Plan de Trabajo Final

Carrera Ingeniería de Sistemas

Facultad de Ciencias Exactas – UNICEN

Tema: Enfoque Inteligente para la Sincronización de la Documentación Arquitectónica con su Implementación

Alumno/s: Sr. Emilio Adolfo Corengia

Director: Dr. Alvaro Soria

Codirector: Dr. Marcelo Armentano

1. Introducción

La *Arquitectura de Software* es un elemento de suma importancia en el proceso de diseño y desarrollo de sistemas que ha sido altamente adoptada en la industria. La visión arquitectónica del sistema es abstracta, deja de lado detalles de implementación, algoritmos o representación de la información y se enfoca en detalles estructurales y de interacción de los elementos de sistema que la componen. Su desarrollo se considera como el primer paso hacia el diseño de un sistema de software y permite capturar decisiones de diseño que servirán para satisfacer sus principales atributos de calidad o drivers como performance, modificabilidad y seguridad. Es un artefacto que permite, en una etapa temprana de desarrollo, llevar a cabo un análisis del diseño del sistema. Por otra parte, la arquitectura de un sistema es clave en etapas de post-desarrollo, sirviendo en su comprensión y mantenimiento, minimizando costos [1].

La Arquitectura de Software abarca tanto aspectos *estructurales* como *comportamentales* del sistema, y es comúnmente documentada a través de un conjunto de vistas. Una vista arquitectónica describe el conjunto de componentes del sistema y las relaciones asociadas a ellos, que a su vez, representan los intereses de uno o más *stakeholders*. Las vistas suelen ir acompañadas de especificaciones comportamentales en forma de escenarios que detallan los principales drivers arquitectónicos. La correcta documentación de la Arquitectura de Software será de suma importancia para el desarrollo y mantenimiento del sistema. Incluso una arquitectura perfecta resulta inútil si no es comprendida por nadie, o peor aún, si es mal interpretada [2].

Una vez que la arquitectura fue materializada [3], esta pasa a una etapa de mantenimiento donde el sistema evoluciona. Esta evolución va provocando que la documentación de la arquitectura pierda representatividad sobre su implementación. Algunos ejemplos son: nuevos requerimientos que provocan una restructuración del sistema; rehúso de código y bibliotecas que alteran el diseño; o distintas tareas de mantenimiento que no siempre se apegan al diseño. Esto se debe, principalmente, a dos problemas arquitectónicos conocidos: la *erosión arquitectónica* y el *desvío arquitectónico* [5]. Se define erosión arquitectónica como las violaciones que se suceden a nivel arquitectónico a lo largo de su evolución. En cambio, el desvío se lo define como la incomprensión u omisión de la arquitectura derivando directamente en problemas de erosión arquitectónica. Como consecuencia, la documentación de la arquitectura debe acompañar adecuadamente la evolución del sistema para que siga cumpliendo su misión/objetivos de negocio y beneficios para la organización.

Por lo tanto, se debe comprobar periódicamente que la arquitectura se corresponda con su implementación para detectar y corregir cualquier diferencia que se pueda presentar. En la práctica, una herramienta que ayude con este tipo de verificaciones resultará de suma importancia a la hora de manejar este tipo de desincronización entre la arquitectura documentada y su correspondiente implementación.

2. Motivación

Una de las principales causas de desvío arquitectónico se debe a que la documentación arquitectónica va quedando, progresivamente, desactualizada. En caso de no tratar este problema adecuadamente, se produce una diferencia entre la arquitectura *documentada* y la *implementada* [1]. Lo que se conoce como desvío arquitectónico, que conduce directamente a problemas de erosión arquitectónica.

Un enfoque tradicional para verificar la representatividad de la Arquitectura de Software es mediante revisiones de código [6, 7]. Estas verificaciones dependen de mapeos entre elementos de la arquitectura (módulos, componentes, capas, responsabilidades, etc.) y elementos de implementación (paquetes, clases, métodos, etc.). Basándose en esos mapeos, es posible recorrer la implementación en busca de divergencias con respecto a la arquitectura documentada. El problema de este enfoque tradicional es que las verificaciones basadas en la implementación suelen consumir demasiado tiempo, además de ser propensas a error. Por esa razón, herramientas que simplifiquen estas verificaciones resultan vitales para que esta tarea se ponga realmente en práctica. Otras propuestas abordan este problema por medio del uso de ingeniería reversa o por medio de análisis estructurales [8-12], con el objetivo de reconstruir la arquitectura del sistema y asegurar manualmente su correspondencia.

Aunque estos enfoques resulten convenientes para resaltar divergencias y ausencias de relaciones entre la arquitectura y su implementación, no resultan igual de efectivos para detectar inconsistencias a nivel comportamental. Este tipo de divergencias resultan muy difíciles de encontrar si se analiza solo la estructura del mismo. Un buen ejemplo son los sistemas con alto nivel de concurrencia, donde una entidad publica un evento, normalmente a un componente *dispatcher* que se ejecuta en un *thread* diferente; y otras entidades interesadas reciben la señal para actuar y llevar a cabo sus tareas. Cuando estos eventos son asincrónicos, el receptor puede activarse dentro de un intervalo variable, posterior a la publicación del evento.

En general, comprobar que los escenarios documentados se mantengan representativos a su implementación sigue siendo una tarea desafiante. Por esta razón, enfoques que permitan sincronizar también los aspectos comportamentales del sistema serán de vital importancia para facilitar la tarea de mantener actualizada la arquitectura durante la evolución del mismo.

3. Objetivos

En particular, esta propuesta se basa en mantener sincronizada la implementación, a medida que evoluciona, con la representación de la arquitectura. La Figura 1 muestra una vista esquemática del enfoque propuesto. La arquitectura estará modelada de forma tal que será capaz de representar aspectos tanto estructurales como comportamentales del sistema. Un arquitecto estará a cargo del mapeo inicial entre los elementos de la arquitectura y los elementos de implementación. Por esta razón, la herramienta estará destinada a escenarios donde se cuente con una arquitectura base estable (documentada previamente por el arquitecto, versión 1 en la figura) que pueda ser utilizada para entrenar la herramienta. El enfoque consistirá en mantener actualizados estos mapeos durante el ciclo de vida del sistema. Durante esta evolución, los desarrolladores van progresivamente introduciendo cambios en el código. Al mismo tiempo que los mapeos arquitectónicos van quedando desactualizados. Se conoce como *delta source-code* al conjunto de cambios entre dos revisiones de código consecutivas (entre las versiones 1 y 2 de la figura por ejemplo). Estos deltas, suelen exponer pequeñas desviaciones de comportamiento entre elementos de la arquitectura y la implementación. Haciendo uso de la herramienta el revisor deberá poner a prueba los principales escenarios arquitectónicos regularmente para evaluar sucesivas revisiones (versión 2 en la figura por ejemplo), la herramienta entonces comparará los resultados con el comportamiento esperado en orden de contar con un *feedback* constante sobre el estado de los mapeos arquitectónicos; esta técnica se la conoce como simulación basada en la arquitectura [2].

Utilizando la información disponible entre dos revisiones de código consecutivas, los mapeos a código, y la distribución de probabilidades generada a partir de la versión anterior del sistema se hace posible detectar desviaciones entre dos revisiones de código utilizando técnicas de aprendizaje de máquina. Y de ser necesario, sugerir cambios al arquitecto que le sirvan para devolver la conformidad a la arquitectura en base a su implementación actual; o bien, que sirvan al arquitecto para informar al equipo de desarrollo que la implementación no se adecua a las especificaciones. En este sentido, la tesis se presentará como una herramienta capaz no solo de detectar desviaciones comportamentales entre distintas revisiones, sino que también será capaz de sugerir cambios en base a la información aprendida.

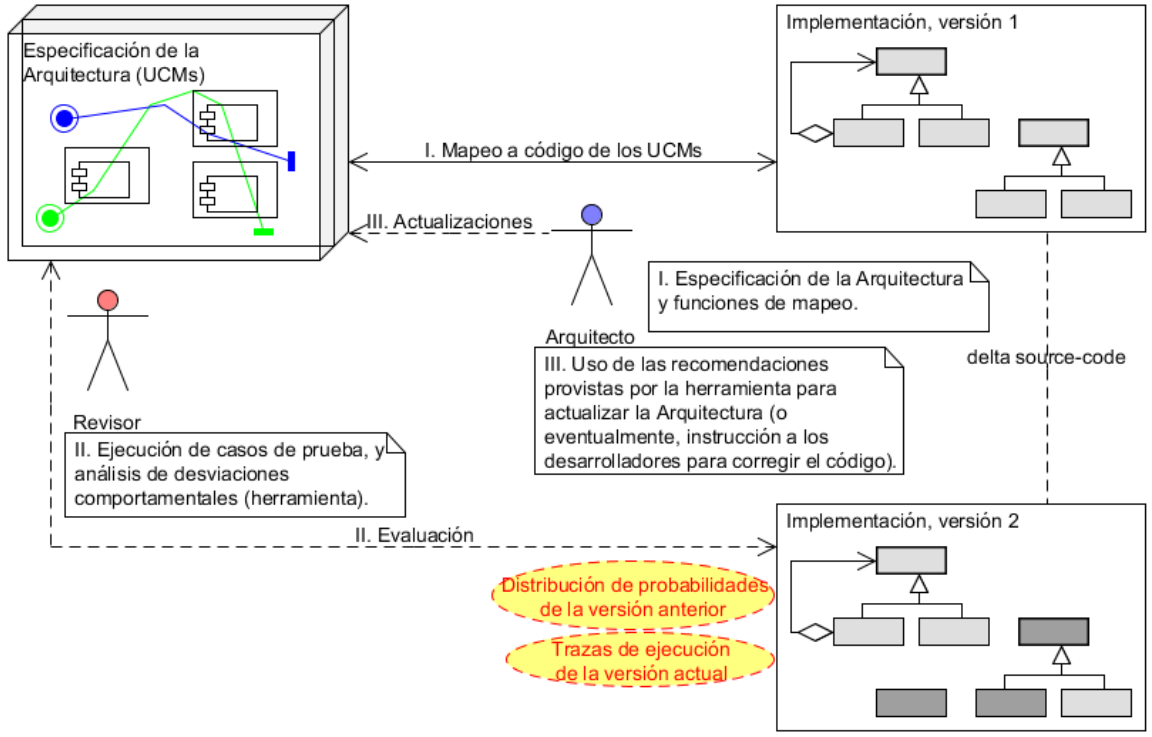

Figura 1: Proceso de sincronización arquitectónica de la herramienta

La arquitectura será representada por medio de la notación *Use Case Maps* (UCMs) [4]. Los UCMs son elementos de diseño de alto nivel, que permiten modelar tanto aspectos estructurales como comportamentales de una arquitectura de un sistema de software. Dan una visión de secuencias causa-efecto sobre el sistema de forma compacta debido a su alto nivel de abstracción, y se componen de tres elementos principales:

• *paths* o *recorridos* que diagraman estímulos a través de los componentes de un sistema;

• *responsabilidades* que unen paths a componentes y

• *componentes* que representan un conjunto de responsabilidades.

Para documentar la arquitectura esta propuesta utilizará *FLABot* [13], una herramienta para la depuración y edición de UCMs. FLABot no solo provee mecanismos de edición, sino que también, mapeos entre cada elemento de la notación (responsabilidades, componentes, etc) a nivel arquitectónico y su correspondiente porción de código a nivel de implementación. Utilizando estos mapeos y haciendo uso de técnicas de instrumentación de código, Flabot es capaz de capturar un conjunto de eventos de bajo nivel, que se conocen como *trazas de ejecución* [14], y relacionar estos eventos con los elementos arquitectónicos. Analizando esta

información, se puede detectar desvíos entre el código en ejecución y su documentación actual. De esta forma, la herramienta utilizará estos mapeos de UCMs para facilitar la tarea de mantener actualizados los escenarios arquitectónicos y la implementación para evitar los mencionados problemas de erosión.

Las trazas de ejecución provistas por FLABot representan la vista real de un sistema en ejecución, a un gran nivel de detalle y cobertura, dificultan la tarea de abstracción y análisis del sistema en desarrollo. Para facilitar esta tarea, se pueden analizar las trazas de ejecución como una secuencia compleja de unidades de ejecución (instrucciones máquina, sentencias, métodos, etc.); o bien como un conjunto de símbolos dentro de un lenguaje conocido en el ámbito del sistema. La tarea de encontrar divergencias entre los UCMs y las trazas de ejecución, se puede simplificar entonces a encontrar y evaluar asociaciones con alto grado de probabilidad entre los distintos símbolos dentro de una traza. De esta forma, es posible plantear el problema de encontrar relaciones causales como un problema símil al *data minning* [15]. Algunos ejemplos de esto último son la corrección de texto corrupto, o detección de relaciones dentro de una cadena de ADN. Normalmente, estos problemas se plantean como una secuencia de símbolos que pueden ser empleados para entrenar un modelo de distribución de probabilidades con el fin de encontrar relaciones entre posibles sub-cadenas. En este sentido, este trabajo empleará esta misma metodología para abstraer y analizar la documentación del sistema en desarrollo.

En particular, los modelos de *Markov* [16-18] resultan favorables para representar este tipo de información en forma de trazas de ejecución. Dado que la probabilidad condicional de estos modelos no cambia sustancialmente si los condicionamos a sub-secuencias precedentes de cierta longitud (sea variable o no). Por lo que los modelos de Markov prometen ser una técnica de aprendizaje conveniente para atacar el problema. De todas formas, no se va a perder generalidad limitando el trabajo a esta única representación, un atributo de calidad a seguir de la herramienta será su modificabilidad a fin de poder evaluar distintos modelos en posibles trabajos futuros sobre el tema.

El objetivo del trabajo propuesto será entonces, la construcción de una herramienta en formato de *plugin* en la plataforma *Eclipse* [19] capaz de adaptarse al ambiente de desarrollo. Presentará opciones de configuración de los algoritmos disponibles para el aprendizaje y evaluación de los UCMs. El usuario (arquitecto en la figura) podrá entrenar la herramienta con versiones estables del sistema, la herramienta (revisor en la figura) utilizará esta información para evaluar cambios de código con respecto a la documentación y sugerirá cambios en los UCMs para buscar restablecer la conformidad entre la arquitectura y su implementación. En particular, se pondrán en práctica modelos de Markov, de orden variable y estático, para representar la información.

4. Cronograma de actividades

A continuación se presenta un cronograma con las actividades propuestas. Su duración estimada es de seis meses.

Actividad	**Duración estimada**
Relevamiento bibliográfico *	2 semanas
Análisis de trabajos relacionados *	2 semanas
Recopilación y diseño de los requerimientos	2 semanas
Análisis y diseño de la solución	6 semanas
Implementación del prototipo	4 semanas
Elaboración/elección de ejemplos y puesta a prueba *	3 semanas
Documentación *	5 semanas

*Las actividades indicadas con * se podrán realizar en paralelo.*

5. Bibliografía

[1] L. Bass, P. Clement, and R. Kazman. Software Architecture in Practice. 2ed. Addison-Wesley, 2003.

[2] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, Addison Wesley, 2002.

[3] I. Jacobson, M. Christeron, and G. Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach.Addison-Wesley, 1992.

[4] R. J. A. Buhr and R. S. Casselman. Use Case Maps for Object-Oriented Systems. Prentice Hall, 1996.

[5] N. Medvidovic, A. Egyed, P. Gruenbacher, Stemming architectural erosion by coupling architectural discovery and recovery, USA, in: STRAW'03, Software Requirements to Architectures Workshop, held at ICSE'03, 2003.

[6] D.L. Parnas, D.M. Weiss, Active design reviews: principles and practices, in: IEEE Computer Society, 8th International Conference on Software Engineering, ICSE'85, 1985, pp. 132–136.

[7] D. Kelly, T. Shepard, Task-directed software inspection, Journal of Systems and Software, 2004, pp. 361–368.

[8] H. Gall, R. Klosch, and R. Mittermeir. Object-oriented re-architecting. In ESEC-5, Berlin, Germany, 1995.

[9] G. Y. Guo, J. M. Atlee, and R. Kazman. A software architecture reconstruction method. In WICSA-1, San Antonio, 1999.

[10] R. Kazman and J. Carriere. View extraction and view fusion in architectural understanding. In 5th International Conference on Software Reuse, Canada, June 1998.

[11] Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. Stemming architectural erosion by coupling ar-chitectural discovery and recovery. In STRAW'03: Second International SofTware Requirements to Architectures Workshop at ICSE 2003, Portland, Oregon, USA, 2003.

[12] Eugen Nistor, Justin R. Erenkrantz, Scott A. Hendrickson, and André van der Hoek. Archevol: Versioning architectural-implementation relationships. In Proceedings of the 12th International Workshop on Software Con-figuration Management, Lisbon, Portugal, September 2005.

[13] A. Soria, J.A. Diaz-Pace, M. Campo, Tool support for fault localization using architectural models, in: IEEE Computer Society, European Conference on Software Maintenance and Reengineering, CSMR'09, 2009, pp. 59–68.

[14] T. Ball, J.R. Larus, Optimally Profiling and Tracing Programs, 1994.

[15] Pyle D. Business Modeling and Data Mining. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publisher, 2003.

[16] M.G. Armentano, A.A. Amandi. Modeling sequences of user actions for statistical goal recognition, 2010.

[17] G. Bejerano. Automata Learning and Stochastic Modeling for Biosequence Analysis, 2003.

[18] D. Ron , Y. Singer , N. Tishby. The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length, 1994.

[19] IBM Corporation and The Eclipse Foundation. Eclipse platform technical overview. 2001, 2003, 2005.