

# CAPITULO VI

## Caso de Estudio

---

### 1. Introducción

El presente capítulo tiene como objetivo mostrar los resultados obtenidos por la herramienta luego de analizar un sistema real, y evaluar la capacidad de la misma para identificar crosscutting concerns en este tipo de sistemas. Se aplica el análisis sobre la versión 1 del sistema Health Watcher [1,2,3].

### 2. Health Watcher

Health Watcher [1,2,3] es una aplicación desarrollada acorde a una arquitectura por capas utilizando tecnología J2EE [654]. El propósito principal de este sistema es permitir a ciudadanos registrar sus quejas referidas a temas de salud. El mismo fue seleccionado por las siguientes razones:

- es un sistema real y suficientemente complejo, con implementaciones tanto en el paradigma de programación orientado a objetos como en el paradigma orientado a aspectos. Ambas versiones fueron diseñadas aplicando principios de modificabilidad [1, 16, 17].
- se han reportado análisis previos del sistema [2], lo cual provee un análisis cualitativo de ambas implementaciones. La disponibilidad de la versión orientada a aspectos permite comprobar la capacidad de la herramienta desarrollada para identificar los crosscutting concerns en la versión orientada a objetos.

- es un sistema real que involucra un gran número de concerns clásicos como por ejemplo concerns de concurrencia, persistencia y distribución. Adicionalmente, la aplicación hace uso de tecnologías comúnmente utilizadas en contextos industriales como RMI (Java Remote Method Invocations) [18], Servlets [22] y JDBC (Java Database Connectivity) [321].

## 2.1. Versión Orientada a Objetos

La versión orientada a objetos del sistema HW fue implementada utilizando el lenguaje de programación Java [11]. A su vez, con el objetivo de lograr modificabilidad y extensibilidad, la arquitectura de este sistema fue organizada en capas. La utilización de este patrón arquitectónico (Layers), junto con la aplicación de patrones de diseño relacionados al mismo, ayudan no sólo a satisfacer dichos atributos de calidad, sino también a disminuir el código entrelazado [1.10, 1.1, 1.17].

Las 4 capas principales del sistema son: interfaz gráfica de usuario (Graphical User Interface GUI), distribución, código de negocio (business) y datos. (Fig. VI – 1)

La capa GUI fue implementada como una interfaz web, utilizando para ello la API de Servlets [22] de Java [11]. Por su parte, la capa de distribución es la responsable de proveer los servicios del negocio en forma distribuida. El acceso a los servicios de HW se realiza mediante la interface *IFacade*, la cual es implementada por *HealthWatcherFacade*. Esta capa es implementada utilizando la tecnología RMI (Remote Method Invocation) [18].

La capa de negocio contiene aquellas clases que representan conceptos del dominio y que contienen las reglas del negocio. Algunas de estas clases, como *ComplaintRecord* y *EmployeeRecord*, permiten el acceso a la capa de datos mediante la interface *IComplaintRep*. A través de esta interface se desacopla la lógica de negocio de la lógica específica de gestión de datos.

Finalmente, la capa de persistencia utiliza la API de JDBC.

Adicionalmente, mediante el uso de patrones de diseño, como Command, Adapter y Decorator [2.1, 2.11, 2.16, 2.17], los requerimientos de reusabilidad y mantenibilidad de la implementación pudieron ser alcanzados.

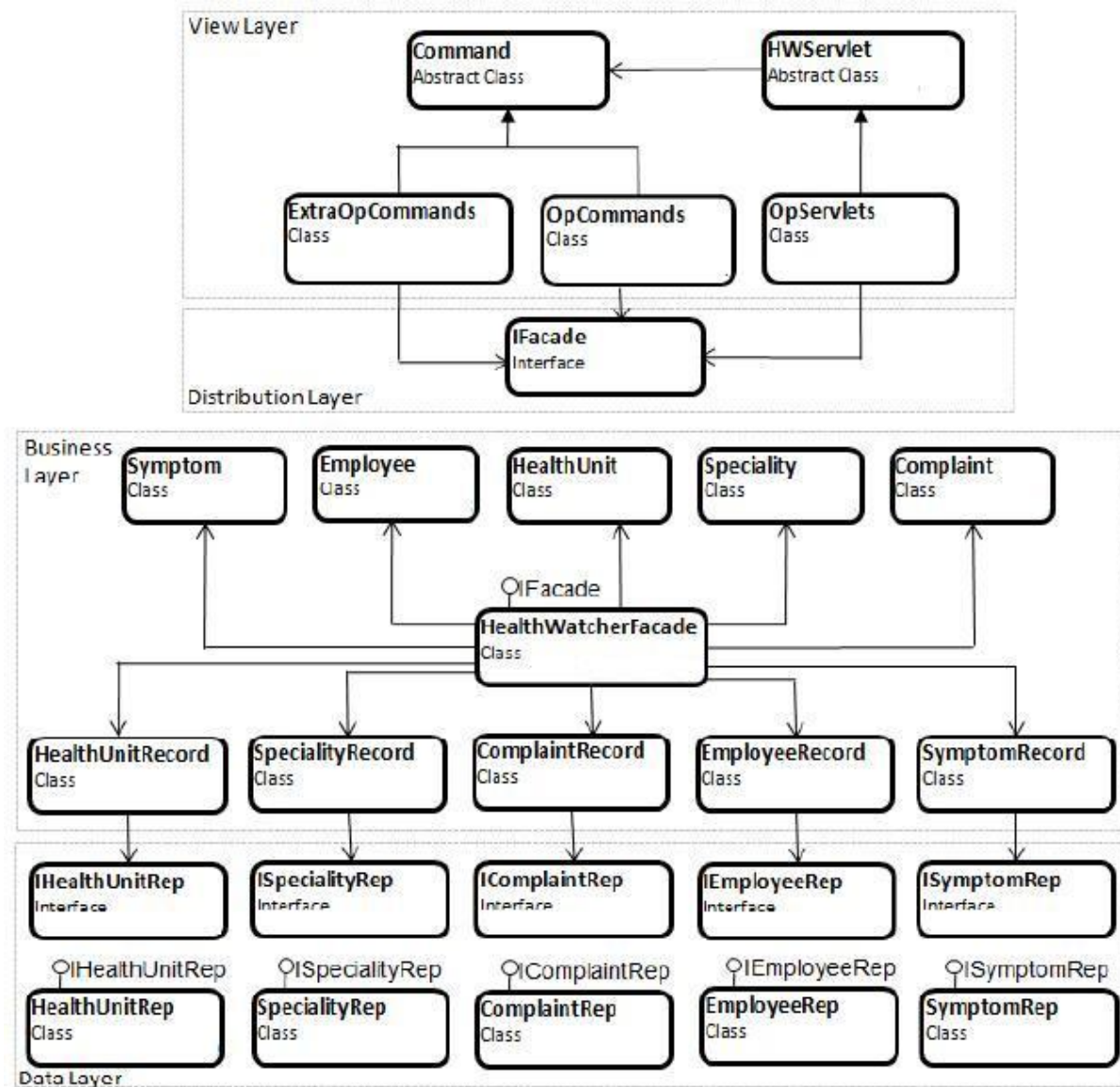


Fig. VI - 1. Arquitectura de HW de la versión OO.

## 2.2. Versión Orientada a Aspectos

Con el objetivo de lograr una mejor separación de concerns, y para evitar código entrelazado y diseminado, este sistema fue reestructurado con el objetivo de introducir aspectos que encapsulen aquellos crosscutting concerns presentes en el mismo.

El diseño de la versión OA se centra en los mismos principios de reusabilidad y mantenibilidad de la versión OO. La única diferencia es que el diseño de esta versión fue llevado a cabo con el fin de separar los crosscutting concerns de persistencia, distribución y concurrencia, del resto de la implementación. Si bien la arquitectura elegida persigue este objetivo, el aislamiento no es absoluto y se pueden encontrar concerns diseminados por las diferentes capas del sistema. La nueva versión implementa de igual manera el patrón arquitectónico por capas, aunque se omite la capa de distribución y se implementa con aspectos (Fig. VI - 2).

A continuación, se describen aquellos crosscutting concerns que fueron encapsulados en aspectos en esta versión del sistema: distribución, persistencia y concurrencia.

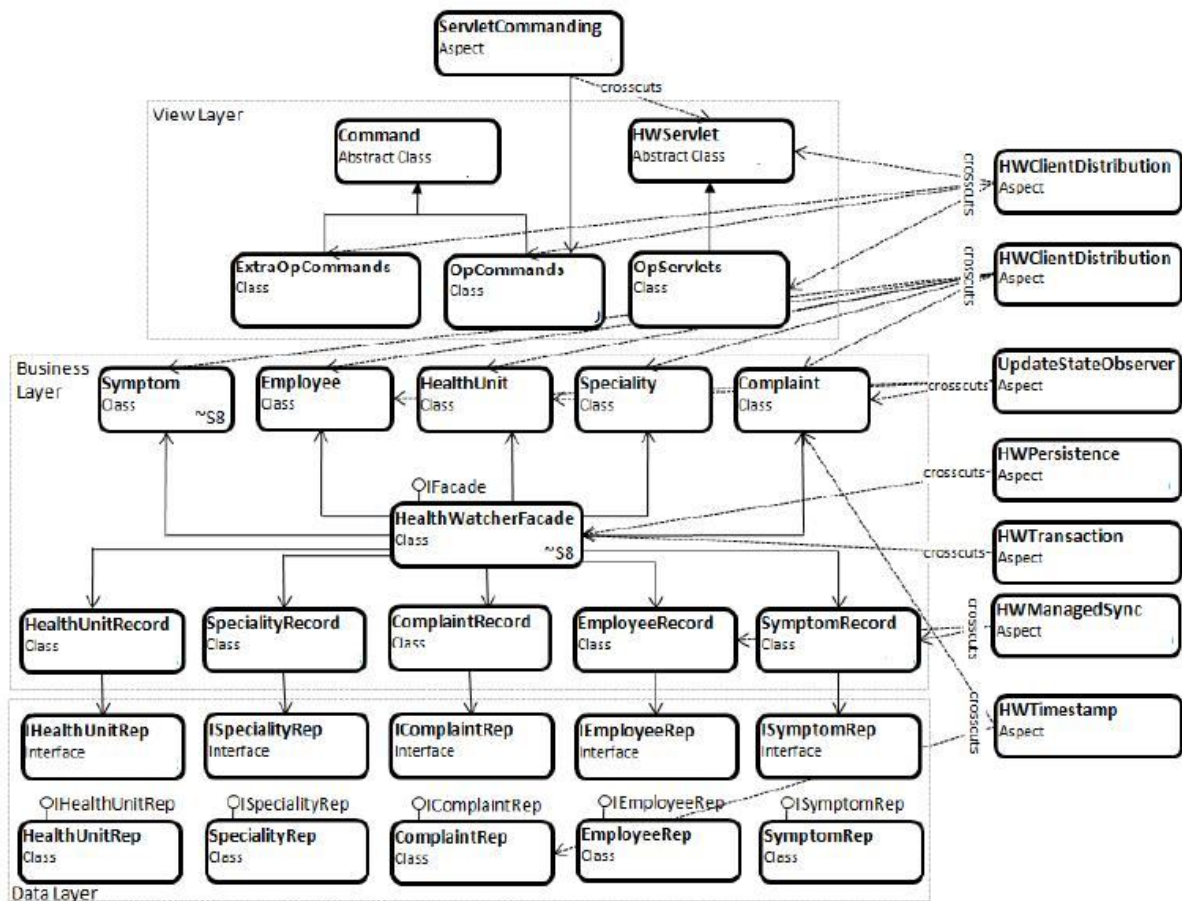


Fig. VI - 2. Arquitectura de HW de la versión OA.

### 2.2.1. Concern de Distribución

El primer paso para encapsular el código correspondiente al concern de distribución en un aspecto consiste en remover el código específico de RMI de la versión escrita puramente en Java. Por lo general, en un sistema con este tipo de arquitectura, el código RMI se encuentra diseminado en la clase *facade* (del lado del servidor) y en las clases de interfaz de usuario (lado del cliente). No obstante, algunas clases de la lógica del negocio también poseen algo de código relacionado a RMI, ya que sus objetos son parámetros y valores de retornos de los métodos del *facade*, los cuales son ejecutados remotamente.

En la versión orientada a aspectos se eliminó el código correspondiente a RMI de todas las clases del sistema, y se implementó la misma funcionalidad de forma separada en un conjunto de aspectos. De esta manera el concern de distribución consiste en un conjunto de aspectos y clases e interfaces auxiliares. Cuando este código contenido en los aspectos es ligado con el código de las clases del sistema, efectivamente aspectualiza el *facade* y las clases correspondientes a la interfaz de usuario; la comunicación entre estas se vuelve remota mediante la distribución de la instancia del *facade* por el cliente y el servidor.

El aspecto *ServerSideHWDistribution* es responsable de la disponibilidad remota de la instancia del *facade* y de asegurar que los métodos de éste último tengan parámetros y valores de retorno serializables, ya que es requerido por RMI. Para que el *facade* esté disponible en forma remota, los aspectos del lado del servidor deben modificar la clase *facade* (*HWFacade*) para implementar la interface remota correspondiente (*IHWFacade*).

### 2.2.2. Concern de Persistencia

Con el fin de evitar que el código correspondiente al concern de persistencia este disperso en las clases de la aplicación, se debe remover dicho código de las clases que componen el sistema. Este concern está concentrado principalmente en las colecciones de datos y en las clases de lógica del negocio.

El código de persistencia incluye, no sólo la interface *IPersistenceMechanism* junto con sus implementaciones, sino también las interfaces de *IBusinessData*. Los aspectos de persistencia afectan el *facade* y las colecciones de datos de la lógica de negocio.

El nuevo código de persistencia incluye aspectos y clases e interfaces auxiliares para satisfacer los siguientes concerns: mecanismo de control de conexión y transacciones, carga parcial y caché de objetos para mejorar la performance y sincronización del estado de los objetos con las entidades de la base de datos para asegurar consistencia.

#### **2.2.2.1. Mecanismo de Control de Persistencia**

Los aspectos que implementan el mecanismo de control de persistencia estan encargados de dar soporte a las operaciones de acceso a los datos. Para ello, los aspectos crean una instancia de la clase que representa al mecanismo de persistencia (una implementación de *IPersistenceMechanism*) y mediante esta clase, manejan la inicialización de la base de datos, la conexión y liberación de los recursos.

El aspecto que controla la persistencia se encarga de inicializar y retornar el mecanismo de persistencia al comenzar la ejecución del sistema, así como también se ocupa de la liberación del recurso.

#### **2.2.2.2. Mecanismo de Control de Transacciones**

Es esencial en un sistema poder garantizar las propiedades ACID [4.5]: atomicidad de las operaciones, consistencia de los datos, aislamiento (isolation) al realizar operaciones y durabilidad de los datos ante fallas. En la versión de HW orientada a objetos, la mayor parte de las operaciones que deben cumplir con las restricciones ACID son invocadas desde la clase *facade*. En consecuencia, esta clase posee código entrelazado relacionado a la transaccionalidad de sus operaciones.

La versión orientada a aspectos intenta modularizar este concern y separar el código entrelazado en un aspecto. Debido a esto, se define un aspecto, *TransactionControlHW*, que

identifica los métodos transaccionales - métodos que deben ser ejecutados como una transacción lógica – y se encarga de comenzar, terminar satisfactoriamente y abortar transacciones. Los métodos transaccionales están incluidos en la interface del facade. De esta manera, el uso de aspectos permitió eliminar del facade todo el código relacionado a este concern.

### **2.2.2.3. Acceso a Datos Bajo Demanda**

Dado que los objetos pueden tener estructuras complejas y estar compuesto de otros objetos, es necesario contar con políticas de acceso a los mismos con el fin de evitar la degradación de la performance del sistema. Por ejemplo, un servicio que permite consultar una lista de quejas posiblemente solo necesite la descripción y el código de cada queja, mientras que un servicio que genere un reporte completo necesitará todos los datos asociados al objeto. El sistema Health Watcher implementa dicha característica agregando un parámetro a dichos métodos de acceso. Este enfoque tiene dos problemas principales, el primero es que el parámetro mencionado no tiene relación con el método en cuestión, y el segundo es que cualquier método que necesite acceder al estado del objeto deberá especificar dicho parámetro.

Con el objetivo de evitar estos problemas, se define un aspecto llamado *ParameterizedLoading* para efectuar el acceso por demanda. Este aspecto agrega a los métodos de acceso el parámetro extra, aunque evita que sea visible para el resto de los servicios del sistema.

### **2.2.3. Concurrencia**

El concern de concurrencia se encuentra relacionado directamente con el concern de persistencia, ya que identifica los puntos de sincronización sobre los datos que se modifican y posteriormente se persisten en la base de datos.

Las capas de negocio y presentación gestionan objetos que deben ser persistidos en la base de datos. Con el fin de garantizar la consistencia de los datos almacenados, el acceso a estos métodos debe estar sincronizado.

Con el fin de separar este concern, las capas mencionadas no deben saber cuando un objeto es persistente o cuando no lo es. Por lo tanto, se deben separar las llamadas a los métodos de sincronización, e implementar la misma funcionalidad en un aspecto que se encargue de controlar la sincronización de los datos.

El aspecto encargado de dicha funcionalidad se denomina *UpdateStateControl*, el cual captura los mensajes a los objetos que se deben persistir y al final del servicio provisto actualiza los datos en el repositorio.

#### **2.2.4. Gestión de Excepciones**

La gestión de excepciones representa un crosscutting concern cuya cobertura es a nivel sistema, es decir, está presente en la mayoría de las clases del mismo. Por lo tanto, el aspecto que implemente dicha funcionalidad tendrá que definir aquellos puntos donde deberán lanzarse las excepciones, y saber cuáles de estas retornar. Por ejemplo, para el caso particular de la gestión de excepciones en los servlets de la aplicación, el aspecto deberá conocer aquellas excepciones específicas a los servlets.

### **3. Identificación de crosscutting concerns en HW**

En esta sección se presentan los resultados obtenidos de aplicar la herramienta desarrollada al sistema Health Watcher (versión orientada a objetos). Para el análisis desarrollado, se optó por aplicar el enfoque Sinergia y la técnica de identificación de métodos redireccionadores, ambos descritos en el capítulo anterior. Se realizaron dos análisis con Sinergia, variando los parámetros de entrada y comparando los resultados entre ambos. Por cada experimento se calcularon tanto la precisión como el recall de la herramienta. La precisión se define como el número de elementos seleccionados (de aquellos encontrados) dividido el número total de elementos encontrados. El recall se



define como el número de elementos seleccionados dividido el número de elementos posibles de encontrar. Para este contexto particular, la precisión se calcula sobre los seeds del sistema, y el recall sobre los concerns.

### 3.1. Sinergia: Análisis I

La Fig. V – 1 muestra los parámetros de entrada seleccionados para la ejecución del enfoque. Se eligió un valor de umbral de 10 e igual valor de confianza (33%) para los tres enfoques. El valor de confianza de Sinergia es del 50%, lo que indica, junto con los valores de confianza seleccionados para los algoritmos individuales, que se reportarán los seeds que hayan sido seleccionados por al menos 2 de los 3 algoritmos.

La Tabla VI -1 exhibe los resultados obtenidos luego de realizado el análisis.

**Fig. VI - 1.** Valores de entrada para Sinergia.

Seed Candidatos	Confianza	Fan-in Seed	Unique Methods Seed	Execution Relations Seed
Método: commitTransaction() Clase: IPersistenceMechanism Paquete: : lib.persistence	99.0%	X	X	X
Método: rollbackTransaction() Clase: IPersistenceMechanism	99.0%	X	X	X

<b>Paquete: lib.persistence</b>				
<b>Método: releaseCommunicationChannel()</b> <b>Clase: IPersistenceMechanism</b> <b>Paquete: : lib.persistence</b>	99.0%	X	X	X
<b>Método: Object getCommunicationChannel()</b> <b>Clase: IPersistenceMechanism</b> <b>Paquete: : lib.persistence</b>	99.0%	X	X	X
<b>Método: beginTransaction()</b> <b>Clase: IPersistenceMechanism</b> <b>Paquete: : lib.persistence</b>	99.0%	X	X	X
<b>Método: getPm()</b> <b>Clase: HealthWatcherFacadeInit</b> <b>Paquete: healthwatcher.business</b>	66.0%	X	-	X
<b>Método: getCodigo()</b> <b>Clase: Complaint</b> <b>Paquete: healthwatcher.model.complaint</b>	66.0%	X		X
<b>Método: errorPage(String):</b> <b>Clase: HTMLCode</b> <b>Paquete: lib.util</b>	66.0%	X	-	X

Tabla VI -1. Resultados Sinergia I.

La Tabla VI – 2 presenta la correspondencia entre los seeds candidatos y los concerns involucrados con cada seed. A continuación, se describe cada seed en particular.

<b>Seed Candidatos</b>	<b>Concern Asociado</b>
<b>Método: commitTransaction()</b> <b>Clase: IPersistenceMechanism</b> <b>Paquete: : lib.persistence</b>	Persistencia (Transacciones)
<b>Método: rollbackTransaction()</b> <b>Clase: IPersistenceMechanism</b> <b>Paquete: lib.persistence</b>	Persistencia (Transacciones)
<b>Método: Object getCommunicationChannel()</b> <b>Clase: IPersistenceMechanism</b> <b>Paquete: lib.persistence</b>	Persistencia
<b>Método: beginTransaction()</b>	Persistencia

<b>Clase: IPersistenceMechanism</b> <b>Paquete: lib.persistence</b>	(Transacciones)
<b>Método: releaseCommunicationChannel()</b> <b>Clase: IPersistenceMechanism</b> <b>Paquete: lib.persistence</b>	Falso Positivo
<b>Método: getPm()</b> <b>Clase: HealthWatcherFacadeInit</b> <b>Paquete: healthwatcher.business</b>	Persistencia (Control de Persistencia)
<b>Método: getCodigo()</b> <b>Clase: Complaint</b> <b>Paquete: healthwatcher.model.complaint</b>	Falso Positivo
<b>Método: errorPage(String):</b> <b>Clase: HTMLCode</b> <b>Paquete: lib.util</b>	Gestión de Excepciones

Tabla VI -2. Seeds candidatos y concerns asociados.

### 3.1.1. Seeds Candidatos: *commitTransaction* y *rollbackTransaction*

Ambos métodos pertenecen a la clase *PersistenceMechanism*, *commitTransaction* indica que se ha finalizado con éxito la ejecución de una operación de tipo transaccional, en cambio el método *rollbackTransaction* permite volver a un estado consistente de la aplicación.

En HW, la mayoría de los métodos transaccionales están definidos en la clase *HealthWatcherFacadeInit*. Dichos métodos no deberían realizar los llamados a *commitTransaction* y *rollbackTransaction*, debido a que no es parte de la funcionalidad básica de la clase. Por lo tanto estamos en presencia de llamados a métodos que implementan el concern de persistencia diseminados en la aplicación. Específicamente corresponden al control de transacciones, los cuales deberían ser encapsulados en un aspecto.

### 3.1.2. Seeds      Candidatos:      *getCommunicationChannel*      y *releaseCommunicationChannel*

Si bien los dos métodos fueron reportados como seeds por las tres técnicas de Sinergia (confianza del 100%), los mismos se corresponden a falsos positivos.

El método *getCommunicationChannel* tiene como objetivo devolver un objeto *Statement* que permite ejecutar consultas SQL a la base de datos, y el método *releaseCommunicationChannel* tiene como objetivo liberar el canal de comunicación. Ambos métodos pertenecen a la clase *PersistenceMechanism*. El contexto en que ambos métodos son utilizados, indica que los mismos implementan funcionalidad relacionada a las clases y métodos que hacen uso de los mismos. Por ejemplo, el método *insert(Address)* de la clase *AddressRepositoryRDB* invoca a *getCommunicationChannel* con el fin obtener el objeto *Statement*, y utilizarlo para ejecutar la inserción.

El hecho de que estos métodos hayan sido reportados como seeds se debe a que los mismos son llamados desde las diferentes implementaciones de los repositorios, haciendo que estos posean un gran Fan-in y estén presentes en muchas relaciones de ejecución.

### 3.1.3. Seed Candidato: *beginTransaction*

El método *beginTransaction()* fue reportado como seed con un 100% de confianza (en realidad la herramienta lo presenta con 99% de confianza ya que se tiene una confianza sobre cada algoritmo del 33%). El mismo pertenece a la interface *IPersistenceMechanism*, y es implementado en forma concreta por *PersistenceMechanism*. La finalidad del método es dar comienzo a una transacción luego de obtener el canal de comunicación.

Los llamados al mismo provienen de la clase *HealthWatcherFacadeInit*, donde los métodos de la misma lo invocan a fin de dar comienzo a una transacción. Dado que la funcionalidad no es propia de los métodos de la clase mencionada se considera a este seed como parte de un crosscutting concern correspondiente a la gestión de las transacciones.

Esta funcionalidad se corresponde con aquella implementada en el aspecto *TransactionControlHW* de la versión OA.

#### **3.1.4. Seed Candidato: *getPm***

El método *getPm()* es reportado como seed solo por dos de los tres enfoques. El enfoque Unique Methods no lo reporta ya que no cumple con la restricción de poseer el tipo de retorno void.

En el contexto de la aplicación, el método *getPm* pertenece a la clase *HealthWatcherFacadeInit*. Este método utiliza el método con la misma signatura de la clase *HealthWatcherFacade* de forma de obtener la instancia del mecanismo de persistencia. Este último es un singleton, el cual retorna la instancia de *IPersistenceMechanism*, y en caso de que la misma no haya sido creada, realiza previamente la inicialización de ésta llamando al método *pmlnit* de *IPersistenceMechanism*.

Este método corresponde al concern de persistencia, específicamente al control de la persistencia (obtener el mecanismo de persistencia e inicializarlo). Claramente, este concern no pertenece a la lógica de inicialización del facade, simplemente se realiza la inicialización en este punto por necesidad, ya que al inicializar los servicios de la aplicación, el mecanismo de persistencia debe ser inicializado. Es por esto, que se decide extraer esta funcionalidad en un aspecto, y refactorizar ambos facades.

#### **3.1.5. Seed Candidato: *getCodigo***

El método *getCodigo* es reportado como seed por dos de los tres enfoques. A pesar de esto, este seed es un falso negativo, ya que es un método getter de la clase *Complaint*.

#### **3.1.6. Seed Candidato: *errorPage***

Este método tiene un valor de confianza del 66%, dado que el análisis de Unique Method no lo reporta como seed debido al tipo de retorno (String) que posee.

El método es utilizado para devolverle al usuario un mensaje de error. Este es invocado desde los distintos servlets al momento en que se presenta una excepción de tipo *RemoteException*, correspondiente a RMI. El método *errorPage* corresponde al concern de manejo de excepciones, por lo que el llamado al mismo es refactorizado en un aspecto. En la nueva versión, este método no será invocado desde los servlets sino será llamado desde dicho aspecto.

La versión orientada a aspectos define un aspecto, *HWDistributionExceptionHandler*, encargado del manejo de excepciones referentes a este tipo de concerns. Tiene como funcionalidad interceptar las excepciones, obtener un objeto sobre el cual presentar los errores, y finalmente presentar el error a los usuarios del servicio.

### **3.1.7. Evaluación del experimento**

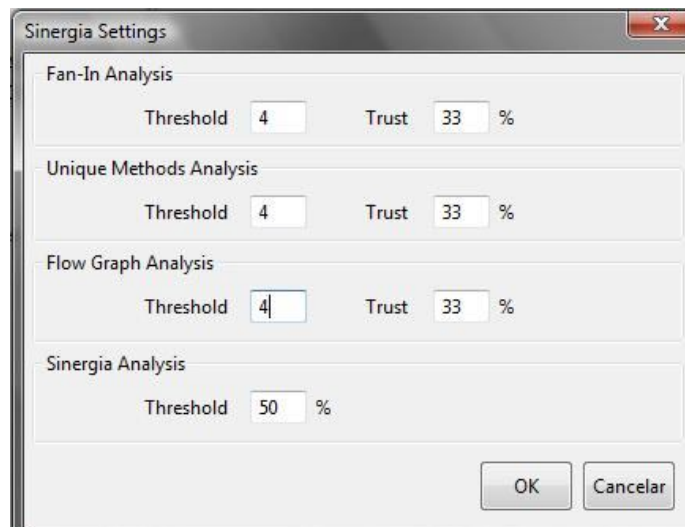
A continuación se presentan valores de interés que resumen el resultado de la ejecución del análisis de Sinergia de la herramienta bajo los parámetros previamente mencionados.

- Número de Seeds Reportados: 8
- Número de Seeds Confirmados: 6
- Número de Falsos Positivos: 2
- Número de Falsos Negativos: 3
- Precisión:  $6 / 8 = 0,75$  (Calculado como el Número de Seeds Confirmados sobre el Número de Seeds Reportados)
- Recall:  $3 / 6 = 0,50$  (Calculado como el Número de Crosscutting Concerns Detectados sobre el Número de Crosscutting Concerns Posibles de Detectar)

### 3.2. Sinergia: Análisis II

En esta sección se presentan los resultados de ejecutar nuevamente el algoritmo de Sinergia seleccionando valores de umbrales inferiores a los establecidos en el primer análisis. Como resultado se obtuvo un conjunto de resultados diferente y lógicamente más amplio al conseguido anteriormente. Esto se realiza con el objetivo de comparar ambos resultados, analizando cuánto afecta el establecimiento de umbrales adecuados en la identificación de crosscutting concerns.

La Fig. VI – 2 muestra los parámetros de entrada elegidos para la ejecución del enfoque. El valor de confianza de Sinergia es del 50%, lo que indica, que se reportarán los seeds que hayan sido seleccionados al menos por 2 de los 3 algoritmos. Por otra parte, los umbrales mínimos para cada tecnica fueron drásticamente disminuidos pasando de un valor de 10 a un valor de 4. La idea de este segundo experimento es observar cómo varían la precisión y el recall respecto del primer experimento. Al disminuir los valores de los umbrales se espera un aumento en el recall (cantidad de crosscutting concerns identificados) y una disminución de la precisión (porcentaje de seeds confirmados sobre seeds reportados).



Analysis Type	Threshold	Trust
Fan-In Analysis	4	33 %
Unique Methods Analysis	4	33 %
Flow Graph Analysis	4	33 %
Sinergia Analysis	50 %	

**Fig. VI – 2.** Valores de entrada para Sinergia II.

La Tabla VI -3 exhibe los resultados obtenidos luego de realizado el análisis.

Seed Candidatos	Confianza	Fan-in Seed	Unique Methods Seed	Execution Relations Seed
Método: commitTransaction() Clase: IPersistenceMechanism Paquete: lib.persistence	99.0%	X	X	X
Método: rollbackTransaction() Clase: IPersistenceMechanism Paquete: lib.persistence	99.0%	X	X	X
Método: releaseCommunicationChannel() Clase: IPersistenceMechanism Paquete: lib.persistence	99.0%	X	X	X
Método: Object getCommunicationChannel() Clase: IPersistenceMechanism Paquete: lib.persistence	99.0%	X	X	X
Método: beginTransaction() Clase: IPersistenceMechanism Paquete: : lib.persistence	99.0%	X	X	X
Método: close () Clase: IteratorDsk Paquete: lib.util	99.0%	X	X	X
Método: validaData(int,int,int) Clase: Date Paquete: lib.util	99.0%	X	X	X
Método: getPm() Clase: HealthWatcherFacadeInit Paquete: healthwatcher.business	66.0%	X	-	X
Método: getCodigo() Clase: Complaint Paquete: healthwatcher.model.complaint	66.0%	X		X
Método: errorPage(String): Clase: HTMLCode Paquete: lib.util	66.0%	X	-	X
Método: errorPageAdministrator(String) Clase: HTMLCode Paquete: lib.util	66.0%	X	-	X
Método: errorPageQueries(String) Clase: HTMLCode	66.0%	X	-	X



<b>Paquete: lib.util</b>				
<b>Método: htmlPage(String,String) Clase: HTMLCode Paquete: lib.util</b>	66.0%	X	-	X
<b>Método: htmlPageAdministrator(String,String) Clase: HTMLCode Paquete: lib.util</b>	66.0%	X	-	X
<b>Método: htmlPage(String,String,int) Clase: HTMLCode Paquete: lib.util</b>	66.0%	X	-	X
<b>Método: hasNext() Clase: IteratorDsk Paquete: lib.util</b>	66.0%	X	-	X
<b>Método: next() Clase: IteratorDsk Paquete: lib.util</b>	66.0%	X	-	X
<b>Método: insert(Address) Clase: AddressRepositoryRDB Paquete: healthwatcher.data.rdb</b>	66.0%	X	-	X
<b>Método: search(int) Clase: AddressRepositoryRDB Paquete: healthwatcher.data.rdb</b>	66.0%	X	-	X
<b>Método: search(int) Clase: HealthUnitRepositoryRDB Paquete: healthwatcher.data.rdb</b>	66.0%	X	-	X
<b>Método: getPm() Clase: HealthWatcherFacade Paquete: healthwatcher.business</b>	66.0%	X	-	X
<b>Método: getCommunicationChannel(boolean) Clase: PersistenceMechanism Paquete: lib.persistence</b>	66.0%	X	-	X
<b>Método: open(String) Clase: HTMLCode Paquete: lib.util</b>	66.0%	X	-	X
<b>Método: getLogin() Clase: HTMLCode</b>	66.0%	X	-	X

<b>Paquete: healthwatcher.model.employee</b>				
<b>Método: closeAdministrator ()</b> <b>Clase: HTMLCode</b> <b>Paquete: lib.util</b>	66.0%	X	-	X
<b>Método: getCode()</b> <b>Clase: Address</b> <b>Paquete: healthwatcher.model.address</b>	66.0%	X	-	X
<b>Método: getDescricao()</b> <b>Clase: Complaint</b> <b>Paquete: healthwatcher.model.complaint</b>	66.0%	X	-	X
<b>Método: getSituacao()</b> <b>Clase: Complaint</b> <b>Paquete: healthwatcher.model.complaint</b>	66.0%	X	-	X
<b>Método: getOccurenceLocalAddress()</b> <b>Clase: AnimalComplaint</b> <b>Paquete: healthwatcher.model.complaint</b>	66.0%	X	-	X
<b>Método: getCode()</b> <b>Clase: HealthUnit</b> <b>Paquete: healthwatcher.model.healthguide</b>	66.0%	X	-	X
<b>Método: getDescription()</b> <b>Clase: HealthUnit</b> <b>Paquete: healthwatcher.model.healthguide</b>	66.0%	X	-	X
<b>Método: getCodigo()</b> <b>Clase: MedicalSpeciality</b> <b>Paquete: healthwatcher.model.healthguide</b>	66.0%	X	-	X
<b>Método: getDescricao()</b> <b>Clase: MedicalSpeciality</b> <b>Paquete: healthwatcher.model.healthguide</b>	66.0%	X	-	X

Tabla VI -3. Resultados Sinergia II.

La Tabla VI – 4 presenta la correspondencia entre los seeds candidatos y los concerns involucrados con cada seed. A continuación se hará una explicación para cada uno de ellos, omitiendo los descriptos en Sinergia I. Los resultados de Sinergia II amplían los seeds candidatos obtenidos en Sinergia I, ya que disminuir los umbrales de cada algoritmo tiene como consecuencia ampliar el espacio de solución de cada uno de ellos en particular.

Seed Candidatos	Concern Asociado
Método: close () Clase: IteratorDsk Paquete: lib.util	Falso Positivo
Método: validaData(int,int,int) Clase: Date Paquete: lib.util	Falso Positivo
Método: errorPageAdministrator(String) Clase: HTMLCode Paquete: lib.util	Falso Positivo
Método: errorPageQueries(String) Clase: HTMLCode Paquete: lib.util	Gestión de Excepciones
Método: htmlPage(String,String) Clase: HTMLCode Paquete: lib.util	Falso Positivo
Método: htmlPageAdministrator(String,String) Clase: HTMLCode Paquete: lib.util	Falso Positivo
Método: htmlPage(String,String,int) Clase: HTMLCode Paquete: lib.util	Falso Positivo
Método: hasNext() Clase: IteratorDsk Paquete: lib.util	Falso Positivo
Método: next() Clase: IteratorDsk Paquete: lib.util	Falso Positivo
Método: insert(Address) Clase: AddressRepositoryRDB Paquete: healthwatcher.data.rdb	Persistencia
Método: search(int) Clase: AddressRepositoryRDB Paquete: healthwatcher.data.rdb	Persistencia
Método: search(int) Clase: HealthUnitRepositoryRDB Paquete: healthwatcher.data.rdb	Persistencia

<b>Método: getPm()</b> <b>Clase: HealthWatcherFacade</b> <b>Paquete: healthwatcher.business</b>	Persistencia
<b>Método: getCommunicationChannel(boolean)</b> <b>Clase: PersistenceMechanism</b> <b>Paquete: lib.persistence</b>	Falso Positivo
<b>Método: getLogin()</b> <b>Clase: Employee</b> <b>Paquete: healthwatcher.model.employee</b>	Falso Positivo
<b>Método: open(String)</b> <b>Clase: HTMLCode</b> <b>Paquete: lib.util</b>	Falso Positivo
<b>Método: closeAdministrator ()</b> <b>Clase: HTMLCode</b> <b>Paquete: lib.util</b>	Falso Positivo
<b>Método: getCode()</b> <b>Clase: Address</b> <b>Paquete: healthwatcher.model.address</b>	Falso Positivo
<b>Método: getDescricao()</b> <b>Clase: Complaint</b> <b>Paquete: healthwatcher.model.complaint</b>	Falso Positivo
<b>Método: getSituacao()</b> <b>Clase: Complaint</b> <b>Paquete: healthwatcher.model.complaint</b>	Falso Positivo
<b>Método: getOccurrenceLocalAddress()</b> <b>Clase: AnimalComplaint</b> <b>Paquete: healthwatcher.model.complaint</b>	Falso Positivo
<b>Método: getCode()</b> <b>Clase: HealthUnit</b> <b>Paquete: healthwatcher.model.healthguide</b>	Falso Positivo
<b>Método: getDescription()</b> <b>Clase: HealthUnit</b> <b>Paquete: healthwatcher.model.healthguide</b>	Falso Positivo
<b>Método: getCodigo()</b> <b>Clase: MedicalSpeciality</b> <b>Paquete: healthwatcher.model.healthguide</b>	Falso Positivo
<b>Método: getDescricao()</b>	Falso Positivo

<b>Clase: MedicalSpeciality</b> <b>Paquete: healthwatcher.model.healthguide</b>	
--	--

Tabla VI -4. . Seeds candidatos y concerns asociados.

### 3.2.1. Seeds Candidatos: Métodos Utilitarios

Bajo este título se agrupan los métodos utilitarios que la herramienta reporta como seed, aunque corresponden a falsos negativos (Tabla VI - 5). Estos sedes son en realidad métodos que presentan funcionalidades utilitarias de la aplicación, como por ejemplo iterar sobre una colección de objetos o funciones referidas a la presentación de los datos.

Método	Clase	Paquete
close ()	IteratorDsk	lib.util
hasNext()	IteratorDsk	lib.util
next()	IteratorDsk	lib.util
validaData(int,int,int)	Date	lib.util
errorPageAdministrator(String)	HTMLCode	lib.util
htmlPage(String,String)	HTMLCode	lib.util
htmlPageAdministrator(String,String)	HTMLCode	lib.util
htmlPage(String,String,int)	HTMLCode	lib.util
open(String)	HTMLCode	lib.util
closeAdministrator ()	HTMLCode	lib.util

Tabla VI -5. Métodos utilitarios.

### 3.2.2. Seed Candidato: *errorPageQueries(String)*

Este método es ejecutado desde los Servlets al reportarse la excepción *ObjectNotFoundException*. En consecuencia corresponde a la lógica del manejo de excepciones y debe ser refactorizado en un aspecto.

### **3.2.3. Seeds Candidatos: *search(int)*, *AddressRepositoryRDB* y *search(int)*, *HealthUnitRepositoryRDB***

Los métodos *search(int)* de la clase *AddressRepositoryRDB*, y *search(int)* de la clase *HealthUnitRepositoryRDB* tienen un valor de confianza del 66%,. El enfoque de Unique Methods no los reporta ya que no cumplen con la restricción de poseer un tipo de retorno void.

El objetivo de estos métodos es obtener información según el valor entero pasado como parámetro. Los llamados a los mismos provienen de los servlets *ServletUpdateHealthUnitSearch* y *ServletSearchComplaintData*, pasando por clases intermedias, incluidos los facades. Este comportamiento se repite en los distintos tipos de repositorios, y servlets que listan información.

Ambos métodos corresponden al concern de persistencia, específicamente a la funcionalidad de acceso bajo demanda explicado previamente. El parámetro utilizado en la búsqueda no tiene relación con la lógica del método *search*, sino que se adiciona para implementar este tipo de acceso. Por esta razón, se separa el concern en el aspecto *ParameterizedLoading*, el cual intercepta los llamados a este tipo de métodos.

### **3.2.4. Seed Candidato: *getCommunicationChannel(boolean)***

El método tiene como objetivo obtener un canal de comunicación con la base de datos. El seed reportado es un falso positivo debido a que este método es utilizado por otros métodos con funcionalidad relacionada.

### **3.2.5. Seed Candidato: *getPm()***

El método *getPm()* de *HealthWatcherFacade* lleva a cabo la creación e inicialización de la instancia del mecanismo de persistencia. El mismo se corresponde con el concern de persistencia. Esto ha sido expuesto en Sinergia I, para el método de igual nombre perteneciente a la clase *HealthWatcherFacadeInit*.

### 3.2.6. Seed Candidato: *insert(Address)*

El método *insert(Address)* pertenece a la clase *AddressRepositoryRDB* y tiene como objetivo realizar la inserción de una dirección (*Address*) en el repositorio. Esta funcionalidad es propia de la clase a la que pertenece. En consecuencia, no corresponde a la implementación de un crosscutting concern. El seed reportado es un falso positivo.

### 3.2.7. Seeds Candidatos: métodos getters

La Tabla VI – 6 resume los métodos getters reportados por Sinergia II como seeds candidatos. Todos ellos son falsos positivos debido a que son métodos propios de cada clase utilizados para consultar su estado.

Método	Clase	Paquete
<code>getCode()</code>	<b>Address</b>	<b>healthwatcher.model.address</b>
<code>getDescricao()</code>	<b>Complaint</b>	<b>healthwatcher.model.complaint</b>
<code>getSituacao()</code>	<b>Complaint</b>	<b>healthwatcher.model.complaint</b>
<code>getOccurrenceLocalAddress()</code>	<b>AnimalComplaint</b>	<b>healthwatcher.model.complaint</b>
<code>getCode()</code>	<b>HealthUnit</b>	<b>healthwatcher.model.healthguide</b>
<code>getDescription()</code>	<b>HealthUnit</b>	<b>healthwatcher.model.healthguide</b>
<code>getCodigo()</code>	<b>MedicalSpeciality</b>	<b>healthwatcher.model.healthguide</b>
<code>getDescricao()</code>	<b>MedicalSpeciality</b>	<b>healthwatcher.model.healthguide</b>
<code>getLogin()</code>	<b>Employee</b>	<b>healthwatcher.model.employee</b>

**Tabla VI -6.** Métodos getters reportados por Sinergia II.

### 3.2.8. Evaluación del experimento

A continuación se presentan valores de interés que resumen el resultado de la segunda ejecución del análisis de Sinergia de la herramienta bajo los parámetros previamente mencionados.

- Número de Seeds Reportados: 33

- Número de Seeds Confirmados: 11
- Número de Falsos Positivos: 22
- Número de Falsos Negativos: 2
- Precisión:  $11 / 33 = 0.3333$
- Recall:  $4 / 6 = 0,6666$

### 3.3. Análisis de Métodos Redireccionadores

La Tabla VI – 7 presenta los resultados provenientes de la ejecución del análisis Métodos Redireccionadores. Se seleccionan las clases candidatas que devuelven un porcentaje de métodos redireccionadores igual o mayor a 50%. A continuación se realiza un análisis de los candidatos encontrados, se muestran en la tabla sólo aquellos que al menos tienen 3 métodos redireccionadores. Este filtro se decidió aplicar dado que sin él se obtiene un gran número de seeds candidatos con tan sólo uno o dos métodos redireccionadores, que no corresponden en general a estructuras de tipo adapter o decorador, resultando en falsos positivos.

Seed Candidatos	Confianza	Cant. Métodos	%
<b>Clase:</b> HealthWatcherFacade <b>Paquete:</b> healthwatcher.business	<b>Clase:</b> HealthWatcherFacadeInit <b>Paquete:</b> healthwatcher.business	16	69,56%
<b>Clase:</b> ComplaintRecord <b>Paquete:</b> healthwatcher.business.complaint	<b>Clase:</b> IComplaintRepository <b>Paquete:</b> healthwatcher.data	3	50%
<b>Clase:</b> HealthUnitRecord <b>Paquete:</b> healthwatcher.business.healthguide	<b>Clase:</b> IHealthUnitRepository <b>Paquete:</b> healthwatcher.data	4	57,1%

Tabla VI -7. Seeds reportados por el análisis Métodos Redireccionadores.

La Tabla VI – 8 muestra un resumen de por que las clases expuestas como seeds candidatos redireccionan a otras clases. Luego se presenta un análisis para cada uno de ellos.



Clase Base	Clase Redireccionada	Propósito
<b>Clase:</b> HealthWatcherFacade <b>Paquete:</b> healthwatcher.business	<b>Clase:</b> HealthWatcherFacadelnit <b>Paquete:</b> healthwatcher.business	Implementación del patron decorator
<b>Clase:</b> ComplaintRecord <b>Paquete:</b> healthwatcher.business.complaint	<b>Clase:</b> IComplaintRepository <b>Paquete:</b> healthwatcher.data	Implementación del patrón wrapper
<b>Clase:</b> HealthUnitRecord <b>Paquete:</b> healthwatcher.business.healthguide	<b>Clase:</b> IHealthUnitRepository <b>Paquete:</b> healthwatcher.data	Implementación del patrón wrapper

Tabla VI -8. Propósito de redirecciones de las clases reportadas como Seeds.

### 3.3.1. Seeds Candidatos: clases Servlets

La herramienta reporta a todos los servlets de la aplicación como clases redireccionadoras con un 100% de confianza, teniendo solo un método cada uno. La razón de este resultado es que los servlets representan la capa de presentación de la aplicación, y todos ellos redireccionan sus métodos a la capa de distribución o negocio. Esto se debe a la arquitectura planteada en la aplicación, ya que las clases servlets presentan al usuario los servicios del sistema, y deben redireccionar estos llamados a las capas inferiores para llevar a cabo lo solicitado. En su mayoría, redireccionan a la clase facade, debido a que en esta convergen los servicios, actuando como punto de entrada y manejando los mismos.

Si bien estas clases son redireccionadoras, provienen de la arquitectura planteada por del sistema, en consecuencia no son computadas como seeds.

### 3.3.2. Seed Candidato: clase HealthWatcherFacade

La clase *HealthWatcherFacade* está destinada a ser el punto de entrada a los servicios de la aplicación. Sin embargo, al realizar un análisis en profundidad, se puede ver que implementa el patrón Decorator. Cuenta con una instancia de la clase *HealthWatcherFacadelnit* a la cual redirecciona un 69,56% de sus métodos (16 del total). La clase *HealthWatcherFacade* tiene como objetivo implementar el concern de distribución, convirtiendo los servicios provistos por *HealthWatcherFacadelnit* en servicios remotos.

La versión orientada a aspectos refactoriza este concern utilizando el aspecto *ServerSideHWDistribution*, el cual permite la conexión remota de los métodos, y elimina la clase *HealthWatcherFacadeInit*, dejando solo la clase *HealthWatcherFacade*.

### 3.3.3. Seeds Candidatos: clases Records

Las clases de la Tabla VI – 9 implementan el patrón adapter para los repositorios de datos. Las clases adaptadoras forman parte de la capa de negocio y brindan una interfaz de acceso que encapsula los repositorios y presenta los servicios de acuerdo a una interfaz específica.

Clase	Paquete
DiseaseRecord	healthwatcher.business.complaint
MedicalSpecialityRecord	healthwatcher.business.healthguide
EmployeeRecord	healthwatcher.business.employee
ComplaintRecord	healthwatcher.business.complaint
HealthUnitRecord	healthwatcher.business.healthguide

Tabla VI -9. Clases adpaters.

Se puede ver, que el patrón adapter se encuentran diseminado en dos clases, y que si se necesita realizar un cambio se deberían modificar ambas clases. Si se utiliza una implementación orientada aspectos para encapsular este comportamiento, la implementación aumentaría la reusabilidad y centralizaría el código en un aspecto.

### 3.3.4. Evaluación del experimento

A continuación se presentan valores de interés que resumen el resultado de la ejecución del análisis de Redirector Methods de la herramienta bajo los parámetros previamente mencionados.

- Número de Seeds Reportados: 3

- Número de Seeds Confirmados: 3
- Número de Falsos Positivos: 0
- Número de Falsos Negativos: 5
- Precisión:  $3 / 3 = 1$
- Recall:  $1 / 6 = 0,1666$

## 4. Análisis de los Resultados

A continuación se presentará una comparación entre los resultados obtenidos de la ejecución de las técnicas ejecutadas. Luego se muestra una comparativa entre los concerns existentes en el sistema y los encontrados por cada una.

### 4.1. Comparación entre Técnicas

Los análisis de Sinergia difieren en el valor de umbral seleccionado para cada algoritmo. El establecimiento de los valores de umbral impacta directamente sobre la cantidad de crosscutting concerns identificados. Si se disminuyen los valores de umbral, trae como consecuencia que la magnitud de la solución aumente, dando lugar a un número mayor de falsos positivos, lo que reduce la precisión del algoritmo. A su vez, el número de crosscutting concerns identificados aumenta, elevando el valor de recall de la herramienta.

El primero de los análisis exhibe mayor precisión ya que se definen valores de umbrales relativamente altos para el sistema, un valor de 10 para cada uno de los algoritmos. Por otra parte, Sinergia II reduce los valores de umbral a 4 para cada algoritmo. En consecuencia, se obtiene mayor cantidad de aspectos candidatos, a costa de encontrar más falsos positivos. En el Gráfico VI – 1 y la Tabla VI – 10 se presentan los valores de seeds confirmadas y falsos positivos de ambos enfoques. Se puede notar que el valor de falsos positivos aumenta considerablemente para el segundo análisis, y no así el valor de seeds

confirmadas. El costo/beneficio que acarrea disminuir el umbral y analizar manualmente los seeds no es favorable.

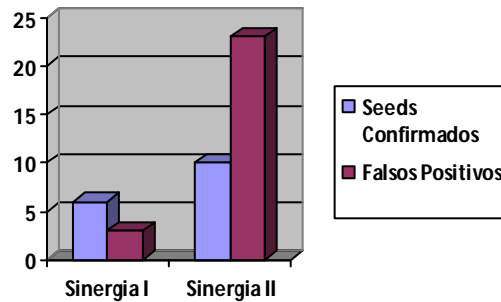


Gráfico VI -1. Resultados Sinergia I y II.

	Sinergia I	Sinergia II
Seeds Confirmadas	6	10
Falsos Positivos	3	23

Tabla VI -10. Resultados Sinergia I y II.

El Gráfico VI – 2 y la tabla VI – 11 reflejan los resultados obtenidos luego de realizar el análisis de métodos redireccionadores sobre el sistema Health Watcher. Estos resultados no pueden ser comparados directamente con los obtenidos en los enfoques anteriores debido a que esta técnica devuelve concerns a nivel de clases, y no a nivel de métodos.

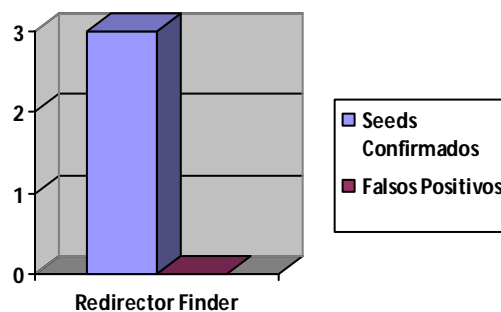


Gráfico VI -2. Resultados Redirector

	Redirector Finder
Seeds Confirmadas	3
Falsos Positivos	0

Tabla VI -11. Resultados Redirector

Los resultados anteriores reflejan la precisión de cada algoritmo (Tabla VI – 12). La precisión de los mismos se calcula como  $\text{seedsConfirmados} / \text{seedsReportados}$ .

	Sinergia I	Sinergia II	Redirector Finder
Seeds Confirmadas	6	10	3
Seeds	8	33	3

Reportados			
Precisión	0,75	0,3030	1

Tabla VI -12. Precisión por algoritmo.

## 4.2. Concerns Existentes y Concerns Reportados por las Técnicas

El sistema Health Watcher presenta 4 tipos de concerns implementados: distribución, persistencia, concurrencia y gestión de excepciones. El concern de persistencia se puede subdividir en 3 concerns más específicos: mecanismo de control de persistencia, mecanismo de control de transacciones y acceso a datos bajo demanda [1, 2]. A continuación, se presenta un análisis de los concern identificados por cada uno de los enfoques. En la Tabla VI – 13 se reflejan los seeds confirmados de cada algoritmo, los falsos negativos y el valor de recall. Este último se calcula como  $\text{concernEncontrados}/\text{concernsExistentes}$ .

- **Sinergia I:** detecta los concerns de control de persistencia, control de transacciones y gestión de excepciones. El resto de los concerns caen en la definición de falsos positivos. Esto se debe a dos razones. En primer lugar, el concern de acceso a datos no está contemplado debido a los valores de umbral seleccionados. El resto no son contemplados debido a la naturaleza de seeds que reporta este enfoque, los cuales son aquellos métodos usados desde otros métodos. Los concerns de distribución y concurrencia no están implementados de tal forma. Por ejemplo, para el último de estos, el control en la concurrencia se logra haciendo que los métodos involucrados estén sincronizados, agregando en su declaración el modificador `synchronized`. El Gráfico VI – 3 y la Tabla VI – 13 reflejan estos valores.
- **Sinergia II:** detecta los concerns de control de persistencia, control de transacciones, acceso a datos y gestión de excepciones. El análisis es similar al realizado en el punto anterior por Sinergia I, solo difiere en el valor de

umbral seleccionado. A razón de esto se detecta el concern de acceso de datos. El Gráfico VI – 4 y la Tabla VI – 13 reflejan estos valores.

- **Redirector Finder:** detecta el concern de distribución, y reconoce patrones adapters en los cuales su refactorización a aspectos es óptima. Estos últimos no fueron encapsulados en aspectos en análisis previos del sistema [1, 2, 3]. Este análisis está destinado a encontrar patrones como adapters y decorators, es por eso que el resto de concerns no son detectados por la técnica y se clasifican bajo el nombre de falsos positivos. El Gráfico VI – 5 y la Tabla VI – 13 reflejan los valores.

	Sinergia I	Sinergia II	Redirector Finder
Seeds Confirmados	6	10	3
Falsos Negativos	2	23	0
Recall	0,5	0,66	0,1666

Tabla VI -13. Precisión por algoritmo.

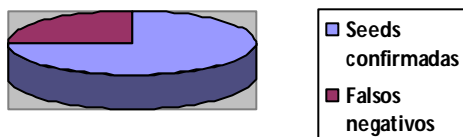


Gráfico VI -3. Concerns Sinergia I.

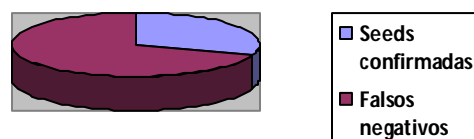


Gráfico VI -4. Concerns Sinergia II.

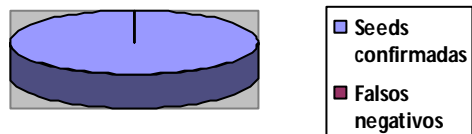


Gráfico VI -5. Concerns Redirector Finder.

# Referencias

---

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. Communications of the ACM, 44(10):59{65, October 2001.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification. Addison-Wesley, second edition, 2000.

[1] Soares, S.; Borba, P.; Laureano, E.: Distribution and Persistence as Aspects. In: Software Practice and Experience, Wiley, vol. 36 (7), (2006) 711-759.

[2] Greenwood, P.; et al.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: ECOOP'07. LNCS, vol. 4609, Springer (2007) 176-200.

[3] **Assessing the Impact of Aspects on Exception Flows: An Exploratory Study**

[3] **Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study**

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification. Addison-Wesley, second edition, 2000.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. Communications of the ACM, 44(10):59{65, October 2001.

[16] S. Soares, et al. "Implementing Distribution and Persistence Aspects with AspectJ". Proc. OOPSLA'02.

[17] S. Soares. An Aspect-Oriented Implementation Method. Doctoral Thesis, Federal Univ. of Pernambuco, 2004.

[22] Hunter, J., Crawford, W.: Java Servlet Programming. O'Reilly and Associates Inc. 1998

[1.10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[1.17] T. Massoni, V. Alves, S. Soares, and P. Borba. PDC: Persistent Data Collections pattern. In First Latin American Conference on Pattern Languages of Programming | SugarLoafPloP, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

[1.1] V. Alves and P. Borba. Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications. In First Latin American Conference on Pattern Languages of Programming | SugarLoafPloP, Rio de Janeiro, Brazil, October 2001. UERJ Magazine: Special Issue on Software Patterns.

[18] S. Microsystems. Java Remote Method Invocation (RMI). At <http://java.sun.com/products/jdk/1.2/docs/guide/rmi>, 2001.

[2.1] V. Alves, P. Borba. Distributed Adapters Pattern: A Design Pattern for Object-Oriented Distributed Applications. Proc. SugarLoafPloP.01, Rio de Janeiro, October 2001.

[2.16] S. Soares, et al. "Implementing Distribution and Persistence Aspects with AspectJ". Proc. OOPSLA'02.

[2.17] S. Soares. An Aspect-Oriented Implementation Method. Doctoral Thesis, Federal Univ. of Pernambuco, 2004.

[3.19] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Pattern, Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, London, UK (1995)

[4.2] G. Booch, I. Jacobson, and J. Rumbaugh. Unified Modeling Language | User's Guide. Addison-Wesley, 1999.

[99] Uira Kulesza, Cláudio Sant' Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, Carlos Lucena, "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study," icsm, pp.223-233, 22nd IEEE International Conference on Software Maintenance (ICSM'06), 2006

[4.5] R. Elmasri and S. Navathe. Fundamentals of D, second edition, 1994.

[321]. <http://java.sun.com/javase/technologies/database/>

[654] <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/>