View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CONICET Digital

Representación de la Evolución y Refactoring de Arquitecturas de Software mediante la Aplicación y Captura de Operaciones Arquitectónicas

María Luciana Roldán[1], Silvio Gonnet[2], Horacio Leone[3]

[1,2,3]INGAR(UTN-CONICET)

Facultad Regional Santa Fe, Universidad Tecnológica Nacional

Avellaneda 3657, 3000, Santa Fe, Argentina

[1]lroldan@santafe-conicet.gov.ar, [2]sgonnet@santafe-conicet.gov.ar, [3]hleone@santafe-conicet.gov.ar

Resumen: *La evolución de arquitecturas de software es consecuencia de cambios como la redefinición de requerimientos, o mejoras en la infraestructura/tecnología del sistema. Es necesario que la introducción de cambios arquitectónicos sea realizada de manera sistemática, a fin de evitar la erosión en el diseño arquitectónico y la pérdida de información vital para la comprensión del diseño obtenido. Los cambios aplicados y las decisiones tomadas deben ser documentados adecuadamente, para que se puedan recuperar posteriormente las soluciones aplicadas y conocer su impacto en la arquitectura. Se propone un modelo para representación del conocimiento durante la evolución de arquitecturas de software, basado en la aplicación de operaciones de evolución y refactoring. Las operaciones ejecutadas son capturadas junto con los elementos arquitectónicos sobre los que operaron, los resultados obtenidos, y los objetivos perseguidos, manteniendo así las trazas entre las diferentes versiones del modelo arquitectónico alcanzadas y la historia completa de su evolución.*

Palabras Claves: *Evolución de arquitecturas de software; Refactorización; Trazabilidad de procesos de diseño; Decisiones arquitectónicas.*

Abstract*: Software arquitectures evolution occurs as a consequence of changes, such us requiremens redefinition or infrastructure/technology improvements. The applying of architectural changes should be done in a systematic way, in order to avoid the design erosion and the lost of important information about the design process that is useful for understanding the obtained design. Applied changes and decisions made should be properly documented in order to make possible recovering them later and understanding their impact on the software architectures. In this work, a model for representing the generated and applied architectural knowledge during software architectures evolution processes is proposed, which it is based on evolution and refactoring operations. The executed operations are captured along with the architectural elements on which they operated, the resulting outcomes, and the pursued design goals. In this way, the approach keeps the traces among the several achieved versions of the software architecture model and the whole evolution history.*

Keywords: *Software architecture evolution; Refactoring; Design process tracing; Architectural decisions.*

INTRODUCCIÓN

La evolución de arquitecturas de software se produce como consecuencia de la aparición de los cambios, una constante en la ingeniería de software, y sobre los que los arquitectos y desarrolladores tienen poco o ningún control. Los cambios en las arquitecturas de software tienen un origen diverso, como la redefinición o modificación de requerimientos, el cambio en la infraestructura o la tecnología en la que se basa un sistema, o como consecuencia de errores o malas decisiones que

deben ser corregidos, cambios en las estructuras organizaciones o procesos de negocios, o nuevas legislaciones. Para que la arquitectura de software no se torne de un tamaño inmanejable y se vuelva difícil de mantener, los arquitectos deberían evitar la introducción caótica de cambios y hacerlo de una manera sistematizada y ordenada. En este aspecto, el refactoring de arquitecturas de software propone mejorar su estructura sin que ello implique cambiar el comportamiento externo del sistema (Fowler et al., 1993). Para evitar el fenómeno que se conoce como "erosión arquitectónica", se han propuesto métodos y patrones de refactoring que brindan a los ingenieros de software soluciones probadas para afrontar necesidades de refactorización recurrentes (Stal, 2014; Stal, 2015). Un ejemplo de estos patrones de refactoring o reestructuración arquitectónica, es el que propone transformar una arquitectura monolítica en una arquitectura cliente-servidor, o en una arquitectura de tres capas que separa claramente la interfaz de usuario, de la lógica de negocio, y la capa de datos.

En este contexto, la evolución de una arquitectura de software se trata de la implementación sistemática de cambios arquitectónicos que se traducen en la adición, eliminación y modificación de los elementos que la conforman, a fin modificar el modelo arquitectónico. Estos cambios ciertamente tienen impactos en la arquitectura de software original, por lo que no deberían llevarse a cabo sin conservar ninguna información de las causas de los mismos, objetivos perseguidos, ni qué elementos se vieron afectados, o qué razones existieron que fundamentan las decisiones de los mismos. Sin embargo por diversos motivos, este "conocimiento" que explica mediante qué transformaciones se llevó a cabo el refactoring de una arquitectura de software no queda documentado o codificado, perdiéndose con el tiempo. Las causas más comunes de esta ausencia de documentación del proceso de evolución son las mismas que se aplican al diseño de la arquitectura de software (Avgeriou et al., 2007; Falessi et al., 2008); están relacionadas a las limitaciones que presentan las herramientas de soporte al proceso, negligencia de parte de los arquitectos o mantainers como consecuencia de falta de tiempo y sobrecarga de trabajo, o la percepción de que la documentación es un "overhead" del cual no obtienen beneficios, ya que ellos quienes producen ese conocimiento, pero muchas veces no son quienes van a emplearlo o "consumirlo". Se hace entonces necesario contar con herramientas que posibiliten la captura de ese conocimiento en forma natural, donde la carga extra en trabajo de refactoring debido a su documentación sea mínimo. A fin de brindar una base para el desarrollo de tales herramientas, en este trabajo se propone un modelo para la captura y representación del conocimiento durante la evolución arquitecturas de software, el cual se basa en la captura de las operaciones de refactoring aplicadas, las cuales son conservadas junto con los nuevos elementos arquitectónicos generados, los objetivos perseguidos con la actividad de refactoring, y que además mantiene la trazabilidad a los elementos arquitectónicos afectados. Dicho modelo se basa en propuestas anteriores de los autores para la captura y representación de diseños ingenieriles (Roldán et al., 2010; Roldán et al., 2013).

TRABAJOS RELACIONADOS

Se han reportado numerosos trabajos de investigación en relación a la evolución de arquitecturas de software (Breivold et al., 2012). Los principales temas dentro de esta línea se relacionan a las propuestas de técnicas que dan soporte al diseño de arquitecturas de software teniendo en cuenta atributos de calidad, la evaluación de la calidad de arquitecturas de software, valoraciones econó-

micas, gestión del conocimiento arquitectónico, y técnicas de modelado. Prácticamente no existen herramientas dedicadas al soporte de la evolución y refactoring de arquitecturas de software, y mucho menos encaran la problemática de conservar y documentar la evolución de una arquitectura de software, es decir, la historia del proceso de refactorización.

Stal (2014) ha contribuido en trasladar los conceptos de refactorización de código (Fowler et al., 1993), como "code smells" (potenciales áreas de código que deberían ser mejoradas), a las arquitecturas de software. También ha catalogado numerosas soluciones recurrentes para el refactoring de arquitecturas de software (Stal, 2015) (patrones de refactoring), para lo cual ha propuesto una forma canónica de documentación de patrones.

Barnes et al. (2014) describen una estrategia teórica para planificar y razonar sobre evolución arquitectónica. Ésta provee elementos para modelar caminos de evolución potenciales y realizar análisis para seleccionar entre los caminos candidatos. El enfoque tiene similitudes con el propuesto en este trabajo en cuanto a que un camino de evolución planificado se modela formalmente como una secuencia de arquitecturas en la cual el primer elemento en el camino es la arquitectura del sistema actual, y el elemento final es la arquitectura deseada. Los enlaces entre los nodos sucesivos en un camino se describen en término de transformaciones arquitectónicas que son seleccionadas a partir de un conjunto predefinido de operadores de evolución. Para mantener registro de cómo evolucionan elementos individuales de la arquitectura, se proponen el uso de etiquetas, empleando una propiedad de identidad de evolución que relaciona los elementos idénticos que existen en los distintos modelos arquitectónicos. En cambio, nuestro enfoque, se basa en un modelo de versionamiento que representa a los elementos de la arquitectura en dos niveles: repositorio y versiones, manteniendo así solo el objeto genérico y todas las versiones que de él se tenga a lo largo del proceso de evolución.

Ivkovic and Kontogiannis (2006) propusieron un framework para el refactoring de arquitecturas de software usando transformaciones de modelos y anotaciones semánticas relacionadas con mejoras de calidad. El contexto de refactoring es descripto utilizando perfiles UML. Además, emplean anotaciones semánticas que se enlazan a las transformaciones aplicadas con el fin de indicar sus efectos sobre objetivos y las métricas que son aplicables. Este framework está enfocado en la aplicación de las transformaciones orientadas al logro de determinados objetivos para mejorar la calidad de la arquitectura de software. Sin embargo, no provee ningún soporte para conservar la historia del proceso de refactorización de la arquitectura de software, no guarda registro de cómo ésta evoluciona, lo único que se conserva es el artefacto final alcanzado. De esta manera, se hace imposible la realización de un análisis posterior al diseño que permita descubrir de qué manera el diseño original cambió, y cuáles fueron los elementos arquitectónicos impactados.

Ahmad et al. (2013) han propuesto PatEvol, un framework conceptual para la adquisición y la aplicación de conocimiento relativo a la evolución de arquitecturas de software, con el objetivo de atacar sistemáticamente los cambios frecuentes en arquitecturas de software. Dicho framework aspira a unificar los conceptos de minería en repositorios de software (para la extracción de patrones) y evolución de software, a fin de posibilitar la adquisición y la aplicación de conocimiento de evolución arquitectónica. Propone realizar la adquisión de conocimiento a mediante el análisis post-mortem de historias de evolución registradas en logs de cambios. Los cambios conservados en el log son convertidos en grafos que capturan las operaciones de cambio sobre elementos arquitectónicos, y luego sobre éstos se aplican técnicas de minería para iden-

tificar patrones de evolución. Por otro lado, dicho framework pretende la aplicación de este conocimiento descubierto, haciendo posible el reuso del mismo durante la evolución de la arquitectura. Sin embargo, se advierte que los resultados reportados aún son preliminares, y se requiere mayor desarrollo de las propuestas. El enfoque de dicho trabajo tiene algunas similitudes con el que presentamos en este trabajo, pero difiere en que nuestra propuesta los cambios aplicados sobre un modelo arquitectónico se capturan directamente en un repositorio basado en tecnología orientada a objetos, lo que ahorra la tarea de trabajar sobre logs de cambios que pueden tener otras representaciones que requieren procesamiento adicional. Además nuestra propuesta, se basa en un modelo para la captura y representación de procesos de diseño, que ha sido aplicado en otros dominios y tiene un desarrollo bastante avanzado, así como una herramienta que permite la validación de las ideas aquí presentadas.

EVOLUCIÓN DE ARQUITECTURAS DE SOFTWARE

Diversas son las causas de los cambios sobre las arquitecturas de software. A continuación enumeraremos algunas, con el objetivo de tener una visión amplia de los distintos dominios de aplicación de la propuesta, y por ende la variedad de operaciones de evolución y refactoring con que un arquitecto debería contar:

1) Redefinición de requerimientos existentes o aparición de nuevos requerimientos. Cualquier nuevo requerimiento implica una nueva funcionalidad o calidad que la arquitectura debe satisfacer. La redefinición de requerimientos puede significar la eliminación de componentes o responsabilidades de componentes que ya no se necesitan, y el agregado de otras nuevas.

2) Necesidad de cambios en la infraestructura o tecnología. Un ejemplo de este tipo de cambio se ha dado en los últimos años con la migración de sistemas tradicionales a un entorno cloud. Las organizaciones buscan minimizar la inversión en infraestructura propia y además adquirir nuevas soluciones informáticas que se adapten rápidamente a los cambios dinámicos del entorno. De esta manera, las soluciones contratadas van desde almacenamiento a servicios de computación, lo que implica un cambio significativo en la arquitectura de software existente.

3) Erosión en el diseño. Muchas veces los problemas de calidad en una arquitectura de software, son consecuencia de la "erosión arquitectónica" producida por los cambios que se han aplicado a la arquitectura en forma caótica o poco sistematizada, sin contar con la documentación adecuada. Este fenómeno hace necesario aplicar nuevos cambios para mejorar aspectos de calidad de la arquitectura de software sin afectar el comportamiento externo del sistema, lo cual se denomina refactoring. En este sentido, el cambio busca evolucionar la arquitectura de software, reorganizando sus elementos en nuevas capas o components lógicos, en vistas a incrementar la calidad de la arquitectura en su totalidad, y superar limitaciones arquitectónicas que atacan ciertos atributos de calidad como por ejemplo escalabilidad, extensibilidad, capacidad de evolución, capacidad de prueba, etc.

4) Nueva iteración en el proceso de desarrollo del sistema. Si bien el proceso de refactoring es aplicable a todo tipo de modelo del proceso de desarrollo, es particularmente útil en el contexto de desarrollo ágil. Por ejemplo, en un proceso Srum, una restructuración de la arquitectura puede ser integrada mediante la inclusión de actividades de refactoring en los sprints o iteraciones. Es recomendable no hacerlo

con la misma frecuencia que el refactoring de código, ni tampoco realizarlo inmediatamente luego de un release (Stal, 2014).

5) Aparición de errores, debidos a malas decisiones de diseño arquitectónico.

La documentación de los cambios que provocan la evolución de una arquitectura de software, así como de sus causas, contribuye a que en una nueva evolución, los diseñadores partan de la comprensión de las transformaciones previas que ya se han aplicado, disminuyendo también así la aparición de erosión arquitectónica. Para que esta documentación realmente se realice es imprescindible contar con herramientas que posibiliten la captura efectiva del proceso de evolución y refactoring de la arquitectura de software.

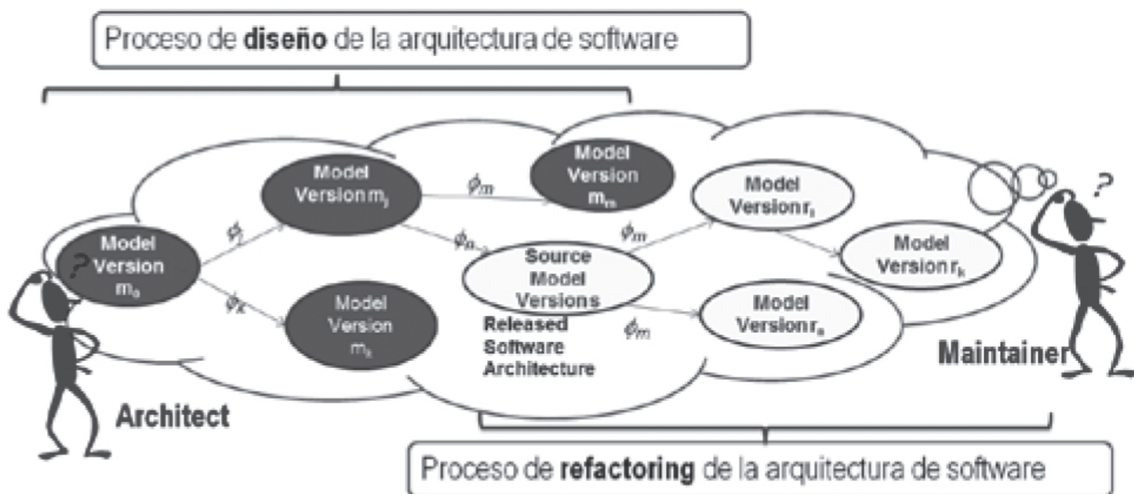

Fig. 1 - Enfoque del modelo para representación del proceso de refactoring arquitectónico

ENFOQUE EMPLEADO

El modelo propuesto adopta un enfoque operacional, en donde las decisiones de refactoring son materializadas como la ejecución de secuencias de operaciones arquitectónicas, las cuales se aplican comenzando por la versión de modelo de la arquitectura del sistema existente, a la cual llamamos versión de modelo origen (Fig. 1). Se parte de la premisa que de la arquitectura de software sobre la que tiene lugar la evolución, fue generada mediante el mismo enfoque (Roldán et al., 2010), aunque no es imprescindible.

Así, durante el proceso de evolución, un proyecto existente es “extendido” , lo que implica: a) reabrir un proyecto existente; b) indicar a partir de qué versión de modelo de arquitectura de software vamos a continuar el proceso (la cual deberá ser un nodo hoja del árbol de versiones que se exploró); c) ampliar el dominio con el que se trabajó (si fuera necesario) con operaciones de evolución/refactoring apropiadas; d) avanzar en el proceso, aplicando operaciones de evolución/ refactoring, las cuales materializan las nuevas decisiones de diseño que toma el arquitecto responsable del mantenimiento (“maintainer”); e) realizar evaluación de la arquitectura (mediante herramientas y técnicas que se puedan aplicar); f) seleccionar la arquitectura final de la evolución.

El modelo subyacente posibilita la captura de las operaciones aplicadas junto a los resultados

(productos del diseño), así como también indicar cuál es el objetivo perseguido con cada transformación aplicada sobre la arquitectura.

MODELO

A continuación se introduce el modelo propuesto, el cual ha sido también empleado en el proceso de diseño de arquitecturas de software (Roldán et al., 2010). Dicho modelo, considera a un proceso de diseño como una secuencia de actividades que operan sobre los productos del proceso, llamados objetos de diseño (DesignObject en Fig. 2). Los objetos de diseño son representaciones de los artefactos diseñados (por ejemplo, componentes y conectores) y las especificaciones a ser satisfechas (por ejemplo, requerimientos de calidad como disponibilidad o performance). Estos objetos evolucionan a medida que se introducen cambios en la arquitectura de software, dando lugar a las distintas versiones que deben ser mantenidas.

Los objetos de diseño son representados en dos niveles, el nivel de repositorio y el nivel de versiones. El nivel de repositorio mantiene una única entidad para cada objeto de diseño que ha sido creado y/o modificado durante un proyecto de diseño, o durante la evolución o refactoring de una arquitectura de software existente. Este objeto es llamado objeto versionable (VersionableObject en Fig. 2).

Además, en el repositorio se representan las relaciones entre los distintos objetos versionables (Association en Fig. 2). Por otro lado, el nivel de versiones mantiene las diferentes versiones de cada objeto de diseño. Éstas son llamadas versiones de objeto (ObjectVersion en Fig. 2). La relación entre un objeto versionable y sus versiones de objeto es representada a través de la asociación Version. Por lo tanto, para un objeto de diseño dado se mantiene una única instancia en el repositorio y todas sus versiones se alojan en el nivel de versiones.

En un momento determinado durante la evolución de una arquitectura de software, los estados asumidos por el conjunto de objetos de diseño relevantes proveen una fotografía del estado del proceso, el cual se denomina versión de modelo (ModelVersion en Fig. 2). Una nueva versión de modelo es generada aplicando una secuencia de operaciones (Ø) sobre una versión de modelo predecesora. Por consiguiente, el esquema de representación de versiones de modelo tiene una estructura de árbol, dónde cada versión de modelo es un nodo y la raíz es una versión de modelo inicial. Debido a que la evolución de modelo es planteada como una historia formada de situaciones discretas, en trabajos previos (Roldán et al., 2013) se adoptó el cálculo de situaciones para modelar formalmente el proceso de generación de versiones. Allí se define el predicado belong(v,m) usando el formato de axiomas de estado sucesor, el cual permite conocer las versiones de objeto (v) que pertenecen a una versión de modelo (m). Esto permite reconstruir la versión de modelo mi+1 aplicando todas las secuencias de operaciones desde la versión de modelo inicial m0. Las operaciones primitivas que han sido propuestas en tal esquema de versionado para representar las transformaciones de las versiones de modelo son add, delete y modify. La operación add(v) incorpora una versión de objeto v que no existía en una versión de modelo previa. En cambio, la operación delete(v) elimina, en la versión de modelo actual, una versión de objeto v que existía en la versión de modelo previa. Además, si un objeto de diseño tiene una versión vp, la operación modify(vp,vs) crea una nueva versión, vs.Cada operación aplicada a una versión de modelo es capturada a través de la clase VersionHistory (ver Fig. 2), la cual permite realizar el seguimiento de la evolución del modelo manteniendo referencias entre las versiones de objeto sobre las que se aplica una operación (Operation en Fig. 2) y aquellas que

surgen como resultado de su ejecución. Además, todas aquellas instancias de VersionHistory que consisten en la captura de una operación individual son agregadas en una instancia de ModelHistory para representar la secuencia de operaciones que causa la evolución.

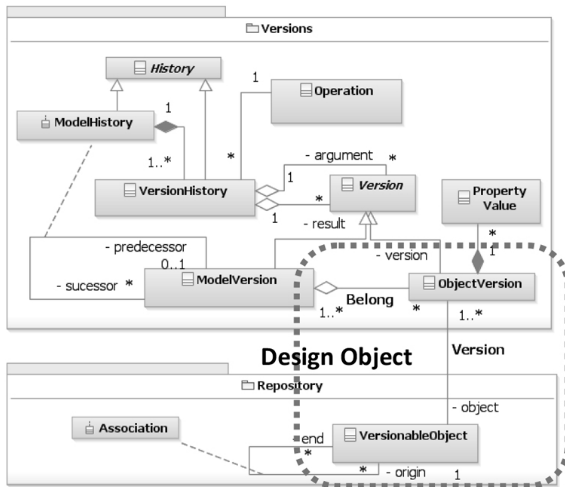

Fig. 2 - Esquema de versionamiento para representar objetos de diseño y capturar su evolución.

En la Fig. 3 se ilustra con un ejemplo simple de qué manera el esquema de versionamiento posibilita la captura de la evolución de una arquitectura de dos capas a una arquitectura de tres capas. En ella se observa que se parte de la versión de modelo i que contiene a la arquitectura de software de partida de la evolución. En esta versión de modelo se cuenta con objetos que representan al sistema y los dos componentes que representan a las capas lógicas del mismo. A continuación se decide avanzar en el refactoring de esa arquitectura, para lo cual se ejecuta la secuencia de operaciones θi que da lugar a una nueva versión de modelo i+1, en la cual aparece una nueva versión de sistema, Sistema-v2. Ese objeto sistema mantiene las asociaciones con otros objetos (componentes) que este objeto ya poseía en la versión de modelo predecesora. Una nueva decisión del diseñador/maintainer de la arquitectura es adicionar una nueva capa en la arquitectura. Esa decisión se materializa en la aplicación de la secuencia de operaciones θi+1, la cual da lugar a la generación de la versión de modelo i+2 que contiene al nuevo componente lógico que representa a la capa adicionada.

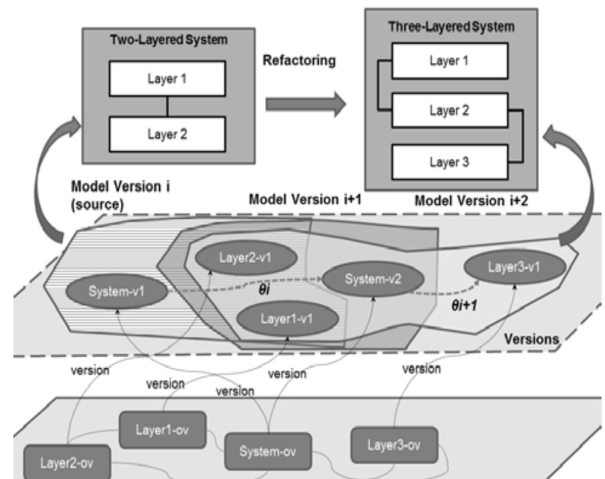

Fig. 3 - Representación de objetos de diseño y captura de la evolución de una arquitectura de software.

El modelo orientado a objetos propuesto provee el paquete Domain (Fig. 4, lado izquierdo), el cual permite la definición de diferentes dominios de arquitecturas de software. Un dominio abarca un conjunto de conceptos específicos al contexto del sistema existente que se desea evolucionar, los cuales constituyen los diferentes tipos de objetos de diseño de los cuales se desea conservar su evolución (DesignObjectType), y los posibles tipos de relaciones que pueden existir entre ellos (DomainRelationship). Los tipos de objetos de diseño pueden ser abstractos (AbstractDesignObjectType) o concretos (ConcreteDesignObjectType), con el objetivo de establecer relaciones de generalización/especialización entre conceptos del dominio. Además, para un tipo de objeto de diseño se definen las propiedades que lo caracterizan (Property) y las operaciones que son aplicables sobre objetos de

tal tipo (Operation). Un dominio específico surge como instanciación de las clases mencionadas. En la Fig. 4 (derecha) se presenta un posible modelo de dominio de arquitecturas de software, el cual incluye conceptos que un arquitecto podría utilizar. Un primer conjunto de tipos de objetos de diseño que constituye el dominio propuesto en este trabajo surge del lenguaje de descripción de arquitecturas empleado, por ejemplo ACME o UML2. En este dominio particular se incluyen conceptos de dos vistas o viewpoints (ISO, 2011): la vista de componentes y conectores (C&C), y la vista de despliegue. La vista C&C principalmente se enfoca en la división de responsabilidades y comportamiento de los componentes de la arquitectura, y en cómo estos se relacionan. Por ejemplo, un arquitecto experto puede definir en el dominio los siguientes tipos de RHasResponsibility que permite enlazar Component y Responsibility y representar el traspaso o delegación de responsabilidades de un componente a otro. RHasResponsibility tiene propiedades como assignedBy para representar quién fue el actor responsable de la acción.

Consideramos ahora tipos de objetos de diseño relativos a la vista de Despliegue (Allocation), la cual es útil para describir cómo los elementos de una arquitectura de software interactúan con otros elementos que no son software, o cómo están lógicamente contenidos, en el ambiente en donde el sistema va a ejecutarse (Clements et al., 2010). En este caso se definen los tipos de objetos de diseño ProcessingNode, Network, CommunicationChannel and NetworkDevice (Fig. 4).

A esta vista también pertenecen otros tipos de

Fig. 4 - Definición de un dominio con elementos de las vistas de C&C y Despliegue.

objetos de diseño Component, Connector, Port, Role and Attachment (Fig. 4, derecha), los cuales son tipos de objetos de diseño derivados del lenguaje ACME (Garlan et al., 2000). Adicionalmente, Characteristic and Responsibility design object types se pueden incluir en este dominio para representar propiedades y comportamiento relevantes de los components y conectores. Por cuestiones prácticas en este dominio, se incluye el tipo de objeto de diseño objetos de diseño ya incluidos en este dominio, como Component y Connector. RAllocated-to permite expresar que de qué manera un elemento de software reside sobre otro elemento del entorno, el cual puede ser la unidad física sobre la que reside. Cabe aclarar, que RAllocated-to es una relación de dominio reificada, es decir es un tipo de relación de dominio que se ha transformado en un tipo de objeto de diseño a fin de aumentar su semántica.

Un componente puede ser asignado a un mismo componente físico o virtual durante todo su ciclo de vida (RFixedAllocated-to), o puede ir cambiando el componente sobre el que se aloja dinámicamente durante la vida del sistema. En este caso dependiendo de la naturaleza del tipo de alojamiento dinámico, se proponen en el dominio distintos tipos de objetos de diseño: RMigrates-to, RCopy-Migrates-to, and RExecution-Migrates-to. La semántica de estas relaciones reificadas es la brindada por Clements et al. (2010). Otra relación reificada incluida en el dominio es RChannelledBy, que posibilita al diseñador dejar explícito en un diseño arquitectónico cuál es el conector físico a través del cual una comunicación lógica tiene lugar. Dependiendo de la necesidad de expresividad en el dominio, los tipos de objetos de diseño pueden ser especializados en elementos más concretos. Por ejemplo un componente puede ser especializado en base de datos, componente de sincronización de datos, componente gateway, virtual machine, etc. (no se presentan en la Fig. 4 por limitaciones de espacio).

Si bien no se detalla en este trabajo por cuestiones de espacio, en un dominio es posible incluir tipos de objetos de diseño que sean relativos a conocimiento de contexto y de razonamiento ("rationale") de la arquitectura, que es deseable documentar o capturar durante el proceso de evolución. Posibles tipos de objeto de diseño a incorporar son FunctionalRequirement y QualityRequirement, QualityScenarios¸ y Constraints. Modelos de dominio conceptuales que incluyen estos tipos de objetos de diseño han sido propuestos en trabajos anteriores.

EL MODELO DE OPERACIONES

Las operaciones primitivas add, delete y modify básicas para la transformación de versiones de modelo, no son suficientes para capturar la semántica de los patrones de evolución y refactoring, por lo que deben ser extendidas con operaciones de mayor complejidad, que permitan además manipular los tipos de objetos de diseño específicos del dominio. Se propone a continuación un modelo de operaciones que sienta la base conceptual para el desarrollo de herramientas computacionales flexibles que permitan especificar y ejecutar operaciones de evolución. En la Fig. 5 se observa que una operación (Operation) es definida como un macro comando (MacroCommand), subclase de Command, que simplemente ejecuta una secuencia de comandos. Para un macro comando debe definirse su cuerpo (expresado mediante la relación de composición entre MacroCommand y Command). Los comandos del cuerpo pueden ser primitivas (tales como add, delete o modify), o funciones auxiliares. Estas últimas son comandos predefinidos en el modelo para iteración, asignación de variables, asignan asociaciones entre objetos del repositorio, etc. Cada comando tiene uno o más argumentos tipados (Argument en Fig. 5), los cuales pueden ser considerados cómo variables. Una variable (Variable en Fig. 5) tiene un tipo y puede ser declarada y usada en el cuerpo de una operación. La clase DataType generaliza los tipos disponibles: PrimitiveDataType, Collection, DesignObjectType y DomainRelationship.

La definición de operaciones en un dominio se realiza como instanciación de las clases del modelo de operaciones. Para su especificación en forma textual, se ha definido una gramática libre de contexto que define cada parte de una operación: su nombre, argumentos y tipos, los comandos que conforman su cuerpo y los resultados.

OPERACIONES

Como se ha introducido previamente, cuando un maintainer va a iniciar un proceso de evolución arquitectónica, parte de una arquitectura inicial.

Esta arquitectura ha sido obtenida mediante un proceso de diseño en el que el diseñador trabajó empleado los elementos definidos en un dominio dado.

Para proceder con la evolución de la arquitectura, puede ser necesario extender ese dominio, con nuevos tipos de objetos de diseño específicos y operaciones de evolución y refactoring que no habían sido consideradas previamente. A continuación se mencionarán y detallarán algunas operaciones que podrían incluirse en el dominio que se ha propuesto en la sección anterior.

se detallas algunas operaciones, y se presenta en la Fig. 6 su correspondiente especificación funcional.

- EvolveSourceSystem. Se aplica sobre la versión de sistema que representa la arquitectura de software actual, a partir de la cual tendrá lugar la evolución (argumento s). Además, se debe proveer el argumento evolutionDescription que es la descripción de la causa de la evolución, el nombre para la nueva versión de sistema. Como resultado de esta operación se obtiene una nueva versión de

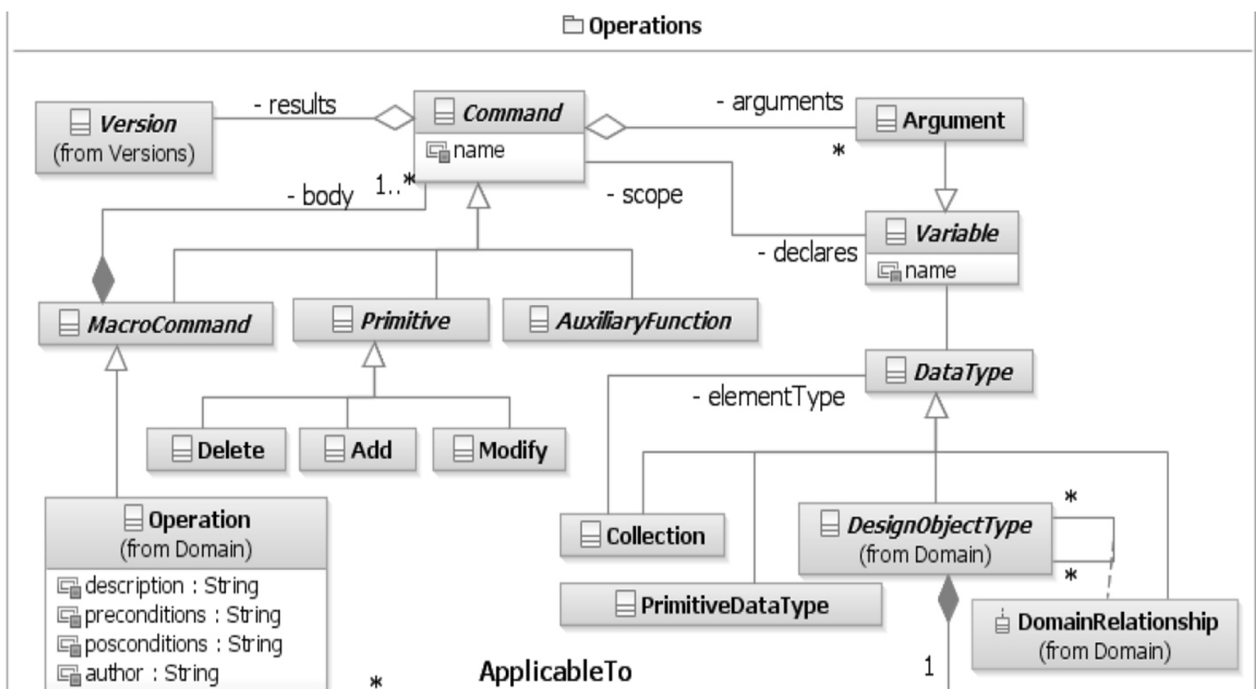

Fig. 5 - Modelo para la definición de operaciones de evolución y refactoring.

OPERACIONES DE INICIO DEL PROCESO DE EVOLUCIÓN

Las operaciones de inicialización se encargan de cuestiones relativas a la gestión del proceso de evolución arquitectónica que se va a llevar a cabo, como por ejemplo indicar cuál será la arquitectura de "partida", o fijar el contexto del proceso definiendo los requerimientos que se desea alcanzar en la próxima evolución arquitectónica. A continuación

modelo, en donde se tiene una nueva versión de sistema que mantiene todas las relaciones existentes con otros elementos de la arquitectura. Sobre esta versión de sistema además se fijan valores para propiedades fecha y autor en forma automática.

- AddNewQualityRequirement. Es similar a la operación addQualityRequirement (Roldán et al., 2010). Se lo propone como nueva opera-

ción, a fin de agregar la semántica de que se trata de un requerimiento que aparece durante la evolución de la arquitectura.

- AddNewFunctionalRequirement. Idem anterior para requerimientos funcionales.
- DeleteOldQualityRequirement. Elimina un requerimiento de calidad que ya no es necesario satisfacer.

```
evolveSourceSystem(s: System, newName: String,
            evolutionDescription: String)
  modify(s, [newName, evolutionDescription]).
End

addNewQualityRequirement(s: System, rn: String,
                  [description: String, priority: Integer])
  addQualityRequirement(s, rn, [description, priority])
end

addQualityRequirement(s: System, rn: String,
                  [description: String, priority: Integer])
 r := add(rn, QualityRequirement,
           [description, priority, 'NoSatisfied'])
 rs := add(null, RReqSist, ['NoEvaluated'])
 addAssociation(rs, s, RRSR)
 addAssociation (rs, r, RRSEA)
end

deleteOldQualityRequirement                    (qr:
  [QualityRequirement])
  lsce := get(QualityScenario, qr)
  for each s en lsce
    deleteQualityScenario(s)
  end for
  delete(qr)
end
```

Fig. 6 - Ejemplos de operaciones de inicio del proceso de evolución.

OPERACIONES DE EVOLUCIÓN/REFACTORING

Contando con el modelo de operaciones, los arquitectos y diseñadores expertos, pueden codificar conocimiento acerca de patrones de refactoring, o cualquier operación de evolución arquitectónica. Este conocimiento puede provenir de sus propias experiencias o provenir de diversas fuentes, como literatura existente sobre soluciones a problemas recurrentes (Ahmad et al., 2013; Barnes et al., 2007; Côté et al., 2007; Ivkovic and Kontogiannis, 2006; Stal, 2014), o bien de prácticas bien conocidas en la industria (Homer et al., 2014). En los últimos años, los patrones de cambio arquitectónicos o de refactoring y los estilos de evolución han probado ser exitosos en promover reuso de expertise para atacar la problemática de la evolución de arquitecturas de software.

Presentamos a continuación algunos ejemplos de operaciones de evolución y refactoring arquitectónico. La cantidad y el tipo de operaciones refactoring dependerá del dominio definido, y por consiguiente de los tipos de objetos de diseño incluidos. Aquí se presentan algunos ejemplos representativos, introduciendo una breve explicación de la operación, y su especificación funcional. Dado que algunas operaciones son extraídas de catálogos de patrones existentes, mantendremos el nombre original de la operación en inglés (Homer et al., 2014; Ivkovic and Kontogiannis, 2006; Stal, 2015):

- Rename Entities. El objetivo es introducir nombres intuitivos para las entidades, de manera de que sea fácil entender el rol de cada entidad en la arquitectura, y ganar expresividad.
- Add New Component/ Add New Connector. El nombre es autoexplicativo. Se define una operación con este nombre especial para indicar que se trata de un componente agregado durante el proceso de evolución o refactoring.
- Add Cloud Provider Network. Agrega un subtipo de Network, que se representa el conjunto de servicios ya sea de plataforma, infraestructura o software contratado con un proveedor.
- Add Virtual Machine. Agrega un tipo de componente especial que representa a un servidor virtual.
- Split Subsystems. Como el nombre lo indica, se busca dividir un componente o subsistema en componentes separados, como por ejemplo una base de datos que se encuentra acoplada e internamente contenida por el componente padre. Es necesario aplicar esta operación de refactoring cuando las partes de un sistema tienen un bajo acoplamiento, cuando se nece-

sita mejorar la modularización de un sistema, para comprenderlo mejor, o cuando un componente posee demasiadas responsabilidades. Patrones más específicos son Client-Server y Three Layers.

- Consolidate Duplicated Components/Connectors. Tiene como objetivo unificar varios componentes o conectores que estén duplicados en uno solo, con el objetivo de disminuir la complejidad y tamaño del sistema.
- Break Dependency Cycles. Un subsistema A podría depender tanto directa como indirectamente de un subistema B (es decir, A podría depender de un subistema C, el cual a su vez dependa de B). Los ciclos de dependencias conducen a sistemas menos mantenibles, menos fáciles de cambiar, menos reusables, probables y fáciles de comprender.
- Insert Transparency Layer / Add Wrapper. La capa de trasparencia es una capa de indirección que puede estar integrada al subsistema, como una máquina virtual o ser una entidad independiente como un Adapter o Facade que reenvía las invocaciones al subsistema. En ambos casos los componentes que acceden no saben nada acerca de los componentes que brindan los servicios. Una forma de implementar la capa de transparencia es a través de wrappers que reenvían las invocaciones a los componentes ocultos del subsistema y realizan algunas acciones antes y después de las invocaciones.
- Add New Responsibility. Simplemente consiste en agregar nuevas responsabilidades sobre componentes ya existentes.
- Delegate Responsibility. Consiste en mover una responsabilidad que poseía un componente a otro componente.
- Extract Responsibility / Extract Interface. Similar a Delegate Responsibility, pero en lugar de moverse la responsabilidad, ésta se "exporta". Para ello se emplea una interfaz abstracta como un bridge o un decorator para separar interfaz de implementación.
- Load balancing. Incorpora un componente que tiene la responsabilidad de distribuir la carga de trabajo de otros componentes.
- Remove Duplicates. Esta operación apunta a mejorar un diseño en donde existen componentes con las mismas responsabilidades.
- Move Entities. Con esta operación se busca mover componentes de un subsistema a otro. Aplicable cuando un componente tiene alto acoplamiento con otro que no se encuentra contenido en el mismo subsistema o componente contenedor, es decir, se ajusta mejor semánticamente a otro subsistema. Implica eliminar conectores existentes, crear otros nuevos entre los puertos que estén involucrados en la nueva configuración.
- ReallocateComponent. Similar a la anterior en cuanto a la configuración de elementos arquitectónicos que puede obtenerse, pero se aplica específicamente para describir el re-despliegue de un componente lógico en un nuevo componente (que puede ser lógico o físico). Como argumento opcional se puede proveer una descripción de qué interfaces o responsabilidades es necesario adecuar. Por esa razón, se crea una nueva versión del componente que ha sido reubicado.
- Map Port, se emplea cuando por ejemplo se hace el despliegue de un componente lógico sobre uno físico, y hay que remapear los puertos.
- Replicate Database. Esta operación se aplica cuando es necesario replicar una base de datos, o una partición de ella, por cuestiones de rendimiento, seguridad o disponibilidad/redundancia. Los argumentos de esta operación son el componente base de datos a replicar, el nombre de la base de datos nueva, el componentes

adonde se alojará la base de datos replicada y los scripts para generar la partición o esquema de base de datos a replicar.

- Synchronize Database. Esta operación se aplica cuando existen (particiones de) bases de datos replicadas que requieren mantenerse sincronizadas con la base de dato maestra. Consiste en la creación de un componente o servicio dedicado a esta tarea. Sus argumentos son el nombre del nuevo componente, sus responsabilidades, y las bases de datos que sincroniza. Puede ser necesario indicar un componente lógico gateway sobre el cual se canalizan las solicitudes de actualización, ya que las bases de datos posiblemente se encuentran desplegadas sobre redes diferentes.

En la Fig. 7 se presentan las especificaciones funcionales para algunas de las operaciones mencionadas. Además se incluyen operaciones arquitectónicas que ya han sido definidas previamente en el dominio (Roldán et al., 2010), y que se usan en la definición de otras operaciones (addComponent y addConnector).

Cabe aclarar que aquí presentamos un posible conjunto de operaciones de evolución y refactoring, pero en un dominio pueden definirse todas aquellas que sean necesarias.

Por otro lado, el modelo de operaciones brinda la posibilidad de definir operaciones orientadas a objetivo (oo). Estas operaciones son similares a las presentadas, pero incorporan un argumento adicional, el cual puede ser el requerimiento de calidad que se espera satisfacer con la ejecución de la operación. Al ejecutarse la operación además de capturarse la operación junto con sus resultados, se generan asociaciones especiales entre los objetos agregados/modificados y el objetivo, de manera de representar la semántica de la "probable satisfacción del objetivo perseguido". Por ejemplo, si la operación replicateDatabase persigue el objetivo de performance se podría aplicar en su lugar una operación alternativa replicateDatabase-oo(arg1, arg2, …, goal), en donde el valor para el argumento goal sea el requerimiento de calidad PerformanceV1 que se ha elicitado para la arquitectura nueva.

ESCENARIO DE APLICACIÓN

Para ilustrar cómo se emplea el modelo, consideraremos un escenario, que ilustra un proceso de migración de una arquitectura de software de un sistema tradicional a una cloud híbrida.

```
addNewComponent(s: System, cn: String, lResps: Collection[String], lPorts: Collection[String],
lprops: Collection[PrimitiveDataType])
   addComponent(s, cn, lResps, lPorts)
end
--------------------------------------------------
addNewConnector(s: System, nc: String, lRoles: Collection[String],
lPorts: Collection[Port], lprops: Collection[PrimitiveDataType])
  addConnector(s, cn, lRoles, lPorts, lprops)
end
--------------------------------------------------
addComponent(s: System, nc: String, l_Resps: Collection[String],
l_Ports: Collection[String], l_props: Collection[PrimitiveDataType])
  c:= add(nc, Component, l_props)
  for each nr in l_Resps
    addResponsibility(c, nr, [ ])
  end for
  for each np in l_Ports
    addPort(c, np, [ ])
end for
 addAssociation(s, c, RSystemComponent)
end
--------------------------------------------------
addConnector(s: System, nc: String, lRoles: Collection[String],
l_Ports: Collection[Port], l_props: Collection[PrimitiveDataType])
  c :=add(nc, Connector, l_props)
  i := 0
  for each nr in l_Roles
    r:= addRole(c, nr, [])
    addAssociation(l_Ports(i), r, RComponentPort)
    i++
  end for
  addAssociation(s, c, RSystemConnector)
end
--------------------------------------------------
replicateDataBase(s: System, c:Database, rn:String,
h: Component, schemaScripts: String)
  dbr := addDatabase(rn, schemaScripts)
  setFixedAllocation(dbr, h)
end
--------------------------------------------------
synchronizeDataBases(s: System, nsync: String,
lresps: Collection[Responsibility] , lports:
Collection[Ports], mdb: Database, rdb: Database,
gatewayComponent: Component)
  sync := addNewComponent(s, nsync, lresps, lports)
  portsSync := get(Port, sync)
  portsMdb := get(Port, mdb)
  portsGwy := get(Port, gatewayComponent)
  np11 := select(portsSync)  // interacts with the user
  np21 := select(portsMdb)  // interacts with the user
```

Fig. 7 - Ejemplos de operaciones de evolución/refactoring.

Se mostrará cómo, partiendo de la versión de la arquitectura de software actual y a través de la aplicación de operaciones de evolución, se puede ir avanzando en el proceso de evolución de la arquitectura, mientras se capturan las decisiones tomadas y los productos generados/modificados durante el mismo.

Sports es una empresa dedicada a la venta mayorista de ropa deportiva, en los últimos años abrió su propio web store para venta minorista. Este nuevo modelo de negocio ha sido exitoso y sus proyecciones de ventas online van en ascenso, por lo que han surgidos nuevos requerimientos de escalabilidad y disponibilidad. Por tal motivo y con el objetivo de abaratar costos de infraestructura han decidido migrar parte de su sistema de ventas minoristas a la cloud. Por otro lado, la empresa además posee un sistema legacy para la gestión de pedidos y ventas mayoristas. Actualmente este sistema es operado por los empleados administrativos de la empresa. Particularmente, los pedidos realizados por los preventistas son comunicados via fax o telefónica, y cargados al sistema por los operadores. Se desea liberar a los operadores de esta carga de trabajo, por lo que se ha pensado incorporar un nivel de acceso especial para preventistas en el sitio web, para que ellos carguen sus propios pedidos. Para que sea posible interactuar con el sistema de carga de pedidos, se contará con un componente wrapper en la nube, que brindará los servicios de acceso al componente de Pedidos Mayoristas. Además, por cuestiones de seguridad, por el momento las bases de datos se mantenendrán bajo control de la empresa, por lo que se conservará el servidor de datos como parte de la infraestructura. Sin embargo, para lograr un mayor rendimiento cuando los clientes minoristas realizan compras se pretende contar en la cloud con un esquema de bases de datos replicado, que abarque solamente el catálogo y stock actual de productos de venta minorista. De esta manera, los datos sobre descripción de producto y la cantidad de unidades disponibles podrán obtenerse sin tener que recuperar dicha información de la base de datos que se encuentra en la intranet de la empresa. Por ese motivo, se contratarán servicio de hosting de bases de datos, que se actualizará con cierta frecuencia para reflejar el catalogo y stock actual.

SEQUENCIAS DE OPERACIONES APLICADAS

Se parte de una arquitectura de software inicial, dada por el sitio de e-commerce actual para ventas minoristas y los componentes legacy de la empresa para gestión de ventas mayorista (Fig. 8). Esta versión de modelo es la arquitectura de software "source" del proceso de evolución de la arquitectura. El maintainer que inicia el proceso ejecuta la siguiente secuencia de operaciones:

ϑ1={selectSourceSystem(SystemSportv1, 'SystemSportEv1', 'Hibride cloud migration…')}.

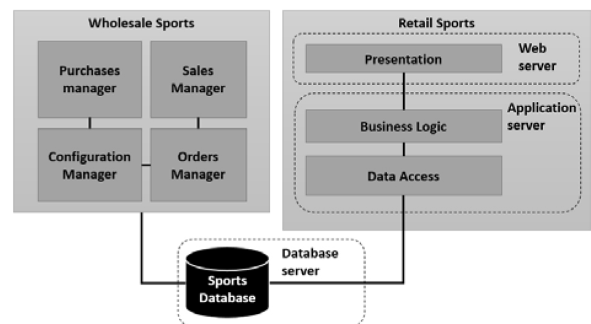

Fig. 8 -Arquitectura de partida del proceso de evolución

A continuación se identifican los requerimientos de calidad y funcionales que se desean alcanzar en el proceso de evolución a llevar a cabo:

ϑ2={addNewQualityRequirement (SystemSportEv1, 'Scalability'), addNewQualityRequirement (SystemSportEv1, 'Performance'), addNewQualityRequirement (SystemSportEv1, 'Availability')}).

A partir de estos requerimientos se toman las primeras decisiones de evolución y refactoring de la arquitectura de software tendientes a alcanzar

dichos requerimientos. Se comienza por decidir contratar servicios a un proveedor de servicios de cloud computing (Cloud Services Provider, CSP), en la modalidad IaaS, y de esta manera migrar los componentes que constituyen la aplicación del web store minorista. Para ello CSP provee una máquina virtual para servidor web sobre la cual se desplegará la capa de presentación del web store. Será necesario adecuar algunas interfaces a este nuevo entorno. Para la capa de lógica de negocios, y la capa de acceso a datos se procede de la misma manera, siendo reubicadas éstas en un componente que es una máquina virtual del servidor de aplicación. También se crea un servidor virtual para la base de datos que se encontrará en la cloud, para evitar la disminución en el rendimiento percibido por los clientes del webstore.

ϑ3={addCloudProviderNetwork(SystemSportv1, CSPNetwork')}

ϑ4={addVirtualMachine(CSPNetworkv1, 'VM-WebServer'),addVirtualMachine(CSPNetworkv1, 'VM-AppServer') , addVirtualMachine (CSPNetworkv1, 'VM-DBServer')}

ϑ5={realocateComponent(Presentationv1, VM-WebServerv1), realocateComponent (BusinessLogicv1, VM-AppServerv1), realocateComponent(-DataAccessv1, VM-AppServerv1)}

Para mantener conexión con la red privada virtual de la empresa con el CSP, se contrata un servicio VPN Gateway que permite la conexión segura a la infraestructura onpremises.

ϑ6={addNewComponent(CSPNetworkv1, 'VirtualNetworkGateway', [Resp-ConnectVPN],[PortVPNG1, PortVNPG2, PortVPNG3, PortVPNG4, PortVNG5]) }

A continuación, se procede a documentar todas las nuevas responsabilidades arquitectónicas que se deben incorporar al componente de presentación del sistema web, ya que se incorpora el acceso para preventistas para la carga de pedidos mayoristas.

ϑ7={addNewResponsibility(Presentationv2,), addNewResponsibility(Presentationv2,'Resp-Autentication'),'Resp-OrderFormPresentation'), addNewResponsibility(Presentationv2,'Resp-OrderFormDefaultValues'),addNewResponsibility(Presentationv2,'Resp-OrderFormValidation'), addNewResponsibility(Presentationv2, 'Resp-Submit')}

Se requiere de un componente wrapper que estará a cargo de encapsular los pedidos y proveer el acceso al módulo de Pedidos del sistema legacy de la empresa. Tener en cuenta que los pedidos mayoristas se continúan cargando en la base de datos maestra de Sport. Las solicitudes enviadas por el wrapper se encaminarán a través del VPN gateway.

ϑ8={addWrapper(CSPNetworkv1,'Orders-Wrapper,[Resp-ReceiveRequest,Resp-ForwardRequest],[PortOW1,PortOW2],Presentationv2, VPNGatewayv1)}

Esta operación incluye el establecimiento de conectores entre los componentes entre los que se inserta el wrapper como intermediario. Por lo tanto, no son explícitamente operaciones ejecutadas por el maintainer, aunque es necesario solicitarle interactivamente algunos valores de argumentos a fin de mapear puertos y roles de conectores apropiadamente. Si no se contara con la potencia del modelo de operaciones para definir operaciones complejas, sería necesario ejecutar un mayor número de operaciones. Por ejemplo:

ϑ8-1={addNewConnector(CSPNetworkv1, 'OW-VPNGateway', ['Role11', 'Role12'],[PortOW1v1, PortVPNG1v1]) }

ϑ8-1={addNewConnector(CSPNetworkv1, 'OW-Presentation', ['Role21', 'Role22'],[PortOW2v1, PortPresentation1v1]) }

Continuando con el escenario, para mejorar el rendimiento del sistema es deseable que el componente de acceso a datos del webstore minorista tenga un acceso rápido (sin latencia) al catálogo de productos a la venta, junto con los precios y

promociones activas. Para ello se requiere replicar el esquema de la base de datos que corresponde a dicha partición, el cual se despliega sobre el componente virtual VM-DBServer que ya ha sido creado. Además para mantener esta base de datos en la cloud actualizada, se crea un servicio de sincronización que funciona en un servidor de la empresa. Tal sincronización se realiza a través de la conexión de la VPN con el gateway.

ϑ9={replicateDataBase(DBSportsv1,'DBSports-Catalogue', 'Products catalogue and marketing schemes', VM-DBServerv1)}

ϑ10={addNewConnector(CSPNetworkv1, 'DataAccess-DBCatalogue',['Role31','Role32'],[PortDA2v2, PortDBC1v1]) }

ϑ11={sincronizeDatabases(SystemSportsEvv1, 'DBSynchronizer', [Resp-FindChanges, Resp-UpdateSchema],[PortSync1, PortSync2], DBSportsv1, DBSports-Cataloguev1, VPNGatewayv1)}

La ejecución de esta última operación también es bastante compleja, como se mostró en la Fig. 7, ya que abarca la creación y el establecimiento de tres conectores entre la base de datos maestra y el sincronizador, entre el sincronizador y el gateway, y entre el gateway y la base de datos reducida en la cloud.

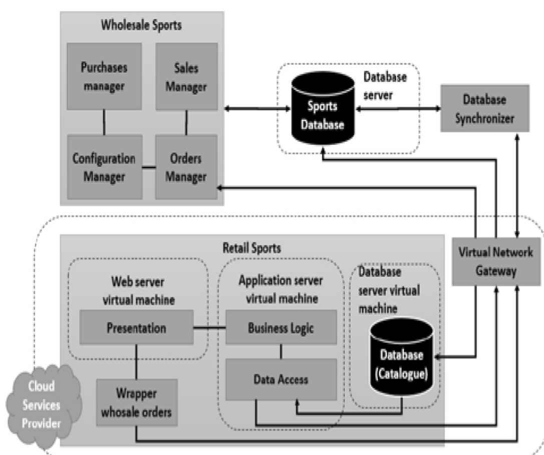

Fig. 9 - Arquitectura final.

Es necesario incorporar además una conexión entre el componente de acceso a datos del webstore con el servidor de la base de datos maestra, a través de la cual se pueden incorporar las ventas minoristas que han sido concretadas.

ϑ12={addNewConnector(CSPNetworkv1, 'DataAccess-VPNGateway',['Role31','Role32'],[PortDA2v1, PortVNPG3v1]) }

La arquitectura de software resultante (Fig. 9) puede continuar evolucionando creándose nuevas instancias de servidores web y de aplicación según demanda, e incorporando componentes dedicados para el balanceo de carga.

CONSULTAS / RECUPERACIÓN DE INFORMACIÓN

A partir del proceso de evolución representado mediante la captura de las operaciones ejecutadas y los productos generados, es posible responder a diferentes preguntas acerca de cómo se llevó a cabo dicho proceso:

i) ¿Cuáles fueron los elementos arquitectónicos afectados por la evolución / el refactoring?
ii) ¿Qué operaciones de evolución o patrones de refactoring se ejecutaron?
iii) ¿Qué requerimientos de calidad se cambiaron, se incorporaron, o se intentaron mejorar?
iv) ¿Qué nuevos productos surgieron?
v) ¿Qué componentes se mantuvieron pero sufrieron modificaciones?
vi) ¿Qué cantidad de operaciones de refactoring fue necesario aplicar?
vii) ¿Qué operación (que materializa una decisión de diseño arquitectónico) dio lugar a la aparición de determinados productos (componentes, responsabilidades, conexiones, etc.)?

CONCLUSIONES

El modelo propuesto posee una doble ventaja. Por un lado favorece y facilita la documentación del proceso de refactoring de arquitecturas de software, conservando la completa evolución a partir de una arquitectura inicial. El uso de esta documentación colabora en la compresión de los cambios que ya se han aplicado, de sus impactos en la arquitectura original. Por otro lado, permite sistematizar la aplicación de cambios en la arquitectura, ya que éstos se traducen en la aplicación de patrones de refactoring que han probado ser soluciones efectivas para problemas recurrentes.

REFERENCIAS

Ahmad, A., Jamshidi, P., and Pahl, C., "A framework for acquisition and application of software architecture evolution knowledge", 14. SIGSOFT Softw. Eng. Notes 38, 5 (August 2013), pp. 1-7.

Avgeriou, P., Kruchten, P., Lago, P., Grisham, P., and Perry, D., "Architectural knowledge and rationale: issues, trends, challenges", SIGSOFT Softw. Eng. Notes 32 (4), 2007, pp. 41–46.

Barnes, J., Garlan, D., and Schmerl, B., "Evolution Styles: Foundations and Models for Software Architecture Evolution", Journal of Software and Systems Modeling, Vol. 13(2), May 2014, pp. 649-678.

Breivold, H. P., Crnkovic, I., and Larsson, M., "A systematic review of software architecture evolution research", Information and Software Technology, 54, 1, January 2012, pp. 16-40.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J., Documenting Software Architectures: Views and Beyond (2nd. ed.), Addison-Wesley Professional, 2010.

Côté, I., Heisel, M., and Wentzlaff, I., "Pattern-Based Evolution of Software Architectures", in ECSA, 2007.

Falessi, D., Cantone, G.v and Kruchten, P., "Value-based design decision rationale documentation: principles and empirical feasibility study", in Proc. Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), Vancouver, BC, Canada, 2008, pp. 189–198.

Fowler, M., Beck, K., Brent, J., Opdyke, W., Roberts, D., Refactoring: improving the design of existing code, Addison-Wesley, 1993.

Garlan, D., Monroe, R. T., and Wile, D., "Acme: Architectural Description of Component-Based Systems", in Foundations of Component-Based Systems, G. T. Leavens and Sitaraman (eds.), Cambridge University Press, 2000, pp. 47–68.

Homer. A., Sharp, J., Brader, L., Narumoto, M., and Swanson, T., Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications. Microsoft patterns & practices, 2014.

ISO/IEC/IEEE, "Systems and software engineering - Architecture description," ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000), 2011.

Ivkovic I. and Kontogiannis, K., "A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations", in Proceedings of the Conference on Software Maintenance and Reengineering (CSMR '06). IEEE Computer Society, Washington, DC, USA, 2006, pp. 135-144.

Roldán, M. L., Gonnet, S., and Leone, H., "Knowledge representation of the software architecture design process based on situation calculus", Expert Systems, 30(1), 2013, pp. 34–53

Roldán, M. L., Gonnet, S., and H. Leone, "TracED: a tool for capturing and tracing engineering design processes", Advances in Engineering Software, 41(9), 2010, pp. 1087–1109.

Stal M. Architecture refactoring foundation incl. a refactoring pattern catalog: https://dl.dropbox.com/u/2228034/ArchitectureRefactoringCatalog.pdf, utimo acceso Mayo de 2015.

Stal, M., "Chapter 3 - Refactoring Software Architectures", in Ali Babar, M. Brown, A. W., and Mistrik, I. (eds.) Agile Software Architecture, Morgan Kaufmann, 2014, pp. 63-82.

Fecha de Presentación: 15/05/15
Fecha de Aprobación: 08/06/15