UNIVERSIDAD NACIONAL DEL CENTRO
DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS

# Una herramienta para la detección de problemas de performance en sistemas J2EE

por
Leonardo Moreno

Dr. Alejandro Zunino
Director

Ing. Marco Crasso
Co-Director

Tandil, Julio de 2008

# Resumen

Desarrollar sistemas distribuidos de gran escala siempre fue complejo. Actualmente, una de las tecnologías más elegidas a la hora de construir un sistema es J2EE, debido a que simplifica el desarrollo al proveer muchos servicios de bajo nivel en forma transparente. Sin embargo más de la mitad de los sistemas desarrollados en esta tecnología no cumplen con sus requerimientos de *performance*. Esto se debe en parte a que sobre la *performance* de un sistema J2EE impacta una compleja combinación de factores, que hace muy difícil para los desarrolladores entender la relación entre sus decisiones de diseño y la *performance* final de la aplicación. Por este mismo motivo, solucionar los problemas de *performance* es una tarea igualmente difícil.

En este contexto, surge la necesidad de simplificar la tarea de optimizar aplicaciones J2EE. Con esa motivación, este trabajo presenta un nuevo tipo de herramientas que pueden asistir a los desarrolladores en la corrección de los problemas de *performance*. La herramienta propuesta analiza una aplicación J2EE buscando problemas conocidos y genera un reporte con las correcciones que se deben hacer para mejorar el desempeño, en forma automática. La información necesaria no es solicitada al desarrollador, sino que se debe obtener monitoreando la aplicación.

Para evaluar la efectividad de esta propuesta, se ha materializado en una herramienta implementada en Java que se utiliza para optimizar 3 aplicaciones.

**Palabras clave**: J2EE, *performance*, optimización, antipatrones de diseño, refactorización.

# Índice general

# Índice de figuras

# Índice de cuadros

# Glosario

**API** Application Programming Interface

**ASP** Active Server Pages

**EJB** Enterprise JavaBeans

**HTTP** Hyper Text Transfer Protocol

**J2EE** Java 2 Enterprise Edition

**JDBC** Java Database Connectivity

**JMS** Java Message Service

**JNDI** Java Naming and Directory Interface

**JSP** Java Server Pages

**JVM** Java Virtual Machine

**PEPA** Performance Evaluation Prediction Algebra

**UML** Unified Modeling Language

**URI** Uniform Resource Identifier

**URL** Uniform Resource Locator

**W3C** World Wide Web Consortium

**XML** eXtensible Markup Language

**XSL** eXtensible Stylesheet Language

**XSLT** XSL Transformations

CAPÍTULO 1

# Introducción

Este capítulo está organizado como sigue: en la sección 1.1, se describen los problemas que motivaron la generación de esta tesis; en la sección 1.2, se describe la solución propuesta en este trabajo; por último, en la sección 1.3, se indica cómo ha sido organizado el resto del trabajo.

## 1.1. Motivación

Desarrollar sistemas distribuidos de gran escala es complejo. Una de las tecnologías más elegidas a la hora de construir un nuevo sistema es J2EE, que simplifica el desarrollo proveyendo un conjunto de servicios estándar [4]. Sin embargo, la tecnología no garantiza por sí misma la buena *performance* del sistema construido. De hecho, más de la mitad de los sistemas desarrollados en esta tecnología no cumplen con sus requerimientos de *performance* [33].

Esto se debe en parte a que sobre la *performance* de un sistema J2EE impacta una combinación de factores, tales como el diseño e implementación de la aplicación, las características de los distintos servidores J2EE, y la eficiencia de las capas de comunicación. Esta compleja combinación de factores hace que sea muy difícil para los desarrolladores aislar y entender el impacto que tendrán sus decisiones de diseño en la *performance* final de la aplicación, siendo esas decisiones las que pueden determinar que el sistema cumpla o no con la *performance* y escalabilidad requerida [6]. En el capítulo 2 se verán estas características en mayor detalle para comprender mejor las dificultades que presenta esta tecnología.

Existen distintas técnicas para predecir el desempeño que tendrá un determinado sistema que se basan en una especificación de sus principales componentes y sus interacciones, y la simulación de distintas condiciones de carga y uso del sistema [8, 25]. A pesar de las bondades de este tipo de modelos de *performance*, estos modelos no aportan información relevante cuando no se posee un conocimiento preciso de condiciones de carga y uso que deberá afrontar el sistema ni del desempeño de componentes integrados, por ejemplo los

servicios provistos por un servidor J2EE. Además de que construir estos sistemas de manera que cumplan con sus restricciones de *performance* es una tarea muy difícil aún para desarrolladores expertos en J2EE, e incluso si se logra construir, en la etapa de mantenimiento el software se vuelve más complejo y pierde parte del diseño original [29], requiriendo un nuevo análisis de *performance*.

Actualmente, impulsados por el crecimiento de la industria del software, nuevos factores dificultan aún más el temprano análisis de las implicancias en *performance* de cada decisión de diseño e implementación de una aplicación. El *time-to-market,* la escasez de expertos en J2EE, y los compromisos para finalizar la construcción de los proyectos o incorporarle nuevas funcionalidades lo antes posible, en muchas ocasiones transforman a la *performance* en un ítem a ser analizado después de que el sistema haya sido codificado, integrado, testeado e instalado.

Claramente, los factores mencionados favorecen a las técnicas de análisis de performance en tiempo de ejecución, por sobre las técnicas que abordan el problema tempranamente. Además, existen características intrínsecas de las técnicas del primer tipo que las hacen apropiadas para los sistemas J2EE, tales como:

- no requieren del código fuente, que habitualmente no está disponible para los componentes integrados, como por ejemplo los servicios provistos por el servidor J2EE;
- permiten trabajar con relaciones entre componentes definidas por configuración que se efectivizan en tiempo de ejecución, como ocurre en el caso de las aplicaciones basadas en inyección de dependencias [22], o integradas mediante mecanismos de reflexión [11].

Surge entonces la necesidad de herramientas que asistan a los desarrolladores en la corrección de los problemas de *performance* que recién van a detectarse cuando el sistema esté terminado. Como consecuencia de esto, la herramienta que ayude a resolver los problemas de *performance* en estos sistemas debe monitorearlos en tiempo de ejecución. Esto permite además una integración del análisis de *performance* en las metodologías iterativas que predominan actualmente, tales como Extreme Programming [19] o Rational Unified Process [26], que requieren una implementación ejecutable del sistema en cada iteración del desarrollo.

Las herramientas actuales que monitorean *performance* [38] tienden a generar excesiva información sobre el sistema en ejecución, lo que puede hacer muy complicada la tarea de localizar el origen de los problemas de *performance.* Estas herramientas pueden dar mucha información, pero no proveen soluciones. Hace falta un desarrollador con mucha experiencia para interpretar esa información y resolver los problemas en forma eficiente [12].

Todos los factores mencionados hacen que gran parte de los proyectos tengan que solucionar problemas de *performance* una vez que el sistema ya está construido. Para lograrlo, los desarrolladores pueden disponer de mucha información generada por herramientas de monitoreo, pero sin la preparación suficiente ni el tiempo para analizar todos los datos y sacar conclusiones, el desarrollador suele perderse y los problemas de *performance* no se solucionan nunca.

## 1.2. Solución propuesta

Dada la situación actual descrita en la sección 1.1, se hace necesaria una herramienta que asista al desarrollador cuando debe encargarse de mejorar el desempeño de una aplicación que no se está comportando satisfactoriamente.

Este trabajo propone una herramienta que tiene como objetivo asistir a ese desarrollador, mediante un proceso automatizado en 3 pasos:

1. Obtener información del sistema 4.1.
2. Identificar problemas de diseño y su solución 4.2.
3. Presentación de resultados 4.3.

El objetivo de esta herramienta es diagnosticar los problemas de diseño que tenga la aplicación y puedan ser corregidos mediante la refactorización de partes del código. Por lo tanto se asume que se dispone libremente del código fuente de la aplicación a corregir, y no se puede utilizar la herramienta si no se tiene acceso al mismo o no puede compilarse con información para depuración.

Teniendo esta herramienta a su disposición, el desarrollador deberá simplemente ejecutar la aplicación, y obtendrá como resultado un listado de problemas, con la solución correspondiente y la indicación de cómo aplicarla. Ya no tiene la necesidad de ser experto en *performance*, ni de analizar grandes cantidades de información para lograr que la aplicación funcione de acuerdo a los requerimientos del sistema.

### 1.2.1. Obtener información del sistema

Para obtener la información de la aplicación a optimizar, la herramienta utiliza técnicas de monitoreo en tiempo de ejecución. Esto obliga a tener construida la parte del sistemas que se quiere optimizar. Para poder monitorear el sistema, la aplicación debe estar ejecutándose. Si la aplicación ya está en producción, entonces es el usuario final quien la usa y se puede obtener información real sobre el uso final de la aplicación. Si todavía está en etapa de desarrollo, la aplicación puede ser ejecutada por el mismo programador. Una alternativa es utilizar herramientas de carga que recorren las diferentes funcionalidades de la aplicación en forma automática.

Se cual sea el usuario del sistema, mientras se ejecutan sus funciones, la herramienta recolecta en forma transparente información acerca de las interacciones entre los distintos componentes del sistema. Esta etapa del proceso está diseñada para consumir la menor cantidad de recursos posibles, de manera que pueda ser utilizada en ambientes productivos sin problema.

La información que se obtiene aquí es similar a la que pueden obtener otras herramientas de monitoreo. La diferencia radica en que la información recolectada no es presentada al usuario, sino que con la misma se construye un modelo dinámico de la aplicación, que es utilizado en el siguiente paso.

Gracias a que se obtiene la información necesaria directamente de la aplicación, no hace falta involucrar a expertos en el negocio. Cualquier desarrollador puede hacer las modificaciones en el código requeridas por la herramienta para que monitoree el sistema y recolecte los datos que precisa.

### 1.2.2. Identificar problemas de diseño y su solución

La descripción del sistema en tiempo de ejecución que se obtiene en el paso anterior es analizada en busca de antipatrones de diseño. Los antipatrones son conceptualmente similares a los patrones de diseño, sólo que lo que documentan son errores cometidos con frecuencia en la construcción del software, y su solución.

La herramienta está preparada para identificar algunos de estos antipatrones, por lo que una vez detectado un problema se le informa al usuario cuáles son las acciones correctivas que debe aplicar, evitándole la larga tarea de analizar grandes volúmenes de información.

El proceso de análisis se lleva a cabo con un motor de reglas, lo que permite definir los antipatrones en un lenguaje declarativo, y agregarlos sin necesidad de tener el código fuente de la herramienta. Esto permite que, en el caso de que se descubra un nuevo problema, se pueda programar una nueva regla para detectarlo y de esa manera automatizar el proceso de encontrar todas las ocurrencias del antipatrón.

El análisis de toda esa información es un proceso computacionalmente costoso, por lo que no se realiza en simultáneo con el paso anterior de recopilación de información, sino que se ejecuta en forma posterior tomando como entrada el modelo dinámico generado. Una ventaja que surge de tener este proceso desacoplado del inicial es que se pueden ejecutar distintos conjuntos de reglas sobre el mismo modelo dinámico, para evaluar diferentes aspectos de la aplicación a medida que crece el entendimiento de la misma.

La capacidad de la herramienta de identificar la ocurrencia de los antipatrones hace prescindible la participación de un experto en optimización de aplicaciones J2EE.

### 1.2.3. Presentación de resultados

Habiendo identificado la ocurrencia de un antipatrón en el sistema, se le indica al usuario aplicar al código el refactor sugerido como solución para el antipatrón encontrado. Utilizando la información de debug que se encuentra en el código compilado de la aplicación, se puede identificar las líneas de código fuente que participan del antipatrón, y son seguramente las líneas que haya que modificar al refactorizar el diseño.

Muchas veces el refactor que se recomienda es la introducción de un patrón de diseño , , al código. De esta manera, la herramienta permite que desarrolladores no especializados en *performance* sepan qué cambios hacer al diseño para que la aplicación mejore su desempeño.

Por la forma en que se detectan los problemas, tampoco es necesario que la persona que realiza el proceso de optimización conozca cómo está implementada la aplicación. La herramienta le indicará qué refactor debe aplicar en qué líneas del código fuente, de manera que cualquier desarrollador pueda resolver los problemas.

## 1.3. Organización

Este trabajo está organizado de la siguiente manera. En el capítulo 2 se presentan las tecnologías que actualmente se utilizan para el desarrollo de aplicaciones J2EE, se expone su complejidad y las principales causas de problemas de desempeño de estas aplicaciones. En el capítulo 3se analizan algunas propuestas para solucionar dichos problemas, mientras que en el capítulo 4 se detalla el enfoque propuesto por este trabajo para solucionar los problemas de desempeño.

Basándose en las ideas de este enfoque se ha implementado un sistema experto que asiste en el proceso de optimización. El diseño y la implementación del mismo se detalla en el capítulo 5. Con el objetivo de medir empíricamente la efectividad del enfoque se utilizó la herramienta para optimizar aplicaciones reales. Los resultados obtenidos se presentan en el capítulo 6.

Finalmente, en el capítulo 7 se presentan las conclusiones del trabajo y se indican las extensiones que se harán al sistema en el futuro.

# CAPÍTULO 2

# Contexto

Hoy en día son cada vez más los desarrolladores que tienen que construir aplicaciones distribuidas y transaccionales para la empresa, y por lo tanto aprovechan la velocidad, seguridad y confiabilidad de las tecnologías del lado del servidor. La plataforma de desarrollo J2EE provee las bases para el diseño, desarrollo, ensamblado e instalación de aplicaciones empresariales basadas en componentes. Esta plataforma ofrece un modelo de aplicaciones distribuidas multicapa, componentes reusables, seguridad unificada y control de transacciones flexible. En consecuencia, J2EE permite reducir costos y acelerar el diseño y desarrollo de aplicaciones.

Este trabajo se enfoca en las aplicaciones J2EE multicapa, por lo que veremos sus características en el resto del capítulo.

## 2.1. Aplicaciones Web y J2EE

En los primeros tiempos de la computación cliente-servidor, cada aplicación tenía su propio programa cliente y su interfaz de usuario, los cuales tenían que ser instalados separadamente en la estación de trabajo de cada usuario. Una mejora al servidor, como parte de la aplicación, requería típicamente una mejora de los clientes, es decir, actualizar el *software* esparcido por las estaciones de trabajo, añadiendo un costo de soporte técnico y disminuyendo la eficiencia del personal.

En contraste, las aplicaciones Web generan dinámicamente una serie de páginas en un formato estándar, como HTML, el cual es soportado por navegadores Web comunes (eje., Mozilla Firefox, Opera, MS Internet Explorer). Aquí, alcanza con actualizar dichas páginas, para reflejar los cambios en todos los clientes. Generalmente, cada página Web individual es enviada al cliente como un documento estático, pero la secuencia de páginas provee de una experiencia interactiva. Además, se utilizan lenguajes interpretados del lado del cliente, tales como JavaScript, para añadir elementos dinámicos a la interfaz de usuario.

Desde el punto de vista de la tecnología J2EE, una aplicación Web es una extensión dinámica de un servidor de aplicaciones, que genera páginas Web interactivas con algún lenguaje textual, como es HTML o XML, además de contenido dinámico. Estas aplicaciones son las más comunes y se consideran *orientadas a la presentación*. Otro tipo de aplicaciones Web son las *orientadas a servicios*, que implementan puntos de acceso remoto a servicios, y típicamente son accedidas por aplicaciones Web orientadas a la presentación.

La especificación J2EE define que los componentes Web proveen la capacidad de extensión dinámica de los servidores Web. Como veremos en la sección 2.4.2 los componentes Web son Java Servlets, páginas JSP, o puntos de acceso remoto a servicios. Si bien las páginas JSP y los *servlets* pueden ser utilizados en forma indistinta, tienen distintas fortalezas. Los *servlets* son más adecuados para aplicaciones orientadas a servicios y para las funciones de control de las aplicaciones orientadas a presentación, como distribuir los requerimientos o generar las respuestas que no son textuales. Por otro lado, las páginas JSP son más adecuadas para generar contenido textual, como pueden ser HTML, imágenes SVG (Scalable Vector Graphics), WML (Wireless Markup Language), o XML.

La especificación J2EE también sugiere la tecnología que se debe utilizar para implementar la lógica de negocio de las aplicaciones. Por lógica de negocio nos referimos a la porción del código que realiza el propósito de la aplicación. Por ejemplo, en una aplicación de control de inventarios, los métodos llamados *verificarExistenciaEnStock* y *ordenarProducto* podrían implementar la lógica del negocio. En los sistemas J2EE la lógica de negocio está implementada en *enterprise beans*, también conocidos como *EJB* por ser las siglas de la tecnología que les da soporte, *Enterprise JavaBeans*. Los enterprise beans o EJBs son componentes que simplifican el uso de los recursos del sistema facilitando el desarrollo de las aplicaciones.

La interacción entre estos componentes suele ser la siguiente:

1. el cliente envía un requerimiento HTTP al servidor Web,
2. el servidor Web transforma el requerimiento en un objeto Java que contiene la información enviada por el cliente, y lo hace accesible a un componente Web, ya sea a través de un parámetro de un método de un servlet o como variable global en un JSP,
3. el componente Web puede interactuar con bases de datos, EJBs, u otros componentes Web para armar un objeto definido por la Java Servlet API que contiene la información que se desea enviar en respuesta al cliente,
4. el servidor Web transforma el objeto creado por el componente Web en una respuesta HTTP que el cliente entiende y se la envía.

Las aplicaciones Web construidas con tecnología J2EE utilizan tradicionalmente una arquitectura multicapa [23], y el paso 3 contiene interacciones remotas que pueden ser causa de problemas de desempeño. En la siguiente sección veremos este modelo arquitectónico con mayor detalle.

## 2.2. Aplicaciones J2EE Multicapa

La lógica de las aplicaciones multicapas se divide entre componentes de acuerdo a su funcionalidad, y los diferentes componentes de la aplicación se instalan en diferentes máquinas

dependiendo de la capa a la que pertenecen. Típicamente estas capas son:

- **Capa Cliente**: componentes ejecutándose en la máquina del cliente, usualmente contienen la interfaz a usuario.
- **Capa Web**: componentes ejecutándose en el servidor J2EE que tienen la responsabilidad de interactuar con los clientes de la aplicación, que usualmente son navegadores de Internet.
- **Capa de Negocio**: componentes ejecutándose en el servidor J2EE, utilizados usualmente por la capa Web o por clientes ricos para realizar las operaciones propias del negocio, ya que estos componentes contienen la lógica de negocio.
- **Capa de datos**: la base de datos suele tener su propio servidor para ejecutarse.

Las aplicaciones J2EE multicapa suelen llamarse de 3 capas más allá de que puedan tener más, dado que se distribuyen generalmente en 3 conjuntos de máquinas: las de los clientes, las máquinas del servidor J2EE, y las de la base de datos. Gráficamente, la Figura 2.1 muestra

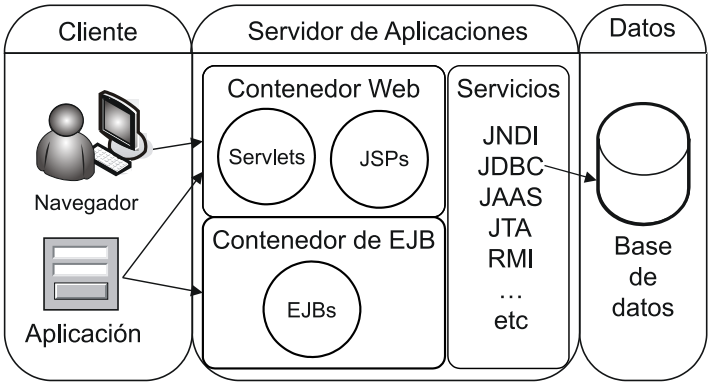


**Figura 2.1:** Vista lógica de la arquitectura de tres capas.

estas capas. Las aplicaciones de 3 capas que siguen este modelo extienden la configuración tradicional de cliente-servidor agregando un servidor de aplicaciones entre los clientes y los datos.

En general, las aplicaciones multicapas son difíciles de escribir porque requieren mucho código de alta complejidad para manejar transacciones, mantener estados, múltiples hilos de ejecución, administrar recursos y muchos otros detalles de bajo nivel [4]. La arquitectura J2EE provee todos esos servicios de bajo nivel en la forma de un servidor de aplicaciones independiente de la plataforma, haciendo que las aplicaciones J2EE sean más fáciles de escribir al no tener que ocuparse de esos detalles. Por otra parte, al ser una arquitectura modular, permite que los componentes J2EE sean fácilmente reutilizables en otra aplicación ya que sólo contienen la lógica de negocio [31]. Por ejemplo, un componente encargado de administrar usuarios, puede ser ensamblado en muchas aplicaciones diferentes sin necesidad de ser modificado, independientemente de que las otras aplicaciones tengan otras políticas de seguridad, otros tipos de clientes, o utilicen diferentes mecanismos de persistencia.

Desde una perspectiva física, la plataforma J2EE utiliza un modelo distribuido y multicapa para las aplicaciones empresariales. Cada capa suele estar instalada en un conjunto distinto de máquinas, tanto para lograr una mayor confiabilidad como para mejorar el desempeño del sistema. Esta distribución provoca que la comunicación entre las diferentes capas tenga un alto costo, dado que las interacciones remotas tienen un alto costo asociado al transporte por la red. Además de tener cada capa en diferentes máquinas, es habitual que cada capa a su vez esté compuesta por diferentes máquinas, por los mismos motivos. En la Figura 2.2

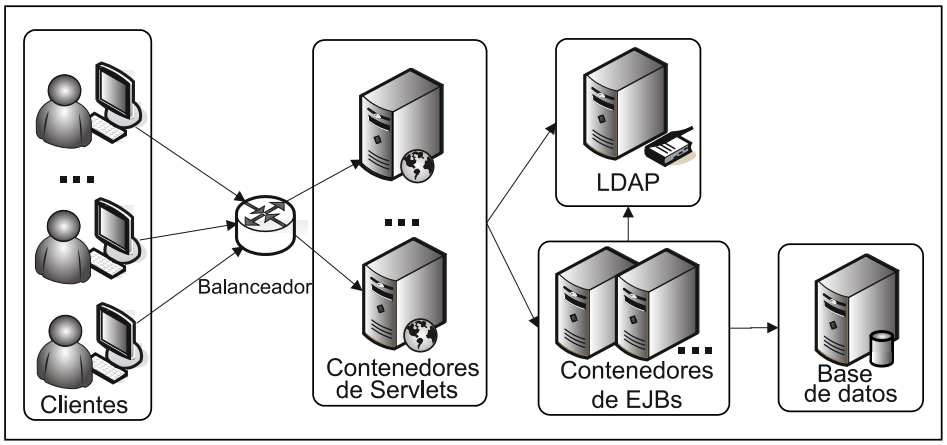


**Figura 2.2:** Vista física de la arquitectura de tres capas.

las flechas representan las interacciones remotas existentes en una aplicación típica. También pueden existir interacciones remotas entre las máquinas de un mismo tipo: los contenedores de *servlets* pueden interactuar para compartir datos de la sesión del usuario, por ejemplo, y los contenedores de EJB pueden interactuar para mantener actualizados los caché de datos.

Cuando aumenta la cantidad de usuarios que usan una aplicación es necesario aumentar la cantidad de máquinas en cada una de las capas para asegurar la calidad de respuesta. Este incremento en la cantidad de máquinas es conocido como *escalabilidad horizontal*, y provoca que se requieran aún más interacciones remotas para satisfacer un requerimiento del usuario.

En la siguiente sección, explicaremos las partes más importantes de un servidor de aplicaciones, concretamente describiremos el rol de los contenedores de componentes.

## 2.3. Contenedores de componentes

Los contenedores son ambientes de ejecución dentro del servidor de aplicaciones. Los contenedores son la interfaz entre los componentes y la funcionalidad de bajo nivel que da soporte al componente –componente Web o Enterprise Bean–. Un servidor de aplicaciones tiene dos contenedores: el *Contenedor de Servlets/JSP* y el *Contenedor de EJBs*. A través de estos contenedores el servidor de aplicaciones provee, de forma transparente para los desarrolladores, servicios de nivel de sistema como entrega de requerimientos HTTP a Servlets, demarcación de transacciones para EJBs, además de seguridad, concurrencia, y administración de ciclos de vida para los componentes que tiene instalados. Gráficamente, la Figura 2.3

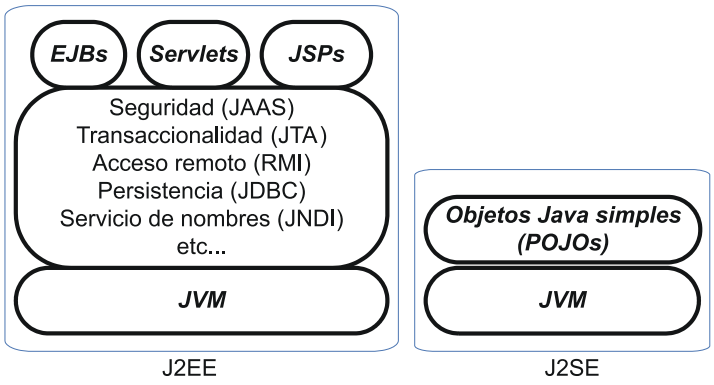


**Figura 2.3:** Comparación entre ambiente de ejecución de componentes J2EE y objetos Java.

representa la diferencia entre los componentes instalados en un contenedor J2EE, que utilizan muchos servicios que les son provistos por el servidor de aplicaciones, y las clases Java tradicionales (conocidas como POJO por "Plain Old Java Object"), que deben interactuar directamente con la máquina virtual (JVM por "Java Virtual Machine").

Uno de los servicios que brindan los contenedores son las búsquedas a través de la interfaz JNDI (Java Naming and Directory Interface). Esta interfaz a directorios JNDI provee a las aplicaciones métodos para realizar operaciones de directorios, como asociar atributos a objetos o buscar objetos por sus atributos. Usando JNDI, las aplicaciones J2EE pueden almacenar y recuperar cualquier tipo de objeto java por nombre. Los servicios de nombre J2EE proveen de un *ambiente de nombres* a los clientes, beans, y componentes Web de la aplicación. Usando un *ambiente de nombres* los componentes se pueden configurar sin necesidad de recompilar o siquiera tener acceso al código fuente del componente. Es el contenedor Web el que implementa este ambiente y lo hace accesible a los componentes. Los componentes J2EE pueden utilizar el servicio de nombres para acceder a objetos provistos por el sistema o definidos por el usuario.

Otro servicio provisto por los contenedores es el de conectividad remota, que determina cómo se efectúan a bajo nivel las comunicaciones entre EJBs y sus clientes. Después que un EJB es creado, es accedido por el cliente como si estuvieran en la misma máquina virtual.

El contenedor además brinda acceso varias varias interfaces de programación de aplicaciones (APIs), para que los componentes Web puedan acceder a directorios, transacciones y email. Una de esas interfaces es JDBC, que permite la ejecución de comandos SQL de forma estándar e independiente del motor de base de datos al que se esté accediendo. JDBC se utiliza para acceder a la base de datos desde componentes de negocio que quieren acceder a los datos persistidos directamente, sin utilizar la simpleza de la persistencia manejada por el contenedor.

Para poder ejecutar un componente Web o enterprise bean, éste debe estar ensamblado en un módulo J2EE e instalado en el contenedor. Un módulo J2EE consiste en uno o más componentes J2EE empaquetados en un archivo junto a un descriptor de despliegue, para ser instalados en un mismo contenedor. Un módulo EJB consta de uno o más enterprise beans, y se empaqueta en un fichero de extensión "jar". Un módulo Web puede contener *servlets* y páginas JSP, y se empaqueta como un fichero de extensión "war". El proceso de ensamblado

consiste en configurar cada componente para que el contenedor sepa administrarlo, además de configurar al servidor J2EE. En el siguiente listado se puede ver la declaración de un enterprise bean de sesión con estado "EJBController" cuyo método "processEvent(Event)" debe ser ejecutado dentro de una transacción:

```
<enterprise-beans>
    <session>
        <ejb-name>EJBController</ejb-name>
        <ejb-class>com.sun.j2ee.EJBController</ejb-class>
        <session-type>Stateful</session-type>
        <transaction-type>Container</transaction-type>
    </session>
</enterprise-beans>

<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>EJBController</ejb-name>
            <method-name>processEvent</method-name>
            <method-params>
                <method-param>com.sun.j2ee.Event</method-param>
            </method-params>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

Hemos mencionado a los componentes que se instalan en los contenedores de aplicaciones. En la siguiente sección veremos en mayor detalle cuáles son estos componentes.

## 2.4. Componentes J2EE

Las aplicaciones J2EE son conjuntos de componentes interrelacionados. Un componente J2EE es una unidad de software que cumple una función y es instalada en un servidor de aplicaciones con sus clases relacionadas y archivos, donde interactúa con otros componentes. La especificación J2EE define los siguientes componentes:

- Clientes Web, *applets* y otras aplicaciones Java son los componentes que se ejecutan en la capa cliente.
- Los componentes Web se ejecutan en el servidor.
- Los EJBs se ejecutan en el servidor.

Los componentes J2EE se escriben en el lenguaje de programación Java y son compilados de la misma manera que cualquier otro programa escrito en el mismo lenguaje. La diferencia entre componentes J2EE y clases java tradicionales es que los componentes J2EE son ensamblados para formar una aplicación, se verifica que respete las imposiciones de la especificación J2EE, y son instalados, ejecutados y administrados por un servidor J2EE. Como vimos en la

Figura 2.3, los contenedores les brindan a los componentes J2EE muchos servicios que facilitan su desarrollo. Para un programador, incorporar a los POJO las características que los EJB ya poseen tendría un costo demasiado elevado.

En las siguientes subsecciones se describen los diferentes tipos de componentes.

### 2.4.1. Clientes Web

Los clientes Web consisten de dos partes:

- páginas dinámicas conteniendo diferentes tipos de lenguajes de tags, como HTML o XML, que son generados por componentes Web ejecutándose en la capa Web.
- navegadores de Internet que dibujan las páginas recibidas desde el servidor.

Los clientes Web son también llamados clientes livianos. Los clientes livianos en general no acceden a bases de datos, ni ejecutan lógica de negocio compleja, ni se conectan a sistemas legados. Cuando se utiliza clientes livianos, esas tareas pesadas son delegadas a los enterprise beans que se ejecutan en el servidor J2EE, aprovechando así la seguridad, velocidad, confiabilidad y servicios de las tecnologías J2EE del lado del servidor.

### 2.4.2. Componentes Web

Los componentes J2EE de la capa Web son *servlets* o páginas creadas utilizando la tecnología JSP. Los *servlets* son clases programadas en el lenguaje Java que procesan requerimientos y generan respuestas dinámicamente. La tecnología JavaServer Pages (JSP) permite incluir fragmentos de código java directamente en documentos basados en texto. Una página JSP es un documento basado en texto que contiene dos tipos de texto: datos estáticos (que puede ser cualquier formato como HTML, XML, etc) y elementos JSP, que determinan cómo se construye el contenido dinámico de la página.

Páginas HTML y *applets* son empaquetados con los componentes Web para ser instalados en un contenedor J2EE pero no son considerados componentes Web porque no son parte de la especificación J2EE. Las clases utilitarias que se usan en el servidor también pueden ser empaquetados con componentes Web pero tampoco son considerados componentes Web.

### 2.4.3. Enterprise Beans

La responsabilidad de los enterprise beans es recibir datos desde programas cliente, procesarlos y enviarlos a bases de datos o sistemas legados. De la misma manera, los enterprise beans recuperan información de la misma capa de datos y los envían de regreso al cliente.

Dado que los Enterprise Beans son los componentes centrales de la arquitectura J2EE la siguiente sección profundiza en el tema.

## 2.5. Enterprise Beans

Los Enterprise Beans son los componentes J2EE que implementan la tecnología Enterprise JavaBeans. Los Enterprise Beans se ejecutan en un contenedor de EJBs y son los componentes del lado del servidor que encapsulan la lógica que resuelve las necesidades de un dominio de negocio particular, como pueden ser el negocio bancario, comercial o financiero.

Los enterprise beans simplifican el desarrollo de aplicaciones grandes y distribuidas. Esto se debe a varios motivos, de los cuales se destacan los que veremos a continuación:

- gracias a que el contenedor de EJB provee los servicios de bajo nivel a los enterprise beans, el desarrollador puede concentrarse en resolver los problemas del negocio. El contenedor, no el desarrollador, es el responsable por los servicios de nivel de sistema, como administración de transacciones y seguridad.
- como la lógica de negocios se encuentra en los enterprise beans y no el cliente, el desarrollador a cargo de implementar el cliente de la aplicación puede concentrarse sólo en la presentación. No debe preocuparse por implementar rutinas de acceso a bases de datos o reglas del negocio. Como resultado los clientes requieren pocos recursos de procesamiento y/o almacenamiento lo cual es especialmente beneficioso para dispositivos móviles.
- como los EJBs son componentes portables, se pueden armar aplicaciones ensamblando beans existentes y ejecutarlas en cualquier servidor J2EE, siempre y cuando se hayan respetados las convenciones que impone esta plataforma.

Los enterprise beans son especialmente útiles si la aplicación contiene alguno de los siguientes requerimientos:

- la aplicación debe ser escalable para poder servir a un número creciente de usuarios. Puede ser necesario que los componentes de la aplicación deban distribuirse entre múltiples servidores. Los enterprise beans pueden ejecutarse en diferentes máquinas, y su ubicación se mantiene transparente a los clientes.
- la integridad de los datos debe estar asegurada por transacciones. Los enterprise beans pueden trabajar transacciones distribuidas, el mecanismo que permite el acceso simultáneo a objetos compartidos.
- la aplicación debe tener diferentes tipos de clientes. Con unas pocas líneas de código, los clientes remotos pueden acceder fácilmente a los enterprise beans. Los clientes pueden ser delgados, de diferente tipo, y muy numerosos.

Hay tres tipos de enterprise beans: beans de entidad, beans controlados por mensajes, y beans de sesión.

Los beans de entidad representan los datos persistentes, que pertenecen a los objetos de negocio de la aplicación. Ejemplos de objetos de negocio podrían ser *Cliente*, *OrdenDeCompra* y *Producto*. Tradicionalmente, en los servidores de aplicación se utiliza una base de datos

relacional como mecanismo de persistencia de estos beans. En estos casos, típicamente cada bean de entidad tiene una tabla asociada en la base de datos, y las instancias del bean tienen una correspondencia uno a uno con los registros de esa tabla.

Los *beans controlados por mensajes* combinan características de los beans de sesión con la tecnología Java Message Service (JMS), permitiendo que un componente de negocio reciba mensajes en forma asíncrona. Los beans controlados por mensajes se comportan como receptores de mensajes JMS, que pueden ser enviados por cualquier componente J2EE o de otra tecnología que utilice JMS.

Los beans de sesión representan conversaciones transitorias con los clientes de la aplicación dentro del servidor de aplicaciones. Otra ventaja de este tipo de beans, es que es el único tipo con que el desarrollador puede implementar *Servicios Web* [42]. Los Servicios Web son una tecnología, basadas en protocolos y lenguajes de la Web, para desarrollar aplicaciones distribuidas que permite alcanzar excelentes niveles de interoperabilidad [34]. Dado que los beans de sesión son utilizados directamente por los clientes, son particularmente importantes y los veremos con más detalle en la siguiente subsección.

### 2.5.1. Beans de sesión

Como su nombre lo indica, un bean de sesión es semejante a una sesión interactiva. Un bean de sesión no es compartido entre clientes, cada bean puede tener un solo cliente, de la misma manera que una sesión interactiva puede tener un solo usuario.

Para acceder a una aplicación que está instalada en el servidor, los clientes invocan métodos en los beans de sesión. Los beans de sesión por su parte ejecutan el trabajo para el cliente, abstrayendo al mismo de la complejidad de iniciar procesos de negocio directamente dentro del servidor de aplicaciones. Los beans de sesión no son persistentes, terminan cuando el cliente se desconecta y ya no pueden ser asociados al mismo. Cuando el cliente termina de ejecutarse, el bean de sesión y sus datos se pierden. Hay dos tipos de beans de sesión, sin estado y con estado.

#### 2.5.1.1. Beans de sesión sin estado

Los beans de sesión sin estado no mantienen un estado de la conversación con el cliente. Cuando un cliente invoca a un método de un bean de sesión sin estado, sus variables de instancia pueden tener estado, pero sólo por la duración de la invocación. Cuando el método finaliza, el estado se pierde. Exceptuando el momento en que los beans de sesión sin estado están ejecutando un método, son todos iguales, por lo que el contenedor de EJB puede asignar cualquier bean a cualquier cliente.

Como los beans de sesión sin estado pueden atender a múltiples clientes, ofrecen gran escalabilidad para las aplicaciones que deben soportar una gran cantidad de clientes. Típicamente, una aplicación requiere menos beans de sesión "sin estado" que "con estado" para soportar la misma cantidad de clientes.

A veces el contenedor de EJB puede guardar un bean con estado en almacenamiento secundario, lo que nunca sucede con los beans que no tienen estado. Esto también es motivo para que el desempeño de los beans sin estado sea superior.

#### 2.5.1.2. Beans de sesión con estado

El estado de un objeto consiste en los valores de sus variables de instancia. En un bean de sesión con estado las variables de instancia representan el estado de una conversación entre un cliente y el bean que lo atiende, comúnmente conocido como *estado conversacional*.

El estado se mantiene durante toda la sesión del cliente. Si el cliente elimina al bean o termina su sesión, el estado del bean desaparece. La naturaleza transitoria del estado de la sesión no es un problema, porque cuando la conversación con el cliente finaliza no hay necesidad de mantener el estado.

## 2.6. Construcción de sistemas J2EE

Hemos visto a lo largo del capítulo las características de la arquitectura J2EE y de qué manera simplifican el desarrollo de aplicaciones distribuidas de nivel empresarial. Sin embargo, la complejidad inherente a los sistemas distribuidos sigue estando presente en las aplicaciones construidas con esta tecnología.

Para construir uno de estos sistemas en forma efectiva se deben aplicar buenas prácticas de diseño, y se debe conocer cómo interactúan los componentes entre sí y con sus contenedores, tanto al momento de diseñar la aplicación como al programar el código que la implementa. Al simplificar el desarrollo, la arquitectura J2EE permite que programadores que quizás no tienen esos conocimientos puedan desarrollar complejos sistemas distribuidos, algo que antes estaba reservado sólo para expertos.

Una consecuencia de esto es que se construyen aplicaciones que no respetan las mejores prácticas de diseño de aplicaciones distribuidas, lo que puede originar varios problemas. Uno de estos problemas particularmente notorio y recurrente es la baja *performance* que tienen las aplicaciones cuando son instaladas en producción y deben ser utilizadas por los usuarios reales [33], que son ocasionados en gran cantidad de casos por el mal uso de la tecnología J2EE.

En el siguiente capítulo veremos diferentes trabajos que analizan el tema y proponen diferentes formas de evitar o solucionar los problemas de desempeño.

# CAPÍTULO 3

# Trabajos Relacionados

El análisis de los problemas de desempeño de los sistemas de software ha sido materia de estudio e investigación desde los orígenes de la ingeniería de software. Diversas metodologías proponen analizar sistemáticamente el diseño de una aplicación, previo a su materialización, con el objetivo de que se garantice que la implementación resultante cumpla con los requerimientos de *performance*. Otros enfoques proponen dividir las aplicaciones en componentes individuales, pero interconectados, y vigilar el desempeño de estos durante su ejecución para identificar componentes con problemas de *performance* y tratar de mejorarlos. La herramienta presentada en este trabajo plantea una enfoque novedoso para encarar la identificación y solución a los problemas de mal desempeño de las aplicaciones J2EE. Por lo que respecta a nuestro conocimiento, no existen muchos antecedentes que intenten solucionar este problema relacionado con J2EE directamente. En el resto de este capítulo se discuten algunas de estas alternativas, que este trabajo pretende unificar en un nuevo tipo de herramientas automáticas.

## 3.1. Software Performance Antipatterns

Para asistir en el proceso de optimizar una aplicación, se han identificado algunos errores comunes de diseño que crean problemas de *performance,* afectando el desempeño de las aplicaciones, y se han documentado como "*Software Performance Antipatterns*".

Conceptualmente, este trabajo está basado en patrones de diseño, que tienen su origen en el catálogo publicado en el libro *Design Patterns* [13]. Los patrones son soluciones reusables y probadas a problemas de diseño frecuentes, documentadas en forma estructurada. En cambio, cada antipatrón describe una situación recurrente que tiene consecuencias negativas en la *performance* de la aplicación, y explica cómo puede solucionarse el problema una vez que se encuentra [5]. Algunas veces, la solución propuesta consiste en aplicar un patrón de diseño ya documentado.

Hay varios antipatrones documentados, entre los que se encuentran los que figuran en el

**Cuadro 3.1:** Algunos Antipatrones.

| Nombre | Situación | Solución |
|---|---|---|
| Camiones Semivacíos | Ocurre cuando se hace una excesiva cantidad de invocaciones remotas para realizar un procesamiento. | Agrupar los pequeños mensajes en uno más grande. |
| Torre de Babel | Ocurre cuando internamente se utilizan diferentes formas de representar los datos, requiriendo múltiples conversiones. | Utilizar la misma representación de los datos en los procesos del camino crítico. |
| Más es Menos | Ocurre cuando hay demasiados procesos ejecutándose, y el sistema debe perder más tiempo administrando los recursos que ejecutando la aplicación. | Mantener acotada la cantidad de procesos o hilos que se crean por debajo del nivel soportado por el sistema. |
| Búsqueda del Tesoro | Ocurre cuando se utiliza el resultado de una invocación para realizar otra invocación, y así sucesivamente. | Proveer caminos de acceso más directos a la información que se requiere. |

Cuadro 3.1. Documentar estas situaciones problemáticas es muy positivo porque permite a los desarrolladores estudiarlas antes de diseñar y construir los sistemas, con lo que pueden evitar la aparición de los problemas.

Pero en el caso de que el sistema ya esté construido y sufra de algunos de estos problemas, aplicar las soluciones propuestas no es una tarea sencilla. La principal dificultad que se presenta es que identificar la ocurrencia de alguno de ellos es una tarea manual y requiere un gran entendimiento del sistema. Se pueden utilizar herramientas de monitoreo, pero en ese caso es necesario que un especialista interprete la información recolectada. Incluso si se llega a identificar la ocurrencia de un antipatrón, por ejemplo detectando la transmisión de muchos paquetes pequeños por la red, la tarea de encontrar el código que lo está provocando debe hacerse manualmente. Por estas razones, una de las principales contribuciones de este trabajo final es la identificación automática de antipatrones.

Quizás el aporte más valioso que realiza este catálogo de antipatrones de *performance* es la asociación entre situaciones problemáticas y su solución. Esto permite que en los casos en que se logra identificar la presencia de la situación descrita por el antipatrón, la solución ya está documentada. Esto último, puede ser utilizado a modo de guía para mejorar el desempeño del sistema. En este sentido, otra de las contribuciones de este trabajo es utilizar estas guías para erradicar un antipatrón automáticamente.

## 3.2. Enterprise Java Performance: Intel Best Practices

En [9] se propone una metodología iterativa para solucionar los problemas de *performance* que puedan surgir en aplicaciones J2EE. Los pasos del método iterativo descripto son los siguientes:

1. Recolectar información del sistema en ejecución, utilizando herramientas de monitoreo y simulando condiciones de carga (por ejemplo con múltiples usuarios de la aplicación).
2. Analizar la información obtenida e identificar el cuello de botella que limita la *performance* de la aplicación.
3. Buscar diferentes alternativas que puedan eliminar el cuello de botella identificado, y seleccionar una.
4. Implementar la solución seleccionada. Es importante que se introduzca sólo una mejora para poder discriminar el impacto de la misma.
5. Pruebas y evaluación del efecto de la mejora implementada, para comprobar que elimina el cuello de botella.

Vale aclarar que aquí la palabra sistema se utilizó para referirnos, indistintamente, a las distintas capas de un sistema J2EE, a saber aplicación, máquina virtual y plataforma. Una vez que se eliminó el cuello de botella, se debe repetir el proceso identificando nuevos cuellos de botella. Idealmente, se itera hasta que la información recolectada en el primer paso indique que el sistema satisface los requerimientos de *performance*.

Este proceso de optimización debe realizarse en tres capas del sistema, en el siguiente orden:

- *Nivel de sistema:* El objetivo al iterar sobre este nivel es asegurarse que el cuello de botella está en la aplicación y no en el sistema (configuración de discos, red, base de datos), ya que en ese caso grandes reestructuraciones de código producirían mejoras mínimas.
- *Nivel de aplicación:* En esta etapa se debe refactorizar la aplicación incorporando buenas prácticas de programación y patrones de diseño, y se deben modificar los parámetros configurables del servidor de aplicaciones para ajustarlos a las características del sistema.
- *Nivel de máquina:* En esta etapa se debe optimizar el uso de la memoria y del procesador, para lo que se puede configurar la máquina virtual de Java y el caché del procesador, por ejemplo.

Esta metodología es muy rigurosa y permite resolver los problemas en cualquiera de los tres niveles planteados. Aunque las herramientas de *profiling* en que se basa esta metodología pueden ayudar a identificar el cuello de botella, encontrar la causa del problema y la solución al mismo sigue siendo un proceso manual que requiere de expertos en la tecnología.

Los niveles de sistema y de máquina, han sido ampliamente estudiados y existe bastante conocimiento disponible [27, 7]. La principal dificultad que se presenta al querer aplicar esta metodología entonces está en encontrar los problemas que pueda haber en el nivel de aplicación, porque requiere de especialistas en J2EE, que es una tecnología bastante reciente, y de especialistas en la aplicación a analizar.

La herramienta propuesta en este trabajo no depende de una metodología en particular. De hecho, puede ser utilizada para automatizar los primeros tres pasos del nivel de aplicación de la metodología presentada en esta sección, disminuyendo la dificultad de aplicar esta metodología.

## 3.3. PASA: A Method for the Performance Assessment of Software Architectures

Lloyd Williams y Connie Smith proponen un método para la revisión de arquitecturas de software [43]. Este proceso, llamado PASA, consiste en 9 pasos conducidos por un equipo de expertos en *performance*, que pueden ser resumidos de la siguiente manera:

1. Presentación del proceso.
2. Los desarrolladores presentan la arquitectura existente.
3. Identificación de casos de uso críticos en cuanto a *performance*.
4. Selección de escenarios de los casos de uso críticos.
5. Identificación de los objetivos de *performance*.
6. Discusión detallada sobre la implementación de los casos de uso críticos, para que el grupo de expertos entienda la arquitectura que da soporte a los escenarios escogidos.
7. El grupo de expertos analiza la información recolectada buscando problemas en la arquitectura que puedan afectar la *performance*.
8. El equipo de expertos presenta soluciones a los problemas encontrados en el paso anterior.
9. Fin del proceso. El equipo de expertos entrega un documento con los problemas encontrados y las correcciones que los desarrolladores deben hacer.

Una ventaja de este enfoque respecto a los otros analizados en este trabajo consiste en que puede ser utilizado antes de la construcción del sistema, lo que permite evitar los problemas antes de que ocurran evitando grandes pérdidas. También es importante destacar que este proceso puede ser aplicado a cualquier tecnología, mientras se dispongan de los expertos en la misma.

Una dificultad que presenta este enfoque es la necesidad de identificar los casos de uso críticos en cuanto a *performance* y armar los escenarios correspondientes. Esto no es sencillo

y está la posibilidad de que no se analicen todas las porciones de la arquitectura que afectan a la *performance*.

El problema principal de este método es que requiere de un equipo de expertos en *performance* y arquitecturas de software que analice el sistema, que no siempre están disponibles y pueden representar un costo alto para el proyecto. También requiere que el sistema esté bien documentado, que si bien es deseable, no ocurre con frecuencia en la realidad.

Nuevamente, para minimizar las dificultades enumeradas, se pueden automatizar los pasos 7 y 8 (búsqueda de antipatrones y de las alternativas de refactorización que lo solucionan, entre otros) mediante el uso de la herramienta propuesta en esta tesis. El uso de esta herramienta hace innecesario el trabajo de identificar manualmente los aspectos críticos de la aplicación en cuanto a *performance*: al monitorear el comportamiento del sistema en tiempo de ejecución, los casos de uso típicos se analizan automáticamente mientras se usa la aplicación.

## 3.4. COMPAS

COMPAS [32] es un *framework* que supervisa la *performance* de aplicaciones J2EE durante su funcionamiento. En particular, supervisa las interacciones entre los componentes de *software* del servidor que conforman el sistema, que en esta tecnología reciben el nombre de *Enterprise Java Beans* (EJBs) [37]. Básicamente, se recolecta información sobre el tiempo de ejecución que demandan los métodos de un EJB. Para lograr esto, se agrega una capa de software a cada EJB (a.k.a. *proxy*) que intercepta la invocación de cada método y calcula el tiempo que un EJB demora desde que recibe la invocación hasta que produce una respuesta. El *framework* presenta una estrategia de monitoreo que se va adaptando en base a los últimos resultados obtenidos y sólo agrega *proxies* a los EJBs que son invocados en las operaciones que tienen un mayor tiempo de respuesta.

Adicionalmente, COMPAS permite visualizar la información recolectada. Una consola gráfica lista los EJBs que están siendo intervenidos con un *proxy* y los métodos que han sido invocados sobre los mismos, especificando la cantidad de veces que se han ejecutado y el tiempo promedio que se demora cada uno en finalizar. Esta consola actualiza la información conforme la ejecución del sistema.También, permite definir reglas para filtrar la información, y mostrar sólo los métodos que se demoran más de un tiempo configurado.

Viendo la consola de esta aplicación, un desarrollador puede determinar si hay problemas de *performance* y, en caso de que los haya, cuáles son los métodos que están demorando demasiado. Con esa información, el desarrollador debe encontrar el problema y buscar una solución apropiada.

Aunque esta herramienta es útil para encontrar la porción de código con problemas de *performance*, no puede indicar cuál es el problema ni cómo solucionarlo. En el caso de que el problema sea originado por un consumo excesivo de los recursos, como puede ocurrir por ejemplo si hay demasiado tráfico en la red y se consume el ancho de banda, la mayoría de los métodos van a demorarse más de lo aceptable, no pudiendo identificarse en ese caso al responsable original.

## 3.5. Performance Prediction of COTS Component-based Enterprise Applications

El objetivo de este trabajo es predecir la *performance* que tendrá un sistema antes de construirlo. Para lograrlo propone construir modelos de *performance* que describan las características de la infraestructura sobre la que funcionarán, que pueden ser contenedores J2EE o de otras tecnologías.

Se limita el alcance del trabajo a las aplicaciones típicas de 4 capas:

1. Clientes ejecutándose en un navegador de Internet.
2. Un servidor de contenidos Web ejecutando la lógica de presentación.
3. Un contenedor de aplicaciones ejecutando la lógica de negocio.
4. Bases de datos o aplicaciones legadas proveyendo datos.

Las capas 2 y 3 componen el *middleware*, el soporte de infraestructura que proporciona a los componentes de negocio de las funcionalidades trasversales como seguridad, transaccionalidad, y distribución. Los componentes de aplicación se ejecutan sobre los mismos, por lo que en tiempo de ejecución están fuertemente acoplados, haciendo que el desempeño final del sistema dependa tanto del rendimiento de los componentes de aplicación como de los componentes de infraestructura.

Teniendo este grupo de aplicaciones en mente, se debe seguir la siguiente metodología organizada en 2 etapas:

1. Se crea un perfil específico al *middleware* sobre el que se va a ejecutar la aplicación, que describe la *performance* de los diferentes componentes de infraestructura independientemente de los requerimientos que pueda tener la aplicación. Para lograrlo, se debe ejecutar en el mismo ambiente una aplicación trivial: con un solo EJB y una sola tabla de dos columnas. Haciendo mediciones al ejecutar esa aplicación con múltiples configuraciones, se puede establecer el costo en *performance* asociado a cada uno de los parámetros del servidor, en diferentes condiciones de carga. Los parámetros que se pueden variar son cantidad de clientes, tamaño del *pool* de conexiones, cantidad de *threads*, política de expiración del caché, etc. El objetivo de este perfil es analizar el comportamiento de la infraestructura del servidor bajo distintas configuraciones, independientemente de las aplicaciones que se vayan a instalar sobre el mismo.
2. Se crea un perfil con las características de la aplicación que se desea evaluar. Se debe especificar cantidad de clientes, complejidad de la lógica del negocio, requerimientos de la base de datos, y caracterizar el uso de los servicios de infraestructura que se necesitarán. Este perfil, analizado con un modelo matemático, permite predecir el valor que deben asignarse a los parámetros de configuración (tamaño del *pool* de *threads* y del de conexiones a la base de datos, por ejemplo) para obtener la mejor *performance* posible.

El enfoque de este trabajo es diferente al de los demás. No pretende encontrar los problemas que pueda haber, sino que busca predecir cuál será la *performance* de una aplicación antes de construirla.

Una ventaja que presenta este método respecto de los otros analizados en este capítulo es que indica cuáles son los valores óptimos para algunos parámetros de configuración del sistema. Si una aplicación no responde adecuadamente por culpa de algún parámetro mal configurado, esta herramienta permite corregir fácilmente el mismo.

Este método también puede ser utilizado para determinar si hay problemas en el diseño de una aplicación: una vez construida se puede comparar el desempeño real con el esperado y, en caso de que no cumpla con las expectativas, se puede concluir que hay posibilidad de corregir partes del sistema para aumentar su capacidad. Lo que no indica de ninguna manera es cuáles pueden ser los problemas, dónde están, o cómo corregirlos. Para responder a esas preguntas se puede utilizar una herramienta como la propuesta en este trabajo, que se complementa muy bien con este método porque buscan solucionar diferentes tipos de problemas.

## 3.6. Performance Techniques for COTS Systems

Putrycz y sus colaboradores [35] proponen solucionar los problemas de *performance* de los sistemas de componentes integrados valiéndose de modelos y mediciones.

Para poder aplicar esta técnica es necesario dedicar tiempo a las pruebas de *performance* en las primera etapas del proyecto de construcción de software, lo que no es muy fácil de lograr en las condiciones actuales en que se requiere que se empiece la producción cuanto antes, y los requerimientos pueden cambiar en etapas posteriores. [3]

El modelado del sistema se hace con una versión adaptada de UML2, que incorpora notación para dar soporte a los conceptos de *performance* requeridos para el análisis. La información que se precisa incluye:

- el tiempo de procesador que requiere cada uno de los servicios que ofrece cada componente
- la cantidad de interacciones con los distintos servicios que realizan los componentes para resolver cada operación
- recursos lógicos que utiliza. Por ejemplo: *threads*, *buffers* y *caches*.

Con esta información alimentando el modelo, se pueden aplicar diferentes técnicas para la predicción de la *performance* que tendrá el sistema en diferentes configuraciones de componentes, para poder determinar cuáles componentes brindarán un mejor comportamiento al sistema.

Una vez que el sistema está construido, se supervisa y se toman nuevas mediciones, que son utilizadas para verificar que la *performance* sea satisfactoria, además de identificar problemas y recalibrar el modelo. El análisis de la información recolectada al monitorear el sistema

puede ser utilizado para hacer correcciones al diseño en base al comportamiento de la aplicación. El modelo actualizado puede dar información que sugiera la necesidad de cambios más radicales.

Esta técnica de modelado tiene utilidad en diferentes etapas del desarrollo.

1. Al comienzo del proyecto, se puede armar el modelo que representa al sistema deseado utilizando conocimiento adquirido en sistemas similares. Si las predicciones de *performance* son insatisfactorias, se puede rediseñar el sistema y reconstruir el modelo. Hacer esto sobre modelos tiene un costo muy inferior al que de hacer modificaciones al diseño después que ya comenzó la construcción del sistema.

2. Durante las etapas de construcción, se puede medir la *performance* de cada componente en forma aislada, e incorporar esa información al modelo para determinar el impacto que tendrán los componentes en el desempeño general del sistema, antes de comenzar la integración. De esta manera se evita la difícil tarea de detectar cuál es el componente que provoca el bajo rendimiento del sistema.

3. En las etapas finales del proyecto, el modelo puede ser utilizado para analizar la escalabilidad del sistema más allá de los límites que haya en el laboratorio de pruebas.

La cantidad y detalle de la información que se precisa sobre la implementación de los componentes dificulta la aplicación de este método. Esta información puede obtenerse de la documentación y las especificaciones técnicas de cada componente que se incorpora al sistema, o en su defecto, debe caracterizarse a través de un proceso de ingeniería reversa. Sin embargo, en las aplicaciones empresariales se suelen integrar cientos de componentes, y no todos tienen disponible su código fuente o especificaciones técnicas detalladas.

## 3.7. Performance Prediction of J2EE Applications using PEPA nets

En [36] se propone predecir la *performance* que tendrá una aplicación J2EE utilizando modelos basados en las redes conocidas como PEPA. Las redes PEPA son redes de Petri estocásticas, un modelo formal de alto nivel para el análisis del desempeño de objetos distribuidos [14]. En [3] se presenta una recopilación de enfoques para predecir la *performance* de un sistema de software basados en modelos similares a las redes PEPA.

Para facilidad del modelado, se divide el sistema a analizar en dos etapas de acuerdo a la interacción entre componentes distribuidos. En la primera etapa se incluye la transmisión de los requerimientos desde la aplicación cliente hasta el servidor. La segunda etapa consiste en el procesamiento que se realiza en el servidor, lo que incluye el acceso a la bases de datos.

El método de optimización propuesto por los investigadores consiste en tres pasos:

1. Se modela la aplicación J2EE teniendo en cuenta los requerimientos funcionales y no funcionales en las dos etapas identificadas. Por cada interacción entre componentes, se debe modelar la transición incluyendo el nodo origen, el nodo destino, la actividad que se ejecuta en la transición y a qué velocidad. Estas velocidades con que se ejecutan las diferentes actividades de transición se factorizan y quedan como parámetros a calcular.

2. Se asignan valores a los parámetros identificados durante el modelado. Los valores de estos parámetros pueden obtenerse consultando a un desarrollador que haya tenido experiencia en proyectos semejantes, pero se recomienda obtenerlos mediante el monitoreo de prototipos o aplicaciones basadas en la misma arquitectura.

3. Se predice la *performance* que tendrá el sistema. Utilizando el modelo se obtiene la demanda que tendrán los diferentes recursos, y en base a esa demanda se pueden obtener los costos asociados y determinar los índices de *performance* asociados a cada interacción del cliente. Esta fase de análisis está automatizada utilizando las herramientas existentes para trabajar con redes PEPA.

La idea de los investigadores es que se midan los parámetros de las arquitecturas estándar y se puedan tener precalculados estos factores, para poder omitir el segundo paso del proceso y obtener los resultados inmediatamente después de construir el modelo.

La principal dificultad que se presenta al querer implementar este método de predecir la *performance* de las aplicaciones a construir, consiste en que se requiere que se modele la aplicación con un alto nivel de detalle antes de empezar a construirla. Esto es complicado de lograr porque requiere una inversión de tiempo importante para hacer esta estimación, y la misma pierde valor si se modifica el diseño, que suele ocurrir varias veces durante el ciclo de vida de los sistemas como consecuencia de cambios en los requerimientos.

Analizando este modelo de redes se puede determinar cuáles son los nodos críticos y predecir el tiempo de respuesta que tendrán las operaciones del usuario, pero si se determina que los mismos no son satisfactorios, es responsabilidad del desarrollador identificar el problema y modificar el diseño del sistema hasta que cumpla con las expectativas de desempeño.

## 3.8. JPManager: A Performance Validation Tool for J2EE Applications

JPManager [15] es un sistema distribuido especialmente concebido para la administración de *performance* de aplicaciones J2EE. Para lograr su objetivo esta herramienta modifica en tiempo de ejecución las clases de la aplicación, utilizando un técnica conocida como *bytecode instrumentation* que consiste en agregar código para que se ejecute antes y después de la ejecución de cada método. El código insertado tiene la responsabilidad de recolectar información, que será analizada por otro componente de JPManager.

El usuario de esta herramienta debe determinar:

- qué entidades se supervisarán,
- en qué parte del código,
- y qué información se recolectará de ese código supervisado.

Por lo general, la funcionalidad necesaria para realizar el monitoreo en tiempo de ejecución, agrega una carga extra a la aplicación. Para minimizar el *overhead* introducido, el enfoque

de JPManager es de monitorear pequeñas partes del código de una aplicación por separado. El objetivo de estrategia de selección detallada es que JPManager pueda ser utilizado en sistemas productivos perjudicando el desempeño general del mismo lo menos posible. Comparativamente, el caso contrario es el de COMPAS, que propone monitorear todos los EJBs de una aplicación. El principal problema de este enfoque es que, generalmente, determinar cuál porción de código se debe monitorear es muy difícil. Además, este enfoque puede no detectar problemas de *performance* que no afecten a los componentes observados. Si el problema afecta directamente a alguno de los objetos supervisados, el programa alertará de la disminución en la velocidad de respuesta, pero no podrá dar orientación que indique la causa del mal desempeño.

Esta herramienta tiene una interfaz gráfica que muestra la información recolectada, en forma de lista y como diagrama de secuencia utilizando UML.

Al igual que las otras herramientas de monitoreo que hemos analizado, JPManager no puede ayudar a resolver los problemas, sino que se limita a evidenciar la presencia de los mismos. Una vez que se determina que la aplicación tiene una velocidad de respuesta lenta, es necesaria una herramienta como la propuesta en esta tesis para ayudar a corregir el problema.

## 3.9. SoftArch/Thin

SoftArch/Thin es un entorno para el modelado a alto nivel de arquitecturas de aplicaciones con cliente liviano. El ambiente posee integrado un generador de código que pretende ser un modelo ejecutable de la arquitectura definida.

Los arquitectos pueden, antes de construir el sistema, realizar un diseño a alto nivel de la aplicación con esta herramienta, incluyendo abstracciones para los clientes, servidores Web, contenedores de aplicaciones, y base de datos. Con esta información se genera automáticamente código J2EE o ASP.NET que representa el modelo esquematizado, junto con *scripts* de carga para simular la ejecución de la aplicación desde una herramienta de evaluación de *performance* (específicamente Microsoft Application Centre Test). El código generado es ejecutado midiendo su *performance*, y los resultados presentados para el análisis de la arquitectura y tecnología seleccionadas.

El objetivo de este entorno es que los arquitectos puedan modelar rápidamente potenciales arquitecturas y obtener buenas estimaciones de la *performance* que tendrán las mismas. Esta herramienta puede ser utilizada iterativamente a lo largo del proceso de desarrollo, desde los simples esquemas iniciales a los diseños detallados que modelan la complejidad real del sistema final. Mientras más información contenga el modelo, menor será la diferencia en *performance* entre el sistema modelado y el sistema real.

El uso de SoftArch/Thin requiere la ejecución de los siguientes pasos:

1. El arquitecto debe modelar inicialmente la arquitectura candidata utilizando el ambiente de modelado, para lo que se le proporciona representaciones de clientes, requerimientos, servidores, componentes, procesos, bases de datos, tablas, y varias relaciones y propiedades de las mismas.

2. La descripción de la arquitectura deseada es modelada gráficamente y guardada en formato XML, que luego es procesado por plantillas XSLT para generar el código fuente JAVA, JSP, C# y ASP, además de archivos de configuración para la herramienta de *testing* y la base de datos.

3. La aplicación así generada se debe instalar en los servidores, y la herramienta de evaluación de *performance* debe ser ejecutada simulando múltiples clientes.

4. Se presentan los resultados de la *performance* para su evaluación, de manera que el arquitecto pueda hacer modificaciones al modelo del sistema para obtener una mejor *performance* si es necesario.

Los resultados obtenidos con este sistema deben ser considerados como una guía simplemente, ya que el código generado no tiene la misma complejidad que el real, y por lo tanto la *performance* de esta aplicación generada tiende a ser mejor que el de la aplicación final.

Una dificultad que se presenta al querer aplicar este método es la necesidad de invertir tiempo en etapas tempranas para modelar el sistema y hacer las pruebas de carga, algo que no es sencillo de hacer dadas las presiones por entregar software rápidamente.

La naturaleza de esta herramienta para predecir el comportamiento de la aplicación hace que no tenga utilidad si se intenta optimizar una aplicación ya construida. En ese caso se podría ejecutar las pruebas de carga sobre la aplicación real, sin necesidad de utilizar las clases generadas automáticamente.

## 3.10. Conclusiones

Los diferentes trabajos que intentan disminuir los problemas ocasionados por el desempeño insuficiente de estos sistemas se basan en el análisis de especialistas. Ya sea para que ejecuten el proceso de optimización propuesto, como sucede con el método PASA o Intel, o para interpretar la información generada por herramientas de monitoreo, como sucede al utilizar COMPAS o JPManager. Ese requerimiento es un limitante importante, ya que por ser una tecnología reciente no hay gran cantidad de expertos J2EE. Sería preferible que los problemas de *performance* pudiesen ser detectados por los mismos desarrolladores, que son los que deben implementar las mejoras y seguir con el mantenimiento.

Aún habiendo expertos y estando el proyecto en condiciones de contratarlos, utilizar gente ajena al mismo presenta otros problemas:

- La velocidad de corrección de los problemas de *performance* es menor, porque los especialistas en *performance* deben primero entender el sistema que se quiere optimizar, y en esa transmisión de información y en la posterior presentación de soluciones alternativas se pierde tiempo que podría haberse evitado si se hubiese confiado realizar el trabajo de optimización del sistema a los mismos desarrolladores.

- Otro problema surge al considerar la evolución del sistema. Teniendo fuera del equipo de trabajo el conocimiento necesario para que el sistema se comporte con la capacidad requerida, requiere volver a llamar al especialista cada vez que tareas de mantenimiento degraden el funcionamiento del mismo.

Respecto a las herramientas de predicción de *performance* que vimos, como el trabajo sobre redes PEPA o SoftArch/Thin, tienen su mayor utilidad en proyectos que aún no han empezado, pero pueden aportar poco a proyectos que ya están construidos. Incluso si con la ayuda de esas herramientas se elige un diseño que soporta la cantidad de usuarios esperada con un tiempo de respuesta aceptable, es necesario continuar con el modelado durante todo el ciclo de vida del software, ya que cualquier cambio que se introduzca en el sistema en la etapa de mantenimiento puede degradar el comportamiento del sistema.

Los antecedentes presentados en este capítulo pueden clasificarse en base a qué parte del proceso de optimización son aplicables, y si se puede ejecutar automáticamente, con asistencia de herramientas, o si debe ser aplicado manualmente. También se pueden clasificar de acuerdo al nivel en que pueden ser utilizados: para resolver problemas de diseño o de configuración del ambiente. Dicha clasificación puede verse en el Cuadro 3.2.

**Cuadro 3.2:** Características de trabajos relacionados.

| | Encontrar el problema | Proponer solución | Señalar código a refactorizar | Problemas de diseño | Problemas de configuración |
|---|---|---|---|---|---|
| Antipatterns | Manual | Automática | Manual | Sí | No |
| Intel Best Practices | Asistido | Manual | Manual | Sí | Sí |
| COMPAS | Asistido | Manual | Automática] | Sí | No |
| PASA | Manual | Manual | Manual | Sí | Sí |
| Prediction | Manual | Automática | Manual | No | Sí |
| Techniques for COTS | Asistido | Manual | Manual | Sí | No |
| Redes PEPA | Asistido | Manual | Manual | Sí | No |
| JPManager | Asistido | Manual | Manual | Sí | No |
| SoftArch/Thin | Manual | Manual | Manual | Sí | Sí |
| Este trabajo | Automática | Automática | Automática | Sí | No |

Ninguno de los trabajos presentados contempla la posibilidad de que se corrijan los problemas de *performance* sin la asistencia de especialistas. Este trabajo pretende cubrir esta parte del ciclo en forma totalmente automática, si bien limitado a los problemas de diseño conocidos. Los problemas de configuración del ambiente o errores de diseño no contemplados por la herramienta deben ser corregidos siguiendo las ideas de los trabajos analizados en este capítulo.

# CAPÍTULO 4

## Un sistema experto para la optimización de aplicaciones J2EE

Antes de que la tecnología J2EE apareciera en el mercado, construir sistemas distribuidos tenía una complejidad mayor. Los sistemas J2EE se basan en los *Enterprise JavaBeans*, que son los componentes de una aplicación J2EE que definen un modelo portable que se integra con un servidor J2EE. El servidor de aplicaciones le provee a los EJBs con servicios del sistema, como soporte de transacciones y acceso a un servicio Java Naming and Directory Interface (JNDI) de directorios. Una de las piezas más importantes de un servidor de aplicaciones, es el contenedor de EJBs. Este es el encargado de administrar las clases de EJBs y sus instancias. Cada EJB está compuesta de interfaces y clases creadas por el programador, y por clases generadas por el contenedor de EJBs a partir de las interfaces provistas. La especificación describe cómo un EJB interactúa con su contenedor y define una forma estándar para que estos componentes sean instalados en el mismo, que consiste en el *deployment descriptor*. Estos descriptores incluyen los nombres de las interfaces y clases que forman parte de cada EJB.

Dentro de las diferentes clases que forman parte del *framework* EJB, las principales son:

- *EJB Home*: tiene la responsabilidad de crear, borrar y encontrar instancias de una clase de EJB. Se compone de dos clases; la *Home Interface* define cuáles son los métodos provistos, y el *Home Object* los implementa.
- *EJBObject*: es generada por el contenedor implementando una *Remote Interface* que define los métodos existentes en la clase EJB. Es el objeto que usa el cliente para realizar llamadas remotas a una instancia de un EJB alojado en el servidor de aplicaciones. La instancia del EJB nunca es accedida directamente por el cliente, sino que cada vez que el cliente invoca un método lo hace sobre el *EJBObject*, y éste es quien delega a la instancia correspondiente, además de agregar la funcionalidad necesaria para su ejecución remota, transaccionalidad, etc.

Gracias a que el *framework* de EJBs permite realizar llamadas remotas con la misma sintaxis utilizada para las invocaciones locales, es frecuente que los programadores no consideren

los costos asociados a las mismas, por ejemplo serialización y transporte por la red, y den gran uso a este tipo de invocaciones. Un uso indebido del soporte para invocaciones remotas genera aplicaciones con diseños que tienen un desempeño deficiente. Prevenir la aparición de estos problemas requiere que se modele el sistema con un alto nivel de detalle antes de construirlo, requiriendo mucho esfuerzo adicional, y aún así ya vimos que estos modelos no se adaptan fácilmente a las modificaciones que sufra el sistema en su ciclo de vida. Generalmente, los problemas de desempeño no se previenen de forma adecuada y son detectados una vez que la aplicación ya está terminada, incluso a veces no se detectan hasta que son reportados por el usuario final. En estos casos, se debe encontrar la causa del mal desempeño en un sistema J2EE con múltiples componentes relacionados, donde es muy difícil identificarla.

Considerando los problemas de *performance* ocasionados por el uso indebido de invocaciones remotas y la dificultad que se presenta al querer optimizar sistemas J2EE, este trabajo introduce un enfoque para automatizar la detección de estos problemas y la incorporación de sus soluciones, esquematizado en la Figura 4.1.

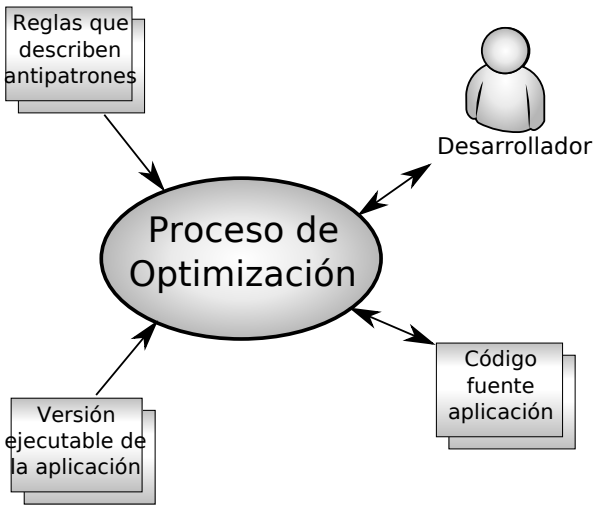


**Figura 4.1:** Esquema general del proceso propuesto.

A grandes rasgos, la idea central de este enfoque consiste en corregir automáticamente los diseños que provocan interacciones innecesarias entre componentes remotos. Para lograr esto, se propone un proceso iterativo y automatizado de 3 pasos. Estos pasos se listan a continuación y se detallan en el resto de la sección, siendo:

1. obtener información de la aplicación que se desea optimizar,
2. identificar problemas de diseño, y
3. aplicar soluciones.

Para los pasos 1 y 2 se analiza el código fuente de la aplicación y el desempeño en tiempo de ejecución de la misma. Para detectar problemas de diseño se analiza la información recolectada en busca de antipatrones de diseño conocidos y documentados [41]. Se debe buscar

dentro del sistema a optimizar las características de cada antipatrón, para poder erradicarlo. La forma de eliminar el problema de diseño suele estar documentado en la misma descripción del antipatrón, y generalmente consiste en aplicar modificaciones a la aplicación. Para aplicar las soluciones se debe modificar el código de la aplicación sin alterar su funcionalidad. Esta mejora es conocida como refactorización [5], y ya existen herramientas que dan soporte automático para algunas refactorizaciones frecuentes [20]. La Figura 4.2 presenta

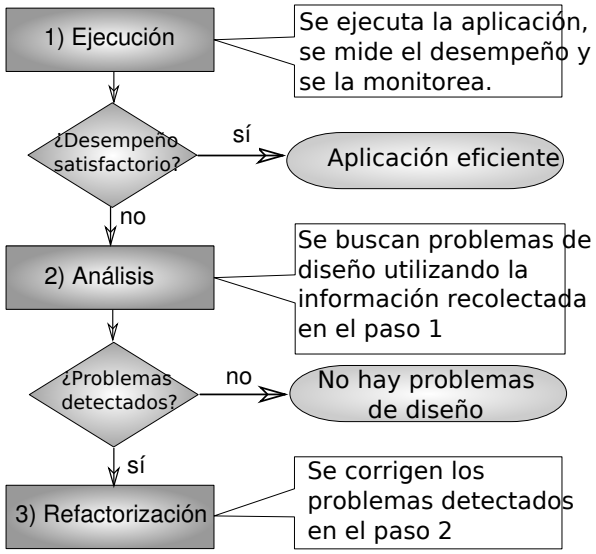


**Figura 4.2:** Diagrama de actividades del proceso propuesto.

gráficamente el proceso descripto.

El enfoque propuesto en esta tesis se materializó en una herramienta aplicable a la optimización de sistemas J2EE. Los detalles acerca del diseño y la implementación de la misma se presentan en el capítulo 5. El rol de esta herramienta es fundamental en el proceso esquematizado porque permite prescindir de expertos en *performance*, y ahorrar mucho tiempo en la identificación de antipatrones. Toda la información que se puede utilizar para detectar estos problemas es analizada automáticamente, evitando el costoso trabajo de buscar manualmente en grandes volúmenes de datos.

## 4.1. Obtener información del sistema

El primer paso en este proceso es determinar qué información se debe obtener del sistema. Esto dependerá de cuáles son los problemas de diseño que se quieren identificar. Por ejemplo, si se desea determinar si está faltando implementar un caché de acceso a un recurso particular, entonces la información que se debe tener disponible es la tasa de accesos al mismo.

Para obtener este tipo de información de la aplicación a optimizar, nuestro enfoque se basa en técnicas de monitoreo en tiempo de ejecución. Idealmente, la aplicación debería estar siendo monitoreada cuando es utilizada por el usuario final, ya que en ese caso se evaluaría el comportamiento del sistema con la carga esperada y los patrones de uso reales. De

otra manera, se corre el riesgo de identificar potenciales problemas que finalmente no van a ocurrir, con la consiguiente pérdida de tiempo en mejorar una parte del sistema que no es problemática.

El objetivo de monitorear la aplicación es recolectar información sobre las interacciones que ocurren entre los componentes distribuidos del sistema, formando un modelo de la dinámica de la aplicación. Ese modelo contiene todas las invocaciones remotas detectadas durante el monitoreo realizado.

A grandes rasgos, para recolectar esta información es necesario interceptar las invocaciones que reciben los componentes. Idealmente, se debe realizar esto de manera no intrusiva, es decir, sin imponer costosas modificaciones sobre la aplicación original. Para lograr esto, existen distintos enfoques, tales como AspectJ que implementa el paradigma de Programación Orientada a Aspectos en Java [24], o los *proxies* dinámicos [30]. Los *proxies* dinámicos son una construcción de Java que permite decorar objetos con interceptores sin modificar la aplicación original. Como el nombre sugiere, los interceptores interceptan las invocaciones que llegan a sus objetos asociados, permitiendo realizar tareas adicionales ante cada invocación. En particular, aquí proponemos que ante cada invocación se registren:

- EJB invocado,
- clase invocadora,
- cantidad y tipos de objetos retornados.

Decorando todos los EJBObjects y EJBHomes de la aplicación se pueden interceptar todas las invocaciones que se realizan, obteniendo toda la información necesaria. Como sólo nos interesa capturar las invocaciones remotas, antes de agregar un proxy a un EJBObject se debe analizar de qué clase extiende, y sólo decorar los que implementen interfaces remotas. La forma de garantizar que todos los EJBs sean decorados es:

1. los EJBHome del árbol de directorios JNDI deben estar decorados,
2. cada vez que un EJBHome retorna un EJB, antes debe decorarlo, y
3. cada vez que como resultado de una invocación a un EJB se retorna otro EJB, antes debe decorarlo.

El producto que se obtiene como resultado del monitoreo es un conjunto de árboles de invocaciones, donde quedan explícitas las interacciones entre componentes remotos que se realizan para satisfacer cada requerimiento externo.

Se genera un árbol por cada requerimiento que realiza el usuario. La información monitoreada se representa como un árbol para mantener las relaciones entre invocaciones remotas. Los componentes de estos árboles son:

- Nodo raíz: Es creado cada vez que se recibe un requerimiento del usuario. La información que contiene incluye a la URL requerida por el usuario, los *timestamp* de inicio y fin del procesamiento, y un identificador del cliente si es posible obtenerlo.

- Nodo interno de Invocación: Representa a un método que ha sido invocado en forma remota. Guarda el nombre de clase del EJB invocado,el método invocado, la clase invocadora y línea del código fuente en que realizó la invocación, además de las cantidades y tipos de objetos retornados.
- Nodo interno de Búsqueda: Representa un acceso al servicio de directorios JNDI para recuperar un EJB. Este tipo de nodo contiene el nombre del recurso al que se accedió, además del nombre de la clase invocadora y línea del código fuente en que se hizo la búsqueda remota.
- Arcos: Los arcos relacionan a los nodos en base a qué proceso estaba siendo ejecutado cuando el nodo fue creado. De esta manera, todos los nodos que pertenecen al mismo árbol, es decir que están conectados mediante arcos al mismo nodo raíz, representan a todas la invocaciones remotas y búsquedas en el servicio de directorios realizados para satisfacer ese único requerimiento del cliente.

Para explicar concretamente en qué consiste este monitoreo se introduce un fragmento de aplicación de ejemplo esquematizado en la Figura 4.3, que corresponde a un operador logueán-

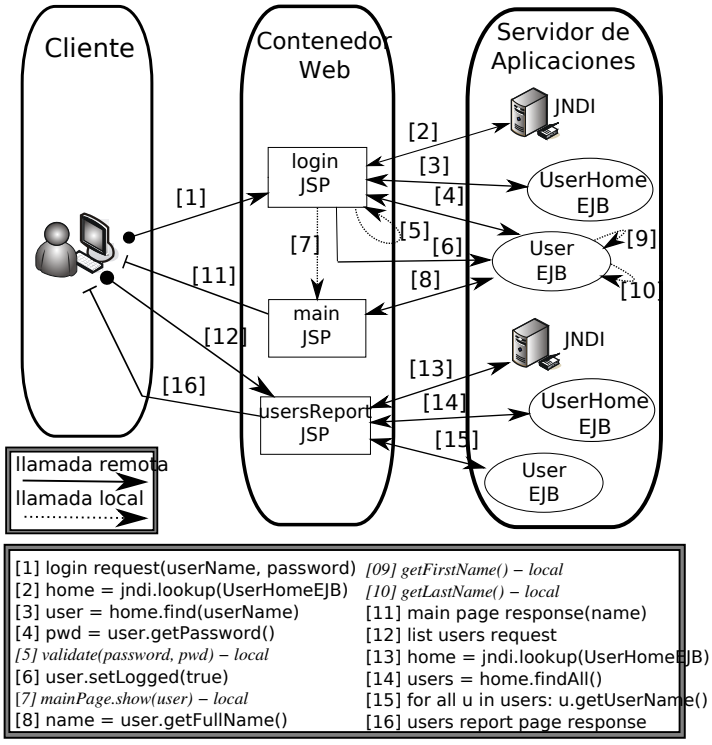


**Figura 4.3:** Fragmento de aplicación a analizar.

dose en una aplicación y presionando un botón para recuperar la lista de usuarios de la misma. Los componentes JSP que aparecen en la figura representan a la interfaz gráfica que utiliza el cliente de la aplicación. El UserEJB es el componente que mantiene los datos de cada usuario de la aplicación, como por ejemplo nombre o apellido, mientras que el UserEJBHome provee los métodos para administrar los UserEJB, como por ejemplo un método

para recuperar los datos de un usuario conociendo su login. A través de los métodos remotos expuestos por UserEJBHome, los clientes pueden recuperar los datos de los usuarios que precisan.

El proceso esquematizado comienza cuando el operador ingresa nombre de usuario y clave en la pantalla de login y presiona la tecla Enter, generando un requerimiento al servidor Web [1] que es atendido por la página llamada "login.jsp". Esta página entonces se conecta en forma remota al servidor JNDI [2] para recuperar el UserEJBHome. Teniendo el HomeObject se ejecuta un método de búsqueda de usuario por userName [3], recuperando el UserEJBObject correspondiente al usuario que inició la operación. Con ese EJBObject se recupera del servidor la clave del usuario [4], e invoca a un método local [5] para validar que coincida con la que se ingresó por pantalla.

Una vez que se verificó que la clave ingresada es correcta, se actualiza el estado del usuario invocando al método setLogged() [6], y se continúa el procesamiento en otra página del mismo servidor [7] llamada main.jsp. Esta página muestra un menú estático de opciones y el nombre completo del usuario logueado, que obtiene realizando una nueva invocación remota con el UserEJBObject [8]. Para poder responder el nombre completo, la instancia del UserEJB del servidor debe consultar dos métodos locales, uno para obtener el primer nombre [9] y otro para recuperar el apellido [10]. Teniendo la página armada, ésta es retornada al operador [11].

El operador solicita entonces ver la lista de usuarios [12] enviando un requerimiento al servidor Web que es atendido por la página usersReport.jsp. Esta página se conecta en forma remota al servidor JNDI [13] para recuperar una referencia al mismo UserHomeEJB que se había solicitado en el segundo paso. Con la referencia al home ejecuta un finder [14] y recupera los EJBObjects correspondientes a todos los UserEJB. Después itera sobre la lista, recuperando del servidor el nombre de cada usuario [15], con una llamada remota por cada uno. Por último, se retorna al operador [16] la página con el listado solicitado.

La Figura 4.4 presenta el resultado de monitorear el flujo de aplicación esquematizado pre-

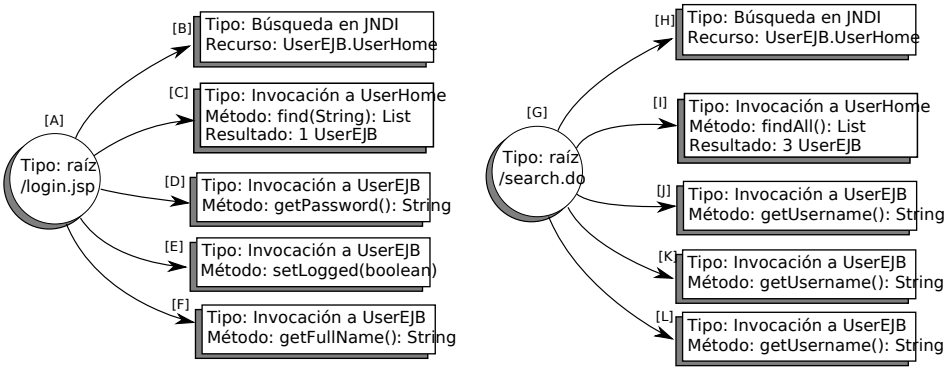


**Figura 4.4:** Árbol generado durante monitoreo.

viamente. Al momento de recibirse el primer *request* se crea el nodo raíz, que en el diagrama está rotulado como [A]. Cuando se recupera el EJBHome del árbol JNDI, la llamada es interceptada por un proxy y se genera el nodo [B] del árbol, que contiene el nombre del recurso accedido. Además de generar el nodo, se decora el EJBHome con un *proxy*, que intercepta el

llamado al método *find*(*String*), y genera el nodo [C] del árbol. Como ese método retorna una colección, se agrega al nodo la cantidad y tipo de resultados, que son necesarios porque algunos problemas de diseño dependen de estas variables. En este caso retorna un EJB, por lo que antes de devolvérselo al cliente le agrega otro *proxy*. Este nuevo *proxy* es el que intercepta la invocación remota a *getPassword*() agregando el nodo [D] al árbol. Después ocurre la validación de la *password* y se redirecciona a la siguiente página, pero ninguna de esas acciones implica llamadas remotas. De hecho, ningún *proxy* intercepta esas llamadas, por lo que son ignoradas y no forman parte del árbol resultante. Esto es exactamente lo que se desea, ya que los procesos locales no generan los problemas de eficiencia que queremos eliminar. Por este mismo motivo se van a ignorar las invocaciones locales que realiza el UserEJB a continuación, correspondientes a los métodos *getFirstName()* y *getLastName()*. Los siguientes nodos [E] y [F] son agregados por el *proxy* del UserEJBObject cuando se invoca a *setLogged*(*true*) y *getFullName*() respectivamente, de la misma manera que hizo cuando se invocó a otro método del mismo EJB. Finalmente, cuando se envía la respuesta al operador, se da por finalizado el árbol.

Dado que llega otro requerimiento de parte del operador, se crea un nuevo árbol de la misma manera que en el caso anterior. La raíz de este árbol se rotula como [G], sobre el lado derecho de la Figura 4.4. En las siguientes secciones seguiremos el proceso de optimización del fragmento de aplicación monitoreado.

## 4.2. Identificar problemas de diseño

La descripción del sistema en tiempo de ejecución que se obtiene en el paso anterior es analizada para detectar patrones conocidos en las interacciones que evidencien la presencia de una falla de diseño a mejorar. Estos patrones son conocidos como *antipatrones* y son conceptualmente similares a los patrones de diseño, sólo que lo que documentan son errores cometidos con frecuencia en la construcción del software. Además de describir el problema, los antipatrones indican cómo se puede solucionar.

Identificar antipatrones en el diseño de una aplicación es una tarea que, dado que la documentación no suele tener el nivel de detalle suficiente o incluso puede estar no actualizada, debe realizarse a partir del código fuente. Analizar las miles de líneas de código en forma manual es una actividad poco práctica, por lo que debe realizarse en forma automática [16]. Por este motivo, aquí se propone representar antipatrones en forma de reglas y analizar si esas reglas ocurren en el modelo generado en el paso anterior.

El análisis de toda esa información es un proceso computacionalmente costoso, por lo que no se realiza en simultáneo con el paso anterior de recopilación de información, sino que se ejecuta en forma posterior tomando como entrada el modelo generado. Una ventaja que surge de tener este proceso desacoplado del inicial es que se pueden ejecutar distintos conjuntos de reglas sobre el mismo modelo de interacciones remotas, para evaluar diferentes aspectos de la aplicación.

Modelar antipatrones como reglas permite que, en el caso de que se descubra un nuevo problema, se pueda programar una nueva regla para detectarlo y de esa manera automatizar el proceso de encontrar todas las ocurrencias del antipatrón. En las siguientes subsecciones, se presentan algunos antipatrones que se pueden detectar mediante este proceso.

### 4.2.1. Acceso redundante por JNDI

La especificación J2EE indica el uso de JNDI para acceder a diferentes recursos. Los servidores J2EE brindan acceso al servidor JNDI, de manera que los clientes pueden recuperar en forma remota todos los recursos que precisan. Estos recursos son principalmente objetos EJBHome y datasources. Los datasources son los objetos que crean las conexiones físicas a la base de datos. Cada vez que un cliente desea acceder a uno de estos recursos lo hace en forma remota, con el costo asociado al transporte por la red y serialización. En la Figura 4.5

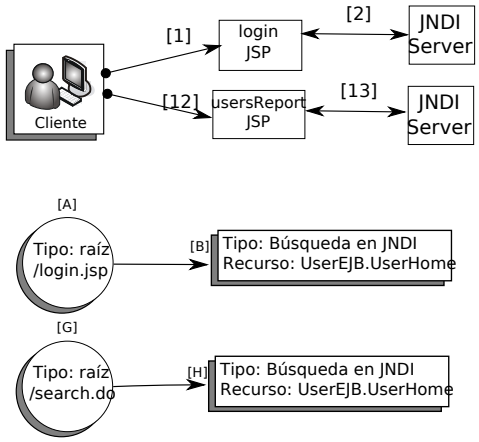


**Figura 4.5:** Antipatrón: acceso redundante por JNDI.

se muestra el fragmento de la aplicación de ejemplo donde se comete este antipatrón, y los nodos del árbol generado durante el monitoreo que le corresponden.

La regla que se utiliza para detectar este antipatrón es la siguiente:

```
when
    existen dos o más nodos de tipo "Búsqueda en JNDI"
    con el mismo recurso
then
    reportar antipatrón('Acceso redundante a JNDI')
```

La forma de solucionar este problema es aplicar el patrón de diseño conocido como Service Locator [1] o EJBHomeFactory [28]. Consiste en que el cliente guarde una referencia del objeto alojado en el árbol JNDI la primera vez que accede al mismo, y a partir de ahí reutilice el mismo objeto para evitar el acceso al servidor JNDI. En la Figura 4.6, que compara la situación problemática con la solución propuesta, se puede ver que se reduce el número de accesos remotos el aplicar el diseño recomendado, que sólo accede al servidor JNDI la primera vez. Intuitivamente, después de aplicar esta refactorización se mejora el tiempo de respuesta de los requerimientos al evitar esa búsqueda remota, y se mejora el desempeño de la aplicación en general porque se reduce la congestión de la red.

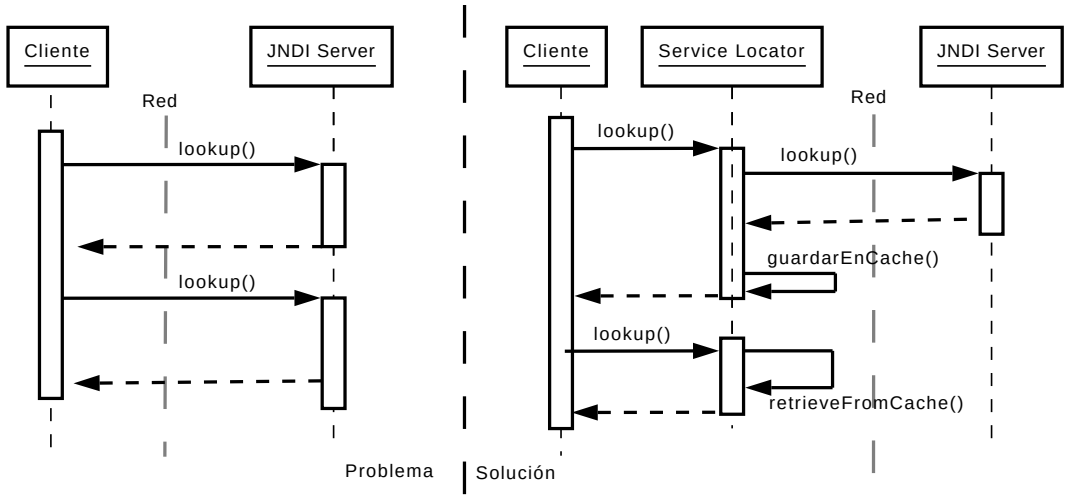


**Figura 4.6:** Solución al antipatrón acceso redundante por JNDI.

### 4.2.2. Múltiples accesos para obtener datos de un objeto

Cada vez que se realiza una invocación remota, hay costos asociados a la serialización, transporte por la red y deserialización de los objetos enviados. Si se realiza una llamada remota por cada dato que se precisa, esos costos adicionales son excesivos y degradan el funcionamiento del sistema. Este antipatrón está documentado en el trabajo *Software Performance Antipatterns* 3.1 bajo el nombre de *Camiones semivacíos*.

En la Figura 4.7 se puede ver el fragmento de la aplicación de ejemplo donde se comete este antipatrón, y los nodos del árbol generado durante el monitoreo que le corresponden.

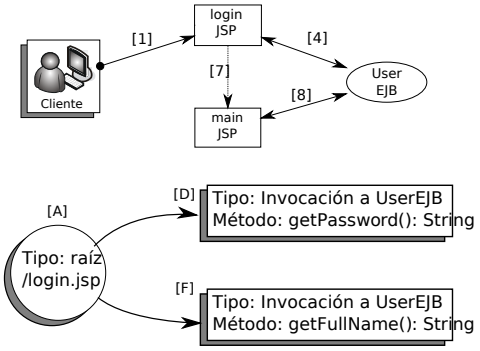


**Figura 4.7:** Antipatrón: múltiples *getters* en el mismo objeto.

La regla que se utiliza para detectar este antipatrón es la siguiente:

```
when
     existen dos o más nodos de tipo "Invocacion" al mismo EJB,
     ejecutando getters,
     y son nodos del mismo árbol
then
     reportar antipatrón('múltiples getters al mismo objeto')
```

La solución para evitar múltiples llamadas remotas para recuperar los datos de un objeto es aplicar el patrón de diseño ValueObject [1]. Su implementación, para el caso de un cliente solicitando información de un artículo, se puede ver en la Figura 4.8. Consiste en utilizar un

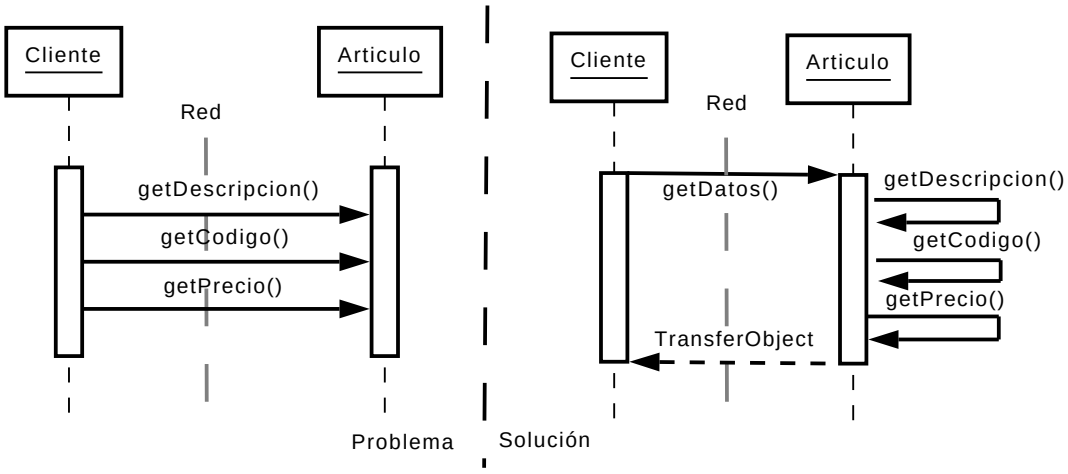


**Figura 4.8:** Solución al antipatrón múltiples accesos a un objeto remoto.

objeto *Transfer Object* que agrupe la información requerida, y pueda ser recuperado con una sola llamada remota. De esta manera, se evitan los costos asociados a las demás llamadas remotas.

### 4.2.3. Múltiples accesos a EJBs por requerimiento

Los clientes de un EJB pueden acceder a varios EJBs simultáneamente. Si por cada requerimiento–ej. establecer un atributo (*setter*) o recuperar otro (*getter*)– el cliente accede directamente a los EJBs que precisa, se requieren tantas llamadas remotas como requerimientos, cada una con su costo asociado. El antipatrón anterior 4.2.2 es un caso particular de éste, que ocurre cuando todas las invocaciones remotas se hacen al mismo EJB, y son todas "getters".

La Figura 4.9 muestra ver el fragmento de la aplicación de ejemplo donde se comete este

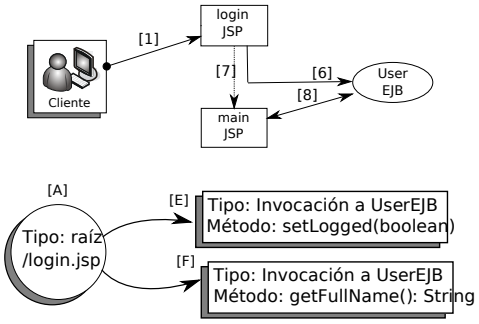


**Figura 4.9:** Antipatrón: Múltiples accesos a EJBs por requerimiento.

antipatrón, y los nodos del árbol generado durante el monitoreo que le corresponden.

La regla que se utiliza para detectar este antipatrón es la siguiente:

```
when
      existen dos o más nodos de tipo "Invocacion"
      y (no son todas invocaciones al mismo EJB
         o al menos un método de los ejecutados no es un "getter")
      y los nodos son del mismo árbol
then
      reportar antipatron('múltiples accesos por requerimiento')
```

Nótese que las condiciones de la regla evitan explícitamente que las ocurrencias del antipatrón anterior 4.2.2 sean detectadas nuevamente, porque la solución para este caso más general es distinta. En este caso la forma de evitar esas múltiples llamadas remotas es aplicando el patrón de diseño Session Facade [1], cuya implementación para un caso hipotético se puede ver en la Figura 4.10, donde se agrega una nueva operación en el servidor remoto que accede

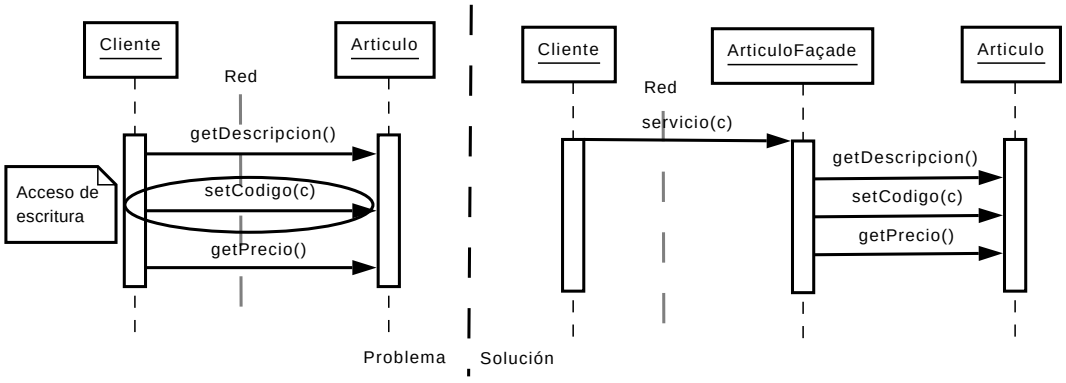


**Figura 4.10:** Solución al antipatrón accesos a múltiples EJBs.

a todos los métodos de los EJBs necesarios en forma local, y retorna todos los resultados requeridos. De esa manera el cliente hace una sola llamada remota al SessionFacade, evitando el costo de tener que hacer varias llamadas. En la figura se resalta el acceso de escritura que diferencia a este antipatrón del anterior.

### 4.2.4. Envío de excesiva cantidad de objetos

Las aplicaciones J2EE tienen frecuentemente facilidades de búsqueda y deben retornar los objetos que reunen las características solicitadas. Si una aplicación retorna un volumen muy grande de información al cliente, a este último le llevará un tiempo excesivo cargar todos los datos y la aplicación se percibe como lenta. Cuando los objetos retornados son EJBs, la situación no hace más que empeorar, por el costo extra asociado a los EJBs.

En la Figura 4.11 se puede ver el fragmento de la aplicación de ejemplo donde se comete este antipatrón, y los nodos del árbol generado durante el monitoreo que le corresponden.

La regla que se utiliza para detectar este antipatrón es la siguiente:

```
when
      existe un nodo de tipo Invocación
      y retorna una colección de objetos
```

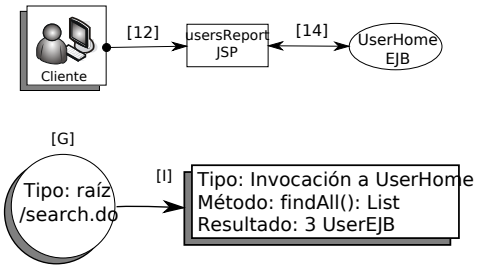


**Figura 4.11:** Antipatrón: Envío de excesiva cantidad de objetos.

```
    y esa colección tiene una longitud mayor a X
then
    reportar antipatron('envío de excesiva cantidad de objetos')
```

La solución a este problema es utilizar el patrón de diseño ValueListHandler [1], que consiste en retornar los objetos encontrados en pequeños grupos, paginados, en vez de todos juntos. La Figura 4.12 muestra que los accesos remotos disminuyen a uno por página en vez de

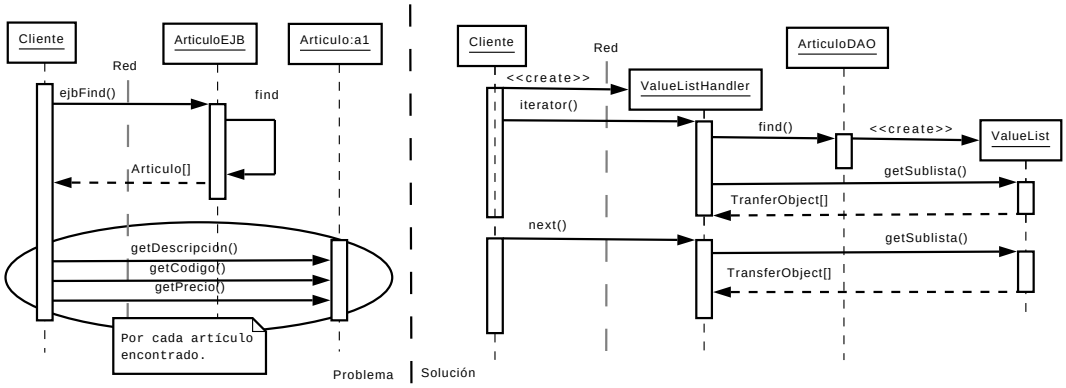


**Figura 4.12:** Solución al antipatrón envío de excesiva cantidad de objetos.

a uno por objeto encontrado. Además, se elimina el uso de EJBs para retornar los objetos, porque propone acceder directamente a la base de datos para popular *Transfer Objects* que serán retornados al cliente.

En esta sección se mostró una manera para representar antipatrones, la cual permite identificar a estos a partir del resultado de monitorear una aplicación. La siguiente sección describe el siguiente paso del proceso de optimización.

## 4.3. Reportar los problemas

Teniendo reglas identificadas para detectar problemas de diseño, y un árbol conformado por las interacciones remotas detectadas en la aplicación, el siguiente paso es utilizar las reglas para determinar la ocurrencia o no de antipatrones en la aplicación. Para esto se puede utilizar un motor de reglas. Un motor de reglas se enfoca en la representación del conocimiento

para expresar lógica proposicional y de primer orden en una forma declarativa, concisa y no ambigua. El motor de reglas, permite almacenar datos en forma de hechos y luego inferir información a partir de reglas. Una regla es una estructura que consiste de condiciones y acciones, según:

```
when
      <conditions>
then
      <actions>
```

Las condiciones se expresan en lógica proposicional, en general, estableciendo una secuencia de hechos que deben ser verdad para que la regla ejecute las acciones. En otras palabras, si se cumplen las condiciones de la regla (parte 1), se ejecutarán las acciones (parte 2). Como se puede observar, esta estructura es la misma que utilizamos para describir los antipatrones, haciendo muy sencilla la implementación de los mismos.

Hay varios algoritmos que se pueden utilizar para comparar los datos y hechos con las reglas. Los más populares son *Linear*, *Rete*, *Treat* y *Leaps*. El motor de reglas que elegimos para ejemplificar este trabajo es *JBossRules*. JBossRules implementa el algoritmo *ReteOO*, que extiende al algoritmo Rete mejorándolo y optimizándolo para sistemas orientados a objetos.

Hay dos métodos de ejecución para los motores de reglas: *Backward Chaining* y *Forward Chaining*. Un sistema Backward chaining es guiado por objetivos, lo que significa que se comienza con una conclusión que se quiere satisfacer. Si no lo logra, entonces busca conclusiones que sí puedan ser verdaderas, conocidas como "sub-objetivos", que pueden ayudar a satisfacer el objetivo actual. El proceso continúa hasta que se satisface la conclusión original o ya no quedan más sub-objetivos. *Prolog* es un buen ejemplo de motor Backward Chaining. En cambio un sistema Forward Chaining es guiado por los datos, y por lo tanto es reaccionario: los hechos son insertados dentro de la memoria de trabajo, lo que provoca que una o más reglas sean verdaderas al mismo momento, siendo agendadas para su ejecución. JBossRules es un motor Forward Chaining, por lo que comienza con un hecho, se propaga a través de las reglas y finaliza con una conclusión.

Los principales motivos para utilizar un motor de reglas en vez de escribir el algoritmo de detección directamente en un lenguaje de programación tradicional son los siguientes:

- Programación declarativa: las reglas son mucho más fáciles de leer que el código.
- Clara separación entre datos y lógica: toda la lógica para detección de patrones está en las reglas. Modificar y agregar reglas no requiere involucrarse con la representación de los datos.
- Velocidad y escalabilidad: el motor de reglas es eficiente y escalable, garantizando que se puedan procesar todos los datos capturados durante el monitoreo.
- Centralización del conocimiento: en los archivos de reglas está almacenado todo el conocimiento sobre antipatrones de diseño que tiene la herramienta.

En particular, el uso del motor de reglas nos facilita la identificación de ocurrencias de antipatrones en el sistema. Muchas veces la solución para erradicar un antipatrón consiste en

introducir un patrón de diseño al código. Volviendo al ejemplo que estamos desarrollando en este capítulo, las recomendaciones que automáticamente se pueden extraer a partir de las reglas son:

- Aplicar el patrón de diseño "Service Locator" para acceder al recurso "UserEJB.UserHome". El código que se debe actualizar para que recupere el recurso a través del ServiceLocator está en loginPage linea 47 y en usersReport línea 73.
- Aplicar el patrón de diseño "Value Object" para encapsular los getters "getPassword" y "getFullName" del UserEJB. Se debe utilizar el nuevo "Value Object" en vez de las llamadas individuales que se hacen desde loginPage línea 53 y mainPage línea 17.
- Agregar un "SessionFacade" para agrupar el acceso a las siguientes operaciones remotas: UserHome.find (loginPage línea 50), UserEJB.getPassword (loginPage línea 53), UserEJB.setLogged (loginPage línea 60) y UserEJB.getFullName (mainPage línea 17).
- Limitar la cantidad de objetos retornados en el método UserHome.findAll, agregando un ValueListHandler. Esta operación esta siendo invocada desde usersReport línea 75.

Con estas recomendaciones un programador puede aplicar las refactorizaciones descriptas. Es su responsabilidad asegurarse de que no se modifique la semántica de la aplicación, ya que no todas las transformaciones pueden hacerse en forma automática. Por ejemplo, para agrupar en el SessionFacade los métodos getPassword y setLogged, es necesario que la lógica de validación de password se haga ahora en el servidor de aplicaciones y no en el servidor Web. Típicamente, el código de validación de claves posee importantes restricciones de seguridad que implican que no pueda estar en un servidor u otro. Por lo tanto, en este caso es el desarrollador quien tomará la decisión sobre cómo aplicar la refactorización. En cambio, soluciones más sencillas que no afectan a la semántica de la aplicación pueden ser automatizadas y podrían llegar a resolverse sin intervención del desarrollador, como por ejemplo agregar un ServiceLocator con caché de EJBHomes.

Una vez que se han aplicado todos los patrones recomendados se obtiene una nueva versión de la aplicación. Esta nueva versión, se debe volver a monitorear para ejecutar las reglas y así determinar la existencia de más problemas de diseño que afectan la *performance*. La Figura 4.13 muestra el resultado del monitoreo de la aplicación optimizada, donde se puede

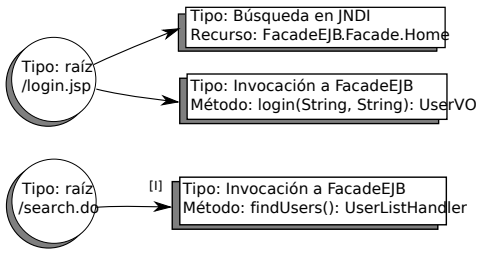


**Figura 4.13:** Resultado del monitoreo de la aplicación optimizada.

ver que ninguna regla daría resultado verdadero, por lo que ya no hay más refactorizaciones que hacer. Se debe notar cómo se han reducido las interacciones remotas de las 10 iniciales a sólo 3, mejorando claramente el uso de los recursos del sistema.

### 4.3.1. Conclusiones

El enfoque propuesto en este trabajo posee características que lo distinguen de los trabajos relacionados presentados en el capítulo anterior. Básicamente, estas características son:

- Extrae la información necesaria para detectar problemas desde la aplicación.
- Detecta antipatrones automáticamente.
- Propone las refactorizaciones necesarias para optimizar la aplicación.

Gracias a que se extrae la información relevante directamente de la aplicación, no hace falta desarrollar modelos para predecir cuál será el desempeño del sistema. Con respecto a la capacidad de identificar la ocurrencia de los antipatrones automáticamente, ésta hace prescindible la participación de un experto en optimización de aplicaciones J2EE. Tampoco es necesario que la persona que realiza el proceso de optimización conozca cómo está implementada la aplicación.

Por otro lado, la manera en que se proponen las soluciones a los problemas encontrados, presenta detalladamente: cuál es el refactor a aplicar y, muy importante, en dónde. De esta manera, el enfoque permite que desarrolladores no especializados en *performance* sepan qué cambios hacer al diseño para que la aplicación mejore su desempeño.

En el siguiente capítulo se presenta el diseño e implementación de una herramienta Java que materializa este enfoque al automatizar los procesos de detección de antipatrones y propuesta de soluciones. Veremos cómo está implementada la herramienta y qué facilidades del lenguaje aprovecha para brindar la mayor cantidad de información posible al programador.

# CAPÍTULO 5

## Diseño e implementación

En este capítulo veremos cómo la idea presentada en el capítulo 4 se ha materializado en una herramienta Java que permite identificar los problemas de *performance* en sistemas J2EE.

## 5.1. Descripción general

Una de las propiedades que se desea para esta implementación de la herramienta es que sea portable, para que pueda ser utilizada en aplicaciones J2EE instaladas en diferentes contenedores. También, para que pueda ser utilizada en ambientes productivos, se cuidó el consumo de recursos de la solución. Muchas decisiones de diseño se tomaron considerando la portabilidad y desempeño de las opciones.

La Figura 5.1 esquematiza la interacción entre los componentes que intervienen en el proceso

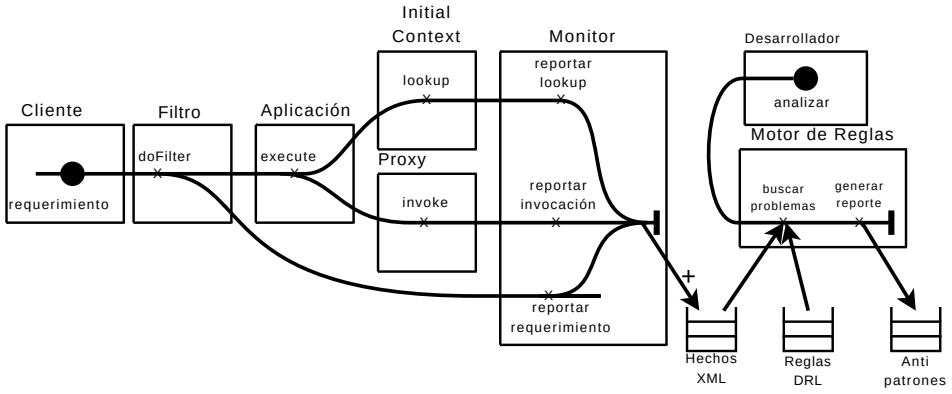


**Figura 5.1:** Vista general del diseño de la herramienta.

de detectar antipatrones a partir de la aplicación en funcionamiento.

El primer paso del proceso, como vimos en la sección 4.1, consiste en recolectar información sobre la aplicación en ejecución utilizando un monitor. Para garantizar que la implementación del monitor sea portable entre los distintos contenedores, todos sus componentes son componentes estándares de la especificación J2EE, como *proxies* dinámicos, filtros e *InitialContext*.

Los *proxies* son objetos que permiten interceptar las invocaciones a métodos,y así insertar comportamiento entre el invocador y el objeto invocado. Desde la perspectiva del invocador, no hay diferencia entre el *proxy* y la clase real que encapsula. En Java, la clase de un *proxy* dinámico es una clase que implementa una lista de interfaces especificadas en tiempo de ejecución, al momento de crearla. Cuando se invoca a un método definido en una esas interfaces sobre una instancia de esta clase, la invocación es transmitida a otro objeto especificado por el programador, conocido como "*Invocation Handler*". Todas las invocaciones realizadas sobre la instancia del proxy son derivadas a un método del *invocation handler* llamado *invoke*, que recibe como argumentos el nombre del método que se quiere ejecutar y los objetos pasados como parámetros al mismo. En general, este método intercepta todas las invocaciones al objeto real, en particular permite agregar la lógica de monitoreo necesaria para generar el árbol de invocaciones remotas, sin modificar el código de la aplicación original.

Los filtros son clases que extienden de *javax.servlet.Filter* y son parte de la especificación de Java *Servlets* desde la versión 2.3. Estos filtros interceptan dinámicamente los requerimientos a los *servlets* y las respuestas de los mismos, pudiendo inspeccionar toda la información intercambiada e incluso modificarla. Típicamente los filtros no generan respuestas, sino que proveen funciones genéricas que pueden añadirse a cualquier *servlet* o página JSP [18]. El método del filtro que se ejecuta en cada requerimiento se llama *doFilter*, y es para nuestro propósito equivalente al *invoke* del *invocation handler*. En esta implementación utilizamos filtros para incorporar lógica de monitoreo a la aplicación en forma transparente y sin necesidad de recompilar el código fuente, ya que se incorporan vía configuración.

Además de interceptar los requerimientos Web con filtros y las invocaciones remotas con *proxies* dinámicos, el monitor también necesita estar informado acerca de los accesos que se hacen al servicio de nombres. El punto de acceso a este servicio es una implementación de la interfaz javax.naming.Context. Por lo general, una aplicación determina en tiempo de ejecución qué implementación del servicio de nombres utilizar, basándose en configuración. El monitor provee una implementación propia para poder interceptar los accesos al servicio de nombres.

La información obtenida por los distintos componentes del monitor es almacenada en un árbol en memoria, en una clase Java llamada Monitor, hasta el momento de persistirla para su uso posterior. La información del monitoreo se guarda temporalmente en memoria para evitar el alto costo que tiene acceder al disco, pero no puede mantenerse en memoria durante toda la ejecución porque es un recurso limitado y conviene utilizarlo para guardar los datos de la aplicación y no del monitor. Por este motivo se graba la información del monitoreo y se libera la memoria a intervalos regulares.

Para grabar la información del monitoreo se utiliza XML. XML, sigla en inglés de Extensible Markup Language (lenguaje de marcas extensible), es un metalenguaje de etiquetas desarrollado por el World Wide Web Consortium (W3C) que permite definir la gramática de lenguajes específicos. Por lo tanto XML no es un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. La función principal de XML es describir

datos, y se propone como un estándar para el intercambio de información estructurada entre diferentes aplicaciones y plataformas. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil. Dadas estas características de XML, y de la existencia de herramientas que generan y procesan documentos XML en forma automática, es que se utiliza para almacenar la información recolectada en esta primera etapa de monitoreo.

El siguiente paso del proceso, como vimos en la Sección 4.2, consiste en detectar los problemas de *performance* utilizando el motor de reglas. JBossRules tiene dos partes bien diferenciadas: la escritura de las reglas, y su ejecución. Las reglas se escriben en un lenguaje propio del motor, basado en lógica de primer orden, que extiende de la lógica proposicional. Una proposición es un enunciado que puede ser clasificado como verdadero o falso. Los antipatrones que la herramienta puede detectar se han codificado como reglas en este lenguaje. Las reglas son compiladas para generar el RuleBase, que es utilizado en el momento de la ejecución para obtener conclusiones a partir de los hechos, que no son más que JavaBeans conteniendo la información recolectada durante el monitoreo.

Ejecutar las reglas es el último paso del proceso, donde se genera un listado de recomendaciones para mejorar el desempeño del sistema, como se explicó en la sección 4.3. Este listado es el resultado final producido por la herramienta. A partir del mismo los desarrolladores deben modificar el código y repetir todo el proceso hasta que la aplicación obtenga el desempeño esperado.

## 5.2. Descripción detallada

En las siguientes subsecciones veremos en mayor detalle cómo están implementados los componentes de esta herramienta.

### 5.2.1. Filtro

El componente que inicia el monitoreo de cada requerimiento es el filtro. Este componente intercepta todos los requerimientos que se hacen al servidor, y es el responsable de crear el nodo raíz del árbol que lo represente. Para hacerlo utiliza la URL solicitada por el cliente, que le es proporcionada por el contenedor de servlets. La URL será utilizada al momento de reportar los antipatrones, para ayudar a identificar los usos que se hacen del código ineficiente.

Otra función que cumple el filtro es la de recibir comandos para grabar el árbol almacenado en memoria a un archivo, liberando la memoria correspondiente, además de recibir comandos para encender y apagar el monitoreo. Esta funcionalidad está presente para dar la posibilidad de que se desactive en casos de sobrecarga del servidor, facilitando su adopción en sistemas productivos.

Para que el filtro pueda interceptar todos los requerimientos que se hacen al servidor, debe ser configurado en la aplicación de acuerdo a la especificación J2EE. La forma de configurar el filtro de la herramienta es editar el archivo *web.xml* de la aplicación Web a analizar y agregar el siguiente fragmento de XML:

```
<filter>
        <filter-name>MonitorFilter</filter-name>
        <filter-class>
            ar.edu.unicen.j2ee.problems.MiFilter
        </filter-class>
</filter>

<filter-mapping>
        <filter-name>MonitorFilter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

El primer fragmento declara el filtro del monitor, que es la clase java MiFilter del paquete ar.edu.unicen.j2ee.problems, y le asigna el nombre de "MonitorFilter". Después se le indica con un patrón ("url-pattern") qué requerimientos deben ser interceptados por el filtro. El patrón "/*" debe ser interpretado como "todos los requerimientos". Con esta configuración se logra que todos los requerimientos sean interceptados por el filtro de la herramienta antes de que se ejecute el código de negocio.

Una vez que el filtro está instalado en la aplicación, tiene la funcionalidad esquematizada en la Figura 5.2, y consiste básicamente en:

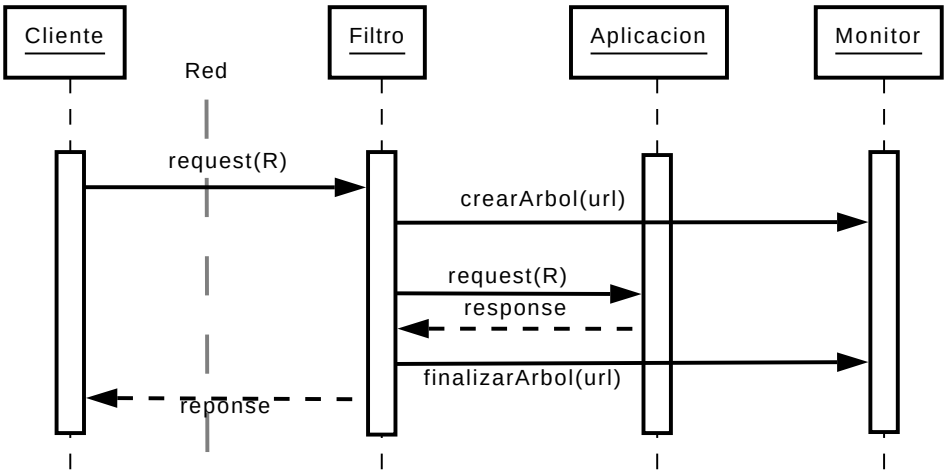


**Figura 5.2:** Comportamiento del filtro.

1. crear el primer nodo del árbol de invocaciones y marcarlo como "árbol activo" para que los *proxies* le agreguen los nodos con las invocaciones remotas que intercepten,
2. delegar a la aplicación la ejecución del requerimiento, y finalmente
3. guardar el árbol en memoria.

Las operaciones sobre el árbol están encapsuladas en el objeto Monitor, por lo que el filtro interactúa con ese objeto en vez de acceder directamente al árbol.

La responsabilidad principal del filtro es determinar el comienzo y fin de cada requerimiento, para que los *proxies* puedan agregar los nodos en el árbol correspondiente. Esos *proxies* son creados por la herramienta, pero para hacerlo debe tener control del *InitialContext*.

### 5.2.2. InitialContext

La clase *InitialContext* es la interfaz con el servicio de nombres y es utilizada por las aplicaciones para recuperar los EJBHome. El monitor requiere que la implementación estándar de J2EE sea reemplazada por una implementación propia, que tiene dos funcionalidades bien diferenciadas, como se resaltan en la Figura 5.3.

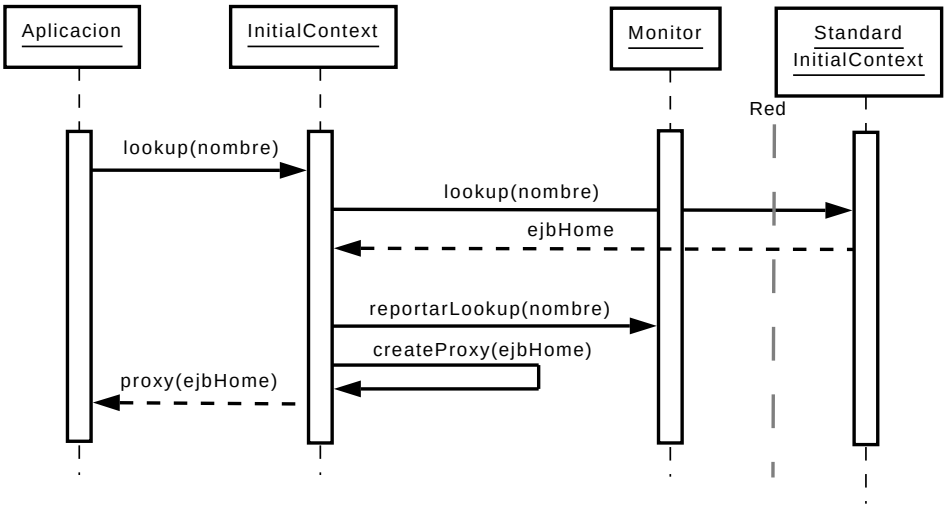


**Figura 5.3:** Comportamiento del InitialContext.

Por un lado, cada vez que se solicita un EJBHome debe registrar el nombre del recurso solicitado. Esta información es necesaria para poder reportar el problema de accesos redundantes por JNDI con la información suficiente para que pueda ser corregido. El nombre del recurso solicitado es recibido por el *InitialContext* como parámetro.

La otra función que cumple el *InitialContext* de la herramienta es la de crear *proxies* sobre las instancias de EJBHome recuperadas desde JNDI, de manera que el cliente recibe el *proxy* y siempre accede a instancias de EJBHome que están intervenidas por el monitor.

### 5.2.3. Proxies

Todos los EJBHomes y EJBs de la aplicación deben ser accedidos a través de un *proxy* dinámico generado por el monitor, de tipo EJBHomeProxy y BeanProxy respectivamente. Estos *proxies* interceptan todas las llamadas a métodos de los EJBHomes y EJBs, y tienen la responsabilidad de crear *proxies* sobre cada EJB que sea retornado al cliente. Para lograrlo, analizan el objeto que retorna cada invocación en busca de EJBs. Este es un proceso recursivo que sigue el siguiente algoritmo:

```
decorarObjeto(_o) {
    si _o es un EJB entonces {
        retornar nuevo proxy para _o
    }
    si _o es una coleccion {
        crear nueva colección _c conteniendo
```

```
            _c[i] = decorarObjeto(_o[i])
        retornar la colección _c
    }
    si _o es un mapa {
        crear un nuevo mapa _m, conteniendo
        _m.dupla[i] = { decorarObjeto(_o.llave[i]),
                        decorarObjeto(_o.valor[i])}
        retornar el mapa _m
    }
    retornar _o sin decorar
}
```

Con este algoritmo el monitor se garantiza que todos los EJBs a los que el cliente tiene acceso están intervenidos, de manera que se pueden interceptar todas las invocaciones remotas. La Figura 5.4 muestra cómo los proxies interceptan las invocaciones remotas y las informan al

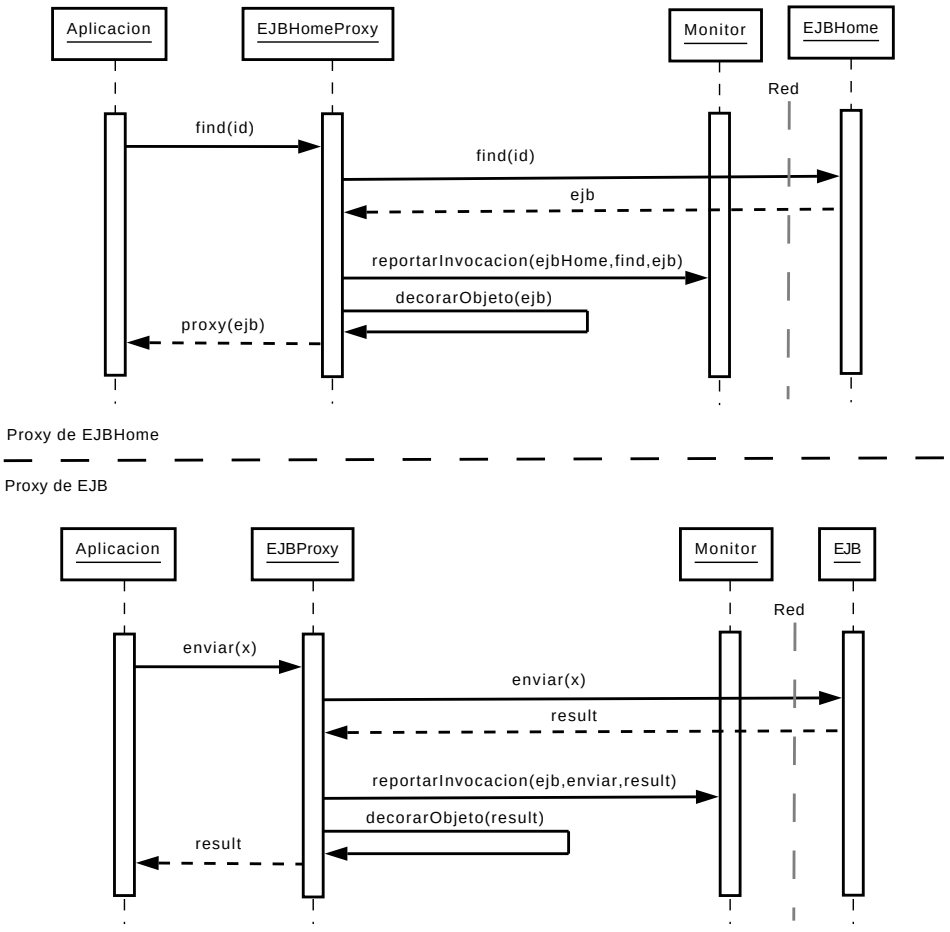


**Figura 5.4:** Comportamiento de los proxies.

Monitor para que cree los nodos correspondientes. En la parte superior de la figura se ve la interacción que ocurre cuando se intercepta una invocación a un EJBHome que retorna un EJB. Se puede ver que primero se reporta la invocación remota al Monitor, y luego se crea un proxy del EJB para retornar al cliente. La parte inferior de la figura muestra una

interacción muy similar, pero en este caso se intercepta una invocación a un EJB que retorna un objeto java simple. El comportamiento es el mismo, salvo que el método *decorarObjeto* no crea ningún proxy, y el objeto que se retorna al cliente es el mismo que retornó el EJB.

Para que se puedan crear los nodos del árbol es necesario que los proxies provean cierta información. Sea que se está interceptando una invocación a un EJB o un EJBHome, la información que se requiere es la misma: el nombre de la clase sobre la que se está ejecutando un método, el nombre del método ejecutado, y el tipo y cantidad de objetos retornados. Los dos primeros datos –nombre de clase y método que se está invocando– son recibidos por el proxy dinámico como parámetros. La información sobre los objetos que se retornan es necesaria para detectar el antipatrón "Envío de excesiva cantidad de objetos", y se obtiene mientras se ejecuta el algoritmo esquematizado previamente. Concretamente, la información que se agrega al nodo es la cantidad de objetos que se retornan, agrupados por tipo de objeto.

Cada vez que un nuevo nodo es creado, éste es agregado al "árbol activo", que es mantenido por el objeto de tipo Monitor.

### 5.2.4. Monitor

El componente que mantiene la información recolectada por el filtro, el InitialContext y los proxies dinámicos se llama Monitor. Implementa los patrones de diseño Singleton y Builder [13] para facilitar la construcción del árbol de invocaciones simplificando la lógica necesaria en los componentes que recolectan la información.

La Figura 5.5 muestra un diagrama de clases de este componente. La variable singleInstance

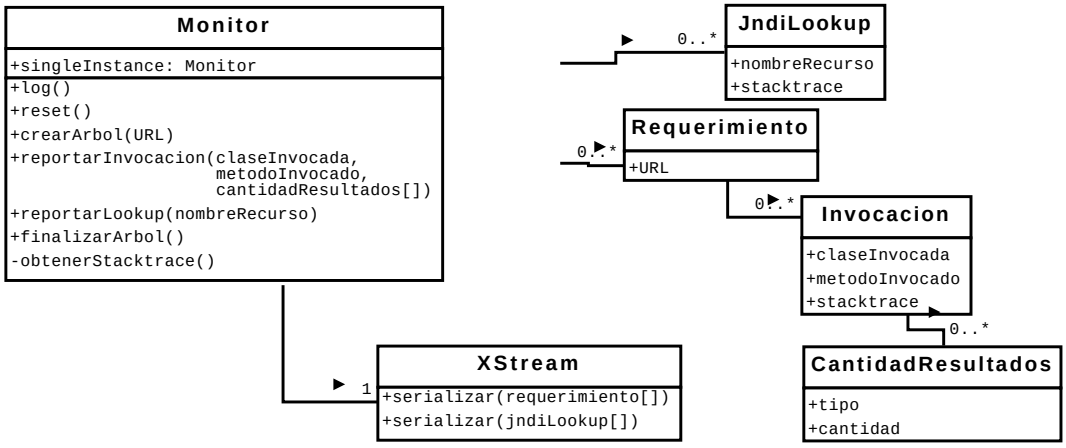


**Figura 5.5:** Estructura de la clase Monitor.

corresponde a la implementación del patrón Singleton, que garantiza que existe una única instancia de la clase Monitor, permitiendo que todos los objetos que recolectan información de la aplicación en ejecución accedan a la misma instancia. El patrón Builder encapsula la estructura interna del árbol, para que los proxies se limiten a informar los datos que obtuvieron sin preocuparse por la implementación del mismo, y está implementado a través de los métodos “crearArbol”, “reportarLookup”, “reportarInvocacion”, y “finalizarArbol”.

Esta clase mantiene un árbol activo por cada requerimiento. Cada árbol es creado por el Filtro al invocar el método “crearArbol” cada vez que intercepta un requerimiento. Los nodos son agregados al árbol cada vez que se invoca a uno de los métodos “reportarInvocacion” y “reportarLookup”. Esto sucede cuando cuando un proxy intercepta una invocación remota o cuando se solicita un recurso al servicio de nombres, respectivamente. El árbol deja de estar activo, es decir que ya no puede ser modificado, una vez que el Filtro que lo creó lo cierra, invocando al método “finalizarArbol” del Monitor. Esto sucede después de que la aplicación original haya terminado de procesar el requerimiento..

Cada vez que se crea un nuevo nodo del árbol, junto con la información enviada por los proxies o el InitialContext, el Monitor agrega el nombre de la clase del objeto invocador y la línea del código fuente en que se realizó el acceso remoto. Esta información es necesaria para que se puedan indicar, al momento de reportar los problemas, cuál es el código que se debe modificar para mejorar el desempeño de la aplicación. Recuperar los datos del invocador no es trivial en Java. Para lograrlo se crea un StackTrace, que es una imagen de la pila de ejecución conteniendo todos los nombres de clases y métodos que fueron ejecutándose para llegar a ese punto, incluidas las líneas del código fuente si se compiló la aplicación con información de debug. Analizando el contenido de ese listado se obtiene con precisión los datos del invocador, que son almacenados en el nodo que está siendo creado.

Además de la interfaz del Monitor, en la Figura 5.5 puede verse la implementación en objetos de la información recolectada durante todo el proceso. El monitor mantiene listas separadas para los accesos al servicio de nombres (JNDI) que para las invocaciones remotas, porque contienen distinta información. Los accesos JNDI sólo guardan el nombre del recurso buscado y el stacktrace, mientras que los requerimientos guardan la URL y un listado de Invocaciones, donde cada invocación a su vez tiene información propia.

El Monitor mantiene estos objetos en memoria hasta que se invoca a su método “reset”, que libera la memoria borrando toda la información recolectada hasta el momento. Para que no se pierdan esos datos, primero se debe invocar al método “log”, que escribe en un archivo toda la información recolectada hasta el momento. El formato elegido para escribir el archivo es XML, para lo que se utiliza XStream [44].

XStream es una librería que serializa objetos Java a XML, y puede reconstruir los objetos Java a partir del XML generado. Fue elegida para esta operación porque sus creadores hicieron énfasis en mantener buena *performance*, consumiendo poca memoria y usando poco procesador. Una vez que la información describiendo el comportamiento de la aplicación está grabada en un archivo en formato XML, puede ser procesada por el motor de reglas.

### 5.2.5. Motor de reglas

Para que el motor de reglas pueda detectar los antipatrones, debemos proveer la descripción de los antipatrones escritas como reglas. Las reglas se escriben en un lenguaje textual propio de JBossRules, que es un formato liviano, con muy poca puntuación, en archivos de texto con extensión "drl". Las reglas deben hacer referencia a los objetos que fueron almacenados en el archivo XML, que son de las clases *JndiLookup*, *Requerimiento*, *Invocacion* y *CantidadResultados*. No es la intención de este trabajo explicar el lenguaje de programación de reglas de JBossRules, pero veamos a modo de ejemplo la implementación de una de las reglas.

Para detectar que es necesario incorporar un ServiceLocator porque se está accediendo más de una vez al mismo recurso del servicio de nombres, se define la siguiente regla:

```
rule "Hace falta un Service Locator con Cache (SLC)"
when
// 1.
        $lookup: JndiLookup(
                $resourceName: jndiName,
                $stackTrace: shortStackTrace
        )
// 2.
        exists JndiLookup(
                jndiName == $resourceName,
                this != $lookup
        )
// 3.
        not Antipatron(
                id == ($resourceName + $stackTrace),
                type == "SLC"
        )
then
// 4.
        insert(new Antipatron(
                ($$resourceName + $stackTrace),
                "SLC",
                "Debe agregar un ServiceLocator con Cache " +
                "para acceder al recurso \"" + $resourceName +
                "\" que es accedido desde " + $stackTrace)
         );
end
```

La primer parte de la regla guarda en variables una instancia de *JndiLookup* –que fue reportada por el InitialContext–, el nombre del recurso al que reporta que se accedió, y el stacktrace con la información de la clase que solicitó el recurso (1). Después verifica que exista otra instancia de *JndiLookup* reportando que se accedió al mismo recurso (2). Y en la última parte de la condición se verifica que no se haya identificado previamente este mismo antipatrón (3), para evitar entradas duplicadas en el informe final.

Si toda la condición se cumple, entonces se ejecuta la segunda mitad, correspondiente a la acción, que en este caso inserta en la memoria de trabajo del motor de reglas un objeto con toda la información conocida sobre el antipatrón (4).

En el apéndice A se presentan las reglas que implementan el resto de los antipatrones actualmente soportados por la herramienta. La reglas que detectan los otros antipatrones tienen la misma estructura, y todas generan un objeto de tipo *Antipatron* cuando se cumplen las condiciones. Las instancias de *Antipatron* creadas por las reglas serán posteriormente utilizadas para armar el informe final que se le presentará al usuario.

### 5.2.6. Reporte

Una vez que se tiene el archivo XML con la información recolectada durante el monitoreo, el motor de reglas debe ejecutar las reglas para encontrar los antipatrones. Como las reglas

están escritas para analizar objetos Java y no XML, primero se utiliza XStream para reconstruir los objetos Java. Estos objetos son los "hechos" del motor de reglas, y son insertados en su memoria de trabajo.

JBossRules ejecuta las reglas, y como resultado del procesamiento en la memoria de trabajo del motor de reglas quedan almacenados los objetos de tipo Antipatron encontrados. Para armar el informe al usuario simplemente se recuperan esos objetos y se imprimen por la salida estándar. Veamos por ejemplo cómo se informa la ocurrencia del antipatrón "Acceso redundante por JNDI", cuya regla vimos en la sección anterior:

```
Antipatron{
        tipo = SLC,
        solucion = Debe agregar un ServiceLocator con cache para acceder
            al recurso "RecordSessionEJB.RecordSessionHome" que es
            accedido desde com.bea.medrec.utils.EJBHomeFactory.lookupHome
            (EJBHomeFactory.java:111)
}
```

Esta información la recibe el usuario de la herramienta, y es la que debe utilizar para corregir los problemas de *performance*. En el siguiente capítulo veremos los resultados obtenidos al probar la herramienta descripta sobre algunas aplicaciones de código abierto.

# CAPÍTULO 6

## Resultados experimentales

En este capítulo analizaremos la efectividad de la herramienta propuesta probándola sobre algunas aplicaciones de código abierto disponibles en Internet. Para la realización de las pruebas se utilizó *The Grinder* [2], un *framework* para evaluación del comportamiento de aplicaciones en tiempo de ejecución. Básicamente, este *framework* permite grabar una secuencia de requerimientos enviados al servidor y reproducirlos en forma automática todas las veces que sea necesario, es decir, que una vez grabada la secuencia de uso permite prescindir del usuario de pruebas o *beta tester*.

El procedimiento seguido en la prueba de la herramienta sobre todas las aplicaciones es el mismo y se describe a continuación:

1. Se instala la aplicación a analizar según las instrucciones de la misma.
2. Se utiliza "The Grinder" para capturar una sesión de uso de la aplicación, y se genera un archivo con una secuencia de pasos –comúnmente denominado *script*– que permita recrearla.
3. Se ejecuta el *script* de "The Grinder" simulando 100 repeticiones de la sesión grabada en el paso 2. Como resultado de esta ejecución, se obtiene una tabla con los tiempos de respuesta de los distintos requerimientos ejecutados, medidos desde el cliente de los servicios.
4. Se modifica la aplicación para activar el monitoreo. Después, se instala la versión modificada en lugar de la original.
5. Se vuelve a ejecutar el script de "The Grinder", pero esta vez los tiempos medidos corresponden a la aplicación siendo monitoreada. Además del reporte de tiempos se obtiene también en esta ejecución el archivo "log_ejecución.xml" con el modelo del comportamiento de la aplicación generado durante el monitoreo.

6. Se utiliza el motor de reglas para ejecutar las reglas creadas para detectar antipatrones sobre el archivo generado en el paso 5, "log_ejecución.xml". Como resultado de esta ejecución se obtiene el listado de antipatrones encontrados con toda la información pertinente para su corrección.

Una vez recolectada toda la información necesaria, se comparan los tiempos de respuesta de la aplicación original contra los correspondientes de la aplicación monitoreada. De esta manera, se espera medir la penalidad en desempeño que impone el uso de esta herramienta de monitoreo, para concluir si es posible utilizarla en sistemas productivos y mantener la calidad del servicio dentro de límites aceptados. Vale destacar que en pos de una evaluación justa, las aplicaciones originales y sus contrapartes modificadas se instalaron sobre el mismo hardware, siendo las características principales de éste: Intel Pentium ejecutando a 1.86GHz con 512 MB de memoria RAM.

Por otro lado, intentaremos determinar la efectividad de este método para detectar antipatrones. En primer lugar, se revisará, manual y exhaustivamente, el código de la aplicación a evaluar para determinar todos los antipatrones que ocurren en la misma. En segundo lugar, se verificará la ocurrencia de los antipatrones detectados y la precisión de la información proporcionada para su corrección, así también como la viabilidad de la misma. En tercer, se analizarán los antipatrones de diseño no hallados, y discutirán los motivos por los cuales estos no fueron encontrados. Finalmente, siguiendo las recomendaciones realizadas por la herramienta, se modificarán las aplicaciones originales y compararán el desempeño de éstas contra el de las versiones originales.

## 6.1. Avitek Medical Records (MedRec)

MedRec es una aplicación de ejemplo que se utiliza en los tutoriales incluidos con la distribución del servidor de aplicaciones Weblogic 8.1. Es una aplicación pequeña que provee las funciones necesarias para administrar una base de datos de pacientes.

### 6.1.1. Antipatrones existentes

La primera parte de este proceso de verificación del enfoque consiste en analizar manualmente el código de la aplicación para detectar los antipatrones de desempeño presentes. A continuación, las siguientes subsecciones describen los antipatrones encontrados durante ese análisis.

#### 6.1.1.1. Acceso redundante por JNDI

Buscando dentro del código de la aplicación los usos de la interfaz JNDI para recuperar en forma remota las EJBHome, se encuentra sólo un uso de la misma.

**com.bea.medrec.utils.EJBHomeFactory:** es utilizada desde toda la aplicación para obtener los EJBHome, y como no implementa ningún caché debe hacer un acceso remoto ante cada solicitud que recibe.

#### 6.1.1.2. Múltiples accesos a EJBs por requerimiento

Leyendo el código de las clases que atienden directamente los requerimientos de los usuarios, se encuentran las siguientes ocurrencias de este antipatrón:

**com.bea.medrec.actions.PhysViewRecordAction:** realiza una solicitud a un EJB remoto *(PhysicianSessionEJB*), que a su vez realiza una invocación remota a *RecordSessionEJB*. Esa clase se ejecuta cuando desde la interfaz se genera un requerimiento a */physician/record.do* para ver los detalles de un paciente, y los métodos remotos que se ejecutan tienen la firma *getRecord(Integer id)*.

**com.bea.medrec.actions.SearchResultsAction:** utiliza el método remoto *PhysicianSessionEJB.-searchPatients(Search)*, que a su vez realiza una invocación remota al método *PatientSessionEJB.-findPatientByLastNameWild(String)*. Esta clase se activa cuando se recibe un requerimiento a */physician/searchresults.do*.

**com.bea.medrec.actions.CreateVisitAction:** invoca a dos métodos remotos en PhysicianSessionEJB, *addRecord(Record)* para guardar una nueva visita de un paciente en la base de datos, y *getRecordSummary(Integer)* para obtener el listado actualizado de las visitas que ha realizado el paciente. A su vez, cada uno de estos métodos invoca en forma remota a otro homónimo en *RecordSessionEJB*. El requerimiento que provoca que esta clase sea ejecutada es */visit.do*.

**com.bea.medrec.actions.ApprovePatientRequestAction:** se utiliza para aprobar un solicitud de un paciente, lo que provoca que se le actualice el estado, se le envíe un *e-mail*, y se recuperen las solicitudes que aún no han sido aprobadas. Cada una de estas 3 actividades es realizada por un método remoto diferente, por lo que se realizan 3 llamadas remotas a los método activateAccountStatus(String,Mail) y findNewUsers() de AdminSessionEJB, y otra llamada remota para enviar el mail con el método composeAndSendMail(Mail,Object[]) de MailSessionEJB. El requerimiento que provoca la ejecución de esta clase es */approve.do*.

#### 6.1.1.3. Envío de excesiva cantidad de objetos

Para encontrar este problema en forma manual se deben analizar todos los métodos que pueden ser invocados en forma remota para determinar la cantidad de objetos que va a retornar, considerando tanto algoritmos, cantidad de registros en la base de datos y características del negocio siendo analizado. En este caso se encuentran dos ocurrencias del problema:

**PatientSessionEJB.findPatientByLastNameWild(String pattern):** este método retorna todos los pacientes que cumplan con el patrón ingresado, que puede ser por ejemplo todos los que contengan una letra especificada por el usuario. Revisando en la base de datos, hay 207 pacientes cuyo apellido contiene la letra 'a', que serán retornados cuando un usuario invoque a este método con el patrón '*a*'. Y a medida que el sistema crezca en cantidad de pacientes, este problema se agravará más y más.

**PhysicianSessionEJB.searchPatients(Search vo):** este método invoca en forma remota al método anterior, y retorna los mismos resultados, por lo que sufre del mismo problema.

### 6.1.2. Antipatrones hallados automáticamente

Tras ejecutar la aplicación y los clientes Grinder para simular su uso, se recupera la información necesaria para que el comportamiento de la aplicación sea analizado por el motor de reglas. A continuación veremos cuáles son los antipatrones de desempeño que encontró la herramienta en forma automática, junto con el detalle que provee para su corrección.

#### 6.1.2.1. Acceso redundante por JNDI

Este antipatrón es reportado con la información que se puede ver en el Cuadro 6.1. La primer

**Cuadro 6.1:** Antipatrones en Avitek Medical Records: acceso redundante por JNDI.

| Recurso | Invocador |
|---|---|
| AdminSessionEJB.AdminSessionHome | lookupHome(EJBHomeFactory.java:111) |
| MailSessionEJB.MailSessionHome | lookupHome(EJBHomeFactory.java:111) |
| PatientSessionEJB.PatientSessionHome | lookupHome(EJBHomeFactory.java:111) |
| PhysicianSessionEJB.PhysicianSessionHome | lookupHome(EJBHomeFactory.java:111) |
| RecordSessionEJB.RecordSessionHome | lookupHome(EJBHomeFactory.java:111) |

columna muestra el nombre del recurso en el árbol JNDI, mientras que la segunda columna indica la línea de código desde donde se está accediendo al mismo. Si bien la información provista por la herramienta incluye nombres de paquetes y clases, acá son omitidos para mejorar la legibilidad. Podemos notar en esta segunda columna que todos los accesos al servidor JNDI se realizan desde la misma línea de código. Esto se debe a que la aplicación, como ya vimos al analizar su código, implementa el patrón de diseño Service Locator pero al no tener implementado un cache está haciendo requerimientos redundantes a través de la red para recuperar repetidas veces las mismas EJBHome. La solución a este problema, tal como sugiere el antipatrón, consiste en incorporar un cache a la clase EJBHomeFactory para acceder a los recursos listados. Tras agregar el cache, volver a instalar la aplicación y ejecutar el monitoreo, este antipatrón desaparece y no es detectado por el motor de reglas.

#### 6.1.2.2. Múltiples accesos a EJBs por requerimiento

Este antipatrón es reportado con la información presentada en el Cuadro 6.2. Este antipatrón fue encontrado 3 veces como se puede ver en el cuadro. El primer requerimiento accede a dos EJBs para recuperar los datos de un paciente. El primer EJB es *PhysicianSessionEJB*, y ése a su vez accede a *RecordSessionEJB*. La razón por la que se realiza esta indirección parece ser de prolijidad: los pacientes son administrados por el *RecordSessionEJB*, que está instalado en su propio contenedor, y la pantalla sólo accede al *PhysicianSessionEJB*. Por lo tanto, todos los requerimientos relacionados a pacientes que se hagan desde las pantallas de la aplicación médica sufrirán esta indirección.

El segundo requerimiento es idéntico al primero, con la única diferencia de que al EJB que accede es a *PatientSessionEJB*. En términos de eficiencia, esto significa que todos los registros

**Cuadro 6.2:** Antipatrones en Avitek Medical Records: múltiples accesos a EJBs por requerimiento.

| Requerimiento | Métodos de EJBs remotos accedidos |
|---|---|
| /physician/record.do | PhysicianSessionEJBImpl.getRecord(Integer)<br>RecordSessionEJBImpl.getRecord(Integer) |
| /physician/searchresults.do | PhysicianSessionEJBImpl.searchPatients(Search)<br>PatientSessionEJBImpl.findPatientByLastNameWild(String) |
| /physician/visit.do | PhysicianSessionEJBImpl.addRecord(Record)<br>PhysicianSessionEJBImpl.getRecordsSummary(Integer)<br>RecordSessionEJBImpl.addRecord(Record)<br>RecordSessionEJBImpl.getRecordsSummary(Integer) |

de pacientes serán serializados y transportados por la red el doble de veces de lo estrictamente necesario.

El último de los requerimientos identificados que presenta este antipatrón está agravado por el hecho de que utiliza dos métodos remotos, uno para agregar una visita de un paciente y el otro para recuperar la lista completa de visitas. Se realizan 4 invocaciones remotas. La solución sugerida por el antipatrón es agregar un nuevo SessionEJB que pueda realizar las dos funcionalidades juntas. Por ejemplo, se podría agregar el método *addRecordAndGetRecordsSummary(Record, Integer)* en el *RecordSessionEJB*, que acceda a los dos métodos requeridos en forma local, permitiendo que con un solo acceso remoto se resuelvan las dos funcionalidades.

Al aplicar la solución sugerida por el antipatrón se reduce el problema de las 4 invocaciones remotas a sólo 2, como en las otras ocurrencias del antipatrón. Para estos casos la solución recomendada no es suficiente y se deben buscar alternativas. Una opción es que las páginas utilicen los EJBs de los registros y pacientes directamente, pero en este caso se pierde la separación conceptual entre aplicaciones, dificultando quizás el mantenimiento. Otra alternativa es que las aplicaciones de Pacientes, Registros y Médicos se instalen juntas, de manera que esas invocaciones puedan ser realizadas en forma local. Con esta última opción, se dificulta ligeramente la actualización de versiones de los componentes, que ya no podría hacerse por separado. En este caso se está cambiando desempeño por facilidad de mantenimiento. Es importante destacar que en estos casos los tiempos de respuesta deben ser priorizados únicamente cuando los problemas de desempeño existen y no se pueden solucionar de otra manera, ya que otros atributos de calidad pueden ser aún más importantes.

A los efectos de esta evaluación, se opta por modificar los clientes del EJB, que son PhysViewRecordAction, SearchResultsAction y CreateVisitAction, para que accedan directamente al RecordSessionEJB en vez de a PhysicianSessionEJB, en todos los usos de esa clase. Al volver a monitorear la aplicación con estas refactorizaciones aplicadas, la herramienta no detecta la ocurrencia de este antipatrón, comprobando que se han logrado eliminar las indirecciones innecesarias.

#### 6.1.2.3. Envío de excesiva cantidad de objetos

Este antipatrón es reportado con la información que se puede ver en el Cuadro 6.3. Se puede

**Cuadro 6.3:** Antipatrones en Avitek Medical Records: envío de excesiva cantidad de objetos.

| Recurso | Invocador |
|---|---|
| AdminSessionEJB.AdminSessionHome | lookupHome(EJBHomeFactory.java:111) |
| MailSessionEJB.MailSessionHome | lookupHome(EJBHomeFactory.java:111) |
| PatientSessionEJB.PatientSessionHome | lookupHome(EJBHomeFactory.java:111) |
| PhysicianSessionEJB.PhysicianSessionHome | lookupHome(EJBHomeFactory.java:111) |
| RecordSessionEJB.RecordSessionHome | lookupHome(EJBHomeFactory.java:111) |

ver que el motor de reglas encontró los dos métodos que habíamos identificado en nuestro análisis previo como ocurrencias de este antipatrón. Antes de aplicar el refactor recomendado por el antipatrón, conviene primero analizar cuál es la funcionalidad que proveen. Los dos métodos corresponden a la pantalla en que se buscan pacientes por apellido con un caracter comodín. La pantalla no permite navegar entre resultados paginados, por lo que todos los pacientes son mostrados de una sola vez. El consumo de red y de procesador es elevado y posiblemente más de lo necesario, ya que la intención de la pantalla es seleccionar un paciente. Al no estar limitada la cantidad de pacientes que se pueden retornar con cada una de estas llamadas, se corre el riesgo de que una consulta genere una respuesta más grande que la capacidad del servidor (ya sea memoria, velocidad de transferencia por la red, velocidad del procesador) provocando la parálisis del mismo.

La solución sugerida para este antipatrón es aplicar el patrón de diseño ValueListHandler y agregar paginado en el cliente. De esta manera, aunque haya millones de pacientes que reunan las características requeridas, sólo se enviarán al cliente en grupos pequeños, garantizando la disponibilidad del servidor. Sin duda alguna esta solución impide la reaparición del problema, pero una solución más simple y menos costosa que vamos a implementar en este trabajo consiste en validar que en la pantalla se ingresen al menos 3 caracteres además de los comodines. De esta manera se impide la ejecución de consultas que retornen demasiados objetos.

Antes de volver a monitorear la aplicación, y como se modificó al cliente para resolver este problema y no el código de la misma, se debe modificar el *script* Grinder para que en vez de buscar los pacientes con el patrón '*a*', la ejecute con el patrón de búsqueda '*har*'. La cantidad de pacientes que incluyen 'har' en su apellido son sólo cinco, por lo que sólo se mostrarán cinco pacientes. En otras palabras, simulamos que el tamaño de página es 5. Efectivamente, al monitorear la aplicación con el nuevo *script* Grinder, este antipatrón no es identificado.

### 6.1.3. Antipatrones no encontrados por el monitoreo

Cuando analizamos manualmente la aplicación habíamos encontrado la ocurrencia de un antipatrón en la funcionalidad de aprobar solicitudes de usuarios, pero no fue detectado

por el motor de reglas. Esto se debe a que para realizar esa acción se requieren permisos de administrador, y el usuario con que se ejecutaron las pruebas no lo era, por lo que nunca se aprobaron solicitudes mientras la aplicación era monitoreada. Para verificar que el problema es detectado automáticamente si se ejecuta esa funcionalidad, nos logueamos en la aplicación como administrador y se vuelve a monitorear ese fragmento de aplicación mientras se aprueba una solicitud, logrando que las reglas lo identifiquen generando el reporte que se muestra en el Cuadro 6.4.

**Cuadro 6.4:** Antipatrón no encontrados en la primera prueba: múltiples accesos a EJBs por requerimiento.

| Requerimiento | Métodos de EJBs remotos accedidos |
|---|---|
| /admin/approve.do | AdminSessionEJBImpl.activateAccountStatus(String, com.bea.medrec.value.Mail) |
| " | AdminSessionEJBImpl.findNewUsers() |
| " | MailSessionEJBImpl.composeAndSendMail(com.bea.medrec.value.Mail, Object[]) |

Este reporte muestra que cada vez que se aprueba la solicitud de un paciente, se están realizando 3 invocaciones remotas, lo cual coincide con los resultados del análisis manual. Si las tres actividades se ejecutaran desde un SessionEJB como recomienda la herramienta, se evitarían 2 invocaciones remotas innecesarias.

### 6.1.4. Análisis del proceso sobre Avitek Medical Records

En la Figura **6.1** se pueden ver los resultados obtenidos en las diferentes pruebas. Los tiempos que se muestran son obtenidos por Grinder utilizando el método currentTimeMillis(), provisto por la biblioteca de utilidades de sistema que acompaña a Java, que retorna la diferencia en milisegundos entre el momento actual y la medianoche del 1 de Enero de 1970. Cabe aclarar que este servicio tiene una precisión de entre 10 y 15 milisegundos en Windows [45], que es el sistema operativo utilizada para realizar estas pruebas. Al hacer 100 pruebas y promediar los resultados esa falta de precisión no es significativa, y sólo puede notarse en páginas que tienen un costo promedio de 0 ms, como ocurre en la medición correspondiente a la página 18. El 0 en este caso debe interpretarse como que las 100 veces que se cargó la página el servidor se demoró menos de 15 ms.

Si comparamos los tiempos de respuesta de cada versión con y sin monitoreo activado, veremos que en gran parte de las páginas no tiene un impacto negativo. Esto se debe a que los proxies sólo se utilizan para decorar los EJBObjects, haciendo que el resto de la aplicación conserve su desempeño normal. Las páginas que presentan la mayor diferencia corresponden a la funcionalidad de búsqueda de pacientes. La razón por la cual recuperar un listado de pacientes tiene un costo mayor al ser monitoreado es que este requerimiento provoca que un EJB retorne muchos objetos (lista de usuarios) haciendo que el proxy dinámico recorra un listado para determinar si hay EJBs dentro del mismo.

Si comparamos los tiempos de respuesta de la aplicación antes y después de la optimización, se destacan las páginas 1, 9, 13, 17, 23, 24 y 30 que en su versión sin optimizar se demoran entre 4 y 7 segundos, y corresponden a las búsquedas de pacientes. La mejora obtenida con el proceso de optimización para estas pantallas fue en promedio del 79 %, debido a que se

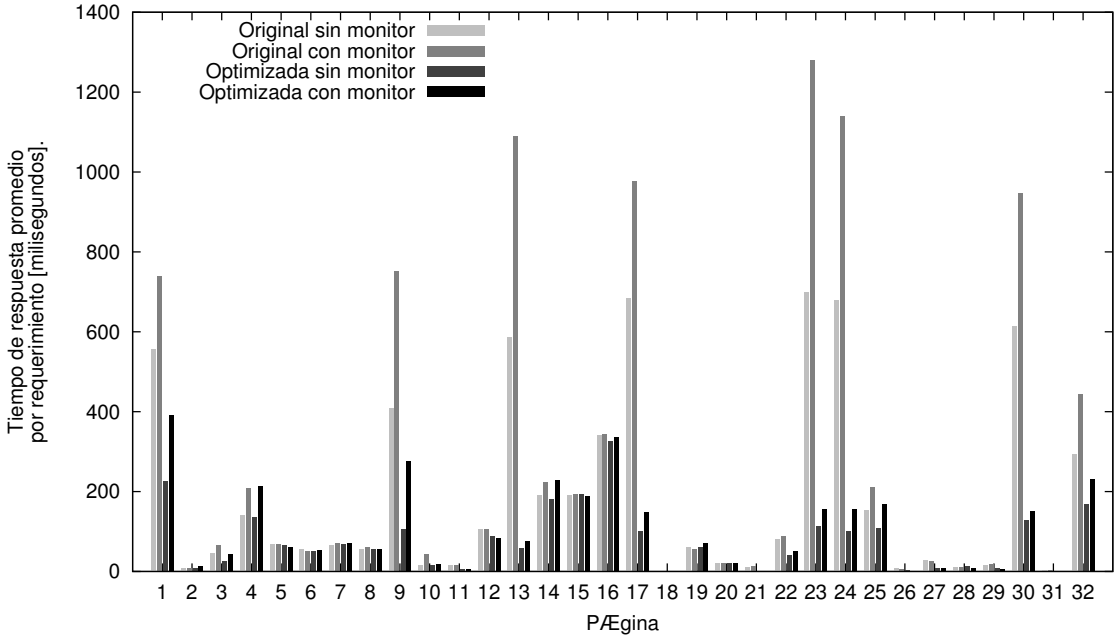


**Figura 6.1: Comparación de tiempos de respuesta de MedRec con y sin monitoreo activado.**

retornan muchos menos pacientes en la versión optimizada. Otras páginas que han tenido una mejora notable son la 11, 21 y 27, correspondientes al alta de visitas de pacientes en que se detectó la ocurrencia del antipatrón "Múltiples accesos a EJBs por requerimiento". Recordemos que antes de la optimización esta operación realizaba 4 invocaciones remotas, que se redujeron a 1 en la versión optimizada, logrando una mejora promedio de estas páginas del 77 %. Por último, las páginas de visualización de detalles de un paciente también han tenido una mejora significativa, debido a que la versión optimizada hace una llamada remota menos. Las páginas correspondientes son la 22 y la 31, que en promedio han reducido el tiempo de respuesta en un 65 %.

En la Figura 6.2 se puede ver una comparación entre los tiempos de respuesta de las dos versiones de la aplicación, además del costo de tener el monitor activado. En este caso los tiempos medidos corresponden a la suma de los promedios visualizados en la Figura 6.1, que podrían ser interpretados como el tiempo promedio de una sesión de un usuario. De acuerdo a las mediciones obtenidas, el costo de activar el monitoreo en la aplicación original es de un 49 %, mientras que en la versión optimizada ese costo se reduce a 32 %. El costo del monitoreo se reduce al optimizar la aplicación porque al hacerlo se disminuye la cantidad de EJBs remotos que se utilizan, y son los que deben ser monitoreados ante cada invocación. Otro dato importante que surge de las pruebas es que la versión optimizada es un 60 % más rápida que la versión original. Esto se debe principalmente a que se reduce la cantidad de objetos que se transfieren en la consulta de pacientes, y se reducen a la mitad las invocaciones remotas en casi todas las operaciones.

La herramienta detectó 6 de los 7 antipatrones conocidos en la aplicación. Además de la correcta detección, se "adaptaron" satisfactoriamente todas las refactorizaciones sugeridas. Esto muestra que la efectividad de la herramienta para esta aplicación fue del 86 %. Ahora

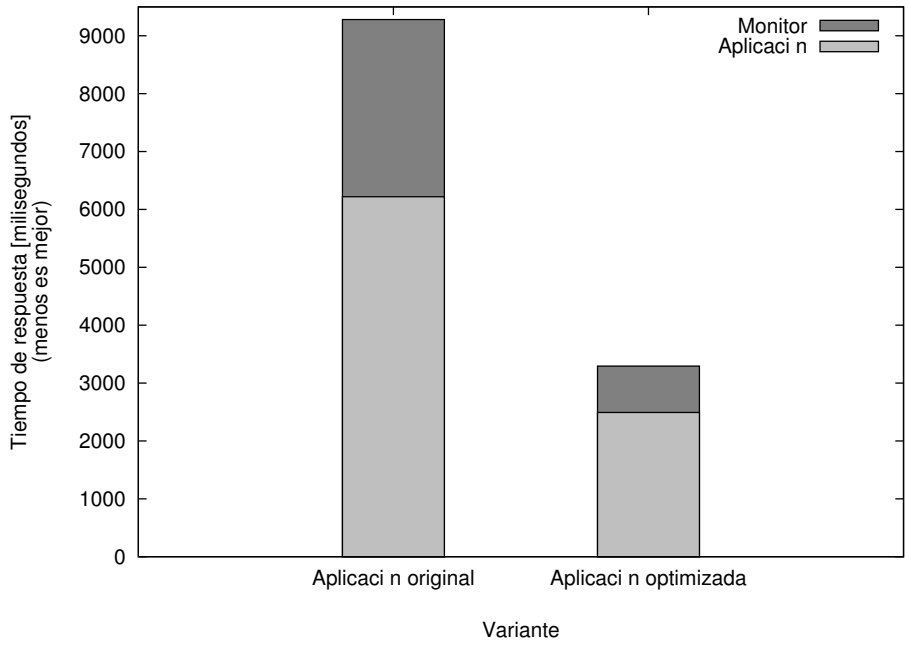


**Figura 6.2:** Comparación de tiempos de respuesta en MedRec.

veremos otra aplicación sobre la que hemos probado la efectividad de este enfoque.

## 6.2. Java Petstore 1.3.1

Java Petstore es una aplicación de referencia desarrollada por Sun Microsystems para demostrar las capacidades que ofrece la plataforma J2EE, con respecto al desarrollo de aplicaciones de negocio robustas, escalables, portables y fácilmente mantenibles. La aplicación consiste en un sitio Web en que el cliente puede registrarse, administrar su carro de compras, y generar una orden para la compra de mascotas, que se instala en el servidor de aplicaciones *Sun Java Application Server* [40].

### 6.2.1. Antipatrones existentes

Al analizar el código fuente de la aplicación se encontraron los antipatrones de *performance* que se detallan a continuación.

#### 6.2.1.1. Acceso redundante por JNDI

Buscando los accesos remotos a través de JNDI se encuentra que no se realizan a través de un cache, por lo que se identifica la ocurrencia de este problema. Las clases que utilizan esta interfaz son:

**AccountEJB:** cada vez que se crea un nuevo EJB de tipo Cuenta se crean automáticamente dos nuevos EJB, uno de tipo Tarjeta de Crédito y otro de tipo Información de Contacto. Para crear los EJBs, de acuerdo a la especificación J2EE se necesitan las EJBHomes correspondientes, que en este caso se recuperan del servidor JNDI.

**CustomerEJB:** cada vez que se crea un nuevo EJB de tipo cliente, se le crean automáticamente dos EJBs, uno de tipo Perfil y otro de tipo Cuenta.

**ProcessManagerEJB:** utiliza un ManagerHomeEJB que recupera del servidor JNDI cada vez que se crea una nueva instancia.

**SignOnEJB:** utiliza el EJBHome de los usuarios para recuperar la clave del usuario y determinar si puede ingresar al sistema.

**CreateUserServlet:** utiliza el EJBHome del SignOnEJB para crear nuevos usuarios.

**SignOnFilter:** utiliza el EJBHome del SignOnEJB para autenticar al usuario que quiere ingresar al sistema.

**ServiceLocator:** implementa el patrón Service Locator pero sin cache.

#### 6.2.1.2. Envío de excesiva cantidad de objetos

**OPCAdminFacadeEJB.getOrdersByStatus(String status):** este método retorna todas las órdenes de compra que están en un estado determinado. El conjunto de órdenes en un estado es potencialmente grande, por lo que este método podría presentar problemas al transportar demasiada información. La aplicación invoca a este método cuando un administrador se loguea, mostrando en pantalla ordenes que están pendientes de aprobación o rechazo.

**PurchaseOrderEJB.getAllItems():** este método retorna un objeto java de tipo LineItem por cada ítem de la orden de compra. Si las órdenes contienen una gran cantidad de líneas, se puede presentar este problema. El método es llamado cada vez que se muestra por pantalla el carro de compras, ya sea para agregar un nuevo ítem, modificar la cantidad de alguno ya agregado, o hacer el checkout para generar la orden de compra.

### 6.2.2. Antipatrones hallados automáticamente

Al ejecutar la aplicación con el monitoreo encendido, se obtiene el archivo XML que permite al motor de reglas detectar los antipatrones presentes en la aplicación. A continuación se analizan los resultados generados por la herramienta.

#### 6.2.2.1. Acceso redundante por JNDI

Este antipatrón es reportado con la información que se muestra en el Cuadro 6.5. La primer columna muestra el nombre del recurso en el árbol JNDI, mientras que la segunda columna indica la línea de código desde donde se está accediendo al mismo. El reporte generado

**Cuadro 6.5:** Antipatrones en Petstore: acceso redundante por JNDI.

| Recurso | Invocador |
|---|---|
| java:comp/env/ejb/ContactInfo | ejbPostCreate(AccountEJB.java:85) |
| | getLocalHome(ServiceLocator.java:93) |
| java:comp/env/ejb/CreditCard | ejbPostCreate(AccountEJB.java:88) |
| java:comp/env/ejb/Account | ejbPostCreate(CustomerEJB.java:78) |
| java:comp/env/ejb/Profile | ejbPostCreate(CustomerEJB.java:82) |
| java:comp/env/ejb/Manager | ejbCreate(ProcessManagerEJB.java:97) |
| java:comp/env/ejb/User | ejbCreate(SignOnEJB.java:63) |
| java:comp/env/ejb/SignOn | getSignOnEjb(SignOnFilter.java:214) |
| | getSignOnEjb(CreateUserServlet.java:91) |
| | getLocalHome(ServiceLocator.java:93) |
| java:comp/env/ejb/ShoppingCart | getLocalHome(ServiceLocator.java:93) |
| java:comp/env/ejb/Address, | getLocalHome(ServiceLocator.java:93) |
| java:comp/env/ejb/Customer | getLocalHome(ServiceLocator.java:93) |
| java:comp/env/ejb/PurchaseOrder | getLocalHome(ServiceLocator.java:93) |
| java:comp/env/ejb/ProcessManager | getLocalHome(ServiceLocator.java:93) |
| java:comp/env/ejb/Catalog | getLocalHome(ServiceLocator.java:93) |
| java:comp/env/ejb/ShoppingClientFacade | getLocalHome(ServiceLocator.java:93) |

por la herramienta incluye nombres de paquetes y clases, que han sido omitidos aquí para facilitar la legibilidad.

Podemos notar en esta segunda columna que varios accesos al servidor JNDI se realizan desde la misma línea de código. Esto se debe a que parte de la aplicación implementa el patrón de diseño Service Locator para beneficiarse de la mantenibilidad que ofrece, pero no tiene implementado un cache por lo que está haciendo requerimientos redundantes a través de la red para recuperar repetidas veces las mismas EJBHome. La solución a este problema que sugiere el antipatrón es agregar un ServiceLocator con cache que debe utilizarse siempre en vez de acceder al servidor de directorios directamente. En este caso en que la clase ServiceLocator ya existe, el problema se soluciona incorporándole un cache para acceder a los recursos, y refactorizar todas las clases listadas para utilizar el ServiceLocator como único punto de acceso.

Las clases de la segunda columna son exactamente las mismas que se identificaron manualmente: ServiceLocator, AccountEJB, CustomerEJB, ProcessManagerEJB, SignOnEJB, CreateUserServlet y SignOnFilter. Después de refactorizar estas clases, una nueva ejecución de la herramienta no detecta la ocurrencia de este problema.

#### 6.2.2.2. Envío de excesiva cantidad de objetos

La herramienta detecta la ocurrencia de este antipatrón y la reporta con la información presente en el Cuadro 6.6. En el cuadro se puede ver que el método remoto getOrderByStatus(String) de la clase ProcessManagerEJB retorna 221 *Transfer Objects* con la información de las órdenes de compra encontradas en la base de datos. El reporte generado por la herramienta incluye nombres de paquetes y clases, que han sido omitidos aquí para facilitar la legibilidad.

**Cuadro 6.6:** Antipatrones en Petstore: envío de excesiva cantidad de objetos.

| Método que retorna muchos objetos | Cantidad y tipo de resultado |
|---|---|
| OPCAdminFacadeEJB.getOrdersByStatus(String status) | 221 OrdersTO |

Este método es utilizado por el administrador cuando quiere revisar las órdenes en estado Pendiente, para pasarlas al estado Aprobada o Rechazada. Como la interfaz de usuario no está preparada para mostrar parte del resultado, deben enviarse todas las órdenes de compra en un solo requerimiento. La solución que sugiere la herramienta es aplicar el patrón de diseño ValueListHandler, lo que implica modificar la interfaz de usuario para que muestre las órdenes en forma paginada. Una alternativa menos costosa que puede ser utilizada en este caso, es siempre mostrar las primeras 10 órdenes en estado Pendiente. Al ir aceptándolas o rechazándolas, cambian de estado y el conjunto de órdenes que se retornarán la siguiente vez que se refresque la página es distinto, permitiendo cumplir con las necesidades del usuario sin impactar negativamente en el consumo de recursos de la aplicación ni requiriendo una refactorización demasiado costosa.

Después de limitar la cantidad de órdenes que se retornan en cada consulta, este antipatrón no es detectado por un segundo análisis de la herramienta.

### 6.2.3. Antipatrón no hallado automáticamente

De todas las ocurrencias de antipatrones que detectamos inicialmente al revisar el código de la aplicación, todas fueron detectadas por la herramienta exceptuando una. La ocurrencia que no fue detectada corresponde al método getAllItems de la clase PurchaseOrderEJB, que habíamos determinado que retornaba demasiados objetos porque no imponía ningún límite y retornaba todos los objetos que existieran en una orden de compra. Durante las pruebas realizadas no fue posible generar una orden de compra con suficientes ítems porque no lo permitían los datos existentes en la base de datos: una orden de compra no puede tener el mismo ítem en más de una línea (una línea contiene ítem y cantidad), y en la base de datos existían sólo 17 ítems a la venta.

### 6.2.4. Análisis del proceso sobre Petstore

Sobre esta aplicación se realizó el proceso descripto. Primero se grabó el *script* que crea un usuario, navega por la aplicación, genera órdenes de compra, se loguea un administrador y solicita ver las órdenes de compra generadas. Teniendo el *script*, se midieron los tiempos de respuesta de los requerimientos al ejecutarlo sobre la aplicación original, y posteriormente sobre la aplicación con el monitoreo activado. Después de corregir los problemas encontrados por la herramienta, se volvieron a tomar los tiempos de respuesta con y sin el monitoreo activado.

En la Figura 6.3 se pueden ver los resultados obtenidos en las diferentes pruebas. Los tiem-

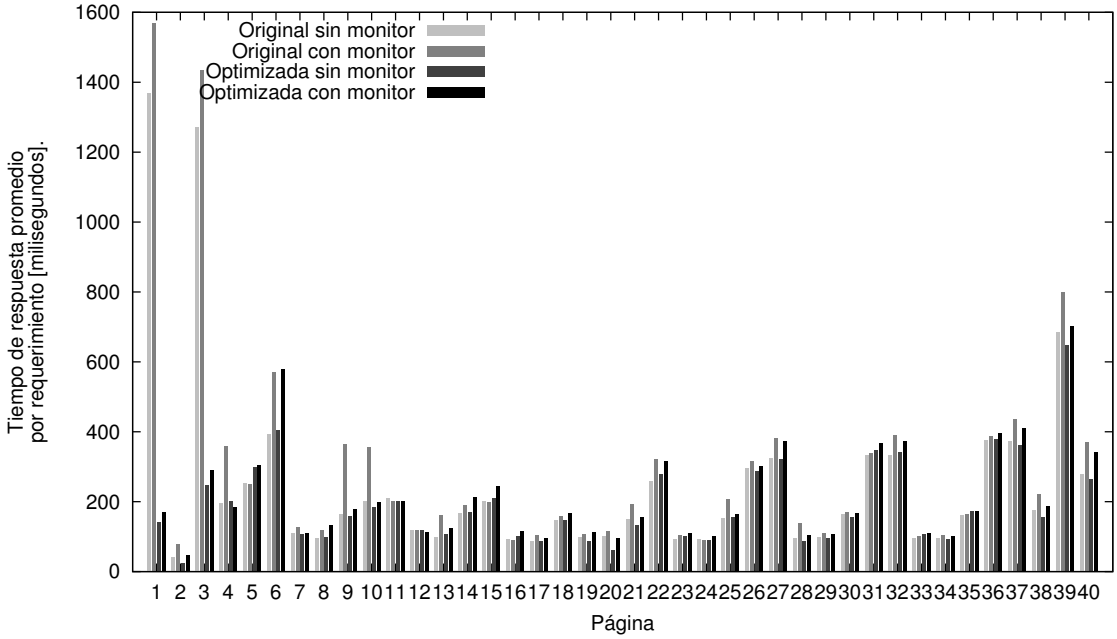


**Figura 6.3:** Comparación de tiempos de respuesta de Petstore con y sin monitoreo activado.

pos que se muestran en esta figura corresponden al promedio de los 100 clientes simulados con Grinder. Vale destacar que la página para mostrar todas las órdenes en estado Pendiente –concretamente, las páginas número 1 y 3–, ha mejorado en forma notoria. Corregir el

antipatrón asociado, "Envío de excesiva cantidad de objetos", produce grandes beneficios porque el ahorro de recursos ocurre en las tres capas de la aplicación: la base de datos debe recuperar menos registros, la capa de negocio debe procesar menos objetos, y la capa Web debe generar mucho menos texto para mostrar al usuario. Gráficamente, esto se refleja en las barras cuyo eje x es igual a 1 y 3. Se puede ver que otros requerimientos que consumen más recursos que el promedio no han mejorado en forma significativa. Específicamente los requerimientos 6, 26, 27, 31, 32, 36 y 37 corresponden a operaciones de grabado de la orden de compra, que no se ven beneficiadas por ninguna optimización. Otro requerimiento que se destaca por consumir más recursos que el resto es la página 39, que corresponde a la generación de una orden de compra. Nuevamente, esta operación no se ve beneficiada por ningún refactor en particular, por lo que la ganancia obtenido en ese caso no es significativa. La Figura 6.4 compara el desempeño de la aplicación antes y después de la optimización,

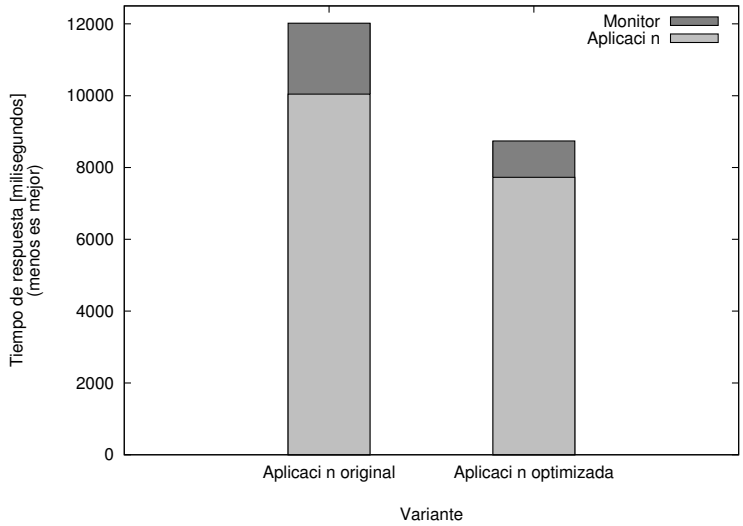


**Figura 6.4:** Comparación de tiempos de respuesta en Petstore.

tomando como referencia la suma de los promedios de cada requerimiento. Se puede ver que la aplicación optimizada se demora un 23 % menos en responder a los requerimientos que la aplicación original, comprobando que las refactorizaciones recomendadas por la herramienta impactan positivamente en el desempeño de la aplicación. Además, en el mismo diagrama se puede observar el *overhead* provocado por el monitoreo, que es del 20 % sobre la aplicación original y del 13 % sobre la aplicación optimizada. La razón por la que el monitor consume menos tiempo sobre la aplicación optimizada es porque tiene menos actividad, ya que se reducen ampliamente las invocaciones remotas que es lo único que se monitorea.

## 6.3. Libra Book Store

Libra es una aplicación Web de código abierto que permite la venta de libros a través de Internet. Fue creada con el propósito de ejemplificar buenas prácticas de arquitectura y diseño para el desarrollo de aplicaciones J2EE utilizando el *framework Struts* [17] y el servidor de

aplicaciones JBOSS [10]. Libra es una aplicación Web que permite el registro de usuarios, la búsqueda de libros y la creación de órdenes de compra.

### 6.3.1. Antipatrones existentes

El primer paso consiste en analizar manualmente el código fuente de la aplicación. Durante este análisis se encontraron los antipatrones de *performance* que se detallan en las siguientes subsecciones.

#### 6.3.1.1. Acceso redundante por JNDI

Buscando los accesos remotos a través de la interfaz JNDI en el código fuente de la aplicación se encuentra que no se realizan a través de un cache, por lo que se identifica la ocurrencia de este problema. Las clases que utilizan esta interfaz son:

**RegistrationAction:** para dar de alta a un nuevo usuario de la aplicación se utiliza al EJB RegisterUserLocal, que es accedido a través de su home, RegisterUserHome, que se obtiene con el servicio de JNDI.

**LoginAction:** cuando se ingresa *nombre de usuario* y *clave* en la pantalla de inicio de la aplicación, el servlet recupera el LoginVerifierHome para autenticar al usuario que quiere utilizar el sistema.

**SearchAction:** al buscar los libros a la venta, el sistema utiliza el SearchBookHome obtenido a través del servicio de directorios para recuperar los libros de la base de datos.

#### 6.3.1.2. Envío de excesiva cantidad de objetos

**SearchBookLocal.search(String subcadena):** este método retorna todos las libros cuyo título contiene el fragmento de texto pasado como parámetro. Si se envía una subcadena muy pequeña (eje. "a") o muy común (eje. "*Introducción a*" ), la cantidad de libros que deben ser retornados puede ser muy alta. El método es ejecutado desde la pantalla principal de la aplicación, en que el usuario ingresa una cadena de texto y presiona el botón "Buscar".

### 6.3.2. Antipatrones hallados automáticamente

El siguiente paso del proceso de evaluación requiere ejecutar la aplicación con el monitoreo encendido, con lo que obtenemos el archivo XML que permite al motor de reglas detectar los antipatrones presentes en la aplicación. A continuación se analizan los resultados generados automáticamente por la herramienta.

**Cuadro 6.7:** Antipatrones en Libra: acceso redundante por JNDI.

| Recurso | Invocador |
|---|---|
| SearchBookBean | execute(SearchAction.java:58) |
| RegisterUserBean | execute(RegistrationAction.java:57) |
| LoginVerifierBean | execute(LoginVerifierAction.java:59) |

#### 6.3.2.1. Acceso redundante por JNDI

Recordemos que este antipatrón es reportado informando el nombre del recurso en el árbol JNDI y la línea de código desde donde se accede al mismo. En el Cuadro 6.7 se puede ver el contenido del reporte generado por la herramienta, de donde se han eliminado los nombres de paquetes y clases para facilitar la legibilidad. Podemos notar en la segunda columna, correspondiente a las líneas de código fuente en que se recuperan los objetos del servicio de directorios, que todos los accesos son desde clases con el sufijo *Action*. Esto se debe a que la aplicación utiliza el *framework* Struts, donde las clases *Action* son los controladores de la aplicación, que implementa el patrón arquitectónico *Model View Controller* [39]. El cliente sólo interactúa con estas clases de control, que son las responsables de acceder a los EJB que contienen la lógica de negocio. El *framework* Struts establece que los controladores implementan el patrón de diseño Command [13], por lo que todos los métodos tienen el mismo nombre: *execute*.

La solución a este problema que sugiere el antipatrón es agregar un ServiceLocator con cache que tiene la responsabilidad de acceder al servidor de directorios, y es la única clase en toda la aplicación que puede utilizar la interfaz JNDI para recuperar los EJBHome. En este caso se aplica la solución tal cual la describe el antipatrón, ya que no existe una clase para tal función en el código como ocurría en las aplicaciones anteriormente analizadas.

Las clases que están provocando la ocurrencia de este patrón son las que están listadas en la segunda columna, y coinciden con las encontradas manualmente al inicio de este análisis: RegistrationAction, LoginVerifierAction y SearchAction. Después de agregar la clase ServiceLocator con caché de EJBHome, y refactorizar esas 3 clases para que utilicen el ServiceLocator en vez de acceder directamente al servicio de directorios, los antipatrones ya no son detectados en los posteriores monitoreos de la aplicación.

#### 6.3.2.2. Envío de excesiva cantidad de objetos

La herramienta detecta la ocurrencia de este antipatrón y la reporta con la información presente en el Cuadro 6.8, donde se puede ver que el método remoto search(String) de la clase

**Cuadro 6.8:** Antipatrones en Libra: envío de excesiva cantidad de objetos.

| Método que retorna muchos objetos | Cantidad y tipo de resultado |
|---|---|
| SearchBookBean.search(String subcadena) | 386 Book |

SearchBookBean retorna 386 objetos de tipo *Book*. La clase *business.ejb.general.Book* no es un EJB, sino que es una clase Java simple que se utiliza para enviar la información de los libros encontrados en la base de datos. Si bien durante el monitoreo se realizó está búsqueda varias veces con diferentes parámetros, en el reporte aparece sólo una vez, ya que a los efectos de solucionar el problema no es relevante cuántas veces o con qué parámetros fue invocado. Los 386 libros que encuentra corresponde a la búsqueda de la cadena "a". Otras búsquedas se han hecho con otras cadenas y han generado distinta cantidad de resultados, siendo el máximo de 400 y corresponden a la búsqueda de la cadena vacía (""). Como ya aclaramos anteriormente, el reporte generado por la herramienta incluye los nombres de paquetes y clases que aquí se omiten para facilitar la legibilidad.

Este método remoto es utilizado para listar los libros que pueden ser comprados en el sitio, así el cliente puede seleccionar el que le interesa. Como la interfaz de usuario no está preparada para mostrar sólo una parte del resultado, se debe enviar la información de todos los libros que reunen la característica buscada en un solo requerimiento, lo que provoca la ocurrencia de este antipatrón. La solución que sugiere el antipatrón consiste en implementar el patrón de diseño ValueListHandler, para lo que hay que modificar la interfaz de usuario permitiendo que se naveguen los resultados en forma paginada. También se debe modificar la firma del método de búsqueda para que pueda retornar todos los resultados de a una página por vez. A los efectos de este trabajo aplicaremos una solución alternativa, la cual ya hemos utilizado en MedRec, consistente en agregar una validación más en la pantalla para impedir el ingreso de una subcadena de texto muy breve, que potencialmente puede generar demasiados resultados. Obligando a que se ingresen al menos 5 caracteres en el campo de búsqueda, los conjuntos de resultados se reducen significativamente, lo que es comprobado al monitorear la aplicación y observar que este antipatrón ya no es detectado.

### 6.3.3. Análisis del proceso sobre Libra

Sobre esta aplicación se realizó el proceso descripto. Primero se grabó el *script* que crea un usuario y navega por la aplicación buscando libros y y realizando compras. Teniendo el *script*, se midieron los tiempos de respuesta de los requerimientos al ejecutarlo sobre la aplicación original, y posteriormente sobre la aplicación con el monitoreo activado. Después de corregir los problemas encontrados por la herramienta, se volvieron a tomar los tiempos de respuesta con y sin el monitoreo activado.

En la Figura 6.5 se pueden ver los resultados obtenidos en las diferentes pruebas. Los tiempos que se comparan en este gráfico corresponden al tiempo promedio de respuesta de cada requerimiento, en las 100 corridas. Como en el caso anterior, se destacan las páginas que presentan el antipatrón "Envío de excesiva cantidad de objetos" y corresponden en esta oportunidad a las búsquedas que retornan demasiados libros, concretamente las páginas 11, 13, 15, 26, 28 y 36. El tiempo de respuesta disminuyó en promedio un 61 % debido a que la cantidad de libros retornada en cada búsqueda fue menor.

La Figura 6.6 compara el desempeño de la aplicación antes y después de la optimización. Aquí el tiempo que se compara es la suma de los promedios utilizados en la figura anterior, que representan una sesión promedio del uso de la aplicación. Se puede ver que el tiempo de respuesta promedio de la aplicación optimizada es 39 % menor que la aplicación original,

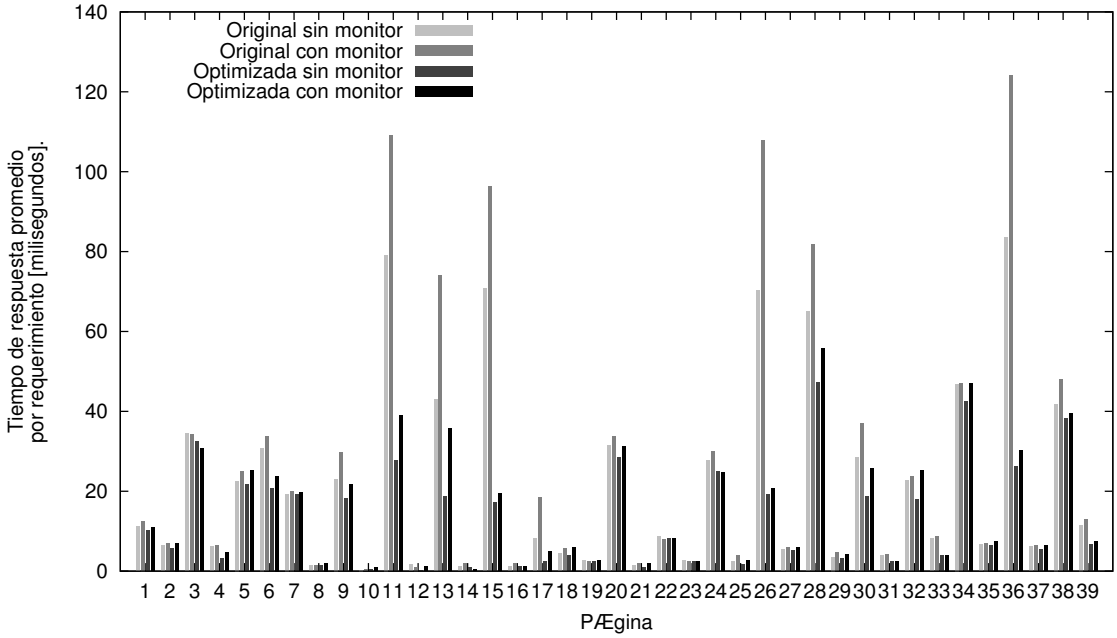


**Figura 6.5:** Comparación de tiempos de respuesta de Libra con y sin monitoreo activado.

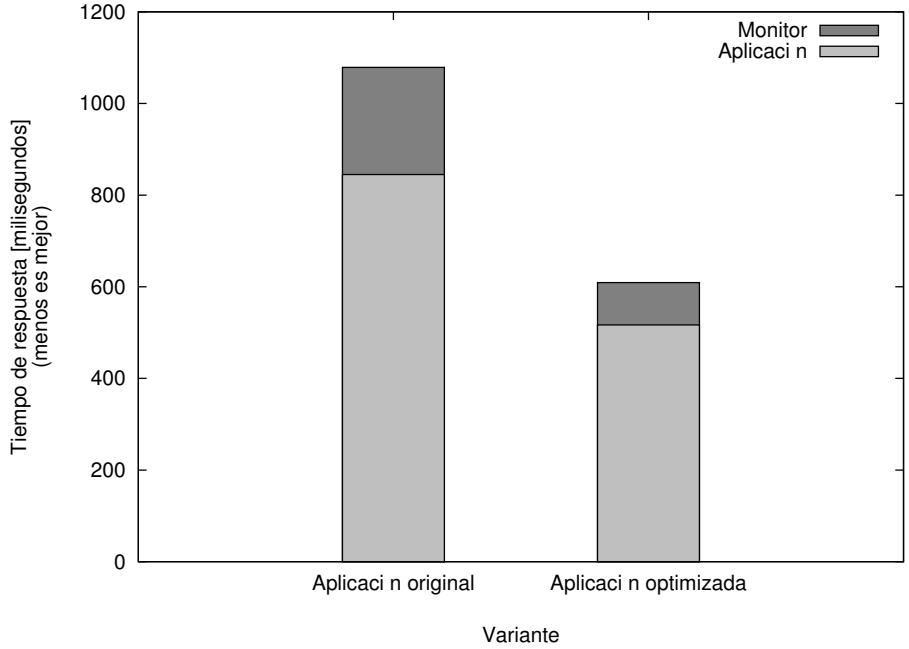


**Figura 6.6:** Comparación de tiempos de respuesta en Libra.

comprobando que aplicar las refactorizaciones recomendadas por la herramienta efectivamente mejoran la *performance*. Además, en la misma figura se puede observar el *overhead* provocado por el monitoreo, que es de 28 % sobre la aplicación original y del 18 % sobre la aplicación optimizada. Como ya habíamos encontrado en las aplicaciones anteriores, el monitor consume más recursos cuando la aplicación tiene más problemas de desempeño. En este caso particular, las respuestas incluyen muchos menos objetos, por lo que el monitor se evita el costo de analizar cada uno de esos objetos retornados. Esto es válido en carácter orientativo, dado que si se aceptan *wildcards* esta alternativa seguirá sufriendo del antipatrón. En cambio, la alternativa sugerida por la herramienta erradica el problema definitivamente.

## 6.4. Conclusiones

En este capítulo hemos analizado la efectividad de la herramienta para detectar antipatrones, demostrando mediante experimentos que la herramienta encontró más del 85 % de los antipatrones conocidos en las tres aplicaciones analizadas.

También hemos comprobado que la herramienta es portable, al menos, entre tres diferentes servidores de aplicaciones. La hemos utilizado en aplicaciones instaladas en Weblogic Application Server con Java 1.4, Sun Java Application Server con Java 1.3, y JBOSS Application Server con Java 5, exactamente de la misma manera. Esto se debe a que no utiliza ninguna característica propia de los servidores de aplicaciones, sino que se utiliza tecnología estándar Java disponible desde la versión 1.3 de la máquina virtual Java. Por lo tanto, la aplicación se debería poder utilizar sobre cualquier servidor de aplicaciones compatible con Java 1.3 o versiones siguientes.

Respecto a las refactorizaciones realizadas, casi en ninguno de los casos fue exactamente la recomendada, por motivos de mantenibilidad. Para los casos en que se debía agregar una nueva clase llamada ServiceLocator, dado que las aplicaciones analizadas ya implementaban de alguna manera ese patrón, habríamos tenido dos clases con la misma funcionalidad y un nivel de indirección innecesario. En los casos en que se requería implementar el patrón de diseño *Value List Handler* para paginar los resultados y evitar retornar demasiados objetos, se logró el mismo objetivo de maneras más sencilla que, dada la funcionalidad involucrada, son igualmente válidas.

Ahora consideremos los antipatrones que no fueron detectados automáticamente por la herramienta. En todos los casos se debe a que no se había ejecutado la funcionalidad que presenta el problema durante la fase de monitoreo. Consideremos por ejemplo el caso de aprobar solicitudes de pacientes, funcionalidad que no se había ejecutado razón por la cual no se había detectado su problema. Es importante destacar que no necesariamente es un error en el uso de la herramienta. Mientras se monitorea, es fundamental recorrer las partes más usadas de la aplicación, y sólo esas. Si asumimos que la funcionalidad de activar cuentas de nuevos pacientes es usada una vez por día, entonces es correcto ignorar este problema y no perder valioso tiempo refactorizando esta parte de la aplicación tan poco usada. Por el contrario, si asumimos que esta funcionalidad es utilizada frecuentemente, entonces fue un error no haberla ejecutado durante el monitoreo, y evidencia la incapacidad de este sistema para encontrar errores en las porciones de código no ejecutadas.

El mismo análisis se puede hacer respecto al otro antipatrón no encontrado automáticamente (ver Sección 6.2.3). Si en la base de datos nunca existen suficientes artículos para generar órdenes de compra demasiado grandes, o el uso habitual de la aplicación garantiza que las órdenes de compra van a tener en promedio una cantidad de artículos acorde a la capacidad del sistema, entonces no es correcto considerar que este antipatrón es un problema en este contexto.

Analizando estos casos se ve la fuerte dependencia que existe entre el uso que se hace de la aplicación mientras se monitorea y los antipatrones reportados. Para lograr los mejores resultados con esta herramienta se debe monitorear la aplicación con usuarios reales, de manera que se obtengan todos los patrones de uso habituales y se detecten todos los antipatrones que representen un problema real en el uso diario de la aplicación.

Otro resultado importante que hemos obtenido es que el tiempo que consume el monitor es distinto para todas las aplicaciones, debido a la variedad de factores de lo que depende: de la cantidad de EJBs que intervengan en la resolución de un requerimiento, la cantidad de métodos remotos invocados y la cantidad de objetos que retornen. El porcentaje insumido en el monitoreo de las aplicaciones va de 13 % a 49 % en las pruebas realizadas. En general en la aplicación MedRec el monitoreo fue más costoso que en las demás, y esto se debe principalmente a su arquitectura. En Petstore y Libra la mayoría de los componentes son locales, mientras que MedRec está compuesto por 2 aplicaciones independientes que se comunican en forma remota entre sí.

En promedio, el *overhead* impuesto por la herramienta a las aplicaciones analizadas fue del 20 % en las versiones que no tienen problemas de desempeño. Considerando que el *overhead* del monitor es el único costo que impone el uso de esta herramienta, ésta puede ser activada en producción, lo que es ideal porque permite que se registren las interacciones de los usuarios reales con datos reales, facilitando la identificación de los problemas que más afectan al uso habitual de la aplicación. Teniendo en cuenta que el uso del monitor en producción es deseable, y que puede haber circunstancias en que ese 20 % de *overhead* sea inaceptable, la herramienta está preparada para ser activada o desactivada en forma remota, sin afectar a la aplicación monitoreada de ninguna manera.

Respecto a la efectividad de las refactorizaciones aplicadas, en todas las aplicaciones el desempeño mejoró tras corregir los problemas encontrados. El porcentaje en que mejora el desempeño depende fuertemente de las características de la aplicación y del uso que se hagan de las funcionalidades más optimizadas. En este caso, los porcentajes de mejora variaron entre 20 % y 60 %.

**Cuadro 6.9:** Efectividad para encontrar antipatrones.

| Caso de estudio | # Antipatrones | # Detectados | # Refactorizaciones efectivas |
|---|---|---|---|
| MedRec | 7 | 6 | 6 |
| Java Petstore 1.3.1 | 9 | 8 | 8 |
| Libra | 4 | 4 | 4 |
| *Total* | *20* | *18* | *18* |

El Cuadro 6.9 resume la efectividad de la herramienta para encontrar los antipatrones de

*performance* en las aplicaciones evaluadas.

# CAPÍTULO 7

# Conclusiones

En este trabajo se presentó un enfoque para la optimización de aplicaciones Web basadas en la plataforma J2EE.

En primer lugar se presentaron las características de J2EE, las cuales la hacen una de las plataformas para desarrollo de aplicaciones distribuidas más utilizadas en la actualidad. Con respecto a esto, la característica principal de J2EE es la utilización de contenedores para proveen los servicios de bajo nivel, permitiendo que los desarrolladores se concentren en resolver los problemas del negocio y no los tecnológicos. Como consecuencia, la tecnología J2EE simplifica el desarrollo de aplicaciones grandes y distribuidas, reduce costos, y promueve la reutilización de componentes. Vimos que las aplicaciones construidas sobre J2EE siguen un modelo multicapa, lo que significa que la lógica se divide entre componentes de acuerdo a la capa a la que pertenecen y que cada capa se instala en un conjunto distinto de máquinas.

Entre los servicios que proporciona un contenedor J2EE se encuentran: seguridad, control de transacciones flexible, administración de estados, ejecución multihilo y conectividad remota. Aunque estos servicios sean provistos en forma transparente al desarrollador, su uso tiene un costo que de ser ignorado puede ocasionar problemas. En particular vimos que para obtener mayor confiabilidad y escalabilidad se distribuyen los componentes, y esto hace que aumenten los costos de las comunicaciones entre capas debido a la serialización y transporte por la red.

En la literatura actual se ha mostrado que un uso indebido del soporte para invocaciones remotas genera aplicaciones con diseños que tienen un desempeño deficiente, los Antipatrones, y que esas deficiencias frecuentemente no se descubren hasta que el sistema está terminado.

Por otro lado, se describieron y analizaron algunos enfoques existentes para la construcción de aplicaciones con buen desempeño, y se rescataron las ventajas e ideas destacables de cada enfoque. Vimos que algunas metodologías proponen analizar sistemáticamente una

aplicación durante etapas previas a su materialización, con el objetivo de que se garantice que la implementación resultante cumpla con los requerimientos de *performance*. En general estas propuestas presentan las siguientes dificultades:

- es necesaria documentación detallada del proyecto, que no siempre está disponible;
- requieren un gran trabajo previo, especialmente de modelado, por lo que no se pueden utilizar en sistemas ya construidos;
- se debe continuar con el modelado durante todo el ciclo de vida del software, ya que cualquier cambio que se introduzca en el sistema en la etapa de mantenimiento puede degradar su desempeño.

Otros enfoques se basan en identificar problemas en los sistemas ya construidos, tarea que debe ser realizada por especialistas siguiendo una metodología y ayudándose con herramientas de monitoreo, muchas veces analizando grandes volúmenes de datos en forma manual. Vimos que en general todos los trabajos que intentan mejorar el desempeño de estos sistemas requieren la intervención de especialistas en *performance*, lo que representa un incremento directo en los costos de desarrollo.

Considerando los problemas que presentan las propuestas existentes se desarrolló una herramienta para asistir en la corrección de problemas de *performance* de sistemas J2EE. Esta herramienta identifica algunos problemas de diseño en forma totalmente automática, sin requerir la participación de especialistas. Para hacerlo extrae la información necesaria desde la aplicación misma, no requiriendo documentación ni expertos en optimización de aplicaciones. Además, indica las refactorizaciones que se deben aplicar a la aplicación para corregir los problemas de desempeño. Las propuestas de solución se presentan con el detalle suficiente para que cualquier desarrollador J2EE pueda aplicar los cambios en la aplicación y lograr que mejore el desempeño. Las ventajas de este enfoque con respecto a los otros que se han analizado son:

- reduce el tiempo necesario para encontrar los problemas
- permite que se solucionen sin recurrir a expertos en optimización

Para encontrar los problemas de *performance* la herramienta utiliza un monitor que recolecta información sobre el comportamiento de la aplicación en tiempo de ejecución. Luego, busca en esa información la ocurrencia de antipatrones de diseño. Para esto, se mostró cómo representar a los antipatrones en forma de reglas , y luego utilizar un motor de reglas afín. Vimos los beneficios que proporciona la utilización de un motor de reglas: programación declarativa, separación entre datos y lógica, velocidad, escalabilidad y centralización del conocimiento. En particular, permite desacoplar las etapas de monitoreo y de búsqueda de antipatrones, facilitando la extensión de la herramienta. Para incorporar nuevos antipatrones a la lista de problemas que la herramienta puede detectar es suficiente con escribir las reglas que los representen.

Más tarde se mostraron los resultados experimentales obtenidos al utilizar la herramienta para optimizar tres aplicaciones de código abierto escogidas al azar. Se midió la efectividad

de la herramienta para detectar antipatrones, la efectividad de las recomendaciones para mejorar el desempeño de la aplicación, y el *overhead* generado por el monitor.

En las pruebas realizadas la herramienta detectó la ocurrencia de todos los antipatrones conocidos en los fragmentos de aplicación monitoreados. Aquí corresponde hacer una aclaración. Encontrar problemas de *performance* en las aplicaciones, leyendo el código fuente es una tarea que presenta muchos desafíos. En primer lugar, hay que conocer *la forma* del código, esto implica: las convenciones de programación utilizadas para nombrar variables, clases, interfaces, tratamiento de errores, documentación, etc. En segundo lugar, hay que conocer las tecnologías subyacentes porque, generalmente, éstas impactan sobre el diseño de las aplicaciones, como se mostró en el caso del *framework* Struts para la aplicación Libra.

Uno de los resultados evidenciados al realizar estas pruebas fue que existe una fuerte dependencia entre el uso que se hace de la aplicación mientras se monitorea y los antipatrones reportados. Se concluyó que para lograr los mejores resultados se debe monitorear la aplicación con usuarios y datos reales, de manera que se detecten todos los antipatrones que representen un problema real en el uso diario de la aplicación. Dado que la herramienta detecta errores en el código en el momento en que es ejecutado, utilizarlo en producción permite que se detecten nuevos problemas cuando cambia el patrón de uso de una aplicación, o se agregan nuevas funcionalidades al sistema.

De los resultados obtenidos al optimizar las aplicaciones se concluyó que el desempeño mejora luego de aplicar las correcciones sugeridas por la herramienta. El porcentaje exacto de mejora depende de las características de la aplicación y del uso que se hagan de las funcionalidades más optimizadas. En las aplicaciones probadas se obtuvieron mejoras de hasta un 60 %.

Como el uso que se hace de la aplicación afecta al tiempo de respuesta de las operaciones, a los efectos de tomar mediciones para validar que las optimizaciones mejoran el desempeño se puede ejecutar el sistema utilizando una herramienta automática que reproduzca el uso esperado de los usuarios. Para esta tarea se pueden utilizar herramientas como *The Grinder [2]* o *JMeter* [21]. La ventaja de esta opción es que se puede repetir una prueba fácilmente para verificar que las refactorizaciones aplicadas mejoraron la *performance* de la aplicación para el uso analizado. Si se depende del uso real de la aplicación, puede haber cambios en los patrones de uso del sistema que no muestren claramente las mejoras obtenidas como consecuencia de los trabajos de optimización realizados. Otra ventaja de esta posibilidad es que se puede utilizar en etapas de desarrollo en que todavía no existen versiones productivas de la funcionalidad que se desea optimizar.

Con respecto al consumo del monitor, se observó que éste depende de las características de la aplicación que se está monitoreando, y en promedio fue de un 20 % . Entonces, la herramienta puede ser utilizada en sistemas productivos siempre y cuando este *overhead* no sea una pena inaceptable, considerando que es el único costo que impone el uso de esta herramienta.

Por último, al realizar estas pruebas hemos comprobado que la herramienta es portable entre diferentes servidores de aplicaciones. Esto se debe a que sólo utiliza tecnología estándar Java disponible desde la versión de Java 1.3.

## 7.1. Limitaciones

Una de las limitaciones de esta herramienta es que no puede detectar problemas de configuración del ambiente. Al enfocarse en problemas en el diseño de la aplicación, la herramienta asume que los recursos requeridos ya están optimizados, y que la red está bien configurada, al igual que los parámetros de la JVM o del contenedor de aplicaciones. Los problemas de configuración deben ser solucionados con anterioridad de manera manual o utilizando otro enfoque, como los que se presentaron en el capítulo 3.

Tampoco puede detectar errores de diseño que no hayan sido incorporados a su base de conocimiento. Los problemas de diseño deben ser especificados en el lenguaje específico del motor de reglas para que puedan ser detectados.

Otra de las limitaciones de esta herramienta es que su habilidad de detectar errores de diseño depende del uso que se hace de la aplicación mientras se monitorea. Para obtener resultados óptimos se deben monitorear sistemas productivos mientras son utilizados por usuarios reales. Si esto no es posible, porque el sistema está aún en la etapa de desarrollo por ejemplo, se corre el riesgo de que se detecten problemas irrelevantes, y que no se detecten los problemas reales. La participación de un experto en el negocio es recomendada para minimizar este riesgo.

## 7.2. Trabajos futuros

El uso del motor de reglas permite que se pueda extender la herramienta de diversas maneras.

Por un lado se agregarán más reglas de manera que la herramienta sea capaz de detectar más antipatrones.

Además, para ampliar la variedad de antipatrones que puedan ser reconocidos se ampliará la capacidad del monitor para que recolecte más métricas de la aplicación durante su ejecución.

El uso de XML para desacoplar el comportamiento de la aplicación de la detección de antipatrones permite reutilizar las reglas en otros contextos. Otra extensión que se hará a la herramienta es crear monitores para otras tecnologías. Se podrán monitorear servidores atendiendo a clientes ricos y no sólo a clientes Web, y otras tecnologías para invocar remotamente aplicaciones J2EE que no utilicen EJBs. Incluso puede utilizarse el motor de reglas para detectar problemas de *performance* en otras plataformas, como .NET.

# APÉNDICE A

## Implementación de Antipatrones

En este apéndice se presentan las reglas utilizadas para implementar la detección de los antipatrones actualmente soportados por la herramienta.

```
rule "Se debe incorporar un Service Locator con Cache (SLC)"
when
        $lookup: JndiLookup(
                $resourceName: jndiName,
                $stackTrace: shortStackTrace
        )
        exists JndiLookup(
                jndiName == $resourceName,
                this != $lookup
        )
        not Antipatron(
                id == ($resourceName + $stackTrace),
                type == "SLC"
        )
then
        insert(new Antipatron(
                ($resourceName + $stackTrace),
                "SLC",
                "Debe agregar un ServiceLocator con Cache " +
                "para acceder al recurso \"" + $resourceName +
                "\" que es accedido desde " + $stackTrace)
         );
end
```

---

```
rule "Se debe incorporar un BusinessSessionFacade (SF)"
when
        $i1: Invocacion(
```

```
                $requerimiento: requerimiento,
                $clase_1: claseInvocada,
                $esGetter_1: getter,
                $descripcion_1: representacion,
                $stacktrace_1: shortStackTrace
        )
        Invocacion(
                requerimiento == $requerimiento,
                this != $i1,
                claseInvocada != $clase_1 ||
                        getter == false ||
                        getter != $esGetter_1,
                $descripcion_2: representacion,
                $stacktrace_2: shortStackTrace
        )
        not Antipatron(
                id == ($descripcion_1 + $descripcion_2) ||
                        id == ($descripcion_2 + $descripcion_1),
                type=="SF"
        )
then
        insert(new Antipatron(
                $descripcion_1 + $descripcion_2,
                "SF",
                "Debe agregar un SessionFacade para acceder a los metodos " +
                $descripcion_1 + " y " + $descripcion_2 +
                " que son invocados cuando se solicita la URL " +
                $requerimiento.getUri() + " desde " +
                $stacktrace_1 + " y " + $stacktrace_2 + " respectivamente.")
        );
end
```

---

```
rule "Se debe incorporar un Value List Handler (VHL)"
when
    $respuestaExcesiva: CantidadResultados(
        cantidad > 50,
        $tipoResultado: tipo
    )
    Invocacion(
        $requerimiento: requerimiento,
        $descripcion: representacion,
        tiposResultados contains $respuestaExcesiva
    )
    not Antipatron(
        id == ($descripcion + $tipoResultado),
        type == "VHL"
    )
then
    insert(new Antipatron(
        $descripcion + $tipoResultado,
        "VHL",
        "Se debe utilizar un Value List Handler " +
```

```
        "para retornar menor cantidad de " + $tipoResultado +
        " desde " + $descripcion +
        ", que ha retornado " + $respuestaExcesiva)
    );
end
```

---

```
rule "Se debe incorporar un Data Transfer Object (DTO)"
when
        $i1: Invocacion(
                getter == true,
                $requerimiento: requerimiento,
                $clase_1: claseInvocada,
                $descripcion_1: representacion,
                $stacktrace_1: shortStackTrace
        )
        Invocacion(
                this != $i1,
                requerimiento == $requerimiento,
                claseInvocada == $clase_1,
                getter == true,
                $descripcion_2: representacion,
                $stacktrace_2: shortStackTrace
        )
        not Antipatron(
                id == ($descripcion_1 + $descripcion_2) ||
                        id == ($descripcion_2 + $descripcion_1),
                type=="DTO"
        )
then
        insert(new Antipatron(
                $descripcion_1 + $descripcion_2,
                "DTO",
                "Se debe agregar un DTO para recuperar " +
                "los datos de " + $clase_1 +
                " cuando se solicita la URL " + $requerimiento.getUri() +
                ". Actualmente los métodos " +
                $descripcion_1 + " y " + $descripcion_2 +
                " son llamados desde " +
                $stacktrace_1 + " y " + $stacktrace_2 +
                " respectivamente.")
        );
end
```

# Bibliografía


[1] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns*. Pearson Education, 2001.

[2] Philip Aston. http://grinder.sourceforge.ne.

[3] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-based performance prediction in software development: A survey. *IEEE Trans. Softw. Eng.*, 30(5):295–310, 2004.

[4] Stephanie Bodoff, Eric Armstrong, Jennifer Ball, and Debbie Bode Carson. *The J2EE Tutorial*. Prentice Hall PTR, 2004.

[5] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley Press, 1998.

[6] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. *SIGPLAN Not.*, 37(11):246–261, 2002.

[7] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazières, Antonio Dias, Margo Seltzer, and Michael D. Smith. The measured performance of personal computer operating systems. *ACM Trans. Comput. Syst.*, 14(1):3–40, 1996.

[8] Shiping Chen, Ian Gorton, Anna Liu, and Yan Liu. Performance prediction of cots component-based enterprise applications. In Journal of Systems and Software, editors, *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, volume 74, pages 35–43, 2005.

[9] Kingsum Chow, Ricardo Morin, and Kumar Shiv. Enterprise Java performance: Best practices. *Intel Technology Journal*, 7(1):32–46, February 2003.

[10] Mark Fleury. *JBoss 4.0: The Official Guide*. Safari Tech Books, 2005.

[11] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, 2004.

[12] Lucian Gabor and John Murphy. A framework for automated software design optimisation. In *Proceedings of Information Technology and Telecommunications*, pages 3–8, 2002.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, USA, 1995.

[14] Stephen Gilmore, Jane Hillston, Leila Kloul, and Marina Ribaudo. Pepa nets: a structured performance modelling formalism. *Perform. Eval.*, 54(2):79–104, 2003.

[15] Jiang Guo, Yuehong Liao, and B. Parviz. A performance validation tool for j2ee applications. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 10 pp.–, 2006.

[16] D. Heuzeroth, T. Holl, and W. Lowe. Combining static and dynamic analyses to detect interaction patterns, 2002. In IDPT, 2002. (submitted to). Welf Lowe and Markus Noga.

[17] James Holmes. *Struts: The Complete Reference, 2nd Edition*. McGraw-Hill Osborne Media, 2006.

[18] Jason Hunter and William Crawford. *Java Servlet Programming*. O'Reilly, 2001.

[19] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison Wesley, Boston, MA, USA, October 2000.

[20] JetBrains. http://www.jetbrains.com/idea/.

[21] JMeter. http://jakarta.apache.org/jmeter/.

[22] Rod Johnson, Juergen Hoeller, Colin Sampaleanu, and Alef Arendsen. *Professional Java Development with the Spring Framework*. Wrox, 2005.

[23] James Edward Keogh. *J2EE: The Complete Reference*. McGraw-Hill/Osborne, 2002.

[24] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, 2001.

[25] Samuel Kounev and Alejandro Buchmann. Performance modeling and evaluation of large-scale j2ee applications. In *Proc. of the 29th International Conference of the Computer Measurement Group on Resource Management and Performance Evaluation of Enterprise Computing Systems*, December 2003. This contribution received Best-Paper Award.

[26] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 2003.

[27] W. C. Lynch. Operating system performance. *Commun. ACM*, 15(7):579–585, 1972.

[28] Floyd Marinescu. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. Wiley, 2002.

[29] Tom Mens and Tom Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[30] Steven John Metsker and William C. Wake. *Design Patterns In Java*. Addison Wesley, 2006.

[31] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 2001.

[32] A. Mos and J. Murphy. Compas: Adaptive performance monitoring of component-based systems. In *26th International Conference on Software Engineering*, pages 35 – 40, 2004.

[33] Jasmine Noel. The state of j2ee application management: Analysis of 2005 benchmark survey. Technical report, Ptak, Noel & Associates, 2005.

[34] Mike P. Papazoglou and Willem-Jan Heuvel. Service Oriented Architectures: Approaches, Technologies and Research Issues. *The VLDB Journal*, 16(3):389–415, 2007.

[35] Erik Putrycz, Murray Woodside, and Xiuping Wu. Performance techniques for cots systems. *IEEE Software*, 22(4):36–44, 2005.

[36] Zhengping Ren, Xiaoming Liu, and Song Huang. Performance prediction of j2ee applications using pepa nets. In *Communication Technology, 2006. ICCT '06. International Conference on*, pages 1–4, 2006.

[37] Ed Roman, Rima Patel Sriganesh, and Gerald Brose. *Mastering Enterprise JavaBeans, 3rd Edition*. Wiley, 2004.

[38] Jack Shirazi. *Java Performance Tuning*. O'Reilly Media, 2003.

[39] Inderjeet Singh, Beth Stearns, Mark Johnson, and The Enterprise Team. *Designing Enterprise Applications with the J2EE(TM) Platform (2nd Edition)*. Prentice Hall PTR, 2002.

[40] SJAS. https://glassfish.dev.java.net/.

[41] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. In *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, pages 127–136, New York, NY, USA, 2000. ACM Press.

[42] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[43] Lloyd G. Williams and Connie U. Smith. Pasa: a method for the performance assessment of software architectures. In *Proceedings of the 3rd international Workshop on Software and Performance*, pages 179 – 189, 2002.

[44] XStream. http://xstream.codehaus.org/.

[45] Richard Zurawski. *The Industrial Information Technology Handbook*. CRC Press, 2004.