



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Validación de Arquitecturas a través de la Programación Orientada a Aspectos

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Claudio Ariel Graiño

Director: Fernando Asteasuain

Buenos Aires, 2015

VALIDACIÓN DE ARQUITECTURAS A TRAVÉS DE LA PROGRAMACIÓN ORIENTADA A ASPECTOS

Las arquitecturas de software se enfocan en el diseño y modelado de sistemas con un alto nivel de abstracción, revelando sólo las principales interacciones entre los elementos de software involucrados, ignorando detalles de carácter implementativo de bajo nivel.

Uno de los desafíos más relevantes dentro del mundo de las arquitecturas de software es poder determinar si, dado un sistema, éste cumple o satisface su especificación arquitectónica. Esto se debe principalmente a dos razones. Por un lado, es difícil llevar una trazabilidad de elementos arquitectónicos a código, ya que conviven en distintos niveles de abstracción. Por otro lado, la especificación arquitectónica sufre de un problema conocido como separación y erosión (drift & erosion). Esto ocurre cuando la especificación arquitectónica queda desactualizada frente a la evolución y/o cambios del sistema.

Muchas veces la estructura estática de un sistema (clases, paquetes, etc) no está relacionada con su comportamiento arquitectónico, por lo que una aproximación razonable para comprobar si un sistema cumple su especificación arquitectónica es observar su comportamiento en ejecución. En particular, dado que el comportamiento arquitectónico puede entrecruzar o atravesar distintas partes del sistema, la utilización de mecanismos avanzados de modularización como los propuestos en la Programación Orientada a Aspectos comprende un enfoque atractivo para encarar el problema descrito.

Dado este contexto, esta tesis pretende combinar mecanismos de la Programación Orientada a Aspectos para poder determinar si la implementación de un sistema cumple con su especificación arquitectónica. Específicamente, en este trabajo se atacará el problema de validar la utilización de conectores al reconstruir la arquitectura dinámicamente de un sistema en ejecución haciendo uso de la programación orientada a aspectos.

Palabras claves: Arquitecturas de Software, Programación Orientada a Aspectos, Validación de conectores, Análisis dinámico

ARCHITECTURE VALIDATION THROUGH ASPECT ORIENTED PROGRAMMING

Software architectures focus on system design and modelling at a high level of abstraction, by revealing only the main interactions between involved software elements, ignoring low level details.

One of the key challenges in the software architectures domain is to be able to determine if a given system's implementation satisfies its architecture. This problem arises mainly from two factors. On one hand, it's hard to translate architectural elements to code, since they live in different abstraction levels. On the other hand, the specification suffers from problems known as drift & erosion. This occurs when the architecture gets outdated in relation to the evolution and/or changes of the system.

Often, the static structure of a system (classes, packages, etc) is not related to the behaviour of the architecture. Therefore, a reasonable approach to check if a system satisfies its architecture is to analyse the behaviour in run time. In particular, since the architectural elements may cross various parts of the system, the use of advanced mechanisms of modularization as the ones proposed by the Aspect Oriented Programming might be an attractive approach to deal with the described problem.

Given this context, this thesis combines mechanisms of the Aspect Oriented Programming to determine if the implementation of a system satisfies its architecture. In particular, in this thesis we will tackle the problem of validating the use of connectors by dynamically generating the architecture of a executing system using aspect oriented programming.

Keywords: Software architectures, Aspect oriented programming, Connector validations, Dynamic analysis

Índice general

1..	Introducción	1
1.1.	Motivación	1
1.2.	Objetivos	2
1.3.	Estructura	3
2..	Conceptos Preliminares	5
2.1.	Arquitecturas de Software	5
2.1.1.	Definición	5
2.1.2.	Estilos arquitectónicos	5
2.1.3.	Vistas	7
2.2.	Programación Orientada a Aspectos	7
2.2.1.	Definición	7
2.2.2.	Conceptos claves	8
2.2.3.	AspectJ	8
3..	Desarrollo	11
3.1.	Elección de Conectores	11
3.2.	Formalización de Conectores	13
3.3.	Modelo de Aspectos	18
3.3.1.	Conectores	18
3.3.2.	Ignorados	21
3.3.3.	Observaciones	21
3.4.	Generalización de Conectores	23
3.5.	Analizadores	25
3.6.	Flow de la Herramienta	30
3.6.1.	Paso a Paso	30
3.6.2.	Ejemplo	30
3.7.	Observaciones	33
3.7.1.	Precedencia e Interferencia	33
3.7.2.	Concurrencia	34
3.7.3.	Alias	34
3.7.4.	Extra: Análisis Alternativos	34
4..	Caso de Estudio	36
4.1.	Sistema de ejemplo	37
4.2.	Proceso de anotación	38
4.3.	Análisis sobre el sistema anotado	45
4.3.1.	Componentes inalcanzables	45
4.3.2.	Conectores mal utilizados	45
4.3.3.	Extra: Notificaciones publish-subscribe	46
4.3.4.	Extra: Estilo pipe & filter	46
5..	Conclusiones y Trabajo Futuro	48

6.. Apéndice	50
6.1. Código Fuente	50
6.2. Output casos de estudio	50
7.. Referencias	53

1. INTRODUCCIÓN

En esta sección se introducirá el presente trabajo, detallando la motivación, los objetivos y cómo se estructura el informe.

1.1. Motivación

Las *arquitecturas de software* se enfocan en el diseño y modelado de sistemas con un alto nivel de abstracción, revelando sólo las principales interacciones entre los elementos de software involucrados, ignorando detalles de carácter implementativo de bajo nivel. De esa manera, facilitan la comunicación y entendimiento de los sistemas [1, 2].

Uno de los desafíos más relevantes dentro del mundo de las arquitecturas de software es poder determinar si un dado sistema cumple o satisface su especificación arquitectónica [3, 4, 5]. Esto se debe principalmente a dos razones. Por un lado, es difícil llevar una trazabilidad de elementos arquitectónicos a código, ya que conviven en distintos niveles de abstracción [3]. Por otro lado, la especificación arquitectónica sufre de un problema conocido como separación y erosión (*drift & erosion*) [6]. Esto ocurre cuando la especificación arquitectónica queda desactualizada frente a la evolución y/o cambios del sistema.

En este sentido, es importante tener una forma de reconstruir la arquitectura a partir del código. Por un lado, porque no es simple determinar si una implementación se corresponde con la arquitectura que implementa. Por otro, porque con el correr del tiempo es difícil mantener una consistencia entre la arquitectura y la implementación y puede ser útil tener una herramienta que ayude a validar dicha consistencia.

Existen tres enfoques principales para determinar la validez de una especificación arquitectónica frente a una dada implementación [3]. El primero de ellos es asegurar la consistencia mediante construcción. Esto se logra incluyendo constructores arquitectónicos en un lenguaje de implementación, como la herramienta ArchJava[3, 9]. Sin embargo, uno de los problemas detectados por este tipo de enfoque es su aplicabilidad, ya que dependen exclusivamente del alcance de las herramientas utilizadas. Por lo tanto no es posible combinarlas con sistemas que estén contruidos de otra manera, o que no puedan ser expresados utilizando dichas herramientas.

El segundo de ellos consiste en utilizar técnicas de análisis estático [10, 11, 12]. Sin embargo, también se han detectado falencias con este enfoque [3]. Pretender entender el funcionamiento de un sistema usando un análisis de código estático puede ser un problema porque los procesos y estructuras en tiempo de ejecución pueden no ser mapeadas fácilmente con las estructuras estáticas que son la base de muchos análisis arquitectónicos. De hecho, estructuras dinámicas pueden existir únicamente cuando el programa entra en ejecución.

Finalmente, el último enfoque consiste en observar el comportamiento dinámico de un sistema. Es decir, monitorear su ejecución. Existen ciertos aspectos que hacen particularmente desafiante este enfoque, que surgen de la diferencia de abstracción entre los conceptos arquitectónicos y la ejecución de código [3]. La creación y funcionamiento de un único componente arquitectónico puede incluir la interacción compleja entre distintas piezas de código, además de que su implementación puede ser distinta en cada sistema. Por ejemplo, una simple publicación en un conector Publish & Subscribe en una vista arquitectónica de componentes y conectores puede involucrar varias llamadas a funciones de código en diversas partes de un sistema. La herramienta *DiscoTect*[3] es una de las alternativas más representativas de este enfoque. Sin embargo, todo el código a desarrollar debe seguir un lineamiento en particular para poder ser integrado en la herramienta.

El trabajo de la presente tesis está centrado en este último enfoque: la reconstrucción de la arquitectura a partir del comportamiento dinámico del sistema. En este contexto, dado que el comportamiento dinámico arquitectónico puede entrecruzar o atravesar distintas partes del sistema, la utilización de mecanismos avanzados de modularización como los propuestos en la Programación Orientada a Aspectos [7] comprende una alternativa atractiva para encarar el problema descrito.

La principal vista arquitectónica enfocada en el comportamiento dinámico es la vista de Componentes y Conectores[2], en la cual se describen procesos en ejecución y la manera en que los mismos se comunican. En esta vista, los conectores juegan un rol fundamental, estableciendo cómo, cuándo, y bajo qué condiciones se comunican dos componentes. Luego, el estudio del presente trabajo se enfoca en validar la utilización de los conectores de esta vista, respondiendo esencialmente a la pregunta: **¿está la implementación del sistema comunicándose de la manera especificada por los conectores de la vista de Componentes y Conectores?**

Para resolver este problema, se utiliza la programación orientada a aspectos que será el medio por el cual se obtendrá, dinámicamente, la información relevante del sistema en ejecución. A partir de estos datos se realizarán diversos análisis que permitirán entender qué componentes existen en el sistema y cuáles son las diferentes formas que tienen para comunicarse. Para que los aspectos pueden aplicarse en la herramienta presentada en esta tesis, se debe anotar el código a validar según su comportamiento y relación con un conector de la arquitectura.

Finalmente, con todos los datos relevados se construye la arquitectura y en particular los conectores utilizados, los cuales mostrarán parte de la estructura dinámica del sistema que permitirá validar su utilización y determinar si la comunicación implementada se corresponde con aquello que estaba especificado.

1.2. Objetivos

El presente trabajo tiene como principal objetivo emplear técnicas de la Programación Orientada a Aspectos (POA) para observar el comportamiento dinámico de un sistema y validar que su comportamiento cumple lo especificado arquitectónicamente. En particular,

tal validación se concentrará en la utilización de los conectores de la vista de Componentes y Conectores.

Para cumplir con dicho objetivo, el análisis se basará en un conjunto conocido de conectores arquitectónicos: client/server, call asincrónico, call sincrónico, publish/subscribe, pipe&filter, router, broadcast y blackboard [8].

Para estos elementos se brindará una especificación formal, y se presentarán mecanismos para validar que un sistema dado los implemente correctamente utilizando herramientas de la POA. Tales mecanismos estarán basados en Aspectos que observarán la ejecución de un sistema relevando la información necesaria para reconstruir elementos arquitectónicos, y comprobar así que su utilización e interacción se corresponde con lo especificado.

El trabajo presentado en [3] es en este sentido un interesante punto de partida. En la presente tesis se buscará evitar que la validación dependa de la sintaxis del código como sucede en [3] para depender de la formalización de los conectores, aunque en un contexto más controlado y restringido, ya que será aplicable solamente a los elementos estudiados.

1.3. Estructura

En el capítulo 2 se explicarán los conceptos preliminares que son necesarios para entender el presente trabajo. Esto incluye entender el significado de las arquitecturas de software, y en particular el concepto de vista y la vista componente y conector; y de la programación orientada a aspectos, que será el medio por el cual se podrá desarrollar la herramienta.

En el capítulo 3 se explicará todo el proceso de desarrollo de la herramienta. Esto incluye la elección de los conectores que pueden utilizar las arquitecturas para realizar las comunicaciones entre los componentes y su formalización, los modelos de aspectos permitirán monitorear al sistema en ejecución, los analizadores que serán quienes procesen la información obtenida de dicho monitoreo y la generación de la arquitectura a partir de la información obtenida de dicho procesamiento. Finalmente esta sección finaliza con un ejemplo que explica en detalle cómo todo lo anteriormente mencionado funciona en conjunto para generar la arquitectura.

En el capítulo 4 se realizará un estudio de un caso de estudio, en el cual se ilustra todo el potencial de la herramienta y se describen las particularidades que se pueden encontrar a la hora de utilizar la herramienta para validar la utilización de conectores en una arquitectura. Ese proceso se inicia anotando el código de la implementación de cierta forma, para finalmente realizar diferentes análisis sobre el *output* que genera la herramienta.

En el capítulo 5 se muestran las conclusiones del trabajo realizado y posibles continuaciones del mismo como trabajo futuro.

En el capítulo 6 se muestra el detalle del código fuente de la herramienta y cómo se ejecuta, además de los *outputs* del caso de estudio.

2. CONCEPTOS PRELIMINARES

2.1. Arquitecturas de Software

En esta sección se explicarán los conceptos relacionados a las arquitecturas pertinentes al trabajo a desarrollar. Para mayor profundidad en estos temas se recomienda leer [1, 2, 6].

2.1.1. Definición

Si bien no existe una definición formal, una arquitectura de un programa o sistema de software puede verse como la estructura o estructuras del sistema, que incluye elementos de software, las propiedades externamente visibles de éstos y las relaciones entre ellos. La arquitectura incluye sólo la parte pública de las interfaces de los elementos. En cambio, detalles privados, que tienen que ver únicamente con la implementación interna, no son considerados arquitectónicos [13].

Es decir, la arquitectura comprende[14]:

- Un conjunto de componentes de software, conexiones y restricciones.
- Una colección de requerimientos (necesidades de los stakeholders).
- Una razón lógica que demuestre que los componentes, conexiones y restricciones define un sistema que, de ser implementado, satisfaga todos los requerimientos.

Además, las arquitecturas de software pueden verse como un intermediario entre los requerimientos de un sistema y su implementación. Al proveer una abstracción de alto nivel de la estructura del sistema, favorece el entendimiento del programa en términos más simples que aquellos de la implementación (los cuales incluyen clases, métodos, etc). Más aún, la descripción de la arquitectura podría ayudar a entender si un sistema satisface requerimientos importantes al razonar la estructura más abstractamente. Finalmente, la arquitectura forma un camino para la implementación del sistema, indicando los lugares principales de procesamiento o almacenamiento de datos, los canales de comunicación y las interfaces que se utilizan para dichas comunicaciones[2].

2.1.2. Estilos arquitectónicos

Los estilos (o patrones) arquitectónicos son un conjunto de características de tipos de relaciones y elementos, y restricciones de cómo deben usarse. Luego, los estilos definen

familias de sistemas en términos de patrones en la estructura de la arquitectura. Más específicamente, determina el vocabulario de los componentes y de los conectores que pueden ser usados en el estilo, junto con las restricciones de cómo pueden ser combinados, que pueden incluir restricciones topológicas (e.g. que no puedan existir ciclos)[15].

Las principales utilidades de aplicar estilos son, por un lado, mejorar la comunicación y el entendimiento de la arquitectura, y por otro, permitir atacar problemas nuevos aprovechando de aprendizajes y experiencias pasadas (“soluciones viejas para problemas nuevos”).

Existen muchos y diferentes estilos arquitectónicos, cada uno con sus particularidades, diferencias y similitudes. La siguiente figura ilustra una posible taxonomía [15]

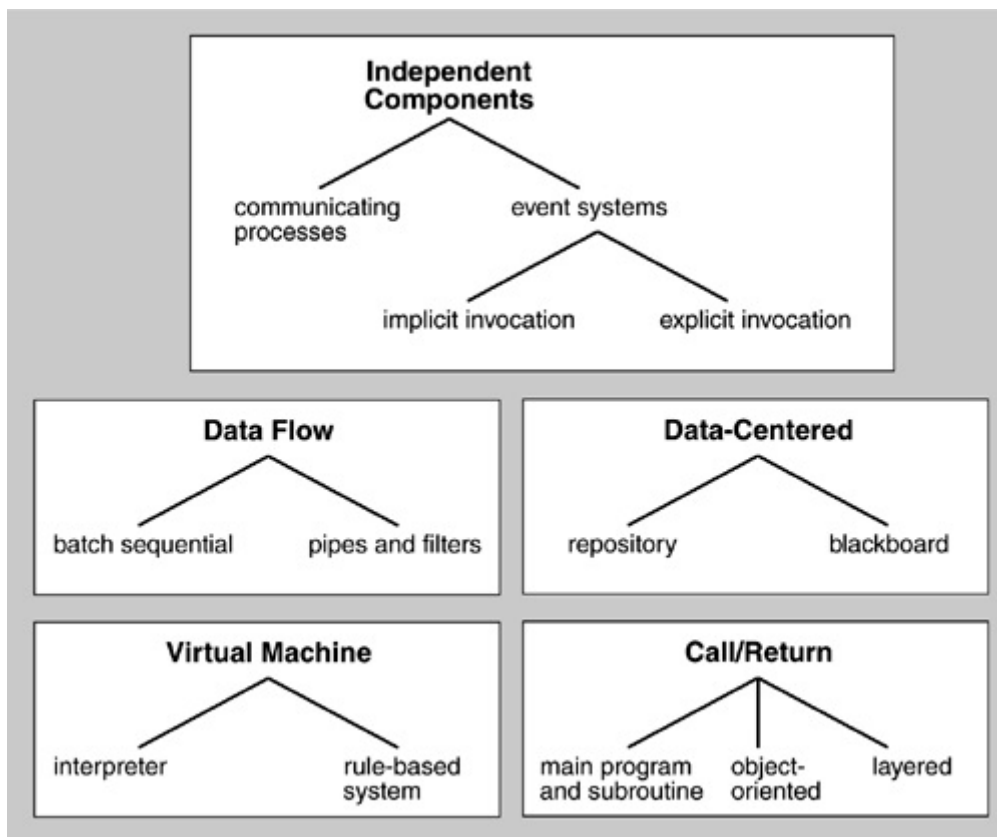


Fig. 2.1: Taxonomía de estilos arquitectónicos

Sólo a modo ilustrativo, se decide presentar brevemente a continuación uno de estos estilos: *pipe & filter*, ya que es el único estilo en el cual se profundizará en la presente tesis. En este estilo, cada componente tiene un conjunto de inputs y otro de outputs. Cada uno de estos componentes, lee de un stream de datos en sus inputs y produce otro stream de datos en sus outputs, comúnmente luego de aplicar algún tipo de procesamiento. Los conectores (*pipes*) funcionan como un conducto por el que pasan los datos del *output* de un filtro al *input* de otro.

2.1.3. Vistas

Una *vista* es una representación del sistema desde cierta perspectiva [1]. Esta perspectiva implica que se ponga el foco en ciertos aspectos específicos, obviando detalles que permitan tener un modelo simplificado pero que involucre exclusivamente a los aspectos del enfoque. La vista permite examinar a un sistema de una manera que sólo se visualice aquello que es relevante para quien examine el sistema.

Existen principalmente tres tipos de vistas:

- Estructuras de módulos, donde los módulos son unidades de implementación, viendo al sistema desde un lugar meramente implementativo.
- Estructuras de asignación, que muestra la relación entre los elementos de software y los elementos de uno o más entornos externos en los cuales el software es creado o ejecutado.
- Estructuras de Componentes y Conectores (C&C), donde los elementos (componentes) son procesos que se ejecutan en el sistema y los conectores la forma en que éstos se comunican entre sí.

Para una mayor descripción sobre vistas arquitectónicas, se deriva a la lectura de [8].

En el presente trabajo, nos concentraremos en la vista de componentes y conectores. Allí conviven los componentes, que son procesos (entidades en ejecución) independientes (están aislados de otros componentes), y conectores, que son procesos que se encargan de unir a los componentes. En los conectores, además, se tiene información de la cardinalidad de los componentes que soporta en la relación, las interfaces que se utilizan para comunicarse y qué propiedades tienen estas comunicaciones. Una definición más profunda de conectores y el conjunto de conectores a estudiar en la presente tesis se encontrará en las secciones *Elección de Conectores* y *Formalización de Conectores*.

2.2. Programación Orientada a Aspectos

En esta sección se explicarán los conceptos relacionados a la programación orientada a aspectos relevantes al trabajo.

2.2.1. Definición

La programación orientada a aspectos (POA, o AOP en inglés) [7] es un paradigma de programación que intenta aumentar la modularización de un sistema al permitir el desacoplamiento de conceptos transversales (cross-cutting concerns) al dominio del problema. El ejemplo clásico es el de logging, que, en general, no es un concepto que exista en el dominio del problema que se está modelando pero que debe estar presente a lo largo

del sistema, haciendo que esté acoplado a diferentes módulos del sistema sin permitir una clara separación de éste.

Para lograr este cometido, lo que propone la POA es invertir la dependencia de estos conceptos transversales. Es decir, en vez de que los diferentes módulos tengan que depender de estos conceptos, se busca que exista un módulo que modele a dichos conceptos y éstos sean los que se encarguen de acoplarse a los módulos que los necesiten. Volviendo al ejemplo del logging, en vez de que todos los métodos tengan que llamar al módulo de loggeo, que el módulo de loggeo se encargue por sí mismo de loggear los diferentes métodos.

A continuación se describen los conceptos claves de la orientación a aspectos que permiten lograr su objetivo.

2.2.2. Conceptos claves

Joinpoints:

Los *joinpoints* son puntos en el flujo de un sistema en ejecución en los que se puede obtener información. Por ejemplo, envíos de mensajes, ejecución de métodos, inicializaciones de objetos, etc.

Pointcuts:

Los *pointcuts* son condiciones para poder filtrar a los *joinpoints* y para definir qué información obtener de éstos. Por ejemplo, detectar los envíos de mensajes que tienen nombre empezando con “get”.

Advices:

Los *advices* representan código que se ejecuta cuando se alcanza un *joinpoint*. En general se utilizan en conjunto con los *pointcuts* para poder definir diferentes acciones a realizar dependiendo del tipo de *joinpoint* que se haya ejecutado.

Aspects:

Los aspectos serán los módulos que modelen a los conceptos transversales (*cross-cutting concerns*). En ellos se pueden definir los diferentes *pointcuts* y *advices* que ayuden a modelar dicho módulo.

2.2.3. AspectJ

AspectJ [16] es un framework que permite implementar la programación orientada a aspectos en sistemas implementados con el lenguaje de programación Java. Por esto, la sintaxis que se utiliza es muy similar a la de dicho lenguaje.

Los aspectos se declaran de forma similar a una clase:

```
1 aspect MyAspect {
2     //pointcuts and advices
3 }
```

Fragmento de Código 2.1: Definición de Aspecto

Dentro de los aspectos se pueden declarar *pointcuts*, indicando en cada uno qué *joinpoints* se quiere detectar. Existen distintos tipos de *joinpoints*, según el lugar donde se quiera aplicar el comportamiento de un aspecto. Los que utilizaremos a lo largo del trabajo son: **call**, que permite detectar el envío de un mensaje; **within**, que permite detectar *joinpoints* dentro de cierta clase; y *withincode*, que permite detectar *joinpoints* que se generan desde dentro de ciertos métodos. Dentro de cada uno de éstos, se debe especificar cómo es el método, describiendo el tipo que retorna, el nombre del mensaje y el nombre de la clase (se utiliza * como *wildcard* cuando se quiere permitir que tome cualquier valor posible).

Como ejemplo, en el siguiente fragmento de código se muestra una implementación de *pointcuts* para un aspecto nombrado *Countries*. En primer lugar, se define un *pointcut* llamado *methodsNamedGetCountry*, el cual podrá capturar todas las llamadas que se realicen al método *getCountry*, sin importar qué parámetros reciba o qué tipo retorne. Luego, se define otro *pointcut* llamado *callsFromClassStates*, que podrá capturar todas las llamadas que se realicen a cualquier método desde la clase *States*.

```
1 aspect Countries {
2     pointcut methodsNamedGetCountry(): call(public * getCountry(..));
3     pointcut callsFromClassStates(): call(* * (..) && within(States);
4 }
```

Fragmento de Código 2.2: Implementación de pointcut

Por último, se debe definir qué acción se quiere realizar para cada uno de estos *pointcuts*. Estas acciones (llamadas *advices*) se definen indicando qué acción se quiere realizar y a partir de qué momento de la ocurrencia del *pointcut*. Este momento se puede definir de tres maneras distintas: antes (**before**), significa antes de que se ejecute el punto en la ejecución del programa que se está detectando; después (**after**), significa después de que ejecute el punto en la ejecución del programa que se está detectando; o **around**, que permite reemplazar la ejecución original (es decir, la que fue captada por el *pointcut*) por el comportamiento del *advice*. En el siguiente fragmento de código se puede ver un ejemplo para entender cómo se implementan los *advices*:

```
1 aspect Countries{
2     pointcut methodsNamedGetCountry(): call(public * getCountry(..));
3
4     before(): methodsNamedGetCountry(){
5         // do something before getCountry method is called
6     }
7     after(): methodsNamedGetCountry(){
8         // do something after getCountry method is called
9     }
10    around(): methodsNamedGetCountry(){
11        // do something to replace the method getCountry
12    }
13 }
```

Fragmento de Código 2.3: Implementación de advices

Se define un *pointcut* que captará las llamadas al método *getCountry* y se le agregan tres advices:

En el primero, *before*, se define el *advice* que será ejecutado después de la llamada al método *getCountry*. Es decir, cada vez que se llame al método *getCountry*, antes de que dicho método pueda ejecutarse, se ejecutará el *advice* definido.

En el segundo, *after*, se define el *advice* que será ejecutado antes de la llamada al método *getCountry*. Es decir, luego de cada llamada a *getCountry* (cuando el método finalice de ejecutarse y antes de que vuelva el control al emisor del mensaje), se ejecutará el *advice* definido.

Por último, en el *around* se define el *advice* que será ejecutado en lugar del método *getCountry*. Es decir, cada vez que sea llamado el método *getCountry*, se ejecutará este *advice* definido en lugar del método original.

3. DESARROLLO

3.1. Elección de Conectores

En esta sección se muestra, explica y justifica la elección del conjunto de conectores elegido para modelar las posibles arquitecturas validables.

Los conectores son una parte fundamental para entender el funcionamiento en tiempo de ejecución de un sistema. Permiten entender cómo se comunican entre sí las diferentes partes del sistema, que en conjunto formarán la arquitectura. También permiten expresar cómo progresan los datos en el sistema, qué protocolos se utilizan, qué comunicaciones pueden ser bloqueantes, asincrónicas o pueden ejecutarse concurrentemente, detectar posibles defectos o vulnerabilidades del sistema, entre otros.

En la literatura existen numerosas propuestas de clasificación y taxonomía de conectores, mostrando distintas variedades, características y complejidades; y cada una con su notación y nomenclatura [6, 8, 13]. Para la presente tesis, se decidió utilizar un conjunto fijo de conectores. Si bien esta elección es arbitraria, la misma permite ilustrar la mayoría de las interacciones de las interacciones entre componentes, tal es así que es similar a la utilizada por Red Hat [17] para la definición arquitectónica de sus productos [18].

Los conectores elegidos son:

- **Mensajes directos, sincrónicos o asincrónicos.** Es el conector más simple, sólo es un mensaje entre una fuente y un destinatario con la particularidad de que en un caso se espera una respuesta y en el otro no.
- **Pipe.** Permite que dos conjuntos de componentes se puedan comunicar a través de una estructura intermedia que guarde los mensajes sin que los componentes tengan que lidiar con la comunicación directa entre sí.
- **Publish-Subscribe.** Permite que componentes (suscriptores) puedan recibir (suscribirse) determinados mensajes que emite otro componente (publicador), con una estructura intermedia que se encargue tanto de abstraer a suscriptores de publicadores como de manejar la comunicación de nuevas publicaciones cuando y como corresponda a los suscriptores.
- **Client-Server.** Permite a múltiples componentes (clientes) enviar mensajes, con una respuesta, a un mismo componente (servidor), algo que no es posible con un simple call sincrónico ya que el receptor en ese caso no puede procesar múltiples respuestas simultáneamente.
- **Router.** Permite enviarle un mismo mensaje a un subconjunto de componentes. Existe un caso particular (**Broadcast**) para enviarle a todos los componentes del

conjunto.

- **Blackboard.** Permite tener un repositorio con eventos. Se puede leer/escribir de él y suscribirse a ciertos eventos de escritura, al igual que un publish subscribe.

3.2. Formalización de Conectores

En esta sección se detalla el proceso de formalización de los conectores, mostrando el comportamiento y características de cada uno de ellos.

La formalización de los conectores permite, en primer lugar, dejar asentado cómo es el comportamiento de cada uno de ellos, y así evitar confusiones y ambigüedades de qué representan. Además, permite entender cómo se producen las interacciones entre los diferentes componentes que participan en cada conector, desde qué protocolo utilizan para conectarse hasta qué características relevantes existen (como bloqueos o mensajes asíncronos). Por último, servirá para aprender sobre las diferencias y similitudes entre los conectores.

Todas estas cuestiones serán útiles para entender cómo se debe actuar para monitorear y detectar las utilidades de estos conectores al momento de analizar las implementaciones de las diversas arquitecturas en tiempo de ejecución, tal cual es el objetivo del presente trabajo.

Una formalización completa incluiría los puertos de los componentes, los roles de los conectores y la manera en que se conectan los roles de los conectores a los puertos de los componentes (e.g. [19]). Dado que el foco de la presente tesis es validar la manera en que se comunican los componentes, es decir la utilización de conectores, alcanza con una formalización más simple, mostrando únicamente cómo se comporta cada conector elegido. Extender este trabajo a una formalización más compleja queda planteado como trabajo futuro.

Se utilizará para la formalización dos tipos de notaciones: diagramas de estados con las transiciones $?$ (que representa transiciones pasivas, donde es otro el que ejecuta la acción) y $!$ (que representa transiciones activas, donde sí se toma la acción para realizar el pasaje) para algunos (*call sincrónico*, *call asíncrono* y *client-server*); y para otros diagramas de componentes, explicando en alto nivel cómo se lleva a cabo la comunicación (*pipe*, *publish-subscribe*, *Router/Broadcast*, *Blackboard*).

A continuación se describe, para cada uno de los conectores elegidos, el comportamiento de cada uno de ellos mostrando qué protocolo se utiliza y, cuando aplica, alguna característica particular de la comunicación.

Call Sincrónico

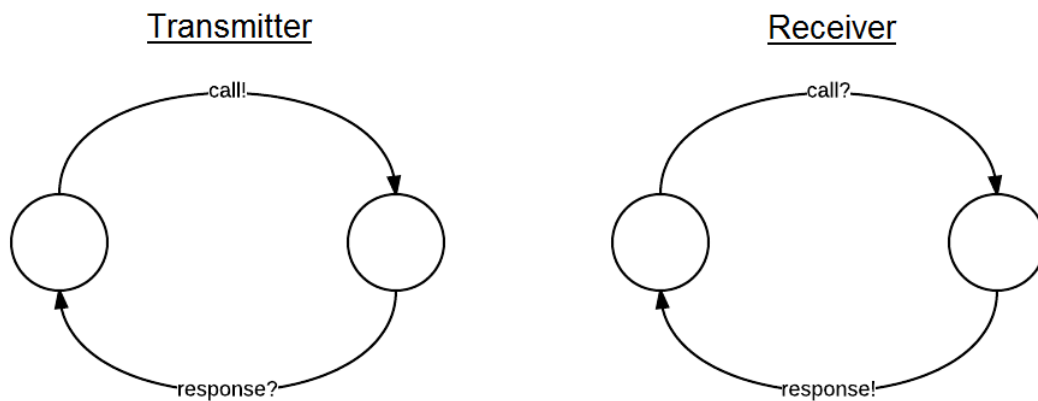


Fig. 3.1: Representación de estados de conector Call Sincrónico

En este caso tenemos dos estados, como se ve en la figura 3.1. El emisor inicia en un estado en el cual puede estar realizando cualquier procesamiento o comunicación. Al efectuar una llamada sincrónica, pasa a un segundo estado del cual sólo podrá salir cuando el receptor realice la respuesta, y por ello queda bloqueado hasta que eso suceda.

La característica más interesante de este conector es justamente el bloqueo que se produce al efectuar la llamada y que ésta tiene como fin una respuesta.

Call Asincrónico

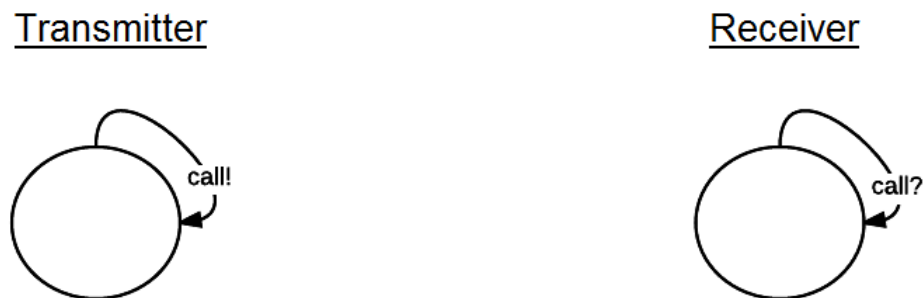


Fig. 3.2: Representación de estados de conector Call Asincrónico

Como se ve en la figura 3.2, en este caso existe un único estado y es una muestra de la característica más relevante del conector: la asincronicidad. El estado inicial, desde el cual puede realizar cualquier actividad, es el mismo que tiene luego de enviar el mensaje asincrónico, ya que no se bloquea esperando una respuesta como en el *call sincrónico*.

A diferencia del *call sincrónico*, el *call asincrónico* no tiene una respuesta.

Pipe

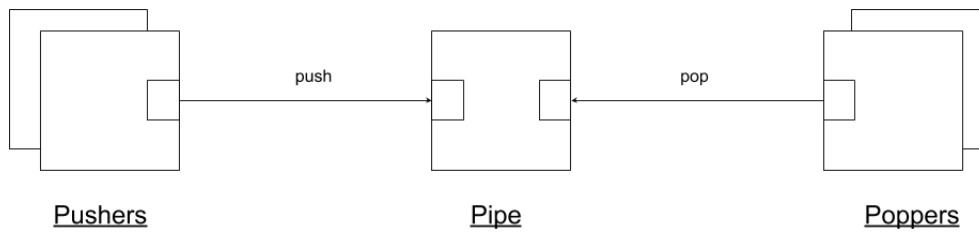


Fig. 3.3: Representación de protocolos de conector Pipe

Como se ve en la figura 3.3, en este conector no existe un emisor y un receptor, sino que existe un componente central llamado *pipe* y dos conjuntos de componentes que emiten unos *push* y otros *pop*, como se puede observar en la figura. Esos conjuntos no interactúan entre sí en el contexto del conector, siempre lo hacen a través del *pipe*, que actúa como un intermediario de la información.

Publish-Subscribe

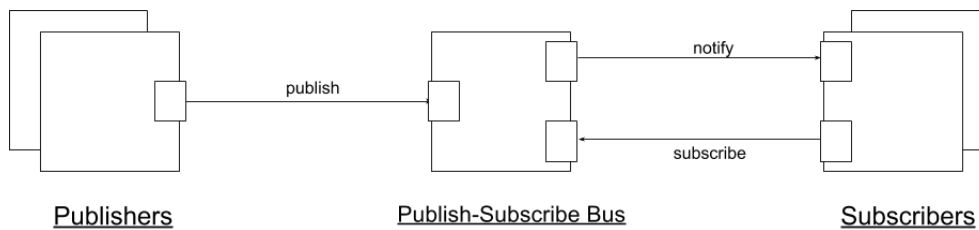


Fig. 3.4: Representación de protocolos de conector Publish-Subscribe

Como se muestra en la figura 3.4, los componentes que interactúan en este conector son: el *publish-subscribe bus*, el conjunto de suscriptores y el conjunto de publicadores. Al igual que en el *pipe*, el *publish-subscribe bus* actúa como intermediario entre suscriptores y publicadores.

El flujo esperado es que los suscriptores se suscriban a ciertos eventos enviándole un mensaje *subscribe* al *bus*. Por otro lado, los publicadores realizan publicaciones emitiéndole un mensaje *publish* al *bus*, y éste a su vez debe comunicarse con quienes estén suscritos a ese evento para notificarlos de la publicación emitiendo un mensaje de notificación *notify*.

Client-Server

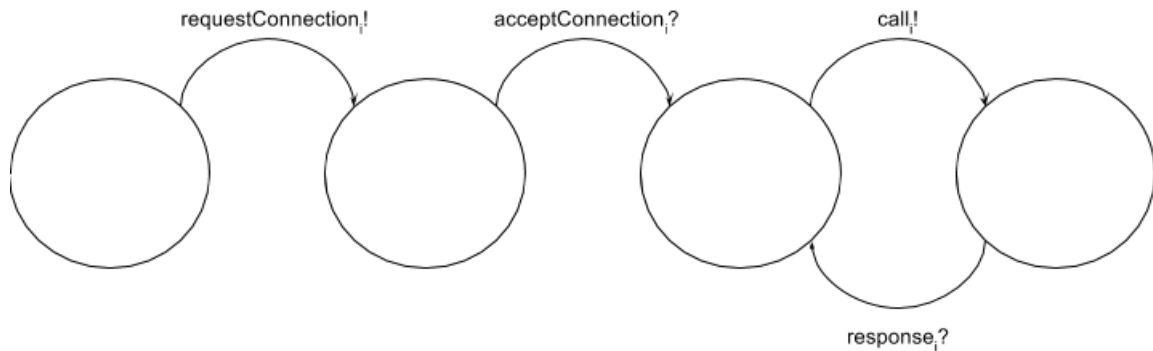
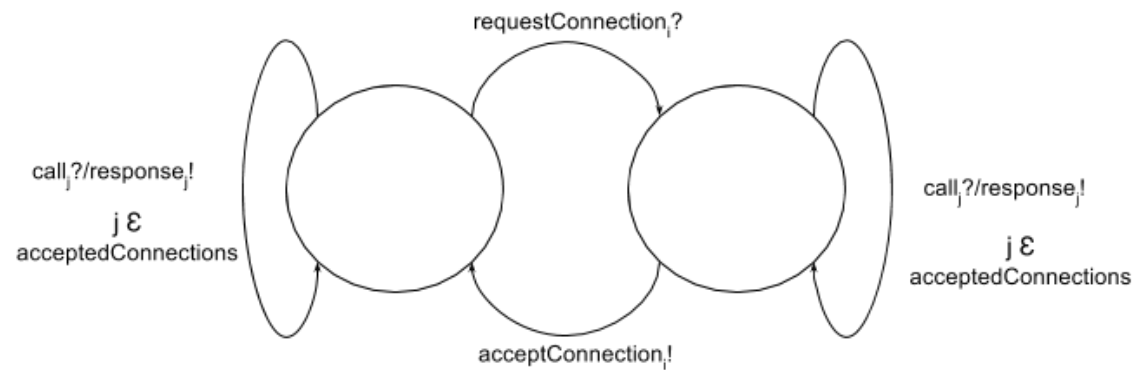
Client**Server**

Fig. 3.5: Representación de estados de conector Client-Server

En este conector volvemos a encontrarnos con sólo dos componentes que interactúan, por un lado los clientes y por otro el servidor, como se muestra en la figura 3.5.

Para el cliente, la comunicación se realiza primero con un *requestConnection* para poder comunicarse, seguido del mensaje sincrónico al servidor. Al igual que en el caso del *call sincrónico*, este mensaje será bloqueante para el cliente.

Para el servidor, deberá poder establecer conexiones mediante el *requestConnection* y deberá poder recibir múltiples mensajes, emitiendo una respuesta, sin la necesidad de bloquearse en cada uno de ellos. Esto implica que los mensajes serán bloqueantes para el cliente pero no para el servidor.

Router/Broadcast

En la figura 3.6 se puede observar la representación del conector *Router*. Es similar al caso del *call asincrónico* pero con un intermediario (sea *router* o *broadcaster*) que se encarga de emitir cierto mensaje a un grupo de componentes destinatarios.

La diferencia entre *router* y *broadcast* es más bien semántica ya que la diferencia

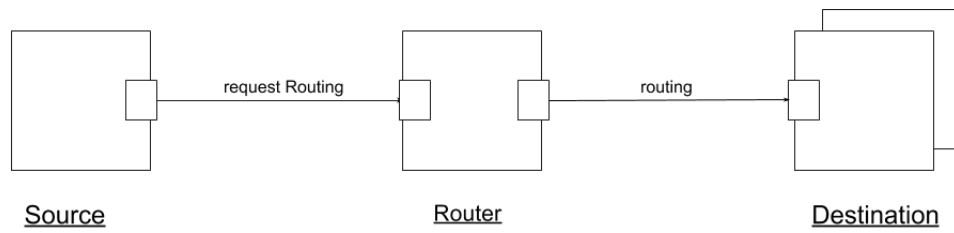


Fig. 3.6: Representación de protocolos de conector Router

implementativa es bastante sutil. En un caso se emite a un conjunto de componentes y en el otro a un subconjunto de un conjunto de componentes,

Blackboard

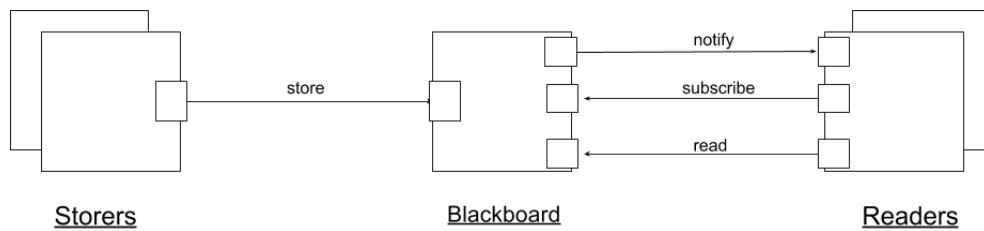


Fig. 3.7: Representación de protocolos de conector Blackboard

En la figura 3.7 se pueden observar los protocolos del conector *Blackboard*. Es, básicamente, un repositorio con la capacidad de suscribirse a eventos. Este repositorio puede recibir mensajes como *store*, *read*, que serían parte del protocolo básico de un repositorio. Pero además, tiene el comportamiento de un *publish-subscribe*, permitiendo la suscripción a eventos que se producen en los *store*.

3.3. Modelo de Aspectos

En esta sección se detallan los modelos de aspectos implementados para monitorear al sistema en ejecución.

Los aspectos serán los encargados de monitorear la ejecución del sistema y relevar los conectores utilizados. Más concretamente, los aspectos son los objetos que modelan cómo se detectan los diferentes tipos de comunicaciones de cada conector en el código en ejecución. Luego, por cada conector tendremos definido un aspecto y en él estarán descritas, según su protocolo y sus características (detallados en la formalización), las diferentes formas (llamadas pointcuts) de captar todos los escenarios relevantes (llamados join points) del código en ejecución.

La formalización de los conectores nos van a ayudar a entender qué hay que detectar y cómo hacerlo. Por un lado, se tiene en cuenta el protocolo que se utiliza en las comunicaciones de cada conector ya que es un factor fundamental en la implementación de cada conector. Por otro, es importante entender el flujo de mensajes y circunstancias particulares, que van más allá del protocolo, porque deberán ser tenidos en cuenta al momento de detectar esas situaciones muy particulares.

Para modelar los protocolos, como fueron detallados en la formalización, se utilizarán anotaciones. Es decir, se anotará el código del sistema a analizar. Estas anotaciones, que se aplican a métodos o clases según el caso que corresponda, permiten detallar en la implementación qué protocolo de qué conector se está modelando. De esa manera, cuando se utilicen dichos métodos o clases, según corresponda, los aspectos podrán detectar que se están utilizando ciertos protocolos para comunicarse.

Los pointcuts aquí mostrados están simplificados con respecto a los que se encuentran en el código para mostrar la información relevante. Los verdaderos pointcuts tienen algunas condiciones para evitar llamados desde main o a métodos/clases internas de java, etc. Para el código exacto referir al apéndice de la presente tesis.

En la sección 3.6. *Flow de la Herramienta* se mostrará un ejemplo donde se verá el funcionamiento interno completo de la herramienta, incluyendo a los modelos aquí definidos.

3.3.1. Conectores

Call Sincrónico

Este conector es el más simple. Asumimos que un call sincrónico es cualquier mensaje que se envía. Por ello, como se muestra en el fragmento de código, el único pointcut que vamos a necesitar definir es un call a cualquier método. [línea 1]

```
1 call(* *(..))
```

Fragmento de Código 3.1: Pointcuts del conector Call Sincrónico

La directiva `call` representa la acción de enviar un mensaje. Además, los asteriscos representan wildcards para ciertos atributos del método, en este caso el primero corresponde al tipo de la respuesta del método y el segundo al nombre. Luego, el pointcut detecta todas las llamadas a cualquier método, que devuelva cualquier tipo y con cualquier número y tipo de argumentos.

Call Asincrónico

Este conector es similar al `call` sincrónico pero de alguna manera debemos modelar el asincronismo del mensaje, y se tomó la decisión de hacerlo mediante `threads`. Luego, vamos a detectar los `calls` asincrónicos como mensajes que se envían desde `threads` como se muestra en [línea 1]

```
1 call(* *(..)) && withincode(void java.lang.Runnable.run());
```

Fragmento de Código 3.2: Pointcuts del conector Call Asincrónico

La directiva `within code` detecta mensajes enviados en la definición del código de un método. En este caso se verifica que se ejecute desde un `thread`.

Pipe

Como vimos en la formalización, este caso involucra más envíos de mensajes, y por ello es que debemos detectar los mensajes que son parte de su protocolo: `pop` y `push`. Para lograrlo, vamos a detectar los mensajes que se envían a los métodos que representan dichos mensajes, que deben estar anotados como `PipePop` [línea 1] y `PipePush` [línea 2] respectivamente.

```
1 call(@PipePop * * (..))
2 call(@PipePush * * (..))
```

Fragmento de Código 3.3: Pointcuts del conector Pipe

Se antepone el nombre de la anotación al del método para describir que se están detectando los mensajes enviados que están `taggeados` con dicha anotación.

Publish-Subscribe

Similar al caso de `Pipe`, también debemos detectar las llamadas de `publish` [línea 1] y de `subscribe` [línea 2], utilizando las anotaciones `Publish` y `Subscribe`.

Adicionalmente, debemos detectar todas las llamadas que se producen al momento de notificar a los suscriptores de una nueva publicación (utilizando la condición `withincode(@Publish * * (..))` para detectar todas las llamadas que se producen desde dicho

método). Luego, interesa diferenciar si esas llamadas son para notificar [línea 3] o no [línea 4], ya que se las quiere manejar de forma distinta. En caso de que sea una notificación, se utilizará para seguir el flujo de publicaciones y notificaciones. En caso contrario, se marcará como procesado para que no sea capturado por otros aspectos.

```

1  call(@Publish * * (...))
2  call(@Subscribe * * (...))
3  withincode(@Publish * * (...)) && call(@PublishNotification * * (...))
4  withincode(@Publish * * (...)) && !call(@PublishNotification * * (...))

```

Fragmento de Código 3.4: Pointcuts del conector Publish-Subscribe

Client-Server En el caso del Client-Server, por un lado se monitorea el mensaje que se envía al servidor para establecer la conexión (que debe ser anotado como RequestConnection) [línea 1], y por otro los mensajes que se envían al objeto que representa a la conexión propiamente dicha (que deberá ser anotada como ClientServerConnection) [línea 2].

```

1  call(@RequestConnection * * (...))
2  call(* @ClientServerConnection *.* (...))

```

Fragmento de Código 3.5: Pointcuts del conector Client-Server

Para detectar los mensajes enviados que pertenecen a clases anotadas de cierta manera, se debe especificar la anotación luego del primer wildcard.

Router/Broadcast

Tanto para Router como para Broadcast, que se comportan análogamente, debemos detectar en primera instancia el llamado a generar el route/broadcast [línea 1] (cuyo método debe ser anotado como Route o Broadcast respectivamente). Luego, debemos detectar qué mensajes se envían desde ese método [línea 2], ya que los receptores de esos mensajes serán los receptores del route/broadcast.

```

1  call(@Route * * (...))
2  call(* * (...)) && withincode(@Route public * * (...))

```

Fragmento de Código 3.6: Pointcuts del conector Router

Blackboard

Análogo al conector Publish-Subscribe, en el Blackboard vamos a detectar el store [línea 1]/subscribe [línea 2] de información del repositorio y los mensajes de notificación a los suscriptores [línea 3]. Al igual que en el caso del Publish-Subscribe, debemos detectar no sólo los mensajes enviados para notificar sino cualquier también cualquier otro que se envíe desde dicho método [línea 4].

Además, hay un mensaje adicional en el protocolo del Blackboard que permite leer información directamente del repositorio, que deberá ser anotado como Read [línea 5].

```
1  call(@Store * * (..))
2  call(@SubscribeBlackboard * * (..))
3  call(@StoreNotification * * (..)) && withincode(@Store * * (..))
4  withincode(@Store * * (..)) && !call(@StoreNotification * * (..))
5  call(@Read public * * (..))
```

Fragmento de Código 3.7: Pointcuts del conector Blackboard

3.3.2. Ignorados

Además de los aspectos utilizados para detectar las interacciones de los componentes, se implementó también un aspecto adicional. Este aspecto no está relacionado a ningún a ningún conector y permite ignorar ciertas partes del sistema (que pueden ser métodos o clases completas). En particular se puede utilizar para ignorar ciertas partes implementativas de los conectores que no son parte de su funcionalidad estándar pero se quieren mantener ocultas.

Un caso de uso común es el de agregarle métodos a los conectores que no son parte de su protocolo, por ejemplo agregar un método `#hasElements` al pipe para saber si éste tiene elementos. De no tener la posibilidad de ignorar a este método, éste se mostraría en la arquitectura como una relación entre quien llame a ese método y el *pipe*, generando una arquitectura distinta a la esperada. Específicamente, esa relación sería un *call sincrónico* ya que se está comunicando sincrónicamente entre dicho emisor y el *pipe*.

Además, puede permitir realizar validaciones de la utilización de conectores en sólo una parte de la arquitectura, al poder utilizar únicamente ciertas porciones del código que sean de interés y descartar las demás, anotándolos como ignorados. Esto puede servir, también, para realizar una validación de la arquitectura de forma iterativa.

3.3.3. Observaciones

Estas implementaciones fueron el resultado de un proceso iterativo en base a distintos tipos de sistemas, utilizando varias implementaciones para los conectores y viendo qué errores surgían y cómo se podía mejorar el modelo de aspectos. Inicialmente, se tomaron sistemas sencillos que utilizaban únicamente un conector. Luego, se agregaron sistemas que implementaban un conector dado de diversas maneras. Finalmente, se utilizaron sistemas que utilizaban varios conectores simultáneamente.

A través de distintas iteraciones, el modelo de aspectos fue evolucionando y ganando madurez hasta llegar al modelo descrito anteriormente. Algunos ejemplos de estas mejoras encontradas se describen a continuación.

Un caso fue el de mensajes enviados desde un conector que no son necesarios para detectar la presencia del conector (por ejemplo la notificación a suscriptores en Publish-Subscribe o Blackboard). Inicialmente al preocuparse únicamente por detectar la presencia

del conector, teniendo en cuenta que estos mensajes no son necesarios para detectar el uso del conector ya que alcanza con suscripciones y publicaciones, estos mensajes eran ignorados. Luego, al ver la arquitectura completa, si estos mensajes no eran capturados por el conector particular de cada uno ellos, se generaría una arquitectura incorrecta al establecer una relación con un conector call return en las llamadas de dichos mensajes. A partir de estos casos, se actualizó el modelo de aspectos para tener en cuenta estas circunstancias.

Otro caso, más implementativo, estuvo relacionado con cómo se detecta que un mensaje es enviado desde cierto lugar. Esto se observa en varios conectores, por ejemplo en el Router al producir el routeo. Inicialmente se veía que estuviera en el contexto del mensaje enviado. El problema que se generaba es que los mensajes que se enviaban desde el receptor del router, también eran captados como parte del routeo. Por esto se debió modificar para que en vez de tener en cuenta el contexto, solo chequee que los mensajes enviados estén presentes en la declaración del método.

3.4. Generalización de Conectores

En esta sección se detalla el proceso de generalización de los conectores que posibilitará el posterior análisis de los datos capturados por los aspectos para la construcción de la arquitectura.

El análisis del comportamiento de los conectores elegido permitió poder generalizar la manera en que éstos funcionan en tres grandes categorías: directo, inactivo y activo.

La idea detrás de esta generalización es abstraer, más allá de protocolos y características particulares de cada conector, su comportamiento. Es decir, cómo se comunican en cada conector, si hay un intermediario o no, cómo se comporta éste, etc. Esta generalización va a permitir que los posteriores análisis sobre la información capturada por los aspectos pueda facilitarse, al no tener que ver cada caso puntual sino las pocas generalizaciones que queden de este estudio.

A continuación se muestra un resumen de los conectores y a qué abstracción pertenecen.

Conector\Analizador	Directo	Inactivo	Activo
Call Sincrónico	X		
Call Asincrónico	X		
Pipe		X	
Publish-Subscribe		X	
Client-Server	X		
Router			X
Broadcast			X
Blackboard		X	

Tab. 3.1: Relación entre Conectores y Analizadores

En primer lugar tenemos a los conectores que conectan a dos objetos **directamente**, sin intermediarios. Es decir, que la relación surge solamente de que un objeto le envíe cierto mensaje a otro. Están incluidos el Call Sincrónico, el Call Asincrónico y el Client-Server. Todos estos conectores necesitan únicamente de un mensaje para establecer la relación, y ese mensaje se hace entre las dos puntas de la conexión.

Por otro lado, tenemos a los conectores que usan a un intermediario pero éste no necesita ejecutar ninguna acción más que recibir los mensajes de los objetos que intervienen en la relación, por ello lo llamamos **inactivo**. Están incluido el Pipe, el Publish-Subscribe y el Blackboard. En los tres, la relación se establece cuando los involucrados envían el mensaje correspondiente (por ejemplo subscribe) al intermediario (por ejemplo publish-subscribe bus) y el intermediario no necesita realizar ninguna acción adicional. En caso

tanto del Publish-Subscribe como del Blackboard, el intermediario puede generar acciones, por ejemplo al notificar a los suscriptores, pero no es algo irrelevante para la determinación de que se utiliza el conector en cuestión.

Por último, se encuentran los conectores que sí necesitan que un intermediario ejecute ciertas acciones sobre otros objetos para que éstos sean parte de la relación, por ello su carácter de **activo**. Están incluidos el Router y Broadcast. En ambos casos, un objeto le envía un mensaje al intermediario (router o broadcaster) para que éste se encargue de hacérselo llegar a quien corresponda.

3.5. Analizadores

En esta sección se detalla el proceso de construir la arquitectura a partir de la información brindada por los aspectos

Los analizadores son aquellos objetos que, a partir de cierta información que se obtenga del sistema en ejecución, construyen la arquitectura de dicho sistema. Como ya fue detallado, cuando el sistema se encuentra en ejecución, los aspectos van monitoreando al sistema para que al momento en que se realice una acción pertinente, ésta sea captada por el aspecto y enviada al correspondiente analizador.

La interfaz que implementan estos analizadores consiste en que a partir de un *joinpoint* y cierta información del contexto en que fue capturado, complementándolo con la información recibida anteriormente, pueda construir un nuevo *snapshot* de la arquitectura (es decir, la arquitectura que puede inferir a partir de todos los datos obtenidos hasta el momento). El *joinpoint* representa el estado en que se encontraba el sistema al momento de ejecutar cierta acción relevante. Por otro lado, el contexto en que fue capturado representa información que conoce el aspecto sobre el comportamiento del *joinpoint* y que puede facilitar el trabajo del analizador. En el contexto se puede incluir cosas como qué aspecto captó la acción o qué protocolo se estaba ejecutando.

En la siguiente sección se presentará un ejemplo, describiendo los diferentes procedimientos internos que se realizan en la herramienta para generar la arquitectura, incluyendo la etapa de los analizadores aquí descritos.

En base a la generalización mencionada en la sección anterior se decidió emplear tres analizadores: **Directo**, **Inactivo** y **Activo**. Los mismos se describen a continuación.

Analizador Directo

El *analizador directo* recibe joinpoints del estilo “*El objeto e de tipo E envió el mensaje M a un objeto r de tipo R*”. Como contexto, en general, recibe qué conector es el que se está utilizando, algo que el aspecto sabe perfectamente ya que cada uno modela a un único conector.

A partir de la información recibida debe inferir cómo puede acoplar estos nuevos datos a los que ya tenía. Por un lado, si la relación entre el emisor y el receptor no existía (siempre en el contexto de un único conector), entonces se debe agregar a la arquitectura la utilización del conector entre esos dos componentes. Por otro, si la relación sí existía, se tiene que analizar si esta nueva información afecta a la relación existente, ya que puede modificarse su aridad en caso de que los objetos que se comunican difieran de los existentes.

Algo interesante es que no importa cuál fue el mensaje que se envió, ya que es parte de la responsabilidad de los aspectos. A lo que respecta al analizador, lo relevante son quiénes se comunican y en qué contexto.

En pseudocódigo, el algoritmo sería:

```
1 def new_joinpoint(joinpoint, context)
2   connector = context.connector()
3   connector_relations = relations.get_relations_of(connector)
4
5   relation = if connector_relations.include?(joinpoint.sender_type,
6             joinpoint.target_type)
7             connector_relations.get(joinpoint.sender_type, joinpoint.target_type)
8             else
9               DirectRelation.new(joinpoint.sender_type, joinpoint.target_type)
10            end
11   relation.add_new_instance(joinpoint.sender, joinpoint.target)
12   connector_relations.add(new_relation)
13 end
```

Fragmento de Código 3.8: Algoritmo analizador Directo

Inicialmente, se obtiene del contexto qué conector se utilizó [línea 2]. Luego, se obtienen las relaciones del conector [línea 3] y se chequea si existe una relación entre emisor y receptor con ese conector [línea 5]. De existir la relación, se utilizará la existente [línea 6], sino se creará una nueva [línea 8]. Finalmente, a la relación antes mencionada se le agrega la nueva *aparición* que surge del *joinpoint* encontrado y ésta es guardada entre las relaciones del conector.

Los *joinpoints* siempre tendrán como *sender* al objeto emisor y como *target* al objeto receptor.

Analizador Activo

En el *analizador activo*, los *joinpoints* que recibirá serán de la forma “*El objeto e de tipo E le envió un mensaje M al intermediario I*” o “*El intermediario I le envió el mensaje M al objeto r de tipo R*”. Como contexto, recibirá tanto el conector al que “pertenece” el mensaje como en cuál de las dos situaciones anteriores se encontraba (esto el aspecto lo conoce ya que maneja ambas situaciones de manera diferenciada).

El análisis del *joinpoint* recibido bifurca dependiendo del tipo de mensaje, es decir, si es un mensaje hacia o desde el intermediario. El análisis pivotea alrededor del intermediario, por lo que hay que analizarlos de manera diferente porque en un caso es el receptor del mensaje mientras que en el otro es el emisor. Cada una de las relaciones va a tener como clave a una instancia del intermediario, por lo que cualquiera sea el *joinpoint* como primer paso deberá generar la relación para esa instancia o reutilizar una existente si ésta existiera. A partir de esto, se guardará por un lado los emisores y por otro los receptores.

En pseudocódigo el algoritmo sería:

```

1 def new_joinpoint(joinpoint, context)
2   connector = context.connector
3   joinpoint_type = context.protocol
4   connector_relations = relations.get_relations_of(connector)
5
6   if joinpoint_type == TRANSMITTER
7     relation = if connector_relations.include?(joinpoint.target)
8       connector_relations.get(joinpoint.target)
9     else
10      ActiveRelation.new(joinpoint.target)
11    end
12
13    relation.transmitters.add(joinpoint.sender)
14  else
15    relation = connector_relations.get(joinpoint.sender)
16
17    relation.receivers.add(joinpoint.target)
18  end
19 end

```

Fragmento de Código 3.9: Algoritmo analizador Activo

El intermediario separa a los *emisores* de los *receptores*. En el caso de los *emisores*, el *joinpoint* tendrá como *sender* al objeto emisor y como *target* al objeto intermediario. Por otro lado, en el caso de los *receptos*, el *joinpoint* tendrá como *sender* al objeto intermediario y como *target* al objeto receptor.

Del contexto se obtendrá el conector [línea 2] y protocolo [línea 3] utilizados en el *joinpoint*.

En caso de que el protocolo fuera de emisor hacia el intermediario, se obtendrá la relación existente [línea 8] o se creará una nueva [línea 10] y se guardará al *emisor* como parte de la relación [línea 13].

En caso contrario, se obtendrá la relación existente [línea 15] y se cargará al nuevo *receptor* [línea 17]. Esta relación se espera que exista ya que para que se ejecuten los mensajes de forwarding desde el intermediario debería haber existido un primer paso de emisión a dicho intermediario.

Analizador Inactivo

En el *analizador inactivo*, los *joinpoints* que recibirá serán siempre de la forma “*El objeto e de tipo E envió un mensaje M al intermediario I*”. Como contexto recibirá tanto a qué conector se refiere el mensaje como a qué protocolo del conector, ya que es necesario para entender cómo se conectan con el intermediario y generar la relación correctamente.

A diferencia del analizador activo, el análisis será el mismo para ambos lados de la relación (que llamaremos *input* y *output* por claridad) ya que en ambos la forma del mensaje es la misma. Similarmente al caso anterior, se pivoteará sobre la instancia del intermediario que será la clave para las relaciones (y por ello en cada *joinpoint* se utilizará la

relación existente, de existir, o se creará una nueva). Luego se guardarán para esa relación, por un lado, los objetos que emitan mensajes de input y, por otro, los objetos que emitan mensajes de output.

En pseudocódigo, el algoritmo sería:

```

1 def new_joinpoint(joinpoint, context)
2   connector = context.connector
3   joinpoint_type = context.protocol
4   connector_relations = relation.get_relations_of(connector)
5
6   relation = if connector_relations.include?(joinpoint.target)
7     connector_relations.get(joinpoint.target)
8   else
9     InactiveRelation.new(joinpoint.target)
10  end
11
12  if joinpoint_type == INPUT
13    relation.inputs.add(joinpoint.sender)
14  else
15    relation.output.add(joinpoint.sender)
16  end
17 end

```

Fragmento de Código 3.10: Algoritmo analizador Inactivo

En este caso, el *joinpoint* incluirá como *target* al objeto intermediario de la relación (que puede ser un *pipe*, *publish-subscribe bus*, etc) y en el *sender* al objeto que realiza la acción.

Del contexto se obtendrá el conector [línea 2] y protocolo [línea 3] utilizados. A continuación se creará una nueva relación [línea 9] o se usará una existente [línea 7] para el objeto intermediario que aparece en el *joinpoint*. Finalmente, dependiendo de si es *input* [línea 13] o *output* [línea 15] se guardará en la relación según corresponda.

Reconstrucción de Arquitectura

Como ya vimos, el sistema en ejecución genera *joinpoints* que son capturados por los aspectos quienes notifican a sus respectivos analizadores, los cuales procesan la información captada y comienzan a darle forma a la arquitectura. Cada analizador conoce toda la información de los conectores que representa, por lo que puede informar sobre las relaciones *bien formadas* que existen (que tienen, al menos, los dos extremos de la relación). Por esto, para construir la arquitectura (acción que se realiza luego de cada *joinpoint* capturado) se debe llamar a todos los analizadores y unir toda la información recibida que será lo que forme la arquitectura.

Para mostrar la información de un conector, el analizador correspondiente debe iterar sobre las relaciones que guarda distinguidas por ese conector. Generalizando, cada relación bien formada será de la forma “*input+* - *conector* - *output+*” (también pueden existir relaciones que no tengan una de estas partes, por ejemplo porque el router todavía

no envío el mensaje a sus destinatarios, las cuales serán ignoradas hasta que sean completadas). Salvando algunos casos, tanto *input* como *output* son conjuntos de objetos, y para determinar la relación arquitectónica final se debe establecer qué componentes son los que participan en la relación. Para realizar esto tiene que ver qué clases distintas hay en cada uno de estos conjuntos, que representarán los componentes, y para cada una de estas clases cuántas instancias distintas hay, que representará la *aridad* de cada uno de estos componentes. Esta relación arquitectónica se va completando y refinando a medida que se ejecuta el sistema.

3.6. Flow de la Herramienta

En esta sección se detallará el flow de la herramienta, explicando cómo todo lo mostrado en las secciones anteriores se junta para producir el output: la arquitectura.

3.6.1. Paso a Paso

A grandes rasgos, los pasos a seguir son:

1. El código está en ejecución.
2. Se ejecuta un joinpoint
3. Si no existe ningún pointcut que matchee al joinpoint, seguir ejecutando desde 1.
4. Se ejecuta un pointcut.
5. Se ejecuta un advice asociado al pointcut. Si el pointcut ya fue procesado, ir a 1.
6. Establecer pointcut como procesado.
7. El analizador correspondiente al aspecto al que pertenece el advice seleccionado, ejecuta el análisis sobre el joinpoint.
8. Se rearma toda la arquitectura.
9. Volver a 1 desde donde estaba ejecutando.

El análisis de la arquitectura se inicia cuando el sistema entra en ejecución. A partir de allí se van ejecutando diferentes *joinpoints* dependiendo de qué acciones toma el sistema. En algún momento se puede ejecutar algún *pointcut* que sea captado por un aspecto. A partir de allí se procederá a realizar el análisis sobre el nuevo *pointcut* detectado con el subsecuente rearmado de la arquitectura adoptando la nueva información. Luego el sistema vuelve a su estado normal de ejecución, generando *joinpoints* que eventualmente son captados por algún aspecto.

A continuación se muestra un ejemplo que detalla cómo se realiza internamente el proceso descrito anteriormente.

3.6.2. Ejemplo

Asumamos que estamos inspeccionando un sistema cuya arquitectura utiliza el conector *pipe*. Por ejemplo, tiene una clase *EmailPipe* con sus métodos *#pushNewEmail* y *#popNextEmail* anotados de la siguiente forma:

```

1 class EmailsPipe {
2     @PipePush
3     public void pushNewEmail(Email email){
4         emails.add(email);
5     }
6     @PipePop
7     public Email popNextEmail(){
8         emails.getFirst();
9     }
10 }

```

Fragmento de Código 3.11: Detalle clase *EmailPipe*

Eventualmente, se envía el mensaje *#pushNewEmail*

```

1 class EmailCreationGUI {
2     public void newEmail(Email emailReceived){
3         emailsContainer.pushNewEmail(emailReceived);
4     }
5 }

```

Fragmento de Código 3.12: Envío mensaje *#pushNewEmail*

El aspecto correspondiente al *pipe* detectará que el *joinpoint* se corresponde con uno de sus *pointcuts*[línea 3] y ejecutará el advice correspondiente. Éste, a su vez, revisará si el *joinpoint* fue procesado[línea 12] y como no lo estaba, le enviará un mensaje al analizador para que realice el análisis sobre el nuevo *joinpoint*[línea 14].

```

1 aspect PipeAspect {
2     pointcut pop(): call(@PipePop * * (..))
3     pointcut push(): call(@PipePush * * (..))
4
5     before(): pop(){
6         if(processed(thisJoinPoint)) return;
7         setProcessed(thisJoinPoint);
8         analyzer.newJoinpoint(thisJoinPoint, "pop");
9     }
10    before(): push(){
11        if(processed(thisJoinPoint)) return;
12        setProcessed(thisJoinPoint);
13        analyzer.newJoinpoint(thisJoinPoint, "push");
14    }
15 }

```

Fragmento de Código 3.13: Aspecto del conector *Pipe*

El analizador al recibir al mensaje procederá a realizar el análisis. Como no existe la relación para ese *pipe*, la creará con el *pusher* correspondiente. Una vez procesado el nuevo *joinpoint*, procederá a construir la arquitectura nuevamente, la cual no contendrá a la relación del *pipe* ya que necesita la segunda parte para existir (alguien que consuma de

él).

Luego de que el *advice* termina de ejecutar, el aspecto correspondiente al *call sincrónico* también detectará el *joinpoint* [línea 2] ya que es un mensaje enviado, pero como ya fue procesado lo ignorará [línea 5].

```

1 aspect CallSyncAspect {
2     pointcut call(): call( * * (..))
3
4     before(): call(){
5         if(processed(thisJoinPoint)) return;
6         setProcessed(thisJoinPoint);
7         analyzer.newJoinpoint(thisJoinPoint);
8     }
9 }

```

Fragmento de Código 3.14: Aspecto del conector Call Sincrónico

Luego, en algún momento, se envía el mensaje *#popNextEmail*.

```

1 class EmailProcessor {
2     public void processEmail(){
3         Email emailToProcess = emailsToProcess.popNextEmail();
4         // ...
5     }
6 }

```

Fragmento de Código 3.15: Envío mensaje #popNextEmail

Nuevamente, desencadena que sea detectado por el aspecto del *pipe* [línea 2] y que sea enviado el *joinpoint* al analizador [línea 9]. Esta vez el analizador ya tiene la relación creada con ese *pipe* y sólo debe agregar al emisor del mensaje entre los consumidores. Esta vez, al crear la arquitectura, en ésta sí aparecerá la relación del *pipe* ya que existen *pushers* y *poppers*. El *output* de la herramienta contendrá:

EmailCreationGUI – Pipe – EmailProcessor

mostrando la relación entre *EmailCreationGUI* y *EmailProcessor* a través de un conector *pipe*.

3.7. Observaciones

En esta sección vamos a explicar observaciones, problemas y soluciones que fueron surgiendo del desarrollo.

3.7.1. Precedencia e Interferencia

Uno de los problemas que surgió al utilizar implementaciones de arquitecturas que usaban distintos tipos de conectores fue el de la *precedencia*. Es decir, cómo hacer para que al enviar un mensaje que es parte de cierto conector, éste sea captado por el aspecto correspondiente. Este problema surge porque los aspectos, y más precisamente los *pointcuts* en ellos definidos, tienen *interferencias* entre sí ya que no son disjuntos. En particular, esta situación tiene lugar cuando dos o más aspectos intervienen en el mismo lugar. La comunidad de aspectos define este problema como “Interferencia de Aspectos” [20]. Si bien existen algunas aproximaciones para resolver este problema [21, 22] *AspectJ*, el framework utilizado para aplicar la programación orientada a aspectos en la herramienta, no tiene una forma de atacarlo directamente, por lo que se debe recurrir a soluciones ad-hoc. Una es hacer que los aspectos sean efectivamente disjuntos, aunque este enfoque trae problemas como acoplamiento entre todos los aspectos además de ser poco escalable. Finalmente, para la solución elegida se decidió definir precedencia entre aspectos.

En primer lugar, se define la precedencia entre los diferentes aspectos mediante una directiva de *AspectJ* que permite decidir ante un *joinpoint* que puede ser captado por más de un aspecto, cuál de estos ejecutará primero. Tal directiva es *declare precedence*. El orden de precedencia estará relacionado a cuán específicos son los aspectos. Los *conectores inactivos* son los más específicos porque su protocolo se mapea directamente a métodos anotados, luego los *conectores activos* que tienen su protocolo parcialmente mapeado directamente a métodos anotados y finalizando con los *conectores directos*, que salvando el caso del *Client-Server* no se mapean a métodos.

Finalmente, la precedencia se establece como se especifica a continuación. Esencialmente, los aspectos más específicos deben tener mayor precedencia (es decir, “correrán” primero) para que sea correctamente relevada la arquitectura. En concreto, el orden establecido de precedencia es el ilustrado en el siguiente fragmento de código:

```

1 declare precedence:
2   PipeAspect, PublishSubscribeAspect, BlackboardAspect,
3     ClientServerAspect,
4   BroadcastAspect, RouterAspect,
5   CallAsyncAspect, CallSyncAspect;
```

Fragmento de Código 3.16: Definición de precedencia de aspectos

Una vez que se tiene definida la precedencia y se sabe que los *pointcuts* se van a ejecutar en el orden esperado, resta definir cómo hacer para que el análisis se haga en sólo uno de estos aspectos. Aquí surge la limitación del framework, ya que no existe una forma

directa de decidir, al ejecutar un *pointcut*, que ya no se sigan ejecutando los subsecuentes [23]. La solución elegida es mantener una colección con todos los *joinpoints* que ya fueron procesados. Al ejecutarse un *pointcut*, se verifica si el *joinpoint* asociado ya fue procesado (viendo la colección) y en caso de que no lo haya sido, se lo procesa y se lo agrega a la colección para que subsecuentes aspectos no lo procesen. Nuevamente, la limitación del framework lleva a recurrir a soluciones ad-hoc como la descrita anteriormente.

3.7.2. Concurrencia

Otro de los problemas que surgió estuvo relacionado con la *concurrencia*. Esto se debe a que nuestros sistemas utilizan *threads*, por ejemplo al enviar mensajes asíncronos, y por lo tanto puede ocurrir que se procesen varios *joinpoints* al mismo tiempo generando problemas de concurrencia en los analizadores. Existen diferentes posibles soluciones para atacar el problema: que los analizadores puedan procesar concurrentemente, que se pueden ejecutar concurrentemente sólo los *pointcuts* que tratan conectores distintos, entre otros. La solución para el problema será realizar una exclusión mutua entre todos los *pointcuts*. Se prefirió adoptar una solución simple para poder poner todo el foco en la validación de la arquitectura, que es el principal objetivo del presente trabajo.

3.7.3. Alias

Otro de los problemas surgió al intentar implementar el conector *client-server*. Este conector se caracteriza porque un cliente establece una conexión con un servidor para luego comunicarse con éste a través de dicha conexión. El problema que surge de esta característica es que los clientes se comunican con la conexión pero esto es algo meramente implementativo, en la arquitectura el cliente se comunica siempre con el servidor.

La solución al problema es permitir que se pueda establecer un *aliasing* entre el servidor y la conexión. De esa manera, cuando se comuniquen con la conexión será como si hablaran directamente con el servidor. Implementativamente, lo que se hace es, al crear la conexión, guardarla como un *alias* al propio servidor. Posteriormente, cuando la conexión recibe un mensaje, se verifica si ésta tiene un *alias* para utilizar a éste en lugar de la conexión.

Esta característica se implementa únicamente en el conector *Client-Server* pero podría ser utilizada por cualquier otro conector que lo necesite ya que la implementación de tratar a un objeto como su *alias* es general.

3.7.4. Extra: Análisis Alternativos

Adicionalmente del estudio de los conectores arquitectura, se realizaron algunos análisis alternativos que permitieran obtener información adicional de la arquitectura. Lo que se busca es brindarle al usuario más herramientas para entender el funcionamiento del sistema en el contexto de la arquitectura y sus conectores. Estos análisis se realizan a partir de

información que ya era captada por los diferentes aspectos de la herramienta aunque sin ser utilizada. Los análisis alternativos son: **Notificaciones de Publish-Subscribe y detección del estilo Pipe & Filter.**

Notificaciones de Publish-Subscribe

En primer lugar, se implementó la característica de mostrar notificaciones para el conector *publish-subscribe*. Puntualmente, la idea es listar qué suscriptores de un *publish-subscribe* reciben qué publicaciones. Esto permitiría analizar el flujo de la información en este conector, detectando posibles errores por suscriptores que reciben información que no deberían recibir o que no reciben información que deberían estar recibiendo.

La implementación de esta característica se realiza aparte del análisis arquitectónico ya explicado anteriormente. Las notificaciones se obtienen del *pointcut* que detecta mensajes que se envían al generar la publicación. De allí se obtiene quiénes son los suscriptores que están recibiendo la notificación y cuál es el evento que generó dicha notificación.

Estilo Pipe & Filter

Por otro lado, se implementó la característica de detección de implementación de estilo *pipe & filter*. Como ya fue explicado, este estilo es una secuencia de componentes (llamados *filtros*) divididos por *pipes* y se espera que estos componentes consuman de un *pipe* e inserten en el siguiente formando un *pipeline*. Adicionalmente, se muestran advertencias cuando hay componentes que no realizan el flujo esperado.

Al igual que en el caso anterior, esta característica se realiza separada del análisis arquitectónico de los conectores. La información que se utiliza es sobre los *pointcuts* referidos a los *push* y *pop* sobre los *pipes*. Para implementarlo, a partir de los *joinpoints* recibidos se va formando el *pipeline* (conjunto de filtros seguido de un pipe seguido de un conjunto de filtros y así sucesivamente) y aquellos casos que rompan dicho *pipeline* serán mostrados como advertencia: por ejemplo puede ocurrir que un componente consuma de un *pipe* e inserte en otro que no sea el siguiente en el *pipeline*, o que inserte en un *pipe* sin haber consumido del anterior.

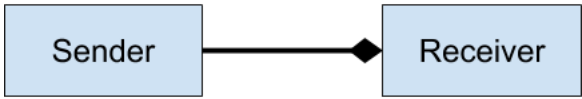
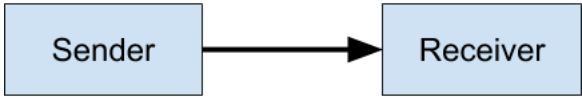
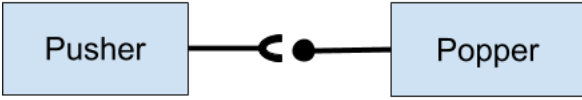

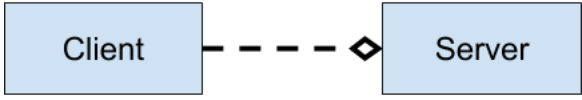
Una vez generado el *pipeline*, siempre y cuando éste exista, se muestran ordenados los conjuntos de filtros que participan de él.

4. CASO DE ESTUDIO

En esta sección se aprovecharán todos los conocimientos aprendidos a lo largo del trabajo para estudiar la herramienta desde la visión de un usuario, mostrando las diferentes facetas que puede encontrar para aprovecharla en el proceso de validar los conectores en la implementación de una arquitectura.

Teniendo en claro el funcionamiento y las motivaciones de las diferentes partes que conforman el presente trabajo, en esta sección se describirán, utilizando un sistema suficientemente complejo como caso de estudio, los diferentes casos de uso de la herramienta. En primer lugar se analiza el proceso de anotación de un sistema ya construido para poder reflejar en la arquitectura lo implementado. Luego, se describirán los distintos tipos de análisis que se pueden realizar una vez que se tiene el sistema anotado.

En esta sección, el *output* de la herramienta se mostrará con una interpretación visual de los componentes, conectores (cuyas notaciones se muestran en la tabla 4.1) y las relaciones. El output real es un listado con las relaciones y las aridades, y se puede encontrar en el apéndice.

Conector	Notación
Call Sincrónico	
Call Asincrónico	
Pipe	
Publish-Subscribe	
Client-Server	

Tab. 4.1: Notaciones de los diferentes tipos de conectores a utilizar en esta sección

4.1. Sistema de ejemplo

Para realizar el caso de estudio utilizaremos como ejemplo un sistema *para el monitoreo y cuidado de una planta*. Este sistema es una extensión de un trabajo práctico de la materia *Ingeniería de Software II [DC-FCEN-UBA]*. Se utiliza este sistema ya que se lo considera suficientemente interesante al presentar una variedad de conectores como *publish-subscribe*, *client-server*, *pipe*, *call sincrónico* y *asincrónico*.

En pocas palabras, el ejemplo consiste en un sistema que lleve el control del crecimiento de una planta. Obtiene información de sensores (agua, luz, humedad), y luego a partir de estos datos deduce instrucciones para el cuidado de la planta (más riego, menos luz, etc.) a partir de conocimientos de botánica. Finalmente, estas instrucciones son llevadas a cabo, y el proceso vuelve a iniciarse.

El principal componente del sistema es el *Botanical Expert*, que es el *cerebro* del sistema. Él se encarga de recibir las órdenes de los usuarios, la información del clima, controlar el crecimiento de la planta, obtener los valores sensados de su estado y comunicarse con el jardinero para realizar aplicaciones de agua o calor.

El *App Interface Server* es la interfaz del sistema que separa el comportamiento interno de los usuarios, los cuales se comunican a través de la *Graphic User Interface*.

El *Clima Server* es un servidor que informa, a través de un *publish-subscribe bus*, los datos del clima tanto diaria como semanalmente.

El *Jardinero* será quien reciba órdenes del *Botanical Expert* para realizarle aplicaciones a la planta, y es quien conoce cómo comunicarse con los *aplicadores* para que realicen su trabajo.

La arquitectura del sistema utilizado se puede observar en la figura 4.1.

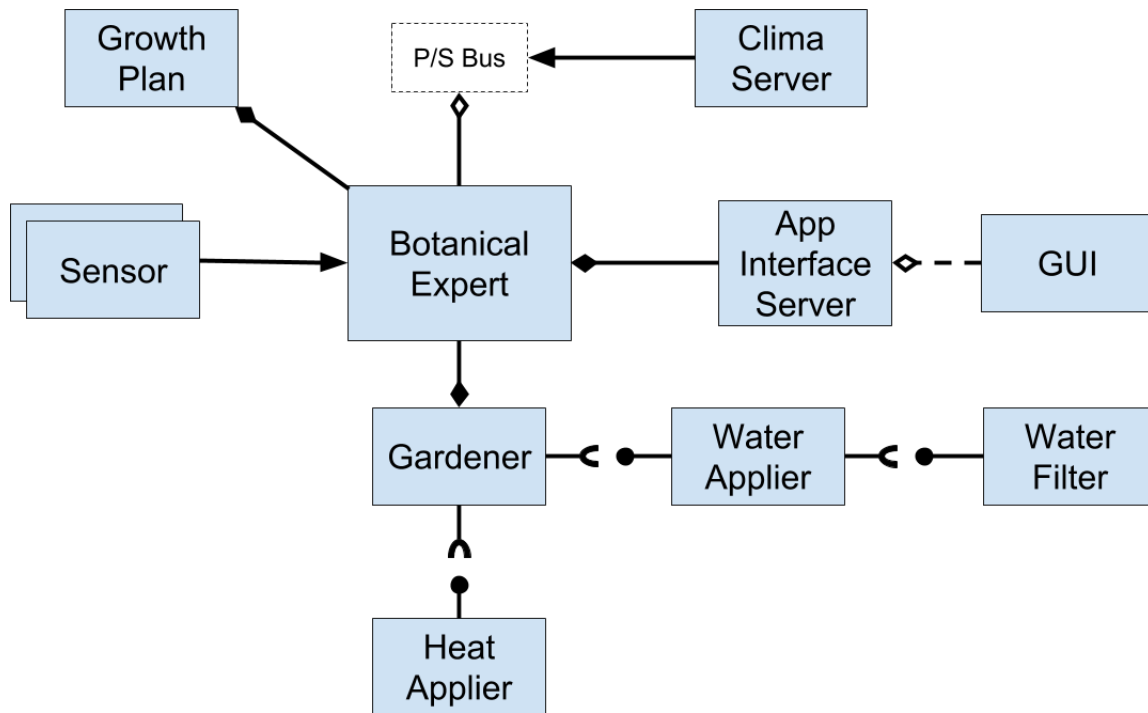


Fig. 4.1: Arquitectura del sistema a utilizar como ejemplo para el caso de uso

El comportamiento principal del sistema puede describirse como sigue. Por un lado, el flow es que el *graphic user interface* se comunica con el *app interface server* que actuará como la interfaz que separa el sistema del usuario. Éste se comunicará con el *botanical expert*. Si es para agregar agua o calor a la planta, el *botanical expert* se comunicará con el *gardener* que será quien lo lleve a cabo al comunicarse con el *heat applier* o con el *water filter* pasando por el *water applier*. También, el *botanical expert* se puede comunicar con el *growth plan* para seguir el estado de la planta.

Por otro, el *botanical expert* recibe periódicamente información de los *sensores* con información del sensado de la planta y del *clima-server* con información del clima, según a qué esté suscrito.

4.2. Proceso de anotación

La herramienta necesita que el usuario le indique dónde se encuentran determinadas estructuras que representan a los conectores para poder construir la arquitectura a partir de la ejecución de la implementación. La forma de indicar eso es a través de anotaciones, donde se anotan métodos y clases para referenciar los diferentes tipos de comunicaciones. En la tabla 4.2 se pueden observar los diferentes tipos de anotaciones que existen. El problema que surge es que las diferentes estructuras mencionadas pueden no ser fácilmente identificables en el código, lo que dificultaría el proceso de anotación. Por esto, se propone una forma de anotar iterativamente a partir de comparar la arquitectura generada con la arquitectura de referencia.

Conector	Anotaciones	Método	Clase
Call Sincrónico	<i>No necesita anotaciones</i>		
Call Asincrónico	<i>No necesita anotaciones</i>		
Pipe	<i>@PipePush</i>	X	
	<i>@PipePop</i>	X	
Publish-Subscribe	<i>@Publish</i>	X	
	<i>@Subscribe</i>	X	
	<i>@PublishNotification</i>	X	
Client-Server	<i>@RequestConnection</i>	X	
	<i>@ClientServerConnection</i>		X
Router	<i>@Route</i>	X	
Broadcast	<i>@Broadcast</i>	X	
Blackboard	<i>@Store</i>	X	
	<i>@SubscribeBlackboard</i>	X	
	<i>@StoreNotification</i>	X	
	<i>@Read</i>	X	
Ignored	<i>@Ignored</i>	X	X

Tab. 4.2: Anotaciones para las interacciones de los conectores

El algoritmo que se utilizará será: ejecutar la herramienta y analizar el *output* con la arquitectura de referencia. De existir diferencias, analizar los resultados para ver la posibilidad de que hayan faltado anotar estructuras en el código. Con las nuevas anotaciones se corre una vez más la herramienta y se repite el ciclo hasta quedar satisfecho con el resultado. Una vez anotado por completo el código, se deberán analizar las diferencias entre la arquitectura relevada y la especificada en la vista de conectores. Este proceso permite una aproximación iterativa e incremental para validar el comportamiento arquitectónico.

Como describe el algoritmo, el primer paso es ejecutar la herramienta y analizar el *output*. Luego de correr la herramienta con el sistema sin anotaciones, el resultado será el mostrado en la figura 4.2

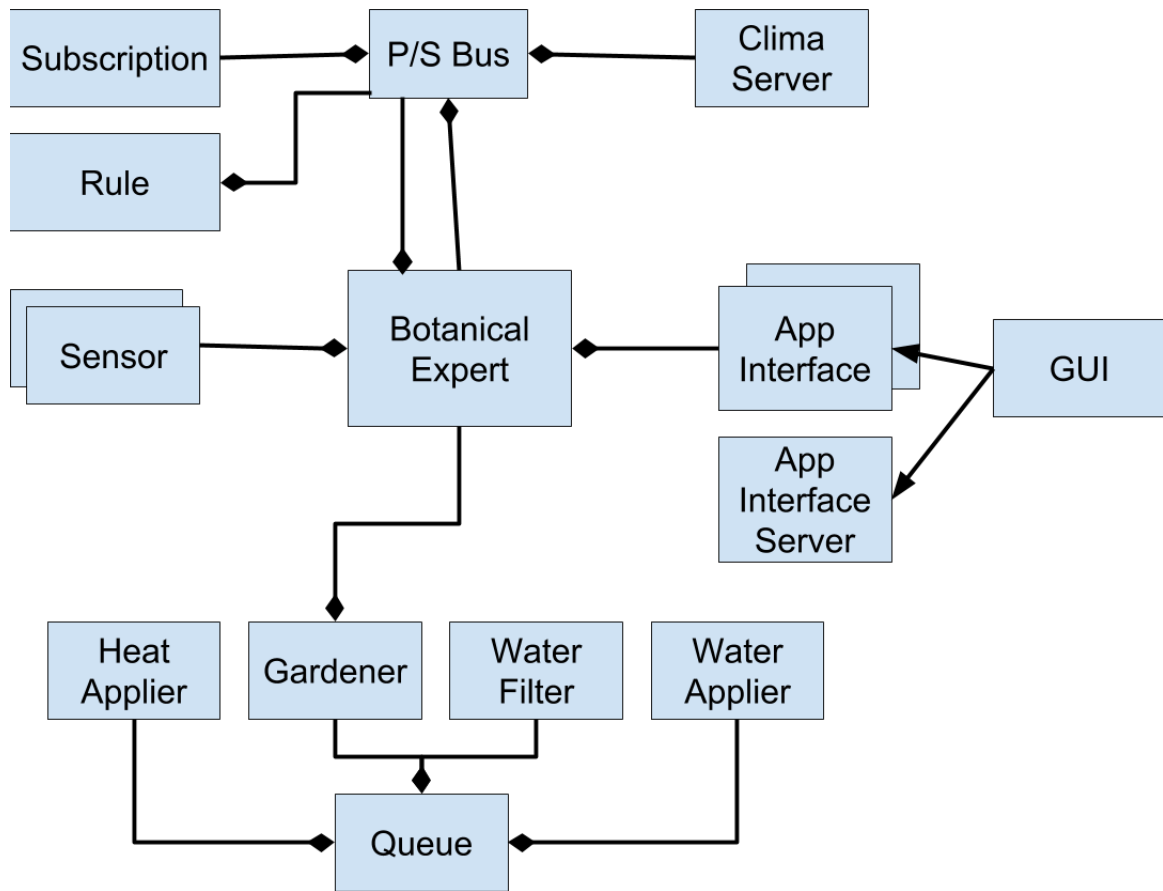
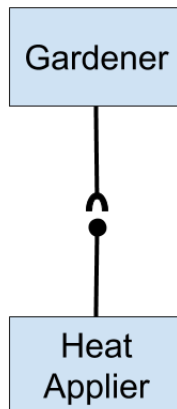


Fig. 4.2: Arquitectura generada a partir del sistema sin anotar

Como se puede observar, el resultado se diferencia claramente de la arquitectura de referencia, y el motivo es que no se indicó dónde se encuentran las diferentes estructuras que forman a los conectores. Por esto los únicos conectores que se utilizan son los directos, tanto *call sincrónico* como *call asincrónico*, ya que éstos no necesitan ser anotados.

El siguiente paso es comparar el *output* con la arquitectura de referencia e intentar ir detectando ciertas partes de la arquitectura para anotar los conectores que correspondan. Por ejemplo, como se muestra en la figura 4.3, se tiene la relación *Gardener - HeatApplier* utilizando un *pipe* pero en el *output* se muestra como una relación entre *Gardener - Queue* y otra entre *Heat Applier - Queue*. Es importante entender, entonces, que *Queue* es quien permite que la relación *Gardener - HeatApplier* exista utilizando un *pipe*. Luego, hay que anotar la clase que representa al *pipe* (que en este caso sería *Queue*) para que la herramienta entienda cuál es su rol en el sistema. Para anotarla, se deben detectar los métodos que representan el protocolo del conector que como se puede ver en la tabla 4.2 son: *push* [línea 6] y *pop* [línea 10]. El código anotado quedaría como se muestra en el fragmento de código 4.1

Arquitectura de Referencia



Output de la Herramienta

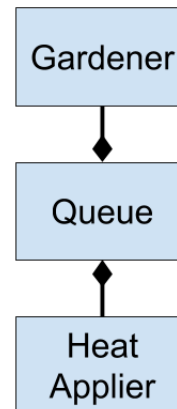


Fig. 4.3: Comparación del conector pipe

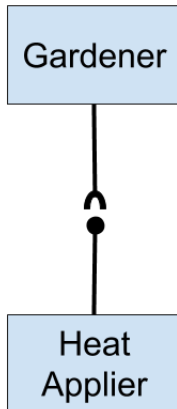
```

1 public class Queue {
2     private LinkedList<String> content;
3     public Queue(){
4         content = new LinkedList<String>();
5     }
6     @PipePush
7     public void push(String dato) {
8         content.push(dato);
9     }
10    @PipePop
11    public String pop(){
12        return content.pop();
13    }
14    public boolean hasElements(){
15        return !content.isEmpty();
16    }
17 }
  
```

Fragmento de Código 4.1: Implementación de Queue

Al volver a ejecutar la herramienta, se puede observar que la utilización del conector *pipe* se muestra donde corresponde, pero aún tenemos la relación *HeatApplier - Queue* con un *call sincrónico*, como se muestra en la figura 4.4. Analizando el código de la clase *Queue* se puede ver que existe el método *hasElements()*[línea 14] como se muestra en el fragmento de código 4.1. Este método no es parte del protocolo del conector pero como se lo utiliza como parte del mismo, se debe indicar esta situación. Para indicar esto se debe agregar la anotación *@Ignored*, cuya función es indicarle a la herramienta que el método es auxiliar y no se espera que se muestre en la arquitectura. Con este cambio se puede observar la correcta utilización del conector *pipe*.

Arquitectura de Referencia



Output de la Herramienta

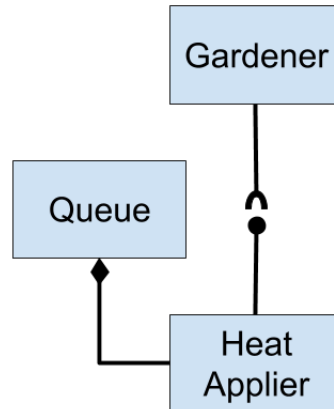
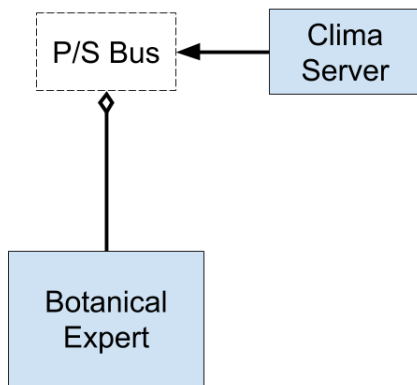


Fig. 4.4: Comparación del conector pipe sin estar completamente anotado

La siguiente iteración es intentar corregir la utilización del *publish-subscribe*. Como se muestra en la figura 4.5, y similar al caso del *pipe* antes visto, al no indicar en el código la presencia del conector (a través de las anotaciones correspondientes) la herramienta no puede detectarlo.

Arquitectura de Referencia



Output de la Herramienta

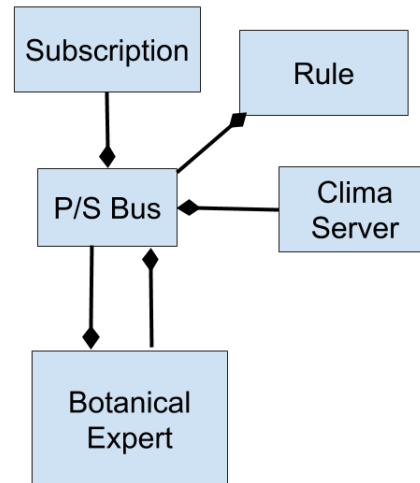


Fig. 4.5: Comparación del conector publish-subscribe

Viendo las relaciones que se formaron, resulta simple anotar los métodos que corresponden teniendo en cuenta los diferentes tipos de anotaciones para este conector, como se muestran en la tabla 4.2. Por un lado, el *Botanical Expert* se comunica con el *Publish-Subscribe Bus* para suscribirse y se anota `@Subscribe`. También, el *Clima Server* publica en el *Publish-Subscribe Bus* y se anota `@Publish`. Por último, el *Publish-Subscribe Bus* notifica a los suscriptores y se anota como `@PublishNotification`. La relación con *Rule* es “interna” del procedimiento que se realiza al recibir una publicación por lo que no necesita

ser anotada. Con estos cambios, como se muestra en el fragmento de código 4.2, podemos representar correctamente la utilización del conector *Publish-Subscribe*.

```

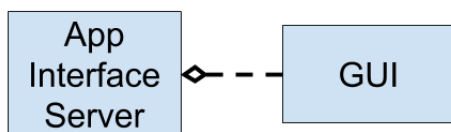
1 public class PublishSubscribeBus {
2     // ...
3     @Publish
4     public void publish(String publication) {
5         for(Subscription subscription: this.subscribers){
6             if(subscription.rule().match(publication))
7                 subscription.subscriber().notifyPublication(publication);
8         }
9     }
10    @Subscribe
11    public void subscribe(Subscriber subscriber, Rule r) {
12        this.subscribers.add(new Subscription(subscriber, r));
13    }
14 }
15 public interface Subscriber {
16     public void subscribe();
17
18     @PublishNotification
19     public void notifyPublication(String publication);
20 }

```

Fragmento de Código 4.2: Implementación del conector Publish-Subscribe

Finalmente, la última iteración del proceso de anotación será corregir la utilización del conector *Client-Server* que se utiliza para relacionar al *Graphical User Interface* con el *App Interface Server*. En la figura 4.6 se puede observar la diferencia entre lo que se obtiene sin anotar y lo que se espera como resultado.

Arquitectura de Referencia



Output de la Herramienta

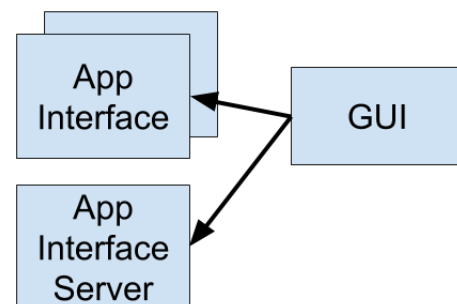


Fig. 4.6: Comparación del conector Client-Server

Lo que se espera conseguir es que se conecte al *Graphical User Interface* con el *App Interface Server* mediante el conector *Client-Server*. Sin anotaciones, se obtiene que el *GUI* se comunica primero con el *App Interface Server* para establecer la conexión y luego se comunica con el *App Interface* (cada vez que se quiere comunicar recibe una conexión nueva, por ello la aridad) para realizarse los pedidos que sean necesarios. Luego, teniendo

en cuenta las anotaciones de este conector que se pueden ver en la tabla 4.2, se debe anotar el establecimiento de la conexión, anotando como *@RequestConnection*; y a la conexión en sí (que sería el App Interface) como *@ClientServerConnection*. El resultado de estas anotaciones se muestra en el fragmento de código 4.3.

```

1 public class ApplicationInterfaceServer {
2     // ...
3     @RequestConnection
4     public ApplicationInterface requestToken() {
5         return new ApplicationInterface(expert);
6     }
7 }
8 @ClientServerConnection
9 public class ApplicationInterface {
10    // ...
11 }

```

Fragmento de Código 4.3: Anotaciones en el conector Client-Server

Una vez que se llega al punto en que se han anotado los diferentes conectores en el código, la herramienta puede reconstruir la arquitectura según fue implementada y a partir de las diferentes partes del código que se ejecuten, como se muestra en la figura 4.7.

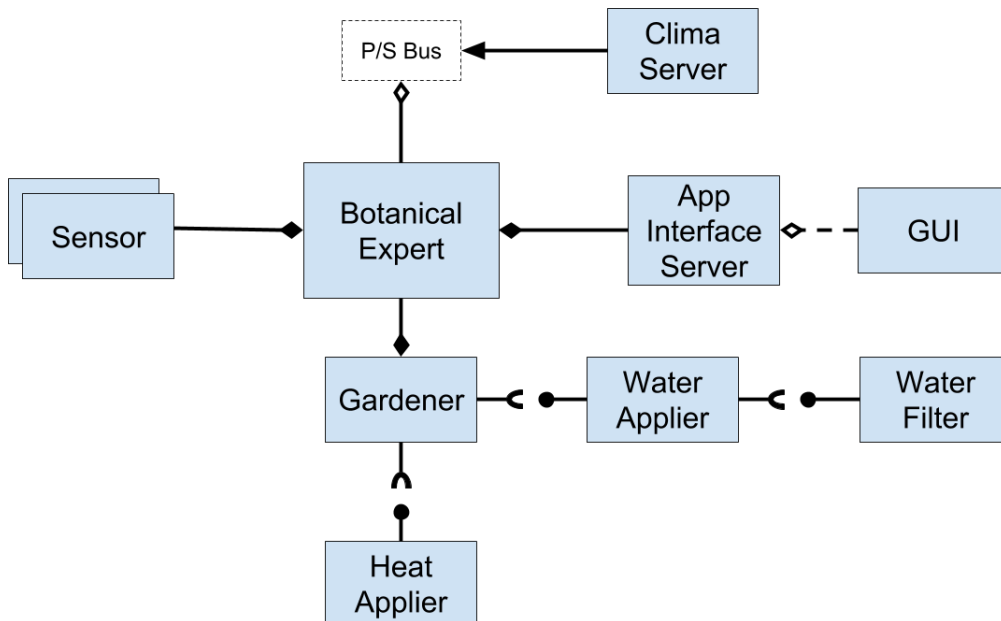


Fig. 4.7: Arquitectura generada luego del proceso de anotación

A continuación se detallan diferentes tipos de análisis y aprendizajes que se pueden realizar a partir del *output* de la herramienta una vez que se tiene el código anotado.

4.3. Análisis sobre el sistema anotado

El problema inicial era que la herramienta “entienda” la utilización de los conectores en la implementación. Una vez que el sistema está anotado, resta analizar la información que se obtiene de ejecutar la herramienta para detectar diferencias entre la implementación y la arquitectura, detectar problemas en la implementación de ciertos conectores (en particular *publish-subscribe* y *pipe*, que son conectores para los que se realizan ciertos análisis alternativos), detectar componentes “inalcanzables”, entre otras cosas.

4.3.1. Componentes inalcanzables

Si se compara la arquitectura de referencia, que se ve en la figura 4.1, con el resultado del proceso de anotación, que se ve en la figura 4.7, se pueden detectar ciertas diferencias. Una de ellas en particular es que el componente *Growth Plan* no se encuentra en el *output* de la herramienta. Esto se puede deber a varias cuestiones, donde cada una merece un análisis distinto.

En caso de que dicho componente haya sido anotado como *ignorado*, entonces el usuario deberá analizar si dicha anotación fue un error o en efecto, no se quiere que se muestre en la arquitectura. En dicho caso la especificación debería ser corregida para no tener en cuenta al componente.

En cambio, si no se han realizado las ejecuciones necesarias que permitieran que se genere algún procedimiento con dicho componente para que se muestre en la arquitectura, se debe analizar si existen dichas ejecuciones. De existir, se puede volver a ejecutar la herramienta realizando dichos procedimientos para que el *output* sea el esperado. De no existir, es un caso en que el componente nunca puede ser alcanzado y muestra un posible defecto en la arquitectura/implementación, ya sea porque no debería estar incluido en la arquitectura ya que no tiene “contacto” con el resto del sistema, o que existen errores implementativos que ocasionan el aislamiento.

4.3.2. Conectores mal utilizados

Otro de los análisis que se puede realizar es el de encontrar diferencias en la utilización de conectores entre la arquitectura y la implementación. Esto se puede deber a errores en la anotación de los conectores, como ya fue mostrado en la sección anterior, o por inconsistencias entre la arquitectura y la implementación.

La importancia de este análisis radica, entre otras cosas, en una de las características que tienen las implementaciones de arquitecturas que es la inconsistencia que se genera con el correr del tiempo y de las personas que van manipulándolas, ya sea porque se modifica la arquitectura pero no se actualiza consistentemente la implementación o viceversa.

Retomando el caso de estudio, al comparar la arquitectura de referencia, que se puede

ver en la figura 4.1, y la arquitectura generada por la implementación, que se puede ver en la figura 4.7, se puede ver que existe una diferencia en qué conector se utiliza para relacionar al componente *Sensor* con el *Botanical Expert*. En la arquitectura de referencia estaba definido para utilizar un *call asincrónico* pero en la implementación se utiliza un *call sincrónico*. Si se analiza el código, como se muestra en el fragmento de código 4.4, se ve que en los mensajes que envían los *sensores* al *botanical expert* se recibe una respuesta y que no son asincrónicos.

```
1 public void sense() {
2     // ...
3     boolean status = expert.newReading(sensedAmount);
4     if(status)
5         log_reading_received();
6     else
7         log_reading_not_received();
8     // ...
9 }
```

Fragmento de Código 4.4: Implementación del método *sense*

Una posibilidad es que la arquitectura haya quedado desactualizada o que fuera incorrecta ya que los *sensores* necesitan que la comunicación sea sincrónica, en cuyo caso se debería actualizar la arquitectura. La otra opción es que la implementación no haya respetado los lineamientos de la arquitectura al comunicarse sincrónicamente, por lo que se debería modificar la implementación para que respete a la especificación.

4.3.3. Extra: Notificaciones publish-subscribe

También se puede realizar otro tipo de análisis para el conector *publish-subscribe*, que trata de mostrar qué notificaciones se envían a los suscriptores. La utilidad es entender qué información están recibiendo los suscriptores y así poder ver el flujo de la información y detectar posibles errores.

En el ejemplo del caso de estudio en particular, se puede ver que la información recibida por el *Botanical Expert* es sobre el clima del día mientras que el *Clima Server* publica eventos de clima diario y semanales. De esta manera, se puede entender que la regla de suscripción establecida por el *botanical expert* funciona correctamente si espera recibir únicamente la información diaria.

4.3.4. Extra: Estilo pipe & filter

Por último, se puede apreciar que el sistema tiene un *pipe & filter*. En particular, esto se da en el flujo de aplicar agua desde el *gardener* hasta el *water filter* pasando por el *water applier*.

La ventaja de esta característica es que permite entender el flujo de datos que se

produce en esa parte del sistema. En caso de que existieran errores en la implementación del *pipe & filter*, también se mostrarían y el usuario podría detectarlos.

5. CONCLUSIONES Y TRABAJO FUTURO

En esta sección se resumirá el camino recorrido para realizar la presente tesis y se analizarán los posibles pasos a seguir.

En esta tesis se atacó el problema de validar arquitecturas utilizando la programación orientada a aspectos monitoreando dinámicamente la ejecución del sistema. Es decir, analizando los sistemas a través de su comportamiento dinámico y no estático. Se analizaron las características y comportamientos de los conectores que se eligieron para formar una amplia variedad de arquitecturas. Luego, se implementó una herramienta que fuera capaz de monitorear el programa en ejecución y a partir de cómo se comunican los diferentes procesos inferir la utilización de los conectores anteriormente mencionados. Para lograr esto se desarrolló el proceso de monitoreo, qué datos se recopilan y cómo, cómo se procesan y por último cómo se infiere la arquitectura a partir de todo lo anterior. Finalmente, se realizó un caso de estudio que permitiera analizar el trabajo realizado desde el punto de vista de un usuario que quisiera hacer uso de la herramienta, detallando los distintos tipos de análisis, problemas y soluciones que le pudieran surgir. Todo esto con el fin de que pueda realizar una correcta validación del uso de los conectores de la arquitectura en su sistema.

Si bien las contribuciones del presente trabajo son relevantes, existen también ciertas limitaciones. Por un lado, el proceso de anotar el código no es proceso trivial ya que implica tener un amplio conocimiento de su funcionamiento, y en sistemas grandes puede ser una amenaza. Sin embargo, existen dos aspectos que pueden aliviar este problema. Por un lado, el hecho de anotar el código es un procedimiento cada vez más presente en el desarrollo de software [24, 25, 26]. Por otro lado, la herramienta permite una aproximación iterativa con la posibilidad de ir aprendiendo del output recibido y mejorar las anotaciones realizadas. También permite concentrarse en porciones específicas, ignorando aspectos que no son relevantes. Por ejemplo, se puede utilizar para validar solamente el comportamiento de un sector de toda la arquitectura que sea de interés. Por otro lado, el trabajo se concentró únicamente en conectores. Si bien los mismos son una parte crucial de la vista de componentes y conectores, sería deseable incorporar a la relevación arquitectónica construcciones más complejas como estilos y patrones arquitectónicos [2]. Un puntapié en este sentido está dado por la sugerencia que brinda la herramienta sobre la posible detección del estilo Pipe and Filter, y el flujo de la información en el conector *Publish-Subscribe*. Esto da buenas perspectivas para un relevamiento más complejo en extensiones del presente trabajo.

A pesar del fructífero trabajo realizado, quedan muchas cuestiones en las cuales se podría continuar el presente trabajo:

- Mejora de la herramienta como producto, brindando un *output* más visual (generando un grafo con los componentes y las relaciones, con las diferentes anotaciones que

usan los conectores), integrando con IDEs para facilitar el uso de la herramienta, etc.

- Atacar el proceso de anotación del sistema desde otros puntos de vista. Por ejemplo, permitiendo dicha generación automáticamente o semi automáticamente con la guía del usuario.
- Analizar la posibilidad de realizar una integración con herramientas arquitectónicas existentes, como por ejemplo *ArchJava* [9] o *DiscoTect* [3].
- Agregar más análisis alternativos, como la detección del *Pipe & Filter* o la muestra de notificaciones en el *publish-subscribe*, que puedan realizarse con la información existente que se obtiene del sistema en ejecución.
- Extensión de las características presentadas para poder relevar comportamiento arquitectónico más complejo como estilos y patrones.

6. APÉNDICE

6.1. Código Fuente

El código fuente del presente trabajo está realizado utilizando el lenguaje *Java*. Para poder correr el código se necesita instalar *AspectJ*, se recomienda utilizar el *IDE Eclipse* que contiene un plugin para instalarlo directamente desde el *IDE* mismo.

Antes de ejecutar los análisis se recomienda modificar en *src/ar/cgraio/tesis/conectores/util/Output.java* la dirección donde se guardará la salida de la herramienta. Luego para realizar la ejecución y el análisis, se debe elegir el paquete que se quiere analizar de *testcases* (o agregar uno nuevo) y dentro de las opciones que brinda el *IDE* elegir *Run as AspectJ/Java application*.

El proyecto contiene los siguientes paquetes:

- **Analyzers:** contiene el código de los analizadores y las definiciones de los distintos tipos de *joinpoints*.
- **Annotations:** contiene las definiciones de las anotaciones que son utilizadas a lo largo de la herramienta.
- **Aspects:** contiene las definiciones de los aspectos. Existe una por cada conector, además de una superclase (“superaspecto”) de la cual heredan el resto.
- **Testcases:** casos de tests variados. Se encuentra el caso de estudio, diferentes sistemas de un solo conector y otros que utilizan diferentes combinaciones.
- **Tests:** tests de la herramienta. Tanto unitarios para los analizadores como de integración sobre algunos de los testcases.
- **Util:** contiene clases y métodos “utilitarios” que son usados a lo largo de la aplicación. En particular, para escribir el *output* de la herramienta.

6.2. Output casos de estudio

Sistema sin anotar

```

1 WaterFilter^1 - callreturn - Queue^1
2 BotanicalExpert^1 - callreturn - Gardener^1
3 ClimatologicalServer^1 - callreturn - PublishSubscribeBus^1
4 Sensor^+ - callreturn - BotanicalExpert^1
5 HeatApplier^1 - callreturn - Queue^1
6 PublishSubscribeBus^1 - callreturn - BotanicalExpert^1
7 BotanicalExpert^1 - callreturn - PublishSubscribeBus^1
8 Gardener^1 - callreturn - Queue^+
9 ApplicationInterface^+ - callreturn - BotanicalExpert^1
10 WaterApplier^1 - callreturn - Queue^+
11 PublishSubscribeBus^1 - callreturn - Suscription^1
12 PublishSubscribeBus^1 - callreturn - Rule^1
13 GraphicalInterface^1 - callasync - ApplicationInterface^+
14 GraphicalInterface^1 - callasync - ApplicationInterfaceServer^1

```

Anotación *@PipePush* y *@PipePop* en *Pipe*

```

1 BotanicalExpert^1 - callreturn - PublishSubscribeBus^1
2 ApplicationInterface^+ - callreturn - BotanicalExpert^1
3 WaterFilter^1 - callreturn - Queue^1
4 PublishSubscribeBus^1 - callreturn - Suscription^1
5 BotanicalExpert^1 - callreturn - Gardener^1
6 Sensor^+ - callreturn - BotanicalExpert^1
7 ClimatologicalServer^1 - callreturn - PublishSubscribeBus^1
8 PublishSubscribeBus^1 - callreturn - Rule^1
9 HeatApplier^1 - callreturn - Queue^1
10 PublishSubscribeBus^1 - callreturn - BotanicalExpert^1
11 WaterApplier^1 - callreturn - Queue^1
12 GraphicalInterface^1 - callasync - ApplicationInterfaceServer^1
13 GraphicalInterface^1 - callasync - ApplicationInterface^+
14 {WaterApplier^1} -- pipe -- {WaterFilter^1}
15 {Gardener^1} -- pipe -- {HeatApplier^1}
16 {Gardener^1} -- pipe -- {WaterApplier^1}

```

Anotación *@Ignored* en *Pipe*

```

1 PublishSubscribeBus^1 - callreturn - BotanicalExpert^1
2 BotanicalExpert^1 - callreturn - Gardener^1
3 PublishSubscribeBus^1 - callreturn - Rule^1
4 BotanicalExpert^1 - callreturn - PublishSubscribeBus^1
5 PublishSubscribeBus^1 - callreturn - Suscription^1
6 ApplicationInterface^+ - callreturn - BotanicalExpert^1
7 Sensor^+ - callreturn - BotanicalExpert^1
8 ClimatologicalServer^1 - callreturn - PublishSubscribeBus^1
9 GraphicalInterface^1 - callasync - ApplicationInterfaceServer^1
10 GraphicalInterface^1 - callasync - ApplicationInterface^+
11 {Gardener^1} -- pipe -- {WaterApplier^1}
12 {WaterApplier^1} -- pipe -- {WaterFilter^1}
13 {Gardener^1} -- pipe -- {HeatApplier^1}

```

Anotaciones en *Publish-Subscribe*

```
1 Sensor^+ - callreturn - BotanicalExpert^1
2 ApplicationInterface^+ - callreturn - BotanicalExpert^1
3 BotanicalExpert^1 - callreturn - Gardener^1
4 {ClimatologicalServer^1} -- publishsubscribe -- {BotanicalExpert^1}
5 GraphicalInterface^1 - callasync - ApplicationInterface^+
6 GraphicalInterface^1 - callasync - ApplicationInterfaceServer^1
7 {WaterApplier^1} -- pipe -- {WaterFilter^1}
8 {Gardener^1} -- pipe -- {WaterApplier^1}
9 {Gardener^1} -- pipe -- {HeatApplier^1}
```

Anotaciones en *Client-Server*

```
1 ApplicationInterfaceServer^1 - callreturn - BotanicalExpert^1
2 Sensor^+ - callreturn - BotanicalExpert^1
3 BotanicalExpert^1 - callreturn - Gardener^1
4 {ClimatologicalServer^1} -- publishsubscribe -- {BotanicalExpert^1}
5 GraphicalInterface^1 - clientserver - ApplicationInterfaceServer^1
6 {WaterApplier^1} -- pipe -- {WaterFilter^1}
7 {Gardener^1} -- pipe -- {HeatApplier^1}
8 {Gardener^1} -- pipe -- {WaterApplier^1}
```

7. REFERENCIAS

- [1] D. Perry and A. Wolf, “Foundations for the Study of Software Architecture”, ACM Software Engineering Notes, Vol. 17, No. 4, pp. 40-52, October 1992.
- [2] D. Garlan, “Formal Modelling and Analysis of Software Architecture: Components, Connectors, and Events”, Formal Methods for Software Architectures, LNCS 2804, September 2003.
- [3] Schmerl, B., Aldrich, J., Garlan, D., Kazman, R., & Yan, H. (2006). Discovering architectures from running systems. *Software Engineering, IEEE Transactions on*, 32(7), 454-466.
- [4] Garcia, J., Ivkovic, I., & Medvidovic, N. (2013, November). A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on* (pp. 486-496). IEEE.
- [5] Garcia, J., Krka, I., Mattmann, C., & Medvidovic, N. (2013, May). Obtaining ground-truth software architectures. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 901-910). IEEE Press.
- [6] Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2009). *Software architecture: foundations, theory, and practice*. Wiley Publishing.
- [7] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., & Irwin, J. (1997). Aspect-oriented programming. In *ECOOP'97—Object-oriented programming* (pp. 220-242). Springer Berlin Heidelberg.
- [8] Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., & Little, R. (2002). *Documenting software architectures: views and beyond*. Pearson Education.
- [9] Jonathan Aldrich.(2008, Feb). Using Types to Enforce Architectural Structure. WICSA '08
- [10] D. Jackson and A. Waingold, “Lightweight Extraction of Object Models from Bytecode,” Proc. 1999 Int’l Conf. Software Eng., 1999.
- [11] R. Kazman and S.J. Carriere, “Playing Detective: Reconstructing Software Architecture from Available Evidence,” *J. Automated Software Eng.*, vol. 6, no. 2, 1999.
- [12] G.C. Murphy, D. Notkin, and K.J. Sullivan, “Software Reflexion Models: Bridging the Gap between Source and High-Level Models,” Proc. 1995 ACM SIGSOFT Symp. Foundations of Software Eng., 1995.
- [13] Bass, L., Clements, P., Kazman, R. 2003. *Software Architecture in Practice* (2nd

edition). Addison-Wesley Professional.

- [14] Barry, B. 1995. USC Center for Software Engineering. <http://www.sei.cmu.edu/architecture/start/glossary/classicdefs.cfm>
- [15] Garlan, D., Shaw, M. 1994. An Introduction to Software Architecture.
- [16] AspectJ. <https://eclipse.org/aspectj/>
- [17] RedHat. <http://www.redhat.com/en>
- [18] Enterprise Integration Pattern. <https://www.redhat.com/es/resources/enterprise-integration-pattern-eip-flashcards>
- [19] Allen, R., Garlan, D. 1994. Formalizing Architectural Connection.
- [20] L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. Analysis of Aspect-Oriented Software (ECOOP 2003), 2003.
- [21] F. Asteasuain, V. Braberman. "FVS: A declarative aspect oriented modeling language". EJS - Electronic Journal SADIO - ISSN 1514-6774. Vol 10 - No. 1 - pp 20-37- Abril 2011.
- [22] Conflicts in AspectJ: Restrictions and Solutions). Casas S., Garcia Perez-Schofield J.B., Marcos C. Revista. IEEE Latin America Transaction. ISSN 1548-0992. Vol 8 Issue 3. June 2010
- [23] Conflicts in AspectJ: Restrictions and Solutions). Casas S., Garcia Perez-Schofield J.B., Marcos C. Revista. IEEE Latin America Transaction. ISSN 1548-0992. Vol 8 Issue 3. June 2010.
- [24] Suzuki, R. (2015, May). Poster: Interactive and Collaborative Source Code Annotation. In Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on (Vol. 2, pp. 799-800). IEEE.
- [25] Joy, M. M., Becker, M., Mueller, W., & Mathews, E. (2012, December). Automated source code annotation for timing analysis of embedded software. In Advanced Computing and Communications (ADCOM), 2012 18th Annual International Conference on (pp. 12-18). IEEE.
- [26] Ferreira, G., Loureiro, E., & Oliveira, E. (2007, March). A java code annotation approach for model checking software systems. In Proceedings of the 2007 ACM symposium on Applied computing (pp. 1536-1537). ACM.