A Tool To Prioritize Code Smell In Distributed Development

H. Vázquez, C. Marcos, S. Vidal and J. A. Diaz Pace

***Abstract*— A code smell is a symptom in the source code that helps to identify a design problem. Several tools for detecting and ranking code smells according to their criticality to the system have been developed. However, existing works assume a centralized development approach, which does not consider systems being developed in a distributed fashion. The main problem in a distributed group of developers is that a tool cannot always ensure a global vision of (smells of) the system, and thus inconsistencies among the rankings provided by each developer are likely to happen. These inconsistencies often cause unnecessary refactorings and might not focus the whole team on the critical smells system-wide. Along this line, this work proposes a multi-agent tool, called D-JSpIRIT, which helps individual developers to reach a consensus on their smell rankings by means of distributed optimization techniques.**

***Keywords*— Code Smells, Prioritization Criteria, Distributed Development, Tool Support, Multi-agent System.**

I. INTRODUCCIÓN

DURANTE la evolución de un sistema de software es común que los desarrolladores tomen decisiones que, si bien están orientadas a entregar funcionalidad importante para el negocio, pueden en contrapartida comprometer el diseño estructural del sistema y dificultar su mantenimiento. Esta situación se explica generalmente con la metáfora de *deuda técnica*. En esta metáfora, las malas decisiones de diseño o el código de baja calidad, cuya mejora es postergada por los desarrolladores, se considera una deuda [1]. Ante una situación de deuda técnica, el desarrollador debe realizar un "pago de intereses", lo cual normalmente implica la refactorización de las partes afectadas del sistema.

Una fuente conocida de deuda técnica son los *code smells* (también denominados anomalías de código) [2]. Un code smell es un síntoma en el código fuente que ayuda a identificar un problema de diseño. De esta manera, la identificación de smells permite a los desarrolladores detectar fragmentos de código que deben ser re-estructurados. Esta re-estructuración se implementa generalmente mediante refactorizaciones [2]. En la práctica, solo es posible refactorizar un conjunto limitado de smells, mayormente debido a restricciones de tiempo, recursos y complejidad. No obstante, la identificación y priorización de code smells es una tarea que plantea desafíos [3]. Por esta razón, diferentes herramientas se han desarrollado para asistir a los desarrolladores en dichas tareas [3] [4] [5] [6].

Una herramienta típica de detección de smells realiza inicialmente un análisis del código fuente del proyecto, luego ordena las instancias de smells encontradas (con algún criterio) y presenta la lista resultante (o ranking) al desarrollador. Por ejemplo, JSpIRIT [3] aplica un análisis estático del código y luego en base a criterios heurísticos, puede recomendar al desarrollador smells "críticos" a refactorizar (por ej., *GodClass*, *IntensiveCoupling*). En los trabajos encontrados en la literatura, este tipo de sistemas de recomendación asumen una modalidad de trabajo *centralizada*, es decir, un único desarrollador (o grupo de desarrolladores) ejecuta la herramienta sobre todo el código fuente. Sin embargo, hoy en día, un número creciente de sistemas se desarrollan en forma *distribuida*, donde los desarrolladores trabajan cada uno en forma remota sobre porciones del sistema [7]. En este escenario, si bien es posible utilizar herramientas como JSpIRIT (en cada sitio remoto), existe el problema que el ranking de smells que obtiene un desarrollador puede tener inconsistencias con el ranking de smells que puede obtener otro desarrollador, debido a que ambos no siempre poseen una visión global del sistema. Estas inconsistencias en los rankings llevan a análisis incorrectos sobre la criticidad de ciertos smells, y a esfuerzos duplicados de refactorización, entre otros.

Ciertamente, la problemática descripta anteriormente puede ser resuelta mediante reuniones de coordinación entre los desarrolladores involucrados, aunque generalmente esto agrega esfuerzos extras al proyecto para gestionar dichas reuniones. Un enfoque alternativo para minimizar reuniones consiste en brindar a cada desarrollador, en su sitio remoto, información adicional para que pueda analizar el ranking de smells desde una perspectiva más amplia, teniendo en cuenta a los desarrolladores afectados potencialmente por algunos de sus smells. En otras palabras, se busca coordinar las distintas instancias de la herramienta de smells (corriendo en cada sitio remoto) respecto al ranking de smells que cada instancia provee a cada desarrollador, y obtener así un ranking global y consistente de smells.

En esta línea, se presenta un enfoque basado en JSpIRIT, llamado D-JSpIRIT, que modela el problema en términos de un sistema multi-agente [8], donde los agentes asisten a un grupo de desarrolladores remotos a lograr consenso sobre el ranking de smells de mayor relevancia global para el sistema. Para lograr dicho consenso, se aplican técnicas de satisfacción distribuida de restricciones (DisCSP) [9]. Los resultados de experimentos controlados con un caso de estudio han mostrado la utilidad del enfoque D-JSpIRIT.

El resto de este artículo está estructurado de la siguiente manera. En la sección II se introduce JSpIRIT como un sistema

H. Vázquez, ISISTAN-CONICET, Argentina, hvazquez@exa.unicen.edu.ar
C. Marcos, ISISTAN-CIC, Argentina, cmarcos@exa.unicen.edu.ar
S. Vidal, ISISTAN-CONICET, Argentina, svidal@exa.unicen.edu.ar
J. A. Diaz Pace, ISISTAN-CONICET, Argentina, adiaz@exa.unicen.edu.ar

originalmente diseñado para identificar smells prioritarios, y se discuten trabajos relacionados. La sección III describe la solución multi-agente propuesta en D-JSpIRIT, e ilustra el funcionamiento del proceso distribuido de consenso. En la sección IV se brindan detalles de la adaptación de DisCSP a la problemática de consistencia de rankings de smells. La sección V presenta un caso de estudio para evaluar la herramienta. Finalmente, la sección VI da las conclusiones del trabajo y analiza trabajos futuros.

II. CONTEXTO

En la actualidad, el desarrollo de software distribuido es una modalidad muy empleada en la industria, mayormente debido a razones de índole económica [7]. En un entorno de desarrollo en el que los miembros del equipo están dispersos geográficamente, es necesario repensar la forma en la que se lleva a cabo el proceso de desarrollo. Sin bien las herramientas de software colaborativo ofrecen facilidades para gestionar un desarrollo distribuido, existen tareas que aún plantean desafíos. En particular, la identificación y resolución de problemas que afectan el diseño estructural "compartido" del sistema, es decir, su *arquitectura de software* [10], requiere un cierto consenso en el equipo de desarrollo (y también con el cliente). Esto se debe a que cada desarrollador, al trabajar en forma remota, puede tener una visión diferente de las prioridades y necesidades de la aplicación, respecto a un conjunto de problemas.

Este trabajo se enfoca en el análisis de code smells en un desarrollo distribuido, con el objetivo de hallar problemas de diseño estructural en un sistema. Dicho análisis puede ser automatizado utilizando distintas herramientas. Sin embargo, el número de smells reportado por las herramientas actuales suele exceder la cantidad de problemas que un desarrollador puede tratar, en particular cuando el tiempo asignado (en un proyecto) para realizar las correspondientes refactorizaciones es limitado. Por otra parte, no todos los smells son igualmente relevantes para los objetivos del negocio o la salud del sistema. Para clarificar los criterios de priorización de smells (y su interpretación subjetiva por parte del desarrollador a cargo), se pueden considerar los siguientes ejemplos:

- Una instancia de *GodClass* en un paquete Java que no ha cambiado a lo largo de varias versiones del sistema puede no ser tan dañino para el sistema como otra instancia de *GodClass* que afecta los paquetes responsables de la lógica del negocio, que cambia frecuentemente debido a requerimientos del cliente. En dicho caso, el segundo *GodClass* sería más prioritario (o urgente) para su refactorización.
- Las instancias de *IntensiveCoupling* en un sub-sistema debieran ser refactorizadas, si el objetivo del desarrollador es lograr que dicho sub-sistema sea más cohesivo (y con un menor acoplamiento con otras clases del sistema). De ser así, dichas instancias irían primeras en el orden de prioridad. Sin embargo, esta reducción del acoplamiento entre clases puede también incrementar la complejidad de ciertas clases proveedoras de funcionalidad.

JSpIRIT como herramienta de priorización de smells

El enfoque JSpIRIT y su herramienta asociada [3, 11] fueron desarrollados para atender las necesidades de priorización de code smells, incorporando criterios de priorización basados en: i) tipo de smell, ii) escenarios de cambio requeridos por la aplicación, e iii) historial de cambios de la aplicación. El enfoque mono-usuario (y centralizado) de JSpIRIT se muestra en la Fig. 1. La evaluación de JSpIRIT en distintos proyectos Java ha mostrado su efectividad para recomendar una lista acotada de smells críticos (o relevantes) de un sistema. La obtención de una buena priorización depende de la provisión de cierta información para parametrizar los tres criterios antes mencionados. Por ejemplo, el desarrollador debe determinar qué tipos de smells son más importantes para su análisis.

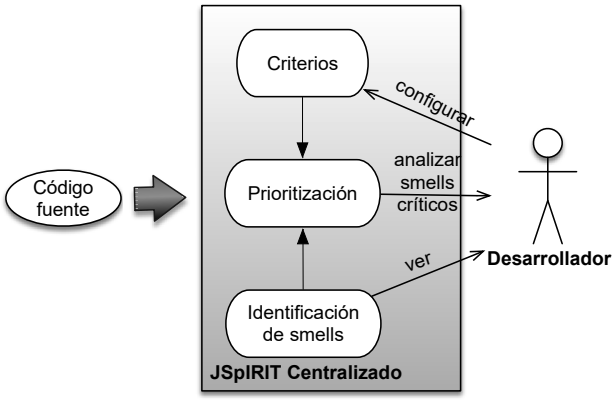

Figura 1. JSpIRIT para un ambiente de desarrollo mono-usuario.

En un desarrollo distribuido utilizando JSpIRIT, la configuración de los criterios de priorización de smells es un aspecto que toma mayor relevancia, debido a que pueden existir múltiples interpretaciones por el hecho de contar con desarrolladores remotos. Por ejemplo, el desarrollador A puede juzgar que los *GodClass* de sus porciones de código son de moderada relevancia, respecto a los *IntensiveCoupling* que juzga con mayor relevancia; mientras que el desarrollador B puede considerar que los *GodClass* de sus porciones de código (solapadas con código que comparte con el desarrollador A) tienen alta relevancia. En consecuencia, es necesario extender JSpIRIT para gestionar dichas diferencias y tender a consensuar los distintos criterios de smells. En la Sección IV se discute un enfoque general para un ambiente distribuido, y se detalla la solución particular de D-JSpIRIT para coordinar el criterio basado en tipo de smells entre desarrolladores remotos.

Trabajos Relacionados

No se conocen hasta el momento trabajos que busquen priorizar code smells en ambientes distribuidos - aunque si existen aplicaciones de DisCSP a ambientes distribuidos. No obstante existen varios trabajos sobre identificación y priorización de smells en sistemas centralizados.

Lanza y Marinescu [12] presentan un catálogo de code smells. Para cada smell, los autores definen una estrategia de identificación basada en un conjunto de métricas que deben superar valores umbrales preestablecidos. Si bien no se define

un proceso formal para la priorización de los smells, los autores sugieren refactorizar primero las clases que contienen un mayor número de smells.

Marinescu [13] propone establecer la criticidad de un smell basado en 3 factores: (i) la influencia negativa en la arquitectura según el tipo de smell, (ii) el tipo de entidad afectada por el smell (por ej., clase o método), y (iii) los valores de las métricas utilizadas para cada tipo de smell.

Arcoverde [4] presenta un enfoque para priorizar smells utilizando diferentes heurísticas. Estas heurísticas están basadas en características tales como: porcentaje de cambios en una entidad (por ej., paquete, o clase), número de bugs encontrados en una entidad durante su historia, cantidad de smells en una clase, o roles arquitectónicos de las clases.

Con respecto al uso de DisCSP en ambientes distribuidos, existen pocos trabajos que aplican estos algoritmos a desarrollo distribuido. Lin y Miao [14] utilizan DisCSP para balancear el trabajo realizado por desarrolladores distribuidos que usan Scrum. Bowring et al. [15] aplican DisCSP para acordar reuniones de un grupo distribuido. Herbsleb et al. [16] modelan la coordinación de decisiones en un proceso de desarrollo de software como un problema de satisfacción de restricciones distribuido.

IV. D-JSPIRIT: JSPIRIT DISTRIBUIDO

Nuestro ambiente de trabajo distribuido consiste de un grupo de desarrolladores, cada uno de ellos interactuando con un IDE que le permite acceder y manipular el código fuente (Java) del proyecto. Este código, generalmente, está almacenado en algún repositorio (por ej., SVN o Git). Por razones prácticas, cada desarrollador está asignado a (o es *owner* de) un subconjunto de artefactos (por ej., paquetes o clases del sistema), pudiendo existir solapamientos en estas asignaciones (*shared ownership*). La Fig. 2 muestra el esquema general de D-JSpIRIT. Cada IDE posee una instancia de JSpIRIT con el propósito de analizar los smells del subconjunto de artefactos de cada desarrollador. En este sentido, cada JSpIRIT puede verse como un *agente de asistencia* [8] al desarrollador, ya que lo ayuda a priorizar smells. El grupo de desarrollo se modela entonces como un sistema multi-agente [8], donde los agentes atienden los rankings de cada desarrollador individual, y buscan una solución consistente a nivel global. En caso de inconsistencia de rankings, cada agente puede aconsejarle a su respectivo desarrollador qué ajustes en sus criterios de priorización podrían resolver las inconsistencias. El desarrollador puede o no aceptar estas sugerencias.

A fin de clarificar la problemática de las posibles inconsistencias entre los rankings de los usuarios individuales, se retoma el ejemplo de dos desarrolladores trabajando en forma conjunta sobre el mismo proyecto, según se esquematiza en la Fig. 3. El desarrollador A tiene ownership sobre el conjunto A del código fuente (que incluye a las clases con smells CS1, CS2, CS4, CS5, CS6), y el desarrollador B sobre el conjunto B (que incluye a las clases con smells CS2, CS3, CS4, CS6, CS7, CS8). Existe un solapamiento entre ambos conjuntos, dado por las instancias de smells CS2, CS4 y CS6. Al ejecutar su agente de JSpIRIT, el desarrollador A obtiene el ranking de la derecha, y en forma análoga el desarrollador B obtiene el ranking de la izquierda (ejecutando su propio agente de JSpIRIT). Al observar los rankings obtenidos por A y B, puede notarse que los análisis de cada desarrollador poseen inconsistencias. Los rankings poseen *conflictos de orden*, ya que el desarrollador A considera que CS2 es más relevante que CS6, mientras que B considera lo contrario. Esto puede deberse, por ejemplo, a que CS2 es una instancia de *GodClass* y CS6 es una instancia de *DispersedCoupling*, y A otorga más prioridad a *GodClass* sobre *Dispersed* Coupling mientras que B prefiere una prioridad inversa.

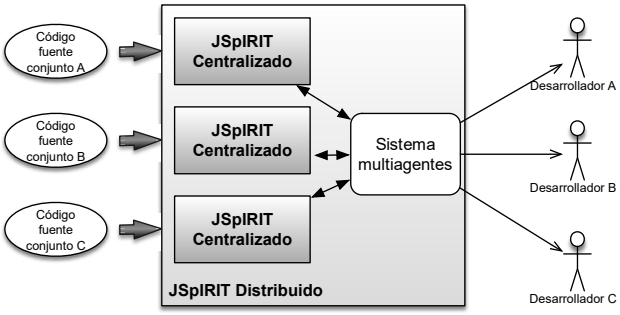

Figura 2. JSpIRIT para un desarrollo con múltiples usuarios.

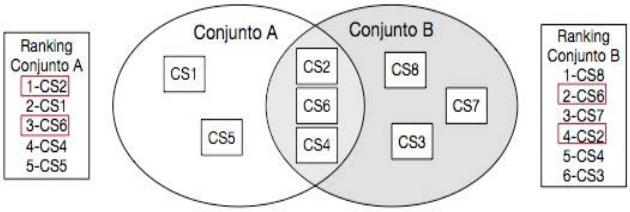

Figura 3. Inconsistencias en smells compartidos por desarrolladores.

El problema anterior puede manifestarse entre más de dos desarrolladores, e involucrar varios conflictos de orden entre distintos tipos de smells. En este punto, los respectivos agentes deben primeramente identificar las inconsistencias compartidas entre rankings y luego llevar a cabo una negociación de prioridades de tipos de smells. El proceso general (para N agentes) se describe en la Fig. 4. En el ejemplo, los agentes A y B inician una negociación y aplican un algoritmo de DisCSP para hacer que uno de los ellos re-considere su orden de prioridad.

Modelado de consistencia de rankings como DisCSP

El problema de controlar las inconsistencias en el ranking de code smells puede ser visto como un problema de satisfacción de restricciones (CSP) [9], particularmente, de restricciones de orden entre los rankings de distintos desarrolladores. En un CSP distribuido o DisCSP, las variables y restricciones se distribuyen entre los distintos agentes de un sistema multi-agente [8], y cada agente busca posibles valores para sus variables de manera que se satisfagan todas las restricciones del problema.

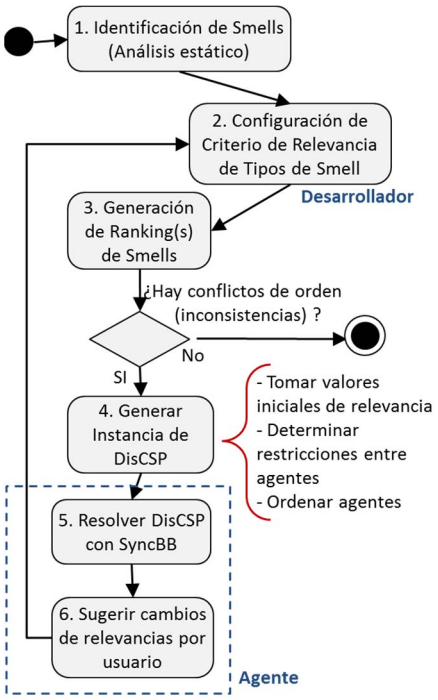

Figura 4. Proceso de coordinación de rankings entre agentes y DisCSP.

Sea un proyecto P, conformado por instancias de code smells de la forma *<tipo de smell, elemento de código que lo contiene>*. Por ejemplo, una tupla puede ser *<GodClass, ClaseA>* para indicar que hay un *GodClass* en *ClaseA*. Sea una serie de sub-conjuntos $G \subseteq P$ (incluyendo posibles solapamientos entre estos sub-conjuntos). Sean m variables $C_1, C_2,..., C_m$ donde m es la cantidad de personas del proyecto y C_i representa el criterio de cada persona para armar su ranking de smells. En DisCSP, se define un agente asociado a cada variable C_i. En particular, en este trabajo C_i modela el criterio de *relevancia por tipo de smell*. Sea un vector $C_i = <r_1, r_2,..., r_n>$ donde n es la cantidad de tipos de smells y r_k representa la relevancia de cada tipo de smell, en una escala de valores enteros *[1..5]* (1 significa máxima relevancia del smell y 5 significa nula relevancia).

A partir de lo anterior, se define la función $ranking(G_k,C_k) = R_k$, donde R_k es el conjunto ordenado de instancias de smells a partir de las relevancias de C_k dadas por el desarrollador sobre el conjunto G_k del cual es owner. Si existen dos grupos G_i, G_j tal que $G_i \cap G_j \neq \varnothing$ (es decir, que comparten instancias de smells), se considera entonces una restricción $P_{i,j}(C_i, C_j)$ donde C_i es el criterio del agente i y C_j es el criterio del agente j. Esta restricción entre criterios determina la posibilidad de un conflicto entre variables DisCSP, y será verdadera si y solo si $\forall x \forall y \in G_i \cap G_j$ la relación de orden $(x, y) \in R_i$ es consistente con la relación de orden $(x, y) \in R_j$, donde $R_i = ranking(G_i,C_i)$ y $R_j = ranking(G_j,C_j)$. Dos relaciones de orden serán consistentes solamente cuando los smells compartidos por ambas se encuentren en el mismo orden (aunque no necesariamente en las mismas posiciones del ranking). Una restricción no cumplida constituye un conflicto de orden. Cada restricción $P_{i,j}$ se asume compartida entre los agentes i y j asociados a las variables correspondientes. En términos de DisCSP, cada agente k debe determinar una asignación de relevancias a su vector C_k de forma tal que todas las restricciones $P_{i,j}$ se satisfagan.

La Fig. 5 muestra una posible configuración DisCSP con tres agentes A, B y C. Además de la restricción entre A y B, derivada de la Fig. 3, se asume también una restricción entre los agentes B y C. En el caso de D-JSpIRIT, donde cada desarrollador inicialmente determina las relevancias de cada tipo de smells, esta asignación se considera como la asignación inicial de su agente respectivo. Luego, puede aplicarse cualquier algoritmo conocido para DisCSP, a fin de que los agentes intercambien mensajes hasta encontrar una solución satisfactoria o bien determinar que no existe solución al problema. En nuestro caso, se utilizó el algoritmo SyncBB (Synchronous Branch-and-bound) [17], que aplica una estrategia branch-and-bound en un entorno distribuido.

En SyncBB, se requiere fijar un orden predeterminado de los agentes, y luego estos agentes intercambian soluciones (asignaciones) parciales en forma ordenada, que son progresivamente extendidas hasta completar las asignaciones. En caso de violación de alguna restricción, el orden de los agentes determina qué agente debe cambiar su asignación por otra alternativa. Si un agente agota sus alternativas y aun así no se satisface alguna restricción, el problema termina en falla. En la Fig. 5 se muestra un posible orden de agentes (A → B → C) y una solución válida computada con SyncBB. En este caso, luego del intercambio de soluciones parciales entre los agentes, los agentes B y C acordaron modificar sus valores de relevancia para obtener un ranking global consistente. En particular, el agente B sugiere reducir la relevancia de God Class en 2 y aumentar la relevancia de Dispersed Coupling en 1, mientras que el agente C sugiere reducir la relevancia de Dispersed Coupling en 2 y aumentar la relevancia de Feature Envy en 1. Los desarrolladores A y B pueden o no considerar estas sugerencias de D-JSpIRIT y efectivamente actualizar las relevancias. En caso de que alguno de los desarrolladores no siga todas las sugerencias, D-JSpIRIT ejecutara nuevamente la negociación, según se muestra en la Fig. 4.

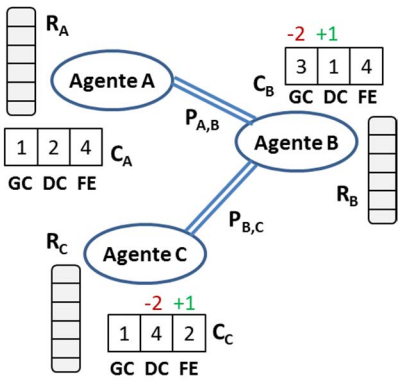

Figura 5. Agentes y problema desde la perspectiva DisCSP (En colores rojo y verde las recomendaciones al usuario para resolver los conflictos).

Asistencia de los agentes a cada desarrollador

Al ejecutar SyncBB, es necesario que cada agente: i) seleccione distintas relevancias de tipos de smells, y ii) evalúe

la satisfacción de las restricciones de orden. Respecto al primer punto, una variación en D-JSpIRIT es que cada agente debe tomar como punto de partida la configuración dada por cada desarrollador. En consecuencia, cuando SyncBB fuerza a un agente a cambiar su asignación, éste debe preferir (en caso de ser posible) un valor de variables que se aparte "lo menos posible" de dicha configuración.

Respecto al segundo punto, una vez que un agente ha realizado una asignación de relevancias, debe analizar posibles inconsistencias entre su ranking y los rankings de los agentes vecinos. Para ello, se aplica un análisis de ciclos como una restricción global del sistema multi-agente. Los rankings de smells son vistos como un grafo, donde cada nodo es un smell, y cada arista indica una relación de precedencia inmediata entre dos smells en alguno de los rankings de los agentes. En particular, se aplica el algoritmo de Tarjan [18] que permite encontrar ciclos en un grafo mediante la búsqueda de componentes fuertemente conexos. Si se encuentra un ciclo en el grafo, quiere decir que existe una inconsistencia entre los rankings de los agentes.

Una vez que cada agente ha determinado una asignación de relevancias que no produce inconsistencias, cada agente propone sus valores a su desarrollador asociado. El desarrollador puede adoptar estas relevancias, o incluso hacer otros cambios no sugeridos por el agente. Al cambiarse la configuración del criterio, se re-computan los rankings respectivos, y el proceso de aseguramiento de consistencia se re-inicia, según lo visto en la Fig. 4.

V. CASO DE ESTUDIO

En esta sección se presenta un caso de estudio con el objetivo de validar el enfoque propuesto. Para ello, se describen las preguntas e hipótesis de investigación. Adicionalmente, se describen la aplicación utilizada como caso de estudio, los procedimientos para obtener los datos y se discuten los resultados obtenidos.

Objetivos

El objetivo es analizar si el enfoque de D-JSpIRIT distribuido es capaz de generar rankings de smells consistentes en un tiempo comparable al de un grupo de desarrollo que realiza el mismo trabajo en forma manual. Utilizando el formato GQM [19], se pretende concretamente: *Analizar* el resultado de aplicar el enfoque de priorización de smells *con el objetivo* de evaluar su desempeño en tiempo y número de inconsistencias *con respecto a la* priorización manual de dichos smells *desde el punto de vista de* los investigadores y desarrolladores *en el contexto de* una aplicación Java.

Pregunta e hipótesis de investigación

La pregunta de investigación específica que será analizada en este estudio es la siguiente:

- **PI.** ¿El uso de D-JSpIRIT facilita a los desarrolladores llegar a un acuerdo sobre el ranking global de los code smells?

A partir de esta pregunta, se derivan 2 hipótesis para confirmar o refutar nuestras expectativas. La expectativa es que el enfoque facilitará la priorización de smells en términos de: (i) calidad de los rankings propuestos (i.e. rankings sin inconsistencias) y; (ii) tiempo invertido para lograr consensos en el ranking global. En este contexto, se define las siguientes hipótesis nulas junto con sus correspondientes hipótesis alternativas:

- $H_{1,0}$: No hay diferencia en el número de inconsistencias entre los rankings generados manualmente y los generados por el enfoque.
- $H_{1,1}$: El número de inconsistencias entre los rankings generados manualmente y los generados por el enfoque es diferente.
- $H_{2,0}$: No hay diferencia entre el tiempo empleado por los desarrolladores para crear el ranking manualmente o automáticamente con el enfoque.
- $H_{2,1}$: El tiempo empleado por los desarrolladores para crear el ranking manualmente o automáticamente con el enfoque es diferente.

Planificación del experimento

Con el objetivo de testear las hipótesis, se simuló un escenario real de negociación de smells sobre un sistema dado. El sistema analizado fue "SubscribersDB", que comprende funcionalidad editorial [3]. Esta aplicación gestiona los datos relacionados con los suscriptores de sus publicaciones y permite realizar diferentes consultas sobre los datos. Se trabajó con la última versión de "SubscribersDB", la cual posee 84 code smells entre los 8 componentes principales de la aplicación. En la Fig. 6 se indican la cantidad de cada tipo de smell encontrados en la aplicación.

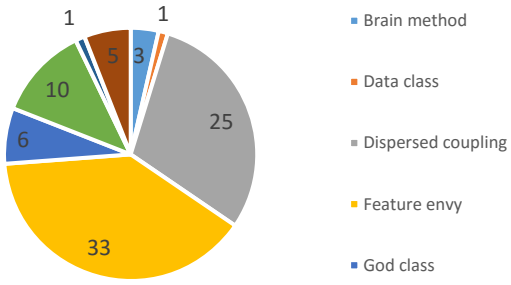

Figura 6. Distribución de smells en SubscribersDB.

Se realizó el experimento en el contexto de un curso de "Mantenimiento y Evolución de Software" dictado en UNICEN. El experimento se ejecutó en una modalidad off-line (en laboratorio bajo condiciones controladas). Los sujetos participantes fueron 24 estudiantes de quinto año de la carrera de Ingeniería de Sistemas. Estos participantes fueron asignados aleatoriamente a 6 grupos de 4 personas cada uno. Los grupos fueron luego divididos en 3 categorías: (A) priorización y negociación manual, (B) priorización asistida (por D-JSpIRIT) y negociación manual, (C) priorización y negociación asistida (por D-JSpIRIT). Específicamente, se requirieron dos tareas

por grupo.

Tarea #1. Cada integrante del grupo asumió el rol de un "desarrollador remoto" que trabaja en conjunto con el resto de los integrantes del grupo en el proyecto SubscriberDB y que desea priorizar los smells de dicho sistema. Cada integrante del grupo recibió información acerca de las clases/paquetes del proyecto sobre las que tiene ownership y el listado total de smells del sistema. En base a su conjunto de clases/paquetes, la persona debía determinar los smells involucrados en su conjunto, y establecer prioridades de manera de poder armar un ranking propio de smells. En el caso de los grupos de la categoría A, las prioridades podían ser definidas basándose sus propios criterios (por ej. tipo de smell, componente con mayor cantidad de smells, mayor número de instancias de un tipo de smell, clases claves en el diseño, valor para el negocio, atributos de calidad, etc.). En el caso de los grupos con categoría B y C, los criterios posibles estuvieron restringidos a los proporcionados por D-JSpIRIT. De esta manera, estos grupos sólo podían seleccionar y configurar los criterios en D-JSpIRIT. Por esta razón, los grupos de las categorías B y C tuvieron una sesión de "entrenamiento" en el uso de D-JSpIRIT de 30 minutos (previa al experimento).

Tarea #2. En los grupos en las categorías A y B, cada integrante debía presentar su ranking a los demás miembros del grupo. Los miembros de un grupo debieron verificar que no haya inconsistencias entre los rankings individuales. En caso de inconsistencias, debían negociar formas de solucionarlas. Cada integrante debía defender su postura tratando de conservar su ranking, y modificarlo solo si estaba de acuerdo con los argumentos presentados por el resto. La modificación consistía en cambiar las prioridades de cada integrante para generar nuevos rankings con mejor consistencia. En el caso del grupo C, esta tarea fue realizada automáticamente por el enfoque D-JSpIRIT distribuido. En este caso, la negociación fue realizada por el sistema multi-agente en base al algoritmo SyncBB de DisCSP.

Al finalizar ambas tareas cada grupo reportó los rankings individuales resultantes de la negociación. Un moderador del experimento (externo) cronometró los tiempos utilizados por cada grupo para cada una de las tareas.

TABLA I
RESULTADOS DE LA NEGOCIACION GRUPAL

Categoría	Grupo	Tiempo (en minutos) Tarea #1	Tiempo (en minutos) Tarea #2	Tarea #1 + Tarea #2	% Rankings individuales con inconsistencias
A	A1	45	70	115	100%
	A2	60	60	120	66%
B	B1	**18**	57	**75**	**25%**
	B2	23	60	83	100%
C	C1	**20**	**15**	**35**	**0%**
	C2	**19**	**12**	**31**	**0%**

Análisis e interpretación de los datos

Durante la tarea #1, los grupos de la categoría A utilizaron en promedio un 260% más de tiempo que los grupos de las categorías B y C (Tabla I). Esto se debe a que los grupos de la categoría A debieron definir qué tipo de criterios utilizarían para priorizar los smells. En cambio, como se dijo anteriormente, los integrantes de los grupos de categoría B y C utilizaron el criterio de "relevancia del tipo de code smell" provisto por D-JSpIRIT. Los grupos de la categoría A definieron como criterios de priorización características tales como: tipo de code smell, cantidad de smells de un mismo tipo, intuición, número de componentes afectados por un smell, clases relevantes en el diseño de la aplicación, etc.

Durante la tarea #2, los grupos de la categoría A y B utilizaron en promedio un 480% y 430% más de tiempo que los grupos de la categoría C, respectivamente (Tabla I). El tiempo de los grupos de las categorías A y B fue destinado principalmente a exponer los criterios de cada integrante, acordar criterios comunes y solucionar inconsistencias en los rankings. En cambio, en el caso de los grupos de la categoría C, dado que estas tareas fueron "reemplazadas" por los agentes del enfoque y realizadas semi-automáticamente, el tiempo fue principalmente destinado a configurar los distintos parámetros de la herramienta.

Teniendo en cuenta el tiempo total utilizado por los distintos grupos, se aplicó el test de Mann-Whitney a los datos a fin de verificar las hipótesis [19]. Como resultado, se pudo rechazar $H_{2,0}$ con un test a una cola con probabilidad de error $\alpha = 0.1$ y p-value=0.06667. De esta manera, se observa que existe una diferencia estadísticamente significativa para afirmar que el tiempo empleado por los desarrolladores para crear el ranking manualmente es mayor al tiempo que deben emplear si utilizan el enfoque de D-JSpIRIT.

Con respecto a la consistencia entre los rankings individuales entregados al finalizar el experimento, se analizaron los pares de rankings que contenían smells en común. En el caso del Grupo A1 el 100% de los pares analizados tenían al menos una inconsistencia (Tabla I). El nivel de inconsistencia encontrado en el Grupo A2 fue también alto: 66%. Con respecto a los grupos en la categoría B, se encontraron inconsistencias en el 100% y el 25% de los pares de rankings de estos grupos. Por último, como era de esperar, no se encontraron inconsistencias en los rankings generados por el enfoque (Grupos C1 y C2) dado que los agentes realizan negociaciones hasta lograr que se eliminen las posibles inconsistencias.

Teniendo en cuenta los valores de inconsistencias encontrados (Tabla I) y aplicando el test de Mann-Whitney, se pudo rechazar $H_{1,0}$ con un test a una cola con probabilidad de error $\alpha = 0.05$ y p-value=0.0476. De esta manera, se observa que existe una diferencia significativa entre el número de inconsistencias generadas por los grupos manualmente y la alternativa basada en D-JSpIRIT.

En resumen, el experimento permitió responder a la PI afirmativamente dado que el enfoque facilita la generación de rankings de smells sin inconsistencias en un tiempo menor o igual al necesario para realizar dicha actividad en forma manual manualmente.

V. CONCLUSIONES

En este trabajo se presentó D-JSpIRIT una herramienta multi-agente que permite priorizar consistentemente los code smells detectados en entornos de trabajo distribuidos. D-JSpIRIT aplica técnicas de satisfacción distribuida de

restricciones (DisCSP) para modelar un conjunto de agentes que representan los intereses de desarrolladores individuales sobre rankings de code smells. Cada agente interactúa con otros agentes para buscar un ranking consistente a nivel global.

Para determinar la factibilidad del enfoque se realizó un caso de estudio en el que se encontró que el uso de D-JSpIRIT permite generar rankings de smells consistentes en un tiempo menor al que debería emplear un grupo de desarrollo distribuido al realizar la misma actividad de forma manual.

Como limitación del enfoque puede mencionarse que el mismo solo implementa el criterio de tipo de smell propuesto por JSpIRIT. En este sentido. como trabajo futuros, se pretende extender D-JSpIRIT para que funcione con múltiples criterios de priorización. Adicionalmente, se buscara tener en cuenta limitaciones adicionales al momento de generar el ranking global. Por ejemplo, el esfuerzo necesario para solucionar un tipo de smell y el tiempo disponible por un desarrollador para realizar esta tarea.

REFERENCIAS

[1] W. Cunningham, "The wycash portfolio management system," OOPS Messenger, vol. 4, no. 2, pp. 29–30, 1993.

[2] M. Fowler, "Refactoring: improving the design of existing code," Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," Automated Software Engineering, pp. 1–32, 2014.

[4] R. Arcoverde, E. Guimarães, I. Macía, A. Garcia, and Y. Cai, "Prioritization of code anomalies based on architecture sensitiveness," Software Engineering (SBES), 2013 27th Brazilian Symposium on. IEEE, 2013, pp. 69–78.

[5] R. Marinescu, G. Ganea, and I. Verebi, "Incode: Continuous quality assessment and improvement," in CSMR, R. Capilla, R. Ferenc, and J. C. Dueas, Eds. IEEE, 2010, pp. 274–275.

[6] D. I. Sjoberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," IEEE Transactions on Software Engineering, vol. 39, no. 8, pp. 1144–1156, 2013.

[7] B. Sengupta, S. Chandra, and V. Sinha, "A research agenda for distributed software development," in Proceedings of the 28th international conference on Software engineering. ACM, 2006, pp. 731–740.

[8] J. Ferber, "Multi-agent systems: an introduction to distributed artificial intelligence," Addison-Wesley Reading, 1999, vol. 1.

[9] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo, "Adopt: Asynchronous distributed constraint optimization with quality guarantees," Artificial Intelligence, vol. 161, no. 1, pp. 149–180, 2005.

[10] L. Bass, "Software architecture in practice," Pearson Education, 2007.

[11] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, "Jspirit: a flexible tool for the analysis of code smells," in 2015 34th International Conference of the Chilean Computer Science Society (SCCC). IEEE, 2015, pp. 1–6.

[12] M. Lanza and R. Marinescu, "Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems," Springer, 2006.

[13] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems." IBM Journal of Research and Development, vol. 56, no. 5, p. 9, 2012.

[14] J. Lin and M. Chunyan, "Context-aware task allocation for distributed agile team," in Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, 2013, pp. 758–761.

[15] E. Bowring, M. Tambe, and M. Yokoo, "Balancing local resources and global goals in multiply-constrained dcop1," Multiagent and Grid Systems, vol. 6, no. 4, pp. 353–393, 2010.

[16] J. Herbsleb and J. Roberts, "Collaboration in software engineering projects: A theory of coordination," ICIS 2006 Proceedings, p. 38, 2006.

[17] K. Hirayama and M. Yokoo, "Distributed partial constraint satisfaction problem," in Principles and Practice of Constraint Programming-CP97. Springer, 1997, pp. 222–236.

[18] R. Tarjan, "Testing flow graph reducibility," in Proceedings of the fifth annual ACM symposium on Theory of computing. ACM, 1973, pp. 96–107.

[19] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, and B. Regnell, "Experimentation in Software Engineering," Springer, 2012.

Hernan Ceferino Vazquez es Ingeniero de Sistemas de la Fac. de Ciencias. Exactas de la Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN-Argentina). Actualmente, es becario de CONICET, estudiante del doctorado en Ciencias de la Computación y docente en UNICEN. Sus intereses de investigación se centran en técnicas para la mejora en el mantenimiento de proyectos Java y Javascript.

Claudia Marcos es profesora de la Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN-Argentina) desde 1991. Pertenece al ISISTAN (Instituto de Sistemas Tandil). Es investigadora de CIC (Comisión de Investigaciones Científicas de la Provincia de Buenos Aires). Ella obtuvo el título de Ingeniero de Software (UNICEN) en el año 1993 y el titulo de Doctora en Ciencias de la Computación (UNICEN) en el año 2001. Sus principales áreas de investigación son evolución de sistemas, ingeniería de requerimientos, y desarrollo ágil.

Santiago Vidal es ingeniero de sistemas y posee una maestría en ingeniería de sistemas y un doctorado en ciencias de la computación de la Fac. de Ciencias Exactas de Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN -Argentina). Es docente de UNICEN e investigador del Consejo Nacional de Investigaciones Científicas y Técnicas de Argentina (CONICET) en el instituto ISISTAN. Sus principales intereses de investigación son: la evolución y el mantenimiento de software.

J. Andres Diaz Pace es profesor de la Fac. de Ciencias Exactas de Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN -Argentina) e investigador del Consejo Nacional de Investigaciones Científicas y Técnicas de Argentina (CONICET). Del 2007 al 2010, fue miembro del Instituto de Ingeniería de Software (SEI, Pittsburgh, USA). Obtuvo el título de Dr. en Ciencias de la Computación en el 2004 en la Fac. de Ciencias Exactas de Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN -Argentina). Sus principales áreas de interés son diseño de arquitecturas dirigidas por la calidad, técnicas de IA aplicadas al diseño. Es profesor de varios cursos de grado y posgrado y ha dirigido varias tesis de grado y posgrado