Extracción y análisis de información estática orientada a la comprensión de programas para Sistemas OO

Enrique Alfredo Miranda[1], Mario Marcelo Berón[1] y Daniel Edgardo Riesco[1]

Resumen

Sin lugar a dudas, una de las tareas más complejas y que más tiempo consume en el ciclo de vida de una aplicación es la de Mantenimiento. En este entorno, las actividades que más tiempo consumen son aquellas que debe ejecutar el programador para lograr un completo entendimiento del sistema. A partir de esta necesidad, es que surge una disciplina de la Ingeniería de Software denominada Comprensión de Programas (CP). En este contexto, muchos autores afirman que un programador efectivamente comprende un programa cuando puede relacionar el Dominio del Problema con el Dominio del Programa. El primero hace referencia a la salida del sistema bajo estudio. El segundo se relaciona con los componentes del sistema utilizados para producir dicha salida. En este artículo se presenta una estrategia que asiste al programador durante el proceso de Comprensión de Programas de sistemas Orientados a Objetos (OO). Dicha estrategia realiza las siguientes tareas: i) extrae el Grafo Estático de Llamadas a Métodos (GELM), ii) aplica filtros que eliminan métodos muy relacionados con detalles de implementación, del GELM y iii) posibilita el análisis del GELM reducido, con el propósito de inferir funcionalidades. De esta manera la estrategia planteada pretende asistir al arduo proceso cognitivo que implica comprender un sistema.

Palabras clave: Ingeniería de Software, Comprensión de Programas, Extracción de Información Estática, Grafo Estático de Llamada a Métodos.

[1] Universidad Nacional de San Luis, Argentina.

Abstract

Undoubtedly, one of the most complex and time-consuming tasks in the life cycle of an application is the Maintenance one. Within this environment, the most time-consuming activities are those that the programmer must execute for a complete understanding of the system. Based on this need, Program Comprehension (PC), a Software Engineering discipline, arises to tackle the problem. In this context, many authors claim that a programmer comprehend a program when it can relate the Problem Domain with the Program Domain. The first refers to the output of the system under study. The second relates to the system components used to produce that output. In this article a strategy that assists the programmer during the understanding of Object Oriented (OO) systems is presented. The strategy performs the following tasks: i) extracts the Static Method Call Graph (SMCG), ii) applies filters that remove methods closely related to implementation details from the SMCG and iii) enables analysis of the reduced SMCG in order to obtain system functionalities. Thereby, the proposed strategy seeks to assist the arduous cognitive process that involves understanding a system.

Key words: Software Engineering, Program Comprehension, Static Information Extraction, Static Method Call Graph.

I. Introducción

Sin lugar a dudas, una de las tareas más complejas y que más tiempo consume en el ciclo de vida de una aplicación es la de Mantenimiento y Evolución de Software (MES) (Bennett y Rajlich, 2000; Pigoski, 1996). Dentro del entorno de MES, las tareas que más tiempo consumen son aquellas que debe ejecutar el programador para lograr un completo entendimiento del sistema (Boehm, 1984; Rugaber, 1995). A partir de la necesidad de asistir al arduo proceso de comprensión mencionado anteriormente, es que surge una disciplina de la Ingeniería de Software denominada Comprensión de Programas (CP). La CP se presenta como un área de investigación útil e interesante para impulsar el trabajo de MES a través de técnicas y herramientas que ayuden al ingeniero de software en el análisis y la comprensión de sistemas.

A través de extensos estudios en Modelos Cognitivos y Teorías del Aprendizaje (Brooks, 1978; Littman y cols, 1987; von Mayrhauser y Vans, 1994), ciertos autores afirman que el principal desafío en esta área consiste en relacionar el Domino del Problema con el Dominio del Programa (Beron, 2010; Lieberman y Fry, 1995; Rugaber, 1995) (ver figura 1). El primero hace referencia a la salida del sistema. El segundo a las componentes de software usadas para producir dicha salida (Beron, 2010; Da Cruz, Henriques, y Pinto, 2009; Storey, 2005). La reconstrucción de este tipo de relación es muy compleja ya que implica:

- Construir una o varias representaciones para el Dominio del Problema.
- Construir una o varias representaciones del Dominio del Programa.
- Elaborar un procedimiento de vinculación.

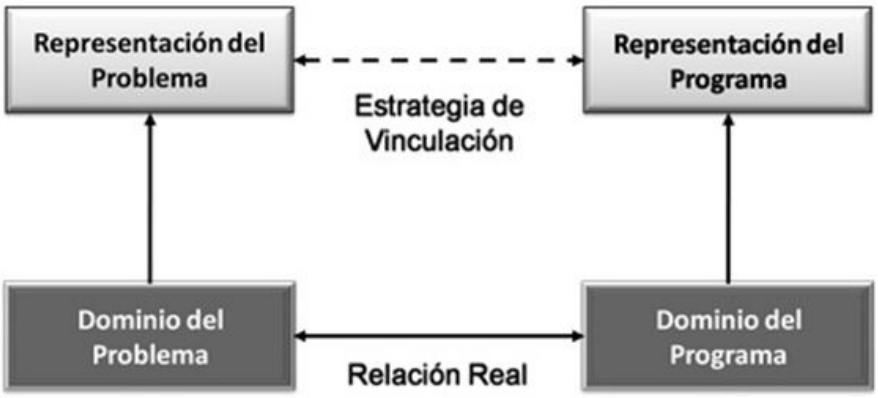

Figura 1: Modelo de Comprensión de Programas

Los tres pasos antes mencionados son de extrema importancia debido a que la integración de los mismos resulta en la elaboración de "verdaderas" estrategias de comprensión (Beron, 2010).

Para crear cada una de las representaciones mencionadas anteriormente, primero es necesario extraer información de ambos dominios. Por esta razón, se utilizan técnicas que permiten obtener información relacionada con determinados aspectos de la estructura y el comportamiento del programa. Dichas técnicas se conocen en el contexto de CP como Técnicas de Extracción de Información (TEI). Las mismas pueden ser clasificadas, en base al tipo de información que extraen, de la siguiente manera: estáticas y dinámicas. Las TEI estáticas indagan en el código fuente sin ejecutar el sistema (De Lucia, 2001; Eisenbarth, Koschke, y Simon, 2001; Rohatgi, Hamou-Lhadj, y Rilling, 2008). Las TEI dinámicas están relacionadas con información en tiempo de ejecución (Ball, 1999; Eisenbarth y cols., 2001; Cornelissen y cols., 2008). Estas últimas acotan el espacio de búsqueda, ya que en cada ejecución no se ven implicadas todas las secciones del código. Es importante mencionar que ambos tipos de técnicas son muy útiles para la elaboración de estrategias de comprensión consistentes y eficaces (Beron, 2010).

Este artículo presenta una estrategia que propone asistir al programador durante el proceso de Comprensión de Programas mediante la extracción y posterior análisis de información estática, con el objetivo de detectar las principales funcionalidades de un sistema OO.

II. Análisis Estático de Sistemas

Las TEI estáticas proveen información valiosa de la *estructura de los programas*[2], analizando artefactos de software relatívos al código fuente del sistema, sin llevar a cabo ejecuciones de los mismos. Por este motivo han sido utilizada con diferentes propósitos en distintas investigaciones (Rohatgi y cols., 2008; Gupta, Soffa, y Howard, 1997; Binkley, 2007).

Una clase de lenguajes muy utilizados son los que posibilitan el desarrollo OO (Orientado a Objetos). Su popularidad se debe a la facilidad con la que se puede interpretar su estructura, las características y/o restricciones que provee el encapsulamiento como también las ventajas que presentan en todas las etapas de desarrollo de software (Booch, James, y Jacobson, 2004; Maletic y cols, 2001).

2.1 Grafo Estático de Dependencia de Métodos

[2] Es decir, como están relacionadas los componentes de software que constituyen el sistema

Teniendo en cuenta lo explicado en el apartado precedente, es posible hacer uso del Grafo Estático de Llamada a Métodos (GELM) (Murphy y cols, 1998) para inferir y analizar las distintas funcionalidades que presenta el sistema.

Definición 1: un Grafo Estático de Llamada a Método (GELM) es una tupla $G=<M,I>$ donde M es el conjunto de nodos que representan los métodos del sistema $M=\{M_1,M_2,M_3,...\}$ e I es una relación binaria definida en MxM donde una tupla (M_i,M_j) representa una llamada estática[3] desde el método M_i al método M_j.

En la figura 2 se muestra un programa escrito en Java. En la figura 3 se muestra el Grafo Estático de Llamada a Métodos para dicho programa.

Desde un punto de vista estático, es posible inferir qué métodos invocados determinan el comportamiento del método que los invoca; en otras palabras, los métodos mantienen fuerte relación entre el *llamador* y el *llamado*.

Por otra parte, teniendo en cuenta todo lo expuesto en este apartado, es necesario resaltar un aspecto fundamental que concierne a esta temática: el volumen de información extraída o en este contexto, el tamaño del GELM.

```
public class X {
        // declaracion de atributos de clase
        public void X(){
                  // inicializacion
                  }
        public void a() {
            b();
            }
        public void b() {
            d();
            c();        }
        public void c() {
            // Se deja vacio a proposito
            }
        public void d() {
            e();
            }
        public void e() {
            f();
            g();
            }
        public void f() {
            // Se deja vacio a proposito
            }
        public void g() {
            // Se deja vacio a proposito
            }
        public void h() {
            i();
            }
        public void i() {
            d();
            }
    }
public class Y {
        // declaracion de atributos de clase
        public void Y(){
             // inicializacion
        }
        public void j() {
             X variableX = new X();
             variableX.d();
        }
    }
```

Figura 2: Ejemplo de Programa Java

[3] En esta investigación se considera una llamada estática cuando figura en el código fuente

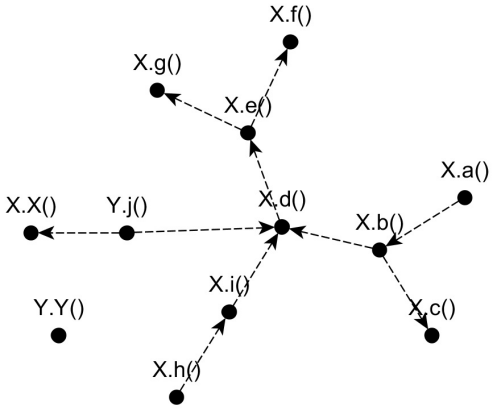

Figura 3: GELM para el código mostrado en la figura 2.

III. Síntesis de Dependencias

Debido al tamaño que puede llegar a alcanzar el GELM, es necesario contar con métodos de reducción de información (Berón y cols., 2007; Spärck Jones, 2007). Como es posible inferir, dichos métodos deben reducir considerablemente la pérdida de artefactos de software sensibles a la lógica del programa. Para esto, es necesario diseñar estrategias que permitan discernir entre artefactos que son relevantes respecto de la lógica subyacente del sistema y los que no lo son. En este trabajo en particular, se utiliza una estrategia que puede dividirse en dos fases, de acuerdo a la esencia de la información filtrada. En los siguientes apartados se describen cada una de estas.

3.1 Primer Fase de Filtrado

Durante la primera fase, se filtran los métodos que por naturaleza son externos a la lógica subyacente del sistema. Teniendo en cuenta esto, los mismos se podrían remover sin afectar la comprensibilidad global del objeto de estudio, ya que son detalles de implementación de bajo nivel. A continuación se mencionan los elementos filtrados en esta instancia de la investigación:

- **Librerías nativas del lenguaje:** por ejemplo, en el caso de Java: java.lang, java.util, java.net, entre tantas otras.
- **Librerías que el usuario defina como irrelevantes.**
- **Métodos redundantes:** teniendo en cuenta lenguajes OO, hasta el momento

se filtran los siguientes métodos: a) constructores externos al código analizado; b) métodos destructores y c) métodos aislados de clases anónimas (Bracha, Darcy, y Von Der Ahe, 2010);

• **Métodos sobrecargados:** se considera importante reducir los métodos sobrecargados que se llaman sucesivamente con distintos parámetros; por ejemplo, getDatos(java.lang.String) llama a getDatos(java.util.Map) que a su vez llama a getDatos(java.lang.long). Notoriamente, este tipos de transitividades aisladas no aportan a la lógica del programa ya que, en el contexto del GELM, simplemente revelan detalles de implementación.

Esta fase requiere de muy poca intervención por parte del usuario, ya que la mayoría de los elementos que serán filtrados poseen altas probabilidades de ser externos al código o de no aportar información relevante respecto de la lógica del sistema analizado.

3.2 Segunda Fase de Filtrado

Durante la segunda fase se llevan a cabo diferentes análisis con el objeto de filtrar métodos propios del sistema, pero que podrían ser ocultados sin afectar la comprensión de las representaciones; en este caso en particular, el GELM.

Para detectar los elementos referenciados en el párrafo anterior se ha utilizado un enfoque similar al planteado por Hamou-Lhadj y Lethbridge (Hamou-Lhadj y Lethbridge, 2006) para sintetizar trazas de ejecución. En dicho trabajo, los autores proponen identificar elementos que no son importantes con respecto a la lógica del programa. A dichos elementos se los denomina utilidades.

El razonamiento subyacente que aplican los autores para determinar las utilidades es el siguiente: *mientras más llamadas desde diferentes "lugares" posea un método, entonces es muy probable que haya sido diseñado con múltiples propósitos, por lo tanto, es muy probable que sea una utilidad*.

Un enfoque similar al expuesto en el párrafo precedente puede ser utilizado con el GELM, es decir, se ocultan los nodos que tienen altas probabilidades de ser utilidades. Para llevar a cabo este filtrado se tienen en cuenta un conjunto de métricas que, por medio de ciertos cómputos, permiten determinar el "grado de utilidad" para cada nodo (método). Hasta el momento se consideran las siguientes métricas: fan-in, fan-out e intermediación de cada nodo en el grafo. La primera indica la cantidad de métodos que invoca el método actual (grado de entrada); la segunda, indica la cantidad de métodos que son invocados por el método actual (grado de salida). La tercera es una medida de centralidad que cuantifica la frecuencia o el número de veces que un nodo actúa como un "puente" a lo largo del camino más corto entre otros dos nodos (Brandes, 2001).

Las primeras dos métricas son utilizadas en la siguiente fórmula:

$$U(r) = \frac{FanIn(r)}{N} * \frac{\text{Log}(\frac{N}{FanOut(r)+1})}{Log\ (N)}$$

donde *r* representa el método para el cual se realiza el cálculo y N es el número total de métodos en el grafo. Si U(r)=0, entonces r no es una utilidad; si U(*r*) tiende a 1 (de acuerdo al sistema analizado), muy posiblemente lo sea. La fórmula que computa U(*r*) es explicada más detalladamente en (Hamou-Lhadj y Lethbridge, 2006).

Claramente, es necesario definir una cota que determine a partir de que valor obtenido, un método se considera como utilidad. Este valor lo debe determinar el usuario y se debe establecer teniendo en cuenta distintos factores como el tamaño del proyecto, el tipo de aplicación, la proporción de información que se desea filtrar, entre otros. Sin embargo, es necesario brindar al usuario distintos mecanismos para la determinación de un valor aproximado. Una forma de aproximar un valor umbral es mediante el cálculo del promedio los valores de utilidad de todos los métodos. De este modo, el valor estimado se calcula teniendo en cuenta los valores que se obtienen en base al sistema analizado y así el mismo está relacionado con el dominio atacado.

Por otra parte, la métrica *intermediación* es utilizada para filtrar posibles falsos positivos determinados mediante la fórmula U(*r*); es decir, en el supuesto caso que un método se determine como utilidad, si su valor de intermediación (normalizado) supera un umbral determinado, entonces muy posiblemente dicho nodo no sea una utilidad.

Para esta métrica, el usuario también debe especificar un valor como umbral para filtrar falsos positivos. De la misma forma que con la función de utilidad, es posible calcular el promedio de todos los valores de intermediación.

Finalmente se procede a filtrar los nodos con utilidades superiores al valor umbral, con el objeto de reducir la información que será representada. Este proceso de filtrado se realiza en cascada; es decir, si se oculta un nodo determinado, además se ocultan los arcos entrantes y salientes del mismo, como así también lo nodos descendientes.

3.2.1 Algoritmo de Filtrado

El algoritmo de filtrado descripto en este apartado recibe como entrada un sistema OO, construye el GELM para el mismo, procesa esta información con el objetivo de filtrar detalles de implementación y finalmente retorna dicho grafo sintetizado.

A continuación se muestran los pasos que lleva a cabo el algoritmo:

Paso 1: generar el GELM.

Paso 2: filtrar artefactos que por naturaleza son externos al sistema o detalles de implementación (primera fase de filtrado, apartado 3.1).

Paso 3: establecer los valores umbrales para la determinación de utilidades e intermediación.

Paso 4: realizar análisis fan-in/intermediación (segunda fase de filtrado, sección 3.2).

Al finalizar, se visualiza el GELM resultante y la cantidad de nodos y arcos en el mismo. El usuario analiza los resultados y de acuerdo a su criterio, este puede ejecutar nuevamente el algoritmo especificando librerías externas a ser filtradas y/o valores umbrales diferentes, estimados de acuerdo al nivel de filtrado deseado.

IV. Identificación de Funcionalidades

Todo los procesos explicados en secciones precedentes sirven como base para mejorar la comprensión de un sistema en particular. Con este objetivo, se han filtrado elementos considerados de poca relevancia con respecto a la lógica subyacente de dicho sistema. En este apartado se propone un método para agrupar las funcionalidades del mismo, con el fin de mejorar la comprensión por parte del programador.

Teniendo en cuenta que en el GELM han quedado sólo métodos sensibles a la lógica subyacente del sistema, una estrategia para aproximar estas funcionalidades es mediante el análisis de fan-in. Los nodos cuyo fan-in es igual a 0 se pueden considerar como los puntos de entrada al sistema. Mediante esta aproximación, todos caminos que comiencen desde dicho nodo describen la funcionalidad que el método implementa. De esta manera, se pueden desagregar las funcionalidades que el sistema presenta, con el fin de analizar por separado el propósito de cada una.

Tomando como ejemplo el grafo mostrado en la figura 3, el resultado de aplicar este método equivale a los siguientes conjuntos de nodos y arcos:

$\boldsymbol{Func}_{\mathbf{X.a()}}$ = *{(X.a(),X.b()); (X.b(),X.c()); (X.b(),X.d()); (X.d(),X.e()); (X.e(),X.f()); (X.e(),X.g()) }*
$\boldsymbol{Func}_{\mathbf{X.h()}}$ = *{(X.h(),X.i()); (X.i(),X.d()); (X.d(),X.e()); (X.e(),X.f()); (X.e(),X.g()) }*
$\boldsymbol{Func}_{\mathbf{X.j()}}$ = *{(Y.j(),X.d()); (X.d(),X.e()); (X.e(),X.f()); (X.e(),X.g()); (Y.j(),X.X()) }*
$\boldsymbol{Func}_{\mathbf{Y.Y()}}$ = *{ }*

Como se puede notar, las funcionalidades se ven plasmadas en cada conjunto $\boldsymbol{Func}_{<n>}$, donde se encuentran todos los caminos que comienzan desde el nodo *n* cuyo fan-in es 0.

Para exhibir las distintas funcionalidades, se cree conveniente utilizar un algoritmo de *despliegue* jerárquico, donde los nodos con fan-in=0 se ubican en la parte superior y se van desagregando hasta finalizar la descripción de la funcionalidad. Tomando como base el ejemplo mencionado en párrafos previos, el grafo se visualiza como en la figura 4.

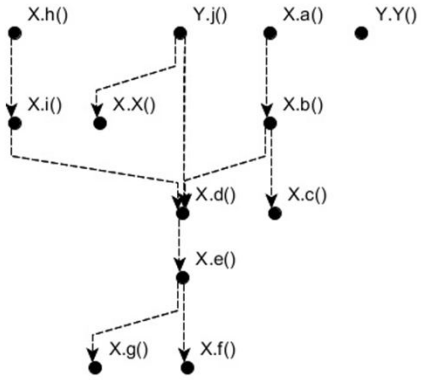

Figura 4: GELM para código mostrado en la figura 2. Visualización de funcionalidades con *despliegue* jerárquico.

Es relevante destacar que en este ejemplo en particular, los métodos no poseen nombres significativos, a diferencia del caso de estudio analizado en la sección 5.

V. Caso de Estudio: Módulo de MercadoPago

En este aparatado se presenta un caso de estudio para mostrar la aplicabilidad del enfoque. En el mismo se analiza el módulo de integración de pagos para la plataforma Mercado Pago de la empresa Mercado Libre[4] [5]. Dicho módulo está escrito en lenguaje Java y utiliza JSON[6] como formato de intercambio de datos. También utiliza ReST (Fielding, 2000) para interactuar con la plataforma de MercadoPago.

Para poder llevar a cabo el análisis de este caso de estudio, se construyó un prototipo que implementa la estrategia planteada en este artículo.

La construcción del GELM fue llevada a cabo utilizando *Dependency Finder*[7]. Se utilizó esta herramienta porque presenta un toolkit que posibilita la extracción y manipulación de todas las dependencias de un sistema escrito en Java; de la misma manera que posibilita exportar las dependencias en diferentes formatos (XML, GraphML, entre otros).

Por otra parte, se ha utilizado *Document Object Model* (DOM) (Nicol, Wood, Champion, y Byrne, 2001) para implementar los filtros explicados en los apartados 3.1, 3.2 y 3.2.1.

[4] http://developers.mercadolibre.com
[5] https://github.com/mercadopago/sdk-java
[6] http://json.org/json-es
[7] http://depfind.sourceforge.net

En la figura 5 se exhibe el grafo obtenido por *Dependency* Finder; el mismo contiene 100 nodos y 188 arcos dirigidos. Se puede notar que hay dos tipos de nodos, los nodos con relleno de color representan métodos definidos en el código analizado, mientras que los nodos sin relleno son métodos no definidos en el código fuente.

Luego de pasar por la primer fase de filtrado, el grafo contiene 57 nodos y 81 arcos; el mismo se puede observar en la figura 6. Como se puede observar, la estrategia de filtrado redujo significativamente la cantidad de información. El punto a destacar es que no se perdió información escencial de la semántica del sistema.

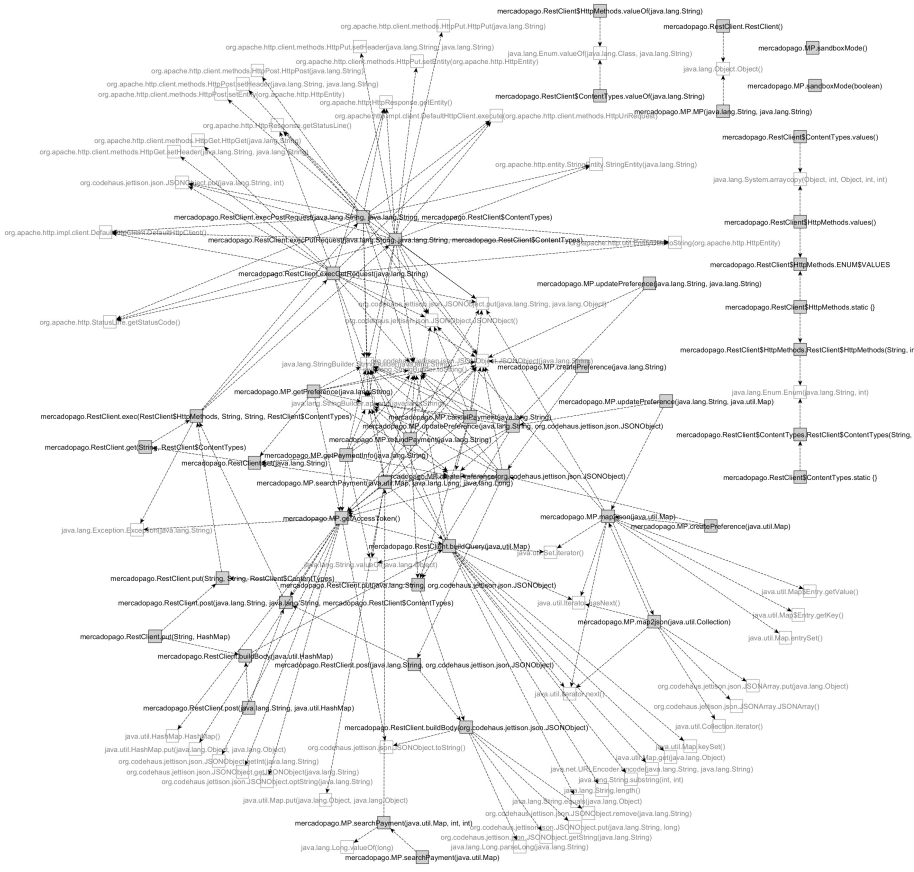

Figura 5: Grafo Estático de Llamada a Métodos inicial para módulo Java de integración de pagos para la plataforma Mercado Pago. Contiene 100 nodos y 188 arcos.

El siguiente filtro busca utilidades en el GELM, como se explicó en el apartado 3.2.1. El resultado de calcular la función de utilidad para todos los nodos permitió obtener los siguientes valores: a) umbral para filtrar utilidades *0,0211*; b) umbral de intermediación *0,123* (para filtrar falsos positivos). En la tabla 1 se muestran algunos ejemplos de métodos y sus correspondientes valores de fan-in (f-in), fan-

out (f-out), Función de utilidad (*U(r)*) e Intermediación (*I(r)*). En la misma también se indica si cada método se considera utilidad (F.*U(r)*), si resultó un falso positivo (Falso+) y si finalmente debe ser filtrado (Oculto). Para mejorar la visibilidad de la tabla, la signatura de los métodos ha sido reducida (por ejemplo, el nombre de paquete *mercadopago* se reduce a *mp,* j*ava.lang.String* a *String*, *org.apache.http* a *...http*, *org.codehaus.jettison.json* a *...json*, entre otros).

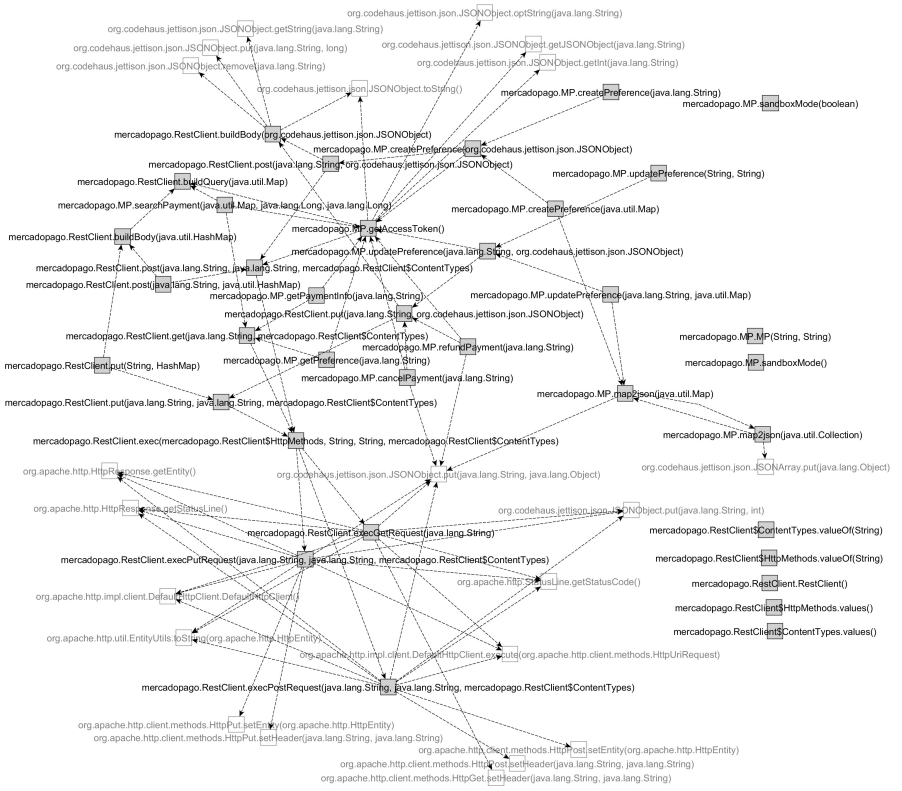

Figura 6: Grafo Estático de Llamada a Métodos para módulo Java de integración de pagos para la plataforma Mercado Pago, después de Primera Fase de filtrado. Contiene 57 nodos y 81 arcos.

La tabla 1 muestra que el método *mp.MP.cancelPayment(String)* obtuvo un valor para *U(r)* de *0*, esto se debe a que no posee arcos que le lleguen, es decir, ningún método lo llama. Esto pasa con todos los métodos cuyo fan-in sea igual a 0; los mismos no serán ocultados por la estrategia. Por otro lado, el método *mp.MP.map2json(Map)* obtuvo un valor para *U(r)* de *0,0389* (mayor que el valor umbral *0,0211*), lo que implica que es un candidato a ser filtrado. Dado que su valor de intermediación (*I(r)=0,02*) es menor que el umbral establecido (*0,123*), no es un falso positivo, por lo tanto éste y sus descendientes deben ser eliminados del

GELM. En el caso de *mp.RestClient.exec(...)*, obtuvo un valor de *U(r)* por encima del umbral de utilidad (*U(r)=0,0351*), sin embargo resultó ser el nodo con más intermediación en el grafo; por lo tanto no se elimina del GELM.

El GELM resultante de aplicar todos los filtros posee posee 36 arcos y 38 nodos teniendo en cuenta también los nodos aislados (8); no obstante, esta clase de nodos, para esta instancia de la investigación, sería conveniente ocultarlos en el alguna de las fases de filtrado para facilitar el posterior proceso de análisis. En la figura 7 se exhibe el GELM final, sin nodos aislados.

Signatura (reducida)	f-in	f-out	U(*r*)	I(*r*)	F. U(*r*)	Flaso+	Oculto
mp.MP.cancelPayment(String)	0	3	0	0	No	No	No
mp.MP.getPaymentInfo(String)	0	2	0	0	No	No	No
mp.MP.map2json(Collection)	1	2	0,0129	0,01	No	No	No
mp.MP.map2json(Map)	3	2	0,0389	0,02	Si	No	Si
mp.RestClient.buildBody(HashMap)	2	1	0,0295	0,01	Si	No	Si
mp.RestClient.buildQuery(Map)	3	0	0,0535	0	Si	No	Si
mp.RestClient.exec(...)	3	3	0,0351	1	Si	Si	No
mp.RestClient.put(String,String,...)	2	1	0,0295	0,25	Si	Si	No
...http.HttpResponse.getStatusLine()	3	0	0,0535	0	Si	No	Si
…json.JSONObject.getInt(String)	1	0	0,0178	0	No	No	No
...json.JSONObject.put(String,int)	3	0	0,0535	0	Si	No	Si

Tabla 1: Análisis para detección de utilidades. El valora umbral para las utilidades es 0,0211. El valor umbral de intermediación es 0,123.

En principio el grafo contenía 100 nodos y 188 arcos; el grafo de la figura 7 contiene 30 nodos y 36 arcos. De esta manera se ha podido reducir en grandes proporciones la información (70% con respecto a los nodos y cerca de 81% respecto de los arcos), teniendo en cuenta que no se han filtrado librerías extras a las nativas de Java.

Tomando como base este GELM, compuesto de nodos que representan métodos altamente relacionados con la lógica subyacente del sistema, es posible realizar el análisis fan-in=0 para detectar las posibles entradas al sistema. En la figura 7 se puede examinar el GELM con un *despliegue* jerárquico, en donde las posibles entradas al sistema se encuentran en la parte superior del mismo. En esta representación se han reducido los nombres de algunos elementos y se han reubicado algunos nodos para mejorar la visibilidad del GELM. A través de los distintos caminos se puede determinar qué métodos determinan el comportamiento de cada método con fan-in=0.

A partir del GELM de la figura 7, es posible inferir las funcionalidades del sistema, de acuerdo a lo expuesto en la sección 4. A modo de ejemplo y con el

propósito de que el artículo quede autocontenido, se considera un conjunto reducido de los nodos con fan-in=0 (señalados en la figura 7); dichos nodos representan los siguientes métodos:

MP.searchPayment(Map,Long,Long), *MP.getPaymentInfo(String)*, *MP.getPreference(String)*, *MP.cancelPayment(String)* y *MP.createPreference(String)*.

A través del análisis de los diferentes caminos que parten de los nodos con fan-in=0, es posible inferir lo siguiente:

- Los 5 métodos analizados invocan al método *MP.getAccessToken()*. Se puede deducir que es necesario obtener un token de acceso para interactuar con los recursos del sitio.
- De manera similar al ítem anterior, todos los caminos llevan al método *RestClient.exec(...)* (cuyo valor de intermediación es 1). Esto claramente se da en esta clase de sistemas que interactúan con recursos accedidos mediante *URIs* específicas.
- Métodos cuyo nombre dan ciertos "indicios" de consulta, como *getPaymentInfo()*, *getPreference()* o *searchPayment()*, invocan (directa o indirectamente) al método *RestClient.get(...)*. Efectivamente, este método accede a recursos webs con carácter de "solo lectura".
- Por otro lado, los métodos *MP.createPreference()* y *MP.cancelPayment()* invocan (directa o indirectamente) a los métodos *post* y *put* respectivamente, de la clase *RestClient.* Claramente, los nombres "crear" y "cancelar" de los métodos mencionados anteriormente, sugieren posibles modificaciones a los recursos webs accedidos a través las operaciones *put* o post.

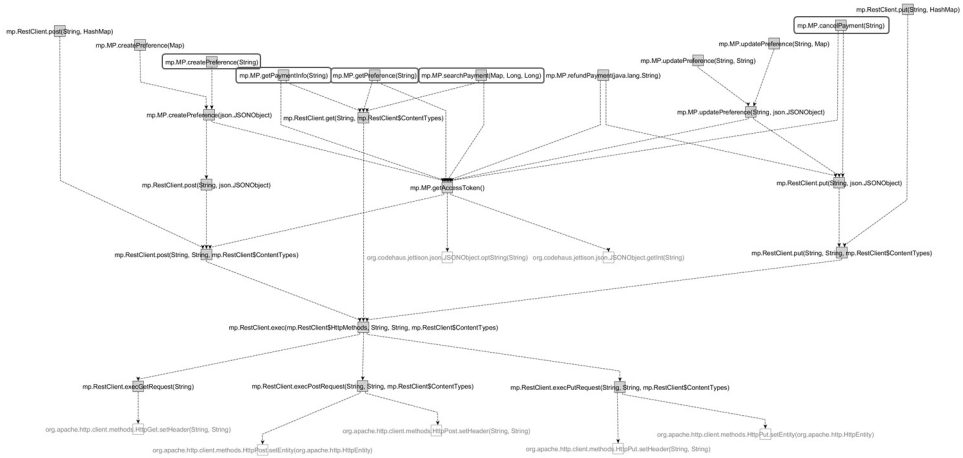

Figura 7: Representación jerárquica del GELM final. Contiene 30 nodos y 36 arcos.

Notablemente, para poder comprender este tipo de aplicaciones utilizando este enfoque, es necesario tener conocimiento de base sobre el contexto en el que está programada la aplicación bajo estudio (llamesé ReST, HTTPS, JSON, etc). Finalmente, es importante destacar que el análisis de la signatura de los métodos es muy importante, ya que brindan información semántica valiosa respecto de la funcionalidad del método.

VI. Conclusiones y Trabajos Futuros

En este artículo se definió una estrategia de CP dirigida a sistemas OO. La misma propone extraer el GELM del sistema (o módulo) bajo estudio y aplicar filtros de información con el objetivo de disminuir el volumen del mismo. Para esto se utiliza un enfoque basado en la detección de métodos que no aportan información relevante respecto de la lógica del sistema analizado. Una vez filtrada la información, el GELM queda compuesto, en gran proporción, por métodos altamente relacionados con el Dominio del Problema del sistema. Finalmente, con el objetivo de inferir las funcionalidades del sistema, se analiza el GELM tomando como base los nodos cuyo fan-in sea cero.

Para probar la eficacia de la estrategia propuesta, la misma se aplicó a un módulo Java de la plataforma de pagos *Mercado Pago*[8]. Este tipo módulos permiten la integración de sistemas de ventas del usuario (por ejemplo en sitios webs o aplicaciones móviles) con la API de la plataforma *e-commerce* antes mencionada. El resultado de la aplicación de la estrategia fue muy satisfactorio debido a que se pudo: i) lograr una significativa reducción del GELM (un porcentaje en reducción de nodos y arcos del 70% y 81% respectivamente) y ii) identificar las principales funcionalidades que ofrece el módulo bajo estudio. Por medio de dichas funcionalidades el usuario interactúa con la plataforma a través de una arquitectura *ReST*, conformando mensajes en *JSON* para lograr consultar o modificar los recursos que la dicha plataforma provee.

La estrategia planteada en esta investigación es un buen punto de partida para elaborar una estrategia más robusta, eficiente y funcional para sistemas de software escritos en lenguajes OO . Claramente, para esto es necesario orientar la investigación en dirección a las siguientes temáticas:

- Enriquecer las dos fases de filtrado propuestas por medio de la detección de otros tipos de métodos que reflejen detalles de implementación, con el principal objetivo de identificar los Casos de Uso del sistema.

[8] https://developers.mercadopago.com/

• Añadir a la estrategia la detección de polimorfismo que se propone en los trabajos (Bacon y Sweeney, 1996) y (Sundaresan y cols., 2000).
• Definir un método más sofisticado y presiso para el cálculo de los valores umbrales de utilidad e intermediación en la estrategia.
• Utilizar estrategias de generalización de conceptos como otro método de síntesis de información (Spärck Jones, 2007).
• Complementar la estrategia con extracción y análisis de información dinámica de los sistemas.
• Incorporar distintas técnicas de Visualización de Software para brindar diferentes perspectivas del sistema.

La incorporación de las estrategias mencionadas en los ítems precedentes asistirá significativamente al arduo proceso cognitivo que implica comprender un sistema, brindando solución a uno de los principales desafíos en Comprensión de Programas: relacionar los Dominios del Problema y del Programa.

Referencias

Bacon, D. F., y Sweeney, P. F. (1996). Fast Static Analysis of C++ Virtual Function Calls. *ACM Sigplan Notices*, 31(10), 324–341.

Ball, T. (1999). The Concept of Dynamic Analysis. En *Software Engineering -ESEC/FSE99* (pp. 216–234).

Bennett, K., y Rajlich, V. (2000). Software Maintenance and Evolution: a Roadmap. En *Proceedings of the Conference on the Future of Software Engineering* (pp. 73–87). New York, NY, USA: ACM.

Berón, M., Henriques, P., Varanda Pereira, M. J., y Uzal, R. (2007). PICS un Sistema de Comprensión e Inspección de Programas. *Congreso Argentino de Ciencias de la Computacion* (CACIC07).

Beron, M. (2010). Program Inspection to interconnect Behavioral and Operational Views for Program Comprehension. *Ph.D Thesis Dissertation at University of Minho*. Braga. Portugal.

Binkley, D. (2007). Source Code Analysis: a Road Map. En *2007 Future of Software Engineering* (pp. 104–119).

Boehm, B. (1984). Software Engineering Economics. S*oftware Engineering, EEE Transactions on* (1), 4–21.

Booch, G., James, R., y Jacobson, I. (2004). *The Unified Software Development Process*. Addison-Wesley Object Technology Series.

Bracha, G., Darcy, J. D., y Von Der Ahe, P. (2010, febrero 23). *Introspection Support for Local and Anonymous Classes*. Google Patents. (US Patent 7,669,184)

Brandes, U. (2001). A Faster Algorithm for Betweenness Centrality*. J*ournal of Mathematical Sociology*, 25(2), 163–177.

Brooks, R. (1978). Using a Behavioral Theory of Program Comprehension in Software Engineering. P*roceedings of the 3rd International Conference on Software Engineering*, 196–201.

Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deurse, A., y Wijk, J. (2008). Execution Trace Analysis through Massive Sequence and Circular Bundle Views. *Report TUD-SERG-2008-008*.

Da Cruz, D., Henriques, P., y Pinto, J. (2009). Code Analysis: Past and Present. *Proceeding of the Third International Workshop on Foundations and Techniques for Open Source Software Certification*, 00, 1-10.

De Lucia, A. (2001). Program Slicing: Methods and Applications. En *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on* (pp. 142–149).

Eisenbarth, T., Koschke, R., y Simon, D. (2001). Aiding Program Comprehension by Static and Dynamic Feature Analysis. En S*oftware Maintenance, 2001. Proceedings. IEEE International Conference on* (pp. 602–611).

Fielding, R. (2000). Representational State Transfer. *Architectural Styles and the Design of Netowork-based Software Architecture*, 76–85.

Gupta, R., Soffa, M. L., y Howard, J. (1997). Hybrid Slicing: Integrating Dynamic Information with Static Analysis. A*CM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4), 370–397.

Hamou-Lhadj, A., y Lethbridge, T. (2006). Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. En *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on* (pp. 181–190).

Lieberman, H., y Fry, C. (1995). Bridging the Gulf Between Code and Behavior in Programming. En *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 480–486).

Littman, D., Pinto, J., Letovsky, S., y Soloway, E. (1987). Mental Models and Software Maintenance. *Journal of Systems and Software*, 7(4), 341–355.

Maletic, J., Leigh, J., Marcus, A., y Dunlap, G. (2001). Visualizing Object Oriented Software in Virtual Reality. En *Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)* (pp. 26–35). Society Press.

Murphy, G. C., Notkin, D., Griswold, W. G., y Lan, E. S. (1998). An Empirical Study of Static Call Graph Extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM*), 7(2), 158–191.

Nicol, G., Wood, L., Champion, M., y Byrne, S. (2001). Document Object Model (DOM) Level 3 Core Specification.

Pigoski, T. (1996). *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. New York, NY, USA: John Wiley & Sons, Inc.

Rohatgi, A., Hamou-Lhadj, A., y Rilling, J. (2008). An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. En *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on* (pp. 236–241).

Rugaber, S. (1995). Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20), 341–368.

Spärck Jones, K. (2007). Automatic Summarising: the State of the Art. *Information Processing & Management*, 43(6), 1449–1481.

Storey, M.-A. (2005). Theories, Methods and Tools in Program Comprehension: Past, Present and Future. *Proceedings of the 13th International Workshop on Program Comprehension*, 181–191.

Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., y Godin, C. (2000). Practical Virtual Method Call Resolution for Java. *ACM SIGPLAN Notices*, 35(10), 264–280.

von Mayrhauser, A., y Vans, M. (1994). Comprehension Processes During Large Scale Maintenance. En *Proceedings of the 16th International Conference on Software Engineering* (pp. 39–48).