View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Servicio de Difusión de la Creación Intelectual

El Análisis Estático como Herramienta de Evaluación en Cátedras con Proyectos de Programación

Martín L. Larrea
Departamento de Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur
Bahía Blanca, Buenos Aires, Argentina
mll@cs.uns.edu.ar

Resumen

Desde el punto de vista de la enseñanza de la programación, resulta fundamental la elaboración de metodologías que permitan evaluar dicha programación. Una de las metodologías más usadas por los alumnos y docentes es la de testing dinámico, esto es ejecutar el código, darle una entrada y comparar la salida observada con la esperada. Sin embargo esta estrategia no está libre de restricciones y creemos que puede ser complementada por el testing estático. En este trabajo presentamos esta otra modalidad de testing, algunas herramientas que le dan soporte y mostramos cómo puede ser aplicado en cátedras que incluyen la evaluación de proyectos de programación. La implementación del testing estático da nuevas herramientas, tanto al alumno como al docente, para evaluar y mejorar la calidad del código producido.
Palabras Clave: Análisis Estático, Testing, Evaluación de Proyectos

Abstract

From the point of view of teaching software programming, the development of methodologies that allow evaluating such programming is essential. One of the methodologies most used by students and teachers is dynamic testing, this is executing the code, giving it an input and comparing the observed output with the expected one. However, this strategy is not free of restrictions and we believe that it can be complemented by static testing. In this work we present this other testing modality, some tools that support it and we show how it can be applied in courses that include the evaluation of programming projects. The implementation of static testing gives new tools, both to the student and the teacher, to evaluate and improve the quality of the code produced.
Keywords: Static Analysis, Testing, Project Evaluation

Introducción

La programación como actividad en la formación del alumno de carreras de informática es fundamental y en consecuencia también lo es la evaluación de dicha programación. Tanto el alumno como el docente requieren de técnicas efectivas de programación ([1]) y verificación. El alumno necesita contar con metodologías y herramientas que le permitan evaluar su programa para poder saber si satisface las condiciones de la cátedra, que a su vez son las mismas necesidades que tiene el docente. Hoy en día, la principal estrategia de evaluación de proyectos de programación, tanto del docente como el alumno, es la de ejecutar el código, ingresar un conjunto de parámetros de entrada y evaluar si la salida observada es la esperada ([2], [3], [4]). Este proceso, sin duda válido como proceso de testeo, se conoce en el área de Verificación y Validación de Software (VyVS) como *testing dinámico.* En este trabajo queremos poner el foco de atención a la contraparte del testing dinámico, y es el *testing*

estático. El principal atractivo que tiene el testing dinámico como método de evaluación de un programa o módulo es que resulta muy intuitivo, natural. Al hacer un programa X, una forma directa de evaluar la calidad de X es ejecutarlo y observar si el comportamiento de X es el esperado. Sin embargo, para poder realizar este tipo de control se debe satisfacer una condición fundamental, el programa o módulo X debe estar completo y compilando. Esto significa que durante el tiempo que se está desarrollando X no es posible acceder a un testing dinámico. Tampoco es posible hacer testing dinámico de aquellos elementos de código que en su naturaleza no son ejecutables, por ejemplo una clase abstracta ([5]). Por estos motivos es que creemos que la introducción del testing estático, en particular el análisis estático, es una excelente complementación como herramienta de evaluación para el alumno y el docente. Se define como testing estático al chequeo manual de un objeto, así también como su análisis mediante herramientas, pero en ningún momento el objeto siendo testeado se ejecuta. El análisis estático, que es la observación del objeto de testeo mediante herramientas, permite evaluar elementos de software que no están completos o que no son ejecutables en su naturaleza. La información que nos brindan estas herramientas sirven para evaluar la calidad de los mismos. Quizás la herramienta más conocida de análisis estático es el compilador, pero existen otras más versátiles y poderosas.

El objetivo de este trabajo es presentar el concepto de análisis estático como herramienta de evaluación en cátedras con proyectos de programación, discutir un conjunto de herramientas gratuitas disponibles y mostrar cómo una de estas puede ser aplicable en un bloque de materias de programación. En tal sentido utilizaremos como caso de estudio tres materias del bloque de programación de las carreras ofrecidas por el Departamento de Ciencias e Ingeniería de la Computación de la Universidad Nacional del Sur.

A continuación, se desarrollarán algunos conceptos de VyVS para poder trabajar sobre un conjunto común de definiciones. Seguido presentamos tres herramientas de análisis estático disponibles actualmente y, como caso de estudio, aplicamos una de ellas sobre tres materias de programación. El trabajo concluye con algunas palabras finales y lo que creemos son los siguientes pasos a realizar sobre esta línea de trabajo.

Verificación y Validación de Software

La *Verificación* y la *Validación* son dos formas de evaluar la calidad de un producto de software. La *Verificación* se ocupa de comprobar que el software cumple los requisitos funcionales y no funcionales de su especificación; mientras que la *Validación* tiene a su cargo comprobar que el software cumple las expectativas que el cliente espera. El área de VyVS está compuesta de muchas técnicas ([6]) cuyo objetivo es determinar la correctitud del objeto a testear y diferentes formas de clasificar dichas técnicas. Aquellas que no ejecutan el objeto a testear sino que sólo lo analizan, como un compilador por ejemplo, son clasificadas como técnicas de testing estático específicamente análisis estático. En cambio, las que ejecutan el objeto de testeo, para por ejemplo evaluar si la salida es correcta o no, son técnicas de testing dinámico. El testing estático es mucho más flexible en su aplicación, al no requerir que el objeto bajo testeo sea ejecutado. La lectura por parte de una persona de un código fuente buscando errores es un tipo de testing estático, también lo es la lectura de un documento de texto buscando inconsistencias. El mismo corrector ortográfico puede ser clasificado como testing estático. En [6] el testing estático está dividido en dos categorías,

examinaciones grupales estructuradas y *análisis estáticos*. La primera se concierne con aquellas evaluaciones en donde el analizador del objeto bajo testeo son uno o más humanos. A los fines de este trabajo nos centraremos en la segunda categoría, donde las evaluaciones del objeto bajo testeo son realizadas mediante herramientas de software. No por esto se debe entender que no consideramos la primera categoría importante, todo lo contrario pero no es el foco en este momento.

Herramientas de Análisis Estático

Existen actualmente numerosas herramientas que pueden ser clasificadas como herramientas de análisis estático. En este trabajo vamos a presentar tres, las cuales representan diferentes puntos dentro del abanico de opciones disponibles. Comenzando desde la más sencilla y limitada en funcionalidad como *CyVis* ([7]), pasando luego a *PMD* ([8]) una de complejidad intermedia y finalmente la más compleja *SonarQube* ([9]).

CyVis

CyVis es una herramienta gratuita desarrollada en Java para Java, por lo que es multiplataforma. Funciona tanto sobre archivos de clases o jars permitiendo calcular métricas sobre los mismos. Dentro de las métricas que se calculan se incluyen cantidad de líneas, métodos y la complejidad ciclomática ([10]). La herramienta es de muy fácil uso debido a su simplificada interfaz gráfica (Figura 1). De toda la información que brinda, la complejidad ciclomática es quizás la más importante. Desde un punto de vista académico, esta información le permitiría al alumno poder identificar métodos demasiado complejos y propensos a errores, y también le serviría para saber qué métodos deberían ser reorganizados en métodos de menor complejidad. Sin embargo, tan limitada funcionalidad hace que no sea una herramienta atractiva para la evaluación de proyectos de programación. Aunque la complejidad ciclomática es una métrica importante, la herramienta no detecta errores, ni potenciales errores en el código. Algo que las siguientes herramientas sí pueden hacer.

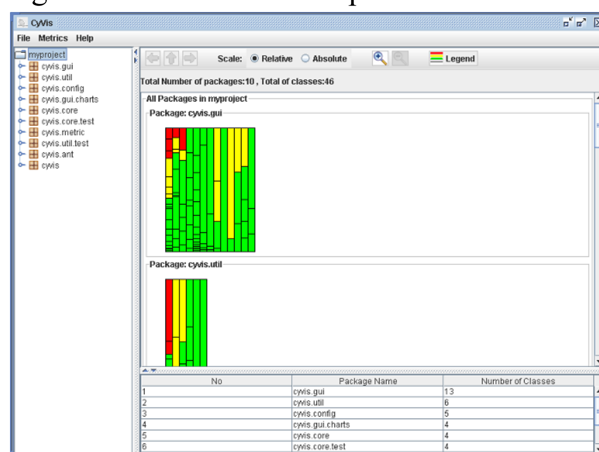


Figura 1. Captura de pantalla de CyVis en la cual se observa una visualización de la complejidad ciclomática de las clases del proyecto. CyVis es una herramienta libre y gratuita desarrollada en Java para Java. La imagen fue obtenida del sitio web oficial de la herramienta ([7]).

PMD

PMD es un analizador estático de código cuyo objetivo es encontrar fallas de programación comunes como variables no utilizadas, bloques de manejo de excepciones vacíos, creación de objetos innecesarios, etc. Fue desarrollado originalmente para trabajar sobre Java y Apex, pero actualmente está disponible para seis lenguajes. PMD opera a partir de un conjunto de reglas (*rulesets*) previamente definidas, cada regla establece qué tipo de control realizará PMD sobre el código fuente. Para poder utilizar PMD primero se debe contar con un *ruleset*, el cual se compone de la selección de reglas existentes dentro de PMD. Actualmente la herramienta ofrece

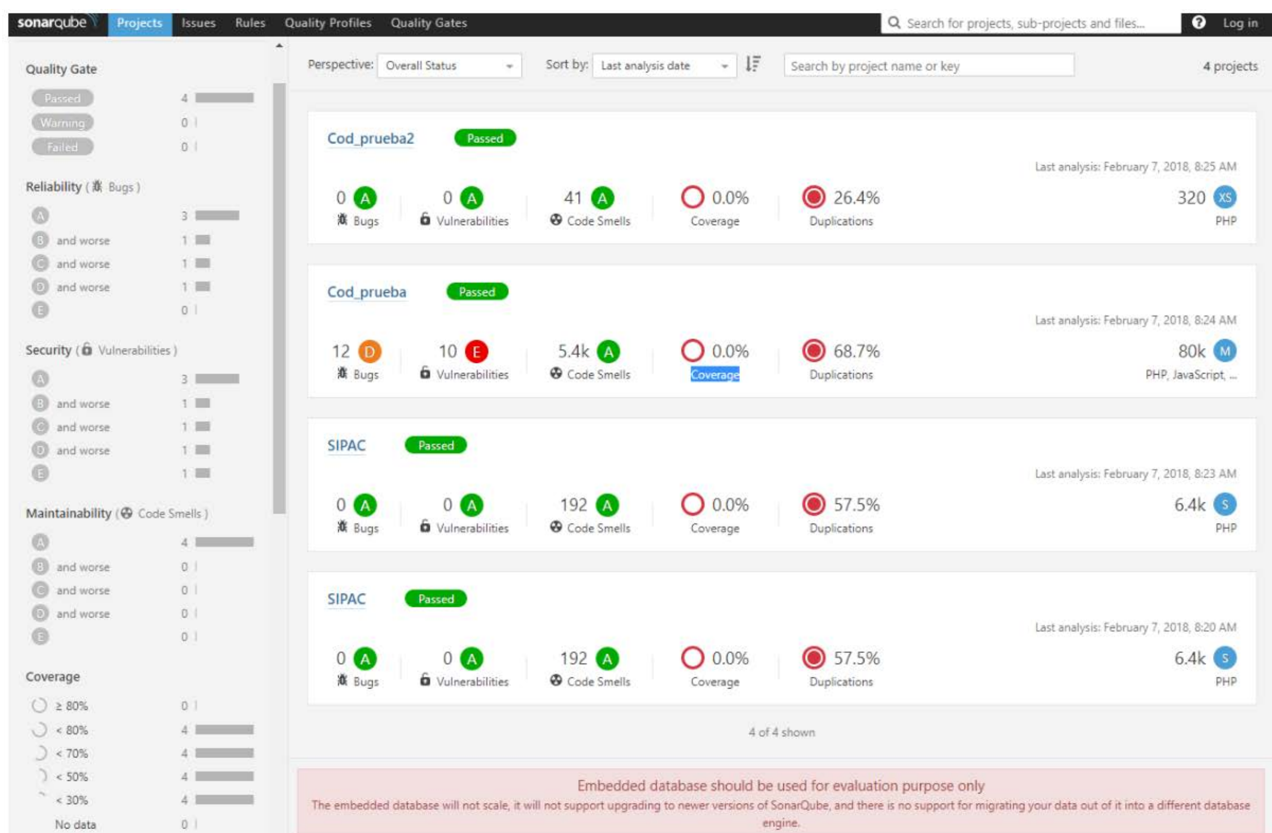


Figura 2. Captura de pantalla de la pantalla de reportes de SonarQube. Aquí se observan los resultados del análisis estático sobre diferentes proyectos, almacenados en repositorios de GitHub. Sin duda esta es la herramienta más poderosa de las tres, y también quizás la más difícil de configurar.

aproximadamente 400 reglas predefinidas y soporta la creación de nuevas reglas. Las reglas existentes se organizan en 8 categorías: *Best Practices, Code Style, Design, Documentation, Error Prone, Multithreading, Performance, Security, Additional rulesets.* Vale aclarar que PMD incluye una regla que analiza la complejidad ciclomática del código, igual que CyVis. Por ejemplo, una cátedra podría establecer como regla de buena programación que todo método debe tener un único punto de salida. Para poder controlar esta condición, en lugar de mirar manualmente todo el código, se podría crear un *ruleset* en PMD con la regla *OnlyOneReturn* ([11]), de la categoría *CodeStyle* y ejecutar PMD sobre el código fuente en cuestión.

La ejecución de PMD es sencilla ya que es una herramienta que funciona por línea de comando, sólo se debe tipear *pmd.bat -d c:\src -R rulesets/java/quickstart.xml -f text* donde *c:\src* es la ubicación del código fuente a analizar y *rulesets/java/quickstart.xml* es el conjunto de reglas para aplicar sobre el código. Los resultados se muestran también por la línea de comando. PMD incluye otra herramienta llamada CPD que se utiliza para encontrar código duplicado, y también funciona por línea de comando. Este software es portable, se distribuye bajo la licencia BSD y posee versiones tanto para Windows como para Linux; además requiere de Java JRE 1.7 o superior.

SonarQube

SonarQube es una plataforma de código abierto para análisis continuo de calidad. Incluye toda la funcionalidad presente en PMD pero además, se presenta como una plataforma web, con una fácil integración con repositorios de código

como GitHub y otras herramientas de testeo. Está disponible para más de 20 lenguajes de programación e incorpora funcionalidades adicionales para la administración y gestión de los procesos de testeo estático y dinámico.
Al igual que con la herramienta anterior, SonarQube ofrece un conjunto de reglas que son los diferentes tipos de análisis que puede realizar sobre el código. Para el lenguaje Java, por ejemplo, actualmente cuenta con 642 reglas ([12]), es decir más de 600 análisis diferentes de calidad del código. El usuario debe indicar qué reglas desea aplicar y la herramienta ofrece un reporte de resultado (Figura 2). Sin duda esta es la herramienta más poderosa de las tres, en especial por su integración con productos de terceros, algo que PMD no puede hacer. Esta ganancia conlleva el costo de una instalación y configuración más compleja que los casos anteriores. SonarQube se distribuye bajo licencia LGPL.

Casos de Prueba

El objetivo de este trabajo es el de introducir el análisis estático como un complemento para la evaluación de proyectos de programación en las cátedras, tanto como una herramienta disponible para el docente como para el alumno. En tal sentido, a continuación vamos a mostrar cómo podemos usar y aprovechar los resultados de una de estas herramientas en tres materias de programación. Cada una de las cuales se ubica en un estadio diferente del desarrollo de la capacidad de programar por parte del alumno.

El Contexto

El Departamento de Ciencias e Ingeniería de la Computación (DCIC), de la Universidad Nacional del Sur, ofrece tres carreras de grado, Ingeniería en Computación, Ingeniería en Sistemas de Información y Licenciatura en Ciencias de la Computación. Estas últimas dos son las que más carga de conceptos de Ingeniería de Software presentan en sus currículas, lo que se refleja en las incumbencias de sus títulos. También común a estas dos carreras es el denominado bloque de programación, un conjunto de cuatro materias de los dos primeros años que son la formación esencial de programación del alumno.
Dicho bloque está formado por las siguientes materias en el orden en el que aparecen en las carreras:
Resolución de Problemas y Algoritmos (RPA): El objetivo principal de la materia es que los alumnos adquieran la capacidad de desarrollar programas para resolver problemas de pequeña escala. El desarrollo de un programa se concibe como un proceso que abarca las etapas de, la interpretación adecuada del enunciado a través del cual se plantea el problema, el diseño de un algoritmo para el problema planteado, ya implementación del algoritmo en un lenguaje de programación imperativo y finalmente la verificación de la solución. La materia se dicta en el primer cuatrimestre del primer año con una carga horaria de 64hs. El lenguaje de programación utilizado es Pascal.
Introducción a la Programación Orientada a Objetos (IPOO). El objetivo de esta asignatura es que los alumnos aprendan y apliquen los fundamentos de la programación orientada a objetos para la implementación en Java de aplicaciones de software de pequeña escala, a partir de una especificación y un diagrama de clases ya diseñado. La materia se dicta en el segundo cuatrimestre del primer año con una carga horaria de 64hs. El lenguaje de programación utilizado es Java.
Estructura de Datos (ED). El objetivo de Estructuras de Datos es que el alumno aprenda con detalle los conceptos de estructuración de datos y los algoritmos para el manejo de las diferentes estructuras de datos que se presentan.

Este proceso se contextualiza en el marco de la programación orientado a objetos y atendiendo a los factores de calidad del software, es especial la eficiencia caracterizada por la noción de orden de tiempo de ejecución de un algoritmo. La materia se dicta en el primer cuatrimestre del segundo año con una carga horaria de 64 hs. El lenguaje de programación utilizado es Java.

Tecnología de Programación (TdP). Esta materia es la cuarta y última materia del bloque inicial de aprendizaje de programación. En ésta los alumnos adquieren un conocimiento más profundo sobre el paradigma de orientación a objetos, sus usos, beneficios y tecnologías asociadas. La materia permite articular los contenidos desarrollados en las materias del área de Programación con los temas que se abordarán a partir de tercer año en las asignaturas del área Desarrollo de Software. La materia tiene tres objetivos primordiales: Introducir nociones elementales de ingeniería de software y el proceso de desarrollo. Establecer la relación del paradigma orientado a objetos con el área de ingeniería de software y el soporte de los conceptos centrales de orientación a objetos en diversos lenguajes de programación. La materia se dicta en el segundo cuatrimestre del segundo año con una carga horaria de 64 hs. El lenguaje de programación utilizado es Java y en algunos casos C#.

La Herramienta

Para este caso de estudio se optó por utilizar PMD. CyVis se consideró muy limitada en su funcionalidad y SonarQube muy compleja, en esta instancia, de instalar y configurar. Con el fin de demostrar las ventajas del análisis estático, PMD ofrece las mejores ventajas sobre las contras antes mencionadas. Lamentablemente PMD no ofrece soporte para el lenguaje de programación PASCAL, lo que deja fuera de este análisis a la materia Resolución de Problemas y Algoritmos. Para este caso de estudio, nos proponemos mostrar en detalle para IPOO y en modo resumido para ED y TdP algunos de los aspectos que se presentan que luego efectivamente mostraremos cómo pueden ser evaluados mediante PMD utilizando proyectos propios de cada cátedra.

IPOO

Como mencionamos recientemente, IPOO introduce al alumno en la Programación Orientada a Objetos. En las Figuras 3, 4, 5 y 6 podemos ver cuatro *slides* del curso en la cual se hacen recomendaciones al alumno a la hora de programar. A partir de estas recomendación, y otras, es que formamos un ruleset denominado *ipoo.xml* el cual se muestra en la Figura 7. Las reglas que se incluyeron, en orden de aparición son:

FieldDeclarationsShouldBeAtStartOfClass. Los campos de una clase deben ser declarados al inicio de la clase.

ClassNamingConventions. Los nombres de las clases deben comenzar con una mayúscula. Esta regla puede ser configurada a gusto de la cátedra a partir del campo *property* que se observa en el ruleset.

AssignmentInOperand. Informa cuando se detecta el uso de la asignación como parte de un condicional.

AssignmentToNonFinalStatic. Informa cuando a una variable estática se le asigna una variable dinámica.

OneDeclarationPerLine. Detecta cuando se realizan múltiples declaraciones por línea, e informa que esto atenta contra la legibilidad del código.

Unused. Múltiples reglas que detectan cuando algún elemento, como puede ser un *import,* un parámetro formal, o una variable local son declarados pero no usados.

OnlyOneReturn. Identifica cuando un método tiene más de un punto de salida.

EL DIAGRAMA DE UNA CLASE

En la etapa de diseño cada clase se modela mediante un **diagrama**:

Nombre
Atributos
Constructores
Comandos
Consultas
Responsabilidades

Los servicios provistos por una clase pueden ser **constructores**, **comandos** o **consultas**.

El diagrama puede incluir **notas** o **comentarios** que describen **restricciones** o la **funcionalidad** de los servicios.

Introducción a la Programación Orientada a Objetos 25

Figura 3. Al momento de programar una clase a los alumnos se les pide que siempre pongan los atributos de clase e instancia luego del nombre de la clase.

CASO DE ESTUDIO: CONTROL DE LA PRESIÓN ARTERIAL

Convenciones

- Declaramos los atributos como privados para que solo sean accesibles dentro de la clase.
- Para **consultar** el valor de cada atributo definimos un **método** que retorna el valor del atributo.
- Cada identificador de clase comienza con una mayúscula, a las variables por lo general le asignamos nombres que comienzan en minúscula.
- Retenemos el orden y los comentarios del diagrama en el código para reflejar su estructura.

Introducción a la Programación Orientada a Objetos 49

Figura 4. Las convenciones de nombres, como por ejemplo utilizar mayúscula al inicio del nombre de una clase puede ser controlado mediante PMD.

Es importante destacar que estas reglas sirven para realizar advertencias, en algunos casos las advertencias pueden ser consideradas errores por parte del alumno, por ejemplo nombrar con minúscula una clase; pero en otros casos puede existir una justificación como por ejemplo en la existencia de dos puntos de salida en un mismo método. Con el *ruleset* creado tal como se muestra en la Figura 7 se procedió a aplicar estas reglas mediante PMD en un proyecto real de la materia. El proyecto en cuestión consta de 5 clases con un total de 189 líneas de código. La Figura 8 muestra la salida de PMD al aplicar *ipoo.xml* sobre el proyecto. En particular, se encontraron múltiples situaciones que no satisfacían las reglas establecidas. Por cada una de estas situaciones, PMD informa el archivo, la línea, la regla que se violó y una breve descripción. PMD detecto 8 situaciones que violaban las reglas establecidas en el *ruleset* de IPOO. En la línea 4 de la clase MatrizGenerica se encontró más de una declaración en una única línea; en la línea 20 de la misma clase se encontró dos variables definidas en la misma línea y que no son usadas luego de su definición.

CONVENCIONES

- Declarar las variables al principio del bloque.
- Usar **identificadores significativos** e **identar** adecuadamente.
- Incluir **comentarios** que describan la **estructura** del código, la **funcionalidad** de cada método y las **responsabilidades** establecidas en el diseño.
- **No exagerar** con los comentarios oscureciendo la lógica de la resolución.
- No escribir comentarios que expliquen características del lenguaje
- Si un método produce un resultado, incluir **una única instrucción de retorno al final**, excepto si todo el código del método es un if-else con instrucciones simples.
- Los atributos de instancia los declaramos **privados**.

Introducción a la Programación Orientada a Objetos 66

Figura 5. Un único punto de salida en aquellos métodos que deban retornar un valor es fácilmente controlable con una regla específica en el *ruleset*.

IDENTIDAD, IGUALDAD Y EQUIVALENCIA

m1 → :PresionArterial maxima=150 minima=75 ← m3
m2 → :PresionArterial maxima=150 minima=72
m4 → :PresionArterial maxima=150 minima=75

```
ig1 = m1 == m2;
ig2 = m1 == m3;
ig3 = m1 == m4;
```

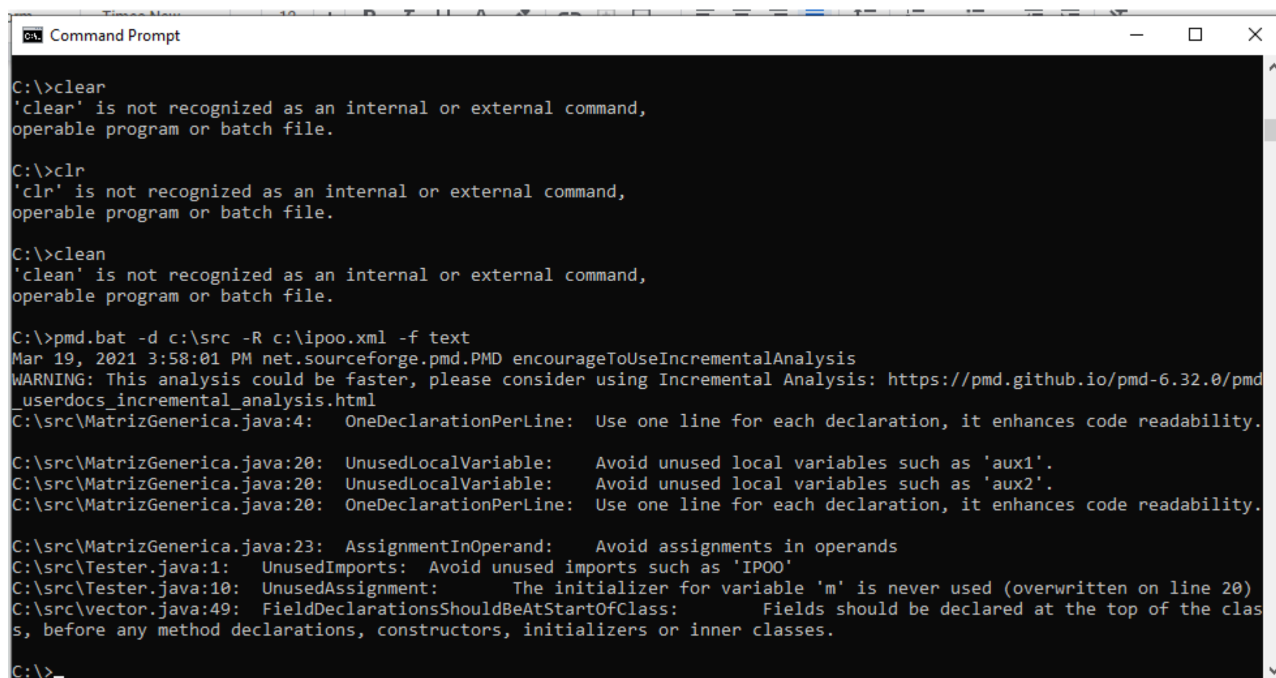
El operador relacional == compara variables de tipo clase, esto es **referencias**.

46

Figura 6. El uso de '=' en un condicional a modo de comparación es un error común y muchas veces difícil de detectar, por lo que la inclusión de una regla para su identificación resulta valioso.

```xml
<?xml version="1.0"?>

<ruleset name="ipoo"
    xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 https://pmd.sourceforge.io/ruleset_2_0_0.xsd">

  <description>
        Ruleset para Introducción a la Programación Orientada a Objetos
  </description>

    <!-- Rules -->
    <rule ref="category/java/codestyle.xml/FieldDeclarationsShouldBeAtStartOfClass" />
    <rule ref="category/java/codestyle.xml/ClassNamingConventions">
    <properties>
        <property name="classPattern" value="[A-Z][a-zA-Z0-9]*" />
        <property name="abstractClassPattern" value="[A-Z][a-zA-Z0-9]*" />
        <property name="interfacePattern" value="[A-Z][a-zA-Z0-9]*" />
        <property name="enumPattern" value="[A-Z][a-zA-Z0-9]*" />
        <property name="annotationPattern" value="[A-Z][a-zA-Z0-9]*" />
    </properties>
    </rule>
    <rule ref="category/java/errorprone.xml/AssignmentInOperand" />
    <rule ref="category/java/errorprone.xml/AssignmentToNonFinalStatic" />
    <rule ref="category/java/bestpractices.xml/OneDeclarationPerLine" />
    <rule ref="category/java/bestpractices.xml/UnusedAssignment" />
    <rule ref="category/java/bestpractices.xml/UnusedFormalParameter" />
    <rule ref="category/java/bestpractices.xml/UnusedImports" />
    <rule ref="category/java/bestpractices.xml/UnusedLocalVariable" />
    <rule ref="category/java/bestpractices.xml/UnusedPrivateField" />
    <rule ref="category/java/bestpractices.xml/UnusedPrivateMethod" />
    <rule ref="category/java/codestyle.xml/OnlyOneReturn" />

</ruleset>
```

Figura 7. Ruleset creado para ser usado por PMD en proyectos de programación de la cátedra de IPOO.



```
C:\>clear
'clear' is not recognized as an internal or external command,
operable program or batch file.

C:\>clr
'clr' is not recognized as an internal or external command,
operable program or batch file.

C:\>clean
'clean' is not recognized as an internal or external command,
operable program or batch file.

C:\>pmd.bat -d c:\src -R c:\ipoo.xml -f text
Mar 19, 2021 3:58:01 PM net.sourceforge.pmd.PMD encourageToUseIncrementalAnalysis
WARNING: This analysis could be faster, please consider using Incremental Analysis: https://pmd.github.io/pmd-6.32.0/pmd
_userdocs_incremental_analysis.html
C:\src\MatrizGenerica.java:4:   OneDeclarationPerLine:  Use one line for each declaration, it enhances code readability.

C:\src\MatrizGenerica.java:20:  UnusedLocalVariable:    Avoid unused local variables such as 'aux1'.
C:\src\MatrizGenerica.java:20:  UnusedLocalVariable:    Avoid unused local variables such as 'aux2'.
C:\src\MatrizGenerica.java:20:  OneDeclarationPerLine:  Use one line for each declaration, it enhances code readability.

C:\src\MatrizGenerica.java:23:  AssignmentInOperand:    Avoid assignments in operands
C:\src\Tester.java:1:   UnusedImports:  Avoid unused imports such as 'IPOO'
C:\src\Tester.java:10:  UnusedAssignment:       The initializer for variable 'm' is never used (overwritten on line 20)
C:\src\vector.java:49:  FieldDeclarationsShouldBeAtStartOfClass:        Fields should be declared at the top of the clas
s, before any method declarations, constructors, initializers or inner classes.

C:\>_
```

Figura 8. Salida que ofrece PMD luego de aplicar ipoo.xml como ruleset a un proyecto entregado por un alumno en IPOO. De la salida se observa que se encontraron múltiples situaciones que violaban las reglas establecidas.

En la línea 23 de esa clase también se encontró el uso de una asignación en un condicional. En la clase Tester se encontró una importación que no es utilizada, y una variable que es definida e inicializada pero luego nunca usada. Finalmente, en la clase Vector, línea 49 se encontró la definición de un atributo que no se encuentra al inicio de la clase.

Vale aclarar en este punto, que algunos de estos errores identificados por PMD en muchos casos también son identificados por algunas de las IDEs disponibles hoy en día. Por ejemplo, Eclipse detecta variables o importaciones no usadas. Sin embargo, PMD puede incluir más reglas que las soportadas por Eclipse, y en el caso particular de IPOO la IDE utilizada no incluye esta funcionalidad.

Debido a las limitaciones de páginas del TE&ET, para las siguientes dos cátedras nos limitaremos a describir qué reglas se incluyeron y qué resultados se obtuvieron sin mostrar los *rulesets* ni las salidas por consola.

ED

La materia ED hace uso del paradigma orientado a objetos visto en IPOO para desarrollar en detalle los conceptos y usos relacionados a las estructuras de datos. En términos del *ruleset* se trabajó en forma incremental a partir del *ruleset* utilizado en IPOO. El nuevo conjunto de reglas, ed.xml, incorporó regales tales como:
EmptyTryBlock. NoPackage, SwitchStmtsShouldHaveDefault, UnnecessaryCast, UnnecessaryConstructor y *AvoidFieldNameMatchingMethodName,* entre otros.

El proyecto Java de la materia ED sobre el cual se aplicó el *ruleset* ed.xml consta de 33 clases con un total de 1671 líneas de código. PMD informó 188 situaciones que violaban reglas establecidas. Entre las situaciones se destacan las siguientes; el uso de modificados de visibilidad redundantes, nombres que no respetan las convenciones de escritura, la comparación de Strings mediante el operador '=' y no el método *equals* y la existencia de métodos con el mismo nombre que un atributo de instancia.

TdP

La última materia del bloque de programación busca acercar al alumno a la programación como desarrollo de software y servir de puente al área de Ingeniería de Software. Tal como ocurrió en la materia anterior, el *ruleset* que se creó para TdP se hizo de manera incremental sobre el de ED. Para tdp.xml se incluyeron reglas vinculadas al manejo de hilos y aquellas asociadas al buen diseño orientado a objeto, como:
GodClass, LawOfDemeter ([13]), SingularField, CyclomaticComplexity y UselessOverridingMethod entre otras.

En esta materia los alumnos concluyen su cursado con el desarrollo de un video juego. Para la prueba de tdp.xml usamos un juego desarrollado en 2019 por tres alumnos; el proyecto cuenta con 94 clases y 2504 líneas de código. PMD encontró 6 atributos de instancias que sólo se usaban en un único método, lo que indicaba que podrían ser reemplazados por variables locales. Dos métodos con complejidad ciclomática mayor a 7. Múltiples violaciones de *Law of Demeter* y finalmente múltiples entidades que eran definidas pero nunca usadas. En este proyecto, motivado por su envergadura, se decidió a probar la herramienta para detección de código duplicado CPD. En este caso se ejecutó por línea de comando *cpd.bat --minimum-tokens 70 --files c:\src* lo que detectó dos métodos iguales en dos clases encargadas de gestionar los movimientos del personaje del juego.

Conclusiones y Trabajo a Futuro

La corrección de proyectos de programación es una tarea que lleva tiempo y consume recursos, en especial recursos humanos. Para poder ser más eficientes en el uso de ese tiempo debemos incorporar nuevas metodologías de testeo, metodologías que complementen y no reemplacen a las actuales. En tal sentido es que introducimos la noción de testing estático y en particular el análisis estático. Esta metodología es soportada por múltiples herramientas con diferentes grados de funcionalidad. A fines de este trabajo, se introdujeron tres herramientas y una de ellas fue utilizada para evaluar proyectos reales de tres materias del bloque de programación de nuestra unidad académica. Creemos que estas herramientas, a disposición de los alumnos y docentes, sería una oportunidad para mejorar la calidad del código y acercar nuevas herramientas a las cátedras.
Como trabajo a futuro, pretendemos trabajar con las cátedras del bloque de programación del DCIC-UNS para crear en conjunto sus respectivos *rulesets* y tutoriales necesarios para que tanto docentes como alumnos puedan comenzar a utilizarlos. También pretendemos avanzar en el uso de SonarQube para su integración con GitHub y con procesos de automatización del testing mediante Jenkins ([14]).

Referencias Bibliográficas

[1] Moroni N., & Señas, P. *Estrategias para la enseñanza de la programación*. I Jornadas de Educación en Informática y TICs en Argentina. 2005.

[2] Wong, W. E., Bertolino, A., Debroy, V., Mathur, A., Offutt, J., & Vouk, M. *Teaching software testing: Experiences, lessons learned and the path forward*. In 2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T) (pp. 530-534). IEEE. 2011.

[3] Calatrava Arroyo, A., Ramos Montes, M., & Segrelles Quilis, J. D. *A Pilot Experience with Software Programming Environments as a Service for Teaching Activities*. Applied Sciences, 11(1), 341. 2011.

[4] Elena García Barriocanal, Miguel-Ángel Sicilia Urbán, Ignacio Aedo Cuevas, and Paloma Díaz Pérez. *An experience in integrating automated unit testing practices in an introductory programming course*. SIGCSE Bull. 34(4) (pp. 125–128.) 2002.

[5] Binder, R. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional. 2000.

[6] Jorgensen, P. C. *Software testing: a craftsman's approach*. CRC press. 2018.

[7] Garg, Mohit, and Mohit Garg. *Analysing The Quality Attributes of AOP using CYVIS Tool*. International Journal of Computers & Technology 4.2c2. 648-653. 2013.

[8] Copeland, T. *PMD applied*. Vol. 10. Arexandria, Va, USA: Centennial Books, 2005.

[9] Campbell, G. Ann, and Patroklos P. Papapetrou. *SonarQube in action.* Manning Publications Co., 2013.

[10] Ebert, Christof, et al. *Cyclomatic complexity*. IEEE software 33.6. 27-29. 2016,

[11] PMD Source Code Analyzer Project. Rule References. Java Rules. Category: Code Style, *OnlyOneReturn*, https://pmd.github.io/latest/pmd_rules_java_codestyle.html#onlyonereturn

[12] SonarQube, Code Quality and Code Security. Java static code analysis, https://rules.sonarsource.com/java

[13] Lieberherr, K. J., & Holland, I. M. *Assuring good style for object-oriented programs*. IEEE software, 6(5), 38-48. 1989.

[14] Jenkins, Open source automation server. https://www.jenkins.io/