Universidad de Córdoba

Máster en Ingeniería Informática

Modelo basado en G3P y clasificación asociativa para la detección de patrones de diseño

Rafael Barbudo Lunar

Director:

Prof. Dr. José Raúl Romero Salguero

Córdoba, febrero de 2018

Índice de contenido

Índice de figuras

Índice de tablas

Resumen

Los patrones de diseño son soluciones generales y reutilizables a un determinado problema de diseño que puede ocurrir durante el desarrollo del software. A pesar de ello, la falta de documentación a menudo dificulta su trazabilidad, provocando que sus implementaciones se pierdan entre miles de líneas de código. La identificación de dichas implementaciones genera múltiples beneficios relacionados con la mantenibilidad y la escalabilidad del software. En este contexto, la detección de patrones de diseño ha atraído una gran atención en el campo de la ingeniería inversa. Las propuestas actuales suelen estar centradas en el estudio, o bien de métricas software, o de propiedades de comportamiento y estructurales.

En este trabajo se propone un modelo en dos fases para la llevar a cabo la detección automática de patrones de diseño a través del uso de técnicas de computación evolutiva y aprendizaje automático. En primer lugar, un algoritmo de programación genética gramatical extrae aquellas propiedades que mejor describen al patrón que se pretende detectar. Este conocimiento se representa como un conjunto de reglas de asociación cuya estructura es definida por una gramática de contexto libre. En segundo lugar, se construye un modelo de detección para identificar las posibles implementaciones del patrón. El uso de la gramática permite el estudio simultáneo de métricas software, así como de propiedades de comportamiento y estructurales del código. El modelo propuesto ha sido empíricamente validado para tres patrones de diseño. Además los resultados obtenidos demuestran la competitividad del modelo frente a las propuestas actuales.

Palabras clave:

Patrones de diseño
Programación genética gramatical
Clasificación asociativa
Ingeniería inversa del software

Capítulo 1

Introducción

Un patrón de diseño (PD) [2] es una solución efectiva y reutilizable, aplicable a la resolución de un determinado problema de diseño durante el desarrollo del software. Su uso da lugar a una serie de beneficios como (1) proporciona un lenguaje y un marco de desarrollo común para los ingenieros; (2) mejora la legibilidad y la mantenibilidad del código; y (3) las soluciones que plantean están probadas y su utilidad ha sido ampliamente demostrada. Esto ha propiciado que los PDs hayan sido aplicados en una gran cantidad de sistemas software. A pesar de ello, su uso no suele ser explícitamente documentado y, por lo tanto, sus implementaciones quedan ocultas entre miles de líneas de código. Desde la perspectiva del mantenimiento del software, la identificación de estos patrones puede aportar varios beneficios. Por ejemplo, puede facilitar la comprensión y la reusabilidad del código. Además, ayuda a comprender las decisiones de diseño seguidas por los desarrolladores. En este contexto, la detección de patrones de diseño (DPD) de un proyecto software se posiciona como una tarea de gran importancia en el campo de la ingeniería inversa del software. Debido a que la inspección manual del código para la DPD es una labor subjetiva y que requiere de un gran esfuerzo, la definición de métodos automáticos ha recibido un gran interés por la comunidad científica. En este contexto, en el que se emplean métodos automáticos para solucionar problemas de ingeniería del software (IS), se enmarca el campo de la Minería de Repositorios Software [3] (MSR, *Mining Software Repositories*). Su objetivo es, mediante el uso de técnicas de minería de datos (MD), descubrir información interesante y previamente desconocida sobre los sistemas software o el proceso por el que se desarrollan.

En este trabajo se presenta un modelo que combina técnicas de computación evolutiva

y aprendizaje automático (ML, *Machine Learning*) para realizar la DPD. El método propuesto consta de dos fases bien diferenciadas. La primera de ellas se encarga de aprender cuales son las características que mejor describen un PD y sus posibles implementaciones. Estas características se pueden referir a (1) métricas software, como el acoplamiento entre clases; (2) propiedades de comportamiento, como la invocación de métodos; y (3) propiedades estructurales, como las relaciones de herencia entre clases. El encargado de realizar este aprendizaje es un algoritmo de programación genética gramatical [4] (G3P, *Grammar Guided Genetic Programming*). Este conocimiento se representa como un conjunto de reglas de asociación (AR, *Association Rule*) de acuerdo a las restricciones impuestas por una gramática de contexto libre (GCL). No obstante, dado el carácter descriptivo de las tareas de minería de reglas de asociación, estas reglas no pueden ser directamente utilizadas para realizar la detección. Por ello, para la segunda fase, se ha adaptado un algoritmo de clasificación asociativa conocido como CMAR [5] (*Classification based on Multiple Association Rules*). La clasificación asociativa (CA) [6] constituye una rama de la MD en la que se integran tareas de minería de reglas de asociación y de clasificación para construir un modelo que se pueda utilizar con fines predictivos.

El modelo propuesto ha sido empíricamente validado para 3 tipos de PDs (*Singleton*, *Adapter* y *Factory Method*). Además, los resultados obtenidos se han comparado con los de la herramienta MARPLE [7] para los datos del repositorio DPB [8] (*Design Pattern detection Benchmark Platform*). Este repositorio contiene implementaciones de PDs de 9 proyectos software reales. Los resultados experimentales demuestran que el modelo aquí propuesto alcanza un gran rendimiento en términos de detección siendo además competitivo con las propuestas actuales. Cabe destacar como el uso de la GCL dota a la propuesta de una mayor flexibilidad y escalabilidad.

El resto del trabajo se organiza como sigue. El Capítulo 2 presenta algunos conceptos básicos e introduce los antecedentes del campo de la MSR y la DPD. En el Capítulo 3 se define en detalle el método propuesto. Esto incluye la definición del algoritmo G3P para la generación de reglas de asociación y del algoritmo de CA para la construcción del modelo de detección. Finalmente, en el Capítulo 4 se desarrolla la fase de experimentación y se muestran los resultados obtenidos, mientras que en el Capítulo 5 se recogen las conclusiones y el trabajo futuro.

Capítulo 2

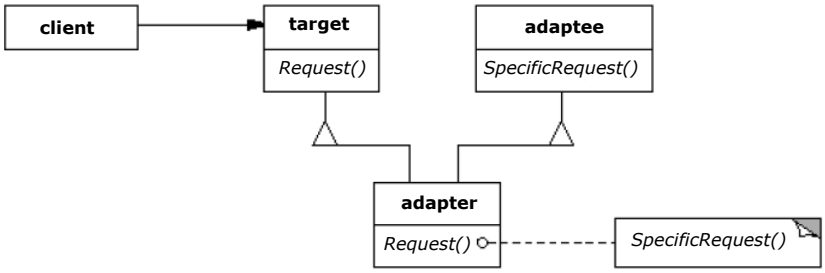
Marco Conceptual

2.1. Patrones de diseño

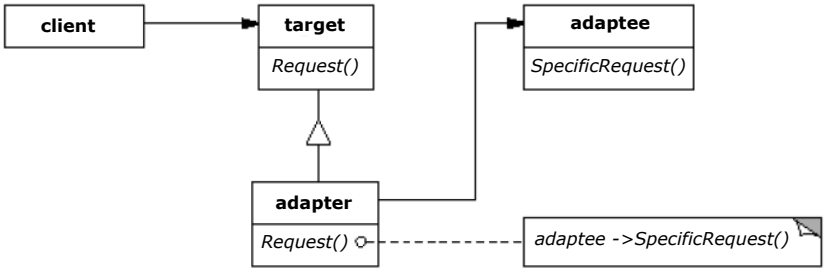
Los patrones de diseño se clasifican en función del problema de diseño que solucionan [2]: los *patrones de comportamiento* se centran fundamentalmente en las interacciones entre clases y objetos; los *patrones creacionales* proporcionan una forma de crear objetos mientras se oculta la complejidad del proceso; y los *patrones estructurales* simplifican la composición de clases y objetos. El *Adapter* es un ejemplo representativo del último grupo, cuyo objetivo es el de permitir el uso de una clase existente (que generalmente proviene de un sistema heredado) cuya interfaz no coincide con la requerida por el sistema.

Cada PD describe un número fijo de roles. Un rol representa una determinada tarea que debe desempeñar un elemento del patrón. En sistemas software orientados a objeto, estos roles son desempeñados por clases o por sus instancias, es decir, objetos. Por ejemplo, siguiendo el ejemplo del patrón *Adapter*, este define cuatro roles diferentes: *adaptee*, *target*, *client* y *adapter*. El rol *adaptee* es desempañado por la interfaz que necesita ser adaptada, mientras que *target* se refiere a la interfaz requerida por el sistema (*client*). Finalmente, el *adapter* define la correspondiente adaptación de los servicios proveídos por *adaptee* al formato requerido por *target*. Es importante destacar que a veces estos roles pueden ser desempeñados por más de una clase u objeto, dificultando el proceso de detección.

Como ya se mencionó con anterioridad, un PD puede ser visto como una plantilla que describe como solucionar un problema de diseño empleando un número de elementos del código. Como consecuencia, pueden existir diferentes implementaciones para un mismo



(a) Alternativa basada en la herencia múltiple



(b) Alternativa basada en la delegación de servicios

Figura 2.1: Alternativas de implementación del patrón *Adapter*

patrón de diseño dependiendo del programador, los requisitos del sistema, etc. En este contexto, la Figura 2.1 muestra dos posibles implementaciones del patrón *Adapter* [2]. Una primera alternativa, mostrada en la Figura 2.1a, se basa en el uso de la herencia múltiple para adaptar una interfaz a otra. Es importante destacar que algunos lenguajes como Java no permiten el uso de este mecanismo de manera directa. Por el contrario, la segunda opción (Figura 2.1b) se basa en la invocación de los métodos de *adaptee* que requieren ser adaptados desde métodos de *adapter* que han sido implementados en base a la interfaz definida por *target*.

En lo que respecta al resto de patrones que serán estudiados en este trabajo, el *Singleton* es un patrón creacional, cuya estructura se muestra en la Figura 2.2. Su objetivo es asegurar que una determinada clase solo tiene una instancia, a la vez que se proporciona un punto de acceso global a la misma. Este patrón define un único rol (*singleton*) el cual es desempeñado, como cabría esperar, por aquella clase que solo puede ser instanciada una vez. El *Factory Method* es otro patrón creacional el cual define una interfaz para crear un objeto, pero permite a las subclases decidir que clase se instancia. Este patrón define 4 roles: *product*, *concrete product*, *creator* y *concrete creator*. Más específicamente, *product* define la interfaz

de los objetos que el patrón va a crear, mientras que *concrete product* implementa dicha interfaz. Por otra parte, *creator* declara el método encargado de la creación de los objetos del tipo *product*, siendo *concrete creator* el responsable de implementar dicho método para crear un *concrete product*. En la Figura 2.3 se muestra la estructura de este patrón.

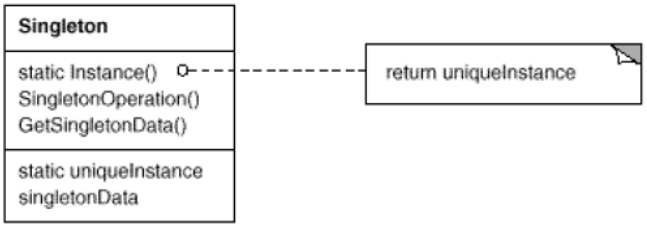


Figura 2.2: Estructura del patrón *Singleton*

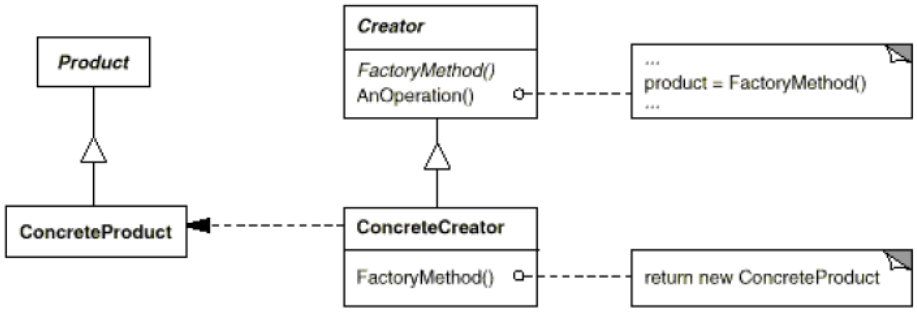


Figura 2.3: Estructura del patrón *Factory Method*

2.2. Minería de Repositorios Software

La MSR es la rama de la inteligencia artificial destinada a la resolución de problemas del área de IS mediante la aplicación de técnicas propias de la MD. En este contexto, las tareas de la IS deben ser formuladas como problemas susceptibles de ser resueltos mediante técnicas de análisis de datos. Estos datos se generan durante el proceso de desarrollo y provienen tanto del código fuente como de las plataformas para la gestión y configuración de sistemas software [9]. Destacan los sistemas de control de versiones, los foros de desarrolladores, los *issue tracker*, las listas de correo o las propias plataformas de descarga y venta de aplicaciones como *Google Play* o *AppStore* [10].

El campo de la MSR ha experimentado una gran evolución en los últimos 10 años. Esto ha provocado que no solo se empleen técnicas de MD cada vez más sofisticadas, sino

que también se exploren nuevas fuentes de datos. Actualmente, la mayoría de trabajos se centran en el análisis y estudio de defectos (*bugs*) en los sistemas software. Dichos trabajos suelen estar relacionados con su corrección [11] o con la predicción de futuras ocurrencias [12]. Otras propuestas se centran en dar soporte al proceso de desarrollo [13], la refactorización del código [14] o al estudio de la evolución del software [15]. Destacan también otras líneas de investigación cuyo objetivo es desarrollar técnicas escalables que permitan trabajar con grandes volúmenes de datos [16, 17, 18].

2.3. Detección de patrones de diseño

En 1996 se desarrolló el sistema Pat [19], el cual fue pionero en la aplicación de técnicas automáticas para la DPD. Desde entonces han sido muchos los trabajos que se han publicado en este campo. El interés ha sido tal que varios autores han tratado de resumir el estado del arte [20, 21, 22]. Los trabajos para la DPD se pueden categorizar según el método de búsqueda empleado. Así, por ejemplo, nos encontramos con propuestas que utilizan técnicas de *similarity scoring* [23, 24]. Estos trabajos representan la información estructural del código y del patrón (generalmente en forma de grafos o matrices) para buscar estructuras similares a las del patrón dentro del código del proyecto. Así, por ejemplo Mayvan *et al.* [25] representan la estructura del proyecto y patrón como un grafo etiquetado donde los nodos identifican el tipo de un artefacto del código y las aristas el tipo de relación entre dos artefactos (Figura 2.4). A partir de estas representaciones, un algoritmo de búsqueda de subgrafos, lleva a cabo el proceso de detección. Otros trabajos se sirven de técnicas de razonamiento lógico [26], las cuales realizan la detección a partir de la definición manual de una serie de propiedades de los patrones. La principal limitación de estas técnicas es que solo permiten buscar definiciones muy estrictas de un PD. Para contrarrestarlo, algunos autores han propuesto el uso de técnicas de lógica difusa [27], aunque suelen generar muchos falsos positivos. Las técnicas basadas en el análisis de conceptos formales [28] emplean la lógica y métodos matemáticos para detectar grupos de elementos con características similares, es decir, implementaciones de PDs. El principal problema de estas técnicas es que son muy costosas computacionalmente.

Todos los trabajos anteriormente citados, aunque en mayor o menor medida, requieren de la inyección de conocimiento por parte de uno o varios expertos que definan cuáles

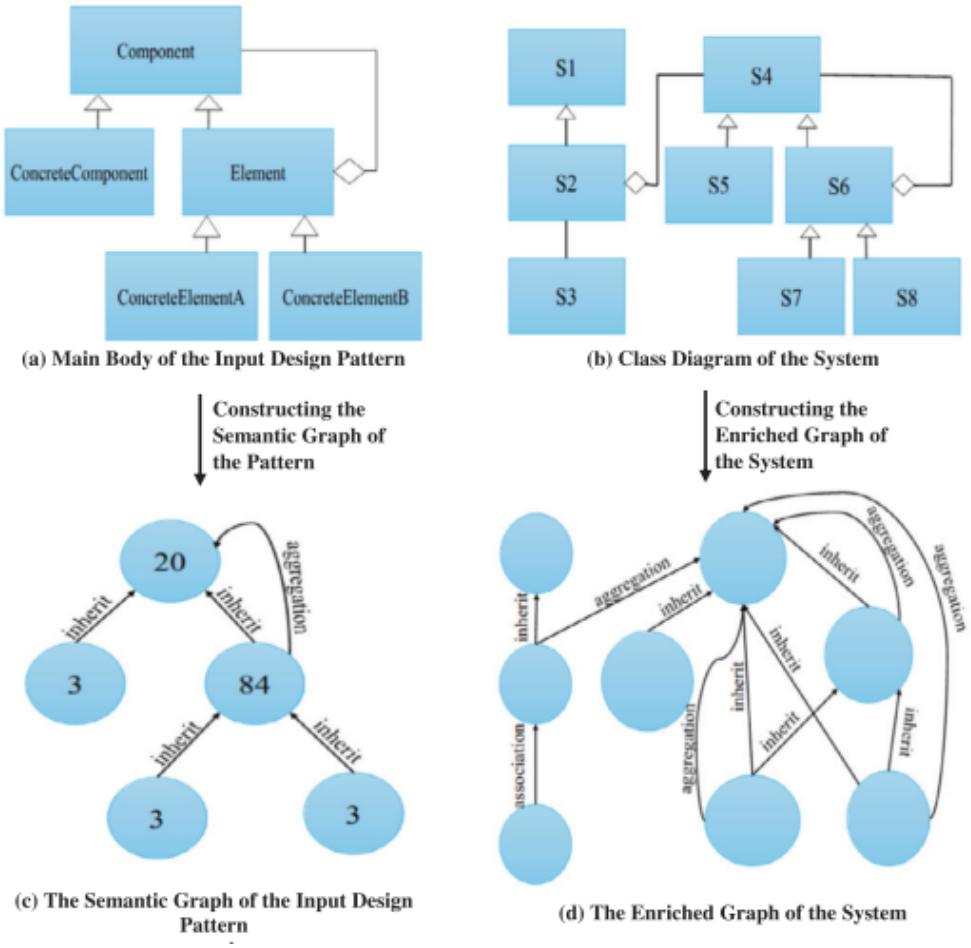


Figura 2.4: Ejemplo de *similarity scoring*

son las propiedades de un determinado PD. Este conocimiento puede no ser suficiente para identificar las posibles variantes de un determinado patrón ya que solo alberga la experiencia y el conocimiento de un grupo de desarrolladores. Por ello, algunos métodos han introducido cierto grado de flexibilidad a la hora de realizar la detección. En este contexto, se ha propuesto el uso de técnicas de ML, las cuales por lo general, son más flexibles y permiten aprender un gran número de variantes de un determinado PD a partir del estudio de repositorios software.

Un trabajo pionero en la aplicación de técnicas de ML en el contexto de la DPD fue la propuesta de Ferenc *et al.* [29]: en él se utilizaban árboles de decisión y redes neuronales para filtrar los falsos positivos que se obtenían mediante un análisis estructural del código [30]. De un modo similar, en [31] se emplea la herramienta DeMIMA [32] para localizar un conjunto de patrones candidatos a partir de una base de conocimiento definida

manualmente. Posteriormente, se emplea el algoritmo JR_{IP} de *Weka* para reducir el número de falsos positivos. En ambos casos, el algoritmo de ML emplea métricas software como entrada. En [1] se propone un algoritmo de clasificación propio para llevar a cabo la detección. Las instancias de dicho algoritmo se codifican como vectores de características formados por $n*k$ valores numéricos, donde n es el número de roles del patrón y k son las métricas software analizadas. Además, se reserva un campo adicional para identificar el tipo de patrón que implementa la instancia (Figura 2.5). Una vez se ha generado el modelo de detección, se emplea un algoritmo de búsqueda de subgrafos isomórficos para, en base a la definición de la estructura de un patrón, extraer candidatos a patrones a partir de las relaciones entre los artefactos del código. En [33, 34] también se emplean métricas software como entradas del algoritmo de ML, sin embargo, en este caso se detectan clases potenciales que podrían desempeñar un determinado rol del PD que se está identificando. A continuación, se emplea esta información para buscar patrones completos. Por el contrario, en [7] se presenta la herramienta MARPLE, la cual emplea propiedades de comportamiento y estructurales del código como entrada para el algoritmo de ML. Es importante destacar que no existe ningún método lo suficientemente flexible que combine métricas software así como propiedades estructurales y de comportamiento para describir a los PDs y sus variantes.

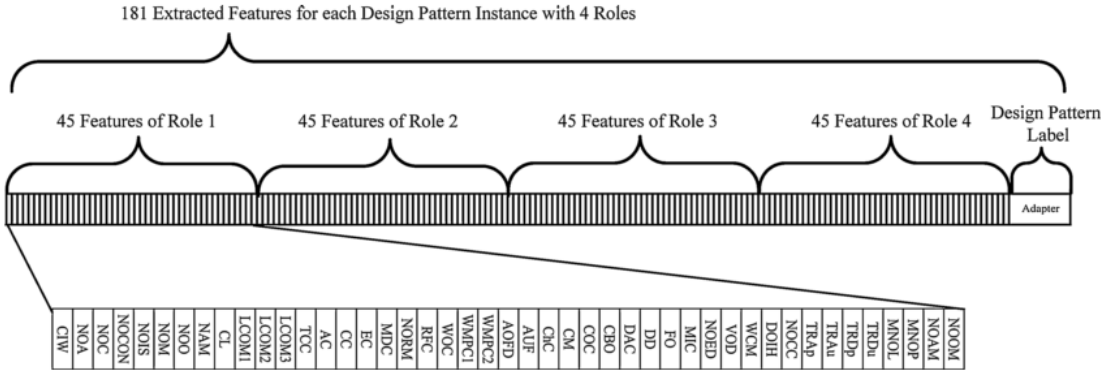


Figura 2.5: Vector de características para detección de Chihada *et al.* [1]

Capítulo 3

Modelo propuesto

La Figura 3.1 muestra el modelo propuesto para realizar la DPD. Como se puede apreciar, la propuesta consta de dos fases bien diferenciadas. La primera se encarga de aprender cuales son las propiedades que mejor describen a un PD y sus variantes, mediante el estudio de un repositorio sobre el uso previo de PDs. Las propiedades que describen estos PDs se representan como un conjunto de ARs cuya estructura es definida por una GCL que establece una serie de restricciones formales [35]. El uso de esta gramática dota a la propuesta de una gran flexibilidad ya que permite el estudio simultáneo de métricas software, así como de propiedades estructurales y de comportamiento.

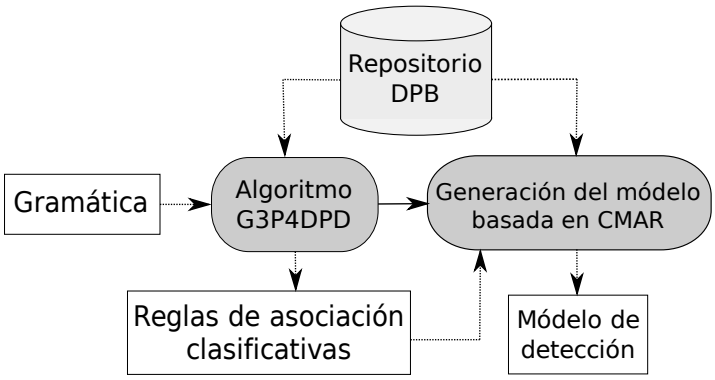


Figura 3.1: Esquema del modelo propuesto

El responsable de la generación de estas reglas es el algoritmo G3P para la DPD (G3P4DPD). Las reglas, que se obtienen en base a la GCL, se denominan reglas de asociación clasificativas [36] (CAR, *Class Association Rule*). Una CAR es un tipo de regla de asociación cuyo consecuente se limita a identificar la clase que se está describiendo (imple-

mentar o no un PD). No obstante, dada la naturaleza descriptiva de las reglas de asociación, las reglas obtenidas no pueden ser directamente utilizadas para realizar la detección. Por ello, para la segunda fase se ha adaptado el algoritmo CMAR para la generación de un clasificador. Este algoritmo ha sido seleccionado ya que se ha demostrado empíricamente que obtiene buenos resultados y a que puede ser integrado con el algoritmo G3P4DPD con relativa facilidad.

3.1. Descripción del algoritmo G3P

El Algoritmo 1 muestra el esquema general del evolutivo. Como se puede apreciar, precisa de cinco entradas: el número máximo de generaciones (*maxGen*), el tamaño de la población (*popSize*), el número de reglas a devolver (*extPopSize*), la gramática (*grammar*) y el repositorio de patrones (*repo*). Este repositorio está compuesto por un conjunto de instancias de PDs (positivas y negativas) y su código fuente. Cada una de estas instancias define los elementos que componen el patrón, así como los roles que desempeñan (mapeo de roles). Nótese como el algoritmo crea una población externa (*extPop*) compuesta por las mejores reglas obtenidas a lo largo del proceso evolutivo. Esta población constituye la salida del algoritmo G3P4DPD.

Algoritmo 1: Extracción de reglas de asociación

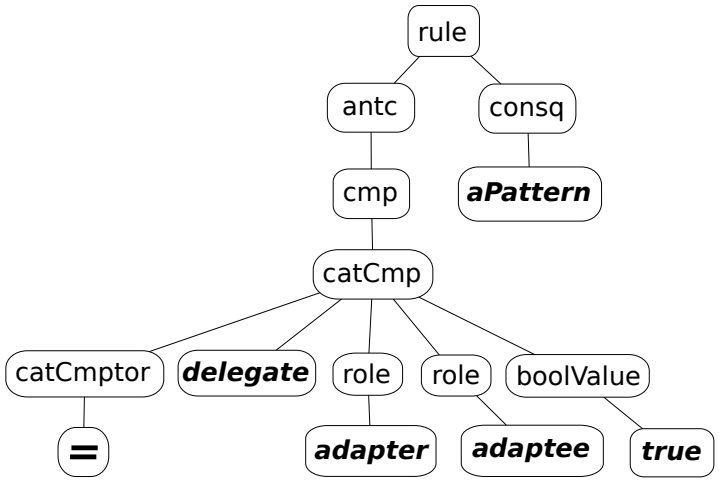
```
Entrada: maxGen, popSize, extPopSize, grammar, repo
Salida  : extPop
pop ← generateRules(popSize, grammar)
extPop ← ∅
evaluate(pop, repo)
while generation < maxGen do
    pop ← select(pop ∪ extPop, popSize)
    pop ← crossover(pop)
    if randon < 0.5 then
        pop ← diversityMutator(pop);
    else
        pop ← dpdMutator(pop);
    end
    evaluate(pop, repo)
    extPop ← update(pop ∪ extPop, extPopSize)
    generation++
end
```

El algoritmo comienza generando, de manera aleatoria, *popSize* individuos en base a las restricciones formales definidas por *grammar* (Sección 3.1.1). La población externa (*extPop*) es inicializada al conjunto vacío. A continuación, se evalúan los individuos de *pop* (véase la Sección 3.1.2). A partir de este punto, el algoritmo itera hasta que se alcanza *maxGen*. En este contexto, los individuos se evolucionan a lo largo de dichas iteraciones mediante la aplicación de operadores genéticos. En primer lugar, el operador de selección escoge *popSize* individuos de la unión entre *pop* y *extPop*. Para este trabajo se ha escogido el torneo binario, el cual toma dos individuos de manera aleatoria y selecciona el que tiene mejor *fitness*. Este proceso se repite hasta que se obtienen los *popSize* individuos. A continuación, el operador de cruce es aplicado (Sección 3.1.3), con una determinada probabilidad, sobre pares de individuos para combinar sus genotipos. Por último, uno de dos posibles operadores de mutación (Sección 3.1.4) es aplicado. La selección se realiza de manera aleatoria teniendo ambos la misma probabilidad de ser aplicados. El primer mutador (*diversityMutator*) es un operador genérico que promueve la diversidad entre los individuos, mientras que el segundo (*dpdMutator*) es un operador específico del dominio de la DPD. Una vez se han aplicado estos operadores se actualiza la población externa con las mejores reglas (Sección 3.1.5).

3.1.1. Representación de los individuos

Cada individuo del algoritmo G3P4DPD se compone de dos elementos: (1) un genotipo, representado en forma de árbol sintáctico y (2) un fenotipo que representa la CAR. La Figura 3.2a muestra el genotipo de un individuo de ejemplo, mientras que la Figura 3.2b muestra su correspondiente fenotipo, es decir, la regla.

La Figura 3.3 muestra la gramática (G) empleada por el algoritmo G3P4DPD, la cual se compone de: un símbolo raíz (S) a partir del cual se inicia el proceso de derivación; un conjunto de símbolos no terminales ($\sum_N$) delimitados por <...>; un conjunto de símbolos terminales ($\sum_T$) representados en cursiva; y un conjunto de reglas de producción (P) que indican las posibles derivaciones de los símbolos no terminales. El símbolo raíz es <*rule*> y se deriva en <*ant*> y <*consq*>, los cuales representan el antecedente y consecuente de la regla, respectivamente. Por una parte, el antecedente está formado por una serie de comparaciones (<*cmp*>), las cuales pueden ser numéricas (<*numCmp*>) o



(a) Ejemplo de genotipo

(= delegate adapter adaptee true) → (aPattern)

(b) Ejemplo de fenotipo

Figura 3.2: Ejemplo de representación de un individuo

categóricas (<*catCmp*>). Independientemente del tipo, cada comparación está formada por un comparador, un operador, un conjunto de argumentos y un valor. Más concretamente, las comparaciones numéricas están formadas por un comparador numérico ($>$, $<$, $\geq$ o $\leq$), un operador numérico (p.ej. *NOC)*, un rol como argumento del operador, y un valor constante (*const)*. Nótese como estos operadores se corresponden con métricas software [37]. Un ejemplo de comparación numérica podría ser "`NOC(target)<1`", la cual indica que la clase que desempeña el rol *target* no tiene ningún hijo (véase la Tabla 3.1). Por otra parte, las comparaciones categóricas se componen de un comparador categórico ($=$ o !=), un operador categórico (p.ej. *delegate* o *typeOf*), un número variable de roles como argumentos, y un posible valor asociado a dicho operador. En este caso, un ejemplo podría ser "`linkArtefact(adapter,adaptee)=directInherit`", la cual indica que la clase que desempeña el rol *adapter* es una subclase directa de *adaptee*. El conjunto completo de operadores, así como sus argumentos y los valores devueltos, se recogen en la Tabla 3.1. En relación al consecuente (<*consq*>), este solo puede ser derivado en dos posibles terminales, *aPattern* y *notAPattern*. El primero indica que la regla hace referencia a un PD, mientras que el caso opuesto lo representa el segundo terminal.

G = (S, $\sum_N$, $\sum_T$, P)

S = <rule>

$\sum_N$ = {<rule>, <antc>, <consq>, <cmp>, <numCmp>, <catCmp>, <numCmptor>, <catCmptor>, <numOp>, <role>, <boolValue> <typeOfValue>, <linkMethodValue>, <linkArtefactValue>, <ctorVisibilityValue>, <aggregationValue>, <adapterValue>}

$\sum_T$ = {*and*, $\geq$, $\leq$, $>$, $<$, $=$, $!=$, *NOM*, *NOC*, *DIT*, *RFC*, *adapter*, *adaptee*, *target*, *true*, *false*, *isFinal*, *isSubclass*, *controlledInit*, *staticField*, *staticFlag*, *conglomeration*, *returned*, *received*, *createObj*, *delegate*, *sameElem*, *typeOf*, *linkMethod*, *linkArtefact*, *ctorVisibility*, *aggregation*, *adapter*, *class*, *absClass*, *enum*, *intface*, *directOver*, *indirOver*, *directImpl*, *indirImpl*, *notLinked*, *directInherit*, *indirInherit*, *private*, *protected*, *package*, *public*, *default*, *decl*, *inhr*, *aPattern*, *notAPattern*}

P =

<rule> ::= <antc> <consq>
<antc> ::= <cmp> | *and* <antc> <cmp>
<cmp> ::= <numCmp> | <catCmp>
<numCmp> ::= <numCmptor> <numOp> <role> *const*
<catCmp> ::= <catCmptor> *isFinal* <role> <boolValue>
| <catCmptor> *delegate* <role> <role <boolValue>
| <catCmptor> *typeOf* <role> <typeOfValue>
| <catCmptor> *linkArtefact* <role> <role> <linkArtefactValue>
| ...
<numCmptor> ::= $\geq$ | $\leq$ | $>$ | $<$
<catCmptor> ::= $=$ | $!=$
<numOp> ::= *DIT* | *NOC* | *CBO* | *NOM*
<role> ::= *adapter* | *adaptee* | *target*
<boolValue> ::= *true* | *false*
<typeOfValue> ::= *class* | *absClass* | *intface* | *enum*
<linkArtefactValue> ::= *directInherit* | *indirInherit* | *directImpl* | *indirImpl* | *notLinked*
<consq> ::= *aPattern* | *notAPattern*

Figura 3.3: Gramática empleada por el algoritmo G3P4DPD

Operadores numéricos	
Signatura	**Descripción**
NOM(R_1)	Número de métodos de R_1
NOC(R_1)	Número de subclases directas de R_1
DIT(R_1)	Profundidad en el árbol de herencia a la que se encuentra R_1
RFC(R_1)	Número de métodos y constructores que pueden ser invocados cuando un objeto del tipo de R_1 recibe un mensaje
Operadores categóricos	
Signatura	**Descripción**
isFinal(R_1)	*true* si R_1 no puede ser heredado, *false* en caso contrario
isSubclass(R_1)	*true* si R_1 es una subclase, *false* en caso contrario
controlledInit(R_1)	*true* si R_1 se instancia así mismo dentro de un bloque *if* o *while*, *false* en caso contrario
staticField(R_1)	*true* si R_1 tiene un campo estático, *false* en caso contrario
staticFlag(R_1)	*true* si R_1 tiene un campo estático y booleano, *false* en caso contrario
conglomeration(R_1)	*true* si R_1 declara algún método que llame al menos a otros 2 métodos de R_1, *false* en caso contrario
returned(R_1,R_2)	*true* si algún método declarado en R_1 devuelve un elemento del tipo R_2, *false* en caso contrario
received(R_1,R_2)	*true* si algún método declarado en R_1 recibe un elemento del tipo R_2 como argumento, *false* en caso contrario
createObj(R_1,R_2)	*true* si R_1 instancia a R_2, *false* en caso contrario
delegate(R_1,R_2)	*true* si R_1 invoca algún método de R_2, *false* en caso contrario
sameElem(R_1,R_2)	*true* si R_1 y R_2 son el mismo artefacto, *false* en caso contrario
typeOf(R_1)	Devuelve el tipo del artefacto que implementa a R_1 (*class*, *absClass*, *enum* o *intface*)
linkMethod(R_1,R_2)	Devuelve *directOver*, *indirOver*, *directImpl* o *indirImpl* si R_1 directa o indirectamente sobrescribe o implementa un método de R_2, *notLinked* en cualquier otro caso
linkArtefact(R_1,R_2)	Devuelve *directInherit*, *indirInherit*, *directImpl*, *indirImpl* si R_1 directa o indirectamente extiende o implementa a R_2, *notLinked* en cualquier otro caso
ctorVisibility(R_1)	Devuelve la visibilidad del constructor menos restrictivo de R_1, i.e. *private, protected, package, public or default*
aggregation(R_1,R_2)	Devuelve información de un atributo del tipo de R_2 declarado en R_1 en términos de su visibilidad y de su instanciabilidad.
adapter(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 cualquier otro caso

Tabla 3.1: Operadores numéricos y categóricos

Es importante destacar como la mayoría de operadores categóricos se basan en los *elemental design patterns* [38], *micro patterns* [39] y *design pattern clues* [7], los cuales

pueden ser visto como operadores categóricos que comprueban si un fragmento de código satisface o no una determinada propiedad en base a valores de *true* o *false*. Por ejemplo, *Abstract Interface* es un *elemental design pattern* que comprueba si un artefacto del código es abstracto o no. En algunos lenguajes como Java, un artefacto no solo puede ser una clase abstracta o concreta sino que además puede ser una interfaz. Este problema podría ser fácilmente solucionado añadiendo diferentes operadores como *isAbstract*, *isConcrete* o *isInterface*. No obstante, los predicados resultantes estarían fuertemente correlados y podrían introducir ruido al proceso de aprendizaje. En este contexto, la GCL permite el uso de operadores categóricos que pueden devolver múltiples valores diferentes (véase el operador *typeOf* en la Tabla 3.1).

3.1.2. Evaluación de individuos

La evaluación es el proceso por el que se mide la aptitud (*fitness*) de un determinado individuo. Para este caso, el *fitness* se corresponde con una medida ampliamente extendida en el contexto de las ARs, el soporte (Ecuación 3.1). Esta métrica mide la frecuencia de aparición de una determinada regla R en el conjunto de datos T. Siendo X la unión de los *itemsets* del antecedente y consecuente de R, el soporte de R con respecto a T se define como la proporción de transacciones t en el *dataset* que contienen a X.

$$soporte(R) = \frac{|\{t \in T; X \subseteq t\}|}{|T|} \tag{3.1}$$

3.1.3. Operador de cruce

El operador de cruce combina el genotipo de dos individuos para generar otros dos. Este operador selecciona, de manera aleatoria, una comparación de los genotipos de cada uno de los padres y las intercambia. Para localizar una comparación en el genotipo, el árbol es recorrido en pre-orden hasta que se encuentra el símbolo <*cmp*>. El final de dicha comparación se alcanza cuando se encuentra uno de los siguientes símbolos: *and*, <*cmp*> o <*consq*>. El no delimitar correctamente una comparación podría ocasionar que se violase la firma de los operadores. Así, por ejemplo, se podrían generar comparaciones inválidas como "`delegate(adapter,NOC)`", en la que un operador recibe a otro (sin argumentos) como entrada. La Figura 3.4 muestra el funcionamiento del operador gráficamente con

un ejemplo. Al final del proceso, el operador devuelve los dos mejores individuos entre el conjunto de padres e hijos.

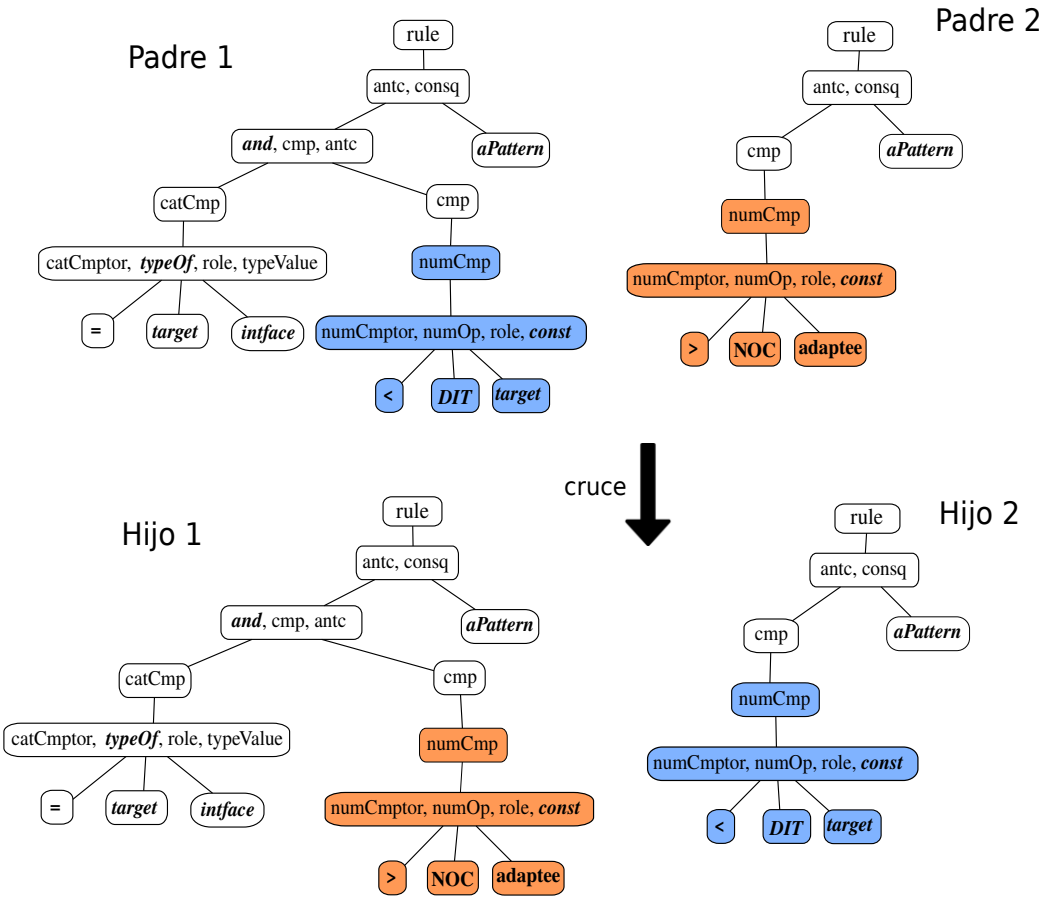


Figura 3.4: Ejemplo de aplicación del operador de cruce

3.1.4. Operadores de mutación

Tras realizar el cruce, el algoritmo G3P4DPD puede aplicar dos operadores de mutación diferentes. El primero (*diversityMutator*) selecciona un número de comparaciones del antecedente de la regla y las reconstruye de manera aleatoria desde el símbolo <*cmp*>. El número de comparaciones a reconstruir se elige mediante una ruleta aleatoria, la cual promueve las pequeñas modificaciones. De este modo, lo más probable es que únicamente se reconstruya una comparación. En la Figura 3.5 se ilustra este proceso. Por otra parte, el segundo operador de mutación (*dpdMutator*) recorre todas las comparaciones de la regla y las niegan con una determinada probabilidad, la inversa del número de comparaciones. Para negar una determinada comparación (numérica o categórica), el operador se

remplaza por su opuesto (p.ej. $>$ reemplazaría a $\leq$). Además, el terminal encargado de describir la clase también puede ser cambiado por su opuesto con un 50 % de probabilidades. Este operador permite obtener reglas que describan tanto a la clase positiva como a la negativa. Así, por ejemplo, si la comparación "`ctorVisibility(singlegon)=private`" está describiendo una propiedad que ocurre frecuentemente en las instancias positivas, es probable que "`ctorVisibility(singlegon)!=private`" describa a las instancias negativas. En la Figura 3.6 se muestra un ejemplo del mecanismo descrito. Independientemente del operador que se aplique, al final se devuelve el mejor individuo (padre o hijo).

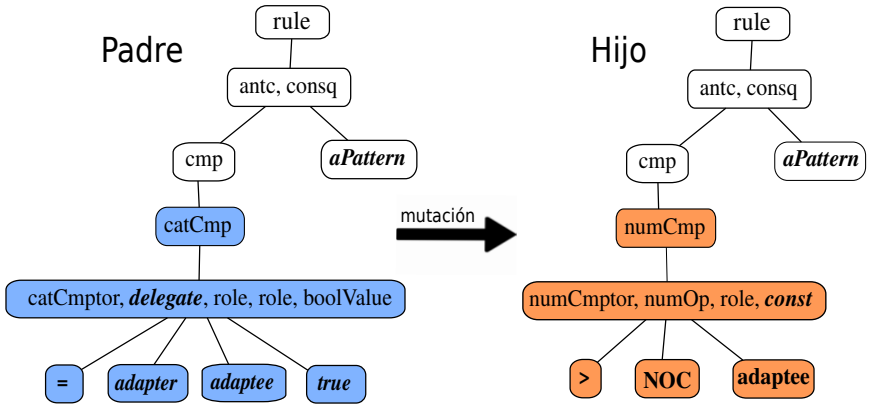


Figura 3.5: Ejemplo de aplicación del operador *diversityMutator*

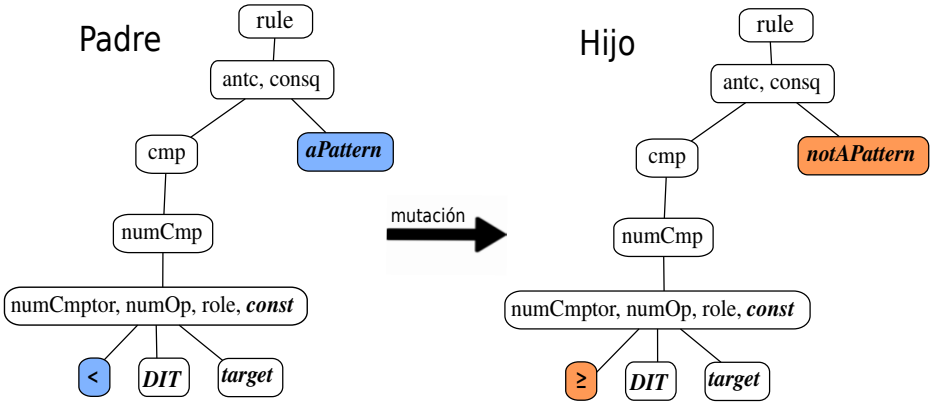


Figura 3.6: Ejemplo de aplicación del operador *dpdMutator*

3.1.5. Actualización de la población externa

Durante la ejecución del algoritmo G3P4DPD se genera una gran cantidad de ARs. Sin embargo, al final solo se devuelve un pequeño subconjunto formado por las mejores,

la población externa (*extPop*). Esta población se actualiza uniendo y filtrando a los individuos de la población actual (*pop*) y de la población externa de la generación anterior. En primer lugar, se ordenan las reglas en base a su confianza. La confianza (Ecuación 3.2) mide con qué frecuencia se cumple el consecuente (C) en las reglas en las que además se cumple el antecedente (A). A continuación, estas reglas son filtradas en base al concepto de precedencia. Dadas dos reglas R_1 y R_2, se dice que R_1 precede a R_2 si y solo sí se cumple alguna de las siguientes condiciones:

- confianza(R_1) > confianza(R_2)
- confianza(R_1) = confianza(R_2) y soporte(R_1) > soporte(R_2)
- confianza(R_1) = confianza(R_2), soporte(R_1) = soporte(R_2) y el número de atributos de R_1 es menor que el de R_2

En base a esto, si una regla A → C precede a otra A' → C' y A ⊂ A', entonces A' → C' es descartada. A continuación, aquellas reglas que no superan un determinado umbral de soporte y confianza son descartadas. Además, se realiza un análisis estadístico para comprobar si el antecedente y el consecuente de la regla están positivamente correlados. Esto ocurre cuando el valor del estadístico χ^2 (Ecuación 3.3) es superior a un determinado umbral. En el caso de que no lo estén, la regla es descartada. De este modo se garantiza que las reglas presenten implicaciones fuertes para poder ser usadas en clasificación.

$$confianza(A \rightarrow C) = \frac{soporte(A \cup C)}{soporte(A)} \tag{3.2}$$

$$\chi^2 = \frac{n * (lift(R) - 1)^2 * soporte(R) * confianza(R)}{(confianza(R) - soporte(R)) * (lift(R) - confianza(R))} \tag{3.3}$$

$$donde \quad lift(A \rightarrow C) = \frac{soporte(A \cup C)}{soporte(A) * soporte(C)}$$

3.2. Descripción del algoritmo de clasificación asociativa

El algoritmo de clasificación asociativa CMAR consta de dos fases. En la primera se ejecuta una variante del algoritmo *FP-Growth* [40] para generar todas las CARs que

superen un determinado umbral de soporte y confianza. No obstante, esto no garantiza que las reglas obtenidas sean aptas para su uso en clasificación, por lo que CMAR define tres estrategias de poda. En primer lugar, para cada regla generada se comprueba si su antecedente y consecuente están positivamente correlados. Si es el caso, la regla es guardada y un método de poda basado en el uso de reglas con mayor precedencia se ejecuta sobre todas las reglas almacenadas. Una vez se han generado todas las reglas se utiliza el concepto de cobertura del *dataset* para filtrar aquellas reglas que representan a un menor número de instancias. Las reglas restantes componen el clasificador cuyo objetivo es determinar si una instancia desconocida implementa o no un PD en la segunda fase. Para ello, este clasificador busca todas las ARs para las que se satisface el antecedente. Si todas las reglas tiene el mismo consecuente, CMAR simplemente asigna su etiqueta a la instancia. Por el contrario, si los consecuentes no coinciden, CMAR divide las reglas en tantos grupos como consecuentes diferentes hubiese. Por último, el algoritmo selecciona que grupo es el más representativo en función del valor ponderado de χ^2 y asigna su etiqueta a la instancia.

A pesar de ello, métodos rígidos como FP-Growth podrían afectar a la aplicabilidad y la flexibilidad del modelo propuesto. Estos métodos, además de ser exhaustivos, trabajan con *datasets* con formatos bastante estrictos en los que se tienen que definir de manera explícita todos los atributos (numéricos y categóricos) que lo conforman. Como consecuencia de ello, para este trabajo únicamente se ha adoptado la segunda fase del algoritmo CMAR. Nótese como el número de atributos puede verse incrementado rápidamente al analizar patrones más complejos, es decir, con un mayor número de roles. Para cada operador de la gramática se generarían varios atributos en función de las posibles combinaciones de argumentos que presentase. Así, por ejemplo "`delegate(adapter,adaptee)`" sería un atributo diferente a "`delegate(target, adapter)`". Es por ello por lo que en este trabajo se emplea el algoritmo G3P4DPD para la generación de reglas. Es importante destacar como las dos primeras estrategias de poda de CMAR han sido integradas durante la actualización de la población externa, mientras que la tercera se ejecuta al final del proceso evolutivo.

Capítulo 4

Experimentación

4.1. Marco experimental

El algoritmo G3P4DPD para la extracción de las reglas de asociación se ha desarrollado en Java tomando como base el *framework* JCLEC [41] (*Java Class Library for Evolutionary Computation*). Para la extracción de propiedades software se han empleado tres librerías de código libre también desarrolladas en Java. Por una parte, para la obtención de métricas software se ha empleado la librería ckjm[1] (*Chidamber and Kemerer Java Metrics*). Por otra parte, para la extracción de propiedades de comportamiento y estructurales se han empleado las librerías *Java Parser*[2] y Javassist[3], las cuales extraen dichas propiedades del código fuente y del *bytecode*, respectivamente.

Las instancias usadas para validar la propuesta han sido extraídas del repositorio DPB, el cual ha sido creado por los autores de MARPLE. Dicho repositorio contiene instancias de 9 proyectos reales escritos en Java. Además, incluye otro proyecto que recoge implementaciones de patrones disponibles en la bibliografía. El motivo por el que se ha seleccionado este repositorio frente a otras alternativas como P-MARt [42] es que DPB es el único que alberga instancias de la clase positiva y negativa. La Tabla 4.1 muestra las características de los proyectos usados en la experimentación, donde UC representa el número de unidades de compilación, es decir, ficheros de código que componen el proyecto. Además, la Tabla 4.2 muestra la cantidad de instancias positivas y negativas para los PDs que se van a analizar.

[1] ckjm: *https://www.spinellis.gr/sw/ckjm*
[2] Java Parser: *https://javaparser.org*
[3] Javassist: *http://jboss-javassist.github.io/javassist/*

Proyecto	UC	Paquetes	Tipos	Métodos	Atributos	LOC
DesignPatternExample	1060	235	1749	4710	1786	32313
QuickUML 2001	156	11	230	1082	421	9233
Lexi v0.1.1 alpha	24	6	100	677	229	7101
JRefactory v2.6.24	569	49	578	4883	902	79732
Netbeans v1.0.x	2444	184	6278	28568	7611	317542
JUnit v3.7	78	10	104	648	138	4956
JHotDraw v5.1	155	11	174	1316	331	8876
MapperXML v1.9.7	217	25	257	2120	691	14928
Nutch v0.4	165	19	335	1854	1309	23579
PMD v1.8	446	35	519	3665	1463	41554

Tabla 4.1: Proyectos software analizados

Patrón de diseño	Positivas	Negativas	Totales
Singleton	58	96	154
Adapter	607	603	1210
Factory Method	561	481	1042

Tabla 4.2: Distribución de las instancias del dataset

La configuración empleada por el algoritmo G3P4DPD y CMAR se muestra en la Tabla 4.3. El tamaño de la población externa ha sido fijado a un valor elevado (500) para evitar que se descarten reglas interesantes. Por otra parte, el número máximo de derivaciones sirve para delimitar el tamaño del genotipo de los individuos y, por lo tanto, el de las reglas de asociación. Este valor se ha fijado a un valor pequeño (8) ya que valores grandes dificultarían la generación de reglas con el soporte necesario. Además, como ya se mencionó, las reglas genéricas tienen mayor precedencia. Con respecto a los umbrales de soporte, confianza, estadístico χ^2 y cobertura, se han tomado los valores por defecto empleados por CMAR [5].

Parámetro	Valor
Número de generaciones	100
Tamaño de la población	100
Tamaño de la población externa	500
Probabilidad de cruce	0.8
Número máximo de derivaciones	8
Umbral de soporte	0.01
Umbral de confianza	0.5
Umbral del estadístico χ^2	3.841
Umbral de cobertura	4

Tabla 4.3: Configuración experimental

	Singleton	Adapter	F. Method
NOM			✓
NOC	✓	✓	✓
DIT	✓		✓
RFC			✓
isFinal	✓		
isSubclass			✓
controlledInit	✓		
staticField	✓		
staticFlag	✓		
conglomeration			✓
returned	✓	✓	✓
received		✓	
createObj	✓	✓	✓
delegate		✓	
sameElem			✓
typeOf	✓	✓	✓
linkMethod		✓	✓
linkArtefact		✓	✓
ctorVisibility	✓		
aggregation	✓	✓	✓
adapter		✓	

Tabla 4.4: Operadores usados en la experimentación

Es importante destacar que la gramática de entrada ha sido parcialmente modificada para el estudio de cada PD. En primer lugar, se han modificado las reglas de producción del símbolo no terminal <*role*> para que este se derive en los roles correspondientes del patrón que se está estudiando. Además, se han empleado diferentes operadores (categóricos y numéricos) para describir las propiedades de cada patrón. Por ejemplo, para describir el patrón *Singleton* se han omitido algunos operadores categóricos como *linkArtefact*, ya que precisa de dos roles diferentes como argumentos. Es importante destacar que algunos operadores, como *aggregation*, sí pueden recibir el mismo rol dos veces como argumento, ya que una clase puede definir un atributo de su mismo tipo. Así pues, la Tabla 4.4 muestra qué operadores se han empleado para la detección de cada patrón. Por una parte, los operadores categóricos se han seleccionado en base a su estructura y colaboraciones según lo definido por Gamma *et al.* [2]. Además, los *design pattern clues*, empleados por Zanoni *et al.* [7], también han sido usados como referencia. Por otra parte, la selección de operadores numéricos para el *Factory Method* se ha realizado en base al estudio realizado por Issaoui

et al. [43]. Para todos los demás casos, la selección de operadores se ha realizado en base a experimentos previos.

Para poder establecer un marco experimental, es necesario considerar que, hasta donde se conoce, no existe ningún método lo suficientemente flexible que permita combinar todos los tipos de propiedades durante el aprendizaje. Como ya se ha mencionado, MARPLE es una herramienta con una gran importancia dentro del área, pero que carece de la flexibilidad y escalabilidad necesaria, en especial a la hora de incorporar métricas software en el estudio. A fin de poder medir el rendimiento de la propuesta, se ha realizado una validación cruzada estratificada de 5 particiones. Además, dado que el algoritmo G3P4DPD es estocástico, se han realizado 30 ejecuciones con diferentes semillas aleatorias para obtener resultados no sesgados.

4.2. Resultados experimentales

La Tabla 4.5 recoge los resultados obtenidos para los tres PDs que se detectan. Las medidas de evaluación empleadas son las comúnmente utilizadas en el contexto de la clasificación. El *accuracy* mide el porcentaje de instancias correctamente clasificadas. El *recall* indica la proporción de patrones de diseño que se detectan mientras que la precisión (*precision*) nos indica, de entre todas estas instancias, cuantas realmente implementaban un patrón de diseño. La especificidad (*specificity*) mide el porcentaje de instancias negativas que han sido correctamente clasificadas. Por último, F_1 es la media armónica de *recall* y *precision*. Por otra parte, la Tabla 4.6 recoge los resultados obtenidos por MARPLE [7, 44]. Además de las ventajas de flexibilidad y escalabilidad ya mencionadas, la propuesta alcanza valores competitivos en términos de las medidas de evaluación analizadas. Los bajos valores de desviación estándar sugieren que el evolutivo es robusto y sus resultados están poco condicionados por la semilla aleatoria. También es interesante destacar como se alcanzan valores muy equilibrados entre las diferentes medidas. Como cabría esperar, los mejores resultados se obtienen para *Singleton* ya que es el más simple al tener un único rol.

A partir de los resultados de la Tabla 4.5, resulta interesante destacar que, aunque la propuesta detecta la mayoría de PDs, existe un número reducido de patrones que no se llega a recuperar. Al analizar los falsos negativos se pueden extraer dos conclusiones. En primer lugar, llama la atención como, entre diferentes ejecuciones, el conjunto de falsos

	Singleton	Adapter	Factory Method
Accuracy	$0,9389 \pm 0,0442$	$0,8554 \pm 0,0199$	$0,8337 \pm 0,0224$
Precision	$0,9283 \pm 0,0833$	$0,8401 \pm 0,0219$	$0,8157 \pm 0,0273$
Recall	$0,9136 \pm 0,0627$	$0,8797 \pm 0,0252$	$0,8954 \pm 0,0483$
Specificity	$0,9541 \pm 0,0571$	$0,8308 \pm 0,0261$	$0,7617 \pm 0,0483$
F_1	$0,9179 \pm 0,0559$	$0,8592 \pm 0,0194$	$0,8529 \pm 0,0226$

Tabla 4.5: Resultados experimentales

	Accuracy	F_1
Singleton	0,9281	0,9026
Adapter	0,8588	0,8478
Factory Method	0,8189	0,8340

Tabla 4.6: Rendimiento de MARPLE

negativos suele estar compuesto por las mismas instancias. Por otra parte, cabe destacar como, por lo general, estas instancias representan variantes particulares de un patrón para las cuales se dispone de muy pocos ejemplos en el repositorio. Esto se hace muy presente en el caso del *Singleton*, donde la mayoría de instancias recurren a un constructor privado para controlar la instanciación. Sin embargo, existe un grupo muy reducido de instancias que emplean excepciones dentro de un constructor público para controlar la instanciación.

Por otra parte, a partir del análisis de las reglas de asociación utilizadas por el clasificador, cabe destacar como la mayoría suele combinar operadores categóricos y numéricos. Esto sugiere que las mejores reglas (las que forman el clasificador) son aquellas que combinan diferentes tipos de propiedades del código para describir a los patrones. A pesar de ello, también existen reglas que solo contienen operadores categóricos o numéricos, siendo el último caso mucho menos frecuente. Esto no resulta extraño ya que Gamma [2] realiza la descripción de los PDs en base a su estructura y colaboraciones, las cuales son analizadas por los operadores categóricos.

Capítulo 5

Conclusiones y trabajo futuro

Normalmente, la trazabilidad del código no es tenida en cuenta durante el desarrollo del software, afectando a la mantenibilidad, escalabilidad y comprensibilidad del producto final. En este contexto, las buenas practicas aplicadas durante el desarrollo, como el uso de patrones de diseño, se pueden perder entre miles de líneas de código. La detección de patrones de diseño en el código fuente se ha posicionado como una tarea de gran importancia dentro del contexto de la ingeniería inversa del software. En este trabajo se ha presentado un modelo de dos fases para la detección automática de patrones mediante el uso de técnicas de computación evolutiva y aprendizaje automático. En primer lugar, el algoritmo G3P4DPD se encarga de extraer las propiedades que mejor describen a un patrón y sus variantes mediante el estudio de un repositorio software. Este conocimiento se representa como un conjunto de reglas de asociación cuya estructura es conforme a una gramática de contexto libre. Esta gramática dota a la propuesta de una gran flexibilidad ya que permite el estudio simultáneo de métricas software, así como de propiedades de comportamiento y estructurales. En una segunda fase se construye un modelo de detección en base a principios definidos por el algoritmo de clasificación asociativa CMAR.

El modelo ha sido empíricamente validado para tres patrones de diseño (*Singleton*, *Adapter* y *Factory Method*) del repositorio DPB, mostrando resultados muy positivos en términos de rendimiento de clasificación. Tomando como referencia los resultados obtenidos por la herramienta MARPLE, la propuesta ha demostrado ser competitiva y robusta en base a medidas objetivas de evaluación como *accuracy*, *precision*, *recall*, *specificity* y F_1. Estos resultados sugieren que el algoritmo G3P4DPD es adecuado para aprender cuales

son las propiedades que mejor describen a un patrón y sus variantes. Además, el uso de la gramática ha demostrado dar lugar a varios beneficios. Por una parte, ha permitido demostrar como el estudio conjunto de métricas software, así como de propiedades estructurales de comportamiento resulta de gran interés a la hora de describir las implementaciones de los patrones. Por otra parte, hace que la propuesta sea más escalable, ya que el uso de operadores con varios valores reduce el ruido y ayuda a construir clasificadores más interpretables.

Como trabajo futuro se planea, en primer lugar, incorporar nuevos operadores categóricos y numéricos que den soporte a un mayor número de patrones de diseño y que mejoren los resultados de los patrones ya soportados. Por otra parte, el campo de las metaheurísticas aporta muchas técnicas que pueden ser combinadas con el algoritmo G3P4DPD como la hibridación con métodos de búsqueda local para generar mejores reglas de asociación. En este contexto, también podría ser interesante el desarrollo de un algoritmo interactivo en el que el experto guíe el proceso de búsqueda de reglas de asociación o incluso la selección de reglas que formarán el clasificador. De esta manera, el modelo de detección se podría adaptar a los requisitos particulares de una organización. Otro aspecto a tener en cuenta podría ser el uso de diferentes algoritmos de clasificación asociativa para construir diferentes modelos de detección.

Por último, se plantea interesante el integrar este modelo automático con IDEs existentes como Eclipse y añadir funcionalidades que permitan recomendar el uso de un determinado patrón en un punto concreto del código durante el desarrollo de acuerdo al conocimiento extraído. En determinadas ocasiones el desarrollador, sin conocimiento de ello, puede realizar implementaciones parciales de un patrón de diseño en un punto del código en el que su incorporación pueda resultar ventajosa. Ante esta situación, sería interesante que el sistema pudiese dar soporte al usuario indicándole que modificaciones debería de realizar para realizar una correcta implementación del patrón.

Referencias

[1] A. Chihada, S. Jalili, S. M. H. Hasheminejad, and M. H. Zangooei, "Source code and design conformance, design pattern detection from source code by classification approach," *Applied Soft Computing*, vol. 26, pp. 357–367, 2015.

[2] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[3] H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Softw. Maint. Evol.*, vol. 19, no. 2, pp. 77–131, 2007.

[4] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'neill, "Grammar-based genetic programming: a survey," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 365–396, 2010.

[5] W. Li, J. Han, and J. Pei, "Cmar: Accurate and efficient classification based on multiple class-association rules," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pp. 369–376, IEEE, 2001.

[6] F. Thabtah, "A review of associative classification mining," *The Knowledge Engineering Review*, vol. 22, no. 1, pp. 37–65, 2007.

[7] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.

[8] F. A. Fontana, A. Caracciolo, and M. Zanoni, "Dpb: A benchmark for design pattern detection tools," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pp. 235–244, IEEE, 2012.

[9] W. Jung, E. Lee, and C. Wu, "A survey on mining software repositories," *IEICE TRANSACTIONS on Information and Systems*, vol. 95, no. 5, pp. 1384–1406, 2012.

[10] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, "Ar-miner: mining informative reviews for developers from mobile app marketplace," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 767–778, ACM, 2014.

[11] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," in *ACM sigsoft software engineering notes*, vol. 30, pp. 1–5, ACM, 2005.

[12] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th international conference on Software Engineering*, pp. 489–498, IEEE Computer Society, 2007.

[13] D. Čubranić and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proceedings of the 25th international Conference on Software Engineering*, pp. 408–418, IEEE Computer Society, 2003.

[14] I. Thapa and H. Siy, "Assessing the impact of refactoring activities on the jhotdraw project," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 2369–2370, ACM, 2010.

[15] D. M. German, "Using software trails to reconstruct the evolution of software," *Journal of Software: Evolution and Process*, vol. 16, no. 6, pp. 367–384, 2004.

[16] N. H. Madhavji, A. Miranskyy, and K. Kontogiannis, "Big picture of big data software engineering: with example research challenges," in *Big Data Software Engineering (BIGDSE), 2015 IEEE/ACM 1st International Workshop on*, pp. 11–14, IEEE, 2015.

[17] C. E. Otero and A. Peter, "Research directions for engineering big data analytics software," *IEEE Intelligent Systems*, vol. 30, no. 1, pp. 13–19, 2015.

[18] M. Nagappan and M. Mirakhorli, "Big (ger) data in software engineering," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pp. 957–958, IEEE Press, 2015.

[19] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, pp. 208–215, IEEE, 1996.

[20] A. Ampatzoglou, S. Charalampidou, and I. Stamelos, "Research state of the art on gof design patterns: A mapping study," *Journal of Systems and Software*, vol. 86, no. 7, pp. 1945–1964, 2013.

[21] B. B. Mayvan, A. Rasoolzadegan, and Z. G. Yazdi, "The state of the art on design patterns: A systematic mapping of the literature," *Journal of Systems and Software*, vol. 125, pp. 93–118, 2017.

[22] J. Dong, Y. Zhao, and T. Peng, "A review of design pattern mining techniques," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 06, pp. 823–855, 2009.

[23] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE transactions on software engineering*, vol. 32, no. 11, 2006.

[24] J. Dong, Y. Sun, and Y. Zhao, "Design pattern detection by template matching," in *Proceedings of the 2008 ACM symposium on Applied computing*, pp. 765–769, ACM, 2008.

[25] B. B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," *Knowledge-Based Systems*, vol. 120, pp. 211–225, 2017.

[26] R. Wuyts, "Declarative reasoning about the structure of object-oriented systems," in *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pp. 112–124, IEEE, 1998.

[27] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proceedings of the 24th international conference on Software engineering*, pp. 338–348, ACM, 2002.

[28] P. Tonella and G. Antoniol, "Inference of object-oriented design patterns," *Journal of Software: Evolution and Process*, vol. 13, no. 5, pp. 309–330, 2001.

[29] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," in *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pp. 295–304, IEEE, 2005.

[30] Z. Balanyi and R. Ferenc, "Mining design patterns from c++ source code," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pp. 305–314, IEEE, 2003.

[31] Y.-G. Guéhéneuc, J.-Y. Guyomarc'h, and H. Sahraoui, "Improving design-pattern identification: a new approach and an exploratory study," *Software Quality Journal*, vol. 18, no. 1, pp. 145–174, 2010.

[32] Y.-G. Guéhéneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 667–684, 2008.

[33] S. Uchiyama, H. Washizaki, Y. Fukazawa, and A. Kubo, "Design pattern detection using software metrics and machine learning," in *First International Workshop on Model-Driven Software Migration (MDSM 2011)*, p. 38, 2011.

[34] S. Alhusain, S. Coupland, R. John, and M. Kavanagh, "Towards machine learning based design pattern recognition," in *Computational Intelligence (UKCI), 2013 13th UK Workshop on*, pp. 244–251, IEEE, 2013.

[35] J. M. Luna, J. R. Romero, and S. Ventura, "G3parm: a grammar guided genetic programming algorithm for mining association rules," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pp. 1–8, IEEE, 2010.

[36] B. L. W. H. Y. Ma and B. Liu, "Integrating classification and association rule mining," in *Proceedings of the fourth international conference on knowledge discovery and data mining*, 1998.

[37] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[38] J. M. Smith, *Elemental design patterns.* Addison-Wesley, 2012.

[39] J. Y. Gil and I. Maman, "Micro patterns in java code," *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 97–116, 2005.

[40] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *ACM sigmod record*, vol. 29, pp. 1–12, ACM, 2000.

[41] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "Jclec: a java framework for evolutionary computation," *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, vol. 12, no. 4, pp. 381–392, 2008.

[42] Y.-G. Guéhéneuc, "P-mart: Pattern-like micro architecture repository," *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, 2007.

[43] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, "Using metric-based filtering to improve design pattern detection approaches," *Innovations in Systems and Software Engineering*, vol. 11, no. 1, pp. 39–53, 2015.

[44] M. Zanoni, *Data mining techniques for design pattern detection.* PhD thesis, Università degli Studi di Milano-Bicocca, 2012.

Apéndices

Apéndice A

Publicaciones asociadas

Publicaciones asociadas a este trabajo:

- R. Barbudo, A. Ramírez, J.R. Romero, S. Ventura. "Descubrimiento de patrones de diseño basado en buenas prácticas: modelo y discusión". XXII Jornadas en Ingeniería del Software y Bases de Datos (JISBD 2017). La Laguna, Tenerife (España). 19-21/07/2017. SISTEDES - Universidad de La Laguna.
- R. Barbudo, J.R. Romero, S. Ventura. "Use of machine learning for design pattern detection with grammar-guided genetic programming". Evolutionary Computation (CEC), 2018 IEEE Congress on. IEEE, 2018 (Enviado).
- R. Barbudo."Discovery of design patterns based on good practices". Second International Summer School on Search-Based Software Engineering (SS-SBSE). Málaga (España). 29/06/2017 (Charla invitada).

Descubrimiento de patrones de diseño basado en buenas prácticas: modelo y discusión

Rafael Barbudo, Aurora Ramírez, José Raúl Romero y Sebastián Ventura

Dpto. de Informática y Análisis Numérico, Universidad de Córdoba**
Campus de Rabanales, 14071 Córdoba, España
{rbarbudo, aramirez, jrromero, sventura}@uco.es

Resumen La complejidad de los sistemas actuales obliga a los ingenieros software a aprender de las buenas prácticas empleadas en proyectos previos como, por ejemplo, el uso de patrones de diseño. Dichos patrones tienen una gran importancia durante la fase de diseño y su posterior implementación genera varias ventajas. En este contexto, se propone el uso de técnicas de minería de datos que ayuden a comprender como otros ingenieros software han implementado dichos patrones. Con la representación adecuada, este conocimiento se podrá utilizar para identificar fragmentos de código susceptibles de ser convertidos en un determinado patrón. Además del modelo, se discuten ventajas y retos asociados.

Keywords: Minería de repositorios software, buenas prácticas, patrones de diseño, programación genética gramatical.

1. Introducción

El desarrollo de un sistema software es un proceso complejo cuyo éxito está fuertemente condicionado por las habilidades de los participantes del proyecto. Es por ello que aprender de la experiencia de proyectos anteriores resulta de especial interés, pues va a permitir al equipo adoptar buenas prácticas. Una de ellas consiste en la incorporación de patrones de diseño, ya que son soluciones de código efectivas y reutilizables que resuelven un problema de diseño en un contexto determinado. Identificar dónde debe implementarse un patrón o cómo el código existente puede adaptarse para acercarse a él son tareas que requieren conocer cuáles son las propiedades que mejor caracterizan al patrón y a sus posibles variantes. Por ello, existen propuestas que, a partir de propiedades estructurales, morfológicas y semánticas definidas por el experto, permiten identificar antipatrones en el código como paso previo a la recomendación [1].

Una alternativa interesante consiste en tratar de inferir esta información de manera automática. Sin embargo, extraer cualquier tipo conocimiento en el ámbito del desarrollo software constituye un reto en sí mismo, pues va a estar oculto entre miles de líneas de código y en otros artefactos software. En este contexto

** Trabajo financiado por el Ministerio de Economía y Competitivad, proyecto TIN2014-55252-P y el Ministerio de Educación, programa FPU (FPU13/01466).

surge la minería de repositorios software [2] (MRS), cuyo objetivo es descubrir información interesante y previamente desconocida a partir de los recursos y plataformas para la gestión y configuración del software, incluido el código. Para ello se sirve de técnicas de minería de datos y aprendizaje automático, como la minería de patrones frecuentes (PF) [3]. En el contexto de los patrones de diseño, las propuestas de MRS se han centrado principalmente en detectar patrones ya existentes bien mediante medidas software [4] o a partir de su morfología [5].

Sin embargo, la idoneidad de un patrón de diseño puede también venir dada por aspectos que, aún estando presentes en el código (relaciones en el diseño, nomenclatura, etc.), podrían no ser directamente capturados por métricas software. Además, al tratase de soluciones flexibles, la morfología de un patrón podría variar sensiblemente si el programador requiriese adaptarlo a ciertas particularidades del proyecto. Con esta idea, este trabajo propone un modelo de MRS que, adquiriendo una base de conocimiento extraída de repositorios software reales sobre el buen uso previo de patrones de diseño, sea capaz de descubrir aquellas partes del código–actualmente en desarrollo–en las que resultara apropiado adoptar un patrón concreto. El modelo se compone de tres fases: (1) búsqueda de PF aplicando un algoritmo basado en programación genética gramatical (*Grammar Guided Genetic Programming*, G3P), para extraer y caracterizar el conocimiento acerca de cómo un tipo de patrón de diseño está siendo llevado históricamente a la práctica, (2) identificación de estructuras de código potencialmente adecuadas para un patrón de diseño; y (3) análisis de la información extraída para el descubrimiento de oportunidades.

2. Marco conceptual

La MRS permite aprender de las prácticas realizadas en proyectos software anteriores. Para ello, la datos son extraídos utilizando diversas fuentes, como repositorios software, documentos y artefactos software, foros de programadores o *bug trackers*, entre otros. Estas fuentes se analizan para descubrir aspectos escondidos en esos datos pero relevantes al proceso de la ingeniería del software. Técnicas como clasificación, asociación, regresión o *clustering* son habituales.

Más específicamente, la minería de PF tiene carácter descriptivo, pues busca descubrir conjuntos de ítems (*itemsets*) que ocurren con una determinada frecuencia (llamada soporte) en el conjunto de datos. La estructura de un *itemset* viene definida conforme a una gramática de contexto libre (GCL), cuyos símbolos representan el conjunto de atributos y operadores que caracterizan el código. Estos operadores se pueden adaptar para trabajar con diferentes lenguajes. En general, cada ítem se representa mediante un predicado del tipo <*atributo*><*operador*><*valor*>, que puede tener naturaleza numérica (LOC>10K) o categórica (X! =*class*). Estos son unidos mediante operadores lógicos para formar los *itemsets*. Bajo el paradigma G3P, este tipo de gramática ya se ha utilizado para extraer reglas en minería de asociación [6] o para la predicción de costes en proyectos software [7], demostrando la flexibilidad que ofrece una GCL para modelar y representar el conocimiento en diferentes dominios.

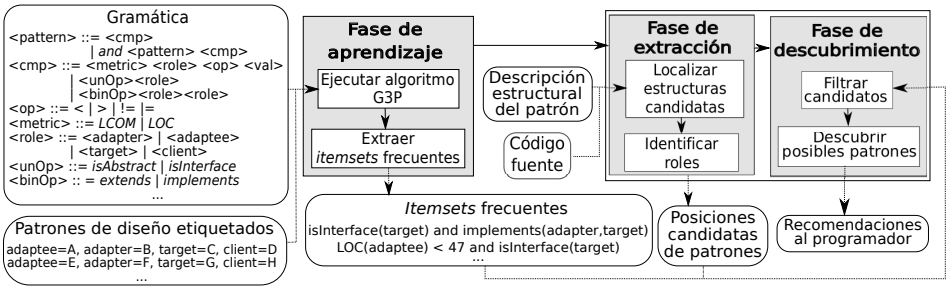


Figura 1. Visión general del modelo

3. Modelo de descubrimiento de patrones

La Figura 1 muestra las fases del modelo propuesto, incluyendo sus entradas y salidas. Inicialmente se aplica la *fase de aprendizaje* sobre un repositorio de código que contiene múltiples desarrollos de un patrón de diseño dado, y en el que son conocidos los roles que implementa cada clase o interfaz[1]. Para ello se ejecuta un algoritmo G3P [6], adaptado para la búsqueda de *itemsets* conformes a la GCL dada. Por ejemplo, considerando el patrón *Adapter*, la gramática representa el lenguaje que declarará los operadores necesarios para definir su estructura (p.ej. una clase abstracta o la implementación de una interfaz). Pueden también incluirse métricas software de interés. Como resultado, se obtiene el conjunto de características asociadas al patrón en el uso histórico. En el ejemplo mostrado sobre el patrón *Adapter* se observa, entre otros muchos aspectos descubiertos por el conjunto de *itemsets*, como el rol *adapter* se desarrolla en una clase, mientras que el rol *target* lo implementa una interfaz. Además, el rol *adaptee* suele ser implementado por una clase con pocas líneas de código (LOC<47).

Ya sobre el código en desarrollo se lleva a cabo la *fase de extracción*, donde se localizan fragmentos con una estructura aproximada a la del patrón con el objetivo de localizar posiciones candidatas en las que éste podría ser aplicado. Considérese que la definición estructural de un patrón es previamente conocida. Por tanto, se tratará de asociar cada elemento del conjunto candidato con un posible rol, excluyendo a aquellos que incumplan las restricciones del patrón o se desvíen en exceso. En el caso del patrón *Adapter*, se buscarían estructuras formadas por cuatro elementos, tanto clases como interfaces, relacionados entre sí de manera que puedan desempeñar los roles definidos por su especificación (*adaptee*, *adapter*, *target* y *client*). Para la realización de esta fase se aplican técnicas de análisis de grafos [4,5].

Finalmente, sobre posiciones estructuralmente candidatas, la *fase de descubrimiento* aplica el resto de aspectos extraídos en fases anteriores para recomendar en qué partes del código puede incorporarse un patrón de diseño. Así pues, aquellas estructuras candidatas del código fuente que no presenten las características definidas por los *itemset* frecuentes son descartadas. Además, se

[1] Se ha utilizado P-MARt: `http://www.ptidej.net/tools/designpatterns`

analizará la similitud con las estructuras que cumplan dichas características en mayor o menor medida, constituyendo los fragmentos de código susceptibles de ser transformados para implementar convenientemente el patrón de diseño. En definitiva, el conocimiento extraído del uso de buenas prácticas en proyectos anteriores se utiliza de base para recomendar de forma comprensible la adaptación del nuevo código a un determinado patrón de diseño.

4. Discusión y temas abiertos

El modelo propuesto combina diferentes tipos de técnicas para conseguir un proceso de soporte al programador más completo que las propuestas existentes. Nótese que los métodos actuales tienden a centrarse en un número limitado de patrones. Además, se focalizan bien en estructuras estrictas para el patrón o bien en aquellas ajustadas a unos valores de métricas software. En este sentido, el uso de una GCL dota de mayor flexibilidad al proceso de aprendizaje al permitir incorporar simultáneamente atributos y operadores de distinta naturaleza.

Uno de los puntos más complejos es la correcta identificación de los fragmentos de código candidatos a ser convertidos en un patrón. Por un lado, es necesario abstraer el nuevo código fuente de su nomenclatura y, en cierta medida, también de su estructura para poder compararla con la del patrón. Esto es aún más complejo al tener presente que los tipos de patrones son muy diferentes entre sí. Por ello, para abstraerlo es importante establecer criterios de similitud que acepten desviaciones respecto del patrón original. Por otro lado, la fase de descubrimiento requiere el estudio de las medidas de interés (y sus umbrales) más apropiadas a la hora de filtrar las estructuras candidatas en base a los *itemset*.

Referencias

1. N. Nahar and K. Sakib, "Automatic recommendation of software design patterns using anti-patterns in the design phase: A case study on abstract factory.," in *3rd Int. Workshop on Quantitative Approaches to Software Quality*, pp. 9–16, 2015.
2. H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *J. Softw. Maint. Evol.*, vol. 19, no. 2, pp. 77–131, 2007.
3. J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data Min. Knowl. Discov.*, vol. 15, no. 1, pp. 55–86, 2007.
4. A. Chihada, S. Jalili, S. M. H. Hasheminejad, and M. H. Zangooei, "Source code and design conformance, design pattern detection from source code by classification approach," *Appl. Soft Comput.*, vol. 26, pp. 357–367, 2015.
5. B. B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," *Knowl-Based Syst.*, vol. 120, pp. 211–225, 2017.
6. J. M. Luna, J. R. Romero, and S. Ventura, "Design and behavior study of a grammar-guided genetic programming algorithm for mining association rules," *Knowl. Inf. Syst.*, vol. 32, no. 1, pp. 53–76, 2012.
7. Y. Shan, R. I. McKay, C. J. Lokan, and D. L. Essam, "Software project effort estimation using genetic programming," in *IEEE Int. Conf. on Communications, Circuits and Systems and West Sino Expositions*, vol. 2, pp. 1108–1112, 2002.

Use of machine learning for design pattern detection with grammar-guided genetic programming

Rafael Barbudo
Dept. of Computer Sciences
University of Córdoba
Córdoba, Spain
rbarbudo@uco.es

José Raúl Romero
Dept. of Computer Sciences
University of Córdoba
Córdoba, Spain
jrromero@uco.es

Sebastián Ventura
Dept. of Computer Sciences
University of Córdoba
Córdoba, Spain
sventura@uco.es

***Abstract*—Design patterns are general, reusable solutions to address a commonly occurring design problem in software development. However, the lack of an appropriate code documentation often hampers traceability, and their use is finally missed among thousands of lines of code. Identifying design patterns in the source code brings significant benefits to the product maintenance and scalability. Hence, design pattern detection has attracted attention in the field of reverse engineering. Current approaches are usually focused on either software metrics or specific behavioural and structural features of the source code.**

In this paper, a novel two-phased model for the automatic detection of design patterns from software repositories is proposed by combining evolutionary computation and machine learning techniques. Firstly, a genetic programming-based algorithm extracts those software properties best describing the pattern to be detected. This knowledge is formulated as association rules, whose syntax is conformant with a context-free grammar. Secondly, the detection model is constructed to identify those parts of the code potentially having implemented a pattern or variant. This model permits the code analysis by jointly considering both software metrics, and behavioural and structural properties. The approach is empirically validated for three different design patterns, the obtained measures also showing its competitiveness in terms of detection performance.

***Index Terms*—design pattern detection, machine learning, grammar guided genetic programming, associative classification, reverse engineering**

I. Introduction

Design patterns (DPs) [1] are general, reusable solutions that address a recurrent design problem within a given context. DPs are not definitive solutions to be directly transformed into code, but they provide a template on how to solve such a problem. As a best practice for programmers, their use improves the quality of a software product in terms of its maintainability, extensibility and understandability. Therefore, they are extensively applied to any application domain. However, code traceability is sometimes not taken into consideration by programmers submitting new versions of their source code. This implies that their use could not be properly documented and these incorporated DPs would be lost among thousands of lines of code with time. From the perspective of software maintenance, the identification of these hidden DPs from a source code project could report a number of benefits. For instance, it may facilitate software comprehension and reusability, apart from a well-structured use of proven code solutions. In addition, it may lead to the understanding of certain design decisions taken during the development phase. As a result, the so-called design pattern detection (DPD) has been positioned as an important task within the field of reverse engineering. Given that a manual inspection might not be a realistic option considering that it is an error-prone, time consuming process, the definition of automatic methods for DPD has attracted the interest of software engineers.

Most discovery approaches used in current automatic DPD methods have something in common: to some extent, they all need the introduction of previous knowledge about the structure and properties of the specific DP being detected. Such a knowledge represents the particular view of a group of experts but could not be enough to detect other implementations or code versions made by other developers, teams or organisations. The set of all the different possible implementations of a DP is known as *variants* [2]. In an effort to overcome this issue, some methods have introduced certain flexibility in order to avoid the need of an exact match. To this end, some authors have applied machine learning (ML) techniques, which are generally more flexible and could be adopted to learn a large number of variants of a given DP. A first method based on ML techniques [3] used decision trees and a neural network to reduce the number of false positives obtained by techniques based on the structural analysis of the system. Object-oriented software metrics are normally preferred as input for these ML algorithms. In [4], a feature vector comprised of $n*k$ elements is used to represent a DP implementation, where n is the number of software metrics and k is the number of roles played by the different elements defined by the DP. On the other hand, MARPLE [5], an Eclipse-based tool for DPD, makes use of structural and behavioural properties of the design patterns as inputs of the ML algorithm, such as object compositions or method delegations. To the best of our knowledge, there is not an approach flexible enough to combine both software metrics and structural and behavioural properties to fully characterise the DP being detected.

This paper proposes a novel two-phased model to accurately detect a design pattern and its variants. A first phase is responsible for learning the characteristics that best describe

This work was supported by the Spanish Ministry of Economy and Competitiveness [project TIN2017-83445-P]; and FEDER funds.

the variants of the DP to be detected. These characteristics may refer to (1) software metrics like the coupling between objects; (2) behavioural properties like method invocations or message passing; and (3) structural properties like inheritance between classes or compositions. With the aim of conducting this learning process, a grammar-guided genetic programming (G3P) [6] algorithm has been developed to extract knowledge about the DP. All this information is encoded as a set of association rules (ARs), whose structure is formed according to a context-free grammar (CFG). Since the extracted ARs could not be directly used to detect a DP and its variants, a second phase is responsible for building the detection model from this set of collected ARs. Here, parts of an associative classification (AC) [7] algorithm, called CMAR (Classification based on Multiple Association Rules) [8], has been adopted to construct the detection model.

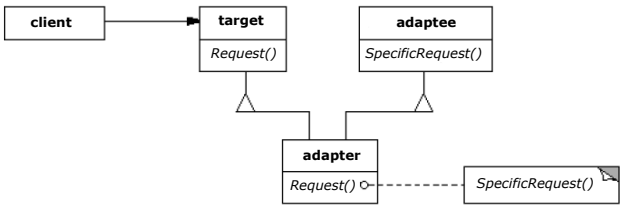
With the aim of showing the flexibility of the proposed model, its applicability for the detection of 3 different design patterns (singleton, adapter and factory method) is shown. In addition, it is empirically validated against the MARPLE tool using 9 real software systems, the DP instances used for the validation having been extracted from the design pattern detection benchmark platform (DPB) [9]. Empirical results show that, apart from the flexibility to simultaneously deal with diverse sorts of properties and patterns, the proposed approach reaches a competitive detection performance compared to the rest of current approaches.

The rest of the paper is organised as follows. The design pattern detection problem is presented in detail in Section II. Section III provides a general view of the two-phased model proposed for DPD. Next, Section IV describes the first step of our approach, consisting in the G3P algorithm for AR generation. The second phase, which partly includes the associative classification algorithm for the generation of the DPD model, is explained in Section V. Then, the approach is empirically validated and compared against other well-known approach from the literature in Section VI. Finally, some concluding remarks are presented in Section VII.

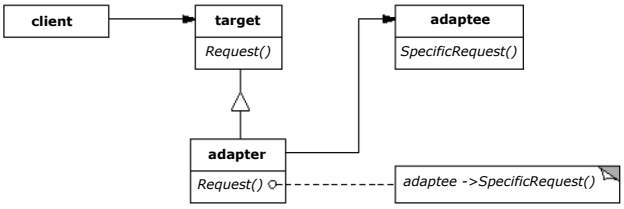
II. Background

Design patterns are classified according to their purpose, i.e. the type of design problem to which they can provide a general solution [1]: *behavioural patterns* are specifically concerned with interactions between classes and objects, *creational patterns* provide a way to create objects while hiding the complexity of the process under the problem circumstances, and *structural patterns* facilitate the composition of classes and objects. An *adapter* is an illustrative example of the latter case, aimed at simplifying the use of an existing class (usually coming from a legacy system) when its interface does not match the one required by the system.

Every design pattern describes a fixed number of roles. A role represents a specific task that should be performed by an element of the pattern. In an object-oriented software system, roles are performed by either classes or their instances (objects). For instance, following the adapter example, this



(a) Alternative based on multiple inheritance



(b) Alternative based on service delegation

Fig. 1: Two possible implementations of an adapter

is composed by four different roles: *adaptee*, *target*, *client* and *adapter*. The *adaptee* is played by the interface that needs adapting, while the *target* refers to the domain-specific interface that will be invoked by the *client* system. Finally, the *adapter* defines the proper adaptation of the services provided by the *adaptee* to the format required by the *target*. Notice that roles are often implemented by more than one class or object, what makes detection even more complicated.

As mentioned above, a DP can be seen as a template describing how to solve in a general way a design problem by using a number of code elements. Consequently, there could be different implementations of the same pattern depending on the programmer, specific requirements, design problem, etc. In this vein, Figure 1 shows two possible solutions [1] for implementing an adapter. A first alternative, shown in Figure 1a, is based on the use of multiple inheritance as a way to adapt one interface into the other. Nevertheless, it is worth noting that some languages like Java do not allow the use of this mechanism in a direct way. In contrast, the second option shown in Figure 1b is based on the *adaptee* delegating invocations to services by the *adapter*, whose signatures are defined according to the *target* interface.

A number of authors have already proposed methods for (semi-)automatic DPD. These methods can be categorized based on the discovery approach being applied. A large number of proposals are based on similarity scoring techniques [10], which usually represent the structural information of the pattern and the system being analysed in form of graphs and matrices. As a general rule, these methods search for substructures that match a predefined structure for the DP graph within the system graph, and usually analyse behavioural properties of the DPs, e.g. message passing, in order to reduce false positives. Other DPD approaches propose reasoning techniques, which could be divided into two main groups:

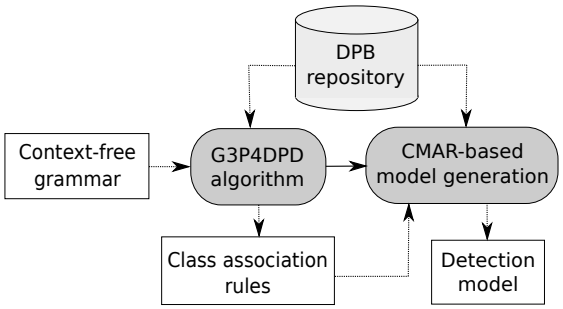


Fig. 2: Two-phased model for design pattern detection

logical reasoning [11] and fuzzy reasoning [2]. In contrast to the precision provided by logical reasoning approaches, fuzzy reasoning-based proposals look for differences between pattern implementations, but suffer from a high number of false positives. Formal approaches are computationally demanding models. Thus, their application [12] is limited to those situations in which the use of logical and mathematical methods brings benefits to DPD detection because of the precise identification of meaningful groups of elements having similar properties.

Nowadays, software projects are stored in extensive software repositories containing a large number of source code files, usually committed by a team of programmers, and evolving along a submission history. Given these circumstances, it is hard to ensure that using static descriptions for the patterns could lead to an exact or approximate match. In contrast, ML techniques provide mechanisms to learn those properties that best describe a given DP and its variants taking a code repository as a data source. As previously mentioned, some initial proposals already make use of software metrics or structural and behavioural properties to automatically detect hidden patterns from the code.

III. Two-phased model for DPD

With the goal of taking advantage of the benefits of ML, we propose a two-phased model for DPD that expands the nature of the information extracted and used to detect patterns from the source code, jointly considering different properties in terms of both structural and behavioural aspects and software metrics (see Figure 2). A first step is responsible for learning the set of software properties that best describe the design pattern under analysis as well as its variants. With this purpose, the recurrent use of the DP throughout a code repository is scrutinized. As a starting point, the DPB repository has been chosen, though other institutional data sources would be also valid instead. It is worth noting that, in contrast to other alternatives like P-mart [13], DPB does not only contain correct pattern implementations, i.e. positive samples, but it also contains a number of classes that could not be properly considered as a pattern, i.e. negative samples, even when some similarities might lead to misclassification.

A CFG is defined to precisely declare the language syntax that allows expressing the software properties and constraints describing these samples in form of association rules [14]. Let I = $\{i_1, ..., i_n\}$ be a set of items, and let A and C be itemsets, i.e., A = $\{i_1, ..., i_j\}$ and C = $\{i_1, ..., i_k\}$, an association rule is an implication of the type $A \rightarrow C$ where A $\subset$ I, C $\subset$ I, and A $\cap$ C = $\emptyset$. In this context, association rule mining [15] is a popular unsupervised learning method including approaches having a descriptive nature. The use of a CFG makes the learning process more flexible, as it allows to combine software properties of different nature, e.g. numerical software metrics or structural properties expressed in form of categorical values, as items of the rule.

A G3P-based algorithm, called G3P4DPD, is responsible for learning the properties that best describe a certain DP and its variants. It generates a set of ARs in compliance with the CFG, and evolves rules by conveniently applying custom genetic operators. The intention is to generate the set of association rules that best describe the design pattern to be detected by extracting defining features from the repository samples. In this case, the collected rules are known as *class association rules* (CARs) [16]. A CAR is a special sort of AR whose consequent is restricted to a classification class label, that is, whether the combined items of the antecedent are implying the implementation, or not, of a DP. After a number of iterations, the G3P4DPD algorithm returns a set comprised by the best CARs extracted along the whole evolutionary process, as described in Section IV.

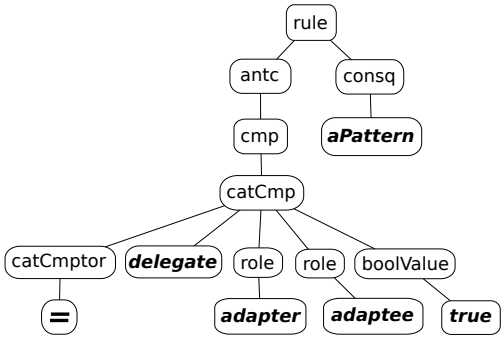
CARs mined by the G3P4DPD algorithm serve as input for building the prediction model for DPD. With this aim, some principles of CMAR have been adopted. CMAR is a well-known algorithm for associative classification, a branch of data mining that integrates association rule mining and classification with the aim of building a predictive model or *classifier* [7]. CMAR is an accurate algorithm [8], modular and easy to integrate with G3P4DPD, that generates a set of CARs by pruning those not being suitable for their use in classification, as detailed in Section V. Finally, this model is responsible for predicting whether a set of classes and objects match a design pattern or some of its variants.

IV. G3P-based algorithm for AR generation

This section describes the G3P4DPD algorithm, responsible for mining the features describing the pattern to be detected. Firstly, the CFG and the encoding of individuals are explained. Then, the evolutionary procedure, including the applied fitness function and genetic operators, is detailed. As a result, the set containing the most accurate CARs is returned to serve as input for the construction of the subsequent detection model.

A. Encoding

Individuals in G3P4DPD are declared at two levels of abstraction: a representation of the solution, i.e. the genotype, in terms of a derivation syntax tree; and the solution itself, i.e. the phenotype, denoting a *class association rule*. Figure 3a shows a sample representation of a genotype, while Figure 3b shows its corresponding phenotype.



(a) Genotype example

(= delegate adapter adaptee true) → (aPattern)

(b) Phenotype example

Fig. 3: Individual encoding example

```
<rule>      ::= <antc> <consq>
<antc>      ::= <cmp> | and <antc> <cmp>
<cmp>       ::= <numCmp> | <catCmp>
<numCmp>    ::= <numCmptor> <numOp> <role> const
<catCmp>    ::= <catCmptor> isFinal <role> <boolValue>
              | <catCmptor> delegate <role> <role> <boolValue>
              | <catCmptor> typeOf <role> <typeOfValue>
              | <catCmptor> linkArtefact <role> <role>
                <linkArtefactValue>
              | ...
<numCmptor>           ::= ≥ | ≤ | > | <
<catCmptor>           ::= = | !=
<numOp>               ::= DIT | NOC | CBO | NOM
<role>                ::= adapter | adaptee | target
<boolValue>           ::= true | false
<typeOfValue>         ::= class | absClass | intface | enum
...
<linkArtefactValue>   ::= directInherit | indirInherit
                        | directImpl | indirImpl | notLinked
<consq>               ::= aPattern | notAPattern
```

Fig. 4: Production rules of the CFG applied by G3P4DPD

As previously explained, each individual is constructed in accordance to a CFG, whose non-terminal elements are derived into terminal symbols that declare the constituent components of the language. Additionally, production rules determine and constrain how ARs are built. This language has been defined to jointly consider both software metrics and structural and behavioural features of a pattern. In addition, this declaration can be customised for a specific DP, if required, resulting in a more accurate expression of the properties of the pattern. Figure 4 shows the production rules of the CFG defining the structure of a valid individual, where strings expressed between <..> stand for non-terminal symbols, and terms in italics stand for terminal symbols. The root symbol, i.e. from which the derivation procedure starts, is <*rule*>. It can be derived into two symbols, <*antc*> and <*consq*>, which represent the antecedent and consequent of the AR, respectively. On the one hand, the antecedent can be formulated as an inclusive sequence of numerical (<*numCmp*>) and categorical (<*catCmp*>) comparisons. On the other hand, <*consq*> can be derived into two possible terminals, expressing whether it is a valid pattern (*aPattern*) or not (*notAPattern*). Notice that the grammar in Figure 4 has been slightly customised for the adapter pattern, the non-terminal <*role*> being derived into *adapter*, *adaptee* and *target*. The role *client* is not required for detection purposes.

As mentioned above, a property of a pattern is expressed in the antecedent of the rule in terms of a single- or multi-comparison predicate. Regardless of their specific type, all comparisons are written as prefix expressions containing a comparator, an operator, one or more arguments, and a value. More specifically, numerical comparisons are used to represent those properties based on software metrics. They are expressed in terms of a numerical comparator like $<$, $>$, $\leq$ or $\geq$; a numerical operator computing an object-oriented software metric [17] like NOM, NOC, DIT or RFC (see Table I for a complete description); an argument receiving a role; and a numerical constant (*const*). As an example, the comparison "`NOC(target)<1`" indicates that *target*, usually implemented by a class, has no subclasses.

On the other hand, categorical comparisons are used to express structural and behavioural properties of a pattern. They are composed by a categorical comparator, namely = or *!=*; a categorical operator like *linkArtefact* or *delegate*; a number of arguments receiving the participating roles; and a value. As an example of a categorical comparison, "`linkArtefact(adapter,target)=directInherit`" indicates that *adapter* is a direct subclass of *target*. Due to the limited space, the CFG shown in Figure 4 omits some categorical operators and terminal values (please see Table I for a complete list of categorical operators).

It is worth noting that most categorical operators are based on elemental design patterns [18], micro patterns [19] and design patterns clues [20], which can be matched by operators checking whether a code entity satisfies a property in terms of *true* and *false* values. As an example, *Abstract Interface* is an elemental design pattern checking if a code artefact is abstract or not. For some languages like Java, an artefact not only might be an abstract or concrete class, but also an interface. This issue could be addressed by simply adding new different operators like *isAbstract*, *isConcrete* and *isInterface*. However, the resulting predicates would be strongly correlated and might add noise to the learning process. The CFG enables the declaration of categorical multi-valued operators, which are able to return different alternative terms instead (e.g. see the *typeOf* operator in Table I).

B. Algorithm description

Algorithm 1 shows the general procedure of G3P4DPD. It requires 5 inputs: the number of generations (*maxGen*), the population size (*popSize*), the number of individuals to be returned (*extPopSize*), a CFG (*grammar*) and a design pattern repository (*repo*). This repository is composed of a set of DP samples and their source code. A DP sample defines the

TABLE I: Grammar operators

Numerical operators	
Signature	**Description**
NOM(R_1)	Number of methods of R_1
NOC(R_1)	Number of children directly inherited from R_1
DIT(R_1)	Depth of the inheritance tree from R_1
RFC(R_1)	Number of distinct methods and constructors potentially invoked when an object of R_1 receives a message
Categorical operators	
Signature	**Description**
isFinal(R_1)	*true* if R_1 cannot be extended; *false* otherwise
isSubclass(R_1)	*true* if R_1 is a subclass
controlledInit(R_1)	*true* if R_1 instantiates itself within an *if* or *while* block
staticField(R_1)	*true* if R_1 declares a static field of any type
staticFlag(R_1)	*true* if R_1 declares a static boolean field
conglomeration(R_1)	*true* if 2 or more methods of R_1 are invoked from another method from R_1
returned(R_2,R_1)	*true* if a value of type R_2 is returned from a method of R_1
received(R_2,R_1)	*true* if a method of R_2 receives a value of type R_1 as argument
createObj(R_1,R_2)	*true* if R_1 instantiates R_2
delegate(R_1,R_2)	*true* if a method of R_1 invokes a method of R_2
sameElem(R_1,R_2)	*true* if R_1 and R_2 are coded by the same artefact
typeOf(R_1)	Returns the type of the artefact implementing R_1 (*absClass*, *intface*, *enum*, *class*)
linkMethod(R_1,R_2)	Indicates if a method of R_1 directly or indirectly overrides (*directOver*, *indirOver*), implements (*directImpl*, *indirImpl*) or is *notLinked* to a method of R_2
linkArtefact(R_1,R_2)	Returns the sort of link between the artefacts playing R_1 and R_2 (*directInherit*, *indirInherit*, *directImpl*, *indirImpl*, *notLinked*)
ctorVisibility(R_1)	Returns the visibility of the less restrictive constructor of R_1 (*private*, *protected*, *package*, *public*, *default*)
aggregation(R_1,R_2)	Returns information about an attribute of R_2 declared in R_1 in terms of its visibility and instanciability
adapter(R_1,R_2,R_3)	Returns if a declared (*decl*) or inherited (*inhr*) method of R_1, implemented from R_3, delegates in a method of R_2; *notLinked* otherwise

elements comprising the pattern and their roles, i.e. the role mapping. Notice that G3P4DPD creates an external archive (*extPop*) composed of the most accurately evaluated individuals obtained along the evolutionary process. The external archive will serve as the output of the algorithm.

As for the procedure, it firstly creates a random population of *popSize* individuals (*pop*) in conformance with the syntax prescribed by *grammar*. The external archive is then initialised to the empty set. Initial individuals are evaluated by computing the fitness function, which calculates the support of the represented rule against the repository (*repo*). The *support* is a well-known measure in association rule mining that indicates how frequently a rule appears in the dataset, as shown in Equation 1, where t stands for the transactions in the dataset T containing the itemset X.

$$supp(X) = \frac{|\{t \in T; X \subseteq t\}|}{|T|} \quad (1)$$

At this point, the algorithm iterates until *maxGen* is reached. Individuals are evolved along these iterations by applying genetic operators. Firstly, the selection operator chooses *popSize* individuals from the union between *pop* and *extPop*. Secondly, the crossover operator is applied, with a certain probability, over pairs of individuals by combining their derivation syntax trees. And thirdly, one of the two mutators is selected at random: a first alternative mutator (*diversityMutator*) is conceived to promote individual diversity, whilst a second choice (*dpdMutator*) is a domain-specific mutator for DPD. Further details on operators are provided next in Section IV-C. Finally, every individual within *pop* is evaluated.

Algorithm 1: G3P4DPD

```
Input  : maxGen, popSize, extPopSize, grammar, repo
Output: extPop
pop ← generateRules(popSize, grammar)
extPop ← ∅
evaluate(pop, repo)
while generation < maxGen do
    pop ← select(pop ∪ extPop, popSize)
    pop ← crossover(pop)
    if randon < 0.5 then
        pop ← diversityMutator(pop);
    else
        pop ← dpdMutator(pop);
    end
    evaluate(pop, repo)
    extPop ← update(pop ∪ extPop, extPopSize)
    generation++
end
```

Having the set of evaluated individuals, the external archive is then updated by merging *pop* and former *extPop*, and pruning the resulting set. To do this, individuals are ranked according to their confidence value (see Equation 2), which measures how often the consequent *C* is true when the antecedent *A* is also true. Rules are pruned following the precedence among rules as prescribed by Liu *et al.* [16]. Therefore, given two rules R_1 and R_2, it is said that R_1 *precedes* R_2 iff any of these conditions is satisfied: (a) conf(R_1) > conf(R_2); (b) conf(R_1) = conf(R_2) and supp(R_1) > supp(R_2); or (c) conf(R_1) = conf(R_2), supp(R_1) = supp(R_2) and the number of comparisons in R_1 is smaller than in R_2. On this basis, if a rule $A \rightarrow C$ precedes a rule $A' \rightarrow C'$ and $A \subset A'$, then $A' \rightarrow C'$ is pruned [8].

$$conf(A \rightarrow C) = \frac{supp(A \cup C)}{supp(A)} \quad (2)$$

Individuals not exceeding a support and confidence threshold are also pruned. In addition, a χ^2 analysis is performed to prune those individuals whose antecedent and consequent are not positively correlated [21]. This happens when the χ^2 statistic value is not greater than a given threshold (see Equation 3). This analysis is performed to only keep those rules having strong implications for the detection.

$$\chi^2 = \frac{n * (lift(R) - 1)^2 * supp(R) * conf(R)}{(conf(R) - supp(R)) * (lift(R) - conf(R))} \quad (3)$$

$$where \quad lift(A \rightarrow C) = \frac{supp(A \cup C)}{supp(A) * supp(C)}$$

C. Genetic operators

As pointed out in Section IV-B, the first operator to be executed is the selector, responsible for selecting individuals from the population for later breeding. More specifically, a binary tournament is applied here, which randomly selects two individuals and keeps the best according to their fitness. This process is repeated until *popSize* individuals are selected.

Then, the crossover operator works by randomly selecting and exchanging a single comparison from the genotype of a couple of individuals. Finding such a comparison requires traversing the derivation syntax tree in pre-order until a <*cmp*> symbol is reached, delimited by *and*, <*cmp*> or <*consq*>, as prescribed by the grammar. Based on their fitness, the two best individuals among parents and offspring are returned to the population.

Finally, G3P4DPD considers two different operators for mutation. The so-called *diversityMutator* selects a number of comparisons and rebuild them by randomly deriving the corresponding <*cmp*> symbols until reaching new terminals. The number of comparisons to rebuild is chosen using a random roulette wheel, which promotes small modifications against general changes of the individual. A second mutation operator, named *dpdMutator*, traverses the derivation tree and negates each comparison with a probability, inverse to the number of comparisons in the antecedent. This is made by replacing the comparator with its opposite (e.g. $<$ would be replaced by $\geq$). In addition, the terminal describing the class label in the consequent is also inverted with a 50 percent chance. The rationale behind this operator is to obtain rules describing both positive and negative samples. For instance, if the comparison "`ctorVisibility(singleton)=private`" is describing a recurrent property of the positive samples in a singleton pattern, "`ctorVisibility(singleton)!=private`" is likely to describe negative samples. Regardless of the operator chosen, the best individual (parent or offspring) is returned.

V. Detection model with associative classification

CMAR has been partly adopted as a model for generating the detection model. The overall approach consists in a two-step associative classification algorithm for building a rule-based classifier. In a first step, a custom FP-Growth algorithm [22] generates all CARs exceeding a support and confidence threshold. This single constraint does not guarantees that rules are suitable for their use in detection, so CMAR defines 3 pruning strategies. Firstly, for each generated rule, CMAR checks if its antecedent and consequent are positively correlated. If so, the rule is stored, and a pruning method is computed using precedence rules. After generating all the rules, those rules describing a reduced amount of samples in terms of the dataset coverage are discarded. The resulting rules compose the classifier, which is responsible for identifying whether a set of classes and objects implements a DP. Secondly, when the classifier receives an unknown sample, it looks for those rules whose antecedent matches the sample. If they all have the same consequent, CMAR simply assigns their label to the sample. Otherwise, these rules are separated into groups according to their respective class labels. Thirdly, CMAR uses a weighted χ^2 measure to find the most significant group and assign its label to the sample.

However, rigid methods like FP-Growth would hamper the applicability and flexibility of the proposal here presented, what motivates that only the second stage of CMAR is adopted. Regarding their use for DPD, these methods require the inputs from the dataset to comply with specific data formats, usually in form of categorical attributes. Besides, it should be noted that the number of attributes could easily increase when a more complex DP is to be detected, e.g. due to a larger number of roles. For each operator, a number of attributes are generated by switching arguments. Hence, a more flexible, scalable method like the G3P4DPD serves instead to mitigate these shortcomings for rule generation. Observe that the two first pruning strategies of CMAR were integrated within the evolutionary process, that is, during the external archive update. The third pruning strategy is performed at the end of the evolutionary algorithm.

VI. Empirical validation

The JCLEC [23] (Java Class Library for Evolutionary Computation) framework has been used to code G3P4DPD, apart from other 3 open source libraries for analysing software properties. On the one hand, numerical operators make use of the ckjm[1] (Chidamber and Kemerer Java Metrics) library for computing software metrics. On the other hand, categorical operators are implemented using Java Parser[2] and Javassist[3] for the extraction of behavioural and structural properties from the source code and Java bytecode, respectively.

Instances have been extracted from the DPB repository, created by the authors of MARPLE, which collects samples from 9 industrial Java projects, and one more adopted from the literature. Table II lists the number properties (types, methods, attributes, lines of code) of each software project used for the experimentation. In addition, Table III shows the number of samples mined to classify the 3 design patterns under study (singleton, adapter and factory method).

Likewise, Table IV shows the configuration applied to G3P4DPD (phase 1), and the parametrisation of the adapted CMAR (phase 2), where the values recommended by the authors are used [8]. The size of the external archive has been set to a high value (500) to avoid discarding interesting rules. Notice that the resulting rules have to be pruned according to

[1]ckjm: *https://www.spinellis.gr/sw/ckjm*
[2]Java Parser: *https://javaparser.org*
[3]Javassist: *http://jboss-javassist.github.io/javassist/*

TABLE II: Properties of the Java projects for DPD

Project	Types	Methods	Attributes	LOC
DesignPatternExample	1749	4710	1786	32313
QuickUML 2001	230	1082	421	9233
Lexi v0.1.1 alpha	100	677	229	7101
JRefactory v2.6.24	578	4883	902	79732
Netbeans v1.0.x	6278	28568	7611	317542
JUnit v3.7	104	648	138	4956
JHotDraw v5.1	174	1316	331	8876
MapperXML v1.9.7	257	2120	691	14928
Nutch v0.4	335	1854	1309	23579
PMD v1.8	519	3665	1463	41554

TABLE III: Instances per design pattern

Design pattern	Positive	Negative	Total
Singleton	58	96	154
Adapter	607	603	1210
Factory Method	561	481	1042

TABLE IV: Experimental setup

Parameter	Value
Number of generations	100
Population size	100
External archive size	500
Crossover probability	0.8
Maximum number of derivations	8
Support threshold	0.01
Confidence threshold	0.5
χ^2 threshold	3.841
Coverage threshold	4

the dataset coverage at the end of the G3P4DPD procedure. Additionally, it is worth remarking that the maximum number of derivations determines the limit of the genotype size, i.e. how long a rule can be. This value has been set to 8, as larger values would hardly generate individuals with admissible values of support, while keeping the trade-off with the application of CMAR, which preferably admits more general rules.

Notice that the input grammar has been slightly customised for each specific design pattern. Table V shows which operators have been required to the detection of every pattern. On the one hand, the categorical operators selected for a given design pattern depend on its structure and collaborations, as defined by Gamma *et al.* [1]. Moreover, the elemental design patterns used by Zanoni *et al.* [5] have been also used as a reference. On the other hand, the selection of numerical operators used for detecting the factory method pattern is founded on the study by Issaoui *et al.* [24]. For all other cases, the selection, e.g. software metrics for the singleton and adapter, has been made based on preliminary experiments.

In order to establish a comparative framework, we first need to consider that, to the best of our knowledge, there is no other approach combining all kinds of properties within the detection model. As discussed in Section I, MARPLE is a compelling proposal, but lacks of flexibility and scalability, particularly for incorporating software metrics. Besides, MARPLE has an exhaustive nature, in contrast to the presence of a stochastic component, G3P4DPD, in our model. Therefore, a stratified 5-fold cross-validation has been applied,

TABLE V: Properties used in experimentation

	Singleton	Adapter	F. Method
NOM			✓
NOC	✓	✓	✓
DIT	✓		✓
RFC			✓
isFinal	✓		
isSubclass			✓
controlledInit	✓		
staticField	✓		
staticFlag	✓		
conglomeration			✓
returned	✓	✓	✓
received		✓	
createObj	✓	✓	✓
delegate		✓	
sameElem			✓
typeOf	✓	✓	✓
linkMethod		✓	✓
linkArtefact		✓	✓
ctorVisibility	✓		
aggregation	✓	✓	✓
adapter		✓	

as well as 30 executions of G3P4DPD with random seeds to reduce bias of the performance evaluation.

The results obtained for the 3 DPs under analysis are shown in Table VI. The performance is observed in terms of 5 classification evaluation measures, giving the average value of all executions and its standard deviation. In short, the *accuracy* indicates the percentage of instances correctly classified; the *recall* calculates the amount of DP retrieved over the total number of instances in the repository; the *precision* measures how many design patterns were positively detected among the retrieved instances; the *specificity* measures the proportion of negative instances correctly identified; and F_1 is the harmonic mean of precision and recall. Performance values of MARPLE [5], [25] are also shown in Table VII, as provided by its authors. Here, apart from the aforementioned benefits, results reveal that the presented approach is also competitive in terms of these objective measures. In general, considering its random nature, we can speculate that G3P4DPD shows a robust behaviour according to the values of the standard deviation. It can be also noted the good trade-off between all the classification evaluation measures. As expected, the detection model for the singleton pattern obtains the best results due to its simplicity.

VII. Concluding remarks

Frequently, code traceability is not properly managed in software projects, hampering the maintainability, scalability and understandability of the product. Here, best practices applied by software programmers, such as the use of design patterns, can be missed among thousands of lines of code. The detection of design patterns from software repositories has been positioned as an important task within the field of reverse engineering. This paper presents a novel two-phased model for automatic DPD, based on evolutionary computation and machine learning techniques. Firstly, the G3P4DPD algorithm has been developed to extract from the organisational repository

TABLE VI: Experimental results

	Accuracy	Precision	Recall	Specificity	F_1
Singleton	$0,9389 \pm 0,0442$	$0,9283 \pm 0,0833$	$0,9136 \pm 0,0627$	$0,9541 \pm 0,0571$	$0,9179 \pm 0,0559$
Adapter	$0,8554 \pm 0,0199$	$0,8401 \pm 0,0219$	$0,8797 \pm 0,0252$	$0,8308 \pm 0,0261$	$0,8592 \pm 0,0194$
Factory Method	$0,8337 \pm 0,0224$	$0,8157 \pm 0,0273$	$0,8954 \pm 0,0483$	$0,7617 \pm 0,0483$	$0,8529 \pm 0,0226$

TABLE VII: Performance of MARPLE

	Accuracy	F_1
Singleton	0,9281	0,9026
Adapter	0,8588	0.8478
Factory Method	0,8189	0,8340

rules describing a diversity of software features about the design pattern—or its variants—to be detected. Here, the use of G3P makes the learning process more flexible, as it allows the simultaneous study of software metrics and behavioural and structural properties for describing such patterns, in contrast to current approaches. Secondly, the detection model is built by adopting some procedures of the CMAR algorithm.

We have validated this model by detecting three different design patterns (singleton, adapter and factory method) from the DBP repository, showing excellent results in terms of detection performance. Being currently MARPLE a top-referenced tool for DPD, the obtained results show that our proposal reaches competitive, robust values in terms of objective classification measures like accuracy, precision, recall, specificity and F_1. The use of a CFG for representing specific-domain knowledge makes our proposal more scalable, as multi-valued operators reduce noise and contribute to construct more interpretable classifiers. In addition, the modular design of our model makes it more adaptable, e.g. new pruning strategies or operators could be easily implemented to improve overall results.

In the future we plan to incorporate new operators to provide a more complete support for new design patterns. Furthermore, we speculate that it is worthwhile integrating this automatic model within existing IDE platforms like Eclipse, and adding capabilities for recommending the development of a specific design pattern in a concrete point of the code according to the knowledge extracted from the source repository.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 338–348.

[3] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele, "Design pattern mining enhanced by machine learning," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ser. ICSM '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 295–304.

[4] A. Chihada, S. Jalili, S. M. H. Hasheminejad, and M. H. Zangooei, "Source code and design conformance, design pattern detection from source code by classification approach," *Appl. Soft Comput.*, vol. 26, no. C, pp. 357–367, Jan. 2015.

[5] M. Zanoni, F. Arcelli Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *J. Syst. Softw.*, vol. 103, no. C, pp. 102–117, May 2015.

[6] R. I. Mckay, N. X. Hoai, P. A. Whigham, Y. Shan, and M. O'Neill, "Grammar-based genetic programming: A survey," *Genetic Programming and Evolvable Machines*, vol. 11, no. 3-4, pp. 365–396, Sep. 2010.

[7] F. Thabtah, "A review of associative classification mining," *Knowl. Eng. Rev.*, vol. 22, no. 1, pp. 37–65, Mar. 2007.

[8] W. Li, J. Han, and J. Pei, "Cmar: Accurate and efficient classification based on multiple class-association rules," in *Proceedings of the 2001 IEEE International Conference on Data Mining*, ser. ICDM '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 369–376.

[9] F. A. Fontana, A. Caracciolo, and M. Zanoni, "Dpb: A benchmark for design pattern detection tools," in *2012 16th European Conference on Software Maintenance and Reengineering*, March 2012, pp. 235–244.

[10] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, Nov. 2006.

[11] C. Kramer and L. Prechelt, "Design recovery by automated search for structural design patterns in object-oriented software," in *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, ser. WCRE '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 208–215.

[12] K. Mens and T. Tourwé, "Delving source code with formal concept analysis," *Comput. Lang. Syst. Struct.*, vol. 31, no. 3-4, pp. 183–197, Oct. 2005.

[13] Y.-G. Guéhéneuc, "P-mart: Pattern-like micro architecture repository," *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, 2007.

[14] J. M. Luna, J. R. Romero, and S. Ventura, "Design and behavior study of a grammar-guided genetic programming algorithm for mining association rules," *Knowl. Inf. Syst.*, vol. 32, no. 1, pp. 53–76, Jul. 2012.

[15] Y. Xu, Y. Li, and G. Shaw, "Reliable representations for association rules," *Data Knowl. Eng.*, vol. 70, no. 6, pp. 555–575, Jun. 2011.

[16] B. Liu, W. Hsu, and Y. Ma, "Integrating classification and association rule mining," in *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, ser. KDD'98. AAAI Press, 1998, pp. 80–86.

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[18] J. M. Smith, *Elemental Design Patterns*, 1st ed. Addison-Wesley Professional, 2012.

[19] J. Y. Gil and I. Maman, "Micro patterns in java code," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 97–116.

[20] F. A. Fontana, M. Zanoni, and S. Maggioni, "Using design pattern clues to improve the precision of design pattern detection tools." *Journal of Object Technology*, vol. 10, no. 4, pp. 1–31, 2011.

[21] S. Brin, R. Motwani, and C. Silverstein, "Beyond market baskets: Generalizing association rules to correlations," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '97. New York, NY, USA: ACM, 1997, pp. 265–276.

[22] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, no. 2, pp. 1–12, May 2000.

[23] S. Ventura, C. Romero, A. Zafra, J. A. Delgado, and C. Hervás, "Jclec: a java framework for evolutionary computation," *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, vol. 12, no. 4, pp. 381–392, 2008.

[24] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, "Using metric-based filtering to improve design pattern detection approaches," *Innov. Syst. Softw. Eng.*, vol. 11, no. 1, pp. 39–53, Mar. 2015.

[25] M. Zanoni, "Data mining techniques for design pattern detection." Ph.D. dissertation, Università degli Studi di Milano-Bicocca, 2012.