Detección de patrones de diseño con GEML: discusión y enfoque práctico*

José Raúl Romero[1,2], Rafael Barbudo[1], Aurora Ramírez[1], Francisco Servant[3]

[1] Dpto. de Informática y Análisis Numérico, Universidad de Córdoba, España
[2] Instituto Interuniversitario de Investigación en Data Science and Computational Intelligence (DaSCI), Andalucía, España
[3] Virginia Tech, Virginia, Estados Unidos de América
{jrromero,rbarbudo,aramirez}@uco.es, fservant@vt.edu

Resumen La adopción de buenas prácticas es fundamental para conseguir código de calidad, elegante, comprensible y mantenible. Sin embargo, con el tiempo, tanto código como documentación se degradan. Para ello, la detección automática de patrones de diseño es un área prominente de investigación en ingeniería inversa, que persigue comprender las decisiones de diseño originales, así como ayudar a redocumentar y recuperar código disperso en un repositorio de software. Son varias las propuestas hasta el momento, mayoritariamente basadas en similitudes o métodos formales. Estas técnicas resultan rígidas en la búsqueda, ya que la codificación de un patrón puede variar según cada equipo de desarrollo. Por ello, surgieron las propuestas basadas en aprendizaje automático. Recientemente GEML se propuso como un método basado en clasificación asociativa y programación genética gramatical, que aprende la forma de estos patrones con el objetivo de aportar flexibilidad, interpretabilidad y precisión en la detección. Este trabajo analiza las características de GEML que influyen en su aplicabilidad práctica por el ingeniero software, comparándolo además con herramientas de referencia en el área.

Palabras clave: minería de repositorios software · detección de patrones de diseño · clasificación asociativa · programación genética

1. Introducción

La adopción de patrones de diseño es una de las buenas prácticas con la que los programadores pueden mejorar la calidad de los productos de software en términos de su capacidad de mantenimiento, elegancia, flexibilidad y comprensibilidad [5]. Teniendo en cuenta estos beneficios, y considerando que la inspección manual es un proceso propenso a errores y que consume mucho tiempo, la detección automática de patrones de diseño (DPD) se ha convertido en un área prominente en la investigación de ingeniería inversa [1], cuyo objetivo es la comprensión de las decisiones originales de diseño, así como ayudar en los procesos de redocumentación, reimplementación y reutilización.

* Parcialmente financiado por ministerios de Ciencia e Innovación (RED2018-102472-T, PID2020-115832GB-I00, FPU17/00799), y Junta de Andalucía (DOC_00944).

La mayoría de técnicas automáticas para DPD buscan estructuras particulares en el código estático [10], por lo que estas deben ser definidas por expertos en una base de conocimiento, que suele ser específicas de la cultura de desarrollo empresarial. El hecho de que el experto defina estas estructuras de código puede imponer rigidez a la técnica de detección, y requerir que las estructuras se cambien y los patrones de diseño deban ser reinterpretados para contextos particulares. Con este fin se propusieron técnicas de aprendizaje automático (ML) para DPD [3], las cuales se enmarcan en el área de la minería de repositorios software (MSR). Estas técnicas aprenden de una colección de ejemplos representativos (muestras) y, por tanto, pueden reconocer implementaciones distintas si se cambia esta colección, esto es, el repositorio organizacional en el que se almacena el código. Además, los enfoques basados en ML proporcionan mecanismos para aprender tanto propiedades estructurales como de comportamiento en el código fuente, y utilizar métricas software. Aun así, los enfoques actuales de ML consideran alternativamente o las métricas de software o las propiedades del código, pero no ambas. Otro aspecto a considerar es que los enfoques basados en ML para DPD se ven afectados por el patrón de diseño analizado y, a menudo, se requiere del ajuste de su configuración con parámetros específicos, lo que puede resultar limitante para los ingenieros software y su puesta en práctica. Finalmente, también la interpretabilidad de los resultados proporcionados por ciertas técnicas basadas en ML resulta un problema. Por ejemplo, los modelos de caja negra son difíciles de entender e interpretar por el experto humano, por lo que es menos probable que se confíe en sus recomendaciones [11].

GEML [2] es un nuevo enfoque basado en ML –aplica clasificación asociativa (AC, *associative classification*)– para DPD a partir de código estático extraído de un repositorio software. Para su implementación utiliza una solución basada en programación genética guiada por gramática (G3P, *grammar-guide genetic programming*). Al igual que otras propuestas basadas en ML, GEML aprende a partir de muestras de patrones, lo que le aporta la capacidad de capturar implementaciones diversas. Igualmente, GEML se diseñó para promover la extensibilidad, la legibilidad y la configurabilidad, frente a las limitaciones de los métodos ML anteriores. Más específicamente, GEML construye un clasificador basado en reglas guiado por una gramática libre de contexto (CFG, *context-free grammar*) configurable, y que permite al ingeniero modificar (añadiendo, quitando o cambiando), por ejemplo, el tipo de elementos de diseño (relaciones, métricas, etc.) que deben utilizarse para la detección. Estas reglas constituyen un mecanismo bien establecido para codificar el conocimiento humano [6], y describen las características distintivas de las instancias del patrón de una manera más comprensible para el ingeniero. En términos de configurabilidad, GEML permite trabajar con una configuración genérica única, bajo la cual, a pesar de tratarse de un método de ML, es capaz de predecir con mejores resultados que el resto de propuestas para la DPD sobre la gran mayoría de los patrones.

En este artículo abordamos un estudio detallado sobre la aplicabilidad de GEML por parte del ingeniero software, discutiendo aspectos específicos relacionados con su flexibilidad, interpretabilidad y precisión en la detección. En

concreto, se estudiarán qué estructuras de microdiseño son más habituales en las soluciones obtenidas para los diferentes patrones. También se analizarán las reglas del clasificador resultante (detector) en términos de su interpretabilidad, y se comparará el rendimiento (precisión detectora) de GEML frente a dos de las herramientas actualmente más utilizadas para la DPD: SSA y Ptidej.

Este artículo se organiza como sigue. En la Sección 2 se explican las propuestas más importantes en el área de DPD. En la Sección 3 se introduce GEML y, posteriormente, la Sección 4 expone un ejemplo de detector resultante, explicando sus componentes y morfología. En la Sección 5 se lleva a cabo la experimentación y discusión de los factores mencionados anteriormente, y que inciden en la aplicabilidad práctica del método por parte de ingenieros no expertos en ML: flexibilidad, interpretabilidad y precisión en la detección. Finalmente, se presentan las conclusiones en la Sección 6.

2. Trabajo relacionado

Se han propuesto múltiples técnicas para la DPD tomando, en su mayoría, el código fuente como entrada. Se han utilizado técnicas de razonamiento basado en la programación lógica declarativa [9], y que utiliza el motor de inferencia de Prolog para buscar coincidencias exactas. También se ha utilizado lógica difusa, o combinado el uso de lógicas con meta-patrones. En todos estos casos, la base de conocimientos debe ser construida por el experto, por lo que los resultados del proceso de detección son dependientes de su definición y podrían estar sesgados. Los métodos formales también permiten aplicar análisis formal de conceptos para encontrar grupos de clases que compartiesen relaciones estructurales comunes que representen patrones de diseño candidatos [14].

Como alternativa, las técnicas basadas en similitud, como la referenciada herramienta SSA [15], buscan subestructuras que se correspondan con una plantilla predefinida del grafo que describe la estructura de un determinado patrón. Se pueden aplicar mecanismos para tratar coincidencias aproximadas, con pequeñas variaciones en el valor de algunas propiedades. Otras variantes se basan en la obtención primero de subestructuras del grafo, para después comparar las signaturas de métodos con lo esperado en el patrón, o bien en mejorar el rendimiento filtrando clases irrelevantes en una fase más temprana. Otras propuestas basadas en anotaciones semánticas han propuesto el uso de analizadores automáticos [12] pero, como reconocen sus autores, las definiciones incompletas o la semántica inadecuada afectan negativamente a la detección. Por otra parte, otros autores han formulado la DPD como un problema de satisfacción de restricciones, definiendo aquellas condiciones estructurales y de comportamiento que debería satisfacer cada patrón en particular. Este tipo de método requiere la definición y formalización manual de las relaciones entre roles. DeMIMA [8] es un ejemplo representativo de este enfoque, ofreciendo explicaciones sobre las restricciones satisfechas y no satisfechas. Además, se ha ampliado para incluir propiedades numéricas como forma de reducir el número de candidatos por rol, estando disponible dentro del conjunto de herramientas Ptidej.

En cuanto a ML, principalmente se ha trabajado con modelos de clasificación para caracterizar los patrones a través de la inspección de código. Un primer trabajo que hizo uso de reglas de asociación (método descriptivo), las utilizó para descartar las clases que no desempeñaran un rol conforme a algunas métricas de software, por lo que este método no era directamente responsable de la detección [7]. También se han utilizado redes neuronales con un conjunto de características definidas manualmente (predictores) para reducir el número de falsos positivos detectados tras un análisis estructural del código [3]. En concreto, una serie de métricas alimentan la red neuronal cuyo objetivo es la identificación de clases que potencialmente desempeñan un rol en el patrón, antes de abordar la detección [16]. También pueden encontrarse variantes basadas en máquinas de vector soporte o técnicas de agrupamiento.

Como novedad, GEML es un método basado en AC, que ofrece las ventajas de los métodos de ML, lo que le permite adaptarse a las políticas de la organización, así como a diferentes estilos de programación. Al igual que otros métodos basados en ML, descubre automáticamente cuáles son las propiedades que mejor definen cada patrón explorando las implementaciones existentes en el repositorio de código. Además, GEML se diferencia de los enfoques existentes en que permite la combinación de métricas y propiedades de software como entrada para el aprendizaje. GEML produce un clasificador basado en reglas, en lugar de detectores de caja negra, como las redes neuronales o las máquinas de vectores soporte, cuyos resultados son modelos difíciles de entender, lo que reduce su probabilidad de adopción por parte de los profesionales [11].

3. GEML para la detección de patrones de diseño

GEML es un método supervisado de DPD, cuyas muestras de entrenamiento se corresponden con instancias de patrones etiquetadas para poder realizar la predicción. Su operación se encuadra en dos pasos diferenciados: entrenamiento y detección. Para el *entrenamiento*, el proceso se inicia a partir de un repositorio que contiene tanto las muestras positivas de los patrones predichos en el pasado, como las muestras negativas. Estas últimas proporcionan información relevante sobre escenarios realistas, ya que se forman con código similar al de un patrón real que, sin serlo, podría llevar a una clasificación errónea del detector. De hecho, un alto número de muestras negativas beneficia a la robustez de la detección, especialmente cuando la cantidad de muestras positivas es baja [13], como puede ocurrir en este dominio. Para seleccionar un conjunto adecuado de muestras negativas, se realiza una fase de preprocesado consistente en la elección de subgrafos aproximados a los patrones.

El proceso de detección de GEML se ejecuta a partir del conjunto de entrenamiento y de la gramática que define el lenguaje sobre el que se construye el modelo de detección. En concreto, se aplica AC, una técnica fundamentada en el uso de reglas *si-entonces* comprensibles sobre un enfoque de clasificación en dos fases. Primero, se extrae un conjunto de reglas que servirá de base al clasificador. Formalmente, si tenemos el conjunto de *items* $\mathrm{I} = \{i_1, ..., i_n\}$, y A

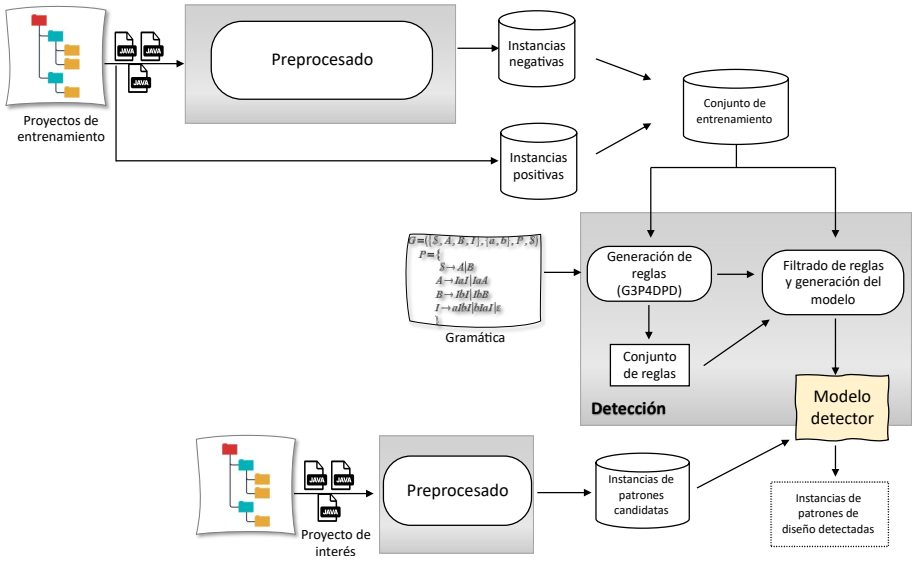

Figura 1: Esquema general de GEML

= $\{i_1, ..., i_j\}$ y C = $\{i_1, ..., i_k\}$ son conjuntos de *itemsets*, entonces una regla de asociación es una implicación de tipo $A \rightarrow C$ donde A $\subset$ I, C $\subset$ I y A $\cap$ C = $\emptyset$. Para calcular su calidad se utilizan los criterios de soporte, que indica cómo de frecuentemente se cumple una determinada regla (A $\cup$ C) dentro del conjunto de datos, y confianza, que mide la proporción de muestras que satisfacen el consecuente de entre aquellas reglas que cumplen con el antecedente.

Las reglas de asociación se construyen utilizando un algoritmo propio, denominado G3P4DPD, que hace uso de una técnica evolutiva, G3P, especialmente concebida para evolucionar programas informáticos de acuerdo con una gramática que define cómo se construyen estos programas, es decir, una regla potencial del modelo de detección de GEML. G3P es una extensión de GP (programación genética), una técnica de computación evolutiva en la que los individuos se codifican como estructuras de árboles, por lo que requiere operadores especializados para manipularlas. En particular, G3P utiliza una CFG para describir las restricciones sintácticas que debe satisfacer cualquier individuo válido (regla). Esta gramática se define en términos de la tupla $\{S, \sum_N, \sum_T, P\}$, donde S es el símbolo raíz, $\sum_N$ es el conjunto de símbolos no terminales, $\sum_T$ es el conjunto de símbolos terminales y P es el conjunto de reglas de producción. Una regla de producción indica cómo se puede reescribir un símbolo no terminal en una de sus derivaciones hasta que la expresión solo contenga símbolos terminales. Formalmente, se expresa como $a \rightarrow B$, donde $a \in \sum_N$ y $B \in \{\sum_N \cup \sum_T\}^*$. Así, cada individuo se crea derivando una secuencia diferente de reglas de producción, representadas por su árbol de derivación. Los elementos de la CFG también se consideran durante la aplicación del cruce y la mutación para garantizar la

producción de reglas de asociación válidas. En particular, el proceso evolutivo está orientado a encontrar un conjunto de individuos de alta calidad.

Conviene detenerse en cómo se generan las reglas a partir de la gramática. Toda regla tendrá la forma $A \rightarrow C$, donde el consecuente tomará la forma de clase binaria, esto es, es un patrón (`aPattern`) o no (`notAPattern`). Para el antecedente, el conjunto de símbolos no terminales permitirán y darán flexibilidad a las reglas en términos de predicados compuestos de condiciones categóricas y numéricas. Así pues, el conjunto de símbolos terminales contiene operadores de comparación de distinta índole, así como elementos para la definición de los roles de patrón (p.ej. *adapter*, *target* o *adaptee* para el patrón *Adapter*). Además, una ventaja distintiva de GEML es su capacidad de hacer uso de condiciones de ambos tipos, categóricas y numéricas, simultáneamente. Las primeras permiten aplicar estructuras de microdiseño (véase la Tabla 1) que definen las relaciones entre los elementos y roles del patrón, así como su morfología, de manera similar a cómo lo realizaría un ingeniero software. Igualmente, las condiciones numéricas permiten el uso de métricas software medidas sobre el código analizado para buscar sesgos o límites conocidos en sus valores. Obsérvese que el ingeniero software puede decidir qué estructuras y métricas utilizar, así como eliminar o reducir el número de ellas para algún caso concreto.

Posteriormente, las reglas son filtradas asegurando que solo se seleccionan aquellas que formarán parte del modelo detector. Para ello, se aplica un algoritmo de *poda* de las reglas, que se ordenan según su confianza, soporte y tamaño (número de comparaciones), respectivamente. A continuación, las muestras del repositorio se escanean buscando las reglas que coinciden en su antecedente. Si la regla clasifica correctamente, se añade al detector y se incrementan las muestras cubiertas en el repositorio. Cuando una muestra se cubre por un número de reglas igual al *umbral*, deja de considerarse. Si una regla detecta las mismas muestras que otras con mayor confianza y soporte, se descarta. Este proceso se repite hasta cubrir todo el repositorio o hasta que no queden más reglas.

Por su parte, la operación de *detección* hace uso del modelo generado por GEML, considerando como entrada el repositorio del nuevo proyecto al que se desea aplicar el proceso. Un primer paso de preprocesado aligera el tiempo de detección encontrando previamente estructuras aproximadas de patrones candidatos para la detección. Tras su ejecución, se devuelven las instancias de patrones de diseño detectados, que pueden incorporarse al repositorio organizativo para futuras detecciones, consiguiendo así que el modelo de detección se adapte progresivamente a la cultura de desarrollo específica. Cada instancia de patrón dentro del repositorio se caracteriza por sus elementos constitutivos, su código fuente y el mapeo de roles, es decir, la correspondencia entre elementos y roles.

4. Ejemplo de detector

Las reglas que forman parte del modelo detector describen las muestras del conjunto de entrenamiento del clasificador. Para llevar a cabo la operación de detección con nuevas muestras (o un conjunto de prueba) se aplican estrategias de

Tabla 1: Ejemplos de métricas y estructuras de microdiseño (extraída de [2])

Signatura del operador	**Descripción**
Métricas software	
NOC(r_1)	Número de hijos directos de r_1
DIT(r_1)	Profundidad del árbol de herencia a partir de r_1
RFC(r_1)	Número de métodos y constructores que pueden ser invocados cuando un objeto de r_1 recibe un mensaje
Estructuras de microdiseño	
isSubclass(r_1)	*true* si r_1 es una subclase; *false* en otro caso
controlledInit(r_1)	*true* si r_1 se instancia a sí mismo dentro de un bloque *if* o *while*; *false* en otro caso
conglomeration(r_1)	*true* si 2 o más métodos de r_1 son invocados desde otro método de r_1; *false* en otro caso
returns(r_1,r_2)	*true* si algún método de r_2 devuelve una instancia de r_1; *false* en otro caso
receives(r_1,r_2)	*true* si un método de r_2 recibe una instancia de r_1 como argumento; *false* en otro caso
createObj(r_1,r_2)	*true* si r_1 instancia a r_2; *false* en otro caso
delegates(r_1,r_2)	*true* si un método de r_1 invoca un método de r_2; *false* en otro caso
typeOf(r_1)	Devuelve el tipo de artefacto que implementa r_1 (*absClass*, *intface*, *enum*, *class*)
linkArtefact(r_1,r_2)	Devuelve el tipo de relación entre r_1 y r_2 (*directInherit*, *indirInherit*, *directImpl*, *indirImpl*, *notLinked*)
ctorVisibility(r_1)	Devuelve la visibilidad del constructor menos restrictivo de r_1 (*private*, *protected*, *package*, *public*)
aggregation(r_1,r_2)	Devuelve información sobre un atributo de tipo r_2 declarado en r_1 en base a su visibilidad e instanciabilidad
adapterMethod(r_1,r_2,r_3)	Devuelve si un método declarado (*decl*) o heredado (*inhr*) de r_1, implementado de r_3, delega en un método de r_2; *notLinked* en otro caso
redirectInFamily(r_1)	Devuelve si un método declarado de r_1 en una clase o interfaz que sea extendida o implementada por r_1, una (*single*) o varias veces (*multi*); *notLinked* en otro caso
sameInterfaceContainer(r_1,r_2)	*true* si $r2$ es una clase o interfaz extendida o implementada por r_1, que define una colección de objetos de una clase o interfaz extendida o implementada por r_2; *false* en otro caso

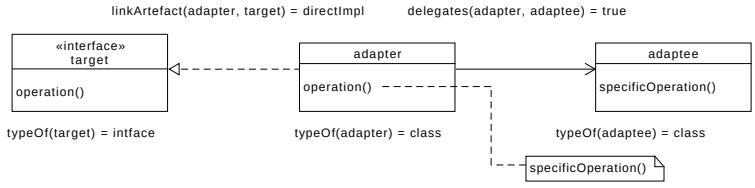

Figura 2: Esquema del patrón *Adapter* con roles y estructuras de relación

clasificación específicas que determinan qué reglas son consideradas para clasificar la nueva muestra. Recordemos que el conjunto de reglas vendrá determinado por el repositorio organizacional sobre el que se entrena, así como por el patrón

de diseño que se pretende detectar. El uso de técnicas de ML nos permite encontrar relaciones ocultas en los datos que, por sí mismas, el ingeniero software no hubiera podido detectar. A modo de ejemplo, en esta sección mostramos el caso del patrón *Adapter*, del que se muestra un subconjunto limitado de reglas en el Código 1.1. Para la detección se utilizó un repositorio con 618 muestras positivas y 603 negativas de este patrón. Obsérvese que las reglas se representan de la forma A -> C. El antecedente (A) lo forman múltiples predicados condicionales que hacen uso tanto de elementos de microdiseño como de métricas software. El consecuente (C) indica si el antecedente describe un patrón o no.

Código 1.1: Subconjunto de reglas del detector para patrón Adapter

```
1  ...
2  ( adapterMethod(adapter,adaptee,target) != notLinked && ctorVisibility(adaptee) != package &&
       DIT(adaptee) > 4.0 && NOC(adaptee) <= 57.0 ) -> ( label == aPattern )
3  ( = adapterMethod(adapter,adaptee,target) == notLinked && conglomeration(target) == true &&
       returns(target,target) != true && RFC(adaptee) >= 67.0 ) -> ( label == notAPattern )
4  ( adapterMethod(adapter,adaptee,target) != notLinked && RFC(adaptee) <= 133.0 &&
       ctorVisibility(adaptee) == private && returns(adaptee,adapter) != true ) -> ( label ==
       aPattern )
5  ( adapterMethod(adapter,adaptee,target) == notLinked && typeOf(adapter) == absClass && RFC(
       adaptee) < 133.0 && createObj(target,adaptee) != true ) -> ( label == notAPattern )
6  ( adapterMethod(adapter,adaptee,target) != notLinked ) -> ( label == aPattern )
7  ( NOC(adaptee) <= 19.0 &&  typeOf(adapter) != absClass && RFC(adaptee) >= 265.0 && DIT(
       adaptee) < 3.0 ) -> ( label == aPattern )
8  ( isSubclass(adapter) != true && conglomeration(adapter) == true ) -> ( label == notAPattern
       )
9  ( isSubclass(adapter) != true && conglomeration(adapter) != true ) -> ( label == aPattern )
10 ...
```

Por ejemplo, la regla de la línea 2 establece que, si (a) no hay ningún método en la clase "adapter" implementado a partir de "target" que delegue en un método de "adaptee" (*adapterMethod(adapter, adaptee, target)* $!=$ *notLinked*); y (b) la visibilidad del constructor de la clase "adaptee" es de paquete (*ctorVisibility(adaptee)* $==$ *package*); y (c) la profundidad del árbol de herencia a partir de la clase "adaptee" es mayor que 4 (*DIT(adaptee)* $>$ *4.0*); y (d) el número total de hijos de la clase "adaptee" es menor que 57 (*NOC(adaptee)* $<=$ *57.0*), entonces es una instancia del patrón *Adapter* (*aPattern*) en el repositorio en estudio. Por su parte, la regla de la línea 5 establece que si no se cumple la condición (a) del ejemplo anterior; y (b) el elemento "adapter" es una clase abstracta (*typeOf(adapter)* $==$ *absClass*); y (c) el número de métodos distintos potencialmente invocados al recibir "adaptee" un mensaje es menor que 133 (*RFC(adaptee)* $<$ *133.0*); y (d) "adaptee" no se instancia desde "target" (*createObj(target,adaptee)* $!=$ *true*), entonces la muestra no es un patrón (*notAPattern*) de tipo *Adapter* en este repositorio.

Los siguientes ejemplos son dos casos extremos de reglas que describen un número significativamente alto o bajo de muestras. Así, la regla de la línea 6 establece que si existe un método de "adapter", implementado a partir de "target", que delegue en un método de "adaptee", entonces es un patrón *Adapter*. Como puede observarse, se trata de una formulación usualmente distintiva de este patrón de diseño, que se cumple en el 46.2 % de las muestras del repositorio (565) con una confianza del 83.8 %. En el otro extremo, la regla de la línea 7 establece que si (a) el número de subclases directas de "adaptee" es menor o igual

a 19; y (b) el elemento con rol "adapter" es una clase abstracta; y (c) el número de métodos distintos potencialmente invocados cuando "adaptee" recibe un mensaje es mayor o igual a 265; y (d) la profundidad de la herencia a partir de "adaptee" es menor que 3, entonces describe un patrón *Adapter*. Obsérvese que, en este caso, esta regla representa a una única instancia de patrón específica, por lo que su confianza es del 100 % (regla exacta). Ambos casos son extremos en valores de soporte, y son presumiblemente útiles para recoger muestras muy específicas del conjunto de entrenamiento (bien generalizando bien particularizando) que, de otra forma, no serían cubiertas.

Finalmente, destacamos también el caso de las reglas de las líneas 8 y 9, para las que una mínima variación en un comparador lógico (p.ej. realizada mediante un operador genético de mutación) puede cambiar el sentido del consecuente de la regla y, aun así, ser aplicables a un número considerable de muestras del repositorio de entrenamiento. Así, la primera regla establece que si (a) el elemento "adapter" es una subclase y (b) dos o más métodos de "adapter" son invocados desde métodos del propio "adapter", entonces no es un patrón. En este caso, la regla se cumple en 122 muestras del repositorio con una confianza del 76.2 %. Por el contrario, si se niega la condición (b) de la regla, se identificará positivamente como patrón para 118 muestras con una confianza del 80.7 %.

5. Resultados experimentales y discusión

5.1. Marco experimental

La experimentación engloba 15 patrones de diseño pertenecientes a las tres categorías definidas por Gamma [5]: creacionales (*Abstract factory*, *Factory method* y *Singleton*), de comportamiento (*Command*, *Iterator*, *Observer*, *State*, *Strategy*, *Template method* y *Visitor*) y estructurales (*Adapter*, *Bridge*, *Composite*, *Decorator* y *Proxy*). Para la fase de entrenamiento de GEML se ha utilizado el repositorio P-Mart[4], que contiene muestras etiquetadas para 9 proyectos Java. Puesto que el número de muestras positivas es bastante limitado para algunos patrones, se ha aumentado el conjunto de datos con otras muestras recuperadas por las herramientas SSA y Ptidej. Además, se han generado muestras negativas a partir del código de los proyectos. Ambas prácticas son habituales en estudios sobre DPD que utilizan ML [3,13]. La Tabla 2 lista los patrones y el número de muestras positivas (+), negativas (-) y totales (T) por patrón. Para la comparación frente a SSA y Ptidej, se utiliza un proyecto no utilizado para entrenar GEML, denominado DPExample [4], que contiene 174 instancias de patrones.

El algoritmo de generación de reglas (G3P4DPD) se ha ejecutado 30 veces con distintas semillas aleatorias y validación cruzada (10 particiones). La configuración de sus parámetros se corresponde con la configuración recomendada en el trabajo original de GEML [2]: población de 100 individuos evolucionados en 150 generaciones; 0.8 de probabilidad de cruce; 25 derivaciones máximas de la gramática (longitud máxima de las reglas); umbrales de soporte (0.01), confianza (0.7) y cobertura (1); y DFML_{χ^2} [2] como estrategia de clasificación.

[4] `http://www.iro.umontreal.ca/~labgelo/p-mart/index.php` (acceso 11/05/2021)

Tabla 2: Patrones de diseño y número de muestras para entrenamiento.

Patrón de diseño	+	-	**T**	**Patrón de diseño**	+	-	**T**
State	147	412	559	Observer	8	14	22
Adapter	65	180	245	Command	5	15	20
Singleton	55	165	220	Fact. Method	3	9	12
Templ. Method	50	147	197	Iterator	3	9	12
Proxy	19	57	76	Visitor	3	9	12
Strategy	7	21	28	Decorator	2	6	8
Abs. Factory	6	18	24	Bridge	2	6	8
Composite	6	18	24				

Tabla 3: Estructuras de microdiseño más frecuentes en las reglas

Patrón	**#Reglas**	**#Estr.**	**Estructuras de diseño**
State	4	6	returns (75), aggregation (50), controlledInit (25), createObj (25), delegates (25), NOC (25)
Adapter	6	11	aggregation (33), conglomeration (33), returns (33), ctorVisibility (17), delegates (17), DIT (17), isSubclass (17), linkMethod (17), NOC (17), sameInterfaceContainer (17), typeOf (17)
Singleton	1	1	aggregation (100)
Templ. Method	6	8	typeOf (50), conglomeration (17), linkMethod (17), NOC (17), NOM (17), receives (17), redirectInFamily (17), RFC (17)
Proxy	3	6	adapterMethod (33), DIT (33), isSubclass (33), NOC (33), returns (33), RFC (33)
Strategy	3	4	createObj (33), linkArtefact (33), linkMethod (33), receives (33)
Abs. Factory	3	3	adapterMethod (33), createObj (33), delegates (33)
Composite	2	3	receives (50), redirectInFamily (50), typeOf (50)
Observer	3	6	adapterMethod (33), aggregation (33), conglomeration (33), createObj (33), NOC (33), receives (33)
Command	3	5	adapterMethod (33), delegates (33), isSubclass (33), linkMethod (33), receives (33)
Fact. Method	1	2	conglomeration (100), ctorVisibility (100)
Iterator	1	3	linkArtefact (100), receives (100), redirectInFamily (100)
Visitor	1	1	typeOf (100)
Decorator	1	1	aggregation (100)
Bridge	1	1	linkArtefact (100)

5.2. Análisis de estructuras de microdiseño

En esta sección se analizan las estructuras de microdiseño que GEML ha encontrado como más relevantes para cada patrón de diseño. La Tabla 3 muestra, para el mejor clasificador obtenido, el número de reglas positivas que describen al patrón y las estructuras que aparecen en ellas (número y frecuencia en %).

Como se observa, para este repositorio, GEML es capaz de caracterizar cada patrón con un pequeño conjunto de reglas (6 o menos). Además, el número de estructuras que aparece en ellas es también reducido, lo que significa que el proceso evolutivo es capaz de identificar el subconjunto de estructuras de microdiseño más representativas para cada patrón. Esta información es relevante para aplicar GEML, pues su gramática podría configurarse eligiendo solo aquellas estructuras relevantes, reduciendo así el tiempo de cómputo. Las estructuras que aparecen asociadas a un número mayor de patrones son *receives*, *aggregation* y *NOC*. Este hecho demuestra que poder combinar propiedades estructurales y métricas de

Tabla 4: Características de las reglas de clasificación para cada patrón.

Patrón de diseño	#Reglas	Longitud media	Soporte	Confianza
State	15	2.53	[0.01, 0.70]	[0.98, 1.00]
Adapter	26	2.35	[0.04, 0.66]	[0.72, 1.00]
Singleton	6	1.67	[0.10, 0.75]	[0.99, 1.00]
Templ. Method	20	2.40	[0.01, 0.60]	[0.80, 1.00]
Proxy	6	2.00	[0.03, 0.68]	[0.98, 1.00]
Strategy	6	2.17	[0.07, 0.68]	[0.75, 1.00]
Abs. Factory	6	1.33	[0.08, 0.58]	1.00
Composite	7	1.86	[0.13, 0.71]	[0.75, 1.00]
Observer	6	1.83	[0.18, 0.59]	[0.83, 1.00]
Command	4	2.00	[0.15, 0.75]	[0.75, 1.00]
Fact. Method	3	1.33	[0.25, 0.67]	1.00
Iterator	3	2.00	[0.25, 0.75]	[0.90, 1.00]
Visitor	2	1.00	[0.25, 0.75]	1.00
Decorator	2	1.00	[0.25, 0.75]	1.00
Bridge	2	1.00	[0.25, 0.75]	1.00

código es un factor importante en DPD que GEML es capaz de explotar. Las estructuras que analizan las relaciones entre clases, como *adapterMethod*, *delegates*, *linkMethod* y *conglomeration*, son también útiles para identificar patrones con varios roles, como *Adapter* y *Abstract Factory*. Ciertamente, la forma en la que se relacionan las clases son características distintivas de estos patrones. Observamos que existen cuatro patrones cuyas reglas solo aplican una estructura de microdiseño: *Singleton*, *Visitor*, *Decorator* y *Bridge*. Por ejemplo, el *Singleton* es un patrón simple compuesto de una única clase que se agrega a sí misma, por lo que *aggregation* es suficiente para detectarlo. El resto de patrones son más complejos, por lo que cabría esperar un mayor número de estructuras para describirlos. Sin embargo, el bajo número de muestras condiciona el tipo de reglas que GEML ha sido capaz de extraer, como se explica en la siguiente sección.

5.3. Interpretabilidad de las reglas

La Tabla 4 recoge varias medidas habituales asociadas a la interpretabilidad del clasificador generado por GEML y a la calidad de las reglas que lo componen. Para la interpretabilidad, se muestra el número de reglas (positivas y negativas), y su longitud (número de expresiones en el antecedente) media. Respecto a la calidad, se detalla el rango de soporte y confianza de las reglas. Un rango de soporte más amplio es indicativo de diversidad de implementaciones, pues no todas comparten las mismas características.

En este repositorio, GEML es capaz de construir clasificadores que, en su mayoría, necesitan menos de 10 reglas para realizar la detección. El hecho de que se obtengan más reglas para ciertos patrones está relacionado con la complejidad del patrón y con el número de muestras disponibles para entrenar. A excepción del patrón *Singleton*, uno de los más simples de describir al tener un solo rol, los tres patrones cuyos clasificadores tienen más reglas son aquellos con más instancias en el repositorio de entrenamiento: *Adapter*, *State* y *Template Method*.

Tabla 5: Resultados de rendimiento predictivo para GEML, SSA y Ptidej.

Patrón	**GEML**			**SSA**			**Ptidej**		
de diseño	Pr	Re	F_1	Pr	Re	F_1	Pr	Re	F_1
State	0.14	0.67	**0.23**	0.07	0.25	0.11	0.09	0.75	0.16
Adapter	0.17	0.33	**0.22**	0.06	0.50	0.10	0.02	0.50	0.04
Singleton	0.81	0.81	**0.81**	0.77	0.81	0.79	0.18	0.71	0.29
Templ. Method	0.55	0.86	**0.67**	0.35	1.00	0.52	0.03	1.00	0.06
Proxy	0.60	0.75	**0.67**	1.00	0.50	**0.67**	0.02	0.75	0.05
Strategy	0.11	0.63	0.19	1.00	0.75	**0.86**	-	-	-
Abs. Factory	0.15	0.79	**0.25**	-	-	-	-	-	-
Composite	0.13	0.67	**0.22**	0.14	0.17	0.15	0.07	0.33	0.11
Observer	1.00	0.29	**0.44**	0.50	0.29	0.36	-	-	-
Command	0.01	0.20	0.02	0.75	0.60	**0.67**	0.06	0.40	0.10
Fact. Method	0.12	0.20	0.15	1.00	0.07	0.13	0.11	0.33	**0.17**
Iterator	0.80	0.80	**0.80**	-	-	-	-	-	-
Visitor	0.42	0.93	0.58	1.00	0.73	**0.85**	1.00	0.13	0.24
Decorator	0.80	0.67	**0.73**	0.26	0.83	0.40	-	-	-
Bridge	0.07	0.50	**0.13**	0.00	0.00	0.00	-	-	-
Media	0,39	0,61	**0,41**	0,46	0,43	0,37	0,11	0,33	0,08

Aunque el número de reglas sea mayor para ellos, las reglas obtenidas tienen una longitud media inferior a 3, lo que las hace fácilmente comprensibles.

Existen tres patrones para los que GEML tiene un comportamiento similar, condicionado por el escaso número de muestras: *Decorator*, *Visitor* y *Bridge*. Para ellos, GEML solo genera un clasificador de dos reglas, una positiva con $soporte = 0{,}25$ y otra negativa con $soporte = 0{,}75$, ambas con $confianza = 1$. Analizando estas reglas, se observa que sus antecedentes están compuestos por una misma expresión, que aparece negada en una de las reglas. Es decir, GEML es capaz de encontrar el único operador que diferencia a las muestras positivas de las negativas, si bien este operador podría no ser suficiente garantía de la presencia real del patrón. En el resto de casos, los valores de soporte son más variados y la confianza es cercana o igual a 1. Por tanto, GEML garantiza encontrar reglas que se ajustan a diferentes implementaciones de ese patrón, permitiendo detectarlas con alta confianza aunque no sean mayoritarias en el repositorio.

5.4. Rendimiento predictivo frente a herramientas de referencia

La Tabla 5 resume los resultados obtenidos por GEML (mejor clasificador entre las 30 ejecuciones), SSA y Ptidej, expresados en base a las siguientes métricas: *precision* (Pr), que calcula la proporción de patrones de diseño correctamente etiquetados como tal; *recall* (Re), que calcula la cantidad de patrones recuperados sobre el total de muestras disponibles; y F_1, que se obtiene como la media armónica entre las dos medidas anteriores por lo que, para cada patrón, se ha indicado la herramienta que obtiene el mejor valor de ésta última. Valores próximos a 1 son preferibles.

GEML ha detectado el 61 % de las implementaciones etiquetadas como patrones de diseño, seguida de SSA (43 %) y Ptidej (33 %). Esto se refleja en valores de *recall* más estables en comparación con las dos herramientas, mejorando

significativamente esta métrica para varios patrones (*Composite* o *Visitor*) o igualando los valores obtenidos por la mejor herramienta (*Singleton*, *Proxy*, *Observer*). SSA sigue una estrategia más conservadora que GEML y Ptidej, lo que reduce el número de falsos positivos (FP) y, por consiguiente, suele obtener valores más altos de *precision*. Este hecho se aprecia especialmente para los patrones *Proxy*, *Strategy*, *Factory method* y *Visitor* donde $Pr = 1$, aunque falla a la hora de detectar el patrón *Bridge*. GEML es más variable en este sentido, aunque la mayoría de los FP se concentran en solo tres patrones de diseño: *Abstract factory* (sin detector disponible en SSA y Ptidej), *Command* y *State*. A pesar de ello, obtiene mejor *precision* que Ptidej para la mayoría de patrones. Cada técnica parece ser superior para un subconjunto diferente de patrones, y no necesariamente divididos según su categoría (creacionales, estructurales o de comportamiento). Los patrones para los que las tres herramientas encuentran la mayoría de implementaciones son *Singleton* y *Template method*, mientras que los más difíciles de detectar son *Factory method*, *Composite* y *Adapter*. Cabe destacar que GEML es capaz de encontrar instancias de patrones no recuperadas por las herramientas para los patrones *State*, *Bridge*, *Visitor* y *Factory method*. En este sentido, GEML es más flexible a la hora de abarcar distintas implementaciones de un mismo patrón gracias a la adaptabilidad al repositorio durante el aprendizaje.

Desde un punto de vista práctico, GEML presenta otras ventajas adicionales respecto a ambas herramientas. En primer lugar, GEML siempre devuelve la definición completa del patrón, esto es, etiqueta una clase por cada rol que compone el patrón. Ptidej, a pesar de identificar correctamente las clases que conforman el patrón, no es preciso a la hora de asignar a cada clase el rol que le corresponde. Es más, devuelve todas las combinaciones posibles, incrementando considerablemente la cantidad de resultados a revisar por parte del ingeniero. SSA también presenta algunas limitaciones respecto a la identificación de roles. Para 12 patrones, esta herramienta no recupera la clase que implementa uno o varios roles del patrón, obligando al ingeniero a revisar manualmente el código para completar la definición del patrón. En segundo lugar, GEML es capaz de detectar implementaciones de cualquier patrón siempre que se cuente con muestras de las que aprender, mientras que SSA y Ptidej requieren extensiones específicas para cubrir cada nuevo patrón. En concreto, ninguna de las dos herramientas soportan la detección de *Abstract factory* e *Iterator*, mientras que Ptidej no incluye otros tres patrones en su catálogo: *Observer*, *Decorator* y *Bridge*. Además, Ptidej considera de forma conjunta los patrones *State* y *Strategy*, requiriendo inspección manual adicional para diferenciar uno de otro.

6. Conclusiones

En este trabajo se presenta el funcionamiento de GEML, un método de detección de patrones de diseño basado en aprendizaje automático con programación genética. GEML es una propuesta reciente que aglutina varias características que lo hacen idóneo para su uso práctico, como su adaptabilidad al repositorio organizacional, la interpretabilidad de sus reglas de decisión y su rendimiento

predictivo, incluso si no se dispone de un número elevado de muestras para aprender. Con GEML, el ingeniero puede comprender, en base a unas pocas reglas, cuáles son las estructuras de microdiseño y métricas software más importantes para describir los distintos patrones implementados en su repositorio.

En el futuro se podrían estudiar otros patrones de diseño, extendiendo el catálogo de estructuras si es necesario. También sería interesante combinar GEML con otras herramientas del área MSR para complementar la toma de decisiones con información obtenida del histórico de versiones o documentación del código.

Referencias

1. Bafandeh Mayvan, B., Rasoolzadegan, A., Ghavidel Yazdi, Z.: The State of the Art on Design Patterns: A systematic mapping of the literature. J. Syst. Softw. **125**, 93–118 (2017)
2. Barbudo, R., Ramírez, A., Servant, F., Romero, J.R.: GEML: A grammar-based evolutionary machine learning approach for design-pattern detection. J. Syst. Softw. **175**, 110919 (2021)
3. Ferenc, R., Beszedes, A., Fulop, L., Lele, J.: Design pattern mining enhanced by machine learning. In: IEEE Int. Conf. Software Maintenance. pp. 295–304 (2005)
4. Fontana, F.A., Zanoni, M., Maggioni, S.: Using Design Pattern Clues to Improve the Precision of Design Pattern Detection Tools. J. Object Technol. **10**(4), 1–31 (2011)
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Pub. (1995)
6. Grosan, C., Abraham, A.: Rule-Based Expert Systems, pp. 149–185. Springer Berlin Heidelberg (2011)
7. Guéhéneuc, Y.G., Sahraoui, H., Zaidi, F.: Fingerprinting design patterns. In: Working Conf. Reverse Engineering. pp. 172–181 (2004)
8. Guéhéneuc, Y.G., Antoniol, G.: DeMIMA: A multilayered approach for design pattern identification. IEEE T Softw. Eng. **34**(5), 667–684 (2008)
9. Kramer, C., Prechelt, L.: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: Working Conf. Reverse Engineering. pp. 208–215 (1996)
10. Mayvan, B.B., Rasoolzadegan, A.: Design pattern detection based on the graph theory. Knowl.-Based Syst. **120**, 211–225 (2017)
11. Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., Meding, W.: The Adoption of Machine Learning Techniques for Software Defect Prediction: An Initial Industrial Validation. In: Joint Conf. Knowl.-Based Softw. Eng. pp. 270–285 (2014)
12. Rasool, G., Mader, P.: Flexible Design Pattern Detection Based on Feature Types. In: IEEE/ACM Int. Conf. Automated Software Engineering. pp. 243–252 (2011)
13. Thaller, H., Linsbauer, L., Egyed, A.: Feature Maps: A Comprehensible Software Representation for Design Pattern Detection. In: Int. Conf. Software Analysis, Evolution, and Reengineering. pp. 207–217 (2019)
14. Tonella, P., Antoniol, G.: Inference of Object-oriented Design Patterns. J Softw. Maint. **13**(5), 309–330 (2001)
15. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T.: Design Pattern Detection Using Similarity Scoring. IEEE T Softw. Eng. **32**(11), 896–909 (2006)
16. Uchiyama, S., Washizaki, H., Fukazawa, Y., Kubo, A.: Design pattern detection using software metrics and machine learning. In: Int. Workshop Model-Driven Software Migration. p. 38 (2011)