles remplazos o adiciones de acuerdo a cómo los elementos interactúan con su ambiente, qué datos consumen y/o producen y qué protocolos de comunicación utilizan.

Esta capacidad de intercambiar elementos de una arquitectura es un aspecto clave dentro de su organización de elementos, interfaces y conceptos operacionales.

La arquitectura puede ser la base para la formación
La arquitectura incluye una descripción de cómo los elementos interactúan para lograr el comportamiento requerido. Esta descripción puede servir como una introducción al sistema para nuevos miembros del proyecto.

Esto tomando en cuenta que, como se mencionó, uno de los principales usos de una arquitectura es servir como medio de comunicación entre los miembros del proyecto. La arquitectura es un punto de referencia común del sistema.

2.2

Patrones de diseño

El diseño de software sirve para definir, organizar y estructurar los componentes del sistema en la solución final que servirá como guía en la implementación del producto.

El diseño orientado a objetos puede ser complicado, más aún cuando se requiere software que, aunque solucione una problemática específica, se comporte como una solución general que permita atacar problemas futuros y nuevos requerimientos.

En general, se debe evitar el rediseño de un modelo orientado a objetos o tratar de minimizarlo. Es complicado mas no imposible obtener un diseño correcto o adecuado desde la primera iteración. Sin embargo, el reúso de diseños exitosos en el pasado facilita el proceso de diseño.

Es común encontrarse con patrones de clases y objetos relacionados, en muchos sistemas orientados a objetos. Dichos patrones resuelven problemas específicos de diseño y hacen los diseños orientados a objetos más flexibles, extendibles, elegantes y reusables.

Esto permite a los diseñadores reusar sus creaciones exitosas, basando sus nuevos diseños en la experiencia.

¿Qué es un patrón de diseño?
Los patrones de diseño son soluciones probadas y documentadas a problemas comunes que se presentan durante la fase de diseño de software.

Cada patrón describe un problema que ocurre en repetidas ocasiones y propone una solución, de tal manera que se puede utilizar millones de veces en múltiples diseños, siendo aplicado no necesariamente de la misma forma en cada uno de ellos.

Un patrón de diseño nombra, abstrae e identifica los puntos clave de una estructura de diseño común, lo que lo hace útil en la creación de diseños orientados a objetos reusables. Identifica las clases e instancias participantes, sus roles y colaboraciones y la distribución de responsabilidades.

Cada patrón se enfoca en un aspecto o problema de diseño en particular, describe cuándo puede ser aplicado, las restricciones, consecuencias y ventajas y desventajas de su uso.

Problemas que ayudan a resolver los patrones de diseño

Definir objetos
Los programas orientados a objetos están hechos a base de objetos. Un objeto encapsula datos y la funcionalidad que opera sobre ellos y ejecuta una operación cuando recibe una petición de un cliente.

Decimos que el objeto está encapsulado debido a que las peticiones son la única forma de ejecutar una operación y las operaciones son la única forma de cambiar el estado del objeto, es decir, sus datos. Esta encapsulación hace invisible la implementación del objeto hacia el exterior.

La parte complicada del diseño orientado a objetos es determinar cómo el sistema se dividirá en objetos. Hay muchos factores que determinan cómo se realiza esta división, muchas veces se contraponen unos a otros. Dichos factores pueden ser tales como la encapsulación, granularidad, dependencias, flexibilidad, rendimiento, evolución, reusabilidad, etc.

Existen muchas metodologías para determinar los objetos adecuados para un sistema. La mayoría de los objetos en un diseño vienen derivados de la fase de análisis como resultado de hacer una representación abstracta del mundo real.

Sin embargo, los diseños finales muchas veces concluyen con objetos que no tienen una contraparte en el mundo real.

Determinar la granularidad
La granularidad de un sistema es la medida de qué tan dividido está el sistema en partes más pequeñas, también se considera la medida de qué tanto se subdivide una entidad más grande en entidades más especializadas y cohesivas.

En un diseño orientado a objetos puede variar tremendamente tanto el tamaño como el número de objetos presentes. Esto se determina considerando la granularidad de los objetos, es decir, qué tan especializados son y cómo se han asignado las responsabilidades. Por lo anterior, un objeto podría ser la fachada de un subsistema completo o podrían existir en nuestro dominio gran cantidad de objetos con una granularidad muy fina, con el costo que conllevaría al sistema la administración de una gran cantidad de objetos.

Especificar interfaces
La interfaz de un objeto son las propiedades y servicios visibles de éste, es la manera de interactuar con él. Cada operación definida en un objeto especifica su nombre, los objetos que toma como parámetros y el valor de retorno. Esto es conocido como la firma de la operación. El conjunto de firmas definido por un objeto conforma su interfaz. La interfaz es el conjunto completo de peticiones que puede recibir un objeto.

Las interfaces son fundamentales en los modelos orientados a objetos. La única manera de conocer un objeto es a través de su interfaz. Por si misma la interfaz no dice nada acerca de la implementación del objeto, todo está encapsulado en su interior.

Los patrones de diseño ayudan a definir las interfaces identificando los elementos clave de los tipos de dato que se intercambian a través de la interfaz, además los patrones pueden indicar qué no poner en la interfaz, definen relaciones entre interfaces y establecen restricciones en las mismas, con el objetivo de conseguir componentes desacoplados y flexibles.

Especificar implementaciones
Un método muy utilizado en el paradigma orientado a objetos, que favorece la reutilización de código, es la herencia.

La herencia consiste en definir la implementación de una clase con base en otra clase jerárquicamente superior. También es posible crear una jerarquía de interfaces utilizando herencia, con lo cual se extienden los servicios brindados por una interfaz.

La herencia es muy importante ya que de ella depende otra característica esencial del paradigma: el polimorfismo.

Todas las clases derivadas de una más abstracta tienen la capacidad de responder a las mismas peticiones que la clase padre, nunca ocultarán alguna de éstas.

Existen beneficios muy importantes en el hecho de manipular objetos en términos de interfaces:

- Los clientes permanecen ignorantes del tipo de objeto que están usando toda vez que estos objetos respetan la interfaz que el cliente espera.
- Por lo anterior, los objetos pueden ser substituidos por otras implementaciones de la misma interfaz, sin afectar al cliente.

Esto reduce las dependencias de implementaciones específicas entre subsistemas, lo que nos lleva al siguiente principio del paradigma orientado a objetos, para lograr un diseño reusable y flexible: Programa hacia una interfaz no hacia una implementación.

Sin embargo, en todo sistema en algún punto se deben de instanciar clases concretas, para lo cual los patrones de diseño creacionales abstraen el proceso de creación de objetos, con lo que se obtienen diferentes formas de asociar una interfaz con una implementación de manera transparente.

Los patrones creacionales aseguran que el sistema estará escrito en términos de interfaces, no implementaciones.

Fomentar el reúso
Un reto muy importante en un diseño orientado a objetos es aplicar los conceptos básicos del paradigma para obtener software reutilizable, flexible y extensible. Los patrones de diseño ayudan a conseguir esto.

Herencia y composición
Las técnicas más comunes para reutilizar funcionalidad en sistemas orientados a objetos son la herencia y la composición de objetos. Como se mencionó anteriormente, la herencia permite definir la implementación de una clase en términos de otra más abstracta. Este tipo de reutilización se conoce como de caja blanca, debido a que la estructura interna de una clase es visible para sus subclases.

Una alternativa a la herencia es la composición. En este caso la nueva funcionalidad es obtenida ensamblando o componiendo objetos para obtener una funcionalidad más compleja. Este estilo de reúso es conocido como de caja negra ya que los detalles internos de los objetos participantes en la composición no son visibles para los demás.

Cada una de estás técnicas tiene sus ventajas y desventajas.

La herencia es definida en tiempo de compilación y en general es sencilla de implementar ya que está soportada directamente por el lenguaje.

Sin embargo, la implementación heredada de la clase padre no puede ser modificada en tiempo de ejecución, es decir, un objeto no puede modificar dinámicamente su clase padre. Además, la implementación de una subclase está íntimamente ligada a la de su clase padre, por lo que cualquier modificación en la clase superior obliga a las subclases a cambiar su implementación.

Cuando se intenta reutilizar una clase es posible que la funcionalidad heredada no sea apropiada para el nuevo problema o el nuevo dominio, por lo que la clase padre muy probablemente deba de ser reescrita o substituida. Esta dependencia limita mucho la flexibilidad y reusabilidad del componente en cuestión.

Por otro lado, la composición de objetos se define dinámicamente en tiempo de ejecución a través de objetos que adquieren referencias a otros. Debido a que las dependencias son accedidas a través de su interfaz la encapsulación no está comprometida.

Cualquier objeto, en una dependencia, puede ser reemplazado por otro en tiempo de ejecución siempre y cuando éstos sean del mismo tipo. Debido a que las dependencias se escriben en términos de interfaces existe muy poco acoplamiento a implementaciones concretas.

Otra ventaja de la composición de objetos es que esta técnica permite encapsular tareas específicas en los objetos, por lo que éstos serán pequeños y se evitará que crezcan hasta llegar a un punto donde sea muy difícil administrarlos por su tamaño.

Lo anterior nos lleva a otro principio del diseño orientado a objetos: Favorecer el uso de la composición de objetos por sobre la herencia de clases.

Idealmente para lograr un reúso importante de componentes, no se tendrían que crear nuevos componentes en caso de existir nuevos requerimientos, solamente se necesitarían ensamblar los componentes existentes en una configuración adecuada, que permita obtener la funcionalidad deseada a través de la composición de objetos. Esto ocurre muy pocas veces debido a que el conjunto de componentes con el que se cuenta en un cierto momento, difícilmente será tan amplio y flexible como para lograr un reúso de componentes de tal magnitud.

Los patrones de diseño pueden ser clasificados dependiendo del problema que ayudan a resolver. A continuación se detallan las categorías más comúnmente utilizadas en el diseño de software.

Patrones de diseño creacionales

Los patrones de diseño creacionales tienen la finalidad de abstraer el proceso de instanciación de objetos. Hacen independiente a un sistema de la forma en que los objetos son creados y compuestos.

Los patrones de diseño creacionales son muy importantes, en la medida que los sistemas evolucionan, basando su implementación más en la composición de objetos que en la herencia. Por lo que comúnmente se define un grupo de funcionalidades fundamentales, que pueden ser ensambladas para formar otras más complejas. Por lo tanto, crear objetos con una funcionalidad en particular requiere más que simplemente instanciar una clase.

Estos patrones de diseño encapsulan el conocimiento acerca de las clases concretas que el sistema utiliza. Además, también ocultan cómo son creadas las instancias de estas clases y cómo se resuelven sus dependencias. El sistema en general lo único que conoce de estas instancias es la información provista por sus interfaces. Todo lo anterior da una gran flexibilidad en cuanto a lo que es creado, quién lo crea, cómo se crea y cuándo.

Factory
Define una interfaz para la creación de objetos, pero permite a sus subclases decidir qué clase instanciar, cede el proceso de instanciación a las subclases.

Una subclase puede sobrescribir la implementación de su clase padre para ofrecer una funcionalidad distinta para el mismo método. Cuando un objeto cliente conoce exactamente la funcionalidad que requiere, este cliente puede

instanciar directamente la clase de la jerarquía que ofrece la funcionalidad requerida.

Sin embargo, hay ocasiones en que un cliente sabe que necesita una clase de una jerarquía (encabezada por las interfaces) pero no conoce la clase específica que requiere. Esta selección de clases puede depender de factores como el estado de la aplicación o la configuración.

Es estos casos el cliente debe de implementar el criterio de selección de la clase adecuada a instanciar.

Este tipo de diseño tiene varias desventajas, cada cliente debe implementar el criterio de selección, lo cual genera un gran acoplamiento entre el cliente y la jerarquía de clases (Figura 2.2.1). En caso de que la jerarquía de clases cambie o se modifique algún criterio de selección, todos los clientes que implementen este criterio deben de ser modificados.

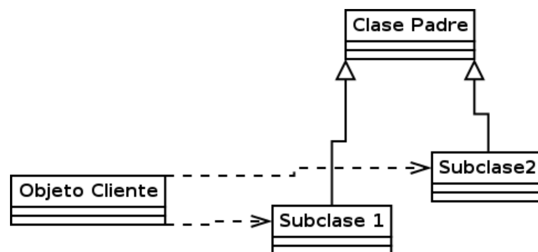

Figura 2.2.1 Acoplamiento entre un objeto cliente y una jerarquía de clases.

En estos casos el patrón Factory recomienda encapsular la funcionalidad requerida para seleccionar e instanciar la clase adecuada en un método conocido como Factory (Figura 2.2.2). Este método selecciona, instancía y devuelve el objeto de la clase seleccionada, basado en el contexto de la aplicación, configuración, reglas de negocio y otros factores.

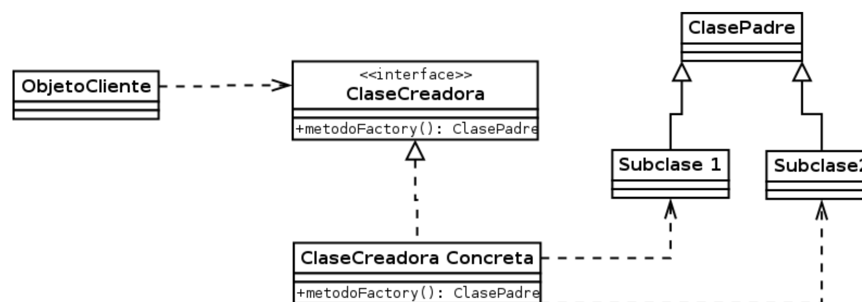

Figura 2.2.2 Uso de un objeto Factory para desacoplar la clase cliente de la jerarquía.

La aplicación del método Factory permite a los clientes acceder a la instancia apropiada, éstos no necesitan conocer ni tratar el criterio de selección de clases ni conocer mecanismos adicionales para instanciar las clases necesarias. Los clientes permanecen desacoplados de toda esta funcionalidad, lo cual hace más flexible el diseño.

Singleton
Restringe a una clase, la cual tendrá solamente una instancia y provee un punto de acceso común a esta instancia.

En un sistema algunas veces puede ser conveniente que la implementación de una clase, solamente tenga una instancia en tiempo de ejecución. Esto puede ser especialmente útil cuando se tiene un recurso limitado y se desea controlar el acceso al mismo mediante un administrador centralizado del recurso (Figura 2.2.3).

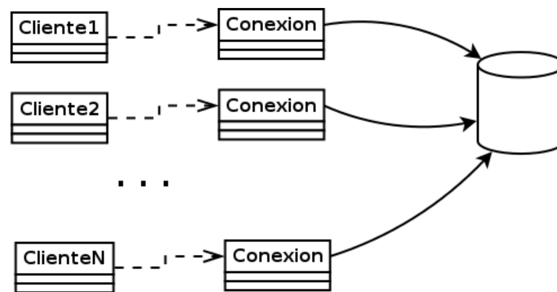

Figura 2.2.3 Acceso a un recurso limitado desde objetos cliente.

También es útil cuando se desea compartir información de manera global. No tiene sentido replicar la información compartida en cada uno de los objetos clientes, simplemente se mantiene una única instancia a nivel global con los datos necesarios.

Para implementar este patrón, la clase debe mantener una referencia de su singleton a nivel clase y de forma privada, para restringir el número de instancias y limitar su acceso respectivamente. Debe contar con un método de inicialización que ponga a punto el singleton antes de que cualquier objeto cliente requiera de sus servicios. No se debe permitir que se cree más de una instancia de la clase, para lo cual es conveniente restringir el acceso a sus métodos constructores.

La clase singleton también debe de proporcionar un punto de acceso global a su única instancia. Los clientes desconocen totalmente cómo es que se crea el objeto singleton.

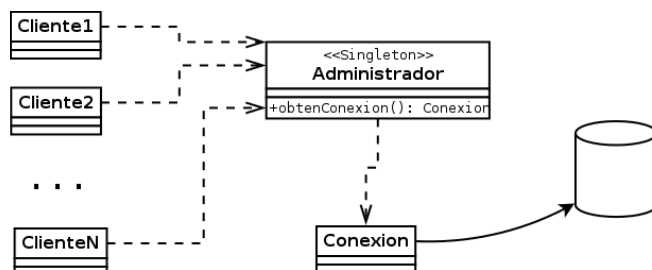

Figura 2.2.4 Uso de un singleton para centralizar y controlar el acceso a un recurso limitado.

Object pool
Define una solución para crear un cache de objetos, a través de un administrador de instancias reutilizables, llamado pool.

El proceso de creación de un objeto puede ser tan simple como reservar y direccionar un espacio en memoria. Por otro lado, también pueden existir objetos tan complejos en su estructura, que para crear uno de ellos se requiere de una cantidad importante de cómputo. Crear continuamente objetos de este tipo puede llegar a afectar de manera significativa el rendimiento de un sistema.

El patrón de diseño Object pool propone una solución a este problema.

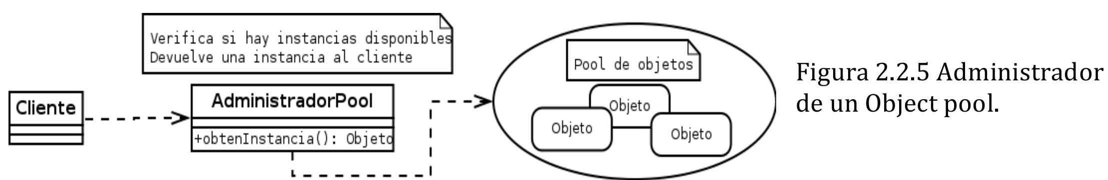

Figura 2.2.5 Administrador de un Object pool.

El pool de objetos "reutilizable" funge como un administrador de las instancias de una clase en específico (Figura 2.2.5). Este pool se encarga de instanciar, destruir y proporcionar a los clientes los objetos que administra. Cuando un objeto cliente requiere de una instancia "reutilizable" la solicita al pool de objetos. El pool verifica si existen instancias disponibles, si es así devuelve un objeto al cliente, en caso contrario creará una nueva instancia. Una vez que el objeto cliente desocupa el objeto "reutilizable", tiene la responsabilidad de "liberarlo" y devolverlo al pool, con la finalidad de que otros clientes puedan ocupar esta instancia.

Con la aplicación de este patrón de diseño los clientes se olvidan totalmente del proceso de creación de los objetos, pueden solicitarlos al pool las veces que así lo requieran.

Este patrón también brinda una abstracción de los recursos que se encuentran limitados dentro de un sistema. Esto lo logra restringiendo el número de objetos que pueden existir dentro del pool. Los objetos administrados por el pool, representan el punto de acceso al recurso limitado. Cuando un objeto cliente solicita una instancia al pool, éste verifica que existan instancias disponibles; en caso de no ser así creará un nuevo objeto, siempre y cuando no se haya alcanzado el límite máximo de instancias permitidas dentro del pool de objetos. Si ya se ha alcanzado este límite, el pool tiene la opción de generar un error, devolver una referencia nula o esperar a que otro cliente libere una instancia y quede disponible para usar.

Con este enfoque múltiples clientes pueden compartir los recursos del sistema de una manera controlada.

Prototype
Especifica el tipo de objetos a crear estableciendo una instancia prototipo. La nuevas instancias son creadas con base al prototipo, es decir, se copia.

El uso de recursos dentro de un sistema es un tema central dentro del desarrollo de software. Como se ha mencionado, el proceso de creación de objetos puede ser computacionalmente muy demandante, por lo que es necesario buscar alternativas a la creación de nuevos objetos.

El patrón de diseño Prototype permite a un objeto cliente crear objetos sin saber su clase o los detalles de cómo fue creado dicho objeto. Es bastante parecido al patrón Factory, sin embargo la diferencia consiste en cómo se crean los objetos. Prototype genera las instancias copiando o clonando una instancia "prototipo".

Para implementarlo debe de existir un administrador de prototipos. A través de este administrador los clientes pueden registrar, modificar o eliminar los prototipos disponibles. También los objetos que así lo requieran obtendrán las instancias a través del administrador (Figura 2.2.6).

Otro de los requerimientos para implementar este patrón es que la clase prototipo debe de implementar un método para realizar la clonación de los objetos. Esta clonación permitirá crear copias exactas de los objetos. Dependiendo de los requerimientos esta copia puede ser superficial o profunda. Superficial cuando se copia el objeto base y también las referencias de sus atributos, con lo cual los objetos clonados apuntarán a las mismos atributos del objeto base. La copia será profunda cuando se clona el objeto base y además se realiza una copia recursiva de sus atributos con lo cual los atributos son iguales pero totalmente independientes de los atributos del objeto base.

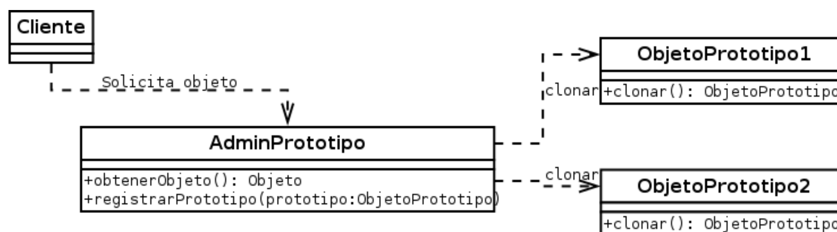

Figura 2.2.6 Creación de objetos a través de prototipos.

Al implementar este patrón se tiene la ventaja de que cuando se requieren crear muchas instancias que sólo difieren entre ellas en el estado, simplemente se crea un objeto base y se va copiando y modificando según sea necesario, lo cual hace más eficiente la creación de objetos.

También se separan los detalles de creación e información de las clases, de los objetos clientes, con lo cual se obtiene un diseño más flexible.

Patrones de diseño estructurales

Los patrones de diseño estructurales están enfocados en cómo se componen las clases y los objetos para formar estructuras más complejas.

Los patrones estructurales de clases utilizan herencia para componer interfaces o implementaciones.

Los patrones estructurales de objetos definen las formas en que los objetos se componen o asocian para realizar una nueva funcionalidad. La flexibilidad obtenida de la composición de objetos viene de poder modificar esta composición en tiempo de ejecución o poder realizar fácilmente una modificación en la implementación de las clases sin afectar los demás componentes del sistema. Esta flexibilidad es imposible de lograr con una composición estática de clases. Estos patrones servirán de guía para estructurar las clases y objetos en un diseño óptimo.

Proxy
Propone la implementación de un objeto sustituto o contenedor de otro objeto con la finalidad de controlar e intermediar el acceso a el.

Para usar los servicios de una clase comúnmente un objeto cliente debe crear una instancia de dicha clase y utilizar directamente sus servicios. Sin embargo, existen algunos escenarios en los cuales al objeto cliente no le es posible acceder a un objeto target (proveedor de servicios) de la forma tradicional. Esto puede ser debido a que el objeto target se encuentra en otro espacio de direcciones ya sea en la misma o en diferente computadora. También es posible que el target no exista hasta que realmente se necesiten sus servicios o porque el objeto target puede ofrecer o negar servicios dependiendo de los privilegios de acceso de los clientes.

En estos casos, para evitar que los clientes deban manejar los requerimientos especiales para acceder al target, el patrón Proxy propone usar un componente separado, conocido como objeto Proxy, para proporcionar un medio de acceso sencillo al target para los clientes (Figura 2.2.7).

El objeto Proxy debe tener la misma interfaz que el objeto target. El Proxy interactúa directamente con el target y se encarga de los detalles específicos de comunicación con este objeto. Con el Proxy los objetos clientes no necesitan manejar ni conocer los detalles especiales para acceder a los servicios del target.

Un cliente puede invocar al Proxy a través de su interfaz y éste redirecciona las peticiones al target. El Proxy oculta a los clientes el hecho de que pueden estar tratando con objetos remotos, objetos que pueden o no estar instanciados o que necesitan de una autenticación especial, es decir el Proxy sirve como un puente entre los clientes y un objeto remoto o un objeto cuya instanciación ha sido diferida hasta que sea necesario.

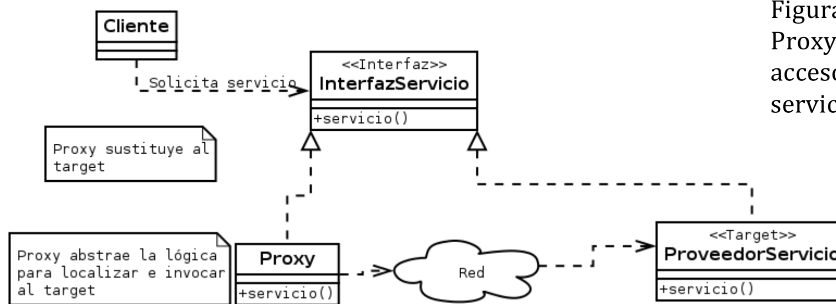

Figura 2.2.7 Objeto Proxy que abstrae el acceso remoto a un servicio.

Los objetos Proxy pueden ser clasificados dependiendo del uso que se les da:

- Proxy remoto: Provee acceso a un objeto localizado en un espacio de direcciones distinto.
- Proxy virtual: Provee la funcionalidad necesaria para permitir la creación bajo demanda de objetos.
- Proxy cache: Provee la funcionalidad necesaria para almacenar los resultados de las operaciones de los objetos target más utilizados.
- Proxy firewall: Protege los objetos target de peticiones de clientes no permitidas.
- Proxy de sincronización: Provee la funcionalidad requerida para permitir acceso concurrente y seguro a un objeto target, desde diferentes clientes.
- Proxy auditor: Provee mecanismos de auditoria antes de la ejecución de un método del target.

Façade

Provee una interfaz unificada a un conjunto de interfaces pertenecientes a un subsistema.

Un subsistema es un conjunto de clases que trabajan en conjunto con el propósito de proveer un conjunto de funcionalidades relacionadas. En general, un subsistema puede estar formado por un gran número de clases. Los clientes de un subsistema pueden requerir interactuar con muchas de las clases del subsistema, este tipo de interacción directa lleva a un gran acoplamiento entre los clientes y el subsistema (Figura 2.2.8). Cualquier modificación que sufra la estructura de interfaces del subsistema afectará directamente a los clientes.

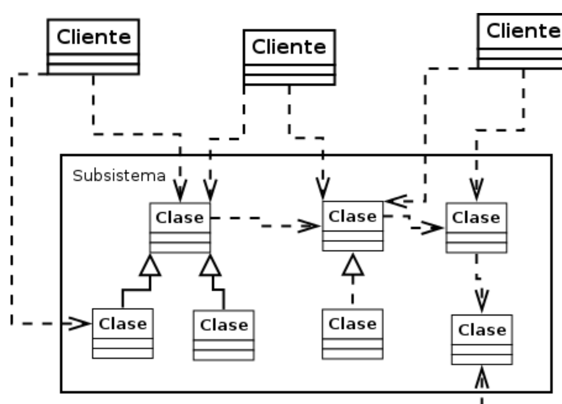

Figura 2.2.8 Clientes accediendo a distintos objetos en un subsistema.

El patrón de diseño Façade es útil en estas situaciones. Este patrón propone el uso de una sola interfaz simplificada de un subsistema, lo cual reduce la complejidad y las posibles dependencias (Figura 2.2.9). Hace que el uso y el mantenimiento del subsistema sea más sencillo.

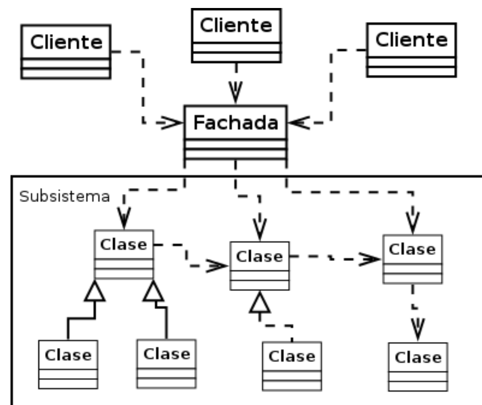

Figura 2.2.9 Fachada ocultando los detalles de implementación de un subsistema.

Un Façade o fachada es una clase que provee esta interfaz simplificada que los clientes utilizarán para interactuar con el subsistema. Con este patrón los clientes interactúan directamente con la fachada en lugar de usar directamente las clases de subsistema. La fachada tiene la responsabilidad de tomar la peticiones de los clientes e interactuar directamente con el subsistema. El patrón Façade fomenta un bajo acoplamiento entre un subsistema y sus clientes. Cuando una clase de un subsistema sufre una modificación los clientes no son afectados.

A pesar de que los clientes pueden utilizar el subsistema a través de la fachada, siempre que lo requieran, los clientes pueden hacer uso directo de las clases del subsistema. Este patrón no restringe el uso directo de las clases del subsistema.

Decorador
Agregar dinámicamente responsabilidades adicionales a un objeto. Provee una alternativa flexible para extender la funcionalidad de un objeto.

Algunas veces es necesario agregar responsabilidades a objetos específicos, no necesariamente a toda la clase. Una manera de lograr esto es mediante la herencia. Sin embargo, el uso de la herencia como solución a este problema es inflexible, ya que se agrega la responsabilidad de manera estática a cada instancia de la subclase. Un cliente no puede elegir cuando requiere que el objeto tenga la responsabilidad extra (agregada por la herencia).

Una solución más flexible es agregar un objeto que envuelve al componente. Este nuevo objeto es el encargado de adicionar las responsabilidades extras al componente original. Este objeto que envuelve es conocido como “decorador”.

El objeto decorador está diseñado para tener la misma interfaz que el objeto original. Esto permite a los clientes interactuar con el decorador exactamente en la misma forma como lo harían con el objeto original (Figura 2.2.10).

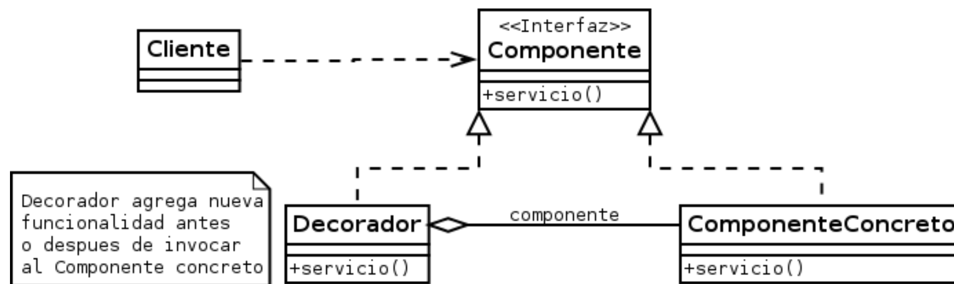

Figura 2.2.10 Objeto que se interpone entre los clientes y la implementación, agregando nueva funcionalidad.

El decorador mantiene una referencia del objeto original, recibe todas las peticiones de los clientes y las redirige al objeto decorado. El objeto decorador agrega funcionalidad adicional antes o después de reenviar las peticiones de los clientes al objeto. Esto facilita que la nueva funcionalidad pueda ser agregada a un objeto en tiempo de ejecución sin tener que modificar su estructura y la implementación de la clase que lo define.

Bridge
Desacopla una abstracción de su implementación de tal forma que ambas puedan variar de manera independiente.

En general, el término abstracción se refiere al proceso de identificar el conjunto de atributos y el funcionamiento de un objeto, que es específico a un uso particular. Esta vista específica de un objeto puede ser diseñada por separado omitiendo atributos y funcionalidad irrelevante para la vista. El objeto resultante puede ser referido como una abstracción. Cada objeto puede tener asociado más de una abstracción y cada abstracción puede ser usada por varios objetos.

En términos de implementación una abstracción puede ser diseñada como una interfaz, con una o más implementaciones concretas.

La herencia es la forma usual de implementar una abstracción. Una clase abstracta define la interfaz de la abstracción y sus subclases concretas la implementan de formas particulares. Esta solución es un tanto inflexible ya que la herencia acopla una implementación a su abstracción, lo cual dificulta modificar, extender y reusar abstracciones e implementaciones de forma independiente.

Además, el uso de la herencia (en la implementación de abstracciones) provoca que cuando existe la necesidad de extender la funcionalidad de una jerarquía de clases, puede existir un crecimiento exponencial de subclases (Figura 2.2.11).

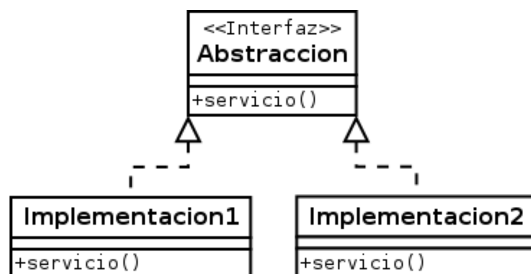

Figura 2.2.11 Jerarquía de clases en la cual las implementaciones se encuentran acopladas a la clase más abstracta.

El patrón Bridge propone para solucionar esta problemática, el uso de un diseño más eficiente y administrable. El diseño de una abstracción usando este patrón separa las interfaces de las implementaciones, ambas se colocan en jerarquías de clases separadas (Figura 2.2.12).

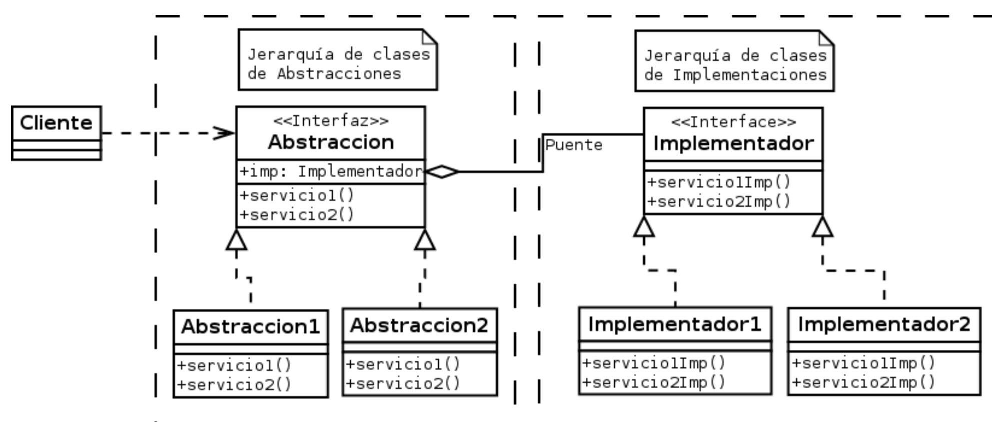

Figura 2.2.12 Abstracciones e implementaciones separadas en jerarquías de clases distintas, unidas por un puente.

La abstracción debe tener una referencia al implementador. Un objeto cliente puede elegir el tipo de abstracción deseada, tomándola de la jerarquía de clases de la interfaz. Después este objeto es configurado con una implementación adecuada, tomándola de la jerarquía de clases de las implementaciones. Cuando un cliente invoca un método en el objeto abstracto, éste redirecciona la invocación a su objeto implementador. El objeto abstracto puede contar con funcionalidad extra antes o después de invocar la implementación. Esta referencia con la que cuenta el objeto abstracto hacia el implementador, es conocida como "puente".

Este tipo de diseño de clases desacopla la interfaz de sus implementaciones y permite a las clases de la interfaz y la implementación ser modificadas sin afectar a su contraparte. Una implementación no está ligada permanentemente a su interfaz, por lo que la implementación de una abstracción puede ser configurada

en tiempo de ejecución y es posible para un objeto cambiar su implementación dinámicamente en tiempo de ejecución.

Patrones de diseño funcionales

Los patrones de diseño funcionales están enfocados en los algoritmos y en la asignación de responsabilidades entre objetos. Este grupo además de describir patrones de objetos y clases, define patrones de comunicación entre ellos. Describen flujos de control complejos que son difíciles de seguir en tiempo de ejecución. Trasladan la atención del control de flujo hacia la forma en que los objetos se interconectan.

Los patrones funcionales de clases se apoyan en la herencia para distribuir la funcionalidad entre las clases.

Los patrones funcionales de objetos utilizan la composición de objetos. Describen cómo los objetos cooperan entre ellos para realizar una tarea difícilmente realizable por uno solo de ellos. Un punto importante que se considera es cómo es que los objetos saben de los demás, cómo se resuelven las dependencias, cómo evitar acoplamiento entre los objetos.

También se enfocan en la encapsulación de funcionalidad en un objeto y delegar peticiones a él.

Command
En general, un sistema orientado a objetos consiste en un conjunto de objetos interactuando entre ellos, los cuales ofrecen una funcionalidad limitada y específica. El sistema responde a ciertos eventos llevando a cabo algún tipo de procesamiento. En términos de implementación, el sistema dependerá de un objeto asignado que se encargará de invocar métodos de los demás objetos, transfiriendo los datos requeridos como argumentos. Este objeto será llamado como el invocador y puede ser visto como parte del cliente. El conjunto de objetos que contienen la implementación de los servicios requeridos para procesar una petición del cliente serán llamados objetos receptores (Figura 2.2.13).

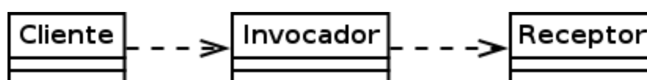

Figura 2.2.13 Flujo de invocación en una aplicación orientada a objetos.

En este contexto el objeto que redirecciona la petición y el conjunto de objetos receptores están altamente acoplados entre ellos ya que interactúan de manera directa. Este diseño tiene muy poca flexibilidad ya que si se requiere agregar nueva funcionalidad se debe de modificar el invocador para que considere el nuevo receptor.

El patrón Command propone desacoplar el invocador del conjunto de receptores, creando una abstracción para el procesamiento o la acción que debe llevarse a cabo como respuesta a la petición de un cliente.

Esta abstracción puede ser diseñada para ofrecer una interfaz común que será implementada por instancias concretas conocidas como objetos Command. Cada uno de estos objetos representa un tipo de petición de un cliente y su correspondiente acción.

Un objeto Command es responsable de ofrecer la funcionalidad requerida para procesar la petición que representa. Pero dicho objeto no debe contener la implementación de esta funcionalidad. Los objetos Command utilizan los receptores para ofrecer la funcionalidad requerida (Figura 2.2.14).

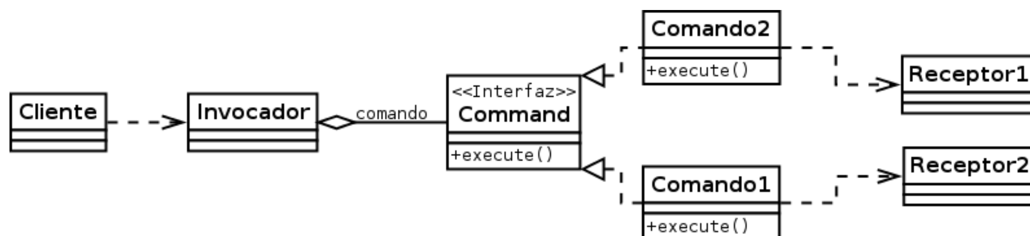

Figura 2.2.14 Invocación de receptores utilizando objetos Command.

Con este diseño ni los clientes ni los invocadores interactúan directamente con los objetos receptores, por lo tanto están desacoplados entre ellos.

Este patrón desacopla los objetos que invocan la operación de aquellos que saben cómo realizarla.

Dicho desacoplamiento permite que se puedan cambiar dinámicamente en tiempo de ejecución las implementaciones de los receptores, con lo cual se modifica la funcionalidad ofrecida, sin afectar los objetos clientes ni los invocadores.

Cuando se requiere nueva funcionalidad se crea un nuevo objeto Command que la ofrece. Para realizar esta adición no es necesario modificar el invocador.

Strategy

Define una familia de algoritmos, los encapsula y los hace intercambiables. Permite a un algoritmo cambiar independientemente de los clientes que lo usan.

Muchos sistemas hacen uso de algoritmos utilitarios, que son ocupados por muchos de sus componentes. A pesar de esto, no es conveniente incluir el código del algoritmo utilitario en cada uno de los componentes que lo usan. Lo anterior debido a que el código del algoritmo se acoplaría completamente a los clientes, provocando que éstos se vuelvan más complejos y difíciles de mantener.

También puede ocurrir que existan variantes relacionadas en la implementación del algoritmo. Para lo cual un cliente debe elegir una implementación específica dependiendo de sus necesidades o del contexto de la aplicación.

El patrón Strategy es útil cuando existe un conjunto de algoritmos relacionados y se requiere que un cliente pueda elegir dinámicamente alguno que satisfaga sus necesidades.

Este patrón propone mantener la implementación de cada uno de los algoritmos en clases separadas. Cada uno de estos algoritmos se conocerán como "estrategia" y los objetos que hacen uso de los objetos estrategia será conocidos como objetos "contexto" (Figura 2.2.15).

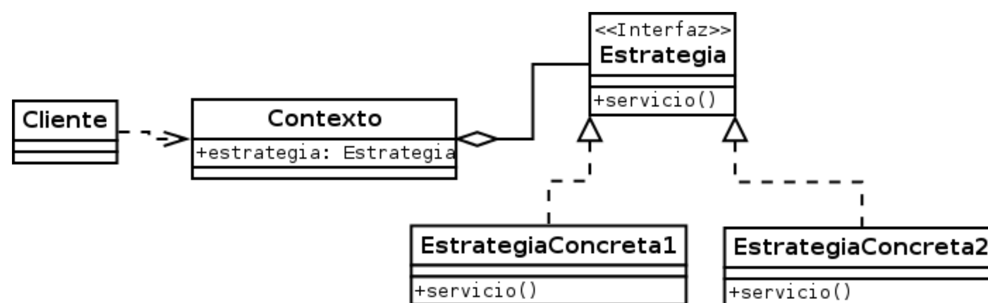

Figura 2.2.15 Implementación de clases con distintas estrategias, ocultando detalles de implementación a los objetos cliente.

Para que un objeto contexto pueda acceder a distintos objetos estrategia de manera transparente, todas las estrategias deben de implementar la misma interfaz, es decir, una interfaz y un número indeterminado de implementaciones concretas correspondientes al número de algoritmos relacionados. Con esto, modificar el comportamiento de un objeto contexto es tan simple como cambiar su objeto estrategia.

Una vez que los algoritmos se encuentran encapsulados en distintas clases, un cliente puede elegir, de entre estos algoritmos, la instancia más apropiada.

Este tipo de diseño separa completamente la implementación de un algoritmo de el contexto que lo utiliza. Por lo tanto, cada vez que la implementación de un algoritmo cambia o se agrega un nuevo algoritmo, tanto el contexto como el cliente no se ven afectados.

Observer
Define una dependencia entre objetos uno a muchos, de tal manera que cuando un objeto modifique su estado, todas sus dependencias sean notificadas y actualizadas automáticamente.

Un reto importante en un sistema altamente distribuido es mantener la consistencia entre sus componentes. Sin embargo, no es deseable lograr esta consistencia acoplando demasiado los componentes del sistema entre sí, ya que esto reduce significativamente su reusabilidad.

Cuando un objeto es dependiente del estado de otro, idealmente debería ser notificado de cualquier modificación en el estado del segundo. Tampoco debería limitarse el número de objetos dependientes.

El patrón Observer propone un diseño para establecer la relación entre los objetos. Permite a los objetos dependientes tener su estado sincronizado con el objeto del cual dependen. El conjunto de objetos dependientes se conoce como observadores y el objeto del cual dependen se le llama sujeto.

Sugiere el uso de un modelo publicador-subscriptor.

Un observador es un objeto con interés o dependencia del estado de un sujeto. Un sujeto puede tener más de un observador. Cada observador necesita saber cuándo un sujeto modifica su estado. La lista de observadores de un sujeto puede cambiar dinámicamente, por lo que cada uno de ellos debe registrarse como observador en el sujeto. Cada vez que un sujeto modifica su estado, éste se encargará de notificar a todos su observadores registrados. Cuando un observador recibe una notificación debe sincronizar el nuevo estado del sujeto. Un observador puede decidir cuándo dejar de seguir el estado de un sujeto, eliminando su registro como observador.

Para que lo anterior sea posible, el sujeto debe proveer una interfaz para registrarse o eliminar su registro de las notificaciones. Los observadores deben proveer una interfaz para recibir las notificaciones de cambio de estado del sujeto (Figura 2.2.16).

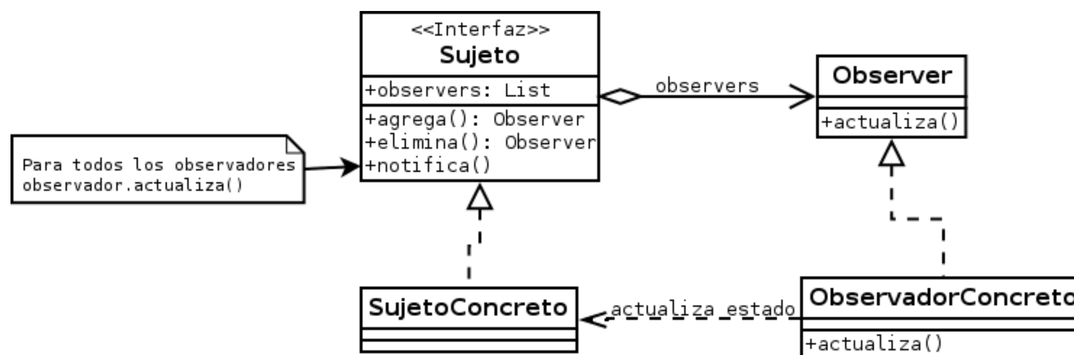

Figura 2.2.16 Objetos observadores que se encargan de vigilar el estado del objeto sujeto.

El sujeto conoce su lista de observadores los cuales implementan una interfaz común, pero el sujeto no conoce la clase concreta de ninguno de sus observadores. Esto minimiza el acoplamiento entre sujetos y observadores.

Memento
Captura y externaliza el estado de un objeto de tal manera que dicho objeto puede ser restaurado posteriormente a este estado, sin romper la encapsulación del objeto.

El estado de un objeto puede ser definido como los valores de sus propiedades o atributos en un tiempo específico. Algunas veces es necesario registrar o persistir el estado de un objeto. Esto es requerido cuando se implementan puntos de control o mecanismos de cancelación (devolver un objeto a un estado anterior) que permitan a un cliente deshacer operaciones o recuperarse de errores. Se debe almacenar la información del estado en algún lugar de donde se pueda recuperar posteriormente.

Sin embargo, los objetos normalmente encapsulan parte o toda su estructura interna, haciendo inaccesible para objetos externos, persistir la información del estado interno de otro objeto. Exponer este estado rompería la encapsulación, comprometiendo la confiabilidad y extensibilidad del sistema.

Aplicar el patrón Memento ayuda a resolver este problema. Un Memento es un objeto que almacena una “foto” del estado interno de otro objeto, llamado originador. El mecanismo de respaldo solicitará un memento al objeto originador cuando necesite un punto de control en el estado del objeto. El originador inicializa el Memento con la información de su estado actual. Solamente el originador puede almacenar u obtener información del memento, éste es opaco para los demás objetos (Figura 2.2.17).

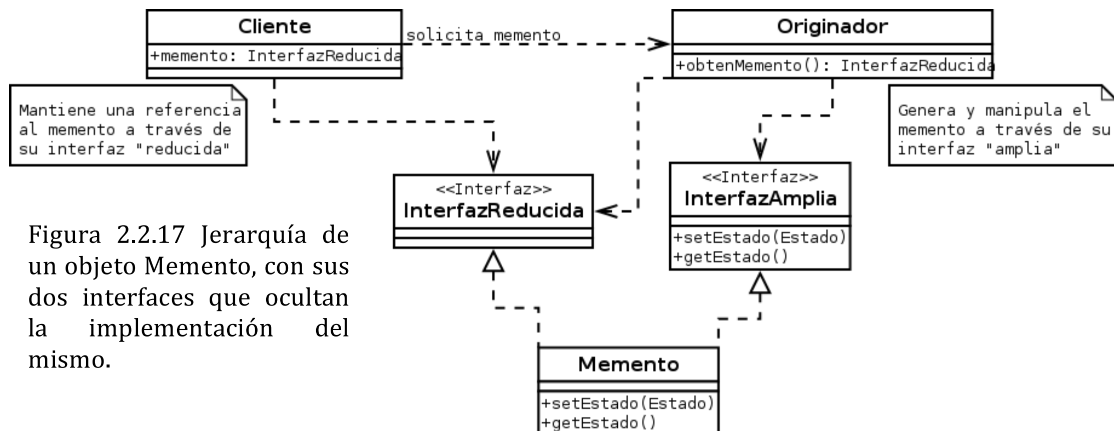

Figura 2.2.17 Jerarquía de un objeto Memento, con sus dos interfaces que ocultan la implementación del mismo.

El objeto originador se encarga de crear un Memento que contiene información acerca de su estado actual y utiliza dicho memento para restaurar un estado anterior.

El memento almacena el estado de un objeto originador. También protege el estado que almacena de accesos no autorizados, es decir, accesos de objetos diferentes al originador. El memento realiza esta protección implementando dos interfaces, una interfaz amplia y otra reducida. El originador tiene acceso al memento mediante una interfaz “amplia”, que le permite definir o recuperar un estado anterior. Mientras que el cliente que solicita el memento sólo tiene acceso

a una interfaz más reducida, por lo tanto el cliente sólo custodia el memento sin operar o examinar la información que contiene.

Patrones GRASP

Después de identificar los requerimientos de un sistema y crear un modelo de dominio, es necesario asignar métodos a las clases y definir los mensajes que deben intercambian los objetos para cumplir con sus requerimientos. La decisión sobre la localización de los métodos y cómo deben interactuar los objetos es muy importante y poco trivial. A esto se le conoce como asignación de responsabilidades.

Una responsabilidad es una obligación de un objeto en términos de su comportamiento. Básicamente se pueden clasificar en responsabilidad de hacer o responsabilidad de saber.

Las responsabilidades de hacer incluyen:

- Hacer algo por si mismo, como crear un objeto o realizar un cálculo.
- Iniciar una acción en otros objetos.
- Controlar o coordinar actividades en otros objetos.

Las responsabilidades de saber:

- Saber acerca de datos encapsulados.
- Saber acerca de objetos relacionados.

Las responsabilidades son asignadas a las clases durante el diseño de objetos. La traducción de responsabilidades en clases y métodos es influenciada por la granularidad de la responsabilidad. Pueden existir responsabilidades que incluyan decenas de clases y cientos de métodos.

Una responsabilidad no necesariamente es lo mismo que un método, sin embargo los métodos son implementados para cumplir una responsabilidad. Las responsabilidades son implementadas usando métodos que actúan en solitario o colaboran con otros métodos y objetos.

Los patrones GRASP (General Responsability Assignment Software Patterns) son principios o guías que ayudan a entender el diseño orientado a objetos y a aplicar dicho diseño de una manera metódica y racional. Ayudan en la asignación de responsabilidades de clases y objetos.

A continuación se mencionan algunos de los patrones que conforman este grupo.

Bajo acoplamiento

El acoplamiento es una medida de qué tanto un elemento está conectado a, tiene conocimiento de, o depende de otros elementos. Un elemento con bajo o débil acoplamiento no depende demasiado de otros elementos. Elementos tales como clases, subsistemas, sistemas, etc.

Existen 5 tipos de acoplamiento, los cuales se mencionan a continuación, ordenándolos desde el nivel más bajo hasta el acoplamiento más alto:

- Acoplamiento de datos: Utiliza listas de parámetros para el intercambio de datos entre componentes, es el tipo de acoplamiento más deseado.
- Acoplamiento zona de datos: Varios componentes comparten variable globales de forma selectiva.
- Acoplamiento control: Un componente comparte banderas de control y parámetros con otros componentes, de tal manera que controla la secuencia de proceso de los demás componentes.
- Acoplamiento zonas compartidas: Dos componentes acceden a un recurso compartido, como memoria compartida o una variable global.
- Acoplamiento de contenido: Cuando un componente accede a la estructura de otro componente, modificando su valores locales o instrucciones.

Un elemento con alto acoplamiento depende de muchos otros. Tales elementos no son deseables ya que traen consigo algunos problemas:

- Son afectados por modificaciones a elementos externos, por lo que son difíciles de mantener.
- Son difíciles de entender cuando se encuentran aislados.
- Difíciles de reusar en otro contexto y configuración, debido a que requieren la existencia de los elementos de los cuales dependen.

El bajo acoplamiento es un principio muy importante en el diseño orientado a objetos y debe ser considerado durante todas las decisiones tomadas durante el diseño de un sistema.

Algunas formas en que se presenta el acoplamiento:

- ClaseA tiene un atributo que refiere a una instancia de tipo ClaseB.
- Un objeto de tipo ClaseA invoca servicios de un objeto de tipo ClaseB.
- Un método de la ClaseA hace referencia a una instancia de tipo ClaseB, puede ser mediante un parámetro, una variable local o el valor de retorno del método.
- ClaseA hereda de ClaseB, directa o indirectamente.

No hay una medida absoluta de cuándo el acoplamiento es muy alto, pero es importante poder estimar el grado actual de acoplamiento y evaluar si el incrementarlo traerá más problemas que beneficios. Normalmente las clases que son genéricas por naturaleza y con gran probabilidad de ser reutilizadas, deben tener necesariamente bajo acoplamiento.

El caso extremo del bajo acoplamiento es cuando no existe acoplamiento entre los elementos. Esto no es adecuado ya que un sistema orientado a objetos es esencialmente un conjunto de objetos interconectados que interactúan entre ellos intercambiando mensajes. Un nulo acoplamiento llevaría a sistemas con componentes monolíticos, poco cohesivos y muy complejos. Un cierto nivel de acoplamiento siempre es necesario para crear un sistema que cuente con

elementos sencillos que colaboren entre ellos para cumplir con sus responsabilidades.

El bajo acoplamiento fomenta el diseño de componentes más independientes que no se ven afectados por los cambios en otros elementos, que son sencillos de entender aisladamente y fácilmente reutilizables.

Alta cohesión
En términos de diseño orientado a objetos, la cohesión es una medida de qué tanto se relacionan y se centran las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas y que no realizan una gran cantidad de tareas o un elemento que tiene responsabilidades que trabajan en conjunto para proveer funcionalidad bien delimitada, ambos tienen alta cohesión.

Existen 7 tipos de cohesión, los cuales se mencionan a continuación, ordenándolos del menor nivel hasta la cohesión más alta:

- Cohesión coincidente: Ocurre cuando las partes de un componente han sido agrupadas en él de manera arbitraria.
- Cohesión lógica: Cuando las partes de un componente se agrupan en él, debido a que corresponden a la misma categoría.
- Cohesión temporal: Cuando las partes de un componente están agrupadas considerando el momento en el cual son ejecutadas.
- Cohesión de comunicación: Varias partes de un componente realizan tareas en paralelo usando los mismos datos de entrada y salida.
- Cohesión secuencial: Un componente realiza distintas tareas secuencialmente, cada una dependiente de la anterior. Es decir se agrupan las tareas relacionadas y dependientes.
- Cohesión funcional: Los elementos de un componente están relacionados en el cumplimiento de una única función. Este tipo de cohesión es la más recomendable.
- Cohesión informacional: Ocurre cuando se realiza una abstracción total de los datos. Este nivel de cohesión es muy difícil de conseguir.

Un elemento con baja cohesión tiene muchas responsabilidades no relacionadas o realiza muchas tareas. Un elemento de estas características puede se muy problemático por lo siguiente:

- Puede ser muy complejo y de gran tamaño, por lo cual es difícil de entender.
- Difícil de reusar.
- Es constantemente afectado por modificaciones, por lo cual es difícil de mantener.

Un elemento con baja cohesión tiene una granularidad muy baja, ya que se le han asignado responsabilidades que debieran ser delegadas a otros objetos.

La alta cohesión es un principio a considerar durante la toma de decisiones de diseño.

Una clase con alta cohesión tiene una cantidad relativamente pequeña de métodos que proporcionan funcionalidad altamente relacionada. Este tipo de clases colaboran con otros elementos para compartir esfuerzos en una tarea de gran tamaño.

Un elemento con alta cohesión tiene muchas ventajas. El alto nivel de funcionalidad relacionada combinado con el pequeño número de operaciones, hacen de un elemento cohesivo fácil de entender, mantener y reutilizar.

Experto
Un diseño orientado a objetos puede definir cientos o miles de clases, una aplicación puede requerir que cientos o miles de responsabilidades sean cumplidas. Durante el diseño, cuando se definen las interacciones entre los objetos, se debe realizar una asignación de responsabilidades a las clases. Una correcta asignación de responsabilidades puede hacer que el sistema resultante sea fácil de entender, mantener y extender.

El patrón Experto es comúnmente utilizado para guiar la asignación de responsabilidades. Propone que los objetos deben cumplir con tareas relacionadas a la información que tienen. A estos objetos se les conoce como expertos.

El cumplimiento de una responsabilidad comúnmente requiere información que se encuentra dispersa a través de múltiples clases u objetos. Esto implica que existen muchos expertos parciales que colaborarán en la responsabilidad.

Este patrón generalmente lleva a diseños donde los elementos de software realizan operaciones que normalmente son hechas por su contra parte del mundo real, el objeto real que representan.

Sin embargo, existen situaciones en las cuales la solución sugerida por Experto no es la óptima, debido usualmente a problemas de acoplamiento o cohesión. Por ejemplo, un objeto del modelo de negocio comúnmente cuenta con toda la información necesaria para almacenarla en la base de datos, por lo cual aplicando el patrón Experto, este objeto de negocio debería tener la responsabilidad de almacenarse en la base de datos. Es más, cualquier objeto de negocio debería tener un método para persistirse. Esto claramente mezcla conceptos (negocio y persistencia), genera un acoplamiento y reduce la cohesión de los objetos.

Estos problemas indican una violación a un principio básico de arquitectura: diseñar para la separación de los principales conceptos de un sistema. Mantener la lógica de la aplicación separada de la lógica de persistencia, y así con los demás conceptos, en lugar de mezclar diferentes conceptos del sistema en el mismo componente.

Creador
La creación de objetos es una de las actividades más comunes en un sistema orientado a objetos. ¿Quién debe ser responsable de crear una nueva instancia de una clase?

El patrón Creador guía en la asignación de responsabilidades relacionadas a la creación de objetos. La intención básica del patrón es encontrar un creador que necesite ser conectado al objeto creado en cualquier evento.

La regla básica indica que se debe de asignar a la ClaseA la responsabilidad de crear la ClaseB si alguna de las siguientes condiciones se cumple:

- ClaseA es una agregación de objetos ClaseB.
- ClaseA contiene objetos ClaseB.
- ClaseA utiliza ClaseB.
- ClaseA tiene los datos de inicialización para crear un objeto de ClaseB (ClaseA es un Experto con respecto a la creación de objetos de ClaseB).

Agregación es una relación muy común entre clases. Creador sugiere que la clase contenedora es un buen candidato para crear los objetos que contiene.

Alguna veces el creador es encontrado al buscar las clases que tienen los datos de inicialización necesarios para crear el objeto.

Comúnmente la creación de objetos es de una complejidad significativa, como el uso de instancias reutilizadas por cuestiones de rendimiento o la creación de objetos de manera condicional dependiendo del contexto de la aplicación. En estos casos es recomendable delegar la creación de los objetos a una clase Factory, que encapsule toda la lógica de generación de instancias.

Controlador
Un evento de sistema de entrada es un evento generado por un actor externo. Estos eventos están asociados con operaciones del sistema, operaciones en respuesta a los eventos.

Un controlador es un objeto interfaz, no de usuario, responsable de recibir o manejar un evento del sistema.

Los sistemas reciben eventos externos, comúnmente relacionados con interfaces de usuario. Otros medios de acceso incluyen, mensajes externos, invocaciones remotas, etc.

En todos los casos se debe elegir un manejador para estos eventos. El patrón controlador sirve como guía para realizar esta elección. Un objeto controlador puede ser visto como una especie de fachada. El controlador recibe peticiones de servicio desde la capa de presentación y coordina su realización, normalmente delegando sus tareas a otros objetos. Existen varios tipos de controladores.

El primer tipo de controlador es un controlador fachada que representa al sistema en su totalidad o un subsistema. Este objeto provee un punto de acceso principal a los llamados de servicio provenientes de la capa de presentación. Este tipo de controladores son adecuados cuando no hay muchos eventos que manejar o cuando no es posible para la interfaz de usuario redireccionar mensajes a controladores alternativos.

Otro tipo de controlador es el controlador de caso de uso, en el cual existe un controlador diferente por cada caso de uso. Ésta es una alternativa a considerar cuando asignar responsabilidades a un solo controlador fachada, lleva a un diseño con baja cohesión y bajo acoplamiento. Típicamente esto ocurre cuando a el controlador se le asigna un número excesivo de responsabilidades.

Un controlador de caso de uso es adecuado cuando existen muchos eventos de sistema a través de diferentes procesos, lo cual hace que el manejo de eventos sea más adecuado separándolo en clases distintas.

Un punto importante del patrón controlador es que los objetos interfaz y la capa de presentación no deben tener la responsabilidad de manejar eventos, éstos deben ser tratados dentro de la lógica de la aplicación o en las capas de dominio.

Los patrones de diseño mencionados en este capítulo, permitieron reutilizar diseños exitosos, en el desarrollo del proyecto. Además, se menciona un conjunto de buenas prácticas que es recomendable implementar, durante la construcción de un sistema.

2.3

Programación orientada a aspectos

Con el paso del tiempo los sistemas de software han ido evolucionando, su complejidad ha ido aumentando y todo indica que esta tendencia continuará. ¿Cómo puede manejarse la complejidad en aumento de los sistemas durante su desarrollo?

Una solución muy común es la modularización. Separando el problema en partes más pequeñas se facilita su diseño e implementación. Normalmente cuando se tienen requerimientos complejos, éstos se separan en múltiples partes como lógica de negocio, acceso a datos y lógica de presentación. Cada una de estas funcionalidades se conocen como responsabilidades del sistema.

Un sistema bancario tiene la responsabilidad, por ejemplo, de gestionar usuarios, cuentas y créditos. Este tipo de funcionalidad se conoce como responsabilidades esenciales ya que forman parte de la funcionalidad central del sistema.

Otras funcionalidades del sistema como la seguridad, registro de bitácoras, pooling de recursos, caché, transaccionalidad, etc, son responsabilidades que pueden abarcar muchos módulos y son genéricas para la mayoría de los sistemas. Este tipo de funcionalidad se conoce como responsabilidades cruzadas.

Para diseñar e implementar responsabilidades esenciales de un sistema la programación orientada a objetos cumple correctamente con esta tarea. Sin embargo, el diseño e implementación de responsabilidades cruzadas es un poco más complejo. Responsabilidades como seguridad, monitoreo o logging no pueden ser implementadas por un módulo correspondiente a cada una. El paradigma orientado a objetos obliga a dispersar la implementación de este tipo de responsabilidades en diferentes módulos. Esta dispersión de código genera un acoplamiento entre responsabilidades y reduce la cohesión de los módulos del sistema, debido a que se mezclan distintas responsabilidades en un solo componente.

Definición

La programación orientada a aspectos (POA) brinda una solución alternativa para el problema al diseñar e implementar adecuadamente responsabilidades cruzadas.

Los aspectos ayudan a modularizar responsabilidades cruzadas. Como se ha mencionado, estas responsabilidades son cualquier funcionalidad que afecta varios puntos de una aplicación. La seguridad es una responsabilidad de este tipo ya que varios métodos en una aplicación pueden tener reglas de seguridad aplicadas.

En la figura 2.3.1 se muestran tres módulos que proveen servicios particulares, para cumplir con una responsabilidad esencial. Los módulos mostrados adicionalmente requieren de funcionalidad auxiliar como seguridad, logging y transaccionalidad.

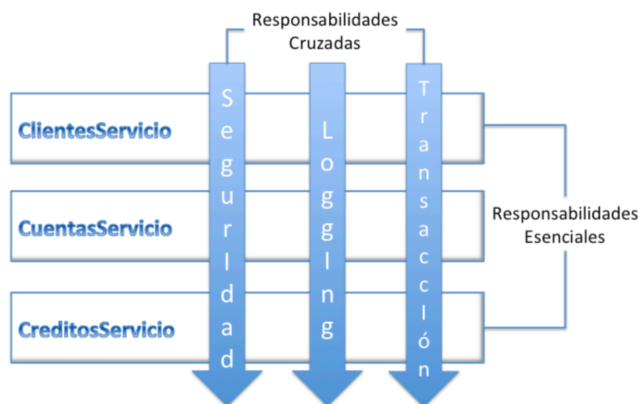

Figura 2.3.1 Responsabilidades cruzadas, se extiende a través de varios módulos del sistema.

Una técnica recurrente para reusar funcionalidad común es la aplicación de herencia o delegación. Pero el uso de la herencia provoca un acoplamiento inherente entre los componentes y la delegación puede ser inconveniente debido a la complejidad en la invocación de objetos delegados o la dependencia creada entre los componentes.

Los aspectos ofrecen una alternativa a la herencia y a la delegación que puede ser más limpia y adecuada en muchas circunstancias. Con la orientación a aspectos, al igual que en el paradigma orientado a objetos, se define la funcionalidad común en un solo lugar, pero con los aspectos se puede definir declarativamente cómo y dónde se aplicará esta funcionalidad sin tener que modificar el componente al cual se le está aplicando.

Con la programación orientada a aspectos las responsabilidades cruzadas pueden ser modularizadas en componentes especiales llamados aspectos. Esto tiene dos beneficios principales. Primero, la lógica para cada responsabilidad está en un solo lugar en vez de tener esta lógica dispersa por varios componentes, como sucede con la programación orientada a objetos (POO).

Segundo, los componentes centrales del sistema son más limpios ya que contienen únicamente lógica específica de las responsabilidades esenciales y las responsabilidades cruzadas se mueven a los aspectos.

Conceptos

Al igual que otras tecnologías la POA cuenta con terminología básica que permite entender y describir su funcionamiento. A continuación se mencionan los conceptos principales.

Aspecto

Un aspecto es un tipo particular de responsabilidad. Un aspecto es una responsabilidad cuya funcionalidad es disparada por otras responsabilidades y en situaciones variadas. Si la responsabilidad no estuviera separada en un aspecto ésta tendría que ser disparada explícitamente desde otra responsabilidad y provocaría que ambas responsabilidades quedaran mezcladas.

Un aspecto describe una funcionalidad que podrá ser aplicada (advice), en ciertos puntos (pointcut) durante la ejecución de una aplicación. Con lo cual se agregan de manera dinámica responsabilidades sin la necesidad de afectar el código del componente al cual se le aplica el aspecto.

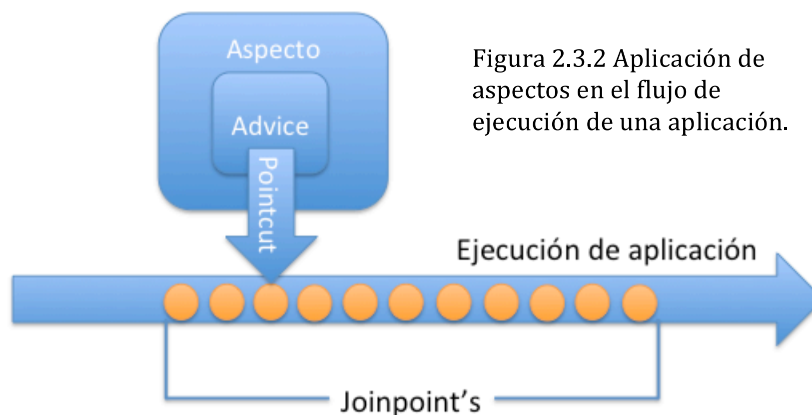

Figura 2.3.2 Aplicación de aspectos en el flujo de ejecución de una aplicación.

Advice

Un advice es el propósito o la tarea que debe cumplir un aspecto. Define el qué y el cuándo del aspecto. Describe la tarea que un aspecto realiza y el momento en cual se debe realizar dicha tarea.

Algunos de los tipos de advice que tiene POA son los siguientes:

- Antes: La tarea del advice se ejecuta antes de que el método relacionado es invocado.

- Después: La tarea del advice se ejecuta después de que el método relacionado es invocado.
- Después de regresar: La tarea del advice se ejecuta después de que el método relacionado es invocado de manera exitosa.
- Después de lanzar: La tarea del advice se ejecuta después de que el método relacionado lanza una excepción.
- Alrededor: El advice envuelve al método relacionado, provee funcionalidad ejecutada antes y después de la invocación.

Joinpoint
Una aplicación puede tener cientos o miles de puntos dentro de ella donde se puede aplicar un advice. Estos puntos son conocidos como joinpoint.

Un joinpoint es un punto en la ejecución de una aplicación en donde se puede conectar un aspecto. Este punto puede ser la invocación de un método, el lanzamiento de una excepción o la modificación de un atributo. Este conjunto de puntos es donde el código de un aspecto puede ser insertado dentro del flujo normal de la aplicación y agregar nueva funcionalidad.

Pointcut
En general, un aspecto no es aplicado a todos los joinpoint dentro de una aplicación. Un pointcut permite limitar los joinpoint en los cuales se aplicará un aspecto en particular.

Un pointcut define el dónde de un aspecto. La definición de un pointcut puede coincidir con uno o más joinpoint, en los cuales deberá ser aplicado el advice.

Regularmente los pointcut se declaran utilizando de manera explicita el nombre de una clase o método, o a través de expresiones regulares que determinan un conjunto de clases y métodos. Algunas implementaciones de POA permiten crear pointcut's dinámicos que determinan la aplicación de un advice basado en decisiones en tiempo de ejecución.

Weaving
Weaving es el proceso de aplicar aspectos a un objeto target para crear un nuevo objeto proxy. Los aspectos son enlazados al objeto target en los joinpoint definidos por el pointcut del aspecto. El enlazado puede ser llevado a cabo en muchos puntos del ciclo de vida de un objeto:

- Tiempo de compilación: Los aspectos son enlazados cuando la clase target es compilada. Para esto se requiere un compilador especial.
- Tiempo de carga de clases: Los aspectos son enlazados cuando la clase target es cargada en memoria. Este proceso requiere un cargador de clases especial que modifica el bytecode de la clase antes de que sea introducida en la aplicación.
- Tiempo de ejecución: Los aspectos son enlazados durante la ejecución de la aplicación. Normalmente un contenedor POA generará dinámicamente

objetos proxy que contienen la lógica del aspecto y que antes o después de la ejecución del aspecto delegan al objeto target.

Introducción

Una "introducción" permite agregar nuevos métodos o atributos a clases existentes. Es el proceso de forzar a una clase a implementar una nueva interfaz. Esto se hace creando un objeto con la nueva funcionalidad e introduciéndolo en una clase existente en tiempo de ejecución, sin tener que modificar el código de la clase, dando a ésta una nueva funcionalidad y estado.

Implementaciones

El término POA fue acuñado en 1996 por Gregor Kiczales quien inició el proyecto AspectJ, la primera implementación. POA es una metodología que puede tener muchas implementaciones. Cada una de ellas ofrece una visión diferente acerca del paradigma orientado a aspectos. A continuación se mencionan algunas.

AspectJ

Es la implementación original desarrollada por Xerox. Tiempo después de su liberación inicial el proyecto fue transferido a la comunidad eclipse.org. En sus primeras implementaciones AspectJ extendía el lenguaje Java a través de palabras reservadas que ayudaban a soportar conceptos de POA y proveía un compilador especial. Para la versión 5 se ofreció la posibilidad de utilizar anotaciones Java.

Spring

Spring es el framework ligero para aplicaciones empresariales más popular. Con el objetivo de cumplir con los requerimientos necesarios para aplicaciones empresariales, provee una implementación de POA basada en interceptores y el patrón de diseño proxy. Su modelo de programación se basa en AspectJ, lo cual ofrece una mejor experiencia de programación y permite a los usuarios de Spring desarrollar aspectos fácilmente.

JBoss

JBoss es un servidor de aplicaciones de código abierto, que ofrece una implementación de POA, que incluye un lenguaje para Pointcut's similar al ofrecido por AspectJ.

Aspect-Oriented C

Es un proyecto de investigación de la Universidad de Toronto que permite el desarrollo con POA en el lenguaje C. Consiste en un compilador que convierte código escrito en Aspect-OrientedC en código ANSI-C. El código resultante puede ser compilado por un compilador compatible con ANSI-C.

PostSharp

Es una implementación de POA para la plataforma .NET que permite aplicar aspectos en tiempo de compilación.

¿Cómo afecta a una arquitectura el uso de POA?

Utilizando únicamente POO las responsabilidades cruzadas son normalmente implementadas agregando el código necesario para cada responsabilidad de este tipo a cada componente (Figura 2.3.3).

Figura 2.3.3
Componentes implementando responsabilidades cruzadas y esenciales, al mismo tiempo.

Cada módulo del sistema implementa responsabilidades esenciales y responsabilidades cruzadas. Aunque existe una separación conceptual entre responsabilidades en el diseño, su implementación las mezcla. Lo cual rompe el principio de responsabilidad simple ya que en este caso una sola clase implementa reponsabilidades esenciales y cruzadas al mismo tiempo. Esto provoca que si es necesario modificar las invocaciones al código relacionado con responsabilidades cruzadas, se debe modificar cada clase que incluya dicha invocación. Lo anterior hace más costosa la implementación de funcionalidad y la corrección de errores.

Con una implementación utilizando POO las responsabilidades esenciales y cruzadas están mezcladas en cada módulo. Además, cada responsabilidad cruzada se encuentra dispersa en varios módulos.

La mezcla de código es provocada cuando un módulo es implementado para cumplir con varias responsabilidades de manera simultánea. Es común considerar responsabilidades como lógica de negocio, sincronización, logging y seguridad cuando se implementa un módulo. Lo anterior lleva a la presencia simultánea de implementaciones de cada responsabilidad y resulta en la mezcla de código.

La dispersión de código es provocada cuando una funcionalidad simple se implementa en múltiples módulos. Debido a que las responsabilidades cruzadas están por definición esparcidas en varios módulos, su implementación también se encuentra dispersa en todos estos módulos.

La mezcla y dispersión de código impactan el diseño y desarrollo de software, provocan menor productividad, menor reúso de código, menor calidad y dificultad para evolucionar.

En POO las responsabilidades esenciales pueden tener un bajo acoplamiento mediante interfaces, sin embargo no hay una manera sencilla de lograr lo mismo con las responsabilidades cruzadas.

Considerar el caso mostrado en la Figura 2.3.4:

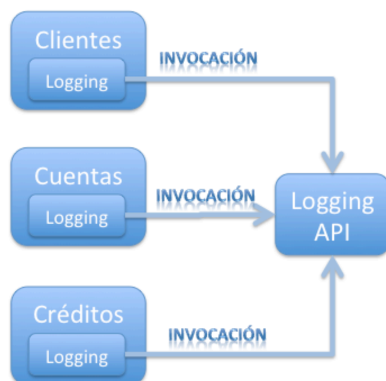

Figura 2.3.4 Módulos cliente, utilizando los servicios provistos por el API de logging.

El módulo de logging ofrece sus servicios a los clientes a través de una interfaz. El uso de una interfaz reduce el acoplamiento entre los clientes y la implementación de la misma. Los clientes que utilizan los servicios de logging a través de la interfaz, desconocen la implementación que están usando en realidad. Este diseño necesita que cada cliente contenga invocaciones al módulo de logging. Dichas invocaciones deben ser incluidas en todos los módulos que requieran logging y por lo tanto se mezclan con el código central del módulo.

Utilizando POA ningún módulo contiene invocaciones al API de logging (Figura 2.3.5).

Figura 2.3.5 POA se encarga de ligar la funcionalidad esencial con la funcionalidad de logging.

La responsabilidad de logging, su implementación e invocaciones, residen solamente dentro del API de logging y el aspecto encargado del logging.

El cambio fundamental provisto por la programación orientada a aspectos es la preservación de la mutua independencia de las responsabilidades individuales. Las implementaciones pueden ser mapeadas fácilmente a cada una de las responsabilidades correspondientes, lo cual resulta en un sistema más sencillo, fácil de entender, más fácil de implementar y más simple de proveer mantenimiento perfectivo y correctivo.

Las características de POA descritas en este capítulo, permitieron tomar ventaja de ellas en la implementación del sistema de bitácoras. En capítulos posteriores se analiza cómo es que un sistema se puede ver beneficiado por este paradigma implementado por Spring Framework.

2.4

Spring Framework

Hasta antes de 1998 la tecnología Java contaba únicamente con la especificación JavaBean para definir componentes reutilizables. Desafortunadamente éstos parecían ser demasiado simples para ser capaces de realizar tareas en el mundo real.

Aplicaciones sofisticadas requieren de servicios adicionales como transaccionalidad, sincronización y seguridad, servicios que no son provistos por la especificación JavaBean. En marzo de 1998 Sun publicó la versión 1.0 de la especificación Enterprise JavaBean (EJB). Esta especificación amplió la noción de componentes Java, proveyendo servicios empresariales, pero descuidando el aspecto de la simplicidad de los JavaBean.

Los EJB no lograron su propósito original: simplificar el desarrollo de aplicaciones empresariales. Los EJB simplifican la interacción con la infraestructura del servidor de aplicaciones durante el desarrollo, pero complican el mismo imponiendo el uso de descriptores de despliegue y la inserción de código específico a la plataforma en los componentes de la aplicación.

En la actualidad se busca que el desarrollo basado en componentes Java vuelva a la simplicidad de sus inicios. Técnicas como la POA y la inyección de dependencias enriquecen a los JavaBean, otorgándoles características que anteriormente eran exclusivas para los EJB. Estas técnicas dan poder a los POJO's (objetos simples de Java) con una programación declarativa utilizada por los EJB, pero sin tener la complejidad de éstos.

Sin embargo, la especificación EJB ha evolucionado para promover un desarrollo basado en POJO's. Utilizan ideas como inyección de dependencias y POA, por lo cual esta especificación es mucho más simple que sus antecesoras. A pesar de esto, otros frameworks basados en POJO's se han establecido como estándares de facto dentro de la comunidad Java.

¿Qué es Spring?

Spring es un framework de código abierto creado por Rod Johnson. Fue hecho con el objetivo de hacer más manejable la complejidad del desarrollo de una aplicación empresarial y hace posible usar simples JavaBean's para realizar tareas reservadas anteriormente para los EJB. Cualquier aplicación Java se puede beneficiar de Spring, no solamente las empresariales, logrando simplicidad, capacidad de pruebas y un bajo acoplamiento.

Spring ayuda en muchas cosas, pero en resumen todos los esfuerzos están enfocados en simplificar el desarrollo en Java. Esto lo logra atacando la complejidad existente con las siguientes estrategias:

Desarrollo sencillo y no invasivo con objetos simples de Java
Es muy común en el trabajo con Java encontrarse con frameworks que imponen a los usuarios el heredar de sus clases o implementar alguna de sus interfaces. Estos frameworks "pesados" obligan a los programadores a escribir clases llenas de código innecesario y acoplándose altamente al framework.

Spring evita, en la medida de lo posible, invadir la aplicación con código de su API. No obliga a implementar interfaces específicas del framework y, de hecho, las clases en una aplicación basada en Spring casi siempre permanecen ignorantes de que están siendo utilizadas por el framework.

Bajo acoplamiento a través de la inyección de dependencias y uso de interfaces
Cualquier aplicación no trivial está compuesta por varias clases que colaboran entre ellas en la realización de una tarea. Normalmente cada objeto es responsable de obtener las referencias de los objetos con los cuales colabora. Esto puede resultar en un código altamente acoplado y con poca capacidad de pruebas.

El acoplamiento puede ser muy problemático por las razones que se han expuesto anteriormente, pero normalmente es necesario tener una cierta cantidad de él. Un componente totalmente desacoplado no puede colaborar con otros. Con el objetivo de que las clases puedan realizar alguna tarea útil, de alguna forma las clases necesitan conocer la existencia de las demás. El acoplamiento es necesario, pero debe ser tratado con cuidado.

La inyección de dependencias permite que las dependencias de los objetos sean asignadas en tiempo de creación, por un tercero que coordina todos los objetos dentro de un sistema. Los objetos ya no son responsables de resolver sus dependencias, las cuales son inyectadas dentro de los objetos que las requieran.

El principal beneficio de la inyección de dependencias es el bajo acoplamiento. Los objetos saben de sus dependencias a través de sus interfaces, no de sus implementaciones o de la forma en que son instanciadas. De esta forma la dependencia puede ser cambiada por otra implementación sin que el objeto dependiente sea afectado.

Uso de programación orientada a aspectos para garantizar componentes cohesivos

POA permite encapsular en componentes reutilizables, funcionalidad que es requerida a través de toda una aplicación.

Como se expuso anteriormente, POA es definida comúnmente como una técnica que promueve la separación de responsabilidades dentro de un sistema. Está centrado en los casos en los cuales un componente tiene responsabilidades adicionales, más allá de su funcionalidad principal. Se puede decir que hay responsabilidades cruzadas que se encuentran dispersas por todo el sistema.

Esta dispersión de responsabilidades genera dos problemas principales. Primero, el código que implementa las responsabilidades cruzadas se encuentra repetido en muchos componentes que así lo requieren. Esto significa que una modificación en estas responsabilidades implicaría modificar todos los componentes involucrados. Incluso si se modularizan estas responsabilidades, las invocaciones a los métodos necesarios tendrían que duplicarse y repartirse por todos los componentes que lo requieran.

Segundo, los componentes contienen código que no se encuentra alineado con su responsabilidad principal.

Los aspectos permiten modularizar las responsabilidades cruzadas y aplicarlas declarativamente a los componentes que así lo requieran. Con esto se obtienen componentes más cohesivos y que se concentran en su responsabilidad principal, permaneciendo ignorantes de cualquier servicio del sistema que colabore con ellos.

En pocas palabras los aspectos permiten que los POJO's permanezcan simples.

Reducción de código repetitivo a través de plantillas

Dentro de Java existen muchas actividades que requieren de código repetitivo para poder ejecutarse. Como ocurre con servicios de conexión a bases de datos (JDBC), colas de mensajes (JMS) o servicios de localización de recursos (JNDI). Spring busca eliminar el código repetitivo, encapsulándolo en plantillas.

Tal es el caso de la plantilla *JDBCTemplate.* Ésta permite a los objetos que requieran acceso a la base de datos, enfocarse en el proceso de consulta de los datos y olvidarse de cumplir con las demandas de código necesarias para utilizar el API de JDBC.

Módulos

Spring Framework consiste en una serie de características que se encuentra organizadas en alrededor de 20 módulos. Estos módulos se agrupan en Contenedor, Acceso a datos/Integración, Web, POA, Instrumentación y Test (Figura 2.4.1).

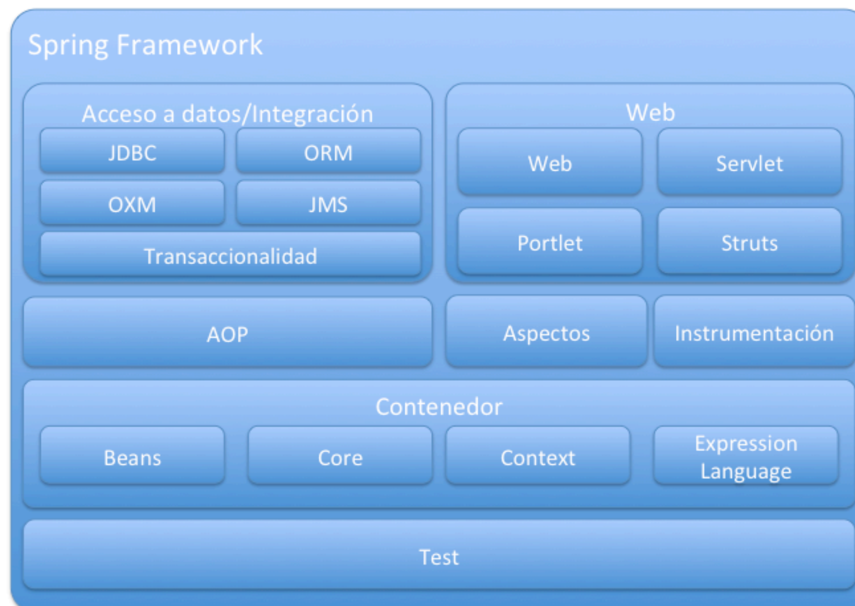

Figura 2.4.1 Módulos de Spring Framework

Spring Container

En una aplicación basada en Spring, los objetos viven dentro del contenedor de Spring. Éste se encarga de crear los objetos, ligarlos, configurarlos y gestionar su ciclo completo de vida, desde su creación hasta su destrucción.

El contenedor es el núcleo del Framework y éste cuenta con muchas implementaciones, categorizadas en fábricas de beans, que son los contenedores más simples y proveen soporte básico para inyección de dependencias, y los contextos de aplicación que se construyen sobre una fábrica de beans proveyendo servicios adicionales.

El contenedor de Spring está formado por los módulos Core, Beans, Context y Expression Language.

Los módulos Core y Beans proveen las partes fundamentales del framework, como la inyección de dependencias. Proveen una implementación sofisticada del patrón Factory. Eliminan la necesidad de programar singletons y permiten

desacoplar la configuración y especificación de dependencias de la lógica central de la aplicación.

El módulo Context se implementa sobre las bases provistas por los módulos Core y Beans, permite gestionar los objetos de una manera similar a un registro JNDI. Extiende la funcionalidad del módulo Beans agregando soporte para internacionalización, manejo de eventos, recursos y la creación trasparente de contextos en entornos, como puede ser un contenedor de servlets. Además, implementa características de Java EE como EJB, JMS y comunicación remota.

El módulo Expression Language provee un poderoso lenguaje para consultar y manipular un grafo de objetos en tiempo de ejecución. Es una extensión de EL definido en la especificación JSP 2.1. Soporta la asignación y obtención de propiedades, invocación de métodos, operaciones lógicas y aritméticas y obtención de objetos desde el contenedor.

Spring POA

Spring POA provee una implementación compatible, del paradigma orientado a aspectos. Permite definir y aspectos particulares y aplicarlos en interceptores de métodos para desacoplar de una manera limpia las implementaciones de distintas responsabilidades.

El soporte de Spring para POA viene en cuatro implementaciones:

- POA clásico basado en proxy's.
- Aspectos manejados por anotaciones de AspectJ.
- Aspectos implementados en POJO's
- Aspectos de AspectJ inyectados.

Las primeras tres son variantes en la implementación de Spring basada en proxy's. Por lo cual Spring POA está limitado a la intercepción de métodos. Si se requiere una intercepción más compleja (como constructores o atributos) se deben implementar los aspectos con AspectJ e inyectando beans de Spring en los aspectos creados.

Características de Spring POA

Los componentes advice son escritos en Java

Los advice creados en Spring deben ser escritos en una clase de Java. La definición de los pointcut, que es donde se especifica los puntos de ejecución en los cuales se aplicarán los advice, son escritos en XML en la configuración de Spring. Por lo cual ambas implementaciones son familiares para cualquier desarrollador de Java.

Spring aplica aspectos en tiempo de ejecución
En Spring los aspectos son ligados a los beans manejados por el framework en tiempo de ejecución (Figura 2.4.2).

El ligado de aspectos se realiza envolviendo los beans dentro de un proxy.

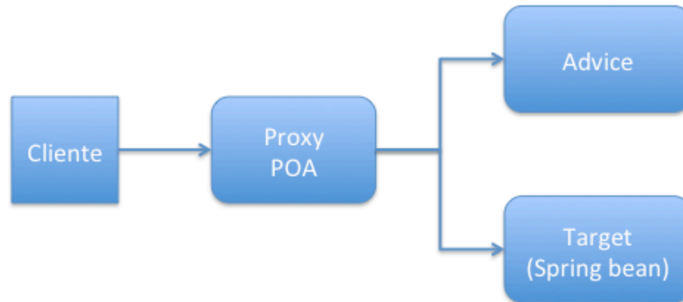

Figura 2.4.2 Implementación de los aspectos en Spring.

El objeto proxy se posiciona como el objeto target, interceptando las llamadas y redireccionando éstas al objeto target.

En el tiempo entre la intercepción del proxy y cuando éste invoca el método en el objeto target, el proxy ejecuta la lógica del aspecto (advice).

Spring crea los objetos proxy, dentro de un contenedor, hasta que éstos son requeridos por la aplicación. Debido a que Spring crea los proxy's en tiempo de ejecución no es necesario un compilador especial para ligar los aspectos a los beans de Spring.

Spring sólo soporta Joinpoint's en los métodos
Debido a que Spring POA está basado en proxy's dinámicos, el framework sólo soporta Joinpoint's en los métodos. Spring carece de aplicación de aspectos en atributos, por lo que no es posible interceptar actualizaciones en los atributos de un objeto. Tampoco se puede aplicar un aspecto en el método constructor de un objeto, por lo cual no es posible interceptar la instanciación de un objeto.

La intercepción de métodos en general cubre la mayoría de las necesidades cuando se trabaja con POA. Pero si se requiere más que solamente aplicación de aspectos en métodos, se puede complementar Spring POA con AspectJ.

El módulo de Spring "Aspectos" provee integración del Framework con AspectJ.

El módulo "Instrumentación" ofrece soporte para la instrumentación de código (adición de byte-code a los métodos con el propósito de obtener información) e implementaciones de cargadores de clases que pueden ser usados en ciertos servidores de aplicaciones.

Spring Data Access/Integración

Consiste en los módulos de JDBC, ORM, OXM, JMS y Transaccionalidad.

El módulo JDBC provee una abstracción que elimina la necesidad del código repetitivo propio de JDBC y el tratamiento de códigos de error particulares de cada manejador de bases de datos.

El módulo ORM provee integración con API's de mapeo objeto-relacional como JPA, Hibernate e iBatis. Utilizando este módulo se pueden aprovechar las características de los ORM en combinación con todas las funcionalidades provistas por Spring.

El módulo OXM provee una abstracción que soporta implementaciones de mapeos objeto-xml como JAXB, Castor y XMLBeans.

EL módulo JMS contiene características para la producción y consumo de mensajes.

El módulo de transaccionalidad provee soporte para el manejo de transacciones de manera programática y declarativa, para los POJO's de una aplicación.

Spring Web

El patrón MVC (modelo vista controlador) es una solución comúnmente aceptada en la construcción de aplicaciones web, en la cual la interfaz de usuario está separada de la lógica de la aplicación. Java cuenta con muchas implementaciones de MVC tales como Struts, JSF, WebWork, Tapestry entre otras.

Aunque Spring cuenta con integración con muchas de las implementaciones más populares de MVC, su módulo web y de comunicación remota, contiene un framework MVC que fomenta las técnicas de bajo acoplamiento en la capa web de una aplicación. Este framework viene en dos implementaciones: un framework basado en servlet para aplicaciones web convencionales y una basada en portlet para desarrollar sobre el API Portlet de Java.

Además de aplicaciones web para usuarios finales, este módulo provee múltiples opciones para construir aplicaciones que interactúan con otras. El módulo de comunicaciones remotas de Spring incluye RMI (invocación remota de métodos), Hessian, Burlap, JAX-WS y un invocador sobre HTTP propio de Spring.

Testing

Spring provee un módulo dedicado a pruebas de aplicaciones basadas en el framework, reconociendo la importancia del desarrollo de pruebas. En este módulo se encuentra una colección de objetos de prueba para la realización de pruebas unitarias de código que trabaja con módulos como JNDI, servlets y

portlets. Para pruebas de integración este módulo provee soporte para cargar una colección de beans en un Applicacion Context específico y realizar pruebas.

Como se ha mencionado en este capítulo, Spring facilita el desarrollo de aplicaciones empresariales, construidas en Java. Los módulos con los que cuenta, servirán de apoyo en el desarrollo de la solución propuesta para la correcta implementación de las bitácoras de un sistema. En capítulos posteriores se menciona cómo es que se implementó el sistema de bitácoras utilizando este framework.

3

Planteamiento del problema

Con el paso del tiempo los sistemas han ido aumentando su complejidad, tanto en tamaño como en las diversas configuraciones que puede tener su arquitectura. Por lo anterior, es importante contar con medios adecuados que permitan conocer y dar seguimiento al estado actual del sistema en operación, en especial para aquellos sistemas en los cuales la interacción del usuario es muy poca.

Existen varias opciones para dar seguimiento a la operación de un sistema. El monitoreo es una técnica muy utilizada en sistemas grandes. Permite hacer una revisión continua del sistema e informa oportunamente de métricas como el uso de memoria, uso de cpu, número de procesos activos, etc y genera alertas cuando alguna de estas métricas sale del rango preestablecido como normal.

Otra técnica común de seguimiento es el logging.

Antecedentes

Logging

El logging es el registro de eventos de interés que ocurren durante la operación de un sistema. Estos eventos pueden ser tales como errores en el sistema, información de accesos de usuarios, la recepción de una petición, el inicio, fin y resultado de un proceso, etc.

Estos registros son generados según se van presentando los eventos en el sistema y se almacenan en un repositorio persistente como un archivo de texto plano o una base de datos.

La información generada por el logging puede ser muy útil si se le da el tratamiento adecuado. El uso del logging tiene las siguientes ventajas:

- Conocer el estado actual del sistema.
- Informar de errores así como facilitar su detección y corrección.
- Depurar el sistema durante la etapa de desarrollo.
- Detectar y solucionar posibles incidentes de seguridad.
- Servir de soporte en tareas de cómputo forense.
- Servir como prueba en la aplicación del principio de no repudio.

El logging ha sido utilizado desde el surgimiento de los primeros sistemas y en la actualidad se sigue empleando ya que continuamente se ha podido comprobar los beneficios que trae consigo, en la operación de los sistemas.

El problema de la implementación del logging dentro de un sistema

En Java la implementación más común del logging es mediante la delegación de servicios a una biblioteca de logging.

Existen muchas bibliotecas que ofrecen implementaciones en Java de servicios de logging. Algunos ejemplos:

- Log4j
- Java Logging API
- Apache commons logging
- SLF4J

En general, una biblioteca de logging tiene la arquitectura mostrada en la Figura 3.1.

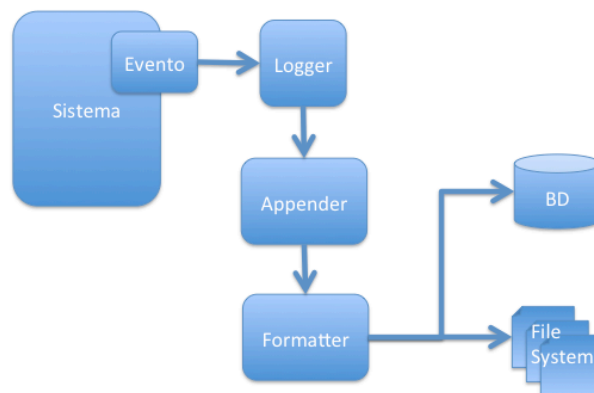

Figura 3.1 Arquitectura de una biblioteca de logging.

Logger: Es el responsable de capturar el mensaje generado por el evento y que se registrará en el log.

Formatter: Es el encargado de dar el formato adecuado, según la configuración previa, al mensaje.

Appender: Una vez generado el mensaje, la biblioteca lo envía al appender adecuado. Un appender escucha mensajes y los publica en alguna de las siguientes formas:

- Lo despliega en la consola.
- Lo escribe en un archivo.
- Lo envía a una base de datos.
- Se envía a una cola de mensajes.
- Se envía por correo.
- Se descarta.

Logging: una responsabilidad cruzada

Cualquier sistema que requiera registrar eventos en un log, puede elegir utilizar alguna de las bibliotecas de logging, que cumpla con sus necesidades.

Sin embargo, un sistema orientado a objetos, para poder utilizar los servicios de logging debe configurar y realizar las invocaciones correspondientes, según lo requiera la biblioteca, como se muestra en el siguiente caso:

```
1  // Realiza su tarea central
2   obj.relizaTarea();
3
4
5
6
7   // Además de su tarea, debe registrar en el log
8
9
10  // Inicializa logger
11  Logger  logger = Logger.getLogger("org.abc");
12
13  logger.setLevel(Level.DEBUG);
14
15  // Registra en el log el evento
16  logger.info("Se ha completado la tarea.");
17  if(debug){
18  //Información para depuración
19     logger.debug("Parametros de entrada:"+param1);
20     logger.debug("Parametros de entrada:"+param2);
21     logger.debug("Parametros de entrada:"+param3);
22  }
```

Se puede observar que el código anterior está muy acoplado a la implementación de logging que se utiliza. El código mostrado cumple con una función central, mostrada por la línea de código número 2. El resto muestra los pasos necesarios para poder obtener un Logger, configurarlo, determinar si se deben generar las bitácoras y posteriormente generar las bitácoras a nivel informativo y de depuración.

A pesar de que en este ejemplo pudo inicialmente haber sido lograda una alta cohesión en términos de la lógica de negocio, al agregar al módulo la funcionalidad necesaria para la generación de bitácoras, fue necesario codificar las líneas del logging. Estas líneas de código son normalmente solicitadas por la biblioteca para poder hacer uso de sus servicios.

Como se expuso anteriormente el logging es una de esas responsabilidades que se utilizan a lo largo de todo el sistema. Por lo tanto, es una responsabilidad

cruzada, es decir, parte de su implementación se encuentra dispersa en varios módulos (Figura 3.2). Por esta razón es normal que se mezcle con las demás responsabilidades.

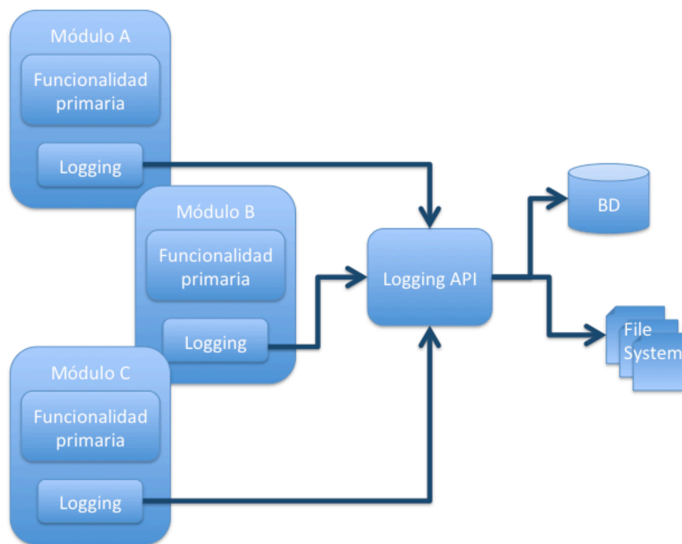

Figura 3.2 Dispersión de la responsabilidad del logging.

Aún existiendo está dispersión de código, el sistema puede trabajar correctamente, cumpliendo con los requerimientos funcionales establecidos. Sin embargo, el problema surge cuando es necesario dar mantenimiento al sistema.

¿Qué pasa cuando se quiere o se tiene que cambiar la implementación del logging? (tal vez por razones ajenas al sistema). Pues bien, la tarea implicaría refactorizar el código del sistema en su totalidad, localizando cada uno de los puntos del mismo donde se utiliza la biblioteca. Este trabajo sin duda puede crecer exponencialmente en función del tamaño del sistema.

Las razones anteriores hacen que un mantenimiento de este tipo pueda llegar a ser muy tardado, traduciéndose en mayores costos, desvío de recursos humanos y muy posiblemente en el retraso de la agenda del proyecto.

Se puede decir que en estos casos donde el logging se implementa de esta manera, no se está cumpliendo con la facilidad de mantenimiento, que cualquier sistema debe tener.

Además de los problemas que se generan en la evolución del sistema, también la mezcla de código provoca que el código presentado, sea más difícil de entender a simple vista. El código central de un módulo se ve entremezclado con el específico del logging. Leer un código de este tipo implica ir filtrando las líneas que no corresponden a la tarea central del módulo analizado.

Esto implica que a un desarrollador de reciente ingreso al grupo de trabajo, le cuesta más trabajo leer y entender el bloque de código que le ha sido asignado. Esto incluso aplica para desarrolladores experimentados que trabajen por primera vez en un módulo específico.

La curva de aprendizaje requerida por el sistema puede presentar una pendiente muy alta dependiendo de la complejidad del código.

Se deduce que si el código es más simple, evitando mezclar el logging con el resto de las responsabilidades, se gana lo siguiente:

- Los mantenimientos serán más rápidos y menos costosos.
- Se podrán integrar nuevos elementos al equipo y obtener resultados más rápidamente de ellos.

Por todo los expuesto anteriormente, surgió la necesidad de lograr una implementación de logging que permita centralizar la lógica especializada en la tarea mencionada.

Dicha implementación debe lograr:

- Separar totalmente el código del logging de las demás responsabilidades.
- Centralizar la administración de los registros generados.
- Facilitar la modificación de la implementación de logging.

Esta implementación se realizó a través del sistema de bitácoras no invasivo. Más adelante se mencionan los detalles de su desarrollo.

4

Sistema de bitácoras no invasivo

4.1

Determinación de objetivos

Como se ha expuesto, el logging juega un papel muy importante dentro de cualquier sistema. Su implementación trae consigo grandes beneficios.

A pesar de esto, el logging pocas veces es considerado durante el proceso de diseño de un sistema. Además, es una responsabilidad que se extiende por casi todo el sistema, es decir, que es requerida para el funcionamiento adecuado de los módulos del sistema.

Por lo anterior es necesario crear un diseño que permita implementar de manera adecuada el logging de un sistema.

Con el sistema desarrollado se pretende :

- Brindar información acerca del estado actual del sistema.
- Facilitar la detección y corrección de errores.
- Generar una evidencia persistente de las actividades realizadas dentro del sistema.

Esto se debe lograr de tal manera que los demás componentes del sistema no se vean afectados por la implementación del logging.

Los componentes deben mantener una independencia de esta implementación. Esta independencia fomentará:

- Bajo acoplamiento: Con el bajo acoplamiento los componentes tendrán una mayor capacidad de reúso.
- Alta cohesión: Dentro de los componentes no se mezclan responsabilidades por lo que éstos son mas sencillos, fáciles de entender y por lo tanto su mantenimiento es más simple y menos costoso.

El objetivo primordial del proyecto es implantar un sistema de logging que no afecte los componentes de un sistema, manteniéndolos simples y enfocados en su tarea principal.

4.2

Análisis de riesgos

Todo desarrollo de software se ve afectado por una gran cantidad de factores, desde los tecnológicos hasta los humanos. Dichos factores pueden representar un riesgo para el proyecto en turno. Por lo cual es necesario identificarlos en etapas tempranas. Esta identificación de riesgos, permitirá conocerlos y dar herramientas para diseñar y poner en marcha un posible plan de eliminación o mitigación de riesgos.

En las siguientes secciones se detalla el proceso de análisis de riesgos para el proyecto actual.

Identificación de riesgos

En la tabla 4.2.1 se listan los riesgos identificados para las siguientes categorías:

- Tecnológicos (Framework, Servidor de aplicaciones, DBMS, la plataforma y el web framework).
- Riesgos con el personal (sobre el reclutamiento, la capacitación y la disponibilidad de éste).
- Riesgos con las herramientas de desarrollo.
- Los riesgos asociados con las estimaciones de tiempo durante el proceso de desarrollo del proyecto.

Tabla 4.2.1 Riesgos identificados

Riesgos tecnológicos	
Framework	Que la implementación de POA no soporte los Joinpoint's requeridos
	Que el framework contenga defectos
	Framework es descontinuado y ya no tiene más soporte por parte de la comunidad que lo desarrolla
Servidor de aplicaciones	Rendimiento inadecuado del servidor
	Sale al mercado una nueva versión del servidor
	Se descontinúe su uso y soporte
Base de datos	Rendimiento inadecuado
	Sale al mercado una nueva versión del DBMS
	Se descontinúe su uso y soporte
Plataforma (Java)	Sale al mercado una nueva versión de la plataforma
	La plataforma presenta defectos y problemas de seguridad conocidos
	La plataforma cae en la obsolescencia
Web Framework	Incompatibilidad con navegadores
	Se descontinúe su uso y soporte

Riesgos con el personal	
Reclutamiento	Dificultad para reclutar gente con los conocimientos necesarios sobre la plataforma, el framework y el paradigma orientado a aspectos
Capacitación	Poca disponibilidad de capacitación para el personal
Disponibilidad	Falta de disponibilidad del equipo de desarrollo por deserciones o enfermedad

Riesgos con las herramientas	
Incompatibilidad	Incompatibilidad de las herramientas con nuevas versiones del framework

Riesgos en la estimación	
Estimación desarrollo	Subestimación de tiempos de desarrollo
Estimación reparación de defectos	Subestimación de tipos de reparación de defectos detectados en el software

Análisis de riesgos identificados

Se evaluaron los riesgos midiendo las probabilidades con las cuales se pueden presentar:

- Bajas (<=30%).
- Medianas (>30 y <=60%).
- Altas (>60 y <=100%).

Y se clasificaron según las consecuencias si llegan a ocurrir:

- Insignificantes: No genera retrasos importantes.
- Tolerables: Retrasos dentro de los límites permitidos.
- Serias: Provoca retrasos mayores.
- Catastróficas: Amenaza la supervivencia del proyecto.

Para este análisis se tomaron en cuenta, experiencias pasadas y el presente de las tecnologías elegidas. A continuación se detallan algunos puntos considerados:

- Framework Spring: Es un framework bastante popular y con una gran comunidad de desarrollo detrás de él. Por lo cual se considera que es bastante estable y es muy probable que su desarrollo continúe por un buen tiempo.
- Servidor de aplicaciones (Glassfish): Servidor de aplicaciones de código abierto. Respaldado por Oracle. Cumple satisfactoriamente con la especificación JEE. Tiene amplias posibilidades de que su desarrollo continúe por parte de la comunidad.
- DBMS (Postgres): De los manejadores de base de datos, con código abierto, más populares. Cuenta con una comunidad desarrolladora bastante activa. Se espera que esta tendencia continúe por buen tiempo.
- Plataforma (Java): La plataforma más popular para desarrollo de aplicaciones empresariales. Tiene el respaldo de Oracle. Últimamente se ha sabido de vulnerabilidades de seguridad bastante graves, que afectan a las aplicaciones desarrolladas con esta plataforma. Se espera que estas vulnerabilidades, así como otros posibles defectos, sean corregidas en próximas versiones.

En la tabla 4.2.2 se muestra el análisis realizado a los riesgos detectados anteriormente, tomando en cuenta las métricas y factores mencionados:

Tabla 4.2.2 Análisis de riesgos identificados

Riesgo	Probabilidades	Consecuencias
Framework: Implementación POA no adecuada	*Medianas*	Serias
Framework: Defectos	*Medianas*	Serias
Famework: Falta de soporte	*Medianas*	Serias
Serv de aplicaciones: Rendimiento inadecuado	*Medianas*	Serias
Serv de aplicaciones: Nueva versión	*Altas*	Tolerables
Serv de aplicaciones: Falta de soporte	*Bajas*	Tolerables
BD: Rendimiento inadecuado	*Medianas*	Serias
BD: Nueva versión	*Altas*	Tolerables
BD: Falta de soporte	*Bajas*	Serias
Plataforma: Nueva versión	*Altas*	Tolerables
Plataforma: Defectos y problemas de seguridad	*Medianas*	Serias
Plataforma: Obsolescencia	*Bajas*	Catastróficas
WebFramework: Incompatibilidad	*Medianas*	Serias
WebFramework: Falta de soporte	*Medianas*	Serias
Personal: Problemas reclutamiento	*Medianas*	Serias
Personal: Falta de capacitación	*Bajas*	Serias
Personal: Falta de disponibilidad	*Altas*	Serias
Herramientas: Incompatibilidad nuevas versiones	*Medianas*	Tolerables
Estimación tiempos de desarrollo	*Medianas*	Serias
Estimación tiempos corrección defectos	*Medianas*	Serias

Plan de mitigación de riesgos

Se ha establecido un plan de acción para tratar de mitigar los efectos de los riesgos detectados. Se pretende que en la medida de lo posible:

- Los riesgos puedan ser evitados con las acciones tomadas.
- En caso de presentarse la situación prevista en un riesgo, se tomen las acciones para mitigar los efectos del mismo.
- En caso de no poder evitar la situación y que los efectos no puedan ser mitigados, se tomará una acción de contingencia para contar una solución alternativa.

En la tabla 4.2.3 se muestra el Plan de mitigación de riesgos.

Tabla 4.2.3 Plan de mitigación de riesgos

Riesgo	Acción
Framework: Implementación POA no adecuada.	Se debe de evaluar extensamente la implementación de POA a utilizar. Tomando en cuenta los requerimientos del proyecto. Buscar alternativas.
Framework: Defectos.	Se documentarán los defectos detectados en la última versión del framework, evaluando su severidad. Esto debe realizarse antes de considerar cambiar la versión del Framework.
Framework: Falta de soporte.	Se debe estar informado, en conjunto con la comunidad desarrolladora del framework, acerca de los planes a futuro del proyecto. Además se debe contar con alternativas a este framework, siempre considerando que se cumpla con los requerimientos iniciales y actuales del sistema, buscando que la modificación del framework sea del menor impacto posible.
Servidor de aplicaciones: Rendimiento inadecuado.	Documentarse acerca de optimizaciones en la configuración del servidor, con el objetivo de encontrar la más adecuada para las necesidades del proyecto. Además de estar en una constante evaluación de alternativas al servidor de aplicaciones actual.
Servidor de aplicaciones: Nueva versión.	Estar al tanto de las nuevas versiones que están en desarrollo por parte del proveedor. Evaluar desde versiones beta la compatibilidad de la nueva versión del servidor de aplicaciones con el sistema actual.
Serv de aplicaciones: Falta de soporte.	Se debe estar al tanto de los planes que tiene el proveedor del servidor de aplicaciones. Además de contar con alternativas confiables de otros productos que pudieran sustituir al servidor de aplicaciones actual.
BD: Rendimiento inadecuado.	Realizar evaluaciones de posibles optimizaciones en la configuración del servidor, así como en los objetos de la base de datos (tablas, índices, etc). Considerar adquirir un hardware más poderoso que cumpla con las necesidades del sistema.

BD: Nueva versión.	Se debe estar al tanto de los planes que tiene el proveedor del DBMS. Además de contar con alternativas confiables de otros productos que pudieran sustituir al DBMS actual.
BD: Falta de soporte.	Se debe estar al tanto de los planes que tiene el proveedor del DBMS. Además de contar con alternativas confiables de otros productos que pudieran sustituir al DBMS actual.
Plataforma: Nueva versión.	Se debe estar al tanto de las nuevas versiones en desarrollo de la plataforma. Se realizarán constantemente pruebas del sistema con las versiones beta de la plataforma, con la finalidad de revisar que el sistema no sea fuertemente afectado por los cambios de versión.
Plataforma: Defectos y problemas de seguridad.	Documentar los defectos y vulnerabilidades de la versión actual de la plataforma. Así como realizar esta evaluación antes de actualizar la versión.
Plataforma: Obsolescencia.	Evaluar si la falta de soporte de la plataforma afecta mucho al sistema actual. Contar con alternativas confiables a la plataforma actual, en caso de que se requiera una migración.
WebFramework: Incompatibilidad.	Evaluar alternativas al web framework actual, buscando que se soporte la mayoría de los navegadores actuales, tratando de cubrir el mayor mercado posible (número de usuarios).
WebFramework: Falta de soporte.	Contar con alternativas que brinden las mismas funcionalidades o cumplan en su mayor parte con ellas.
Personal: Problemas reclutamiento.	Buscar adquirir recursos que puedan ser capacitados. Con lo que en un futuro se contarán con mejores recursos que a su vez apoyarán en las tareas de capacitación de los nuevos miembros del equipo.
Personal: Falta de capacitación.	Buscar opciones de capacitación dentro del mismo equipo de desarrollo. Considerar capacitación en el extranjero.
Personal: Falta de disponibilidad.	Se debe de realizar un traslape en las actividades del equipo de desarrollo. Buscando que todos conozcan el trabajo de los demás y mitigando el efecto que tendría la ausencia de un miembro del equipo.
Herramientas: Incompatibilidad nuevas versiones.	Evaluar alternativas a las herramientas actuales.
Estimación tiempos de desarrollo.	Informarse más detalladamente de los detalles de los requerimientos y de las capacidades del equipo de desarrollo.
Estimación tiempos corrección defectos.	Informarse más detalladamente de los detalles de los requerimientos y de las capacidades del equipo de desarrollo.

El análisis de riesgos presentado, permitió hacer una evaluación de los factores que afectan el desarrollo del proyecto. Estos riesgos identificados y su correspondiente plan de mitigación, fueron de ayuda en la estimación de tiempos de desarrollo y permitieron reducir las posibles desviaciones imprevistas en el proyecto.

4.3

Planificación

Software utilizado

Para el desarrollo del sistema se utilizó el siguiente software:

Plataforma: Java
Plataforma creada por Sun Microsystems. Uno de sus principales componentes es el lenguaje de programación del mismo nombre. Este lenguaje es de propósito general y orientado a objetos, con una sintaxis bastante parecida al lenguaje C, sin tener todas las características de bajo nivel de este último. Su principal objetivo es el ser compilado una vez y ser ejecutado en cualquier plataforma.

Java es una de las plataformas de desarrollo más populares en la actualidad ya que permite crear aplicaciones de escritorio, empresariales, web y para dispositivos móviles.

Entorno de desarrollo: Eclipse
Es un entorno de programación general que se implementa como una plataforma compuesta por uno o más plug-ins. Contiene un conjunto de plug-ins (JDT) que proveen soporte para el desarrollo de cualquier aplicación Java incluyendo plug-ins de Eclipse.

Application Framework: Spring
Spring es el framework más popular de desarrollo de aplicaciones empresariales en Java. Provee soporte "ligero" para este tipo de aplicaciones. Fue creado con el objetivo de simplificar el desarrollo de una aplicación empresarial y hace posible usar simples JavaBean's para realizar tareas reservadas anteriormente para los EJB. Cualquier aplicación Java se puede beneficiar de Spring, no solamente las empresariales, logrando simplicidad, capacidad de pruebas y un bajo acoplamiento.

Servidor de aplicaciones: GlassFish
Es un servidor de aplicaciones de código abierto desarrollado inicialmente por Sun Microsystems. Implementa las tecnologías especificadas por la edición

empresarial de Java (JEE). De hecho es la implementación de referencia de la plataforma.

Sistema gestor de base de datos: PostgreSQL
Es un manejador de base de datos objeto-relacional, de código abierto. Está basado en la implementación original de POSTGRES desarrollado por la Universidad de California.

Soporta gran parte de el estándar SQL, además de ofrecer muchas características modernas, como integridad transaccional y un control de concurrencia multiversión.

Web Framework: DOJO
Es un web framework que establece su plataforma sobre tecnologías estándares como HTML5, CSS3 y JavaScript. Cuenta con una librería de widgets que permite el desarrollo ágil de aplicaciones de internet ricas, siendo soportado por la mayoría de los navegadores modernos.

Puntos de control

Se determinaron los puntos de control del proyecto (Tabla 4.3.1), que servirán para evaluar el avance del proyecto durante su desarrollo. Estos puntos se definieron tomando en cuenta las principales actividades del desarrollo, así como los principales componentes que se prevé formen parte del sistema final.

Tabla 4.3.1 Puntos de control del proyecto

Definición de la arquitectura del sistema
Diseño general del sistema
Diseño de la base de datos
Diseño de la capa de persistencia
Diseño de el módulo de integración POA
Diseño de la capa de negocio
Diseño de la capa de servicios
Diseño de la capa de presentación
Implementación de la base de datos
Implementación de la capa de persistencia
Implementación de el módulo de integración POA
Implementación de la capa de negocio
Implementación de la capa de servicios
Implementación de la capa de vista

Agenda del proyecto

Tomando como referencia los puntos de control determinados anteriormente, se derivaron de ellos un conjunto de tareas, con el objetivo de cumplir con el desarrollo del proyecto en tiempo y forma. Estas tareas están plasmadas en la Tabla 4.3.2.

Tabla 4.3.2 Agenda del proyecto.

Id	Tarea	Duración	Inicio	Fin	Ant
1	Análisis	2 días	25/02/13	27/02/13	
2	Análisis de requerimientos	2 días	25/02/13	27/02/13	
3	Definición de requerimientos Funcionales	1 día	25/02/13	26/02/13	
4	Definición de requerimientos no funcionales	1 día	26/02/13	27/02/13	3
5	Diseño	4 días	27/02/13	05/03/13	
6	Diseño de la Arquitectura general del sistema	1 día	27/02/13	28/02/13	4
7	Diseño del modelo de objetos de dominio	1 día	28/02/13	01/03/13	6
8	Diseño del modelo entidad relación	1 día	01/03/13	04/03/13	7
9	Diseño detalle procedimental e interfaces.	1 día	04/03/13	05/03/13	8
10	Desarrollo	46 días	05/03/13	08/05/13	
11	Persistencia	9 días	05/03/13	18/03/13	
12	Análisis y definición de objetivos	1 día	05/03/13	06/03/13	9
13	Desarrollo	5 días	06/03/13	13/03/13	
14	Implementación modelo E-R	1 día	06/03/13	07/03/13	12
15	Desarrollo objetos de dominio	1 día	07/03/13	08/03/13	14
16	Implementación de pool's de conexiones	1 día	08/03/13	11/03/13	15
17	Desarrollo de DAO's (correspondientes a objetos de dominio)	2 días	11/03/13	13/03/13	16
18	Pruebas	3 días	13/03/13	18/03/13	17
19	Integración AOP	8 días	18/03/13	28/03/13	
20	Análisis y definición de objetivos	1 día	18/03/13	19/03/13	18
21	Desarrollo	5 días	19/03/13	26/03/13	
22	Implementación del ambiente web gestionado por spring	1 día	19/03/13	20/03/13	20
23	Implementación de bean's context aware	1 día	20/03/13	21/03/13	22
24	Implementación de inyección de aspectos en tiempo de ejecución	3 días	21/03/13	26/03/13	23
25	Pruebas	2 días	26/03/13	28/03/13	
26	Pruebas y validación de la carga de aspectos desde la persistencia	1 día	26/03/13	27/03/13	24
27	Pruebas y validación de la correcta aplicación de aspectos a los objetos de negocio	1 día	27/03/13	28/03/13	26
28	Servicios	5 días	28/03/13	04/04/13	
29	Análisis y definición de objetivos	1 día	28/03/13	29/03/13	27
30	Desarrollo	3 días	29/03/13	03/04/13	
31	Implementación de los servicios REST	3 días	29/03/13	03/04/13	29
32	Pruebas	1 día	03/04/13	04/04/13	
33	Pruebas y validación de los servicios	1 día	03/04/13	04/04/13	31
34	Interfaz de usuario	24 días	04/04/13	08/05/13	
35	Análisis	1 día	04/04/13	05/04/13	33
36	Diseño de las interfaces	1 día	05/04/13	08/04/13	35
37	Desarrollo	18 días	08/04/13	02/05/13	
38	Implementación vista de autenticación	1 día	08/04/13	09/04/13	36
39	Implementación vista de bitácoras generadas	5 días	09/04/13	16/04/13	
40	Implementación de vista de notificaciones	3 días	09/04/13	12/04/13	38
41	Implementación de vista de errores	2 días	12/04/13	16/04/13	40
42	Implementación vista alta-baja de bitácoras	3 días	16/04/13	19/04/13	41
43	Implementación vista de estadísticas	9 días	19/04/13	02/05/13	
44	Vista de invocaciones a métodos (datos, graficas)	4 días	19/04/13	25/04/13	42
45	Vista de performance de invocaciones (datos, graficas)	5 días	25/04/13	02/05/13	44
46	Pruebas	4 días	02/05/13	08/05/13	45
47	Verificación	3 días	08/05/13	13/05/13	46
48	Validación	4 días	13/05/13	17/05/13	47

La agenda definida para el proyecto permitió que éste se entregara en tiempo y forma, siempre tomando en cuenta los puntos de control definidos anteriormente.

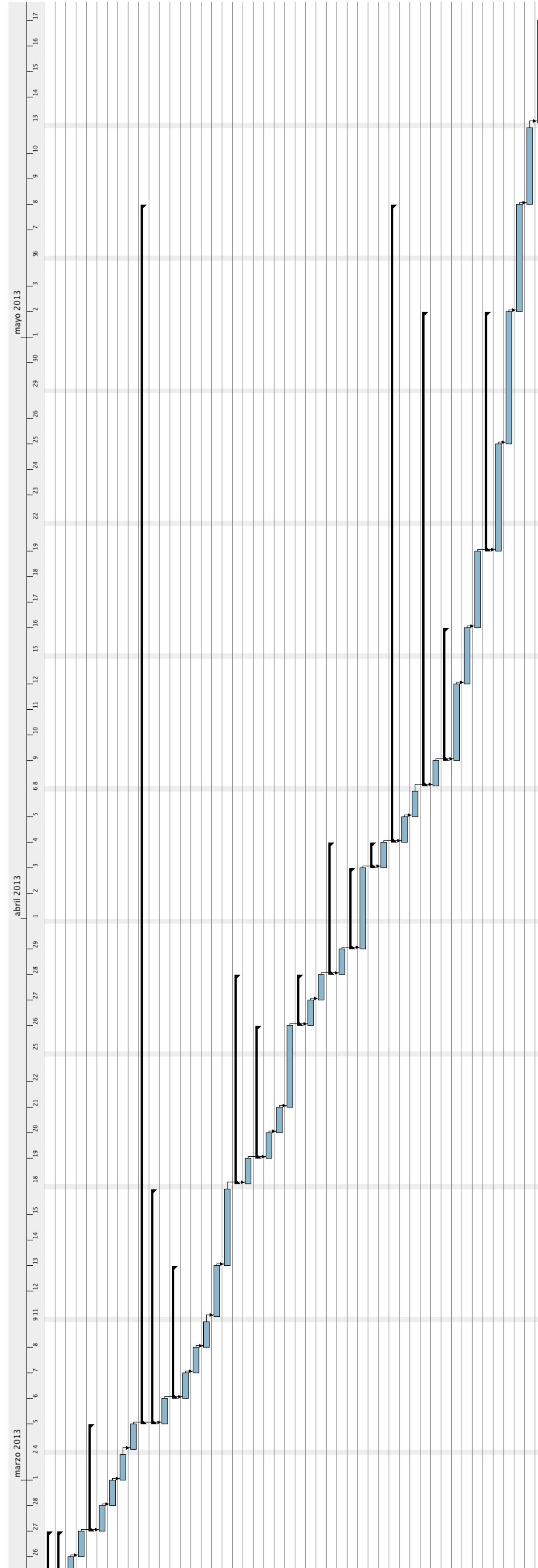

Figura 4.3.1 Diagrama de Gantt de la agenda definida para el proyecto

En la Figura 4.3.1 se muestra el diagrama de Gantt correspondiente a las tareas definidas en la agenda del proyecto. En él se puede apreciar gráficamente el tiempo que se planeó dedicar a cada una de las actividades, además de hacer notar la precedencia de las mismas.

La planeación se hizo tomando en cuenta el modelo de ciclo de vida en espiral. En este modelo, cada ciclo representa una fase del desarrollo y se pueden tener tantos ciclos como lo requiera el proyecto.

Cada componente del sistema está planeado para ser desarrollado en un ciclo de la espiral. Por lo cual para cada uno de ellos se realizó una planeación, un análisis de riesgos, la ejecución del proceso de desarrollo y la evaluación.

En este capítulo se especificó el plan que guió el desarrollo del proyecto. Se establecieron los puntos de control del mismo. Además, se definió el orden y los tiempos de entrega, de cada una de las tareas necesarias para cumplir con todos los objetivos del proyecto.

En los capítulos siguientes se detallan las tareas establecidas en esta sección.

4.4

Desarrollo

4.4.1

Análisis

El sistema propuesto permitirá la generación dinámica de bitácoras, sin la necesidad de modificar el sistema objetivo.

El sistema de bitácoras brindará información valiosa acerca de los eventos que se presentan cotidianamente en el sistema objetivo. Esta información será especialmente útil:

- En el monitoreo diario del sistema.
- En la detección y corrección de errores.
- Como prueba de no repudio de las operaciones realizadas en él.
- En el tratamiento de un incidente de seguridad.

Además, el sistema de bitácoras permitirá que se pueda realizar un monitoreo de la actividad cotidiana en el sistema objetivo.

Este monitoreo se llevará a cabo a través del registro de las invocaciones de los métodos, quedando almacenado el número y la duración de cada una de las invocaciones a los métodos. Esta información permitirá:

- Contar con una estadística de las funciones del sistema más utilizadas.
- Conocer y estar en posibilidad de corregir problemas en el performance de la aplicación.
- Detectar posibles picos en el uso del sistema objetivo.

Glosario

Logs: Registro de eventos en un sistema.

Sistema objetivo: Cualquier sistema al que se le pueden generar logs.

Sistema de bitácoras: El sistema encargado de generar los logs configurados.

Objeto: Unidad base dentro de un sistema orientado a objetos que encapsula información y funcionalidad relacionada con dicha información.

Método: Funcionalidad encapsulada en un objeto que puede o no producir una salida para cero o más entradas.

Invocación: El llamado a ejecución de un método de un objeto.

Requerimientos

1. El sistema debe acoplarse a una aplicación para generar logs del sistema, según se necesite.

2. El sistema debe contar con un control de acceso de usuarios.

3. El sistema debe permitir visualizar los métodos dentro del sistema objetivo, a los cuales se les pueden generar logs.

4. El sistema debe contar con la opción de generar logs para cierto método, tomando como base la lista de métodos del punto 3.

5. El sistema permitirá visualizar los logs que se han generado.

6. El sistema permitirá habilitar el monitoreo de ejecución de métodos, en el cual se registrará:
    - Método invocado.
    - Estatus éxito/error.
    - La duración de la ejecución del método.

7. El sistema permitirá visualizar las estadísticas de ejecución de los métodos.
    - Los más utilizados (número de invocaciones).
    - Los de mayor tiempo de ejecución.

8. El sistema mostrará gráficas de número de ejecuciones por minuto y la duración de cada una de las ejecuciones.

Casos de uso

Tomando en cuenta los requerimientos listados, se modelaron un conjunto de casos de uso. Se utilizó UML para modelar dichos casos de uso (Figura 4.4.1.1)

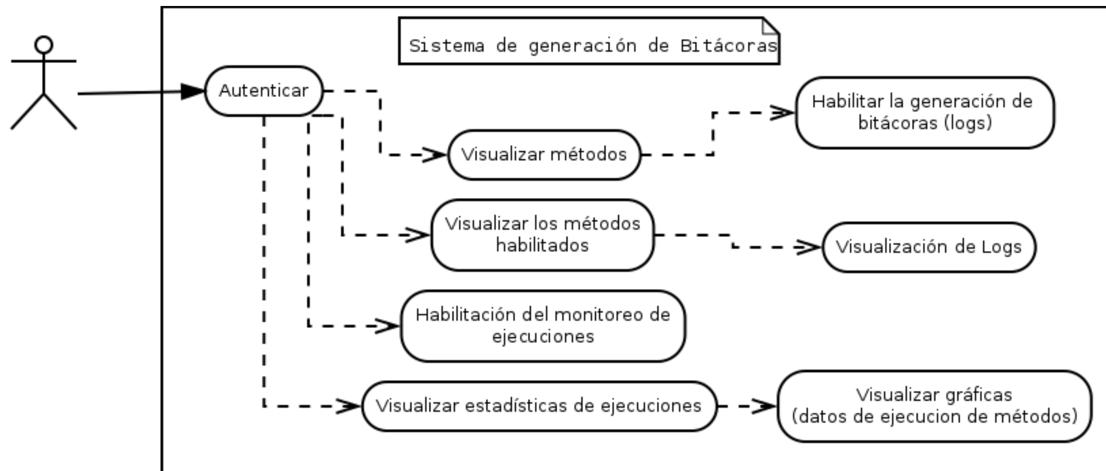

Figura 4.4.1.1 Casos de uso del sistema

En las tablas siguientes se detallan cada uno de los casos de uso del sistema:

Tabla 4.4.1.1 Autenticación de usuarios.

Función	**Autenticación de usuarios**
Descripción	Permite autenticar a un usuario, determinando si éste debe o no tener acceso al sistema.
Entradas	Nombre de usuario y password.
Salidas	El acceso al sistema o la denegación del mismo.
Acción	Es la entrada al sistema.
	El usuario brinda su nombre y su password.
	El usuario solicita acceso.
	El sistema determina si los datos son correctos y si el usuario tiene permitido el acceso.
	Se otorga el acceso.
	En caso negativo el usuario puede reintentar el acceso otras 4 veces.
	En el quinto intento fallido de acceso, el sistema bloqueará al usuario.

Tabla 4.4.1.2 Visualización de métodos.

Función	Visualización de métodos
Descripción	Permite visualizar todos los métodos a los cuales se les pueden aplicar la generación de bitácoras dentro de un sistema.
Entradas	Ninguna o el nombre de un objeto o método.
Salidas	Los objetos o métodos resultantes de la búsqueda.
Acción	El usuario ingresa el nombre de un objeto o método.
	El usuario solicita la búsqueda del objeto o método.
	El sistema presenta los objetos junto con sus métodos asociados que cumplen los criterios de búsqueda.
	Si no se ingresa ningún criterio el sistema muestra todos los objetos y sus métodos.
	En caso de no existir información el sistema lo indicará así.

Tabla 4.4.1.3 Habilitar la generación de bitácoras.

Función	Habilitar la generación de bitácoras
Descripción	Permite activar la generación de bitácoras (logs) para un método específico.
Entradas	El método, el mensaje a presentar, bandera indicando si se visualizarán los parámetros, bandera indicando si la entrada en la bitácora se generará antes de la ejecución, después de la ejecución, después de una ejecución exitosa o después de una ejecución fallida, bandera indicando si se presentará mensaje de error en caso de existir.
Salidas	Éxito o error en la habilitación.
Acción	Previamente el usuario seleccionó un método (visualización de métodos).
	El usuario definirá los parámetros de entrada.
	El usuario solicitará la habilitación de la generación de bitácoras para el método seleccionado.
	El sistema habilitará la generación de bitácoras según los parámetros de entrada.
	A partir de ese momento se iniciará la generación de bitácoras según lo indiquen los parámetros de entrada.
	El sistema reportará el éxito o fracaso de la habilitación.

Tabla 4.4.1.4 Visualización de métodos habilitados para la generación de bitácoras.

Función	Visualización de métodos habilitados
Descripción	Permite visualizar los métodos para los cuales se encuentra habilitada la generación de bitácoras.
Entradas	Ninguna o el nombre de un objeto o método.
Salidas	Los objetos o métodos resultantes de la búsqueda.
Acción	El usuario ingresa el nombre de un objeto o método.
	El usuario solicita la búsqueda del objeto o método.
	El sistema presenta los objetos junto con sus métodos asociados que cumplen los criterios de búsqueda.
	Si no se ingresa ningún criterio el sistema muestra todos los objetos y sus métodos.
	En caso de no existir información el sistema lo indicará así.

Tabla 4.4.1.5 Visualización de logs.

Función	Visualización de logs
Descripción	Permite visualizar los logs generados para uno o varios métodos en un cierto rango de tiempo.
Entradas	Uno o varios métodos, rango de fechas.
Salidas	Logs generados en el rango de fechas especificado para los métodos definidos en los parámetros.
Acción	El usuario ingresará los datos de los parámetros de entrada.
	Solicitará la búsqueda de los logs.
	El sistema mostrará los logs que cumplan con los criterios de búsqueda, en caso de existir.
	En caso de que no existan datos el sistema lo indicará.

Tabla 4.4.1.6 Habilitación del monitoreo de ejecuciones de métodos.

Función	Habilitación del monitoreo de ejecuciones
Descripción	Permite habilitar o deshabilitar el monitoreo de las invocaciones de métodos, así como la duración de cada invocación.
Entradas	Bandera indicando si se habilita o deshabilita el monitoreo.
Salidas	Éxito o fracaso de la operación.
Acción	El usuario indicará si desea habilitar o deshabilitar el monitoreo.
	El usuario solicitará la modificación del estatus del monitoreo.
	El sistema modificará el estatus del monitoreo.
	El sistema informará del éxito o fracaso de la operación.

Tabla 4.4.1.7 Visualización de las estadísticas de ejecuciones de métodos.

Función	Visualización de estadísticas de ejecuciones
Descripción	Muestra los métodos, el número de invocaciones que han tenido, además de la duración total de las invocaciones y su duración promedio.
Entradas	Uno o varios métodos, rango de fechas.
Salidas	Los datos de las invocaciones de los métodos, según lo indican las entradas.
Acción	El usuario definirá las entradas (criterios de búsqueda).
	El usuario solicitará la búsqueda de las estadísticas.
	El sistema mostrará las estadísticas disponibles para los métodos según los criterios de búsqueda.
	En caso de no existir información el sistema lo indicará.

Tabla 4.4.1.8 Visualización de las gráficas, con los datos de las ejecuciones de métodos.

Función	Visualización de las gráficas
Descripción	Se mostrarán gráficas presentando los datos de las ejecuciones totales por minuto y la duración promedio de las ejecuciones en un minuto.
Entradas	Un método específico.
Salidas	Los datos de las gráficas.
Acción	El usuario seleccionará un método (visualización de estadísticas de ejecuciones).
	El usuario solicitará la información de ejecuciones del método.
	El sistema presentará las gráficas con la información de las ejecuciones del método seleccionado.
	En caso de no existir información el sistema así lo indicará.
	Las gráficas serán interactivas, con el fin de que el usuario pueda navegar por los datos mostrados.

Los casos de uso mostrados, son los resultantes del análisis realizado sobre los requerimientos del sistema. El cumplimiento del desarrollo de estos casos permitió que los objetivos del proyecto se cumplieran.

La funcionalidad detectada en esta etapa, servirá como punto de entrada para el diseño del sistema, como se detalla en el siguiente tema.

4.4.2

Diseño

Se definirá la arquitectura del sistema y tomando como base los requerimientos definidos anteriormente, se diseñará el modelo de la base de datos.

Posteriormente se definen las clases, agrupadas en capas, las cuales fueron identificadas en la definición de la arquitectura.

Por último, se definen las interacciones de las clases, mediante diagramas de secuencia, diagramas que modelan cada uno de los casos de uso identificados anteriormente.

Arquitectura

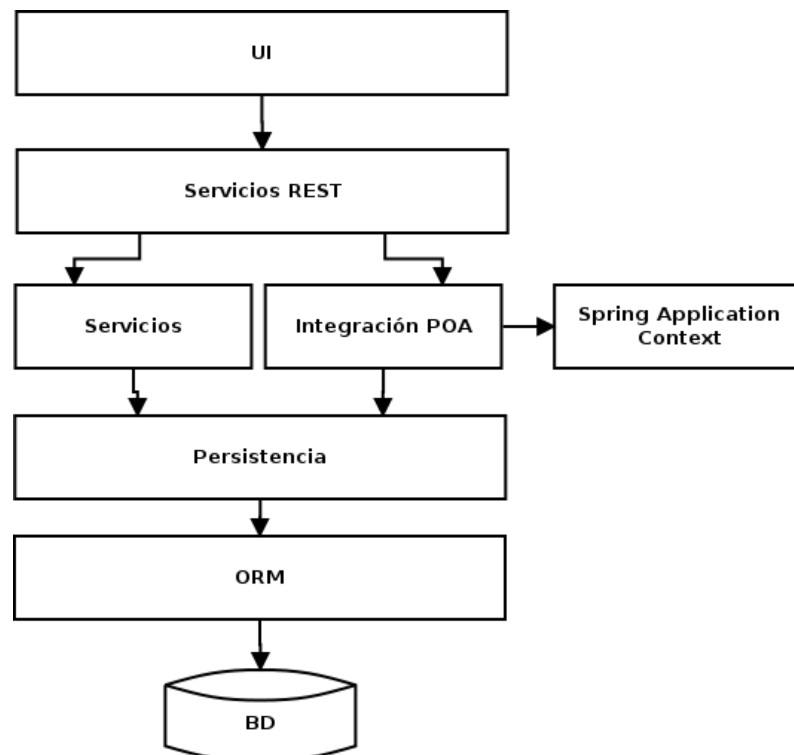

Figura 4.4.2.1 Arquitectura del sistema.

Para definir la arquitectura del sistema se utilizó el patrón arquitectónico de capas. Este patrón se aplica comúnmente en desarrollos de gran tamaño. Tiene el objetivo de dividir las responsabilidades y hacer más manejable la complejidad del sistema. Las responsabilidades se agrupan por categoría, creando una capa por cada categoría.

En la figura 4.4.2.1 se muestran las capas que fueron definidas en la arquitectura del sistema. A continuación se describe cada una de ellas:

- Capa de presentación o UI (user interface): Permite al usuario final interactuar con el sistema a través de una interfaz sencilla e intuitiva.
- Capa de servicios REST: Es una capa web que expone los servicios del sistema como servicios REST. Estos servicios serán utilizados por la capa de presentación.
- Capa de servicios: Es un conjunto de servicios que proveen la funcionalidad específica del sistema. Son utilizados por los servicios REST.
- Capa de integración POA: Un conjunto de servicios que proveen acceso al Application Context de Spring. Son utilizados por los servicios REST.
- Capa de persistencia: Ofrece un grupo de clases que abstraen el acceso a datos. Los utilizan las capas superiores que requieren información de la base de datos.

Diagrama entidad relación

En la Figura 4.4.2.2, se muestra el diagrama entidad-relación que se modeló como guía para la implementación de la base de datos.

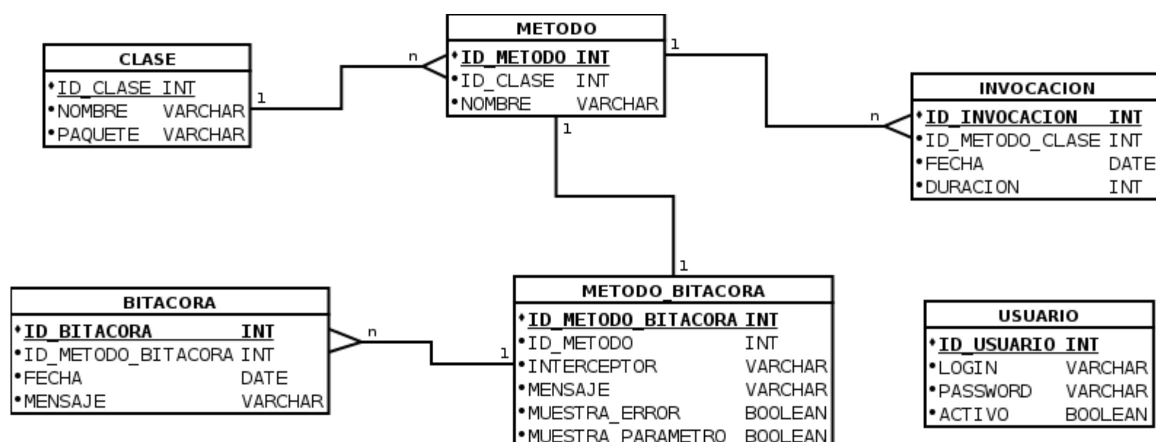

Figura 4.4.2.2 Diagrama entidad-relación del sistema

Descripción entidades

CLASE
Entidad que almacena los datos que representan clases de un sistema orientado a objetos.

- ID_CLASE: Identificador de la clase.
- NOMBRE: Nombre de la clase.
- PAQUETE: Paquete al que pertenece la clase.

METODO
Entidad que contiene los datos que representan un método de una clase.

- ID_METODO: Identificador del método.
- ID_CLASE: Referencia al identificador de la clase, a la cual pertenece el método.
- NOMBRE: Nombre del método.

METODO_BITACORA
Entidad que contiene los datos (configuración) de los métodos habilitados para generación de bitácoras.

- ID_METODO_BITACORA: Identificador del registro.
- ID_METODO: Identificador del método asociado, al cual se le generan bitácoras.
- INTERCEPTOR: Define si la bitácora se genera antes, después de una ejecución exitosa o después de una ejecución fallida. “ANTES”, “DESPUÉS” o “ERROR”, respectivamente.
- MENSAJE: Cadena de texto que se muestra cuando se genere la bitácora.
- MUESTRA_ERROR: Bandera que indica si debe mostrarse el mensaje de error, en caso de presentarse durante la ejecución del método.
- MUESTRA_PARAMETRO: Bandera que indica si se despliegan en la bitácora los valores de los parámetros de entrada del método, para fines de auditoría.

BITACORA
Entidad que contiene los datos de las bitácoras generadas por el sistema, estas bitácoras se generan cada vez que se invoca el método asociado en METODO_BITACORA.

- ID_BITACORA: Identificador de la bitácora.
- ID_METODO_BITACORA: Identificador de los metadatos de la generación de bitácoras.
- FECHA: Fecha y hora en la cual se generó la bitácora.
- MENSAJE: Cadena de texto generada, con base en los metadatos asociados en METODO_BITACORA.

INVOCACION
Entidad que almacena los datos correspondientes a cada una de las invocaciones de los métodos, se relaciona con METODO.

- ID_INVOCACION: Identificador de la invocación.
- ID_METODO_CLASE: Identificador del método, al cual corresponde la invocación registrada.
- FECHA: Fecha y hora en la cual se realizó la invocación.
- DURACION: Cantidad de milisegundos que tomó en completarse la invocación del método.

USUARIO
Entidad que almacena los datos correspondientes de los usuarios registrados en el sistema.

- ID_USUARIO: Identificador del usuario.
- LOGIN: Identificador alfa-numérico del usuario. Nombre de usuario.
- PASSWORD: Cadena alfa-numérica de seguridad que permite autenticar al usuario.
- ACTIVO: Estatus del usuario, sólo los usuarios activos podrán hacer uso del sistema.

Diseño de interfaces

Cada capa definida anteriormente en la arquitectura del sistema debe contar con una interfaz pública. Esta interfaz será utilizada por capas superiores y permitirá que el desarrollo del sistema pueda ser en paralelo. En la Figura 4.4.2.3 se muestran las interfaces diseñadas agrupadas por capa.

Servicios

<<Interfaz>>
UsuarioService
+usuarioValido(login:String,password:String): boolean
+agregaUsuario(login:String,password:String)
+actualizaUsuario(login:String,password:String, estatus:boolean)
+eliminaUsuario(login:String)

<<Interfaz>>
BitacoraService
+obtenBitacora(metodo:MetodoTO,fechaInicio:Date, fechaFin:Date): List<BitacoraTO>

<<Interfaz>>
MonitoreoService
+habilitarMonitoreo(habilitar:boolean)
+obtenEstadisticasGeneral(): List<EstadisticaTO>
+obtenEstadisticasMetodo(metodo:MetodoTO): List<InvocacionTO>

<<Interfaz>>
MetodoService
+habilitaBitacora(metodoBitacora:MetodoBitacoraTO)
+obtenMetodos(clase:ClaseTO): List<MetodoTO>
+obtenMetodosHabilitados(clase:ClaseTO): List<MetodoTO>

<<Interfaz>>
ObjetosService
+obtenerObjetos(): Map<String,Object>

Integración POA

<<Interfaz>>
FachadaContextManager
+obtenObjetosContexto(): Map<String,Object>

Persistencia

<<Interfaz>>
ClaseDao
+obtenClases(): List<ClaseTO>
+obtenClasePorId(idClase:int): ClaseTO
+insertaClase(clase:ClaseTO)
+eliminaClase(clase:ClaseTO)
+obtenClasePorNombre(nombreClase:String): ClaseTO
+actualizaClase(clase:ClaseTO)

<<Interfaz>>
MetodoBitacoraDao
+obtenMetBitPorMetodo(metodo:MetodoTO): List<MetodoBitacoraTO>
+insertaMetodoBitacora(metodoBitacora:MetodoBitacoraTO)
+eliminaMetodoBitacora(metodoBitacora:MetodoBitacoraTO)
+actualizaMetodoBitacora(metodoBitacora:MetodoBitacoraTO)
+obtenMetodoBitacoraTodos(): List<MetodoBitacoraTO>

<<Interfaz>>
MetodoDao
+obtenMetodosPorClase(clase:ClaseTO): List<MetodoTO>
+obtenMetodoPorId(idMetodo:int): MetodoTO
+insertaMetodo(metodo:MetodoTO)
+eliminaMetodo(metodo:MetodoTO)
+actualizaMetodo(metodo:MetodoTO)

<<Interfaz>>
BitacoraDao
+obtenBitacoraPorMetodo(metodo:MetodoTO): List<BitacoraTO>
+insertaBitacora(bitacora:BitacoraTO)

<<Interfaz>>
UsuarioDao
+obtenUsuario(login:String): UsuarioTO
+insertaUsuario(usuario:UsuarioTO)
+actualizaUsuario(usuario:UsuarioTO)

<<Interfaz>>
EstadisticaDao
+obtenEstadisticasGeneral(): List<EstadisticaTO>
+obtenEstadisticasMetodo(metodo:MetodoTO): List<InvocacionTO>
+insertaEstadisticaMetodo(estadistica:EstadisticaTO)

Figura 4.4.2.3 Interfaces por capa.

Diagrama de clases

Clases de dominio

Clases correspondientes a los objetos de dominio. Encargados de encapsular y transportar los datos entre las distintas capas del sistema (Figura 4.4.2.4).

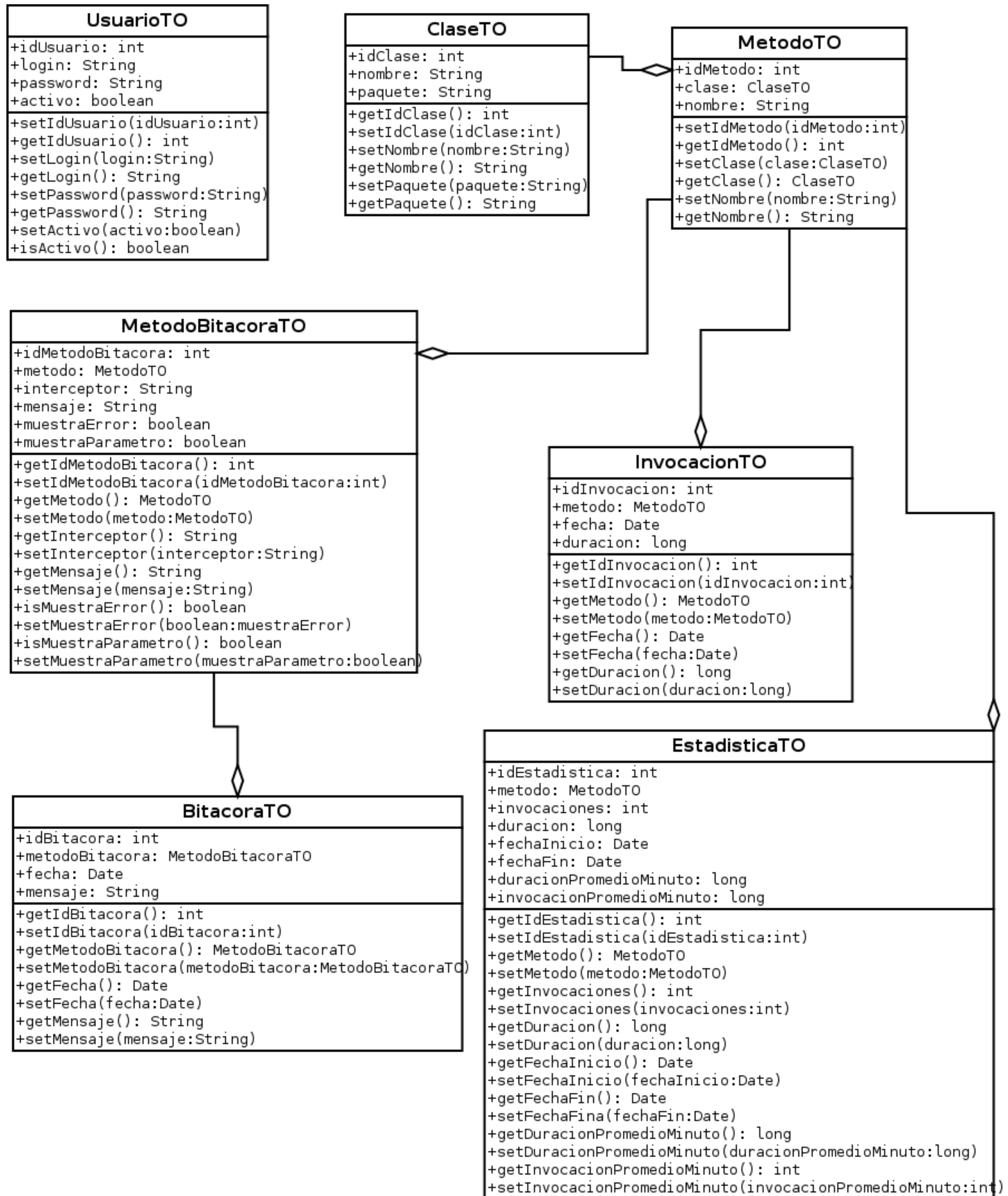

Figura 4.4.2.4 Diagrama de clases de los objetos de dominio.

UsuarioTO: Encapsula los datos de los usuarios.

ClaseTO: Encapsula los datos de las clases registradas en el sistema.

MetodoTO: Encapsula los datos de los métodos registrados en el sistema. Mantiene una referencia hacia la clase (*ClaseTO*) a la cual pertenece el método.

MetodoBitacoraTO: Encapsula los datos de los métodos registrados para la generación de bitácoras. Mantiene una referencia al método al cual pertenece la configuración (*metodo*).

BitacoraTO: Encapsula los datos de un registro en la bitácora del sistema. Mantiene una referencia hacia la configuración de la bitácora (*metodoBitacora*).

InvocacionTO: Encapsula los datos de una invocación realizada a algún método del sistema. Mantiene una referencia al método al cual corresponde la invocación (*metodo*).

EstadisticaTO: Encapsula los datos del total de invocaciones de un método, tales como el número invocaciones y duraciones promedio. Mantiene una referencia del método al cual corresponden.

Persistencia

Conjunto de clases que se abstraen el acceso a datos (Figura 4.4.2.5).

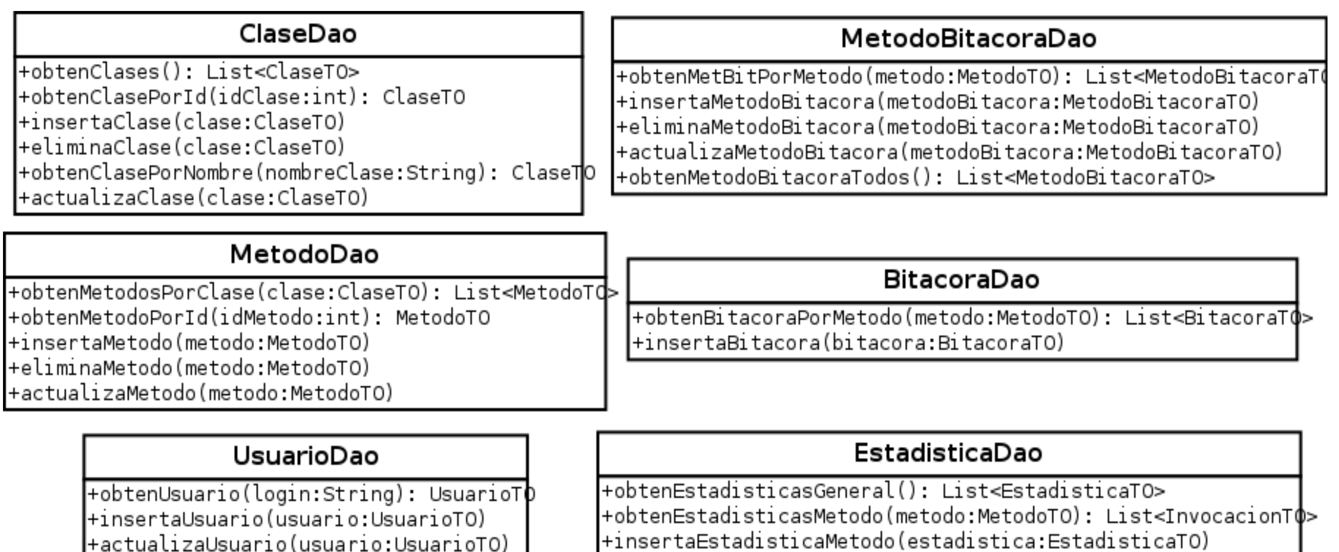

Figura 4.4.2.5 Diagrama de clases de la capa de persistencia.

ClaseDao: Clase que se encarga del acceso a datos correspondiente a la entidad *CLASE*. Hace un mapeo hacia la clase **ClaseTO**.

MetodoDao: Clase que se encarga del acceso a datos correspondiente a la entidad *METODO*. Hace un mapeo hacia la clase **MetodoTO**.

UsuarioDao: Clase que se encarga del acceso a datos correspondiente a la entidad *USUARIO*. Hace un mapeo hacia la clase **UsuarioTO**.

MetodoBitacoraDao: Clase que se encarga del acceso a datos correspondiente a la entidad *METODO_BITACORA*. Hace un mapeo hacia la clase **MetodoBitacoraTO**.

BitacoraDao: Clase que se encarga del acceso a datos correspondiente a la entidad *BITACORA*. Hace un mapeo hacia la clase **BitacoraTO**.

EstadisticaDao: Clase que se encarga de obtener los datos correspondientes a las invocaciones de los métodos dentro de la aplicación. Utiliza la entidad *INVOCACION.* Devuelve objetos de tipo **EstadisticaTO** e **InvocacionTO**.

Integración POA

Conjunto de clases que se encargan de interactuar con el contenedor de Spring (ApplicationContext). Tienen funciones como obtener objetos del contenedor o inyectar proxy's a los cuales se les aplicó un aspecto (Figura 4.4.2.6).

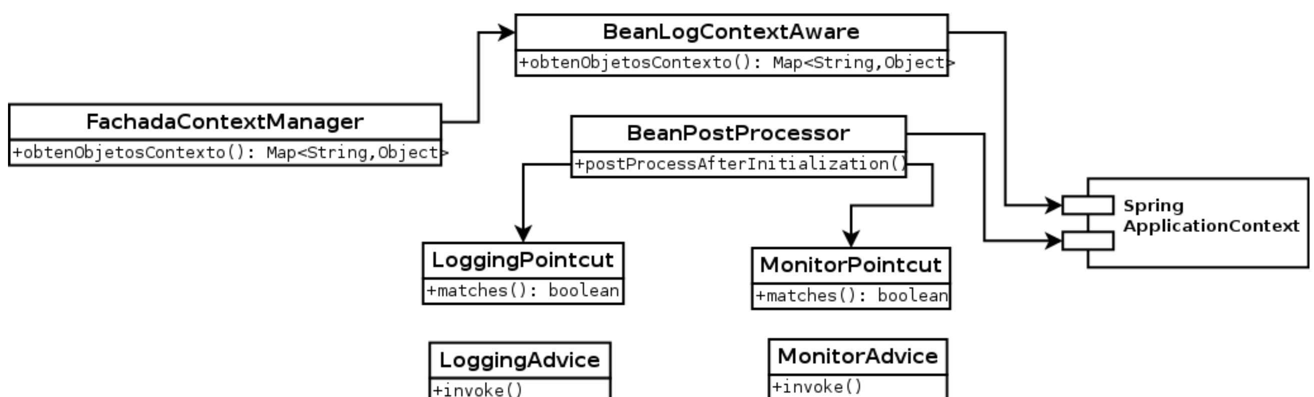

Figura 4.4.2.6 Diagrama de clases de la capa de integración.

FachadaContextManager: Clase que funge como fachada del subsistema. Se encarga de solicitar al ***BeanLogContextAware*** los objetos existentes dentro del contenedor y los devuelve al cliente.

BeanLogContextAware: Clase que se encarga de interactuar con el contenedor. Mantiene una referencia al contenedor. Devuelve los objetos registrados en el contenedor y registra al objeto ***BeanPostProcessor***.

BeanPostProcessor: Clase que se encarga de inyectar al contenedor los proxy's generados tomando en cuenta los criterios definidos en los Pointcut's.

LoggingPointcut: Clase que contiene el criterio para la generación de proxy's (aspectos). Carga las definiciones desde ***MetodoBitacoraDao,*** que define los objetos y métodos a los cuales se les generarán bitácoras.

LoggingAdvice: Clase que contiene la lógica de generación de bitácoras. Este advice será aplicado a todos los objetos-métodos que defina ***LoggingPointcut.***

MonitorPointcut: Clase que contiene el criterio para la generación de los proxy's. Normalmente este pointcut permitirá que se creen proxy's para todos los objetos de la aplicación y de esta manera poder monitorearlos.

MonitorAdvice: Clase que contiene la lógica para el monitoreo de las invocaciones de los métodos de la aplicación. Este advice será aplicado por ***MonitorPointcut*** a todos los objetos en el contenedor, con lo que se podrán registrar sus invocaciones mediante ***EstadisticaDAO***.

Servicios

Clases que exponen la funcionalidad central del sistema. Este conjunto de servicios podrá ser expuesto mediante otra interfaz (en el caso de la arquitectura definida anterioremente: REST services), para ser utilizados por la vista y permanecer desacoplados de ésta (Figura 4.4.2.7).

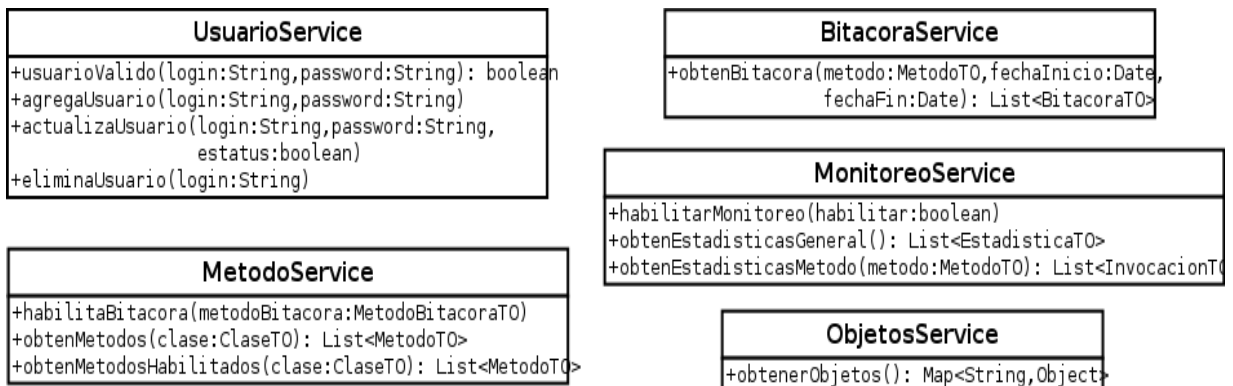

Figura 4.4.2.7 Diagrama de clases de la capa de servicios.

UsuarioService: Encapsula la funcionalidad específica de la gestión de usuarios. Permite validar, agregar, actualizar o eliminar la información de un usuario.

ObjetosService: Se encarga de obtener los objetos del contenedor de Spring, consulta ***FachadaContextManager*** para obtener la lista de objetos.

MetodoService: Contiene la funcionalidad relacionada con la gestión de los métodos dentro del sistema. Permite consultar: los métodos disponibles dentro del sistema y los métodos que han sido habilitados para la generación de bitácoras. También permite habilitar la generación de bitácoras sobre un método.

BitacoraService: Contiene la funcionalidad para obtener los registros generados en la bitácora, para un método específico y un cierto rango de fechas.

MonitoreoService: Contiene la funcionalidad para gestionar el monitoreo de invocaciones dentro del sistema. Permite habilitar/deshabilitar el monitoreo, obtener el resumen general de las invocaciones y obtener el conjunto de estadísticas de invocaciones para un método.

Detalle procedimental

A continuación se describen las interacciones entre las clases descritas anteriormente (detalle procedimental), esta interacción es de tipo dinámica, es decir, en tiempo de ejecución.

Función: Autenticación
Describe la interacción entre la interfaz de usuario, el servicio Rest de usuarios, el servicio de usuarios y el objeto de acceso a datos de los usuarios. Estos objetos se coordinan para lograr la autenticación de un usuario (Figura 4.4.2.8).

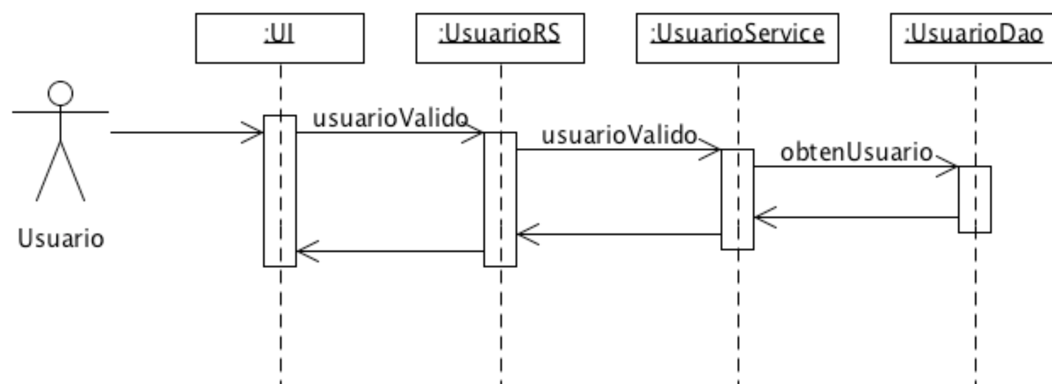

Figura 4.4.2.8 Diagrama de secuencia de la autenticación de un usuario.

Función: Visualización de métodos
Describe la interacción de la interfaz de usuario con los componentes necesarios para visualizar los métodos existentes en el contenedor de Spring (Figura 4.4.2.9).

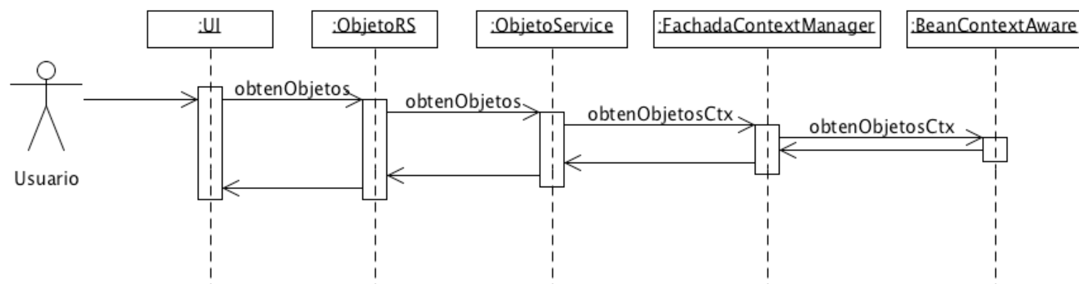

Figura 4.4.2.9 Diagrama de secuencia de la visualización de métodos dentro del contenedor de Spring.

Función: Habilitación bitácoras
Describe la interacción de la interfaz de usuario con el servicio Rest y los demás objetos necesarios para la habilitación de bitácoras (Figura 4.4.2.10).

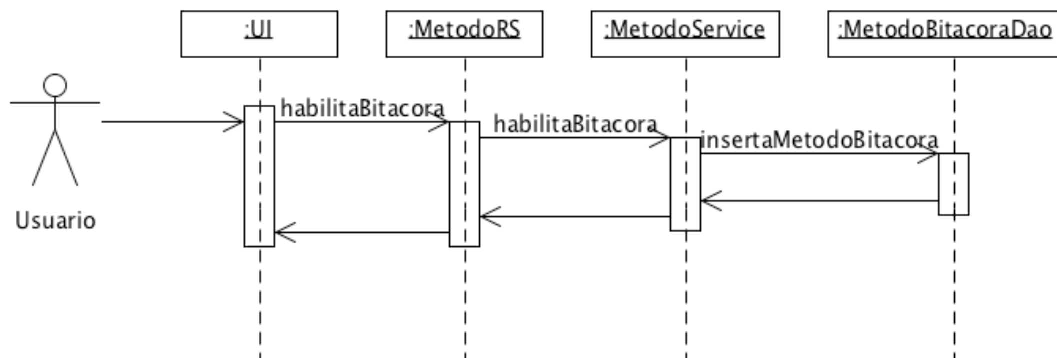

Figura 4.4.2.10 Diagrama de secuencia de la habilitación de la generación de bitácoras.

Función: Visualización métodos habilitados
Describe la interacción de la interfaz de usuario con el servicio Rest correspondiente. Esto con el objetivo de mostrar los métodos habilitados para generar bitácoras (Figura 4.4.2.11).

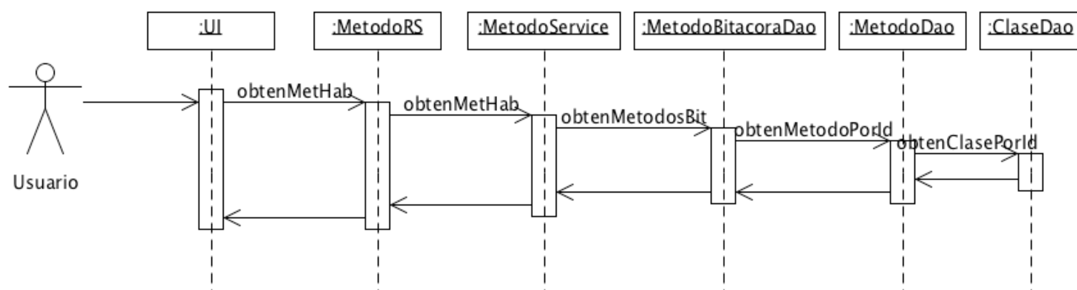

Figura 4.4.2.11 Diagrama de secuencia de la visualización de métodos habilitados para la generación de bitácoras.

Función: Visualización de logs
Describe la interacción de los componentes, cuyo objetivo es mostrar en la interfaz de usuario, los registros generados en la bitácora (Figura 4.4.2.12).

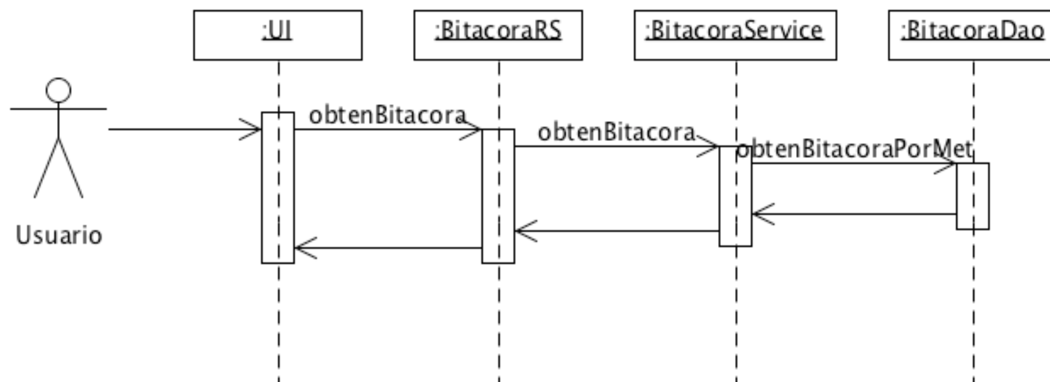

Figura 4.4.2.12 Diagrama de secuencia de la visualización de bitácoras.

Función: Habilitación del monitoreo de invocaciones
Describe la interacción de los componentes, cuyo objetivo es habilitar el monitoreo de invocaciones de los métodos del sistema (Figura 4.4.2.13).

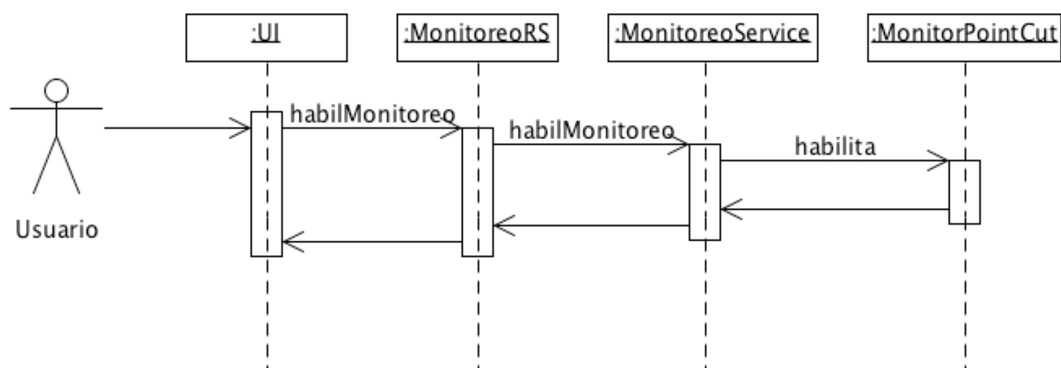

Figura 4.4.2.13 Diagrama de secuencia de la activación del monitoreo de invocaciones de métodos.

Función: Visualización estadísticas invocaciones
Describe la interacción de los componentes, cuya finalidad es mostrar en la interfaz de usuario, las estadísticas de invocaciones (Figura 4.4.2.14).

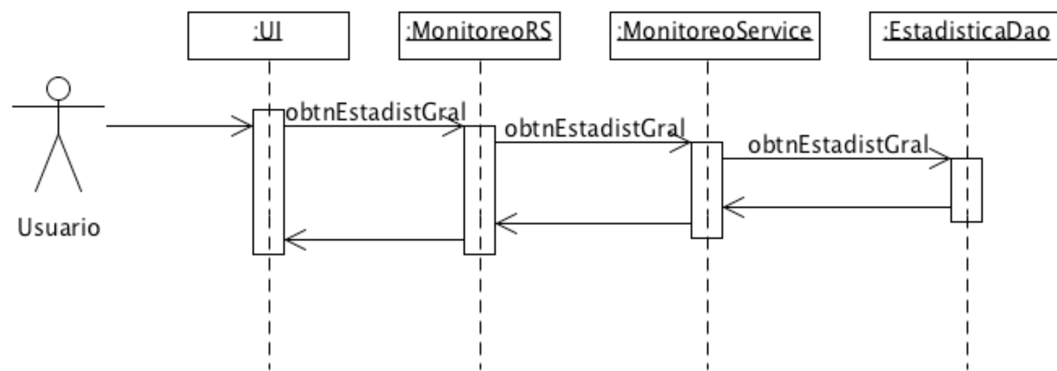

Figura 4.4.2.14 Diagrama de secuencia de la visualización de estadísticas de invocaciones.

Función: Visualización gráficas
Describe la interacción de los componentes, cuyo objetivo es extraer los datos de las invocaciones de un método y generar la gráfica que facilite la visualización de esta información (Figura 4.4.2.15).

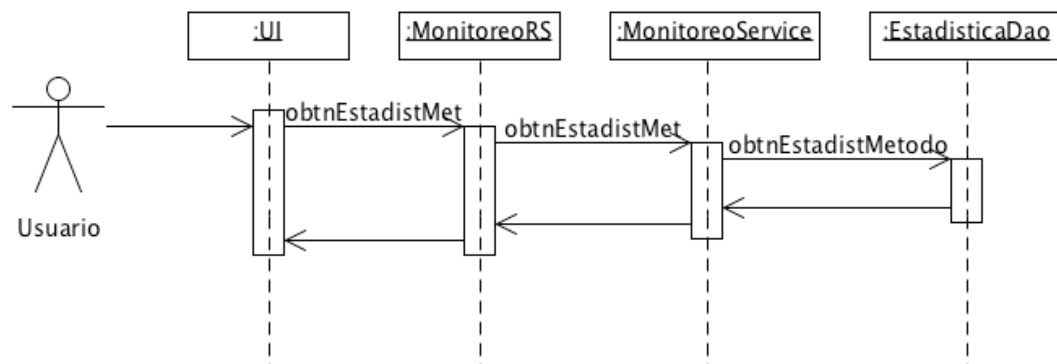

Figura 4.4.2.15 Diagrama de secuencia de la visualización de gráficas.

Función: Integración POA – Creación de Proxy's
Diagrama de la secuencia de creación de objetos proxy en el ApplicactionContext, tomando como base la configuración hecha anteriormente (Figura 4.4.2.16). La configuración se toma de MetodoBitacoraDao.

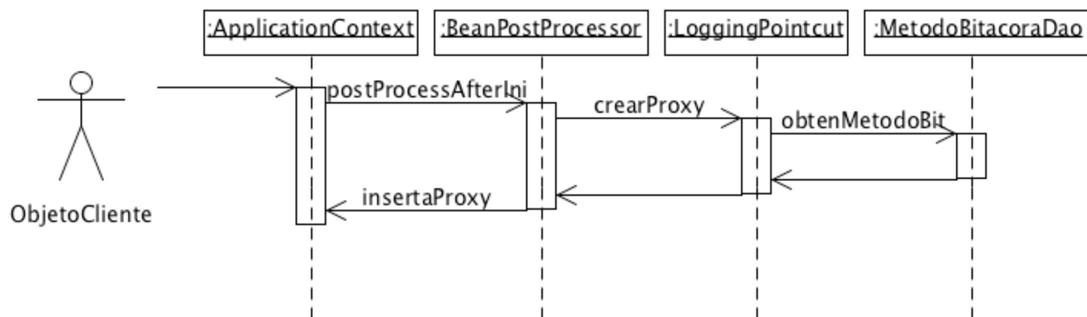

Figura 4.4.2.16 Diagrama de secuencia del proceso de creación de proxy's dentro del contenedor de Spring.

Función: Integración POA – Generación de bitácoras
Diagrama de la secuencia de la generación de bitácoras. Las bitácoras se generarán antes y/o después de la invocación del objeto target, dependiendo de la configuración obtenida de *MetodoBitacoraDao* (Figura 4.4.2.17).

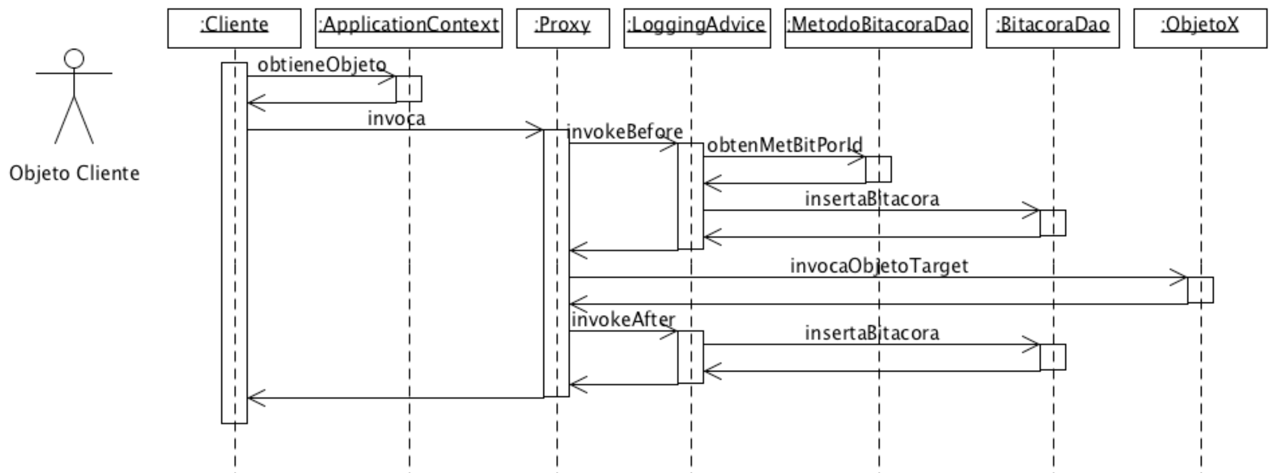

Figura 4.4.2.17 Diagrama de secuencia de la generación de bitácoras mediante POA.

Como resultado del diseño, se obtuvieron un conjunto de diagramas. Estos diagrama sirvieron como guía para las etapas posteriores del desarrollo del proyecto.

También estos diagramas forman parte de la documentación de la arquitectura del sistema. Como se ha mencionado, esta documentación permitirá informar a las personas interesadas en el proyecto, cuáles son las estructuras estáticas y dinámicas que conforman el sistema.

4.4.3

Implementación

La implementación de una solución es la etapa del desarrollo de software en la cual los modelos, algoritmos, estándares y diseños son construidos. Esta realización es llevada a cabo mediante la programación y la integración de componentes de software preexistentes. El resultado de la implementación es una versión ejecutable del sistema que deberá cumplir con los requerimientos establecidos inicialmente.

A continuación se detalla la implementación del proyecto del sistema generador de bitácoras.

Esta implementación comprende desde la capa de persistencia, hasta la capa encargada de la presentación de los datos al usuario final.

Implementación de la base de datos

La persistencia en una aplicación de tipo empresarial, es fundamental para almacenar la información relevante para el sistema. Esta información se mantiene de forma persistente en la base de datos, más allá de peticiones o sesiones del usuario.

Es importante que la base de datos del sistema, sea administrada por un sistema especializado para este fin. Para tal requerimiento, existen los sistemas gestores de base de datos o DBMS.

Para el sistema que se desarrolló se eligió utilizar el manejador de base de datos PostgreSQL. Éste es un DBMS de código abierto que además es relacional y orientado a objetos.

Objetos de la base de datos

Considerando el diagrama entidad relación, que se elaboró en la etapa de diseño, se crearon las siguientes tablas en la base de datos:

CLASE
Esta tabla almacena los datos de las clases que son registradas dentro del sistema. Son identificadas por el nombre de la clase y el paquete al cual pertenecen. Su identificador principal es la columna id_clase. Este identificador es asignado mediante una secuencia (sq_clase).

SQL pane
```
CREATE TABLE clase
(
  id_clase integer NOT NULL DEFAULT nextval('sq_clase'::regclass),
  nombre character varying(120),
  paquete character varying(120),
  CONSTRAINT clase_pkey PRIMARY KEY (id_clase )
)
```

METODO
Esta tabla almacena los datos de los métodos que son registrados dentro del sistema. Son identificados por el nombre del método. Su identificador principal es la columna id_metodo. Este identificador es asignado mediante una secuencia (sq_metodo).
Tiene referencia hacia la tabla clase(id_clase). Con esta relación se mantiene la referencia hacia la clase a la cual pertenece el método.

SQL pane
```
CREATE TABLE metodo
(
  id_metodo integer NOT NULL DEFAULT nextval('sq_metodo'::regclass),
  id_clase integer,
  nombre character varying(200),
  CONSTRAINT metodo_pkey PRIMARY KEY (id_metodo ),
  CONSTRAINT metodo_id_clase_fkey FOREIGN KEY (id_clase)
      REFERENCES clase (id_clase) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
```

METODO_BITACORA
Esta tabla almacena los metadatos de las bitácoras configuradas sobre un método. Su identificador principal es la columna id_metodo_bitacora. Este identificador es asignado mediante una secuencia (sq_metodo_bitacora).

Tiene referencia hacia la tabla metodo(id_metodo). Con esta relación se mantiene la referencia hacia el método a la cual pertenece la configuración de la bitácora.

SQL pane
```
CREATE TABLE metodo_bitacora
(
  id_metodo_bitacora integer NOT NULL DEFAULT nextval('sq_metodo_bitacora'::regclass),
  id_metodo integer,
  interceptor character varying(30),
  mensaje character varying(400),
  muestra_error boolean,
  muestra_parametro boolean,
  CONSTRAINT metodo_bitacora_pkey PRIMARY KEY (id_metodo_bitacora ),
  CONSTRAINT metodo_bitacora_id_metodo_fkey FOREIGN KEY (id_metodo)
      REFERENCES metodo (id_metodo) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
)
```

BITACORA
Esta tabla almacena las bitácoras de ejecución, que han sido generadas para un método específico. Su identificador principal es la columna id_bitacora. Este identificador es asignado mediante una secuencia (sq_bitacora).

Tiene referencia hacia la tabla metodo_bitacora(id_metodo_bitacora). Con esta relación se mantiene la referencia hacia el método al cual corresponde la bitácora.

SQL pane
```
CREATE TABLE bitacora
(
  id_bitacora integer NOT NULL DEFAULT nextval('sq_bitacora'::regclass),
  id_metodo_bitacora integer,
  fecha timestamp with time zone,
  mensaje character varying(1000),
  CONSTRAINT bitacora_pkey PRIMARY KEY (id_bitacora ),
  CONSTRAINT bitacora_id_metodo_bitacora_fkey FOREIGN KEY (id_metodo_bitacora)
      REFERENCES metodo_bitacora (id_metodo_bitacora) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
)
```

INVOCACION
Almacena los datos de las invocaciones realizadas sobre un método. Su identificador principal es la columna id_invocacion. Este identificador es asignado mediante una secuencia (sq_invocacion).

Tiene referencia hacia la tabla metodo(id_metodo). Con esta relación se mantiene la referencia hacia el método al cual pertenece la invocación.

SQL pane
```
CREATE TABLE invocacion
(
  id_invocacion integer NOT NULL DEFAULT nextval('sq_invocacion'::regclass),
  id_metodo_clase integer,
  fecha timestamp with time zone,
  duracion integer,
  CONSTRAINT invocacion_pkey PRIMARY KEY (id_invocacion ),
  CONSTRAINT invocacion_id_metodo_clase_fkey FOREIGN KEY (id_metodo_clase)
      REFERENCES metodo (id_metodo) MATCH SIMPLE
      ON UPDATE CASCADE ON DELETE CASCADE
)
```

USUARIO
Almacena la información de los usuarios registrados para poder acceder al sistema. Se almacena el login, password y el estatus del usuario. El password se almacenará utilizando un hash md5.

SQL pane
```
CREATE TABLE usuario
(
  id_usuario integer NOT NULL DEFAULT nextval('sq_usuario'::regclass),
  login character varying(50),
  password character varying(80),
  activo boolean,
  intentos integer DEFAULT 0,
  CONSTRAINT usuario_pkey PRIMARY KEY (id_usuario )
)
```

Implementación de los módulos

Los módulos del sistema fueron desarrollados utilizando las siguientes tecnologías:

- Plataforma de desarrollo – Java: Es una tecnología conformada por una plataforma y un lenguaje de programación. Tiene la particularidad de ser multiplataforma. Por lo anterior, sumado a la facilidad para desarrollar en ella, la ha convertido en la plataforma de desarrollo más popular en la actualidad.

- Framework – Spring: Es un framework de Java muy popular. Tiene el objetivo de fomentar las buenas prácticas, además de hacer más simple el desarrollo de aplicaciones empresariales.
- Herramienta de construcción del proyecto – Maven: Es una herramienta de gestión de proyectos, que permite administrar las dependencias de los módulos, ejecutar pruebas unitarias además de construir, empaquetar y desplegar dichos módulos.
- Servidor de aplicaciones – GlassFish: Servidor de aplicaciones de código abierto. Fue desarrollado inicialmente por Sun. Es la implementación de referencia de la plataforma empresarial de Java.

Los módulos del sistema fueron implementados como se detalla a continuación:

logSystemTO

Paquete: *org.logSystem.dominio*
Producto resultante: *logSystemTO.jar*

Módulo que implementa los objetos de dominio que son compartidos y utilizados por el resto de los módulos.

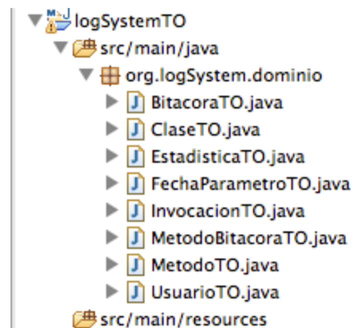

Figura 4.4.3.1 Clases implementadas en el módulo logSystemTO.

logSystemPersistencia

Paquete: *org.logsystem.persistence*
Producto resultante: *logSystemPersistencia.jar*

Este módulo implementa la capa de persistencia. Gestiona el acceso a datos dentro de la aplicación.

Mediante la configuración de un contexto de Spring (logSystemPersistence-context.xml) se logró que la configuración necesaria para acceder a la base de datos, permaneciera centralizada y separada de la implementación de los objetos de acceso a datos.

El acceso a datos se implementó utilizando un ORM (object-relational mapping) llamado MyBatis.

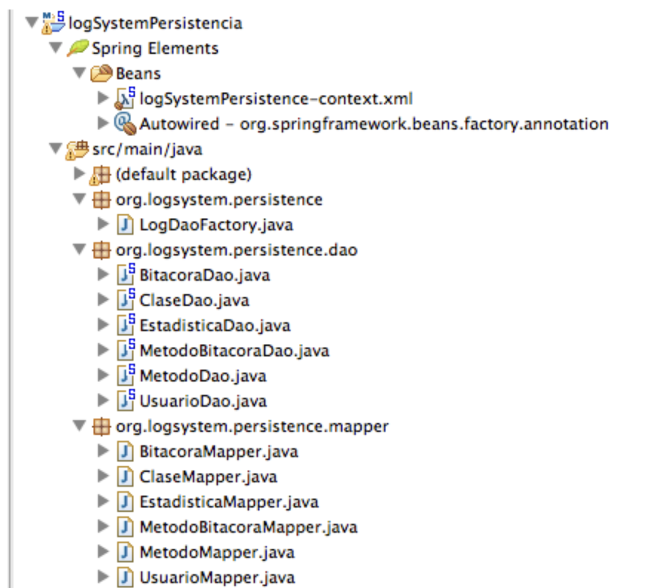

Figura 4.4.3.2
Clases implementadas en el módulo logSystemPersistencia.

logSystemAOP

Paquete: *org.logSystem.integracion*
Producto resultante: *logSystemAOP.jar*

Módulo encargado de abstraer las interacciones del sistema con el contexto de Spring. Ofrece servicios para listar los objetos existentes en el contexto.

También contiene las clases que gestionan la orientación a aspectos ya que cuenta con la lógica necesaria que implementa el Pointcut y el Advice tanto del logging como del monitoreo de ejecuciones.

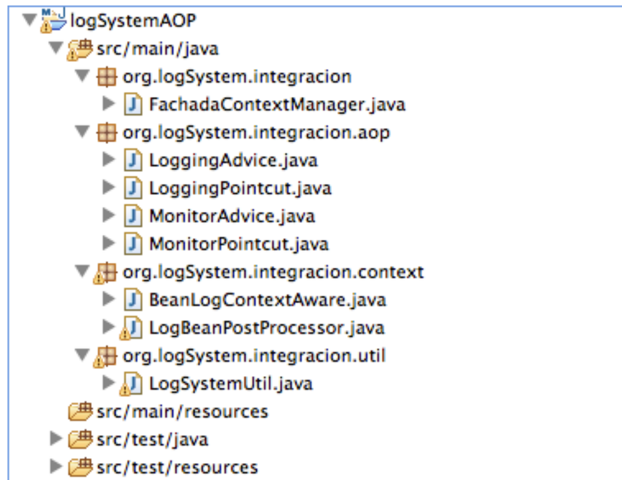

Figura 4.4.3.3 Clases implementadas en el módulo logSystemAOP.

logSystemService

Paquete: *org.logSystem.servicio*
Producto resultante: *logSystemService.jar*

Este módulo abstrae y encapsula la capa de servicios del sistema. Para cumplir con sus funciones utiliza la capa de persistencia.

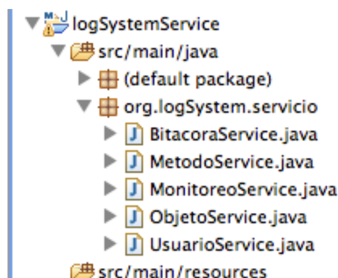

Figura 4.4.3.4 Clases implementadas en el módulo logSystemService.

logSystemREST

Paquete: *org.logSystem.web.rest*
Producto resultante: *logSystemREST.jar*

Aqui se exponen los servicios del sistema, como servicios REST. Se implementó como un fragmento web, cuyo objetivo es ser incluido en una aplicación web.

Cuando se incluye el módulo dentro de una aplicación implementada con Spring, hace que se publiquen un conjunto de servicios REST, que serán utilizados por la capa web para gestionar la generación de bitácoras dentro del sistema target.

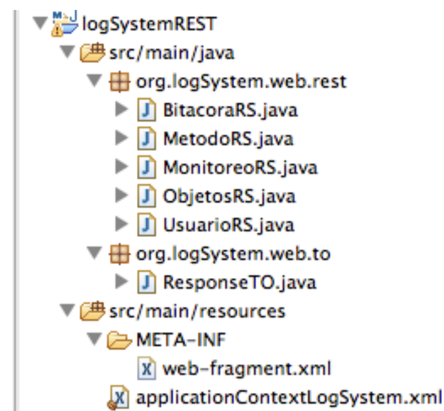

Figura 4.4.3.5 Clases implementadas en el módulo logSystemRest.

logSystemWebClient

Paquete: *org.logSystem.web.client*
Producto resultante: *logSystemWebClient.war*

Es una aplicación web cuya finalidad es administrar la generación de bitácoras en el sistema target.

Es un cliente que simplemente consume los servicios REST expuestos por el módulo *logSystemREST* a través de la aplicación target.

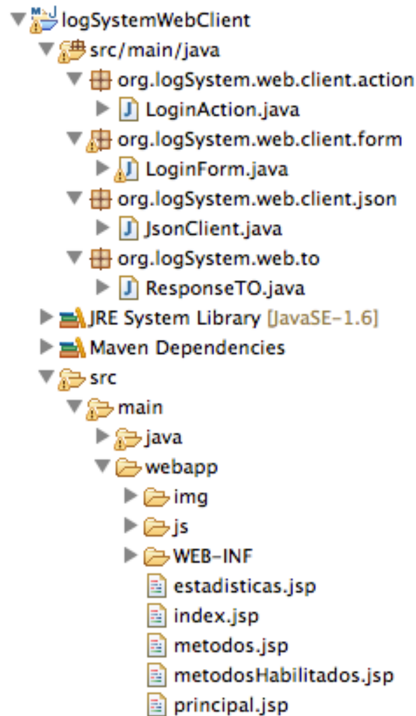

Figura 4.4.3.6 Clases y componentes web implementados en la aplicación cliente del sistema.

El módulo implementa los casos de uso definidos durante la etapa de análisis, a través de varias pantallas:

Función: Autenticación
Permite autenticar a un usuario, determinando si éste debe o no tener acceso al sistema. Se muestra en la figura 4.4.3.7.

Iniciar sesión
Login:
Password:
Entrar

Figura 4.4.3.7 Pantalla de autenticación del sistema.

Función: Visualización de métodos

Permite visualizar todos los métodos a los cuales se les pueden aplicar la generación de bitácoras dentro de un sistema. Se muestra en la figura 4.4.3.8.

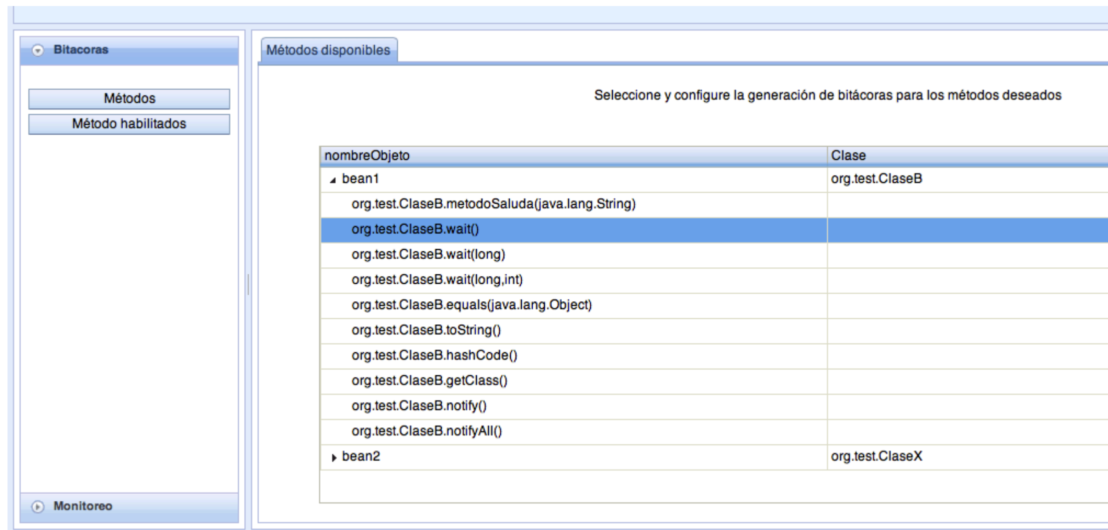

Figura 4.4.3.8 Pantalla que muestra los objetos y métodos existentes en el contenedor de Spring.

Función: Habilitar la generación de bitácoras

Permite activar la generación de bitácoras (logs) para un método específico. Figura 4.4.3.9.

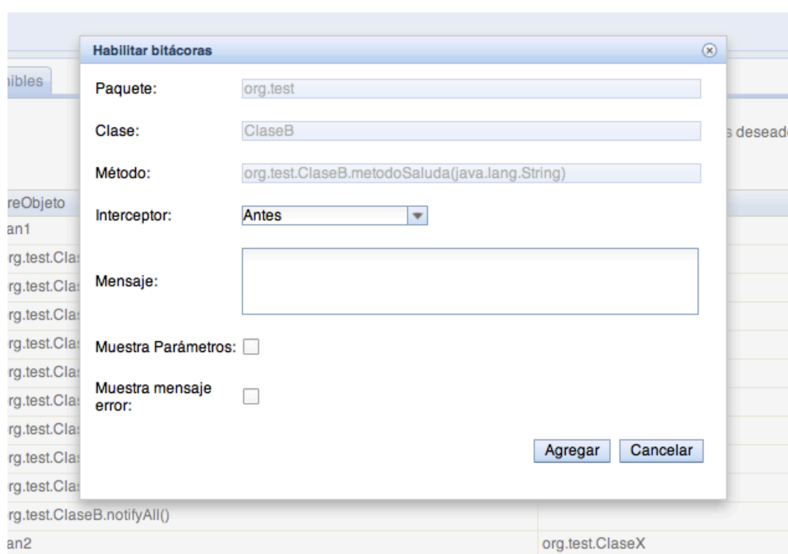

Figura 4.4.3.9 Pantalla que permite configurar y habilitar la generación de bitácoras.

Función: Visualización de métodos habilitados.

Permite visualizar los métodos para los cuales se encuentra habilitada la generación de bitácoras. Se muestra en la figura 4.4.3.10.

Bitacoras

Métodos

Método habilitados

Monitoreo

Métodos habilitados

Visualización de bitácoras generadas para los métodos habilitados

Clase	Nombre método	Interceptor
org.test.ClaseB	org.test.ClaseB.notify()	ERROR
org.test.ClaseB	org.test.ClaseB.metodoSaluda(java.lang.String)	DESPUES
org.test.ClaseB	org.test.ClaseB.getClass()	ANTES
org.test.ClaseB	org.test.ClaseB.equals(java.lang.Object)	DESPUES
org.test.ClaseX	org.test.ClaseX.obtenNombre(java.lang.String,java.lang	DESPUES

Figura 4.4.3.10 Pantalla que muestra los métodos que han sido habilitados para la generación de bitácoras.

Función: Visualización de logs

Permite visualizar los logs generados para uno o varios métodos en un cierto rango de tiempo. Se muestra en la figura 4.4.3.11.

Bitácoras por método

Bitácora de invocaciones correspondiente al método:
org.test.ClaseX.obtenNombre(java.lang.String,java.lang.String,int)

Fecha	Mensaje
28/04/2013 20:18:33:615	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 20:18:32:224	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 20:18:31:009	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 20:18:29:842	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 20:18:28:353	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 20:18:27:437	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 19:41:11:310	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 19:41:10:071	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 19:41:08:933	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 19:41:08:203	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 18:55:36:130	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 18:55:35:395	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 18:55:33:777	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 18:55:31:365	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 18:25:18:077	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)
28/04/2013 18:25:16:761	Termina ejecución: Se ha ejecutado el metodo ::Parametros(hola ll ramon ll 1 ll)

Figura 4.4.3.11 Pantalla que muestra las bitácoras generadas para un método.

Función: Habilitación del monitoreo de ejecuciones
Permite habilitar o deshabilitar el monitoreo de las invocaciones de métodos, así como la duración de cada invocación (Figura 4.4.3.12).

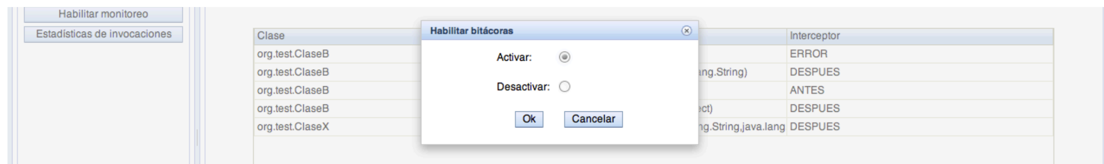

Figura 4.4.3.12 Pantalla que permite habilitar el monitoreo de invocaciones.

Función: Visualización de estadísticas de ejecuciones
Muestra los métodos, el número de invocaciones que han tenido, además de la duración total de las invocaciones y su duración promedio (Figura 4.4.3.13).

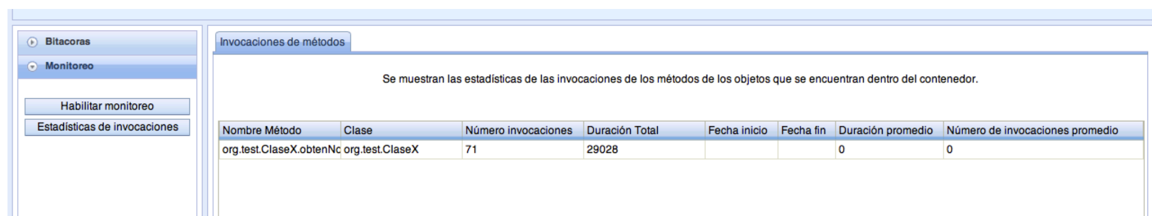

Figura 4.4.3.13 Pantalla que muestra las estadísticas de ejecuciones de métodos.

Función: Visualización de gráficas
Se mostrarán gráficas presentando los datos de las ejecuciones totales por minuto y la duración promedio de las ejecuciones en un minuto (Figura 4.4.3.14).

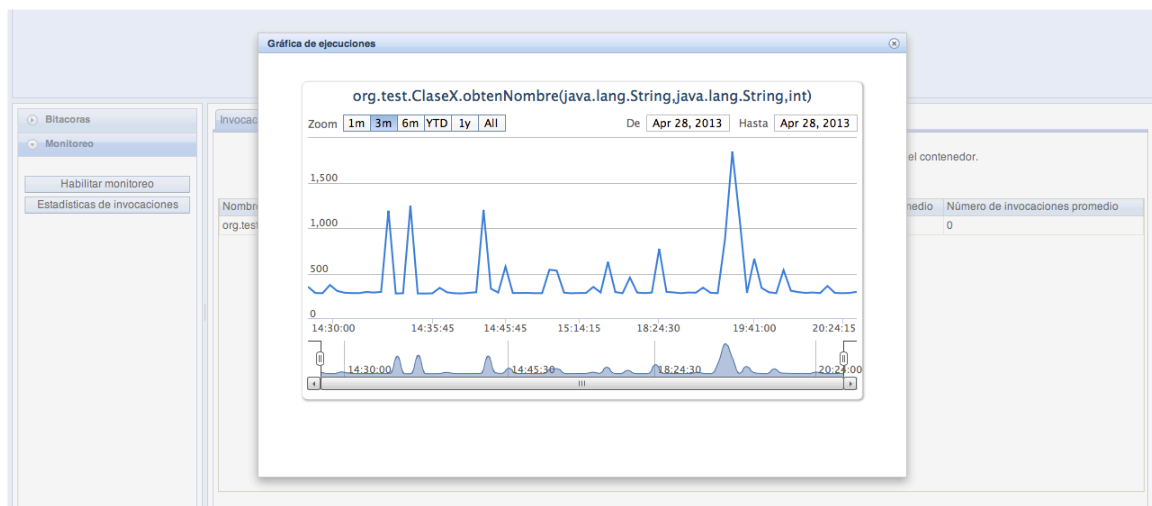

Figura 4.4.3.15 Pantalla que muestra los datos gráficamente de las invocaciones de un método.

Despliegue
Como resultado del desarrollo se obtienen los siguientes módulos:
- logSystemTO
- logSystemPersistencia
- logSystemAOP
- logSystemService
- logSystemWebClient

Estos módulos se desplegarán en el servidor de aplicaciones en la estructura que se detalla en la figura 4.4.3.16.

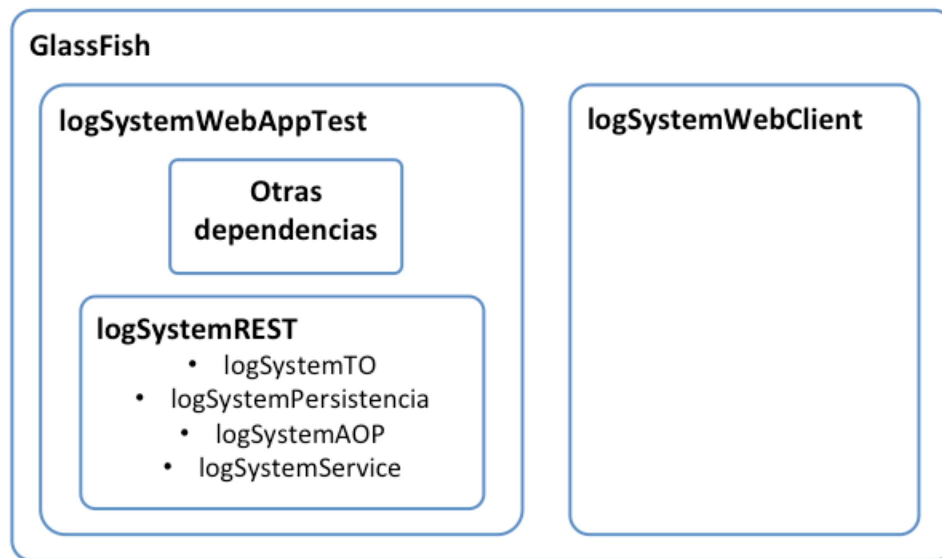

Figura 4.4.3.16 Diagrama de despliegue de la aplicación.

logSystemWebAppTest es una aplicación web de prueba. Dicha aplicación está implementada con Spring. Como cualquier otra aplicación cuenta con un grupo de dependencias requeridas para cumplir con sus funciones.

Con el objetivo de poder generar las bitácoras del sistema, a la aplicación "*logSystemWebAppTest*" se le agrega la dependencia del web fragment desarrollado: *logSystemREST*. Con esto la funcionalidad de generación de bitácoras estará disponible para la aplicación target.

La configuración de la generación de bitácoras, queda expuesta a través de un conjunto de servicios REST, que se publica a través de la aplicación target (*logSystemWebAppTest*). Estos servicios pueden ser consumidos por el cliente web *logSystemWebClient*. Lo anterior permitirá configurar y visualizar las bitácoras generadas a través de la interfaz gráfica provista por la aplicación web *logSystemWebClient.*

En este capítulo se expuso, el proceso de implementación del sistema. La implementación tiene como finalidad obtener ya sea mediante programación, configuración, instalación o integración de componentes, una versión del sistema que pueda ser ejecutada. Dicha versión debe ser capaz de cumplir con la totalidad de los requerimientos establecidos.

La implementación del sistema de bitácoras se realizó utilizando la plataforma Java, como una base importante. Además, el desarrollo se apoyó en Spring. Este framework facilitó la implementación de servicios empresariales dentro de la aplicación.

Al final, el sistema resultante fue capaz de ser desplegado en un servidor de aplicaciones para su puesta en ejecución.

Es importante que el sistema desarrollado pueda demostrar que cumple con los requerimientos. Y es deber del proceso de pruebas, el verificar que la versión ejecutable del sistema cumpla en su totalidad con los objetivos planteados con anterioridad.

En los próximos temas se lleva a cabo esta verificación.

4.4.4

Pruebas

Un sistema en desarrollo debe demostrar que es capaz de realizar las tareas para las cuales fue construido. La fase de pruebas tiene como objetivo demostrar que el sistema cumple con las funciones requeridas, además de ayudar a detectar posibles defectos en la programación. Lo anterior debe ocurrir antes de que el sistema sea puesto en un ambiente productivo.

Pruebas unitarias

Las pruebas unitarias permitieron verificar el correcto funcionamiento de los componentes del sistema. En esta etapa se probaron uno a uno los métodos de las clases definidas, ya que los métodos son el tipo más simple de componente.

Diseño de casos de prueba

Se diseñaron, después de la implementación, casos de prueba para cada uno de los componentes del sistema. Se listan a continuación:

ClaseDao

Para este componente que pertenece a la capa de persistencia, se diseñaron pruebas unitarias que manipulan registros de prueba dentro de la base de datos. Por esa razón es necesario verificar el estado de la base de datos después de ejecutarlas .

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
insertaClaseTest	Un objeto ClaseTO	N/A	N/A	Nuevo registro en la entidad Clase	Permite verificar el correcto funcionamiento de las inserciones de objetos de tipo Clase
insertaClaseNullTest	Referencia nula	N/A	N/A	Excepción	Verifica que no se permitan insertar objetos nulos dentro de la base
insertaClaseDobleTest	Un objeto ClaseTO existente en la base	N/A	Debe existir el objeto de entrada	Excepción	Verifica que no se permiten insertar objetos duplicados en la base
eliminaClaseTest	Un objeto ClaseTO	N/A	Debe existir el objeto de entrada	Objeto eliminado	Permite verificar que se eliminan correctamente los registros de la base asociados a la entidad Clase

obtenClasePorNombre	Nombre de clase	Un objeto de tipo ClaseTO	N/A	N/A	Ayuda a verificar que se puede obtener correcta-mente el objeto asociado a un nombre de clase
actualizaClaseTest	Un objeto ClaseTO	N/A	N/A	Objeto modifi-cado en la base, en caso de existir	Ayuda a verificar que los objetos se actualizan co-rrectamente con la infor-mación contenida dentro del parámetro
actualizaClaseIncorrectaTest	Un objeto ClaseTO	N/A	N/A	Excepción	Verifica que no se permita actualizar objetos, con información incorrecta

MetodoDao
Para este componente que pertenece a la capa de persistencia, se diseñaron pruebas unitarias que manipulan registros de prueba dentro de la base de datos. Por esa razón es necesario verificar el estado de la base de datos después de ejecutarlas .

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
insertaMetodoTest	Un objeto MetodoTO	N/A	N/A	Nuevo registro en la entidad Metodo	Verifica el correcto fun-cionamiento de las in-serciones de objetos de tipo Metodo
actualizaMetodoTest	Un objeto MetodoTO	N/A	N/A	Objeto modificado en la base, en caso de existir	Verifica que los objetos se actualizan correc-tamente con la infor-mación contenida den-tro del parámetro
actualizaMetodoIncorrectoTest	Un objeto MetodoTO	N/A	N/A	Excepción	Verifica que no se permita actualizar ob-jetos, con información incorrecta
eliminaMetodoTest	Un objeto MetodoTO	N/A	N/A	Objeto eliminado	Verifica que se eliminan correctamen-te los registros de la base asociados a la entidad Clase
obtenMetodosPorClaseTest	Un objeto ClaseTO	Un objeto MetodoTO	N/A	N/A	Permite verificar que se obtiene correcta-mente la información de un método con base en la clase a la que pertenece
obtenMetodosPorIDTest	Identificador de método	Un objeto MetodoTO	N/A	N/A	Caso para verificar que se obtiene correcta-mente la información de un método con base a su identificador

MetodoBitacoraDao

Para este componente, que pertenece a la capa de persistencia, se diseñaron pruebas unitarias que manipulan registros de prueba dentro de la base de datos. Por esa razón es necesario verificar el estado de la base de datos después de ejecutarlas .

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
insertaMetBitTest	Un objeto MetodoBitacoraTO	N/A	N/A	Nuevo registro en la entidad MetodoBitacora	Caso para verificar el correcto funcionamiento de las inserciones de objetos de tipo MetodoBitacora
actualizaMetBitTest	Un objeto MetodoBitacoraTO	N/A	N/A	Objeto modificado en la base, en caso de existir	Verifica que los objetos se actualizan correctamente con los datos dentro del parámetro
actualizaMetBitIncorrectoTest	Un objeto MetodoBitacoraTO	N/A	N/A	Excepción	Verifica que no se permita actualizar objetos con información incorrecta
eliminaMetBitTest	Un objeto MetodoBitacoraTO	N/A	N/A	Objeto eliminado	Permite verificar que se eliminan correctamente los registros de la base asociados a la entidad Clase
obtenMetBitPorMetodoTest	Un objeto MetodoTO	Un objeto MetodoBitacoraTO	N/A	N/A	Permite verificar que se obtiene correctamente la información de un métodobitácora con base al método

EstadisticaDao

Para este componente que pertenece a la capa de persistencia, se diseñaron pruebas unitarias que manipulan registros de prueba dentro de la base de datos. Por esa razón es necesario verificar el estado de la base de datos después de ejecutarlas .

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
insertaEstTest	Un objeto InvocacionTO	N/A	N/A	Nuevo registro en la entidad - invocacion	Verifica el correcto funcionamiento de las inserciones de objetos de tipo Invocacion
obtenEstGralTest	Rango de fechas	Lista EstadisticaTO	N/A	N/A	Verifica que se obtienen correctamente las estadísticas generales de invocaciones
obtenEstMetTest	Un objeto MetodoBitacoraTO	Lista EstadisticaTO	N/A	N/A	Verifica que se obtienen correctamente las estadísticas de invocaciones de un método

BitacoraDao

Para este componente que pertenece a la capa de persistencia, se diseñaron pruebas unitarias que manipulan registros de prueba dentro de la base de datos. Por esa razón es necesario verificar el estado de la base de datos después de ejecutarlas .

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
insertaBitTest	Un objeto BitacoraTO	N/A	N/A	Nuevo registro en la entidad Bitacora	Verifica el correcto funcionamiento de las inserciones de objetos de tipo Bitacora
insertaBitInvalidaTest	Un objeto BitacoraTO	N/A	N/A	Excepción	Verifica que no se permita insertar objetos, con información incorrecta
obtenBitPorMetodoTest	Un objeto MetodoTO	Lista BitacoraTO	N/A	N/A	Verifica que se obtienen correctamente las bitácoras de invocaciones de un método

UsuarioDao
Para este componente que pertenece a la capa de persistencia, se diseñaron pruebas unitarias que manipulan registros de prueba dentro de la base de datos. Por esa razón es necesario verificar el estado de la base de datos después de ejecutarlas .

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
obtenUsuarioTest	Nombre de usuario	Un objeto UsuarioTO	N/A	N/A	Verifica la obtención de datos de un usuario
insertaUsuarioTest	Un objeto UsuarioTO	N/A	N/A	Nuevo registro en la entidad Usuario	Verifica el correcto funcio-namiento de las inserciones de objetos de tipo Usuario
actualizaUsuarioTest	Un objeto UsuarioTO	N/A	N/A	Objeto modificado	Verifica que los objetos se actualizan correctamente con la información contenida dentro del parámetro

UsuarioService
Para las pruebas unitarias de los componentes de la capa de servicios, se utilizó un objeto de prueba (mock) para emular el comportamiento de un objeto DAO. Este mock únicamente devuelve objetos de prueba.

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
usuarioValidoTest	Nombre de usuario, password	boolean	N/A	N/A	Verifica la correcta validación de un usuario
usuarioInvalidoTest	Nombre de usuario, password	boolean	N/A	N/A	Verifica la correcta validación de un usuario con datos incorrectos

ObjetoService
Para las pruebas unitarias de los componentes de la capa de servicios, se utilizó un objeto de prueba (mock) para emular el comportamiento de un objeto DAO. Este mock únicamente devuelve objetos de prueba.

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
obtenerObjsTest	N/A	Map	N/A	N/A	Verifica la correcta obtención de los datos de los objetos del AppCtx

MetodoService
Para las pruebas unitarias de los componentes de la capa de servicios, se utilizó un objeto de prueba (mock) para emular el comportamiento de un objeto DAO. Este mock únicamente devuelve objetos de prueba.

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
habilitaBitTest	Información Metodo	Un objeto UsuarioTO	N/A	N/A	Verifica la inserción de métodos en la base
obtenMetodosTest	N/A	Lista Metodo Bitacora TO	N/A	N/A	Verifica la obtención de datos de los métodos

BitacoraService
Para las pruebas unitarias de los componentes de la capa de servicios, se utilizó un objeto de prueba (mock) para emular el comportamiento de un objeto DAO. Este mock únicamente devuelve objetos de prueba.

Caso de prueba	Entrada	Salida	Precondiciones	Postcondiciones	Descripción
obtenerBitTest	Nombre método	Lista Bitacora TO	N/A	N/A	Verifica que se obtienen correctamente las bitácoras de invocaciones de un método

Implementación

Las pruebas unitarias del proyecto se implementaron utilizando un esquema de pruebas automatizadas, para lo cual se utilizó Junit 3.8.2.

Como se mencionó anteriormente Maven se encarga de gestionar la construcción completa de un proyecto, incluyendo la ejecución de las pruebas unitarias. Maven para este propósito, cuenta con un ciclo de vida como se detalla en la figura 4.4.4.1.

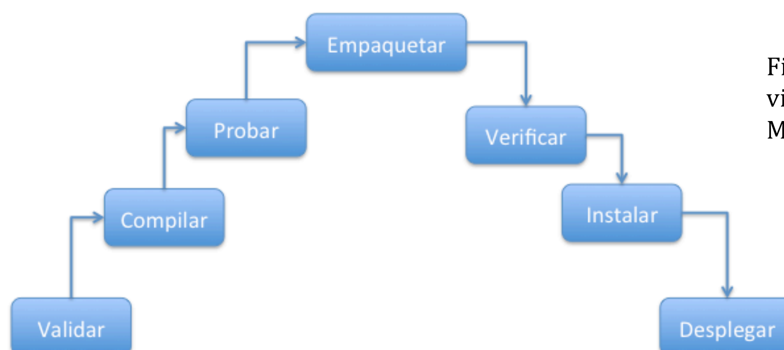

Figura 4.4.4.1 Ciclo de vida de un proyecto Maven.

Este ciclo de vida considera que las pruebas unitarias, contenidas en el proyecto, son ejecutadas después de realizar la compilación y antes del empaquetamiento.

Por lo anterior, para el proyecto actual se incluyeron las pruebas unitarias implementadas con Junit en cada uno de los módulos. Este esquema de pruebas automatizadas tiene las siguientes ventajas:

- Las pruebas se ejecutan cada vez que se construye el proyecto.
- Se ejecutan pruebas de regresión, cada vez que se realiza una modificación en el código y se compila.
- Sólo es posible instalar un artefacto cuando éste ha pasado exitosamente sus pruebas unitarias.

Pruebas de sistema

Las pruebas de sistema durante el desarrollo, involucran la integración descendente de los componentes para crear una versión ejecutable del sistema y de esta forma probarlo de manera integra. Este tipo de integración implica que los módulos se prueben desde el programa principal hacia los módulos inferiores, de forma jerárquica. En esta etapa se verifica que los componentes sean compatibles, interactúen de manera adecuada e intercambien datos correctamente a través de sus interfaces.

Con el objetivo de probar el sistema, se desarrolló una aplicación target de prueba. En ella se instaló el sistema generador de bitácoras. La configuración de las aplicaciones quedó como se detalla en la figura 4.4.4.2.

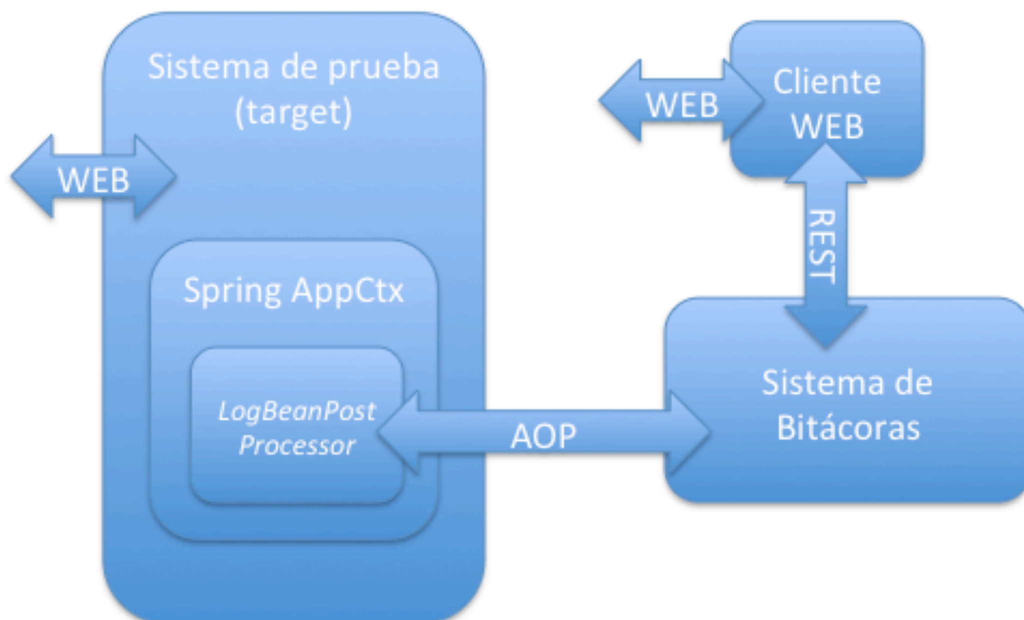

Figura 4.4.4.2 Configuración del ambiente de pruebas del sistema.

El sistema target debe ser una aplicación web basada en Spring. Dentro de la configuración de esta aplicación (applicationContext) se importa el archivo de configuración del sistema de bitácoras. Este archivo incluye un objeto (*LogBeanPostProcessor)* cuya función principal es generar mediante programación orientada a aspectos los objetos proxy de los componentes existentes en la aplicación target.

Al incluir el sistema de bitácoras dentro de la aplicación target, se exponen dentro de ésta una serie de servicios REST, que serán utilizados por el cliente web para configurar y visualizar las bitácoras del sistema.

A continuación se enlistan las pruebas que se diseñaron para el sistema de bitácoras. Se agrupan por caso de uso:

Autenticación

Prueba	Ingresar usuario y password válidos
Comportamiento esperado	El sistema debe otorgar el acceso al sistema
Precondiciones	Al menos un usuario activo
Descripción	Se validó que existiendo un usuario activo en la base de datos, el sistema, desde la pantalla de login, permitirá el acceso al sistema siempre y cuando se proporcionen las credenciales adecuadas (login y password)

Prueba	Ingresar usuario y password inválidos
Comportamiento esperado	El sistema debe indicar que existió un error y denegar el acceso al sistema
Precondiciones	Debe existir al menos un usuario activo
Postcondiciones	Al fallar en un intento de ingresar, se registrará como un intento fallido en la base de datos
Descripción	Se validó que el sistema desde la pantalla de login, deniega el acceso cuando los datos ingresados son inválidos. Se probó que el sistema muestra un mensaje de error descriptivo. También se debe de registrar un acceso inválido en la base de datos. En caso de existir 5 accesos inválidos, el usuario se bloquea por razones de seguridad. Se considera que este número es adecuado para equilibrar los posibles errores en la autenticación y evitando un posible ataque de fuerza bruta

Prueba	Ingresar enviando los campos vacíos
Comportamiento esperado	El sistema debe indicar que login/password son campos requeridos y denegar el acceso
Precondiciones	Al menos un usuario activo
Descripción	Se validó que el sistema desde la pantalla de login, deniega el acceso cuando los datos usuario/password están vacíos. El sistema debe indicar mediante un mensaje descriptivo, que estos campos son requeridos

Visualización de métodos

Prueba	Ingresar a la pantalla de visualización de métodos
Comportamiento esperado	Al ingresar a la pantalla de visualización de métodos, se deben mostrar los “beans” existentes dentro del ApplicationContext de la aplicación target
Precondiciones	Debe existir al menos un “bean” en el ApplicationContext de la aplicación target
Descripción	Se validará que al ingresar a la pantalla de visualización de métodos se muestren todos los “beans” existentes. Esta lista de beans debe coincidir con los que están dentro del contenedor de Spring de la aplicación target. Además, se debe de mostrar un listado de cada uno de los métodos disponibles de cada bean

Prueba	Ingresar a la pantalla de visualización de métodos cuando está abajo el contenedor de Spring
Comportamiento esperado	Al ingresar a la pantalla de visualización de métodos, se debe mostrar un mensaje descriptivo indicando que el servicio no está disponible
Precondiciones	Debe estar disponible la aplicación target, pero el contenedor de Spring debe estar abajo
Descripción	Se validará que al ingresar a la pantalla de visualización de métodos se muestre un mensaje adecuado, para el caso en el que el contenedor de Spring no se encuentre disponible, por cualquier situación

Habilitación de la generación de bitácoras

Prueba	Habilitar con información correcta la generación de bitácoras
Comportamiento esperado	Al habilitar la generación de bitácoras se comenzarán a generar las mismas cada vez que se invoque el método asociado
Precondiciones	Haber seleccionado previamente un método de la pantalla "Visualización de métodos"
Postcondiciones	Se generó un registro con la configuración (MetodoBitacora), y se inició la generación de bitácoras del método asociado
Descripción	Se validó que al seleccionar un método de la pantalla "Visualización de métodos" y configurar los parámetros de generación de bitácoras, se genera correctamente la configuración en la base de datos. Además de corroborar que se están generando las bitácoras para el método asociado, cada vez que se invoca este último

Prueba	Habilitar con información incorrecta la generación de bitácoras
Comportamiento esperado	El sistema debe informar que se ha ingresado información incorrecta
Precondiciones	Haber seleccionado previamente un método de la pantalla "Visualización de métodos"
Descripción	Se validó que al seleccionar un método de la pantalla "Visualización de métodos" y configurar los parámetros de generación de bitácoras con información incompleta o incorrecta, el sistema informe con un mensaje adecuado la existencia de uno o varios campos con información incompleta o incorrecta

Visualización de los métodos habilitados

Prueba	Ingresar a la pantalla de visualización de métodos habilitados
Comportamiento esperado	Al ingresar a la pantalla de visualización de métodos habilitados, se deben mostrar los métodos que han sido habilitados para la generación de bitácoras
Precondiciones	Uno o más métodos habilitados para la generación de bitácoras
Descripción	Se validó que al ingresar a la pantalla de visualización de métodos habilitados, se muestren todos los métodos que tengan esta condición en la base de datos

Prueba	Ingresar a la pantalla de visualización de métodos habilitados cuando no existan
Comportamiento esperado	Al ingresar a la pantalla de visualización de métodos habilitados y que en ese momento no haya ningún método habilitado se debe mostrar un mensaje indicando esta situación
Precondiciones	No deben existir métodos habilitados para la generación de bitácoras
Descripción	Se validó que al ingresar a la pantalla de visualización de métodos habilitados y que en ese momento no exista ninguno habilitado, se muestre un mensaje indicando esta situación

Visualización de logs

Prueba	Ingresar a la pantalla de visualización de logs
Comportamiento esperado	Al ingresar a la pantalla de visualización de logs se deben mostrar los logs generados hasta ese momento
Precondiciones	Haber elegido un método habilitado en la pantalla de "Visualización de los métodos habilitados"
Descripción	Se validó que al elegir un método habilitado para la generación de bitácoras, se muestren todos y cada uno de los registros generados en la bitácora del método asociado. Estos registros deben corresponder a los almacenados en la base de datos

Prueba	Ingresar a la pantalla de visualización de logs, cuando no existen logs
Comportamiento esperado	Al ingresar a la pantalla de visualización de logs se debe mostrar un mensaje indicando la inexistencia de bitácoras para el método
Precondiciones	Se debe haber elegido un método habilitado en la pantalla de "Visualización de los métodos habilitados". Para este método no deben existir registros en la bitácora
Descripción	Se validó que al elegir un método habilitado para la generación de bitácoras y que para éste no existan registros en la misma, se muestre un mensaje indicando esta situación

Habilitación del monitoreo de ejecuciones

Prueba	Habilitar el monitoreo de ejecuciones
Comportamiento esperado	Al habilitar el monitoreo, se notificará el éxito de la operación
Precondiciones	Debe estar deshabilitado el monitoreo de ejecuciones
Postcondiciones	Monitoreo de ejecuciones habilitado
Descripción	Se validó que al seleccionar la habilitación del monitoreo de ejecuciones, éstas empiecen a ser registradas en la base de datos

Prueba	Deshabilitar el monitoreo de ejecuciones
Comportamiento esperado	Al deshabilitar el monitoreo, se notificará el éxito de la operación
Precondiciones	Debe estar habilitado el monitoreo de ejecuciones
Postcondiciones	Monitoreo de ejecuciones deshabilitado
Descripción	Se validó que al seleccionar la deshabilitación del monitoreo de ejecuciones, se detenga el registro en la base de las ejecuciones

Visualización de estadísticas de ejecuciones

Prueba	Ingresar a la pantalla de visualización de estadísticas de ejecuciones
Comportamiento esperado	Al ingresar a la pantalla de visualización de estadísticas de ejecuciones se mostrará un listado con dicha información
Precondiciones	Pueden o no estar registradas invocaciones en la base de datos
Descripción	Se validó que al ingresar a la pantalla de estadísticas se muestre fielmente los datos de invocaciones registrados en la base de datos. Se muestran los datos por cada uno de los métodos en el sistema

Visualización de gráficas

Prueba	Visualizar una gráfica de ejecuciones
Comportamiento esperado	Al seleccionar un método se muestra una gráfica de las invocaciones de un método específico
Precondiciones	Haber seleccionado previamente un método en la pantalla de “Visualización de estadísticas de ejecuciones”
Descripción	Se validó que al seleccionar un método en la pantalla “Visualización de estadísticas de ejecuciones” se muestra una gráfica de todas las invocaciones de dicho método

Las pruebas detalladas en esta sección permitieron realizar un proceso de verificación, de los requerimientos del sistema. El éxito del desarrollo depende de que el producto obtenido cumpla con las expectativas que se plantearon inicialmente.

A continuación se hace un análisis de los resultados obtenidos después de la realización de este proyecto.

5

Resultados

El producto final del desarrollo del sistema de bitácoras es un software que puede ser utilizado por cualquier aplicación web que desee centralizar la generación, configuración y gestión de bitácoras del sistema.

Inicialmente se tomó en cuenta un conjunto de requerimientos que ayudaron a dar forma al sistema final.

A continuación se valida el cumplimiento de los requerimientos.

"El sistema debe acoplarse a una aplicación para generar logs del sistema, según se necesite"

Una aplicación web basada en Spring, puede hacer uso del sistema de bitácoras. Su integración con cualquier proyecto es muy sencilla.

El proyecto target debe de incluir la dependencia del sistema de bitácoras. Al construir el proyecto con Maven esto se hace más sencillo. La dependencia de Maven sería de la siguiente forma:

```
<dependency>
	<groupId>org.logSystem.web.rest</groupId>
	<artifactId>logSystemREST</artifactId>
	<version>1.0.0</version>
</dependency>
```

También se debe incluir la configuración de Spring del sistema de bitácoras. Esto quiere decir que en el archivo *applicationContext.xml* del sistema target se importa la configuración del sistema de bitácoras. Esto último se hace de la siguiente manera:

```
<import resource="applicationContextLogSystem.xml"/>
```

Estas configuraciones hacen que el sistema de bitácoras esté integrado al sistema target. La configuración, consulta y gestión de las bitácoras queda expuesta en la aplicación target a través de un conjunto de servicios REST.

Por ejemplo, si la aplicación target se encuentra desplegada en la siguiente url:

http://localhost:8080/logSystemWebAppTest/

Sobre ese mismo contexto (logSystemWebAppTest) se publican los siguientes servicios REST:

Bitácoras
/bitacora/metodo/{nombreMetodo}
Obtiene todas las bitácoras del método indicado por el parámetro

Configuración de bitácoras
/metodo/habilitaBitacora/{paquete}/{nombreClase}/{nombreMetodo}/{interceptor}/{mensaje}/{parametros}/{error}
Configura la generación de bitácoras para el método indicado en los parámetros.

/metodo/metodos/
Obtiene la lista de métodos que ya han sido configurados para la generación de bitácoras.

Monitoreo de ejecuciones
/monitoreo/activacion/{activar}
Activa o desactiva el monitoreo de ejecución de métodos.

/monitoreo/estadistica/general/{fechaInicio}/{fechaFin}
Obtiene las estadísticas generales de ejecuciones para el rango de fechas especificados. Es para todos los métodos.

/monitoreo/estadistica/metodo/{nombreMetodo}
Obtiene todas las estadísticas de ejecución para el método indicado en el parámetro.

Gestión de objetos en el contexto de Spring
/objeto/listaObjetos
Obtiene los metadatos de todos los objetos que existen dentro del contexto de spring, de la aplicación target.

Control de usuarios
/usuario/usuarioValido/{login}/{password}
Realiza la validación de un usuario

/usuario/altaUsuario/{login}/{password}
Registra un usuario

/usuario/eliminaUsuario/{login}
Elimina un usuario

"El sistema debe contar con un control de acceso de usuarios"

Cuando se intenta acceder al sistema de generación de bitácoras (http://localhost:8080/logSystemWebClient/), éste verifica si el usuario se ha autenticado; si no es así muestra la pantalla de inicio (Figura 5.1), solicitando usuario y contraseña.

Figura 5.1 Pantalla de inicio del sistema.

Esta pantalla realiza la autenticación del usuario utilizando los servicios REST expuestos en la aplicación target.

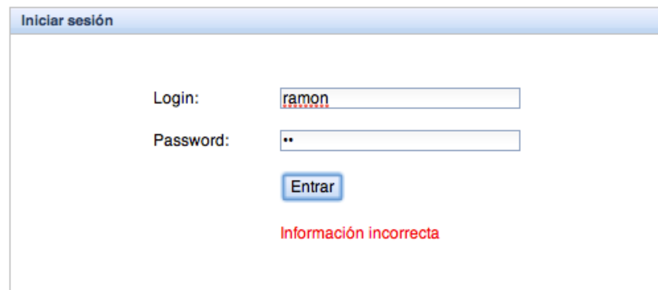

Figura 5.2 Autenticación de un usuario.

Si la información es incorrecta, la pantalla lo indicará así (Figura 5.2). Si el login existe y se falla 5 veces al autenticarse, por seguridad se bloqueará el usuario.

"El sistema debe permitir visualizar los métodos dentro del sistema objetivo, a los cuales se les pueden generar logs"

En el menú principal del sistema de bitácoras, bajo la categoría de "Bitácoras", está la opción indicada por "Métodos". Esta pantalla (Figura 5.3) permite visualizar todos los objetos que existen dentro del contenedor de Spring de la aplicación target.

Métodos disponibles

Seleccione y configure la generación de bitácoras para los métodos deseados

nombreObjeto	Clase
bean1	org.test.ClaseB
org.test.ClaseB.metodoSaluda(java.lang.String)	
org.test.ClaseB.wait()	
org.test.ClaseB.wait(long)	
org.test.ClaseB.wait(long,int)	
org.test.ClaseB.equals(java.lang.Object)	
org.test.ClaseB.toString()	
org.test.ClaseB.hashCode()	
org.test.ClaseB.getClass()	
org.test.ClaseB.notify()	
org.test.ClaseB.notifyAll()	
bean2	org.test.ClaseX
org.test.ClaseX.obtenNombre(java.lang.String,java.lang.String,int)	
org.test.ClaseX.getHTML(java.lang.String)	
org.test.ClaseX.obtenEdad(java.lang.String)	

Figura 5.3 Pantalla de visualización de métodos del sistema.

Al seleccionar cualquiera de los métodos mostrados, se puede configurar la generación de bitácoras para el método indicado (Figura 5.4).

Habilitar bitácoras

Paquete: org.test

Clase: ClaseX

Método: org.test.ClaseX.obtenNombre(java.lang.String,java.lang.String,int)

Interceptor: Despues

Mensaje: Se ha ejecutado el método obtenNombre

Muestra Parámetros: ☑

Muestra mensaje error: ☐

Agregar Cancelar

Figura 5.4 Pantalla de configuración de bitácoras.

En la pantalla mostrada se debe especificar, si la bitácora se genera antes, después de una ejecución exitosa o después de un error. También se configura el mensaje que se muestra en la bitácora, además de seleccionar si se desea mostrar los parámetros del método y el mensaje del posible error generado.

Al agregar la configuración de la bitácora, ésta se puede validar en la pantalla de visualización de métodos configurados, la cual se detalla más adelante.

"El sistema debe contar con la opción de generar logs para cierto método"

La configuración de las bitácoras para un método específico, se realiza como se menciona en el punto anterior. Se selecciona alguno de los métodos especificados, se configura en la pantalla mostrada y se agrega a la configuración de bitácoras (Figura 5.5).

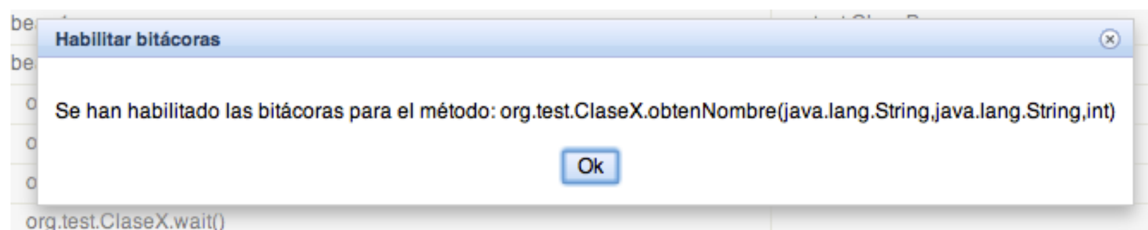

Figura 5.5 Habilitación de generación de bitácoras.

"El sistema permitirá visualizar los logs que se han generado"

Bajo la opción "Métodos habilitados" en la sección de bitácoras, está la pantalla que muestra la lista de métodos que han sido configurados para generación de bitácoras. Para nuestro caso, se muestra el método que fue recientemente configurado.

Visualización de bitácoras generadas para los métodos habilitados

Clase	Nombre método	Interceptor
org.test.ClaseX	org.test.ClaseX.obtenNombre(java.lang.String,java.lang	DESPUES

Figura 5.6 Visualización de bitácoras generadas para un método.

Al seleccionar cualquiera de los métodos que han sido configurados. Se mostrarán las bitácoras que han sido registradas en el sistema. Se muestra la fecha del registro y el mensaje generado por el sistema.

En el caso mostrado en la Figura 5.7, se generaron tres registros en la bitácora del método especificado.

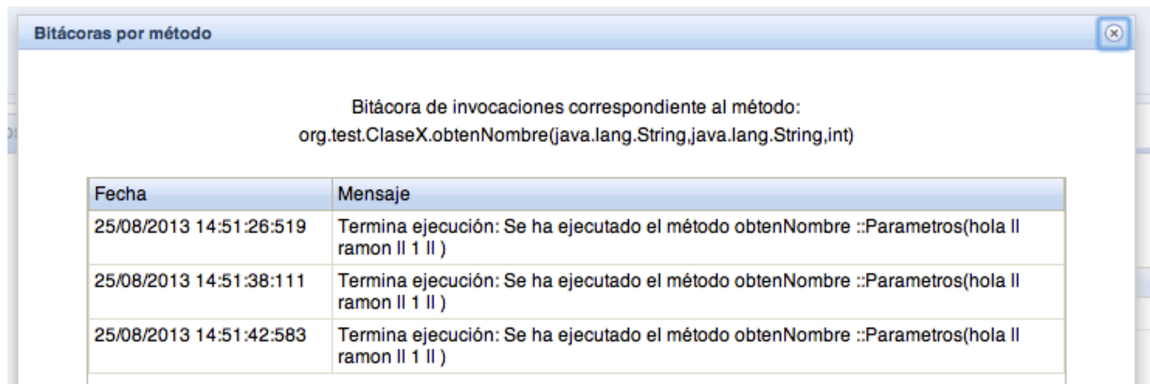

Figura 5.7 Bitácora de un método.

"El sistema permitirá habilitar el monitoreo de ejecución de métodos"

En la sección de Monitoreo, se encuentra la opción "Habilitar monitoreo". Esta opción muestra la pantalla (Figura 5.8) que permite habilitar/deshabilitar el monitoreo de las ejecuciones de métodos.

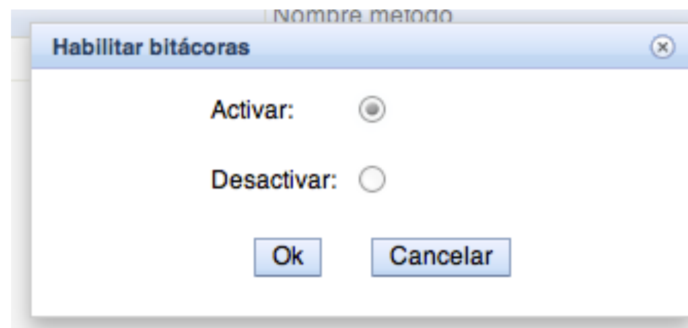

Figura 5.8 Activación del monitoreo de las ejecuciones de métodos.

Al habilitar el monitoreo, cada vez que se realiza una invocación a un método dentro del sistema target, el sistema de bitácoras registra dentro del sistema:

- El método invocado.
- El estatus de la invocación.
- La duración de la ejecución del método.

Todas estos datos podrán ser visualizados en otras pantallas, las cuales se detallan más adelante.

"El sistema permitirá visualizar las estadísticas de ejecución de los métodos"

En la categoría de Monitoreo, bajo la opción de "Estadísticas de invocaciones", se muestra la pantalla que presenta el resumen de las invocaciones para todos los métodos.

Invocaciones de métodos

Se muestran las estadísticas de las invocaciones de los métodos de los objetos que se encuentran dentro del contenedor.

Nombre Método	Clase	Número invocaciones	Duración Total	Fecha inicio	Fecha fin	Duración promedio	Número de invocaciones promedio
org.test.ClaseX.obtenN	org.test.ClaseX	4	1263			0	0

Figura 5.9 Pantalla de estadísticas
de las ejecuciones de los métodos.

En la tabla mostrada en la Figura 5.9 se pueden visualizar cúales son los métodos más invocados, los que han demorado más en ejecutarse, el número de invocaciones promedio por minuto, así como su duración promedio.

"El sistema mostrará gráficas de número de ejecuciones por minuto y la duración de cada una de las ejecuciones"

Al seleccionar cualquiera de los registros mostrados en la vista de estadísticas de invocaciones (mencionado en el punto anterior), se muestra una gráfica (Figura 5.10) interactiva del comportamiento de las invocaciones en un cierto periodo de tiempo.

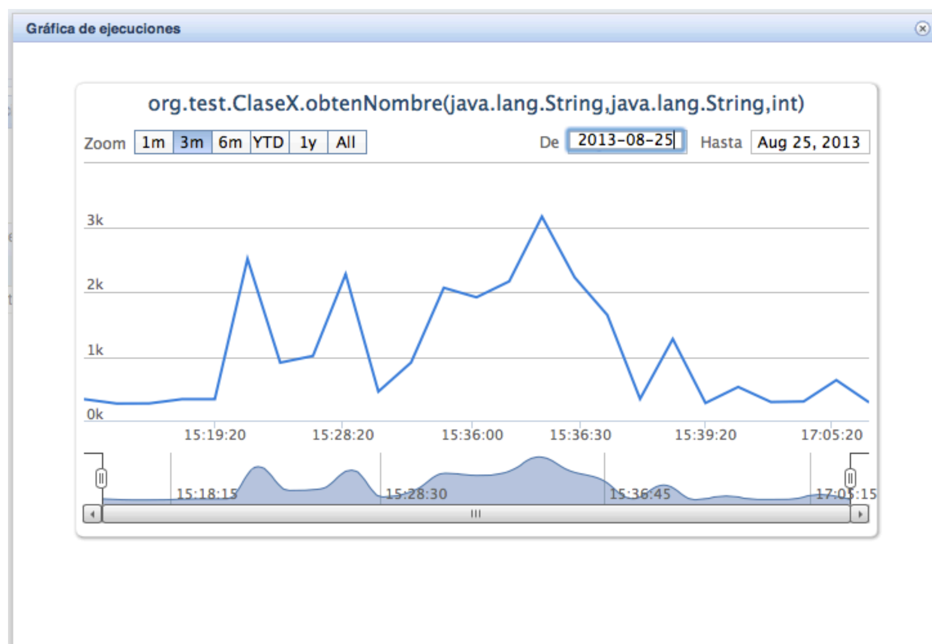

Figura 5.10 Gráfica de la duración de
las invocaciones de los métodos.

El monitoreo de invocaciones está condicionado a que se haya habilitado previamente dicho monitoreo.

En el presente capítulo se listaron los requerimientos que fueron identificados en la etapa del análisis del sistema y se realizó una verificación del cumplimiento de los mismos.

La versión ejecutable del sistema se puede acoplar a una aplicación web basada en Spring, permitiendo al sistema generador de bitácoras, tener acceso a los objetos de la aplicación web.

Una vez acoplado, el sistema facilita la configuración de las bitácoras a generar. El sistema desarrollado realiza todas sus tareas, sin modificar de alguna forma, el sistema target. De igual manera, no se necesita intervenir en el código base de la aplicación web, para iniciar la generación de registros en la bitácora.

En resumen, el sistema generador de bitácoras actúa sobre el sistema target de una forma no invasiva. En el siguiente capítulo, se analiza el impacto que tiene en una aplicación web, la utilización de un sistema de bitácoras no invasivo.

6

Impacto

La facilidad que ofrece el sistema de generación de bitácoras para crear un ambiente en el cual una aplicación puede registrar sus eventos relevantes sin afectar su implementación básica, es fundamental para mantener separadas las responsabilidades asignadas al sistema.

La implementación del sistema de bitácoras fomenta el principio de separación de responsabilidades. Esto quiere decir que cada componente del sistema debe implementar una y sólo una responsabilidad. Esta separación lo que busca es que los componentes de un sistema se mantengan simples y fáciles de entender.

¿Cómo ayuda el sistema de generación de bitácoras en esta separación?

Comúnmente un sistema que requiere registrar eventos en una bitácora, lo hace delegando estas responsabilidades a una biblioteca especializada. Sin embargo, dependiendo de las necesidades de cada módulo, el código central de cada uno de ellos puede irse poblando con llamadas a la biblioteca de logging.

Cada una de las llamadas al objeto ***logger*** indica una delegación de responsabilidad hacia la biblioteca de bitácoras. El código fundamental del componente, se ve invadido por funcionalidad específica de bitácoras.

En general, componentes como los que se describen, pueden ser representados como se muestra en la Figura 6.1.

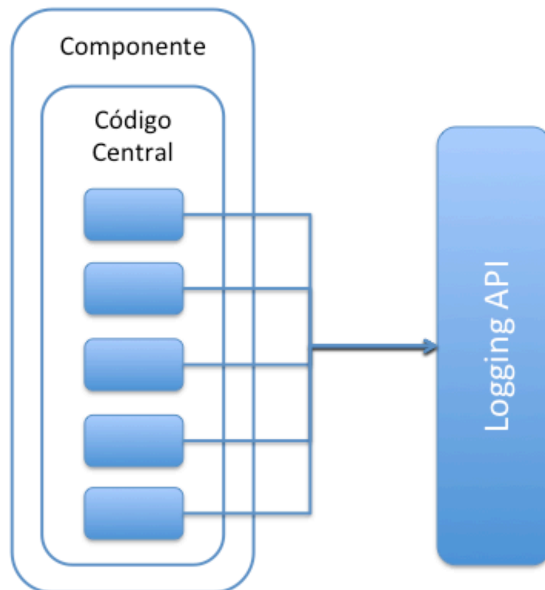

Figura 6.1 Implementación de bitácoras antes del sistema de bitácoras.

En la Figura 6.1, se puede ver cómo es que el código central de un componente se ve afectado por un número indeterminado de llamados al API de logging. Esto provoca que las responsabilidades se mezclen y por lo tanto el componente resultante sea menos mantenible.

Para entender este punto sólo haría falta imaginar el trabajo que requeriría un cambio en la interfaz del API de logging o la sustitución del API actual. Cada una de las invocaciones, que se encuentran dispersas por todo el código, tendrían que ser localizadas y modificadas. Obviamente el costo de dicha refactorización sería muy alto, directamente proporcional al tamaño del sistema.

Con el sistema de generación de bitácoras, se resuelve esta dispersión de código. El sistema centraliza la funcionalidad específica de generación de bitácoras. Además haciendo uso de POA se logra aplicar la funcionalidad mencionada, en los puntos necesarios del sistema target, haciéndolo sin necesidad de modificar el código.

Al implementar el sistema de bitácoras en una aplicación web, los componentes del sistema target, se comportarán como se muestra en la figura 6.2.

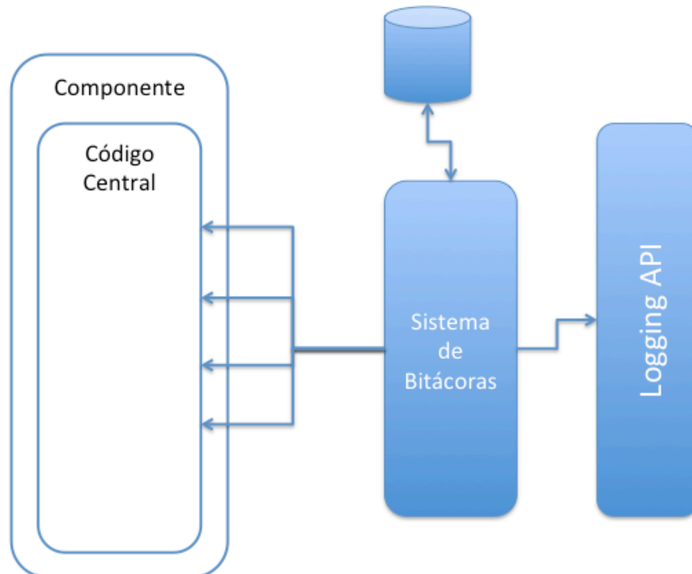

Figura 6.2 Implementación de bitácoras, utilizando el proyecto desarrollado.

Las responsabilidades se invierten. El código central del componente se limpia totalmente de llamadas al API de bitácoras, es más, permanece totalmente ignorante de que alguien más está generando sus bitácoras de ejecución.

Ahora el encargado de esta responsabilidad, es el sistema de bitácoras. Este sistema utilizando la información provista por una base de datos, determina los puntos en la ejecución del componente, donde es útil generar un registro en la bitácora.

Todo lo anterior fue logrado gracias a POA. Con la implementación provista por Spring, fue posible crear dinámicamente objetos proxy cuya responsabilidad principal es la generación de bitácoras. Todo esto sin necesidad de modificar la implementación de alguno de los componentes.

Ahora al tener componentes libres de la lógica de generación de bitácoras se tiene un software de mayor calidad. Se logra que los componentes solamente tengan una responsabilidad. Los componentes son más simples.

¿Por qué buscar que los componentes de un sistema sean simples?

Una solución simple siempre será mejor. Una implementación simple de un componente trae consigo otros beneficios, como mayor facilidad de mantenimiento y mayor capacidad de reúso.

Las características mencionadas anteriormente siempre serán deseables y son necesarias para considerar que un software es de calidad.

El sistema de generación de bitácoras logra encapsular una de las responsabilidades más comunes en un sistema: el logging.

Esta modularización de la responsabilidad del logging, apoyada con POA, provoca una separación total entre el código base del sistema y el código referente al logging.

¿Cómo se benefician las aplicaciones web?

Las aplicaciones web al estar formadas por componentes más sencillos, provocan que los nuevos desarrollos sean más simples, rápidos y baratos. Un código más sencillo es más fácil de entender. Un código más fácil de entender es más fácil de modificar. Un código más fácil de modificar hace que su evolución sea más ágil.

A final de cuentas la evolución de los componentes se ve beneficiada con la implementación del sistema de bitácoras. Proyectos futuros así lo percibirán.

El sistema obtenido sin duda ayuda a que el sistema target sea de mayor calidad, separando responsabilidades. Sin embargo, esta separación no es suficiente, ya que en el diseño del software pueden surgir muchas más responsabilidades que abarquen todo o gran parte del sistema.

Responsabilidades como concurrencia, seguridad o transaccionalidad, son del tipo de responsabilidades cruzadas, que amenazan la calidad del código implementado, debido a que a pesar de que en la fase de diseño estas responsabilidades pueden ser consideradas como un módulo aparte, en la fase de implementación su código puede estar disperso por todo el sistema, mezclándose con el código base de cada componente. Esta condición reduce la cohesión, aumenta el acoplamiento entre componentes y por lo tanto reduce la calidad del software.

En resumen, el sistema de generación de bitácoras muestra cómo es que deben ser implementadas las responsabilidades cruzadas, de tal manera que se evite la mezcla y dispersión de código.

Es conveniente para futuras implementaciones, que este modelo sea considerado para el desarrollo de requerimientos similares.

7

Conclusiones

Como se trató con anterioridad, una responsabilidad es cualquier tarea que un sistema debe cumplir.

Un sistema siempre tiene un objetivo principal que debe llevar a cabo. Dicho objetivo es conocido como responsabilidad esencial del sistema. Sin embargo, es común que las responsabilidades, para cumplir sus tareas, requieren de responsabilidades "secundarias", las cuales ofrecen servicios de sistema genéricos y auxilian a las responsabilidades esenciales en el cumplimiento de sus tareas.

Algunas veces estas responsabilidades auxiliares son requeridas por una gran parte de los componentes del sistema. Esta necesidad puede generar problemas, como la dispersión de código y la mezcla de responsabilidades.

La alta cohesión y el bajo acoplamiento son dos características siempre deseables en el software. Hacen que el software sea de más calidad y su evolución sea más factible: se reducen los costos de detección y corrección de errores, y el costo de la implementación de modificaciones y nuevas características del sistema.

Por todo lo anterior, es de suma importancia lograr una separación adecuada de responsabilidades desde las etapas de análisis y diseño (cuando se identifican y asignan responsabilidades) y mantenerla durante la implementación del sistema a desarrollar.

Existen varias propuestas de diseño para lograr la separación de responsabilidades deseada.

En la presente tesis se propone el uso del paradigma orientado a aspectos.

Dicho paradigma propone precisamente una solución óptima en la implementación de responsabilidades cruzadas.

Propone encapsular la lógica de responsabilidades cruzadas en componentes llamados aspectos. Posteriormente los aspectos son aplicados de manera dinámica en puntos específicos durante el flujo de ejecución de un componente.

Con la implementación propuesta se separan totalmente las responsabilidades esenciales de las cruzadas. Incluso se puede decir que la responsabilidad esencial no tiene conocimiento alguno de la existencia del componente implementador de la responsabilidad cruzada.

Implementando las responsabilidades cruzadas con POA, se logra que los componentes del sistema sean altamente cohesivos, al implementar únicamente una responsabilidad y que mantengan un bajo acoplamiento debido al desconocimiento total de la existencia de la responsabilidad cruzada.

Para implementar utilizando programación orientada a aspectos, no existe un lenguaje que sea puramente de este paradigma. En cambio existen lenguajes orientados a objetos que cuentan con extensiones que ofrecen esta funcionalidad.

Spring Framework es una de las varias implementaciones que existen en el mercado. Ofrece una implementación bastante completa del paradigma orientado a aspectos.

Este framework pretende simplificar el desarrollo de aplicaciones empresariales en la plataforma Java. Busca promover el uso de buenas prácticas con el objetivo de generar un software de mayor calidad.

Por lo anterior, se consideró conveniente el uso de este framework para la implementación del proyecto.

El sistema de generación de bitácoras es una aplicación que pretende separar la lógica de generación de bitácoras de cualquier sistema que esté basado en Spring framework.

La generación de bitácoras es una responsabilidad bastante común en los sistemas actuales. Es esencial en el monitoreo del comportamiento de un sistema.

Esta responsabilidad es del tipo cruzado, ya que puede ser requerida por todo el sistema, con el objetivo de llevar un registro detallado de los eventos relevantes que se presentan durante la ejecución cotidiana del sistema.

El sistema de generación de bitácoras ofrece una propuesta de cómo se puede implementar una responsabilidad cruzada, en este caso el logging, utilizando programación orientada a aspectos. De tal manera que la implementación resultante encapsule adecuadamente la responsabilidad cruzada, fomentando la alta cohesión y el bajo acoplamiento.

Aplicando este sistema sobre un sistema web (conocido como sistema target) se logra que la lógica de generación de bitácoras, que pudiera requerirse en el sistema target, se separe totalmente de éste. Toda la implementación de generación de bitácoras se localiza en el sistema hecho para este fin.

La utilización de la programación orientada a aspectos hace que el ligado de la responsabilidad cruzada con la esencial, sea llevado a cabo en tiempo de ejecución. Por lo que la responsabilidad esencial permanece ignorante de la cruzada.

La utilización del sistema de generación de bitácoras ayuda a mejorar la calidad del software, sin embargo no es suficiente. Existen muchas más responsabilidades que deben ser consideradas y tratadas, para evitar los problemas que se han mencionado.

El uso del modelo de ciclo de vida en espiral, fue importante debido a que permitió segmentar el problema, en casos más pequeños. Esto abrió las puertas para que el análisis y el diseño de cada una de las partes involucradas fuera más sencillo.

También el uso de este modelo dio la posibilidad de que en etapas más tempranas del proyecto, se contara con artefactos ejecutables, lo cual permitió realizar una evaluación continua del avance del proyecto.

El desarrollo se apoyó también en herramientas como Maven, que permitió automatizar gran parte de la implementación del proyecto, tales como la construcción, resolución de dependencias, pruebas unitarias y despliegue.

La preparación profesional recibida en la Facultad de Ingeniería, fue determinante en la preparación y elaboración del presente trabajo. La conjunción de las diversas disciplinas cursadas durante la estancia en la Facultad permitió que la labor de investigación y desarrollo se llevara a cabo con éxito.

Bibliografía

Bass Len, Clements Paul, Kazman Rick, "Software Architecture in Practice", Addison Wesley, 2nd Edition, Boston E.E.U.U. Abril 2003.

Collofello James "Introduction to Software Verification and Validation" Carnegie Mellon University, Software Engineering Institute, 2nd Edition, E.E.U.U. 1988.

Fowler Martin, "UML Distilled" Addison-Wesley, Third Edition, E.E.U.U. 2004.

Gamma, Helm, Johnson, Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software" Addison Wesley, Printing 37th, E.E.U.U. 2009.

Gomaa Hassan, "Software Modeling & Design" Cambridge University Press, 1st Edition, E.E.U.U. 2011.

Johnson, Rod "Spring Framework Reference Documentation" Spring Source, 3.2.0 Release, E.E.U.U. 2012.

Kiczales, Lamping, Mendhekar, Maeda, Videira, Loingtier, Irwin, "Aspect-Oriented Programming" Springer-Verlag, 1st Ed, E.E.U.U. 1997.

Kuchana Partha "Software Architecture Design Patterns in Java" Auerbach, 1st Edition 2004.

Larman, Craig, "Aplying UML and Patterns" Prentice Hall, Second Edition, E.E.U.U 1997.

Noble, Schmidmeier, Pearce, Black, "Patterns of Aspect-Oriented Design" EuroPlop, 1st ed, E.E.U.U. 2007.

Pollice Gary, "A look at aspect-oriented programming" IBM developer Works, E.E.U.U. 2004.

Rozanski Nick, Woods Eoin, "Software Systems Architecture" Addison Wesley, 1st Edition, E.E.U.U. 2005.

Russell Matthew "Dojo The definitive guide" O'Reilly, 1st Edition, E.E.U.U. 2008.

Sommerville Ian, "Software Engineering" Pearson 9na Edición, E.E.U.U. 2011.

Walls Craig, "Spring in Action" Manning, Third Edition, E.E.U.U. 2011.

Zaldívar Esquivel, Orlando; Zaldívar Zamorategui, Orlando, "Apuntes de Ingeniería de Software", Facultad de Ingeniería, UNAM, México, 2012.

Mesografía

"Design Patterns" http://sourcemaking.com/design_patterns (consultado el 15/04/2013).

Eeles Peter, "What is a Software Architecture?" Febrero 2006, IBM developer Works http://www.ibm.com/developerworks/rational/library/feb06/eeles/ (consultado el 10/03/2013).