Identification of Architectural Technical Debt: an Analysis Based on Naming Patterns

Paul Mendoza del Carpio
Universidad La Salle
Arequipa, Perú
(+51) 95 979 7933
pmendozadelcarpio@gmail.com

ABSTRACT
Hasty software development can produce immediate implementations with source code unnecessarily complex and hardly readable. These small kinds of software decay generate a technical debt that could be big enough to seriously affect future maintenance activities. This work presents an analysis technique for identifying architectural technical debt related to non-uniformity of naming patterns; the technique is based on term frequency over package hierarchies. The proposal has been evaluated on projects of two popular organizations, Apache and Eclipse. The results have shown that most of the projects have frequent occurrences of the proposed naming patterns, and using a graph model and aggregated data could enable the elaboration of simple queries for debt identification. The technique has features that favor its applicability on emergent architectures and agile software development.

CCS Concepts
• Software and its engineering→Automated static analysis
• Software and its engineering→Patterns

Keywords
Architectural technical debt, naming pattern, static analysis.

1. INTRODUCCIÓN
Optar por una fácil o rápida solución a corto plazo en una actividad de cualquier fase del desarrollo de software (i.e.; requisitos, diseño, implementación), puede generar una deuda técnica acumulable, la cual en un determinado período de tiempo puede llegar a ser lo suficientemente grande para afectar entregas posteriores, dificultándose el abordar o concretar un resultado satisfactorio [6] [24] [37]. La deuda comprende cualquier aspecto del software que es reconocido como inapropiado, y que no ha sido atendido en su momento, entre ellos se puede mencionar código fuente complejo que requiere ser reestructurado o refactorizado [24]. Tal deuda es un tema cuyo interés se ha venido incrementado a través de los años [36], como lo muestran estudios publicados [37], y reportes de búsqueda en Google Trends (véase la figura 1). Frecuentemente la deuda técnica, cuando es introducida, es poco visible para los tomadores de decisiones en el desarrollo de software [5]. El desarrollo de técnicas de identificación y monitoreo de incidencias de deuda técnica es importante para que la deuda sea explícita y sea atendida en el momento correcto [3] [11] [22] [24] [35] [37].

.
2016 8th Euro American Conference on Telematics and Information Systems (EATIS), Cartagena de Indias – Colombia. April 27 - 29

978-1-5090-2435-3/16/$31.00 ©2016 IEEE

La deuda técnica puede ser causada al no preserxvar el diseño arquitectónico, o al no usar convenciones o estándares de programación [35]; el incluir ésta como un factor de decisión dentro del desarrollo de software requiere información acerca de las incidencias de deuda técnica existentes en el sistema de software, dónde se encuentran éstas, y su magnitud, tal información puede ser obtenida mediante el análisis de código fuente [5].

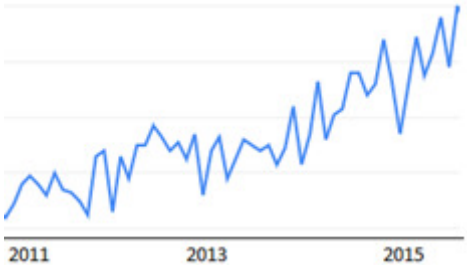

Figura 1. Búsquedas de "technical debt".

La contribución del presente trabajo puede ser resumida como sigue:

1. una técnica de análisis para identificar deuda técnica arquitectónica por no uniformidad de patrones, y
2. un conjunto de patrones de nombrado de clases dentro de la jerarquía de paquetes de un sistema de software.

2. DEUDA TÉCNICA ARQUITECTÓNICA (DTA)
La DTA es un tipo de deuda técnica que comprende soluciones sub-óptimas respecto a atributos de calidad internos o externos definidos en la arquitectura pretendida (se comprometen principalmente atributos de mantenibilidad o evolucionabilidad del software) [2] [11]. Cabe señalar que los cambios del software que no están relacionados directamente al comportamiento externo del sistema, y sí están relacionados a cualidades de diseño, son frecuentemente postergados o abandonados con el fin de reducir el tiempo de entrega del producto de software [3], lo cual incrementa las incidencias de DTA.

La DTA es una de las deudas más relacionadas al código fuente [24], sin embargo en la práctica es difícil de identificar, puesto que ella no proporciona comportamiento observable a usuarios finales [11] [36], y puede cambiar con el tiempo debido a información obtenida de los detalles de implementación [2]. En consecuencia, no sería adecuado asumir que la DTA puede ser completamente identificada desde un inicio [2].

En [2] se presenta un conjunto de ejemplares de DTA, entre ellos se tiene la DTA de no uniformidad de patrones, relacionada a convenciones de nombrado aplicadas en parte del sistema que no son cumplidas en otra parte del sistema de software [2]. Este ejemplar de DTA es el abordado por el presente trabajo.

Por otro lado, diversos enfoques ágiles consideran a la arquitectura como una característica emergente donde no se lleva a cabo un diseño anticipado; pero se reestructura código fuente, y se realiza un refinamiento sobre los elementos arquitectónicos [21]. La refactorización es una práctica regular utilizada en enfoques ágiles, y que es aplicada frecuentemente sobre código fuente [1]; ella contribuye a que una arquitectura satisfactoria emerja, mejorando la estructura interna de la aplicación, haciendo más comprensibles las partes de la arquitectura, y evitando el deterioro de la misma, especialmente en aquellas que son definidas en forma ligera [15] [21]. El realizar una refactorización incompleta es una causa de DTA que puede dejar parte de una DTA y además generar nueva DTA [2]. La refactorización puede ser realizada manualmente, o en forma semi o completamente automática. El enfoque completamente automático realiza la identificación y realiza las transformaciones necesarias, sin embargo la confirmación para aceptar los resultados de la refactorización es decidida finalmente por un humano [1] [16]. El presente trabajo posibilita una refactorización completamente automática, atendiendo la etapa inicial de identificación (mediante el análisis en presentación); aplicándose posteriormente una transformación mediante una refactorización de nombrado de clases. Ésta última es un tipo de refactorización global (i.e.; afecta clases en más de un paquete) [10] con un nivel de API (Application Programming Interface) [30], que es bastante usada en forma automatizada en entornos de programación [8] [30], y que habitualmente es empleada con fines conceptuales y de reorganización [25], distinguiéndose de otras formas de refactorización por favorecer la trazabilidad del software [1].

3. PATRONES DE NOMBRADO

Conforme un software evoluciona, su código es una fuente de información que se encuentra efectivamente actualizada, y contiene información relevante acerca del dominio de la aplicación [14]. El código complejo es una de las mayores fuentes de deuda técnica [22]; donde el uso consistente de convenciones de nombrado definidas por una arquitectura, facilita y acelera actividades de comprensión de un sistema de software [34]. Sin embargo, estas convenciones podrían no mantenerse en forma consistente a lo largo de todo un sistema de software. Tal fenómeno se puede amplificar en equipos ágiles [2]; donde se brinda un empoderamiento a los equipos en términos de diseño, diferentes equipos de desarrollo trabajando en forma paralela acumulan diferencias en su diseño y arquitectura, y las políticas de nombrado no son siempre expresadas en forma explícita y formal, surgiendo discrepancias y requiriendo esfuerzo [2].

La importancia de los nombres de clases radica en que éstos determinan la legibilidad del código, su portabilidad, mantenibilidad, y accesibilidad a nuevos miembros del equipo, conectando el código fuente al dominio del problema [19]. También, expertos en la industria resaltan la importancia del nombrado de identificadores en el software [12] [28] [31]. Por tanto, tal importancia puede llegar a niveles de análisis arquitectónico, donde el identificar términos de componentes es una tarea menos complicada cuando los identificadores están conformados por palabras completas o abreviaturas significativas [9] [14]. En las siguientes subsecciones se presenta un conjunto de patrones de nombrado inspirados en la consideración proporcionada a los paquetes como importantes elementos de organización de código fuente; los patrones son definidos de acuerdo al uso frecuente de un término en el nombre de clases dentro de una jerarquía de paquetes subyacente. Ejemplos de los patrones son presentados utilizando distintos paquetes en cada patrón, con el propósito de mostrar la presencia de los patrones en distintos proyectos reales de las organizaciones Apache y Eclipse.

3.1 Patrón 1: Paquete

El término es empleado frecuentemente en clases incluidas en un mismo paquete, véase la figura 2 que muestra paquetes del software Apache MyFaces. Se define a f como un valor de frecuencia mínima; T es el conjunto de los términos usados en nombres de clases; P es el conjunto de los paquetes; C(p) es el conjunto de las clases de $p \in P$; y C(p,t) es el conjunto de las clases de p que usan el término $t \in T$. Los términos t de este patrón son tales que $(| C(p,t) | / | C(p) |) \geq f$, y $| C(p) | > 2$.

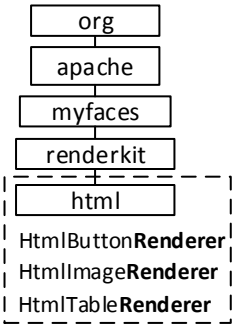

Figura 2. Ejemplo de patrón 1.

3.2 Patrón 2: Nombre de paquete

El término es empleado frecuentemente en clases incluidas en paquetes con un mismo nombre, véase la figura 3 que muestra los paquetes del software Eclipse EGit. Se define a M como el conjunto de nombres de paquetes; F(m) es el conjunto de paquetes con nombre $m \in M$; y F(m,t) es el conjunto de paquetes con nombre m que contienen clases que usan el término t. Los términos t de este patrón 2 son tales que $(| F(m,t) | / | F(m) |) \geq f$, y $| F(m) | > 2$.

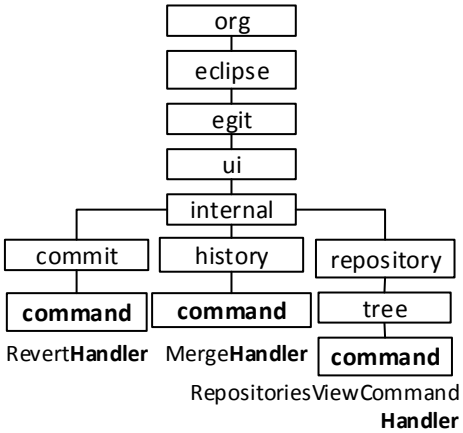

Figura 3. Ejemplo de patrón 2.

3.3 Patrón 3: Nombre y nivel de paquete

El término es empleado frecuentemente en clases incluidas en paquetes con un mismo nombre y a un mismo nivel de la jerarquía de paquetes, véase la figura 4 que muestra paquetes del software Apache Hadoop. Se define a N como el conjunto de niveles de paquetes; G(n,m) es el conjunto de paquetes con nombre m que se encuentran ubicados en el nível $n \in N$; y G(n,m,t) es el conjunto de paquetes de nível n con nombre m que contienen clases que

usan el término t. Los términos t de este tercer patrón son tales que (| G(n,m,t) | / | G(n,m) |) ≥ f, y | G(n,m) | > 2.

3.4 Patrón 4: Paquete superior inmediato

El término es empleado frecuentemente en clases incluidas en paquetes que pertenecen a un mismo paquete superior, véase la figura 5 que muestra paquetes del software Eclipse BPMN2. Se define a H(p) como el conjunto de paquetes contenidos en el paquete p; y H(p,t) es el conjunto de paquetes contenidos en p que contienen clases que usan el término t. Los términos t de este patrón son tales que (| H(p,t) | / | H(p) |) ≥ f, y | H(p) | > 2.

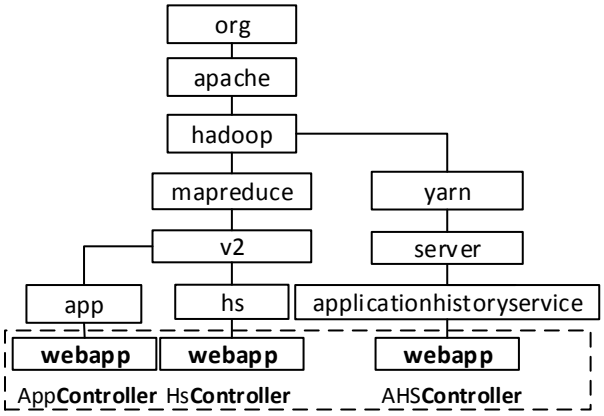

Figura 4. Ejemplo de patrón 3.

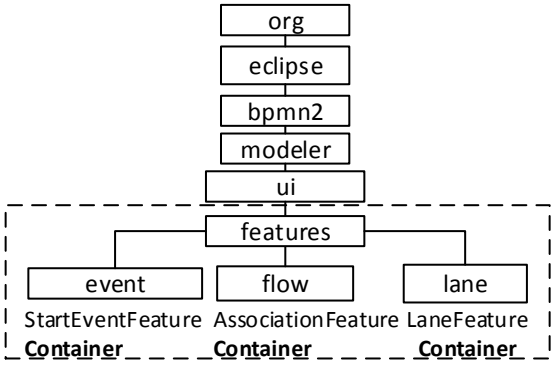

Figura 5. Ejemplo de patrón 4.

4. ANÁLISIS DE CÓDIGO FUENTE

Los pasos seguidos por el análisis propuesto son: lectura de paquetes y clases; creación de un grafo de términos, paquetes y clases; creación de nodos agregados; y consulta de términos frecuentes y su frecuencia en el grafo. Las siguientes subsecciones proporcionan mayor detalle de las características relevantes.

4.1 Modelo de almacenamiento basado en grafos

Los términos obtenidos son almacenados en una base de datos basada en grafos, tal modelo fue elegido por sus capacidades de visualización, y su facilidad en la adición de etiquetas a nodos y en la creación de nodos con data agregada. La figura 6 muestra un diagrama de clases en notación UML (i.e.; Unified Modeling Language) que bosqueja el modelo conceptual empleado, aquellos conceptos estereotipados como Agg constituyen tipos de nodos creados con data agregada. Seguido se da una descripción de los conceptos menos intuitivos, aquellos que inician con las letras TI en adelante se denominarán "nodos TI agregados".

- TerminoIndice: ocurrencia de un término en una determinada posición del nombre de clase.
- Nivel: nivel de la jerarquía de paquetes.
- TIPaquete: TerminoIndice en un paquete determinado. Útil para obtener términos en el patrón 1.
- TINombrePaquete: TerminoIndice en paquetes con un mismo nombre no calificado (e.g.; el nombre no calificado del paquete org::eclipse::birt es birt). Útil para obtener términos en el patrón 2.
- TINivelPaquete: TerminoIndice en paquetes a un mismo nivel de la jerarquía y con un mismo nombre no calificado. Útil para obtener términos en el patrón 3.
- TIPaquetePadre: TerminoIndice en paquetes que tienen un mismo paquete superior inmediato. Útil para obtener términos en el patrón 4.
- Tendencia: nodo TI agregado con una frecuencia determinada alta, mayor a cero (no aplica el patrón) y menor igual a 1 (cumple el patrón en todas sus ocurrencias). Constituye un término frecuente.

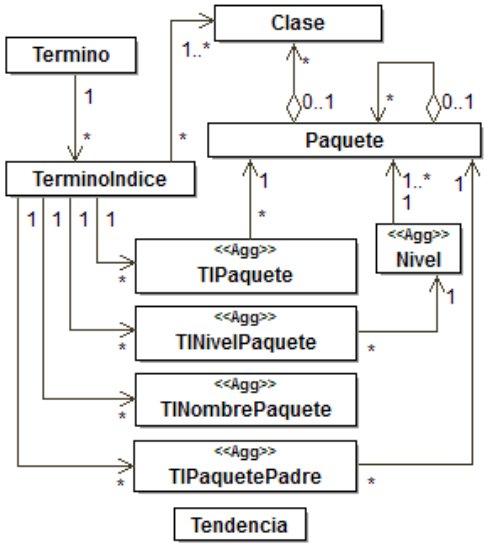

Figura 6. Modelo conceptual de nodos.

El realizar las consultas directamente sobre el código fuente sin utilizar un grafo, dificultaría la agregación de datos y podría reducir las ventajas de la misma. Asimismo, el contar con un grafo almacenado permite que éste sea consultado y visualizado gráficamente por los arquitectos de software para propósitos más allá de los mostrados en este trabajo.

4.2 CQL basado en lenguaje de consulta de grafos

Los CQL (i.e.; Code Query Language) han sido desarrollados para realizar un análisis exhaustivo del código fuente de aplicaciones de software [27]. En este trabajo, se emplea el lenguaje de consulta de grafos de la base de datos como CQL, ello a razón que los lenguajes de bases de datos cuentan con un conjunto extenso de mecanismos de consulta preparados para el manejo de considerables cantidades de datos, y en el caso de aquellos que manejan grafos, adicionalmente presentan capacidades de visualización gráfica de nodos y relaciones.

4.3 Análisis de frecuencia de términos

Se emplea un análisis basado en la frecuencia de términos con una amplitud de colección [32], frecuencia relacionada al número de veces que un término ocurre en una colección (e.g.; nombres de clases organizadas en paquetes). La frecuencia está determinada por el porcentaje de ocurrencias del término en diferentes clases dentro de un mismo paquete (patrón 1) o en diversos paquetes (patrones 2, 3, 4). Para cada ocurrencia se considera la posición del término dentro del nombre (e.g. para ClientProtocol, el término Protocol se encuentra en la segunda posición).

5. EVALUACIÓN Y RESULTADOS

El presente trabajo se puede considerar una propuesta válida para la DTA, por corresponder al ejemplar de no uniformidad de patrones de nombrado presentado en [2], y por atender una deuda que afecta la mantenibilidad o evolucionabilidad del software, sin encontrarse entre temas no aceptados ampliamente como deuda técnica [37]. El enfoque de esta propuesta brinda importancia a los nombres de clases, y éstos determinan la mantenibilidad y legibilidad de un software, entre otros [4] [7] [18] [28] [29] [33] [34]. De acuerdo al estándar ISO/IEC FDIS 25010, la mantenibilidad incluye los siguientes atributos de calidad: modularidad, reusabilidad, analizabilidad, modificabilidad y testabilidad. Considerando que el identificar componentes es una tarea menos complicada cuando los identificadores están conformados por términos significativos [9] [14], el análisis presentado puede favorecer la analizabilidad, y modificabilidad al obtener términos frecuentes que podrían ser relevantes en el sistema por su uso común (y ser representativos) y/o que podrían no ser significativos. Asimismo, si los patrones de nombrado presentados no son encontrados en una implementación de software, se podrían evidenciar elecciones pobres de diseño e implementación en cuanto a términos empleados, lo cual afecta a artefactos de casos de prueba [17]; en tal sentido el análisis también podría favorecer la testabilidad.

La tabla 1 muestra proyectos considerados en adelante, se tienen los siguientes datos para cada uno de ellos: LOC (cantidad de líneas de código), CA (cantidad de archivos), CP (cantidad de paquetes), CT (cantidad de términos).

Para la evaluación de la técnica de análisis propuesta, se implementó una aplicación que obtiene los términos utilizados en nombres de clases según el estilo de codificación CamelCase (estilo predominante por su facilidad de escritura y adopción [7] [13]), y almacena éstos en una base de datos Neo4j (base de datos de grafos estándar de la industria [26]). La aplicación ha sido utilizada sobre veinte proyectos de las organizaciones Apache y Eclipse (véase la tabla 1). El código fuente de los proyectos empleados fue obtenido de los repositorios de Apache y Eclipse en GitHub (https://github.com/). Algunos nombres de proyectos fueron simplificados para ser mostrados; sus nombres en los repositorios de GitHub son: eclipselink.runtime, hudson.core, scout.rt, servicemix-components.

La aplicación genera nodos TI agregados (véase la sección 4.2), y etiqueta como Tendencia aquellos con una frecuencia mayor o igual a 0.8 según cada patrón. En adelante se muestran los siguientes datos acerca de los términos frecuentes (i.e.; etiquetados como Tendencia) por proyecto para cada patrón: N (cantidad de términos frecuentes), Min (frecuencia mínima), Max (frecuencia máxima), Avg (frecuencia promedio), Stdv (desviación estándar de la frecuencia), TN (cantidad de términos frecuentes con una frecuencia menor a 1). Las tablas 2, 3, 4, y 5 muestran los datos de frecuencia mencionados para los patrones 1, 2, 3, y 4, respectivamente.

Tabla 1. Proyectos analizados

	Proyecto	LOC	CA	CP	CT
Apache	JMeter	156900	842	102	526
	Hadoop	860382	4246	404	1403
	MyFaces	161973	816	76	419
	Camel	543222	4070	518	1172
	OpenJPA	324139	1385	54	619
	Wicket	288225	1814	277	804
	ActiveMQ	315613	2122	151	656
	OpenEJB	432266	2758	204	1050
	Geronimo	133804	1087	120	621
	ServiceMix	91837	621	132	314
Eclipse	Birt	1883941	7743	746	1384
	Egit	137718	775	78	402
	BPMN2	190873	1109	96	408
	Scout	415805	3021	691	812
	Xtext	396344	2699	360	1011
	OSEE	593489	6141	815	1496
	EclipseLink	890456	3643	324	994
	Hudson	146540	904	83	687
	EMF	478261	1228	179	475
	Jetty	259521	1315	151	639

El patrón 1 es el patrón más empleado del conjunto; y el patrón 3, el patrón más restrictivo, es el menos empleado. La cantidad de proyectos que no presentaron ocurrencias para alguno de los patrones es ínfima. En general, los términos frecuentes tienen un cumplimiento del patrón en más del 90% de ocurrencias (i.e.; valor Avg de 0.9), presentándose casos al 100%.

Los valores de TN mostrarían la cantidad de incidencias de DTA por no uniformidad de patrones; y el porcentaje que representa TN respecto a N mostraría el porcentaje de términos frecuentes que no fueron uniformemente aplicados. Se sugiere que el valor aceptable de este último porcentaje sea definido por el equipo de desarrollo de acuerdo al grado de uso de convenciones de nombrado, y a cuán bien definida se encuentre la arquitectura del software.

A fin de mostrar la simplicidad en las consultas, producto del modelo de la figura 6, se presenta la siguiente consulta en lenguaje Cypher, que permite obtener términos frecuentes junto a sus paquetes correspondientes para el patrón 1. La figura 7 muestra un fragmento de los nodos obtenidos.

```
MATCH(t:Tendencia:TIPaquete),(p:Paquete
{fullName:t.packageFullName}) RETURN t,p
```

Como se puede ver en la figura 7, los resultados de una consulta pueden ser presentados en forma gráfica mediante grafos,

mostrando las relaciones entre los diferentes tipos de nodos involucrados (e.g.; paquetes en gris, términos en blanco).

Tabla 2. Frecuencia de términos para el patrón 1

Proyecto	N	Min	Max	Avg	Stdv	TN
JMeter	15	0.8	1	0.930	0.086	9
Hadoop	72	0.8	1	0.972	0.057	25
MyFaces	17	0.8	1	0.956	0.068	8
Camel	188	0.8	1	0.970	0.062	51
OpenJPA	9	0.8	1	0.914	0.096	5
Wicket	54	0.8	1	0.941	0.082	25
ActiveMQ	34	0.8	1	0.959	0.065	13
OpenEJB	28	0.8	1	0.942	0.077	12
Geronimo	16	0.8	1	0.921	0.078	9
ServiceMix	31	0.8	1	0.960	0.077	10
Birt	90	0.8	1	0.946	0.074	51
EGit	11	0.8	1	0.933	0.080	7
BPMN2	21	0.8	1	0.909	0.079	18
Scout	71	0.8	1	0.945	0.083	26
Xtext	36	0.8	1	0.951	0.074	16
OSEE	125	0.8	1	0.937	0.079	70
EclipseLink	71	0.8	1	0.958	0.066	32
Hudson	18	0.8	1	0.981	0.056	2
EMF	35	0.8	1	0.946	0.075	23
Jetty	34	0.8	1	0.954	0.071	13

Las convenciones de código pueden presentarse mayormente en forma de prácticas comunes que siguen cierto consenso, antes que en forma de reglas impuestas [19]. El análisis propuesto permite identificar un consenso en el uso frecuente de un término según los patrones presentados. Puesto que acciones de refactorización también pueden introducir elecciones pobres de diseño e implementación, el evidenciar tal consenso emergente en el código fuente, es útil antes de realizar refactorización [2] [19].

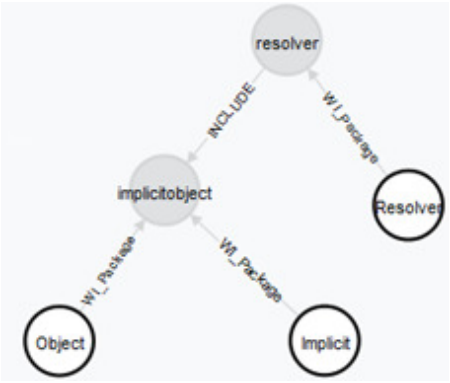

Figura 7. Nodos de términos frecuentes y paquetes.

En la tabla 6 se muestran algunos términos que son frecuentes pero no cumplidos al 100% de sus ocurrencias (i.e.; contados como TN en las tablas 2, 3, 4 y 5, tienen una frecuencia menor a 1). Nótese que varios de estos términos coinciden con conceptos del dominio de cada proyecto y con conceptos empleados en diseños y arquitecturas populares; mostrándose que es posible obtener conceptos emergentes y significativos del código fuente. La siguiente consulta obtiene los términos de TN para todos los patrones.

```
MATCH (t:Tendencia) WHERE t.percentage < 1
RETURN DISTINCT t.word
```

Tabla 3. Frecuencia de términos para el patrón 2

Proyecto	N	Min	Max	Avg	Stdv	TN
JMeter	2	0.938	1	0.969	0.044	1
Hadoop	34	0.800	1	0.965	0.073	7
MyFaces	2	1.000	1	1.000	0.000	0
Camel	21	0.800	1	0.949	0.085	8
OpenJPA	6	1.000	1	1.000	0.000	0
Wicket	9	1.000	1	1.000	0.000	0
ActiveMQ	1	1.000	1	1.000	0.000	0
OpenEJB	1	0.800	1	0.900	0.141	1
Geronimo	0	0.000	0	0.000	0.000	0
ServiceMix	10	0.800	1	0.911	0.090	5
Birt	34	0.800	1	0.951	0.081	11
EGit	5	0.800	1	0.960	0.089	1
BPMN2	3	1.000	1	1.000	0.000	0
Scout	59	0.800	1	0.935	0.089	26
Xtext	10	0.833	1	0.933	0.086	4
OSEE	41	0.800	1	0.963	0.072	10
EclipseLink	17	0.800	1	0.980	0.060	2
Hudson	1	1.000	1	1.000	0.000	0
EMF	17	0.857	1	0.960	0.056	7
Jetty	4	0.889	1	0.944	0.064	2

Se han buscado trabajos similares al propuesto en las bibliotecas digitales ACM, IEEE Xplore y ScienceDirect; las consultas empleadas para cada biblioteca se muestran a continuación.

Para ACM:

```
recordAbstract:(+"technical debt" name names
naming identifier identifiers)
```

Para IEEE Xplore:

```
("Abstract":technical debt) AND
("Abstract":name OR "Abstract":names OR
"Abstract":naming OR "Abstract":identifier
OR "Abstract":identifiers)
```

Para ScienceDirect:

```
ABS("technical debt") AND (ABS(name) OR ABS(names) OR ABS(naming) OR ABS(identifier) OR ABS(identifiers))
```

Tabla 4. Frecuencia de términos para el patrón 3

Proyecto	N	Min	Max	Avg	Stdv	TN
JMeter	2	0.857	1	0.952	0.082	1
Hadoop	17	0.800	1	0.945	0.082	6
MyFaces	0	0.000	0	0.000	0.000	0
Camel	18	0.833	1	0.991	0.038	1
OpenJPA	0	0.000	0	0.000	0.000	0
Wicket	2	1.000	1	1.000	0.000	0
ActiveMQ	0	0.000	0	0.000	0.000	0
OpenEJB	0	0.000	0	0.000	0.000	0
Geronimo	0	0.000	0	0.000	0.000	0
ServiceMix	9	0.875	1	0.963	0.060	2
Birt	18	0.833	1	0.995	0.029	1
EGit	5	1.000	1	1.000	0.000	0
BPMN2	2	1.000	1	1.000	0.000	0
Scout	63	0.800	1	0.977	0.063	8
Xtext	2	0.800	0.8	0.800	0.000	2
OSEE	20	0.800	1	0.974	0.069	3
EclipseLink	1	1.000	1	1.000	0.000	0
Hudson	1	1.000	1	1.000	0.000	0
EMF	17	0.833	1	0.967	0.063	6
Jetty	4	1.000	1	1.000	0.000	0

de implementación, se diferencia de otros enfoques que emplean el contenido del código fuente (e.g.; métodos, sentencias) para identificar deuda técnica.

Tabla 5. Frecuencia de términos para el patrón 4

Proyecto	N	Min	Max	Avg	Stdv	TN
JMeter	3	1.000	1	1.000	0.000	0
Hadoop	6	0.800	1	0.967	0.082	1
MyFaces	0	0.000	0	0.000	0.000	0
Camel	26	0.800	1	0.954	0.075	10
OpenJPA	0	0.000	0	0.000	0.000	0
Wicket	2	0.896	1	0.965	0.060	1
ActiveMQ	60	0.800	1	0.915	0.037	57
OpenEJB	2	0.875	0.889	0.882	0.008	2
Geronimo	3	0.857	1	0.952	0.082	1
ServiceMix	6	0.906	1	0.974	0.042	2
Birt	41	0.800	1	0.974	0.059	8
EGit	1	1.000	1	1.000	0.000	0
BPMN2	5	0.800	1	0.922	0.078	4
Scout	20	0.800	1	0.950	0.075	9
Xtext	7	0.800	1	0.919	0.089	4
OSEE	29	0.800	1	0.952	0.084	9
EclipseLink	33	0.800	1	0.977	0.062	7
Hudson	0	0.000	0	0.000	0.000	0
EMF	16	0.800	1	0.947	0.085	3
Jetty	5	0.833	1	0.900	0.091	3

La cantidad de resultados obtenidos para ACM, IEEE Xplore y ScienceDirect es de 64, 0, y 1, respectivamente. El resultado en ScienceDirect es un capítulo de un libro que presenta consejos para aplicar refactorización. Los resultados obtenidos de ACM en su mayoría son estudios del alcance, causas, impacto y características de la deuda técnica; unos cuantos levemente relacionados al presente son referidos a análisis estático sobre código fuente, pero inspeccionando métodos y sentencias de código. En consecuencia, se puede afirmar que no se encontraron propuestas enfocadas en el nombrado de artefactos de implementación, similares a este trabajo. Luego, siendo este trabajo una propuesta particular para la DTA de no uniformidad de patrones (véase sección 2), éste puede ser utilizado junto a técnicas enfocadas a otros ejemplares de DTA.

6. CONCLUSIONES

Los patrones de nombrado mostrados presentaron ocurrencias frecuentes en diversos proyectos de las organizaciones Apache y Eclipse, mostrándose que los términos frecuentes por lo general cumplían cada patrón en más del 90% de sus ocurrencias.

El análisis propuesto identifica deuda técnica arquitectónica por no uniformidad de patrones de nombrado; los cuales son aplicados frecuentemente, pero no son seguidos en toda parte del sistema. El enfoque utilizado, basado en patrones de nombrado de artefactos de implementación, se diferencia de otros enfoques que emplean el contenido del código fuente (e.g.; métodos, sentencias) para identificar deuda técnica.

El uso de la base de datos basada en grafos ha sido de relevancia, para independizar el análisis realizado de las limitaciones que podría presentar una herramienta CQL convencional [27]; realizándose agregación de data incorporada en nuevos nodos, y permitiéndose facilitar la elaboración de consultas, que en un CQL convencional pudieron ser más complejas, no posibles de realizar, o consumir un tiempo de respuesta excesivamente extenso.

La propuesta presentada es aplicable bajo un enfoque de desarrollo ágil, donde se promueve enfocarse en características del producto y llevar un cuidado acerca de la incertidumbre respecto a DTA [2]. El análisis realizado sobre código fuente no requiere como entrada un documento de arquitectura, y es automatizable mediante la ejecución de consultas en forma continua durante el desarrollo de software, permitiendo llevar un seguimiento de la DTA. Asimismo, los términos frecuentes descubiertos pueden ser de utilidad para identificar y establecer nuevos conceptos emergentes para la arquitectura del software.

7. TRABAJO FUTURO

Se tiene como trabajo futuro definir métricas de arquitectura de software medibles a partir del código fuente, los datos empleados en el presente trabajo serán empleados para el análisis de datos. Tal trabajo será desarrollado dentro del marco del proyecto

ProCal-ProSer, proyecto bajo el Contrato 210-FINCYT-IA-2013 y parcialmente soportado por el Departamento de Ingeniería de la Pontificia Universidad Católica de Perú.

Tabla 6. Términos frecuentes según patrones de nombrado

Proyecto	Términos
JMeter	Controller, Converter, Editor, Gui, JDBC, Meter.
Hadoop	Chain, Client, Container, Event, Scheduler.
MyFaces	Handler, Html, Impl, Implicit, Renderer, Tag.
Camel	Bean, Cache, Command, Filter, Task, Yammer.
OpenJPA	Concurrent, Distributed, Identifier, Managed.
Wicket	Bean, Checker, Handler, Resolver, Socket.
ActiveMQ	Adapter, Bridge, Broker, Command, Factory.
OpenEJB	Binding, Command, Entity, Factory, Thread.
Geronimo	Command, Deployment, Manager, Validation.
ServiceMix	Component, Factory, Filter, Interceptor, Ws
Birt	Action, Adapter, Filter, Handler, Validator.
EGit	Blame, Command, Git, Handler, Index, Node
BPMN2	Adapter, Editor, Event, Flow, Task, Validator.
Scout	Activity, Browser, Inspector, Job, Page, Service,
Xtext	Facet, Fragment, Module, Page, Resource, Ui
OSEE	Action, Command, Client, Service, Word.
EclipseLink	Accesor, Converter, Query, Resource, Table.
Hudson	Team, X
EMF	Action, Adapter, Command, Factory, Model.
Jetty	Bean, M, Response, Socket, Web

8. REFERENCIAS

[1] Anas Mahmoud and Nan Niu. 2014. Supporting requirements to code traceability through refactoring. *Requir. Eng.* 19, 3 (September 2014), 309-329. DOI=http://dx.doi.org/10.1007/s00766-013-0197-0

[2] Antonio Martini, Jan Bosch, and Michel Chaudron. 2015. Investigating Architectural Technical Debt accumulation and refactoring over time. *Inf. Softw. Technol.* 67, C (November 2015), 237-253. DOI=http://dx.doi.org/10.1016/j.infsof.2015.07.005

[3] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. 2015. The financial aspect of managing technical debt. *Inf. Softw. Technol.* 64, C (August 2015), 52-73. DOI=http://dx.doi.org/10.1016/j.infsof.2015.04.001

[4] Ben Liblit, Andrew Begel and Eve Sweetser. 2006. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*.

[5] Carolyn Seaman, Yuepu Guo, Clemente Izurieta, Yuanfang Cai, Nico Zazworka, Forrest Shull, and Antonio Vetrò. 2012. Using technical debt data in decision making: potential decision approaches. In *Proceedings of the Third International Workshop on Managing Technical Debt* (MTD '12). IEEE Press, Piscataway, NJ, USA, 45-48.

[6] Chris Sterling. 2010. *Managing Software Debt: Building for Inevitable Change* (1st ed.). Addison-Wesley Professional.

[7] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Softw. Engg.* 18, 2 (April 2013), 219-276. DOI=http://dx.doi.org/10.1007/s10664-012-9201-4

[8] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2012. How We Refactor, and How We Know It. *IEEE Trans. Softw. Eng.* 38, 1 (January 2012), 5-18. DOI=http://dx.doi.org/10.1109/TSE.2011.41

[9] Florian Deissenboeck and Markus Pizka. 2005. Concise and Consistent Naming. In *Proceedings of the 13th International Workshop on Program Comprehension* (IWPC '05). IEEE Computer Society, Washington, DC, USA, 97-106. DOI=10.1109/WPC.2005.14 http://dx.doi.org/10.1109/WPC.2005.14

[10] Gustavo Soares, Rohit Gheyi, Emerson Murphy-Hill, and Brittany Johnson. 2013. Comparing approaches to analyze refactoring activity on software repositories. *J. Syst. Softw.* 86, 4 (April 2013), 1006-1022. DOI=http://dx.doi.org/10.1016/j.jss.2012.10.040

[11] Ivan Mistrik, Rami Bahsoon, Rick Kazman, and Yuanyuan Zhang. 2014. *Economics-Driven Software Architecture* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Kent Beck. 2008. *Implementation patterns*. Addison Wesley.

[13] Latifa Guerrouj. 2013. Normalizing source code vocabulary to support program comprehension and software quality. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 1385-1388.

[14] Latifa Guerrouj, Massimiliano Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2014. An experimental investigation on the effects of context on source code identifiers splitting and expansion. Empirical Softw. Engg. 19, 6 (December 2014), 1706-1753. DOI=10.1007/s10664-013-9260-1 http://dx.doi.org/10.1007/s10664-013-9260-1

[15] Lianping Chen and Muhammad Ali Babar. 2014. Towards an Evidence-Based Understanding of Emergence of Architecture through Continuous Refactoring in Agile Software Development. In *Proceedings of the 2014 IEEE/IFIP Conference on Software Architecture* (WICSA '14). IEEE Computer Society, Washington, DC, USA, 195-204. DOI=10.1109/WICSA.2014.45 http://dx.doi.org/10.1109/WICSA.2014.45

[16] Marija Katić and Krešimir Fertalj. 2009. Towards an appropriate software refactoring tool support. In *Proceedings of the 9th WSEAS international conference on Applied computer science* (ACS'09), Roberto Revetria, Valeri Mladenov, and Nikos Mastorakis (Eds.). World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 140-145.

[17] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to

smell bad. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (ICSE '15), Vol. 1. IEEE Press, Piscataway, NJ, USA, 403-414.

[18] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, New York, NY, USA, 38-49. DOI=10.1145/2786805.2786849 http://doi.acm.org/10.1145/2786805.2786849

[19] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). ACM, New York, NY, USA, 281-293. DOI=10.1145/2635868.2635883 http://doi.acm.org/10.1145/2635868.2635883

[20] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, USA, 233-243.

[21] Muhammad Ali Babar, Alan W. Brown, and Ivan Mistrik. 2013. *Agile Software Architecture: Aligning Agile Processes and Software Architectures* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[22] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. Measure it? Manage it? Ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 2015). ACM, New York, NY, USA, 50-60. DOI=http://dx.doi.org/10.1145/2786805.2786848

[23] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt* (MTD '11). ACM, New York, NY, USA, 17-23. DOI=http://dx.doi.org/10.1145/1985362.1985366

[24] Nicolli Alves, Thiago Mendes, Manoel de Mendonça, Rodrigo Spínola, Forrest Shull, and Carolyn Seaman. 2016. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology.* 70, 100-121.

[25] Nikolaos Tsantalis, Victor Guana, Eleni Stroulia, and Abram Hindle. 2013. A multidimensional empirical study on refactoring activity. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research* (CASCON '13). IBM Corp., Riverton, NJ, USA, 132-146.

[26] Peter Macko, Daniel Margo, and Margo Seltzer. 2013. Performance introspection of graph databases. In *Proceedings of the 6th International Systems and Storage Conference* (SYSTOR '13). ACM, New York, NY, USA, , Article 18 , 10 pages. DOI=10.1145/2485732.2485750 http://doi.acm.org/10.1145/2485732.2485750

[27] Raoul-Gabriel Urma and Alan Mycroft. 2012. Programming language evolution via source code query languages. In *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools* (PLATEAU '12). ACM, New York, NY, USA, 35-38. DOI=http://dx.doi.org/10.1145/2414721.2414728

[28] Robert C. Martin. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship* (1 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[29] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering* (CSMR '10). IEEE Computer Society, Washington, DC, USA, 156-165. DOI=10.1109/CSMR.2010.27 http://dx.doi.org/10.1109/CSMR.2010.27

[30] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E. Johnson, and Danny Dig. 2013. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European conference on Object-Oriented Programming* (ECOOP'13), Giuseppe Castagna (Ed.). Springer-Verlag, Berlin, Heidelberg, 552-576. DOI=http://dx.doi.org/10.1007/978-3-642-39038-8_23

[31] Steve McConnell. 2004. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.

[32] Thomas Roelleke. 2013. *Information Retrieval Models: Foundations and Relationships* (1st ed.). Morgan & Claypool Publishers.

[33] Venera Arnaoudova, Massimiliano Di Penta and Giuliano Antoniol. 2015. Linguistic antipatterns: What they are and how developers perceive them. Empirical Software Engineering. 1-55.

[34] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. 2014. On the Comprehension of Program Comprehension. *ACM Trans. Softw. Eng. Methodol.* 23, 4, Article 31 (September 2014), 37 pages. DOI=http://dx.doi.org/10.1145/2622669

[35] Zadia Codabux, Byron J. Williams and Nan Niu. 2014. In Proceedings of the International Conference on Software Engineering Research and Practice (SERP'14).

[36] Zengyang Li, Peng Liang, Paris Avgeriou, Nicolas Guelfi, and Apostolos Ampatzoglou. 2014. An empirical investigation of modularity metrics for indicating architectural technical debt. In *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures* (QoSA '14). ACM, New York, NY, USA, 119-128. DOI=http://dx.doi.org/10.1145/2602576.2602581

[37] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *J. Syst. Softw.* 101, C (March 2015), 193-220. DOI=10.1016/j.jss.2014.12.027 http://dx.doi.org/10.1016/j.jss.2014.12.027