Refactorización de Aplicaciones Java Legadas Usando Desarrollo Basado en Componentes

Andrés Paz[1], Hugo Arboleda[1] y Jean-Claude Royer[2]

[1] Grupo i2t, Universidad Icesi, Calle 18, No. 122-135, Cali, Colombia
andres.paz@correo.icesi.edu.co, hfarboleda@icesi.edu.co
[2] Grupo ASCOLA, Mines de Nantes - INRIA, 4 Rue A. Kastler, 44307 Nantes, Francia
Jean-Claude.Royer@mines-nantes.fr

Resumen. Las industrias emplean un número elevado de aplicaciones de software que fueron desarrolladas usando el lenguaje Java. Muchas de estas aplicaciones soportan actividades que son críticas para las empresas y por ello están siendo modificadas constantemente. En este artículo presentamos una estrategia para asistir la refactorización de aplicaciones Java legadas a una orientación por componentes y que se basa en el principio de hacer las decisiones de arquitectura explícitas en el código fuente. Esto con el objetivo de mitigar la degradación de arquitecturas y apoyar el mantenimiento y evolución del software de las empresas. Nuestro enfoque incluye la identificación de componentes a partir del código legado de acuerdo al cumplimiento de un conjunto de reglas que aseguran de forma estática la integridad de comunicación entre los componentes identificados. Además de esto, proporcionamos acciones de refactorización de patrones de diseño orientados a objetos que solucionan violaciones al conjunto de reglas.

Palabras clave: Arquitectura de software, erosión de arquitectura, ingeniería de software orientada a componentes, integridad de comunicación, Java, refactorización, aplicaciones legadas, patrones de diseño.

1 Introducción

La industria del software debe lidiar con el dinamismo del mundo real. Es por esto que las aplicaciones de software deben ser mantenidas para corregir fallas, mejorar su seguridad, desempeño u otros atributos de calidad, e integrar nuevas funcionalidades. Sin embargo, a medida que se mantienen y evolucionan estos sistemas se produce una degradación gradual, o *erosión*, de su arquitectura como consecuencia. Esta *erosión* sucede porque los desarrolladores no llegan a ser conscientes de las intenciones arquitecturales del sistema que están desarrollando, es decir, no comprenden completamente la arquitectura. Algunos ejemplos de estas intenciones pueden ser: *disponibilidad* del sistema, *seguridad*, *desempeño*, *escalabilidad*. Las intenciones arquitecturales se encuentran expresadas en las especificaciones de diseño que representan la arquitectura del software y son una parte esencial para la comprensibilidad, documentación, construcción, evolución, análisis, verificación,

reusabilidad y gestión de cualquier desarrollo de software [1]. Las especificaciones de diseño son luego traducidas a código por los desarrolladores. Cada que los desarrolladores realizan cambios en el código del sistema sin tener en cuenta las especificaciones de diseño del mismo, hace que se incumplan cada vez más las intenciones arquitecturales originales.

Se han propuesto diversos lenguajes, métodos de desarrollo y refactorización en beneficio de los profesionales del software para mitigar la erosión arquitectural, y soportar el mantenimiento y evolución del software [2]. La ingeniería de software orientada a componentes [3] (CBSE, por sus siglas en inglés, *Component-Based Software Engineering*) es una línea central de la ingeniería de software que se ha interesado por la modularización, separación de preocupaciones (separation of concerns), y arquitectura del software. En CBSE, los sistemas se construyen a partir de *componentes de software* reutilizables que proveen y requieren unos servicios, y que tienen enlaces de comunicación bien definidos entre ellos. Un componente de software, de acuerdo con la definición dada en [4], *i)* es una unidad, es decir, no se puede dividir y es autocontenido (excepto por su interfaz declarada), *ii)* especifica una interfaz (o interfaces) con los servicios que provee, *iii)* especifica dependencias de contexto, es decir, los servicios que requiere para funcionar correctamente, y *iv)* puede ser parte de un *componente compuesto*. Un componente compuesto se construye a partir de otros componentes. Estas propiedades hacen que cualquier componente pueda ser reemplazado por otro que provea los mismos servicios o unos servicios que sean compatibles.

Como se mencionó en el párrafo anterior, CBSE impulsa la modularización del software y hace que las intensiones arquitecturales sean explícitas. Además de esto, también permite verificar restricciones arquitecturales y usar principios de programación estrictos como la propiedad de *integridad de comunicación* (IC) [5, 6]. La propiedad de IC establece que dos componentes sólo se pueden comunicar si se ha definido previamente un canal de comunicación entre ellos. Debido a esta restricción no existen y no pueden existir canales de comunicación que estén ocultos. Esta es una de las fortalezas que presenta una orientación a componentes, lo que le permite a los ingenieros especificar explícitamente y revisar automáticamente algunas de las decisiones de arquitectura. De esta forma se logra limitar activamente la posibilidad de erosionar la arquitectura del software.

Hemos trabajado en la definición de una estrategia y una herramienta para reestructurar código Java legado. Una representación de la arquitectura de la aplicación se extrae y se refactoriza a una arquitectura orientada por componentes siguiendo un modelo de componentes ligero y un conjunto de reglas para verificar potenciales violaciones a la propiedad de IC [7, 8]. Estas reglas permiten distinguir entre un tipo dato y un tipo componente al tener en cuenta el principio de que un tipo componente no viola ninguna de ellas, mientras que un tipo dato sí lo hace. En particular, en este artículo presentamos una vista general del conjunto refinado de reglas de [8] junto con algunos ejemplos de estas. No es nuestro objetivo presentar un listado exhaustivo de ellas, por lo que remitimos al lector a [7, 8]. Nuestro trabajo de refactorización de aplicaciones Java con el conjunto de reglas nos ha llevado a proponer acciones de refactorización para patrones de diseño pues son muy valiosos para las aplicaciones orientadas a objetos, sin embargo, la mayoría de ellos no cumple con la propiedad de IC.

Este artículo está organizado de la siguiente manera. La sección 2 presenta los trabajos relacionados de distintos autores y describe sus características. En la sección 3 se describen los principales elementos de nuestro modelo de componentes y se explica de forma general el conjunto de reglas que empleamos para asegurar la propiedad de IC en código Java legado. La sección 4 contiene las acciones de refactorización de patrones de diseño. Y finalmente, en la sección 5 se expone un resumen de nuestras contribuciones y el trabajo futuro.

2 Trabajos Relacionados

Consideramos los trabajos relacionados con la recuperación de componentes [9] y la prevención de la degradación de las arquitecturas en programación Java. Diferentes autores han propuesto estrategias para mitigar la erosión de las arquitecturas de software de las aplicaciones Java. Entre los enfoques abordados para enfrentar el problema se cuentan, entre otros, el reconocimiento de arquitecturas y la recuperación de arquitecturas [10], el uso de métricas en procesos de refactorización, extracción de componentes y el análisis de conformidad de arquitecturas. A continuación se describen brevemente estos enfoques.

Entre los diferentes desarrollos que evalúan la desviación de la arquitectura de una aplicación y tratan de reparar, entender y prevenir este fenómeno está el reconocimiento de arquitectura (architecture recognition) y la recuperación de arquitectura (arquitecture recovery). De acuerdo con estos enfoques, generalmente se analiza el código fuente y se compara contra una colección de patrones para generar un modelo de información que contiene componentes y conectores. Mendonça y Kramer [11] analizaron los límites de algunas herramientas de recuperación de la arquitectura de sistemas legados. El reconocimiento y la recuperación de arquitecturas no son convenientes porque, por un lado, no es siempre posible hacer coincidir estos elementos. Por otro lado, se limita la recuperación de componentes que ya existen en la aplicación. Estas son estrategias complementarias a nuestro enfoque, aunque más generales que detectar las violaciones potenciales de IC.

En [12], los autores combinan la recuperación de arquitectura y el análisis de dependencias de cambios (change dependency analysis), y consideran los archivos fuente de la aplicación Java como componentes. Esto reconoce componentes en algunos casos “artificiales”, es decir, componentes que se asumen como tales por la ubicación común de los archivos pero no por sus responsabilidades asociadas. Esto hace que el enfoque se aleje del principio que promueve CBSE y por ello se dejan de aprovechar sus ventajas mencionadas en párrafos anteriores.

En el contexto de refactorización existen varias herramientas de ayuda para evaluar la calidad de las aplicaciones y guiar los procesos de restructuración. Sin embargo, hay una carencia de herramientas para evaluar la calidad de código fuente orientado a componentes. Las herramientas basadas en métricas, es decir, en medidas cuantificables y reproducibles, son un ejemplo. Pero manejar métricas en componentes de software y arquitecturas es todavía un área de investigación que no es lo suficientemente madura para proporcionar un conjunto estandarizado de métricas y soporte para herramientas. En [13] y [14] se hace referencia a dos estudios existentes

sobre métricas de componentes de software. En sus conclusiones, [13] establece que *"se necesita más trabajo"* para madurar los enfoques de métricas de componentes de software y los autores también señalan una carencia de herramientas automatizadas. Goulão y Abreu en [14] revisaron y compararon los enfoques existentes; ellos señalan que *"hay una falta de madurez en las propuestas existentes"*. Las herramientas basadas en métricas dan información acerca de las cualidades globales del software tales como la reusabilidad, mantenibilidad y capacidad de evolución. No obstante, ellas no son adecuadas para comprobar propiedades semánticas como la propiedad de IC. En la actualidad, los enfoques existentes no se enfrentan con subtipos de componentes lo que podría afectar seriamente las definiciones de las métricas.

Las herramientas de extracción de componentes y tipos compuestos [15], [16] y [17] están generalmente basadas en métricas. Estas herramientas pueden ser utilizadas para sugerir componentes y luego para evaluar la calidad de los resultados. Ellas se dedican a la búsqueda de nuevos tipos de componentes en el código fuente que no es necesariamente código CBSE o no está diseñado con el rigor necesario. Las herramientas PMD [18] y FindBugs [19] hacen un análisis estático del código fuente o del bytecode, pero no son adecuadas para evaluar la calidad CBSE, o incluso para asegurar un buen estilo de programación con componentes jerárquicos.

Inspectores de conformidad de arquitectura, como SAVE [20], están dedicados a verificar la conformidad de una arquitectura dada con la que es extraída del código fuente. Se basan en un modelo de componentes ligero que no comprende subtipos de componentes ni componentes de primer nivel. Además, requieren una arquitectura dada, que tal vez puede no tenerse a la mano.

ArchJava [6, 21] es una extensión de Java que hace cumplir la integridad del flujo arquitectural de comunicación a través de reglas complejas. El enfoque hace uso de un compilador basado en el Java Reflection API que impone la propiedad de IC. Sin embargo, el uso de un lenguaje nuevo, aún en el caso de un lenguaje basado en Java, tiene dos implicaciones negativas. Por un lado, implica una curva de aprendizaje para desarrolladores Java en contextos donde el mantenimiento ya es por sí mismo una actividad altamente consumidora de tiempo. Y por otro lado, las aplicaciones a las que se les hace mantenimiento dejan de estar en un lenguaje bien conocido y con una amplia comunidad, Java, y pasan a ser escritas en un lenguaje donde no hay experticia comprobada por parte de la comunidad de practicantes y desarrolladores. AliasJava [22] es una extensión de ArchJava, que permite la detección de comunicación a través de datos compartidos. La estrategia de AliasJava sigue los principios de ArchJava presentando los mismos inconvenientes en lo referente a curva de aprendizaje del lenguaje y aplicaciones nuevas desarrolladas en un nuevo lenguaje poco conocido y sin documentación ni soporte difundido.

3 Modelo de Componentes y Reglas de IC

Como parte de nuestro trabajo previo hemos desarrollado una estrategia y herramienta para reestructurar código Java legado siguiendo un modelo de componentes basado en clases e interfaces. En [7] definimos la herramienta que permite reconocer componentes existentes en el código fuente Java. Además, establecimos el modelo de

componentes y un conjunto de reglas para verificar potenciales violaciones a la propiedad de IC. En [8] presentamos un conjunto ampliado de reglas para detectar las violaciones a la propiedad de IC y varias acciones de refactorización que corrigen las violaciones detectadas por la herramienta. Esta sección presenta una breve vista general del modelo de componentes y del conjunto de reglas de IC propuestos.

3.1 Modelo de Componentes

En nuestro enfoque empleamos un modelo de componentes estricto donde todos sus elementos tienen un equivalente directo con una construcción Java. Citando a [23]: *"El beneficio de esta asignación es acortar la distancia entre una arquitectura basada en componentes y su implementación, lo que mejora la reusabilidad, adaptabilidad y capacidad de mantenimiento de sistemas de software basados en componentes."* El modelo de componentes se basa en los siguientes principios: *i)* los tipos componentes son tipos verdaderos, lo que significa que se pueden instanciar para generar componentes, *ii)* los componentes se comunican a través de una política estricta de paso de mensajes basada en llamado de métodos, *iii)* los componentes pueden ser concretos o abstractos, *iv)* se soportan subtipos, y *v)* los componentes compuestos se construyen a partir de una estructura de clases que contiene subcomponentes.

Este modelo de componentes tiene asociado un conjunto de reglas capaz de hacer cumplir estáticamente la propiedad de IC en código legado. Cuando se refactoriza una aplicación, nuestra idea es considerar un tipo solo como un tipo componente si no es responsable de violaciones a estas reglas de IC. Por el contrario, un tipo dato puede utilizarse para crear estructuras de datos, por lo que puede ser usado para crear enlaces ocultos.

Dado que estamos considerando el análisis estático de código fuente, sólo tratamos con la información de tipos. Al conjunto de tipos (clases, interfaces, genéricos) en el código fuente los llamamos los tipos de interés. En nuestro modelo de componentes agrupamos estos tipos de interés en tres categorías: *i)* tipos dato (DTipos), *ii)* tipos componente (CTipos), y *iii)* tipos externos (ETipos). Una instancia de un CTipo es un componente, mientras que un valor es una instancia de un DTipo. Un ETipo es un tipo externo al proyecto en estudio.

Nuestro modelo de componentes proporciona algunas características originales como un tipo compuesto verdadero, subtipos y reglas de IC. Cabe resaltar que no todas estas características son consideradas en los modelos de componentes de los enfoques presentados en la sección 2, excepto en ArchJava [21]. Sin embargo, no deja de ser bastante estricto. Por ejemplo, no consideramos el uso del Java Reflection API como en ArchJava [6] o Fractal [24]. Esto se debe a que nuestra intención es proporcionar un modelo de componentes Java puro que es fácilmente accesible desde código legado.

3.2 Reglas de IC

Las propiedades básicas de CBSE son difíciles de hacer cumplir, y pocos lenguajes son capaces de hacerlo (*e.g.* [5, 21, 22]). Es por esto que la propiedad de IC es difícil de comprobar estáticamente. Los fundamentos y las reglas de nuestro enfoque provienen mayormente de ArchJava, pero los hemos modificado y ampliado. En nuestro enfoque consideramos que es valioso elegir un conjunto de reglas estáticas y más estricto. Nuestras reglas se utilizan para calificar los tipos de interés en DTipos y CTipos, e informar infracciones adicionales. Estas infracciones deben ser analizadas y corregidas por el diseñador de software en un paso de refactorización avanzado. Si no se reportan infracciones, nuestro conjunto de reglas asegura estáticamente que las instancias calificadas como CTipos no escapan de su componente padre que los contiene.

Al igual que en el lenguaje ArchJava, evitamos las comunicaciones ocultas que se han establecido a través del intercambio de datos. Sin embargo, ArchJava emplea información adicional y explícita acerca de las conexiones, patrones y expresiones de conexión. En nuestro enfoque utilizamos el código Java puro y no tenemos construcciones específicas para definir enlaces de comunicación. Los canales de comunicación entre tipos son una información muy débil para ser usada como criterio necesario. Así, sin más información no podemos asegurar completamente, como lo hace ArchJava, que las instancias de los componentes, creadas estática o dinámicamente, se comunican como se espera. No obstante, argumentamos que depender de reglas estáticas y no tener información arquitectural explícita en el código son supuestos razonables para una refactorización sin problemas de código Java legado.

Siguiendo el trabajo sobre ArchJava [6, 21], consideramos que un componente no se puede pasar como una referencia en un servicio ni ser una parte de un tipo de interés. Modificamos algunas reglas sobre subtipos y prohibimos algunos downcasts en lugar de agregar controles dinámicos para ellos. Adicionamos nuevas reglas para verificar el uso de genéricos, excepciones y enumeraciones. El conjunto de reglas cuenta con 18 reglas agrupadas en las siguientes 7 categorías:

1. **Signaturas incorrectas:** Las reglas de signaturas incorrectas buscan signaturas de métodos incorrectas en la aplicación bajo estudio. Estas signaturas incorrectas son prohibidas en CTipos. Como ejemplo de las reglas de IC presentamos las reglas de esta categoría:
    1) Tipos que se pasan como parámetros de, o retornados por, servicios incluidos en CTipos o DTipos son DTipos; la signatura del método se califica como una signatura incorrecta.
    2) La regla 1) también se aplica a cualquier constructor, independientemente de su modificador.
    3) Métodos no visibles pueden tener tipos componentes en sus signaturas, siempre y cuando se les llame en `this` o `super`.
2. **Composición:** Las reglas de composición buscan referencias a componentes que han sido encapsuladas como valores, pues pueden ser capturados y enviar mensajes indirectamente al componente.

3. **Subtipos:** Las reglas de subtipos buscan que los subtipos de CTipos no sean usados como parámetros o resultados en servicios. Sin embargo, DTipos pueden ser subtipos de CTipos.
4. **Arreglos y genéricos:** Las reglas para arreglos y genéricos buscan accesos indirectos a CTipos en las referencias guardadas en los arreglos y genéricos. En esta categoría añadimos nuevas reglas, por ejemplo: un CTipo no puede ocurrir en una *realización de un genérico*[1].
5. **Clases anidadas:** Las reglas para clases anidadas buscan que las clases anidadas dentro de DTipos sean DTipos.
6. **Clases de tipo excepción:** Las reglas para clases de tipo excepción buscan que toda clase de tipo excepción sea un DTipo.
7. **Clases de tipo enumeración:** Las reglas para las clases de tipo enumeración buscan que toda clase de tipo enumeración sea un DTipo.

Para resumir las demás reglas, las dos clases de tipos que somos capaces de identificar (DTipo y CTipo) tienen las siguientes propiedades. Ambas clases de tipos no pueden definir atributos visibles ni estáticos de un tipo *componente agregado*. Un tipo componente agregado se define como un tipo componente, un arreglo de tipos componente o una realización de un genérico de tipos componente. Métodos no visibles pueden tener en sus signaturas tipos componente agregado. Un DTipo es un tipo normal pero no puede tener atributos (visibles o no visibles) de tipos componente agregado. Un CTipo: *i)* no es una clase de excepción o enumeración, *ii)* no puede tener un DTipo como clase interna, *iii)* no hereda de un DTipo, genérico o realización de un genérico, *iv)* no es un parámetro de tipo formal de un arreglo o genérico visible, y *v)* puede tener atributos no visibles de tipo componente agregado. En un CTipo, los miembros no visibles sólo deben llamarse en `this` o `super`. Para un listado detallado de las reglas de IC, remitimos al lector a [7, 8].

En código legado pueden existir muchos problemas relacionados con violaciones de la propiedad de IC. Para evitar imponerle al usuario la carga adicional de lidiar con todos estos problemas, hemos propuesto que los problemas más generales, es decir, los que tienen relación con subtipos, composición y genéricos, deben ser reportados y corregidos en primera instancia. Definimos entonces dos niveles de control para nuestro enfoque. Primero se encuentra el conjunto de reglas para detectar las violaciones de IC y clasificar los tipos de interés en DTipos y CTipos. El segundo nivel de control se dirige a los problemas menos importantes o más locales, a los cuales llamados infracciones, y que se reportan al usuario por separado. La razón de esta distinción entre violaciones a las reglas de IC e infracciones es que las primeras están relacionadas directamente con los tipos de interés, sus métodos y atributos. En cambio, las infracciones están más relacionadas con las expresiones específicas que se encuentran dentro del código, como son limites de comunicación, expresiones de conversión (cast), signaturas ilegales de subtipos y llamadas a miembros no visibles. No es recomendable solucionar los problemas locales en el código antes de tener un conjunto adecuado de tipos componente.

[1] Una realización de un genérico es el resultado de sustituir los tipos actuales por parámetros de tipos formales en una definición de un genérico.

4 Refactorización de Patrones de Diseño

En nuestros experimentos refactorizando aplicaciones Java, en particular de las disponibles en [25], con las reglas presentadas en la sección 3.2, nos encontramos con algunas limitaciones de nuestro enfoque. Algunos patrones de diseño de software [26, 27] generaron varias violaciones a estas reglas. En la refactorización de las aplicaciones se realizaron varias acciones para eliminar, donde fuera posible, las violaciones causadas por estos patrones. En esta sección presentamos las acciones aplicadas a algunos de estos patrones de diseño y nuestras conclusiones acerca de ellos. No es nuestra intención presentar una discusión amplia de cada patrón de diseño y solo tuvimos en cuenta su forma general. Esto último implica que las conclusiones podrían cambiar en implementaciones particulares.

Figura 1. Patrón Singleton

El patrón *singleton* de la Figura 1 es una forma de definir un valor compartido. Este valor no puede ser un componente debido a que viola las reglas de la categoría signaturas incorrectas. Si no se espera que el valor sea visible fuera de su padre que lo encierra, el cual es un caso excepcional cuando se implementan patrones singleton, crear un atributo no modificable y no visible es una buena alternativa para transformar el DTipo en un CTipo. Este es un punto ya identificado en [28]. Sin embargo, esta acción de refactorización no es aplicable cuando la intención del patrón es compartir un atributo público y estático.

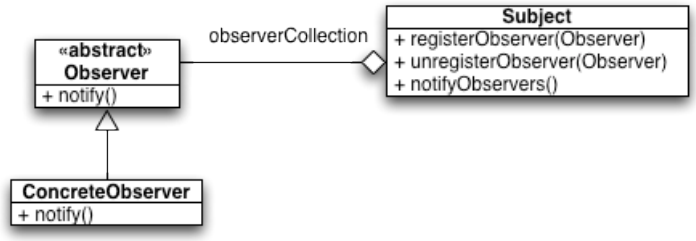

Figura 2. Patrón Observer

La implementación del patrón *observer* de la Figura 2 permite que un objeto observable sea monitoreado por un observador suscrito, quien recibe las notificaciones cuando cambia el estado del objeto observable. Varias signaturas incorrectas con malos argumentos surgen en la implementación general del patrón debido a que se violan las reglas de la categoría signaturas incorrectas. Esto hace que el observable y el observador sean marcados como DTipos. Para solucionar esto, por un lado, se pueden utilizar los atributos disponibles en la estructura del observable como parámetros en el servicio de actualización del observador para reemplazar la instancia del observable. Por otro lado, la aparición del observable en el servicio de actualización del observador puede ser reemplazado con un *objeto valor* [29]. Un

objeto valor (Value Object o VO), actualmente conocido como objeto de transferencia de datos (DTO, por sus siglas en inglés, *Data Transfer Object*), es un patrón de diseño utilizado para transportar datos entre niveles y capas para reducir el número de llamados a métodos. El patrón objeto valor puede ser utilizado como medio para eliminar signaturas incorrectas. Las acciones anteriores se aplicarán también para cualquier instancia del observador que se encuentre en un servicio del observable. Sin embargo, estas soluciones sólo permiten que el observador tenga acceso a los datos desde el observable pero evitan el uso de los servicios de este.

De nuestros experimentos con el conjunto de reglas aplicado sobre patrones de diseño podemos concluir que al refactorizar la mayoría de estos patrones: *i)* se cambia el comportamiento de la aplicación bajo estudio, *ii)* se restringe al patrón a operar en un contexto de trabajo limitado, y/o *iii)* se aumenta el acoplamiento de la aplicación, que en consecuencia aumenta los canales de comunicación ocultos en algunas partes que podrían estar fuera del alcance del patrón.

Como alternativa a estos inconvenientes con los patrones de diseño orientados a objetos (OO) ha surgido el uso de los bien conocidos actuales patrones de diseño de Edición Empresarial (EE) [29], con el fin de poder preservar la propiedad de IC. Tales patrones de diseño EE están concebidos para ser utilizados en el contexto de las aplicaciones CBSE; el mismo contexto de nuestro trabajo. No hemos estudiado numerosos patrones de diseño EE dirigidos al contexto de este trabajo para confirmar que ninguno de ellos viola la propiedad de IC, pero su naturaleza CBSE nos da un buen punto de partida para utilizarlos como una alternativa para los patrones de diseño OO. Como se ve anteriormente, la implementación del patrón de diseño Objeto Valor es un ejemplo de un patrón de diseño EE usado para refactorizar aplicaciones OO. El patrón de diseño Objeto Valor se presenta como un comodín para solucionar violaciones de nuestras reglas de IC cuando los tipos de interés escapan de sus padres envolventes, pero sólo se utilizan para consultar o configurar sus datos. Sin embargo, el uso de objetos valor tiene algunos inconvenientes. Por ejemplo, se puede requerir de memoria adicional y afectar el desempeño de la aplicación. Un mapeo completo entre los patrones de diseño EE y los patrones de diseño OO como medio de refactorización de aplicaciones OO en aplicaciones CBSE, sigue siendo actualmente un tema abierto de investigación.

5 Conclusiones

Hacer que las decisiones arquitecturales queden más explícitas en el código fuente de las aplicaciones es una tendencia para mitigar la erosión arquitectural y apoyar la evolución y el mantenimiento del software. La propiedad de integridad de comunicación (IC) es uno de los medios que puede emplearse para esto y para mantener la consistencia entre la documentación de la arquitectura y la implementación. A pesar de todo el trabajo desarrollado en torno a ArchJava, esta propiedad no ha sido ampliamente utilizada. En este contexto, hemos propuesto un conjunto de reglas que aseguran la propiedad de IC en código Java legado de acuerdo con un modelo de componentes Java.

En tanto se reutilizan principalmente las reglas de ArchJava, exponemos varias diferencias. Nuestro enfoque considera una estricta comprobación estática, detección detallada de componentes y pasos pequeños de refactorización. Además, la estrategia incremental aplicada ayuda a garantizar que las medidas de refactorización son confiables. Asimismo, agregamos nuevas reglas para subtipos, genéricos, excepciones y enumeraciones.

Analizamos varios proyectos Java en más profundidad, y hemos identificado algunas limitaciones de nuestro enfoque. Por ejemplo, el hecho de que los patrones de diseño no son compatibles con él. Estos limites implican que nuestro enfoque debe utilizarse con precaución, pues refactorizar completamente una aplicación con tipos componentes no siempre es deseable, incluso si es posible.

Respetar la propiedad de IC puede traer algún valor en la evaluación y refactorización de aplicaciones Java. Sin embargo, esta no es una tarea fácil y se requiere de un buen conjunto de reglas. Futuros trabajos se centrarán en abordar la separación del conjunto de reglas en subconjuntos que estarían dirigidos a restringir y "relajar" los procesos de extracción de componentes y de refactorización. Adicionalmente, se espera probar estos subconjuntos con varios proyectos Java, y comparar y analizar los resultados que se obtengan. También buscamos mejorar el conjunto de acciones de refactorización propuesto para que incluya el rediseño de patrones de diseño de software orientados a objetos en el contexto de desarrollo basado en componentes y la propiedad de IC.

Referencias

1. Garlan, D.: Software architecture: a roadmap. En: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, pp. 91--101. ACM, New York, NY, USA (2000)
2. Hochstein L., Lindvall, M.: Combating architectural degeneration: a survey. En: Information & Software Technology, Volume 47, Issue 10, pp. 643--656. (2005)
3. Medvidovic, N., Taylor, R. N.: A classification and comparison framework for software architecture description languages. En: IEEE Transactions on Software Engineering, Volume 26, Issue 1, pp. 70--93. (2000)
4. Bosch, J., Szyperski, C. A., Weck, W.: Component-oriented programming. En: ECOOP Workshops, pp. 34--49. (2003)
5. Luckham, D. C., Kenney J. L., Augustin, L. M., Vera, J., Bryan, D., Mann, W.: Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, Volume 21, Issue 4, pp. 336--355. (1995)
6. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: connecting software architecture to implementation. En: Proceedings of the 24th International Conference on Software Engineering (ICSE-02), pp. 187--197. ACM Press (2002)
7. Arboleda, H., Royer, J. C.: Component types qualification in Java legacy code driven by communication integrity rules. En: Proceedings of the 4th India Software Engineering Conference, ISEC '11, pp. 155—164. ACM, New York, NY, USA (2011)
8. Arboleda, H., Royer, J. C.: Java component refactoring based on communication integrity violations. En: Proceedings of the 9th Belgian-Netherlands Software Evolution Seminar, pp. 115--129. Lille, France (2010)
9. R. Koschke. Atomic Architectural Component Recovery for Program Understanding and Evolution. Proceedings of the International Conference on Software Maintenance (ICSM'02), pp. 478--. IEEE Computer Society, Washington, DC, USA (2002)

10. Vasconcelos, A., Werner, C. Architecture recovery and evaluation aiming at program understanding and reuse. Proceedings of the Quality of software architectures 3rd international conference on Software architectures, components, and applications (QoSA'07), pp. 72—89. (2007)
11. Mendoça, N. C., Kramer, J.: Requirements for an effective architecture recovery framework. En: Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, pp. 101--105. ISAW '96, ACM, New York, NY, USA (1996)
12. Stringfellow, C., Amory, C. D., Potnuri, D., Andrews, A. A., Georg, M.: Comparison of software architecture reverse engineering methods. En: Information & Software Technology, Volume 48, Issue 7,, pp. 484--497. (2006)
13. Mahmood, S., Lai, R., Kim, Y.-S., Kim, J. H., Park, S. C., Oh, H. S.: A survey of component based system quality assurance and assessment. En: Information & Software technology, Volume 47, Issue 10, pp. 693--707. (2005)
14. Goulão, M., Abreu, F. B.: Software Quality Measurement: Concepts and Approaches. En: Information Technology, Chapter: An overview of metrics-based approaches to support software components reusability assessment. ICFA Books, India (2007)
15. Chouambe L., Klatt, B., Krogmann, K.: Reverse engineering software-models of component-based systems. En: CSMR, IEEE, pp. 93--102. (2008)
16. Chardigny, S., Seriai, A., Tamzalit, D., Oussalah, M.: Quality-driven extraction of a component-based architecture from an object-oriented system. En: CSMR, IEEE, pp. 269—273. (2008)
17. Becker, S., Hauck, M., Trifu, M., Krogmann, K., K. N, J.: Reverse engineering component models for quality predictions. En: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, pp. 199—202. European Projects Track (2010)
18. PMD, http://pmd.sourceforge.net/ (2009)
19. FindBugs – Find Bugs in Java Programs, http://findbugs.sourceforge.net/ (2009)
20. Knopel, J., Lindvall, M.: Software architecture visualization and evaluation. http://www.fc-md.umd.edu/save/about.aspx (2009)
21. Aldrich, J., Chambers, C., Notkin, D.: Architectural reasoning in ArchJava. En Proceedings ECOOP 2002, Vol. 2374 of LNCS, pp. 334—367. Springler Verlag (2002)
22. Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. En: M Odersky (ed.), ECOOP '04 – Object-Oriented Programming European Conference, Volume 3086 of Lecture Notes in Computer Science, pp. 1--25. Springeler-Verlag, Oslo, Norway (2004)
23. da Silva Jr., M. C., de Castro Guerra, P. A., Rubira, C. M. F.: A Java component model for evolving software systems. En: ASE, pp. 327--330. IEEE Computer Society (2003)
24. Bruneton, E., Coupaye, T., Leclereq, M., Quéma, V., Stefani, J.-B.: The Fractal Component Model and Its Support in Java. Software Practice and Experiencia.
25. Proyecto Cupi2, http://cupi2.uniandes.edu.co (2011)
26. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
27. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd Edition, Prentice Hall (2004)
28. Abi-Antoun, M., Aldrich, J., Coelho, W.: A case study in re-engineering to enforce architectural control flow and data sharing. Journal of Systems and Software, Volume 80, Issue 2, pp. 240--264. (2007)
29. Alur, D., Malks, D., Crupi, J.: Core J2EE Patterns: Best Practices and Design Strategies.. 2nd edition, Prentice Hall (2003)