ITBA
Instituto Tecnológico
de Buenos Aires

INSTITUTO TECNOLÓGICO DE BUENOS AIRES – ITBA

ESCUELA DE INGENIERÍA Y GESTIÓN

Code Manipulation in PMD

AUTORES: Comercio Vázquez, Matías Nicolás (Leg. Nº 55309)
Ibars Ingman, Gonzalo Exequiel (Leg. Nº 54126)

DOCENTE TITULAR O TUTOR: Sotuyo Dodero, Juan Martín

TRABAJO FINAL PRESENTADO PARA LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN INFORMÁTICA

Lugar: Av Eduardo Madero 399, C.A.B.A., Argentina
Fecha: 14 de agosto de 2018

Instituto Tecnológico de Buenos Aires

Proyecto Final presentado para la Obtención del Título de Ingeniero en Informática

Code Manipulation in PMD

Comercio Vázquez, Matías Nicolás (Legajo 55309)
mcomerciovazque@itba.edu.ar

Ibars Ingman, Gonzalo Exequiel (Legajo 54126)
gibarsin@itba.edu.ar

Supervisado por

Sotuyo Dodero, Juan Martín

14 de agosto de 2018

Índice

1. Resumen 4

2. PMD 4

3. Objetivos Iniciales 4

4. Premisas 6

5. Timeline 6

6. Desarrollo 8

6.1. Análisis 1: Overview de PMD 8

6.2. Flujo de trabajo ideal 9

6.3. Análisis 2: Herramientas existentes de reparación 10

6.3.1. Eclipse 11

6.3.2. Intellij IDEA 12

6.4. Propuesta 1: Eclipse Like 14

6.4.1. Manipulación: ¿AST o Código Fuente? 14

6.4.2. Operaciones de texto sobre un documento 16

6.4.3. Token Genérico 17

6.4.4. Manejo de eventos sobre el AST 19

6.5. Análisis 3: Implementación Demostrativa Uso 20

6.6. Objetivos Reformulados 24

6.7. Análisis 4: JavaParser 26

6.7.1. Expresiones Lambda en JavaParser 28
6.8. Propuesta 2: Manipulación de tokens por regiones 29
6.9. Propuesta 3: Manipulación explícita de tokens 33
6.10. Propuesta 4: Estructuras de Nodos 35
6.10.1. StructureElement 36
6.10.2. Sincronización 37
6.10.3. NodeMetaInfo 38
6.11. Propuesta 5 (final): Estructuras RegExp 41
6.11.1. Introducción 42
6.11.2. Presentación de la idea 45
6.11.3. Flujo de manipulación 50
6.11.4. Flujo de creación 58
6.11.5. Definición de NodeSyntax 62
6.11.6. Parser: Instanciación 66
6.11.7. Parser: Funcionamiento 74
6.11.8. Sincronización 84
6.11.9. Manipulación de hijos de nodos 90
6.11.10.Utilización en un nuevo lenguaje 91
6.11.11.Definición de nuevos *StructureElements* 92
6.12. Anexo 94
6.12.1. Conversión de AST a Texto 94
6.12.2. Aplicación de fixes en el flujo de PMD 95
6.12.3. Reporte de una violación con fix 95

6.12.4. Cache/Análisis Incremental para fixes 96
6.12.5. Versionado 96

7. Conclusiones **99**

8. Trabajo Futuro **100**

Referencias **100**

1. Resumen

En el siguiente informe se presenta de manera detallada la arquitectura propuesta para manipular código fuente en la herramienta de análisis de código *PMD*[3], que en un futuro podría ser utilizada para implementar un sistema de arreglo automático de código en dicha herramienta. Además, se resume el proceso de análisis y diseño que permitió llegar a esta arquitectura final, y se mencionan y referencian otros trabajos realizados a lo largo de este proceso.

2. PMD

PMD es un analizador estático de código. Encuentra (potenciales) bugs o malas prácticas comunes en programas incluyendo pero no siendo extensivo a variables no utilizadas y creación de objetos innecesarios. Soporta Java, JavaScript, Salesforce.com Apex y Visualforce, PLSQL, Apache Velocity, XML, XSL.

Adicionalmente incluye un Copy Paste Detector (CPD). Este encuentra código duplicado en todos los lenguajes mencionados anteriormente y además en C, C++, C#, Groovy, PHP, Groovy, Ruby, Fortran, Scala, Objetive C, Matlab, Python, Go y Swift.

Ya que el mismo es un proyecto open source[1], cualquier desarrollador puede realizar desde reportes de issues hasta aportes. La herramienta se encuentra en desarrollo continuo, estableciendo objetivos a corto y mediano plazo[2].

3. Objetivos Iniciales

Actualmente, PMD es capaz de recorrer un Abstract Syntax Tree (AST) para detectar violaciones en el código. Estas violaciones son detectadas por reglas que pueden estar escritas de dos maneras: utilizando el patrón visitor sobre el AST o realizando consultas XPath. Debido a la manera en que están

[1] https://github.com/pmd/pmd
[2] https://github.com/pmd/pmd/wiki/Roadmap-and-future-directions

modelados los nodos, se puede conocer la siguiente información que está involucrada en cada violación:

- Archivo
- Token(s) involucrados
- Líneas y columnas involucradas

El objetivo inicial principal era proveer diferentes maneras de modelar *autofixes* (es decir, reparación automática de código) opcionales para las violaciones detectadas, de tal manera que la herramienta no solamente las detecte sino que también provea y arregle las mismas de forma automática.

Los autofixes debían ser expuestos de la siguiente manera:

El sistema de autofixes debía proveer la capacidad para:

- Ser utilizado a través de PMD cuando se lo ejecuta a través de la línea de comandos, directamente aplicando los arreglos sobre el código fuente. Esto, entre otras cosas, implicaba:
  - Generar las modificaciones de código necesarias para integrar el flujo de autofixes a PMD.
  - Diseñar e implementar una arquitectura que permitiera modificar (directa o indirectamente) el código fuente.
  - Implementar la posibilidad de escribir autofixes a través de: 1) implementaciones en código Java; 2) un sistema acorde al que se utiliza para escribir reglas XPath. El objetivo de esto era mantener la mayor consitencia posible con las formas en que se escribían las reglas de detección de violaciones.
  - Permitir la escritura de autofixes
  - Soportar el *cacheo* de los autofixes en caso de que no se deseen aplicar los mismos en el momento de detectar las violaciones de tal manera que en una ejecución posterior se apliquen si el usuario lo deseara. Para ello se debía aprovechar el sistema de *caché* existente en PMD.
- Ser utilizado en los IDEs que contaran con el plug-in de PMD (como es el caso de Eclipse, donde los autofixes de PMD actuarían como los *quick fixes* de este IDE).

- Integrarse con herramientas generadoras de reportes, como Arcanist, para informar los arreglos realizados.

4. Premisas

Las premisas que se tuvieron en mente durante el proceso de diseño e implementación de todas las arquitecturas fueron:

- Facilidad de uso para los usuarios finales de PMD, lo que significa mantener bajo el costo de adopción de usuarios nuevos de la herramienta en lo relacionado a la nueva funcionalidad de autofixes.
- Escalabilidad: Facilidad de adaptación a nuevos lenguajes de PMD, incluyendo las nuevas features de cada uno de ellos.
- Mantenibilidad (sobre todo para los desarrolladores de PMD).
- La cantidad de modificaciones sobre el código existente en PMD debe ser la menor posible, y en caso de tener que realizar estas modificaciones, evitar Breaking API Changes[3].
- Las reparaciones automáticas de código debían modificar solamente los sectores involucrados, evitando la reescritura completa del código fuente.

5. Timeline

En las figuras 1 y 2 se presentan los tiempos de análisis y desarrollo desde el comienzo hasta el final del proyecto, utilizando como unidad de medida el mes. El proyecto se extendió desde abril de 2017 hasta julio de 2018. Los meses que se encuentran en formato negrita son aquellos que consideramos fueron los más importantes en cuanto a avances significativos en el proyecto.

En la sección 6 se detallan cada una de las etapas del proyecto, que se clasifican en *Análisis* y *Propuesta*, según el siguiente criterio.

[3] Un Breaking API Change implica realizar cambios sobre cualquier entidad públicamente accesible, por ejemplo, la firma de los métodos de una interfaz, lo que evitaría poder garantizar compatibilidad entre los minor releases de la herramienta.

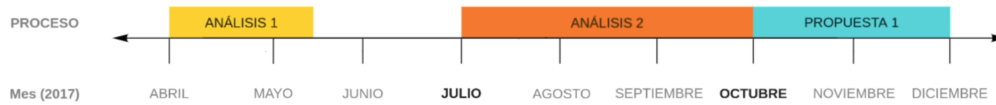

Figura 1: Análisis y Propuestas durante 2017

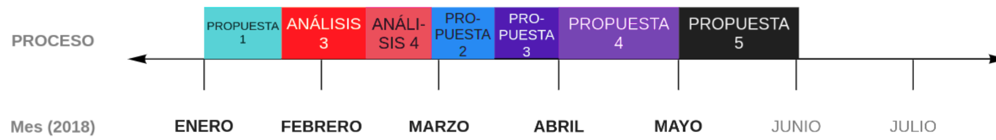

Figura 2: Análisis y Propuestas durante 2018

- Análisis: comprende el estudio y comprensión de funcionalidades de herramientas existentes que puedan resultar de utilidad para lograr los objetivos propuestos.
- Propuesta: comprende la etapa de planificación e implementación de una arquitectura candidata a lograr los objetivos propuestos.

El proyecto en términos generales lo desarrollamos de forma continua a excepción de los meses mayo, junio y diciembre de 2017 debido a parciales, entregas de trabajos prácticos o exámenes finales.

6. Desarrollo

6.1. Análisis 1: Overview de PMD

Antes de comenzar a proponer soluciones se analizó el flujo principal de PMD. Esto implicó leer y entender su implementación. De esta manera se buscó poder proyectar cómo se iba a introducir la funcionalidad deseada. En este proceso fue de vital importancia consultar a nuestro tutor no sólo sobre las decisiones de implementación tomadas en la herramienta hasta ese momento sino también para que nos guiara en las zonas de código (clases e interfaces) más importantes a considerar.

Dentro de la herramienta, el módulo que más interesó analizar fue *core* que contiene toda la funcionalidad principal y es independiente de cualquier lenguaje de programación. Toda implementación requerida por los lenguajes se encuentra en módulos separados, uno por lenguaje. Dentro del módulo core, nos centramos en comprender:

- La clase *AbstractNode* que es la clase base de la que extienden todos los nodos especializados de cada lenguaje, donde los nodos son utilizados para construir un Abstract Syntax Tree (AST) del código fuente.
- De qué manera se detectan las violaciones en un archivo.
- Una vez encontrada una violación, de qué manera se reporta.

Independientemente de la implementación a realizar, se decidió planificar cómo iba a ser el flujo principal que involucraba, una vez encontradas, arreglar las violaciones y plasmar estos arreglos en el código fuente. Para ello se debían resolver dos funciones principales:

1. Una función que reciba el AST y los fixes y devolviera un AST con los fixes aplicados, es decir, el código arreglado representado como un AST.
2. Transformar el AST en código fuente y guardarlo en el mismo archivo que el original.

Para encarar el problema en el primer punto, se decidió investigar soluciones de otras herramientas como las de los IDEs Eclipse[5] e Intellij IDEA[7] que proveen de esta funcionalidad.

Para el segundo punto, como PMD requiere de un parser por lenguaje que convierte la cadena de tokens de un código en un AST, se creyó que también existía una función inversa que automáticamente tradujera el AST en código fuente o bien que esta función pudiese generarse automáticamente. Sin embargo, dicha abstracción no existía y tampoco fue de interés agregarla hasta ese momento en la herramienta ya que la manera en que se realizan los reportes de las violaciones requiere de traducir únicamente regiones con violaciones sobre el código, sino conocer regiones particulares del código.

Cabe destacar que la decisión de modificar primeramente el AST y que estos cambios se tradujeran en cambios en el código fuente no fue trivial (de hecho, no fue la que se consideró originalmente), puesto que esta decisión cambiaba radicalmente el enfoque del problema y la dificultad de su resolución. La otra alternativa era proveer la arquitectura de autofixes para que éstos modificaran directamente el código fuente (es decir, sin modificar el AST). Sin embargo, luego de diversos debates de las limitaciones de esta implementación y del potencial de la segunda solución en lo que respecta a las premisas estipuladas (ver sección 4), por no mencionar que la vasta mayoría de herramientas que implementan los fixes de esta forma, concordamos con nuestro tutor meses más adelante que continuar con la segunda opción era la mejor alternativa de las dos. Para más detalle sobre esta decisión, referirse a la sección 6.4.1.

6.2. Flujo de trabajo ideal

Debido a los compromisos que se debieron cumplir en la Universidad a partir de mayo de 2017, se decidió suspender la continuidad del proyecto hasta julio de 2017.

A partir de julio, la metodología de trabajo propuesta al tutor fue la siguiente:

1. Decidir el orden de resolución de los problemas con el objetivo de llegar a los objetivos planteados.
2. Realizar un análisis exhaustivo sobre soluciones existentes, si las hubie-

ra.

3. Proponer una solución al tutor. En caso de ser aceptada, dividir la implementación en Pull Requests (PR) a realizar en el repositorio de PMD. En caso de ser rechazada, dependiendo de las razones, iterar sobre la misma o descartarla por completo. En este punto, cabe aclarar con respecto a los PRs:
   - Cada PR debe ser conciso, es decir, poseer una cantidad de líneas que lo haga auditable.
   - Cada PR debe poseer la suite de tests correspondiente a la funcionalidad implementada junto con la documentación adecuada.
   - Se debe iterar el PR hasta que el mismo se considere aceptado por parte de los auditores.

El medio de comunicación principal con nuestro tutor fue Google Hangouts[4], que lo utilizamos para realizar consulas, muestras mensuales de avances realizados, recibir recomendaciones sobre herramientas o implementaciones que pudieran adaptarse para alcanzar los objetivos planteados (como fue el caso de los *quick fixes* de Eclipse analizado en la sección 6.3 e implementado en la sección 6.4), y demás. Cabe destacar que nuestro tutor estuvo siempre disponible (sin importar ni el día ni la hora) para comunicarse con nosotros por estas cuestiones. Además, al ser uno de los principales contribuidores y uno de los dueños del repositorio de PMD, siempre se encargó del auditado y aceptación de los PRs.

6.3. Análisis 2: Herramientas existentes de reparación

Llevó aproximadamente 3 meses analizar soluciones existentes, desde julio hasta octubre. Las dificultades en la comprensión de los flujos de las herramientas analizadas impactaron directamente en el tiempo del primer análisis sobre las cuales basamos la propuesta inicial. Antes de continuar, cabe destacar que la sugerencia de realizar este análisis, en particular el de la herramienta de Eclipse, fue sugerido por nuestro tutor.

[4] `https://hangouts.google.com/`

6.3.1. Eclipse

Todo el código involucrado en el flujo de fixes es extenso e incluso se han encontrado clases con alrededor de 5000 líneas. Si bien presenta una documentación extensa, no siempre es clara y debía recurrirse a la implementación. También se realizaron pruebas creando un proyecto únicamente con dependencias de Eclipse para probar la manipulación del AST, que sirvió para conocer la manera en que se expone a los desarrolladores el modelado de fixes. Es decir, que en el caso de Eclipse, los usuarios de esta arquitectura realizan manipulaciones sobre los nodos del AST, y estas manipulaciones, mediante un sistema adecuado, se traducen en operaciones sobre el código fuente. Toda esta (manipulación del AST y traducción del AST a código fuente) es la arquitectura que luego se comenzó a implementar, según lo detallado en la sección 6.4.

Una de las clases encontradas en las que se basó esta primer propuesta fue *Document*[4] del paquete *JFace*. Esta clase se encarga de representar el código fuente como un String al que se le aplican operaciones de agregado, modificación y borrado en una región específica del archivo. Las operaciones se aplican sobre el documento luego de asegurar que no se van a realizar más operaciones.

Las operaciones se guardan en un árbol en la que no existe restricción sobre la cantidad de hijos que puede tener un nodo. De esta manera si existía una operación que estaba incluida en la región de otra operación ya existente, entonces esa nueva operación seguro es descendiente de aquella que la contiene. En caso de que se intentara insertar una operación cuya región interesecara con la región de otra operación, se consideraba un error y no se permitía el agregado de la misma. La razón del no agregado es que no se podía determinar un orden total de las operaciones, sino parcial. Ésto fue un punto importante a discutir previo a la implementación sobre PMD ya que o bien se debía asegurar que este caso nunca ocurriría o resolverlo de otro modo. En la documentación de Eclipse no figuraba el por qué podía llegar a ocurrir. También se realizó una consulta[6] en los foros de Eclipse pero no hubo respuesta.

Finalmente, llegamos a la conclusión que por la manera en que se iban a aplicar los fixes, nunca podía ocurrir. Se da un detalle más profundo en la sección 6.4, donde se implementa no solamente esto sino también la conversión de operaciones de AST a operaciones de texto.

Lo primero que se pensó fue en transladar la clase con sus respectivas dependencias a PMD y adaptarlo. Sin embargo, esto fue desestimado inmediatamente (con el consejo de nuestro tutor) porque con total seguridad se estaba acarreando funcionalidad que no se iba a utilizar y eso conllevaba a que el tamaño de PMD aumentase innecesariamente, impactando directamente sobre toda la comunidad. Dicho esto, se procedió a hacer una implementación sin utilizar código de la dependencia, tal como se menciona en la sección 6.4.

6.3.2. Intellij IDEA

Intellij IDEA[7] es un IDE que provee de manera nativa el equivalente a quick fixes en Eclipse; marca a través de la misma interfaz que la de edición de texto mejoras a diferentes secciones de código. La manera en que estos quick fixes se exponen es a través de intentos de acción que muestra una lista de operaciones a aplicar sobre el código. Si el usuario decide aplicar una mejora que IDEA propone al código del lenguaje en el que se esté programando, se realiza una modificación sobre una representación del archivo que está en memoria, reflejando además los cambios finalmente en el archivo si es que se decide guardar el mismo.

En su implementación, Intellij es la capa que se ocupa de parsear los archivos y crear el modelo sintáctico y semántico para realizar manipulación de código; esta capa lleva el nombre de PSI (Program Structure Interface).

Para representar los archivos el modelo utilizado es el de un árbol PSI, compuesto por una jerarquía de elementos PSI, y estos pueden contener descendientes que son a su vez elementos PSI. Para cada lenguaje se debe realizar una extensión de la interfaz un archivo PSI, por ejemplo, una interfaz[8] para Java y luego una implementación de la misma.

Intellij parsea archivos en 2 pasos:

1. Se construye un AST con nodos que son implementaciones de la interfaz **ASTNode**. Cada nodo AST tiene un tipo de elemento asociado **IElementType** y esos tipos son definidos por el plugin del lenguaje. Al nodo raíz se le da un tipo especial a través de la interfaz **IFileElementType**. Los nodos AST tienen una relación o mapeo directo a regiones de texto en el documento que se presenta. Los nodos hoja representan tokens individuales, mientras que nodos de niveleas más alto representan fragmentos de múltiples tokens. Las operaciones a realizar

sobre los nodos AST, tales como inserción, borrado o reordenamiento se reflejan sobre el texto del documento.

2. Se construye un árbol PSI sobre el AST agregando semántica y métodos para la manipulación específica de cada lenguaje; en otras palabras es una capa que cubre toda la implementación genérica de nodos AST y la utiliza para un lenguaje, pudiendo realizar operaciones específicas, como modificaciones del nombre de un método o el agregado de modificadores visibilidad, para el caso de Java, utilizando estos métodos. Los nodos PSI están representados por clases que implementan la interfaz **PsiElement** y son creados por el plugin del lenguaje en el método *ParserDefinition.createElement()*. Así como se distingue el caso del nodo raíz en el AST, también se distingue en el nodo raíz PSI.

Existe un manual[9] en el que se muestra qué se debe implementar en un plugin para soportar un lenguaje nuevo. Lo primero que fue evaluado en este ejemplo es qué tan verborrágico era implementar un fix respecto de la complejidad que nosotros consideramos para lo que está cambiando.

Para agregar soporte a un lenguaje nuevo, y es esta una de las razones que se va a comentar por las cuales se decidió desestimar este camino, se mencionan algunas funcionalidades que deben implementarse en el plugin del lenguaje:

- Registro de tipo de archivo
- Implementación del lexer: define cómo los contenidos de un archivo se convierten a tokens
- Implementación del parser del lenguaje y PSI (comentado anteriormente)
- Syntax y Error Highlighting
- References and Resolve: Permite a los usuarios encontrar del uso de un elemento PSI (por ejemplo el acceso a una variable o un método) la declaración de ese elemento.
- Autocompletado
- Encontrar usos de una declaración: es el proceso inverso de **References and Resolve**. En vez de buscar la declaración a partir de un uso, buscar

todos los usos de una declaración de, por ejemplo, una variable o un método.

- Refactorización por nombres
- Safe Delete Refactoring: Al momento de eliminar alguna referencia, advertir de los usos y al confirmar, tener las implementaciones de los nodos que permiten el safe delete.
- Code Formatter
- Code Inspections and Intentions

La principal razón por la cual se descartó este modelo fue que la cantidad y la complejidad de código a integrar a PMD atentaba contra las premisas de trabajo planteadas en la sección 4, sobre todo en lo que conscierne a escalabilidad, mantenibilidad y breaking API changes. De hecho, la cantidad y complejidad del código eran superiores a las propuestas por la arquitectura de Eclipse, sin mencionar que en el caso de Eclipse la comunidad de desarrolladores es mucho más vasta y abierta a consultas. Sin embargo, es importante destacar que la documentación de Intellij facilitó considerablemente la lectura de la implementación que proponen y que ésta en comparación con la de Eclipse es mucho más rica en contenido.

Dicho esto, como se puede ver en la sección 6.4, la primera propuesta implementada estuvo fuertemente basada en el modelado propuesto por Eclipse.

6.4. Propuesta 1: Eclipse Like

6.4.1. Manipulación: ¿AST o Código Fuente?

Uno de los puntos más discutidos durante la etapa de análisis previa e incluso durante el desarrollo de esta etapa fue el hecho de decidir si modelar los fixes de manera tal que estos se apliquen directamente sobre el código fuente (es decir, sin modificar el AST), o bien si modelarlos de manera tal que se modifique el AST y finalmente, de alguna forma, estos cambios se reflejen en el código fuente.

Luego del análisis de la sección anterior, se hacía evidente que las herramientas implementadas realizaban modificaciones sobre el AST para luego

traducirlas a cambios sobre el código fuente. Y en un primer momento, este fue el approach que comenzamos a implementar. Sin embargo, nuestro tutor cuestionó el hecho de por qué los fixes no devolvían directamente los cambios a realizar sobre el código fuente (o una variante semidirecta de esta opción), por lo cual antes de contestarle con completa certeza, tuvimos que realizar una mayor tarea de análisis sobre esta cuestión.

Cabe destacar que tomar un camino u otro cambiaba completamente la complejidad del problema a resolver, dado que implementar un sistema que modifique el AST y que luego esos cambios se impacten sobre el código fuente requería por lo menos el doble de trabajo de nuestra parte, puesto que no sólo involucraba crear la estructura para proveer los fixes, sino también la arquitectura necesaria para realizar la traducción de cambios del AST a cambios en el código fuente.

Las razones (de nuevo, discutidas un sinfín de veces entre nosotros) por las que se decidió optar por realizar las modificaciones sobre el AST y luego traducir esos cambios a cambios en el código fuente fueron principalmente tres:

- En uno de los flujos de funcionamiento de PMD, cada regla que revisa violaciones realiza una *visita* por todos el AST creando los reportes de violaciones que haya encontrado. El fix de cada uno de esos reportes de violaciones debía realizarse en el momento en que terminara la visita dicha regla, previo a la recorrida de la siguiente regla. Con esto, las modificaciones realizadas por los fixes de estas violaciones debían poder ser vistas por las sucesivas reglas que visiten el AST, para poder trabajar sobre un contexto correctamente actualizado.
  En definitiva, dado que las reglas que detectan violaciones trabajan analizando el AST y no el código fuente, las modificaciones debían estar impactadas sobre el AST.

- La interfaz a implementar para realizar la manipulación del código debía ser fácilmente utilizable por los usuarios de PMD que desearan escribir los fixes (esto es una de las premisas de trabajo detallada en la sección 4). Dado que escribir fixes manipulando las propiedades de los nodos (y por ende el AST) es inmensamente más sencillo que escribir manipulaciones del código fuente, esta opción parecía ser la correcta en este sentido. No sólo eso, sino también que al ocurrir que en muchas otras herramientas de manipulación y reparacion de código las modi-

ficaciones se realizaban sobre los nodos, implementarlo de esta forma aseguraba una compatibilidad con las costumbres y modelos mentales de usabilidad que los usuarios poseen.

- La cantidad y complejidad de código a agregar en cada fix para realizar la modificación del código fuente correspondiente era notoriamente superior a la necesaria para realizar las manipulaciones a través de las propiedades de los nodos y que éstas se traduzcan (de manera transparente a los usuarios) a cambios en el código fuente. Esta cantidad y complejidad de código atentaban contra las premisas de facilidad de uso, mantenibilidad, y escalabilidad especificadas en la sección 4. Además, la posibilidad de repetición de código por cada uno de los fixes para manejar casos similares, conjuntamente con el hecho de requerir conocimiento especializado sobre cómo PMD maneja internamente el código fuente, hacían a esta opción (que cada fix manipulara de manera semidirecta o directa el código fuente) aún menos atractiva.

6.4.2. Operaciones de texto sobre un documento

La primer propuesta que se realizó para pasar el código arreglado a archivo es la mencionada en la sección 6.3.1. Los cambios completos pueden visualizarse en el siguiente PR: `https://github.com/pmd/pmd/pull/828`.

Para ello, se representó el archivo a aplicarle operaciones de modificaciones como una interfaz llamada **Document**, en la cuál se le aplican operaciones (arreglos) de inserción, modificación o borrado en una región del archivo. Las regiones se representan de dos maneras, según cuando se utilicen:

- **RegionByLine** es una interfaz que representa una sección de texto en el archivo pudiendo obtener el número de línea y columna en donde comienza y termina la operación, es decir, una tupla *(beginLine, beginColumn, endLine, endColumn)*. Se adoptó esta representación de región ya que PMD en su implementación original de los tokens utiliza la misma representación, si bien no utilizando esta interfaz, ya que si se creara un objeto **RegionByLine** por token, podría suponer un overhead de memoria innecesario.

- **RegionByOffset** es una interfaz que almacena una tupla *(beginOffset, length)* que significa el índice en donde comienza la región y la longitud

de la misma, utilizando como unidad de longitud 1 byte. Esta conversión desde **RegionByLine** es posible ya que al momento de crear un documento, se almacena un mapa *(Line, Offset)*, es decir, cada línea del archivo se mapea a un índice cuya unidad es la misma que el offset del **RegionByLine**. Se crea al momento de pasar la operación al documento. Ésto facilita la aplicación de las operaciones sobre el archivo, como es explicado más adelante.

En la implementación de **Document**, se decidió utilizar una lista para almacenar los fixes ordenados por región, es decir, si se tienen 2 regiones, aquella que termina antes que el comienzo de la otra poseerá un índice menor en la lista (comparando instancias de **RegionByOffset**). Este orden se utiliza para recorrer la lista una única vez y al mismo tiempo que se recorren los bytes del archivo para reescribirlo. De esta forma, cuando se copian los bytes del archivo, se revisa si la posición actual del cursor se encuentra en el comienzo de la próxima operación a aplicar, revisando el *beginOffset*; si es así, entonces se aplica esa operación y se saltean tantos bytes del archivo original como longitud de la región. En el caso de una inserción, la longitud de una región es igual a cero, en el borrado es la cantidad de bytes a eliminar y en la modificación, la cantidad de bytes que se reemplazan por otro conjunto de bytes (notar que el reemplazo se puede ver como una composición de una operación de borrado seguida de otra de inserción). Para evitar tener que mantener todo el archivo en memoria, se utilizaron un buffer de lectura y otro de escritura.

La implementación comentada previamente fue aceptada[5] e integrada a PMD 6.1.0, si bien no utilizada como parte de la propuesta final, debido al descarte de utilizar como modelo a Eclipse.

6.4.3. Token Genérico

La implementación de tokens está reflejada por la implementación de los parsers de lenguajes, que almacena información como:

- La referencia al siguiente token
- Una referencia opcional a un token especial, que puede contener, por ejemplo, comentarios de código.

[5] `https://github.com/pmd/pmd/pull/828/files`

- La imagen o su representación en texto
- Región del token (explicado previamente)

Si bien todos los lenguajes poseían una clase equivalente y generada automáticamente por JavaCC, no estaba centralizada en el módulo *core*. Para poder realizar una manipulación de tokens independiente de los lenguajes, seguro había que realizar una interfaz común a todos los lenguajes y debido a eso se realizó otro aporte a la herramienta[6] que agrega la interfaz **GenericToken** en el módulo *core* y que todas las implementaciones generadas por JavaCC extiendan de esa interfaz. Esto también ayuda a introducir una funcionalidad adicional a la herramienta, independiente del arreglo de código.

Ya que PMD utiliza una cache con el objetivo de evitar analizar archivos que no fueron modificados y a los cuales ya se les generó un reporte, fue un objetivo encontrar una manera de extender la funcionalidad existente sin agregar un overhead significativo al almacenar los fixes además de las violaciones. Fue claro en su momento que existían diversas maneras de almacenar los fixes, pero dependían de la manera en que estos se modelaran:

- En la existencia de una clase que representara un fix, se serializaría la instancia.
- Almacenar las operaciones de texto.

La implementación sobre la que se decidió implementar y es compatible con la propuesta final (sección 6.11), si bien no integrada a la herramienta se explica en la sección 6.12.

La primer propuesta encarada para transformar el AST en texto fue construir una entidad que, por lenguaje, realice un parseo del AST y devuelva un String que sea el código fuente. Notar que cualquier nodo puede ser un nodo raíz del AST, no necesariamente debe ser el nodo raíz del archivo fuente, con lo cuál se podría traducir solamente la porción de código necesaria. Existía el problema que se debía realizar una implementación de un parser por lenguaje para reconstruir todo el código y la barrera de entrada para introducir un nuevo lenguaje sería alta para los desarrolladores de PMD, violando una de las premisas. Debido a esto, se perdería todo el formato que tuviese el archivo (e.g. espacios o tabs, carriage, line feeds) ya que el traductor debería

[6] `https://github.com/pmd/pmd/pull/679`

imponer su manera de representar esa clase de caracteres entre tokens. En términos de performance, sin importar qué porcentaje del código que fuera cambiado, se regeneraría todo el código, teniendo que mantener el String en memoria, agregando un overhead que dependía del tamaño de los archivos a analizar. En este caso se llegó a realizar una implementación parcial pero fue descartada a la semana.

Luego de saber que la lista de tokens modificados también almacenaba caracteres de formato como los mencionados anteriormente, la segunda opción fue recorrer directamente esa lista de tokens. De esta manera no sólo evitábamos mantener el código fuente repetido en memoria, respecto de la solución anterior, sino que también se mantenía el formato del archivo en su mayor parte, evitando tener que ejecutar posteriormente al arreglo de código, alguna herramienta que corrija el formato del mismo.

Para poder realizar una manipulación de los nodos de un AST, sin importar de qué lenguaje se tratase, se decidió agregar operaciones de inserción, modificación y borrado de nodos sobre **AbstractNode**.

El caso de la remoción de hijos directos de un nodo o de la remoción de un nodo respecto de su padre se resolvió en el siguiente PR: `https://github.com/pmd/pmd/pull/696`. Para la inserción, reemplazo y remoción de nodos descendientes, los cambios completos pueden visualizarse en el siguiente PR: `https://github.com/pmd/pmd/pull/858`. En este último PR se presentó un punto de inflexión, explicado en 6.5.

6.4.4. Manejo de eventos sobre el AST

Para poder poder traducir las operaciones realizadas sobre el AST a operaciones de texto, se partió de analizar y adaptar el sistema que utiliza Eclipse; adaptar ya que la implementación de los nodos es completamente diferente al de PMD. En el caso de Eclipse, los nodos poseen como parte de su estructura campos cuyo tipo es específico. En cambio, en PMD, todos los nodos comparten la misma estructura, teniendo una lista de hijos cuyo tipo no es especificado.

El sistema de Eclipse utiliza eventos, llamados *RewriteEvent*s, que describen los cambios que se realizaron sobre los nodos. Se distinguen dos tipos de nodos: Nodos lista y nodos tradicionales. En líneas generales, los eventos para los nodos tradicionales pueden ser de tipo *Inserted*, *Removed*, *Replaced*.

En este caso, las operaciones se realizan sobre propiedades que pueden ser propias o de algún hijo. Los eventos para los nodos lista pueden ser *Children Changed* y *Unchanged*. Cada vez que se realizan cambios sobre algún nodo, esos cambios se traducen a eventos y son enviados a un flujo común a cualquier nodo para almacenar los mismos. Cuando se requiere convertir los cambios sobre el AST a operaciones de texto, se realiza un procesamiento de esos eventos y se obtienen las operaciones de texto a aplicar sobre el documento.

El problema principal que existe es que los eventos estan preparados para realizar operaciones sobre propiedades de los nodos. Como en PMD no existe esa estructura, había 2 maneras de resolverlo: crear una estructura genérica que mantuviera propiedades de los nodos para los cuales hay que proponer una implementación para cada uno de ellos (esta es la forma que Eclipse adoptó) o bien buscar la manera de realizarlo de la manera más genérica posible. El primer caso no se consideró factible ya que la cantidad de código que se hubiera tenido que agregar hubiese sido considerablemente grande y también lo hubiera hecho poco mantenible. La implementación genérica era más atractiva ya que buscaba implementar la menor cantidad de código posible y fue la que finalmente se decidió implementar[7].

Si bien se llegó a realizar una implementación parcial que parte de obtener los eventos y almacenarlos, el problema era manejar el historial de operaciones sobre un nodo y traducirlos a operaciones de texto, ya que se debía asegurar una implementación por nodo que replicara las ideas de Eclipse, para hacerlo funcional. Es por esto que, tal como se menciona en la sección 6.5, esta opción se terminó descartando por completo.

6.5. Análisis 3: Implementación Demostrativa Uso

En la audición del PR de la propuesta 1 en el que se realiza la inserción, reemplazo y remoción de nodos hijos, nuestro tutor realizó el siguiente comentario[8]:

> I'm not particularly happy however at removing a method we just added in 6.0.0. I fear some lack of proper planning on how the feature will work, by focusing too early on the abstract operations

[7] https://github.com/MatiasComercio/pmd/tree/add-rewrite-events
[8] https://github.com/pmd/pmd/pull/858#discussion_r164313003

on the node (bottom-up), before actually figuring out what the higher layers of abstraction would require (top-down).

El comentario estaba totalmente acertado ya que realizamos cambios que implicaban breaking API changes con respecto a cambios introducidos en un PR nuestro anteriormente aprobado.

A partir de este momento nos dimos cuenta de que el sistema de realizar PRs en la medida en que la solución se fuera desarrollando era poco eficiente ya que, dada la dimensión del problema a resolver, entendíamos que los cambios que se podrían introducir en PMD podrían no ser finales.

Este hecho se consideró un punto de inflexión en el proyecto, sobre todo en lo que respecta a la manera en que se estaba encarando el cumplimiento de los objetivos. Así, se decidió frenar la implementación actual y re-pensar, de manera top-down, el flujo de autofixes a implementar en PMD.

Para ello, se decidió implementar un flujo semifuncional completo [9] de aplicación de fixes, intentando realizar cambios sobre el AST para la regla *ForLoopCanBeForeachRule*[10]. Se dice *semifuncional* puesto que la idea era, a partir de realizar modificaciones sobre el proyecto de PMD, detectar exactamente las zonas donde debían introducirse nuevas funcionalidades, y desarrollar una arquitectura de prueba que permitiera de manera rápida probar la factibilidad de la propuesta a realizar para el flujo de autofixes.

Lo que hicimos es, básicamente, mostrar los cambios necesarios para:

- Proveer una interfaz para las reglas para poder reportar las violaciones.
- Aplicar cambios sobre el AST.
- Traducir los cambios a texto.

Esto sirvió enormemente para:

1. Demostrar al tutor cuáles eran nuestras intenciones futuras de implementación, de manera tal de sincronizar los conocimientos y la justificación de las decisiones que estábamos tomando.

[9] https://github.com/MatiasComercio/pmd/commits/test-autofix

[10] `https://pmd.github.io/pmd-6.3.0/pmd_rules_java_bestpractices.html#forloopcanbeforeach`

2. Darnos cuenta de que la factibilidad de continuar con la propuesta 1 era poca, sobre todo por lo que concierne a:

    a) Implementar el sistema de manejo de historial de manipulaciones sobre cada uno de los nodos, necesario para realizar la traducción de estos cambios a texto.
    Se intentó adaptar de manera genérica el manejo de este historial, aunque sin éxito. Esto se debió principalmente a la incompatibilidad existente en el tratamiento de los nodos por parte de Eclipse y de PMD (recordar, por ejemplo, la diferencia entre nodos lista y nodos tradicionales existente en Eclipse y no en PMD, marcada en la sección 6.4.4).
    La otra opción era adaptar el sistema de nodos de Eclipse para PMD, lo que no sólo aumentaba de sobremanera el esfuerzo requerido para implemementar los fixes, sino que también violaba la premisa de introducir la menor cantidad de modificaciones posibles sobre el código fuente de PMD.
    De hecho, se hubiera tenido que adaptar una gran parte del sistema existente del módulo *core* que es compartido por todos los lenguajes, sin mencionar la alta probabilidad de necesitar adaptar cada uno de los lenguajes ya implementados en PMD, lo que hacía aún más inviable esta opción.

    b) La cantidad de código que debía implementarse por lenguaje para soportar esta traducción de cambios sobre el AST a texto. **En este punto, fue vital reconocer que la arquitectura a proponer debía *contentar* dos tipos de usuarios distintos:**

        1) Los usuarios finales: son aquellos que utilizarán el sistema de fixes para programar correcciones automáticas de las violaciones detectadas por las reglas.
        2) Los usuarios internos: son aquellos que adaptarán el sistema de fixes para soportar su utilización en un nuevo lenguaje dentro de PMD.

    En el momento en que se reconocieron estos dos tipos de usuarios, las premisas originales de trabajo se adaptaron, puesto que ahora no sólo debía considerarse la facilidad de uso de la solución que se propusiera, sino también la facilidad de adaptación a nuevos lenguajes. El reconocimiento de este hecho no fue tenido en cuenta hasta este momento; sin embargo, se decidió impactar desde un principio en la sección de las premisas (ver sección 4).

En definitiva, con el trabajo realizado, reconocimos que la implementación de una arquitectura similar a la que propone Eclipse era inviable, no sólo por la gran dificultad y esfuerzo que significaba atacar el problema del manejo del historial de *RewriteEvent*s, sino también por lo relacionado con los usuarios internos de PMD, puesto que la cantidad de líneas necesarias para adaptar esta solución para un nuevo lenguaje excedía con facilidad las 10 mil líneas de código, las cuales eran en su vasta mayoría de alta complejidad.

En este último aspecto, independientemente de la cantidad de líneas de código, la principal complejidad que se presentaba era la traducción de los *RewriteEvent*s a operaciones efectivas a realizar sobre el código fuente, puesto que se debía analizar particularmente el contexto en que dicho *RewriteEvent* fue ejecutado para realizar la correcta manipulación. Es decir, la traducción del *RewriteEvent* a operación sobre texto debía especializarse por cada nodo de cada lenguaje, y además, dentro de un mismo nodo, debían considerarse todos los casos posibles para dejar el código en un estado correcto luego del cambio a realizar.

Por ejemplo, supongamos el caso en que se tiene una expresión lambda de Java como la siguiente:

```
1 unused -> { /* some code */ }
```

Supongamos que el parámetro *unused* no se usa dentro del código de la expresión lambda, por lo que alguna regla de PMD detecta esto y el fix asociado pretende remover el parámetro. Si se utilizara el sistema de fixes de Eclipse, el cambio se realizará inmediatamente sobre el AST y además se generará un *RewriteEvent* de tipo *Removed*, que guardará, entre otras cosas, el nodo que acaba de removerse (en este caso, el que contiene al parámetro). Luego, cuando terminen de realizarse todas las modificaciones sobre el AST, se recorrerán todos los *RewriteEvent*s almacenados (en este caso uno sólo), y cada uno se traducirá a cambios sobre el código fuente. En la arquitectura modelada por Eclipse, la forma de realizar esta traducción es procesar el *RewriteEvent* en el nodo que corresponda, en este caso, en el nodo que representa a la expresión lambda cuyo parámetro se removió. La lógica implementada en tal caso debe tener en cuenta, entre otras cosas:

- Considerar el tipo de operación realizada para efectuar el cambio correspondiente en el texto.

- Al tratarse de un remove, recordar de remover la coma asociada a este parámetro, que puede ser la que esté antes o después, considerando también los casos especiales que son la remoción del primer y del último parámetro.
- Si no quedan parámetros, agregar los paréntesis correspondientes en caso de que no existan.

Notar que programar estos chequeos, conjuntamente con el resto de los casos a implementar incluso sólo para un nodo de un lenguaje, significan una complejidad y una cantidad de líneas de código que atentan principalmente contra las premisas de mantenibilidad y escalabilidad listadas en la sección 4.

Antes de continuar, cabe mencionar que el modelado de fixes de Intellij IDEA analizado en la sección 6.3.2 fue descartado junto con el de Eclipse puesto que presentaba limitaciones similares al modelado de Eclipse en cuanto a la adaptación de la solución propuesta a PMD y a su escalabilidad en lo que respecta a la adopción para cada lenguaje. En particular, notar que en estas herramientas (y en todas las analizadas hasta el momento y en las que se analizaron después) las arquitecturas de nodos y de fixes estaban completamente acopladas, cada una aprovechando al máximo las features de la otra arquitectura.

6.6. Objetivos Reformulados

Tal como se menciona en la sección anterior (6.5), reconocer la limitación de adaptar el sistema de fixes de Eclipse para PMD fue un punto de inflexión en el proyecto. En particular, los dos hechos más destacados en este aspecto fueron:

- La detección de que el manejo del historial de *RewriteEvent*s era prácticamente inviable.
- La escalabilidad de la arquitectura era muy pobre en lo que respecta a su adaptación para diferentes lenguajes, afectando directamente a los usuarios internos de nuestra propuesta en PMD.

Notar que ambas cuestiones están ligadas directamente con el hecho de las manipulaciones sobre el código y su traducción a cambios en el código fuente.

Esto nos llevó a considerar que, más que el flujo de autofixes a integrar en PMD en sí, el *core* de la cuestión consistía en poder implementar un flujo de manipulación que cumpliera lo mejor posible las premisas de la sección 4.

Una vez planteado el flujo de uso esperado para los autofixes (ver sección 6.5), se concentró toda la atención y la energía en la arquitectura que permitiría manipular los nodos AST y traducir estas manipulaciones en los cambios correspondientes en el código fuente.

Con el objetivo de acelerar los tiempos de diseño y desarrollo de cada una de las alternativas, se decidió crear un proyecto similar a PMD pero muchísimo más simplificado, es decir, que tuviera solamente la mínima cantidad de clases necesarias para garantizar que la arquitectura a desarrollar fuese lo más independiente del proyecto actual posible, pero que pueda asegurar su integración en un futuro. Además, este nuevo proyecto poseía implementado solamente un lenguaje, que consistía en una simplificación del lenguaje Java, lo que permitía generar algunas expresiones explícitamente seleccionadas por sus características representativas de casos variados en los que podría darse la manipulación de código. Con esto, lo que se tuvo es una cantidad mucho más reducida de nodos sobre los cuales probar las diferentes propuestas, facilitando así la velocidad de iteración de las mismas.

Notar que esto se alejó de la metodología de trabajo pactada con el tutor (ver sección 6.2); sin embargo, permitía atacar el problema de la manipulación de código de manera mucho más concreta y previniendo cuáles iban a tener que ser exactamente los cambios a introducir en PMD en un futuro, solucionando la falta de planificación en lo que respecta al funcionamiento de la arquitectura, según lo marcado en la sección 6.5.

Con esto, el objetivo reformulado del proyecto, y por lo tanto, el de los análisis y las propuestas de las siguientes secciones, es resolver el problema de la manipulación del AST y la traducción de estas modificaciones a cambios en el código fuente. Para ello, lo que se hará es diseñar una arquitectura específica para la implementación actual de PMD que cumpla con las premisas de la seccion 4. Además, se implementará una prueba de concepto que muestre que la arquitectura propuesta cumple con los requisitos especificados.

6.7. Análisis 4: JavaParser

Con el foco ya puesto en lo que tiene que ver estrictamente con la manipulación de código, se analizaron herramientas que estuviesen orientadas a lograr este propósito, sin importar el lenguaje para el que estuviesen implementadas, puesto que lo que se quería extraer era la idea, la arquitectura que permitía su funcionamiento.

Entre todas las herramientas analizadas, JavaParser [11] fue la más destacada. En particular, contaba con dos features que la hacían sobresalir por sobre el resto.

- *LexicalPreservation*: es la feature que permite realizar manipulaciones del código (sobre los nodos del AST) que al traducirse en cambios de texto a aplicar sobre el código fuente sólo modifican las secciones correspondientes, manteniendo así el estilo de código original lo mayor posible. Esta feature se explica con muchísima claridad en el libro de JavaParser [10].
- *ConcreteSyntaxModel*: es una clase donde se definen las estructuras de todos los nodos (de Java) en base a elementos individuales que describen una característica propia del nodo.

Ambas ideas, mutuamente relacionadas, permitirían cumplir completamente todas las premisas estipuladas en la sección 4, sobre todo en lo que concierne a facilidad de uso y de adaptación.

Sin embargo, se encontraros varias limitaciones.

En primer lugar, el mecanismo que aprovechaba las bondades de los elementos individuales, llamados *CsmElement*s (que permiten escribir las estructuras de los nodos en la clase *ConcreteSyntaxModel* para realizar el *LexicalPreservation*) estaba muy customizado para funcionar con los elementos ya implementados y no era fácilmente extensible. Es decir, las características de los nuevos lenguajes debían de sí o sí adaptarse al uso de estos elementos (lo cual no era garantía de seguridad de funcionamiento) o se debían realizar las modificaciones en este aspecto para garantizar esta adaptabilidad, lo que significaba cambiar considerablemente su lógica de funcionamiento. El costo de adaptación de la arquitectura, en este sentido, era alto.

En segundo lugar, existían aspectos particulares de la implementación de

la herramienta que diferían considerablemente de PMD.

1. Los nodos eran parseados de forma de garantizar ciertas propiedades que eran aprovechadas por las clases que permitían la manipulacion de código. Por ejemplo, cada nodo poseía un campo que representaba a un tipo de hijo en particular, por lo que el orden de los hijos era intrínseco al tipo de nodo (cada nodo sabe cuál es el orden de sus hijos, y el tipo de ellos), no al orden dado en un vector de hijos como en el caso de PMD (donde además, cada nodo de un lenguaje, al menos en todos los casos analizados, no sabe de qué tipo son sus hijos a menos que realice una búsqueda interna sobre ellos).

2. Al igual que en Eclipse, se volvía a encontrar la diferenciación entre nodos tradicionales y nodos lista realizada en Eclipse (ver sección 6.4.4).

3. Todos los nodos poseían código automáticamente generado en base a sus características particulares (otra feature increíble de esta herramienta), que era utilizado por la arquitectura de manipulación de código.

Todas estas particularidades hacían que la adaptación de la arquitectura de manipulación propuesta por *JavaParser* fuese aún menos directa, pudiendo derivar en incluso mayores dificultades que las encontradas al implementar el modelo de Eclipse.

Al final de esta sección analizamos el caso de los nodos que representan expresiones lambda de Java en *JavaParser* para mostrar estas particularidades mencionadas, y mostraremos brevemente algunas de las diferencias con este mismo tipo de nodo en el caso de PMD.

En tercer lugar, luego de realizar pruebas de benchmarking, se encontró que cuando se mantenía el estilo de código usando *LexicalPreservation*, los tiempos de ejecución (incluso sin hacer ninguna modificación) eran considerablemente superiores que cuando no se lo mantenía, lo cual es de por sí inadmisible en PMD.

Al evaluar estas limitaciones se decidió en un principio descartar esta opción. Sin embargo, las features destacadas y la arquitectura planteada en JavaParser gustaron tanto que fueron fuentes inspiradoras de las arquitecturas propuestas en las secciones 6.10 y 6.11, formando parte entonces de la propuesta final.

6.7.1. Expresiones Lambda en JavaParser

En el fragmento de código 1 se muestra un poco del código de la clase que representa a las expresiones lambda en *JavaParser*.

```
1  public final class LambdaExpr extends Expression implements
    ↪  NodeWithParameters<LambdaExpr> {
2    private NodeList<Parameter> parameters;
3    private boolean isEnclosingParameters;
4    private Statement body;
5    // ...
6
    ↪  @Generated("com.github.javaparser.generator.core.node.PropertyGenerator")
7    public Statement getBody() {
8        return body;
9    }
10
11
    ↪  @Generated("com.github.javaparser.generator.core.node.PropertyGenerator")
12   public LambdaExpr setBody(final Statement body) {
13       assertNotNull(body);
14       if (body == this.body) {
15           return (LambdaExpr) this;
16       }
17       notifyPropertyChange(ObservableProperty.BODY, this.body, body);
18       if (this.body != null)
19           this.body.setParentNode(null);
20       this.body = body;
21       setAsParentNodeOf(body);
22       return this;
23   }
24   // ...
25 }
```

Código 1: Expresión lambda en *JavaParser*.

Notar el uso de campos particulares que reconocen el tipo de hijos que posee este nodo en esta herramienta: un único hijo de tipo lista (de manera similar a lo que ocurre en Eclipse), que representa a los parámetros de la expresión lambda, y otro hijo que representa el cuerpo de la expresión. Además, notar que se guardan propiedades utilizadas específicamente para la manipulación de código y su posterior conversión a texto, como el campo *isEnclosingParameters*, que es *true* en el caso de que los parámetros se encuentren rodeados por paréntesis. En PMD, cada uno de los parámetros de una expresión lambda es tratado de manera indistinta que los nodos que

representan el cuerpo de la expresión lambda (en lo que respecta a la relación de hijo), puesto que todos pertenecen al mismo vector. Además, la arquitectura de PMD está pensada para realizar la manipulación de los hijos de un nodo (el agregado, el reemplazo y la remoción) de manera genérica, y no particularizada para cada nodo de cada lenguaje (como sucede en *JavaParser*) puesto que implementar cada caso requeriría un enorme esfuerzo de mantenimiento y escalabilidad de la herramienta (con esto nos referimos a todas las features existentes en PMD).

También se puede ver el uso de la generación automática de código en los métodos anotados como *@Generated*. Este tipo de métodos (y también las propiedades que son autogeneradas, como el caso del enum *ObservableProperty*) son justamente algunos de los que permiten realizar la manipulación de código mucho más amena y particularizada para cada tipo de nodo, aprovechando al máximo sus propiedades. Esta feature no existe en PMD hoy en día, y agregar una arquitectura de este tipo implicaría no sólo violar la premisa de realizar la menor cantidad de cambios posibles sobre la herramienta 4, sino que también agregaría una carga extremadamente alta de trabajo a la ya representada por la necesidad de diseñar una arquitectura que permita la manipulación en sí.

6.8. Propuesta 2: Manipulación de tokens por regiones

El principal problema que enfrentábamos entonces es que había muchísimas arquitecturas diferentes de manipulación de código implementadas en otras herramientas, y cada una estaba desarrollada de manera particular conforme a sus necesidades. Como se analizó en cada uno de los casos anteriores, y en base al fracaso al intentar adoptar el modelo de fixes de Eclipse a PMD, el costo de adoptar una arquitectura de manipulación fiel a la implementación actual de PMD que respetara las premisas listadas en la sección 4 era alto, cuando no muy alto, requiriendo en muchos casos cambios sobre el código existente de PMD, lo cual no era para nada negociable.

Fue por estas razones que decidimos que en base a la experiencia que habíamos adquirido analizando y entendiendo el funcionamiento de otras herramientas era factible crear un sistema de manipulación de nodos y de traducción de dichas manipulaciones a cambios en el código fuente que estuviese específicamente diseñado para funcionar sobre la implementación actual de nuestra herramienta (PMD).

El primer approach pensado (pero no implementado sino hasta después) consistió en traducir inmediatamente las manipulaciones sobre los nodos en cambios sobre los tokens de PMD asociados al nodo donde se producen dichas manipulaciones.

Sin embargo, previo a su desarrollo, se intentó realizar una generalización de este approach para evitar trabajar directamente con la manipulación de tokens, ya que se entendía que era un proceso que podría llegar a ser muy tedioso. Esta generalización conforma la propuesta de esta sección.

La idea detrás de toda esta arquitectura era poder simplificar y generalizar completamente la traducción de manipulaciones en el AST a cambios en la cadena de tokens, haciendo que las operaciones de remoción e inserción de propiedades e hijos de los nodos consista simplemente en desensamblar y ensamblar regiones de tokens.

Recordemos que según lo detallado en la sección 6.7, *JavaParser* permite describir la estructura de los nodos a través de elementos estructurales que representan cada una de sus características. Para evitar tener que crear todo el sistema de elementos estructurales, lo que se intentó hacer fue crear descripciones de los nodos en base a tipos de *regiones* tokens que engloben propiedades de los nodos, y en caso de que dicha propiedad fuese manipulada, que los tokens de esa región sean directamente manipulados según correspondiera. Notemos que con este approach se lograba reducir notablemente la complejidad de implementación de la idea encontrada en JavaParser, puesto que sólo se identificaban regiones de tokens según propiedades, algo que todos los nodos de todos los lenguajes de PMD poseen, permitiendo de esta forma generalizar el modelo de manipulación propuesto por esta herramienta y adaptarlo según el código existente en PMD.

Se identificaron tres tipos de regiones:

- *ChildRegion*: consistían en regiones que abarcaban tokens que eran propios de algún hijo del nodo en cuestión.
- *ExtendedChildRegion*: consistían en regiones que abarcaban una o más *ChildRegion* y tokens que existían debido a la relación de cada uno de estos *ChildRegion*.
- *OwnRegion*: consistían en regiones que abarcaban tokens propios del nodo en cuestión, que representaban propiedades o características propias, y cuya existencia no dependía de la presencia/ausencia de sus

hijos.

Por ejemplo, considerar el caso del nodo *FormalParameters* de Java, que representa los parmámetros de un método, inclyuendo los paréntesis. En este caso, estos nodos eran segmentados en regiones de la siguiente forma.

- Los paréntesis eran categorizados dentro de elementos *OwnRegion*.
- Todos los parámetros (que eran nodos hijo del nodo *FormalParameters* de tipo *FormalParameter*) y los tokens de comas que separan a cada hijo eran considerados como *ExtendedChildRegion*, con los tokens de cada hijo categorizados dentro de un *ChildRegion* y cada coma puesta en su *OwnRegion* correspondiente.

Con esto, la estructura en regiones de un nodo de tipo *FormalParameters* quedaba conformada como sigue:

```
1  FormalParameters:
2    - OwnRegion("("),
3    - ExtendedChildRegion(OwnRegion(","), ChildRegion(FormalParameter))
4    - OwnRegion(")")
```

Código 2: Estructura en regiones de un nodo *FormalParameters* de Java

Con esto, lo que se hacía previo a cualquier modificación sobre el nodo en cuestión, era utilizar la estructura en regiones para escanear los tokens actuales del nodo y catalogarlos en regiones de tokens, que luego serían utilizadas para facilitar la manipulación de la cadena de tokens subyacente que representa el código fuente.

Una vez catalogadas las regiones de tokens, bastaba realizar la manipulación deseada. Por ejemplo, para remover los tokens asociados a un parámetro, bastaba con detectar la región de tokens asociada al parámetro removido (es decir, al nodo hijo que representa el parámetro removido), pedirle su primer y último token, y desensamblar estos tokens de la cadena (para ello, se debía buscar el token previo al primer token y setearle como *next* el siguiente token del último token de la región removida).

Si bien la descripción de manipulación de la cadena de tokens parece sencilla, esta propuesta presentaba múltiples dificultades.

En primer lugar, la detección de las regiones de tokens era un problema en sí, puesto que debía sincronizarse cada región descrita en la estructura con los tokens correspondiente en la cadena parseada. Para ello, debía implementarse un sistema que permita no sólo hacer esta sincronización, sino también que sea lo suficientemente sofisticado como para detectar cada uno de los hijos y sus regiones de tokens, además de tener que .ªdivinar"la cantidad de hijos o veces que puede repetirse una región, sin mencionar la ubicación de la aparición de cada uno de los elementos dentro de las regiones de tipo *ExtendedChildRegion*. La implementación realizada estaba especializada en un primer momento para nodos que tuviesen la *pinta* de los nodos de tipo *FormalParameters*, pero al intentar realizar una generalización para otros casos, su inviabilidad fue puesta en evidencia.

Para entender un poco más esto, tomemos la estructura mostrada en el fragmento 2. Notemos que un nodo de tipo *FormalParameters* puede tener más de un hijo de tipo *FormalParameter*, y que estos hijos están separados por comas. Sin embargo, en el fragmento mostrado, no se indica ni la cantidad de veces que se espera que aparezca esta región así como tampoco si el orden de detección será siempre "una coma, y luego un hijo", o eventualmente pueden existir casos (que de hecho, existen) donde sólo se encuentren los tokens de un hijo (como es el caso, según la estructura mostrada, en que se parsea el primer parámetro).

En segundo lugar, si se llegara a implementar esta propuesta, debía proveerse el grado de flexibilidad suficiente para que cada lenguaje pudiese, o bien expresar sus nodos utilizando los tres tipos de regiones ya creados, o bien extendiendo estos casos de regiones para customizarlos según sus necesidades. Ello implicaba desarrollar una arquitectura que permitiera considerar, a futuro, una cantidad de casos particulares (al menos uno por cada nodo), que impedía mantener la simpleza de esta propuesta: no bastaba con manipular regiones de tokens solamente, sino que había propiedades inherentes a cada lenguaje que consideraban casos especiales mucho más sofisticados que esto.

Por ejemplo, tomemos el caso de los modificadores en Java. En PMD, los modificadores son una propiedad de los nodos. Por lo tanto, si tuviese que representarse esta propiedad en la estructura de los nodos utilizando las regiones creadas (*ChildRegion*, *ExtendedChildRegion*, *OwnRegion*), la elección más correcta sería utilizar una *OwnRegion* que abarque dicha región de tokens. Sin embargo, al manipular un modificador, lo que ocurre es que se debe modificar dicha *OwnRegion* de manera interna, es decir, no remover ni agregar una región de este tipo. Es por esta razón, que para representar

regiones de tokens que identifiquen modificadores, se debía crear una región nueva, pero que no se comporte como una región ordinaria (puesto que se debía evitar su inserción/remoción en caso de manipulación, por lo anteriormente explicado), lo que hacía aún menos intuitivo y atractivo el uso de esta propuesta.

Estas dificultades hicieron que el desarrollo de esta propuesta se descarte inmediatamente, por lo que se pasó a elegir la opción de manipular directamente los tokens, que si bien se entiende que puede resultar en un proceso tedioso de implementar, resultaba más viable de desarrollar sin requerir el agregado de tanta lógica nueva sobre el código existente en PMD. Además, como se menciona en la siguiente sección, se trató de generalizar la mayor cantidad de casos de manipulación de tokens para disminuir el costo de implementación de esa propuesta.

6.9. Propuesta 3: Manipulación explícita de tokens

Habiendo descartado la opción de identificar regiones de tokens dentro de un nodo, se procedió a considerar la opción de manipular directamente los tokens de los nodos. Esto es: mantener actualizados los punteros de los tokens de acuerdo a las operaciones de manipulación realizadas sobre los nodos del árbol.

La ventaja que provee este approach por sobre el anterior es que no se debe implementar ninguna arquitectura que permita identificar sectores de los nodos para automatizar su manipulación, sino que lo que se hace es, en principio, hacer que cada nodo sea responsable de actualizar la cadena de tokens en base a las modificaciones que reciba. Decimos en principio, porque lo que intentamos hacer fue generalizar la mayor cantidad de casos posibles, de manera de evitar que se genere duplicación de código en este sentido. Para ello, al comenzar a desarrollar esta propuesta, lo que hicimos fue implementar en la clase *AbstractNode* (que en PMD es la clase de la cual extienden todos los nodos de cada lenguaje) un sistema de traducción de operaciones sobre los hijos de un nodo (que es una propiedad que comparten todos los nodos que extienden de *AbstractNode*) a cambios sobre la cadena de tokens subyacente.

En definitiva, durante marzo se realizó el análisis de todos los casos posibles de manipulación de hijos de un nodo cualquiera para garantizar que la traducción a cambios en la cadena de tokens sea genérica (al menos en lo que respecta a la propiedad "nodos hijo"). Como se dijo, la idea principal detrás

de esto era poder garantizar consistencia y evitar duplicación de código en la mayor cantidad de casos posibles, para que la cantidad de especializaciones a realizar por cada nodo de cada lenguaje sea la mínima, lo que en definitiva haría más tentadora esta propuesta en términos de mantenibilidad y escalabilidad.

Para analizar los casos relacionados a la manipulación (solamente inserción y remoción, dado que el reemplazo se entendía como una remoción seguida de una inserción) de hijos de un nodo, lo que se hizo fue partir de ASTs de 1 nodo (caso nodo raíz), 1 nodo con 1 hijo, 1 nodo con varios hijos, ASTs de 3 niveles y así sucesivamente hasta comprobar que todo el resto de los casos se reducían a realizar inducción estructural. Por cada caso, analizamos cómo debían quedar actualizados los punteros de los tokens de los nodos, y luego de este análisis, procedimos a realizar la implementación de todos estos casos.

Además de proveer una implementación genérica, se dió la posibilidad de adaptar esta implementación según las características particulares de cada nodo. Incluso, dado que con esta propuesta no se forzaba a ningún nodo a realizar modificaciones sobre la cadena de tokens siguiendo una determinada lógica (contrariamente a lo que ocurría en la propuesta anterior - ver sección 6.8), sino que cada propiedad de cada nodo podía ser libremente *sincronizada* en la cadena de tokens, la implementación de las traducciones de manipulaciones de las propiedades a cambios en los tokens era muy flexible y customizable.

Gracias a esta posibilidad de customización tan flexible, implementar, por ejemplo, la traducción de cambios en los modificadores de los nodos del lenguaje Java consistía simplemente en desarrollar la lógica de manejo de tokens correspondiente en una clase común a todos los nodos que tuviesen la propiedad de modificadores (en el caso de Java en PMD, se trata de la clase *AbstractJavaAccessNode*). Para probar la viabilidad de esta solución, se desarrolló esta lógica e incluso se implementó de tal forma de respetar el orden existente de los modificadores en el código fuente, agregando o removiendo sólo aquellos modificadores que realmente fueron manipulados.

Con esto, se tuvo la primera implementación completamente funcional de todo el proyecto, en la cual era posible realizar modificaciones en los parámetros de un método y en los modificadores del mismo (a través de sus nodos) y que estos cambios se tradujeran a cambios en el código de manera correcta. Hasta el momento, era todo un logro.

Sin embargo, aunque la generalización de los casos de traducción de la manipulación de hijos a cambios en la cadena de tokens parecía buena, al comenzar a extender esta propuesta para otros nodos se hizo evidente que la cantidad de especialización necesaria para utilizar esta arquitectura seguía siendo alta, lo que atentaba contra las premisas de mantenibilidad y escalabilidad detalladas en la sección 4.

No conformes con ello, y siendo esta la principal razón, se decidió evolucionar esta opción a una que permitiera realizar dicha traducción de manera de garantizar la mantenibilidad y escalabilidad que ahora no eran suficientes.

6.10. Propuesta 4: Estructuras de Nodos

Con la necesidad de solucionar la problemática de mantenibilidad y escalabilidad de la arquitectura anterior, procedimos al diseño de una nueva propuesta. La idea era generalizar de cierta forma el caso base, para que la particularización que debiera hacerse por cada tipo de nodo especializado fuese mínima, de forma de que la cantidad de código necesario para soportar la traducción de las manipulaciones realizadas sobre el nodo a código fuente fuese la mínima posible.

Teniendo en cuenta estos criterios, se planteó utilizar un sistema similar a la analizada en la herramienta *JavaParser* (ver sección 6.8). La diferencia principal radicaba en la cantidad de elementos que se identificaban como bloques constructores de la *estructura* de un nodo: *Child*, *Children* y *Token*. Para más detalle sobre estos bloques constructores, consultar la sección 6.10.1.

La implementación de esta idea comenzó la primera semana de abril, conjuntamente con el diseño, y luego se continuó dos semanas solamente con el diseño ya que en la primera implementación se evaluó que era factible y fructífero seguir por este camino. Finalmente, luego de definir las interacciones y operaciones a realizar junto con las particularidades a implementar por cada lenguaje, se implementó la propuesta hasta principios de mayo.

La cantidad de código agregado fue superior al de la propuesta anterior; sin embargo, la arquitectura propuesta era más genérica, por lo que funcionaba para un mayor número de casos sin necesidad de demasiada especialización, lo que cumplía con la premisa especificada al comienzo de esta sección.

Además, aprovechando que ahora se poseía una descripción de la estructura de cada uno de los nodos, nos propusimos idear un mecanismo que permitiera crear dinámicamente instancias de nodos a partir de esta información. Y lo logramos. Para ello, fue necesario implementar una nueva estructura, llamada *NodeMetaInfo*, que se detalla en la sección 6.10.3.

En definitiva, esta propuesta presentaba múltiples ventajas por sobre la anterior, no sólo solucionando sus problemas, sino también ofreciendo la posibilidad de realizar las especializaciones que fueran necesarias para adaptarla a las particularidades de cada lenguaje, agregando los tipos de *StructureElement*s que fuesen necesarios, junto al comportamiento que se requiriera en cada caso (siguiendo con el ejemplo de las propuestas anteriores, este sería el caso de los nodos que presentan la propiedad de modificadores en el caso de Java).

Hasta el momento, era la mejor propuesta que habíamos concretado. Sin embargo, y a pesar de sus enormes ventajas, presentaba principalmente un problema: esta arquitectura no soportaba la posibilidad de que un nodo tuviese múltiples formas de representarse, al menos no de manera intuitiva, directa, y que cumpliera con las premisas de mantenibilidad y escalabilidad mencionadas en la sección 4. Esto motivó la búsqueda de una solución que derivó en la siguiente propuesta, detallada en la sección 6.11.

6.10.1. StructureElement

Los elementos estructurales o *StructureElement* (abreviado *SE*) permiten describir la forma o estructura de los diferentes nodos de cada lenguaje, de una manera intuitiva y fácilmente legible, como si se tratara de *bloques* descriptores. Estos elementos permiten identificar regiones de tokens dentro de los nodos que mapean características intrínsecas de ellos en el código fuente.

En el caso particular de esta propuesta, se crearon tres elementos estructurales genéricos que reconocían una sección determinada del nodo.

- *ChildSE*: identifica, dentro de la estructura de un nodo, una región de tokens que corresponde a los tokens de un hijo, del tipo especificado al construir el elemento estructural de este tipo.
- *ChildrenSE*: identifica, dentro de la estructura de un nodo, una región

de tokens que corresponde a los tokens de varios hijos de diferentes tipos (todos especificados al construir el elemento estructural de este tipo), y con un elemento estructural de tipo *TokenSE* que servía como separador de cada uno de los hijos. Este tipo de elemento estructural era sustancialmente distinto al anterior, puesto que se necesitaba manejar el caso del separador y otras particularidades en al lógica de actualización que no se presentaban en el caso del *ChildSE*. La lógica de este elemento estructural era la más compleja y sutil de manejar, y poco generalizable.

- *TokenSE*: identifica, dentro de la estructura de un nodo, una región de un token que no corresponde a ninguno de sus hijos.

Además, cabe destacar la flexibilidad de este diseño al ofrecer la posibilidad de crear nuevos *StructureElement*s, que podrían ser utilizados para identificar características propias de los nodos de un lenguaje dado (por ejemplo, el caso de los modificadores en Java).

La idea de utilizar estos elementos estructurales es la de, por un lado, describir la estructura de los nodos de un lenguaje dado garantizando la legibilidad y facilidad de implementación, y por el otro, ofrecer la posiblidad de realizar una *sincronización* automática y lo más genérica posible de las propiedades de los nodos en el código fuente (detalladas en la sección 6.10.2). Con esto, podemos cumplir además las permisas de mantenibilidad y escalabilidad.

6.10.2. Sincronización

Para realizar la sincronización automática y genérica de las propiedades manipuladas de los nodos en el código fuente, lo que se hizo fue obligar a cada elemento estructural implementado (ver la sección 6.10.1) a saber, en primer lugar, reconocer la estructura original de los nodos, y luego, remapear las características de los nodos en su estructura.

Para el reconocimiento de la estructura, los *StructureElement*s debían implementar el método *scan*. En definitiva, lo que se hacía con esto eran almacenar internamente, previo a cualquier modificación, cuáles eran los **tokens** originales del nodo en cuestión que no correspondían a ningún segmento de tipo *ChildSE* ni *ChildrenSE*. Con esto, lo que se buscaba hacer era tener tokens de referencia en la cadena de tokens completa del código fuente que permitieran, en un futuro, avanzar hasta puntos **obligatorios** o *mandatory*,

es decir, que siempre estuviesen presentes en la respresentación en texto de un nodo, con el objetivo final de poder desplazarse sobre esta cadena al momento de realizar la sincronización de características del nodo.

Para la sincronización de las propiedades de los nodos en la cadena de tokens, los *StructureElement* debían implementar el método *sync*. Con esto, lo que hacía cada elemento estructural era sincronizar la propiedad correspondiente del nodo en la cadena de tokens, en base a la lógica especificada. Si bien la estructura del nodo (*NodeStructure*) estaba encargada de coordinar la lógica de sincronización de cada uno de los elementos estructurales, cada *SE* era capaz de reconocer a los otros elementos estructurales y realizar su sincronización en base al contexto en el que se encontraba. Esta lógica de identificación del contexto hacía que el código creciera en complejidad conforme se comenzaban a considerar nuevos casos, lo que atentaba contra la mantenibilidad y la escalabilidad de la arquitectura.

En la siguiente propuesta (ver sección 6.11), no sólo se actualiza la lógica necesaria para realizar el proceso de sincronización (lo que impacta directamente en las implementaciones requeridas en los *StructureElement*s), sino que además se soluciona, entre otros, el problema de considerar diferentes casos según el contexto en que se encontrara cada *SE* (es decir, la sincronización pasa a depender exclusivamente del elemento estructural, independientemente de los elementos que lo rodeen).

6.10.3. NodeMetaInfo

Con esta subarquitectura es posible instanciar dinámicamente nodos a partir de la información/descripción de su estructura. Dado que con los *StructureElement*s ya se tiene una noción de los tokens que componen a un nodo, lo único que faltaba era la información necesaria para instanciar el nodo adecuado de un lenguaje particular, de manera genérica e independiente de dicho lenguaje.

La idea consiste esencialmente en crear una clase que extiende de otra llamada *NodesMetaInfo* que contiene un mapa donde cada clave es una clase de nodo de un lenguaje dado y el valor la información necesaria para instanciar ese tipo de nodo (de ahí que el nombre de la clase que almacena esta información sea *NodeMetaInfo*). La clase *NodeMetaInfo* implementa el método *newNode*, que utiliza directamente la clase y el tipo (*kind*) de nodo adecuado para realizar la instanciación a través de *reflection* (ver fragmento

de código 3).

```
public class NodeMetaInfo<T extends AbstractNode> {
  private final Class<? extends T> clazz;
  private final int kind;
  // ...
  public final <C extends Node> T newNode() {
    try {
      return clazz.getDeclaredConstructor(Integer.TYPE).newInstance(kind);
    } catch (InstantiationException | IllegalAccessException |
    ↪  NoSuchMethodException | InvocationTargetException e) {
      LOGGER.warn("newNode - Could not create instance of type {}.
    ↪  Returning null...", clazz);
      return null;
    }
  }
}
```

Código 3: Método *NodeMetaInfo.newNode*

Para utilizar este mecanismo de creación de nodos, cada lenguaje debe extender la clase *NodesMetaInfo* implementando los métodos específicos acorde al lenguaje.

Analicemos el método de inicialización del mapa con la metainformación de todos los nodos de un lenguaje, que se muestra en el fragmento de código 4.

Los métodos *getAllClassesNames*, *getNodePrefix* y *getNodePackage* son métodos abstractos a definir por la clase concreta del lenguaje a implementar. Luego, lo que se hace es, por cada uno de las clases de los nodos de dicho lenguaje, obtener su nombre para luego buscar la clase correspondiente utilizando *reflection*. Una vez que se encuentra la clase, se guarda en el mapa junto con la metainformación correspondiente (que consiste de esa clase y del tipo de nodo, pero que puede ser fácilmente extensible para contener cualquier otra información necesaria).

Notar que esta implementacion de *initializeNodeMetaInfoPerName* está pensada para lenguajes que utilizan como generador de parser a *JavaCC*, permitiendo aprovechar todo el *stuff* generado por esta herramienta. De hecho, en *PMD*, con excepción de Apex, el resto de los lenguajes que implementan detección de violaciones a través de reglas utilizan *JavaCC*.

En caso de que no se utilice *JavaCC* para un lenguaje dado, o se quiera

```java
public abstract class NodesMetaInfo<T extends AbstractNode> {
  protected void initializeNodeMetaInfoPerName(final Map<Class<? extends
   ↪ T>, NodeMetaInfo<? extends T>> nodeMetaInfoPerName) {
    final String[] namesArray = getAllClassesNames();
    final String nodePrefix = getNodePrefix();
    final String nodePackage = getNodePackage();
    for (int nodeKind = 0; nodeKind < namesArray.length; nodeKind++) {
      final String baseName = namesArray[nodeKind];
      final String className = nodePackage + "." + nodePrefix + baseName;
      final Class<? extends T> clazz;
      try {
        clazz = Class.forName(className).asSubclass(baseNodeClazz);
      } catch (final ClassNotFoundException e) {
        handleClassNotFound(e, baseName);
        continue;
      }
      nodeMetaInfoPerName.put(clazz, new NodeMetaInfo<>(clazz, nodeKind));
    }
  }
}
```

Código 4: Método *NodesMetaInfo.initializeNodeMetaInfoPerName*

customizar la lógica de inicialización del mapa, el método *initializeNodeMetaInfoPerName* puede sobreescribirse para implementar la lógica necesaria de acuerdo a las necesidades específicas del lenguaje.

Veamos por ejemplo la extensión de la clase *NodesMetaInfo* especializada para Java. Notar que para este lenguaje se que utiliza *JavaCC*, por lo que en este caso basta definir los métodos abstractos de la clase padre. El fragmento de código 5 muestra esta especialización, llamada *JavaNodesMetaInfo*. Cabe remarcar lo sencillo que resulta especializar esta subarquitectura estos casos.

Una vez definida esta clase, lo único que resta por hacer es utilizarla en el constructor de la clase donde se definen las estructuras de los nodos del lenguaje correspondiente: en el caso de esta propuesta, se trataba de la clase extendida de *AbstractNodesStructure* y especializada para el lenguaje en cuestión; en el caso de la propuesta siguiente (ver sección 6.11), se trata de la clase extendida de *AbstractNodesSyntax* (ver sección 6.11.5 para más detalles sobre esta clase).

```
public class JavaNodesMetaInfo extends NodesMetaInfo<AbstractJavaNode> {
  // Picked from Java.jjt
  private static final String NODE_PACKAGE =
↪  "net.sourceforge.pmd.lang.java.ast";
  private static final String NODE_PREFIX = "AST";
  // ...
  protected String[] getAllClassesNames() {
    return JavaParserTreeConstants.jjtNodeName;
  }

  protected String getNodePackage() {
    return NODE_PACKAGE;
  }

  protected String getNodePrefix() {
    return NODE_PREFIX;
  }
}
```

Código 5: Clase *JavaNodesMetaInfo*

6.11. Propuesta 5 (final): Estructuras RegExp

El objetivo principal que motivó esta nueva arquitectura fue el hecho de que, como se marcó, la anterior no contemplaba los casos en que un nodo puede tener múltiples representaciones.

Tomemos el caso de los nodos *LambdaExpression* de Java:

```
void LambdaExpression() :
{ checkForBadLambdaUsage(); }
{
  VariableDeclaratorId() "->" ( Expression() | Block() )
| LOOKAHEAD(3) FormalParameters() "->" ( Expression() | Block() )
| LOOKAHEAD(3) "(" VariableDeclaratorId() ( "," VariableDeclaratorId()
↪  )* ")" "->" ( Expression() | Block() )
}
```

Código 6: Definición del nodo *LambdaExpression* de Java en la gramática de JavaCC

Notemos que este tipo de nodos puede escribirse de tres formas distintas, en este caso dependiendo específicamente del tipo de nodos que tenga como

hijos. Incluso, notemos que, en caso de que un nodo *LambdaExpression* tenga un único hijo de tipo *VariableDeclaratorId*, este hijo puede estar rodeado o no por tokens de tipo paréntesis (dependiendo de si se trate del tercer o del primer caso mostrados en el fragmento 6, respectivamente).

Con esto en mente, la propuesta fue encontrar una arquitectura que mantuviera las bondades de la anterior, pero que permitiera además solucionar el problema de la correcta selección de la estructura de los nodos.

La primera pregunta que podría surgir es: ¿En base a qué información debe seleccionarse la forma correcta de escribir los nodos? Dado que la manipulación de los nodos se da, al menos de manera abstracta y común a todos los nodos, a través de los hijos, se entiende que esta es una (sino la única) manera de poder realizar dicha selección.

Dicho esto, la arquitectura a desarrollar debería permitir:

- Determinar la representación en tokens de un nodo a partir de los hijos que posee, de manera independiente al lenguaje del que provenga el nodo.
- Especializar los elementos usados por esta arquitectura, para que la representación en tokens de un nodo sea modificada en base a los cambios que se realicen sobre otras propiedades suyas. Por ejemplo, en el caso de los nodos de Java que así lo permitan, si se incrementara la profundidad del array subyacente, deberían agregarse los tokens correspondientes a esta modificación (es decir, un par de corchetes).
- Cumplir con las premisas de trabajo especificadas en la sección 4.

6.11.1. Introducción

Antes de presentar la propuesta con algún nivel de detalle, vamos a especificar brevemente de dónde surgió la idea de la arquitectura, y mencionar la forma en que se irán presentando los conceptos en cada una de las subsecciones.

Comencemos por ver su origen. Lo que necesitamos hacer es construir una arquitectura que de alguna forma elija o *construya* la estructura correspondiente a un nodo en base a sus propiedades (que pueden ser cualesquiera).

En la búsqueda de un sistema que haga algo similar a esto, nos topamos con el proceso biológico de transcripción del ADN [1], que funciona de la siguiente forma: una proteína se posiciona sobre una secuencia promotora, y junto con la ayuda de otras proteínas, comienza a leer cada base de una de las dos tiras del ADN, generando otra tira con las bases complementarias que va leyendo, hasta llegar a una secuencia de parada, momento en el cual libera la tira que fue generada.

Lo que tratamos de hacer fue imitar este proceso: dado un nodo, desde su primer token (secuencia promotora dentro de toda la cadena de tokens del programa) hasta su último token (secuencia de parada), lo que se hace es generar una estructura a partir de su información. A este proceso lo llamamos transcripción.

En nuestro caso, el nodo actual es el ADN y las propiedades del nodo leídas de una forma específica son las bases (según el proceso de transcripción biológico). Esto último, en la arquitectura implementada reciben el nombre de *ParsedValue*s).

La lectura está a cargo de un parser que utiliza subyacentemente un NFA (autómata finito no determinístico) construído o *compilado* a partir de una descripción de la *sintaxis* del nodo en cuestión. El equivalente biológico del parser es la proteína que realiza la lectura de una de las tiras del ADN.

¿Con qué elementos se genera la estructura de la transcripción? En lugar de utilizar las bases complementarias biológicas, lo que se usan son los *StructureElement*s pero *wrappeados* junto con otra información en una clase denominada *SEInfo*.

Una vez que se tiene la estructura correspondiente a un nodo en base a sus propiedades, lo que se hace es *sincronizar* la información de los nodos en dicha estructura, utilizando también como guía la estructura anterior del nodo, que tiene su información previa a cualquier manipulación. Durante esta sincronización, se actualiza o *sincroniza* la cadena de tokens asociada al nodo para reflejar sus propiedades actualizadas, cuidando de modificar solamente las regiones de tokens correspondientes a propiedades manipuladas.

Implementar este mecanismo de transcripción requirió el desarrollo de múltiples elementos. Para poder introducir los conceptos y elementos desarrollados de manera progresiva con el objetivo de terminar teniendo un conocimiento profundo de la arquitectura diseñada, se optó por realizar una explicación por secciones del estilo *BFS*, donde en cada sección se introdu-

cen varios conceptos de la arquitectura pero sólo se detallan los primordiales para lograr explicar el objetivo de dicha sección, referenciando las otras secciones donde se explican con detalle el resto de los conceptos introducidos pero no profundizados. Si bien las secciones pueden abordarse en el orden que se desee, se recomienda al lector seguir el orden natural de lectura.

La organización de lo que sigue es la siguiente.

- En la sección 6.11.2 se presenta el mecanismo que hace posible el funcionamiento de la arquitectura diseñada. Además, se presentan algunas clases y conceptos que se irán desarrollando con más detalle en las siguiente secciones.
- En la sección 6.11.3 se muestra el uso y funcionamiento general de la arquitectura diseñada para el caso de la manipulación de nodos y su correspondiente traducción a cambios en la cadena de tokens, lo que permite obtener un overview de múltiples elementos de la arquitectura y su interacción en sí (esto último hace que esta sección sea de suma importancia para el entendimiento de las secciones siguientes).
- En la sección 6.11.4 se muestra el uso y funcionamiento general de la arquuitectura diseñada para el caso de la creación de nodos.
- En la sección 6.11.5 se explican con detalle los elementos que permiten describir la sintaxis de los nodos de cada lenguaje, que luego serán utilizados para construir la maquinaria involucrada en el proceso de transcripción, es decir, el NFA que utilizará el parser, y que finalmente se utilizarán para generar las estructuras de los nodos en base a su información actual.
- En la sección 6.11.6 se explica el mecanismo de conversión o *compilación* de la sintaxis para construir el NFA que utilizará el parser.
- En la sección 6.11.7 se explica detalladamente cómo se lleva a cabo el proceso de transcripción que permite generar la estructura correspondiente a un nodo en base a su información actual.
- En la sección 6.11.8 se explica cómo, una vez elegida la estructura correspondiente a un nodo, se vincula su información en dicha estructura y en la cadena de tokens, de manera de garantizar que los cambios a realizar sobre el código fuente sean locales a las propiedades manipuladas. Notar que este paso es el propiamente dicho de *traducción* de las manipulaciones de los nodos a cambios en la cadena de tokens.

- En la sección 6.11.9 se explican los métodos genéricos de manipulación de hijos de nodos que quedaron implementados en la arquitectura propuesta.
- En la sección 6.11.10 se listan los pasos necesarios para utilizar la arquitectura propuesta en un nuevo lenguaje, referenciando las secciones donde se explican con mayor detalle los elementos involucrados.
- En la sección 6.11.11 se listan los pasos necesarios a considerar para implementar nuevos elementos estructurales, que son los que van a permitir representar las particularidades de los nodos de los nuevos lenguajes donde se desee utilizar la arquitectura propuesta.

6.11.2. Presentación de la idea

Para cumplir con lo especificado previamente, se diseñó e implementó un mecanismo que permite construir la estructura correspondiente a un nodo (clase *Structure*) en base a su información actual. La estructura, al igual que como se especificó en la propuesta anterior (ver sección 6.10.1), está compuesta de elementos estructurales (*StructureElement*), pero esta vez *wrappeados* con información extra en una clase denominada *SEInfo*, que además permite saber: 1) si el elemento estructural actual es o no obligatorio; 2) la región de tokens del nodo que le corresponde.

Este mecanismo consiste esencialmente en un parser que, mediante distintas políticas de parseo, puede ir construyendo la estructura correspondiente a un nodo conforme va consumiendo su información, la cual es provista en *unidades de lectura* (que pueden variar según la política mencionada).

Una vez que el parser determina cuál es la nueva estructura del nodo, lo que se hace es **sincronizar** la información actualizada del nodo en esta nueva estructura, utilizando también como referencia la información almacenada en la estructura vieja del nodo. Finalmente, se reemplaza la estructura vieja por la nueva (que también contiene la nueva información).

Esta política de sincronización utilizando la estructura vieja y remapeando la información del nodo a la estructura nueva permite que el mapeo de las modificaciones que se realizan sobre las propiedades de los nodos o sobre el AST sean traducidas a tokens de manera diferida y sin necesidad de almacenar cada una de las operaciones realizadas. Esto ayuda a poder trabajar sobre el código sin necesidad de tener que operar sobre la cadena de tokens hasta

que la sincronización sea explícitamente solicitada. Además, dado que la sincronización no necesita realizarse cada vez que se hace una modificación, la carga del contexto de manejo de tokens para su correspondiente actualización es mucho más aprovechada, puesto que, en el mejor caso, esta arquitectura permite que dicha carga sea realizada una única vez por nodo modificado (independientemente de la cantidad de modificaciones que el mismo haya sufrido).

Symbol De manera subyacente, para lograr su objetivo el parser utiliza un tipo de autómata finito no determinístico (NFA) que es capaz de identificar qué elementos estructurales componen la estructura actual de un nodo e ir creando las mencionadas estructuras.

Recordemos que un NFA está representado formalmente por una tupla de 5 elementos, que son:

- Un conjunto finito de estados Q
- Un conjunto finito de símbolos de entrada Σ
- Una función de transición $\Delta : Q \times \Sigma \rightarrow (Q)$
- Un estado inicial $q_0 \in Q$
- Un conjunto de estados finales F, tales que $F \subseteq Q$

Dado que cada nodo de cada lenguaje debe tener su propio NFA para reconocer su estructura, es vital proveer una forma sencilla de especificar cuáles son los elementos del NFA de cada uno de ellos. Aprovechando la equivalencia existente entre los NFAs y las expresiones regulares, y dada la facilidad expresiva que ofrecen estas últimas, se decidió utilizar expresiones regulares para luego convertirlas o **compilarlas** en los NFAs equivalentes.

Debido a que la lógica de parseo a realizar en los NFAs debe ser customizable y varía de acuerdo al tipo de parseo a realizar, en lugar de utilizar los operadores tradicionales de las expresiones regulares (que son en cierta forma estáticos, ya que siempre poseen el mismo comportamiento al parsear un input y pueden parsear sólo un conjunto de símbolos de entrada), se crearon **Símbolos** con un poder de expresión y adaptación superior.

Los **Símbolos** se dividen en dos subtipos: **Operadores** y **Operandos**. Los símbolos de operadores pueden componerse con cualquier otro símbolo, mientras que los operandos son símbolos finales.

En principio, la distinción entre estos dos subtipos de símbolos es meramente una cuestión de etiquetas que podrían permitir, en un futuro, asegurar diferentes comportamientos a extensiones que se hagan del parser o del proceso de compilación de las expresiones regulares al NFA. Sin embargo, es oportuno realizar la distición mencionada en el párrafo anterior ya que permitirá entender para qué se pretende utilizar cada tipo de símbolo.

En particular, se diseñaron e implementaron los siguientes símbolos operadores (a su lado se detallan los caracteres correspondientes a las expresiones regulares ordinarias):

- AlternationSym (|)
- ConcatenationSym (implícito cuando se escriben dos símbolos consecutivos)
- ZeroOrOneSym (?)
- ZeroOrMoreSym (∗)
- OneOrMoreSym (+)

Por otra parte, los símbolos operandos son los elementos estructurales *ChildSE* y *TokenSE* detallados en la propuesta anterior (ver sección 6.10.1), a los que se les agregó la lógica de comportamiento necesaria para poder ser utilizados en la nueva arquitectura (esta lógica se detallará más adelante).
Notar que el elemento estructural *ChildrenSE* ya no es necesario, puesto que con el poder expresivo de esta nueva propuesta un *ChildrenSE* es equivalente a requerir que existan cero o más elementos de tipo *Child* con algún token como separador luego del primer *ChildSE*.

Antes de continuar, es importante aclarar que para poder lograr la traducción o compilación de la expresión regular escrita mediante estos símbolos al NFA equivalente, fue necesario hacer que cada símbolo tenga la capacidad de traducirse en un fragmento del NFA final, lo que involucra no sólo traducirse a sí mismo, sino también saber, en caso de que se trate de un símbolo operador, cómo vincular correctamente cada uno de los fragmentos NFA correspondientes a los símbolos que contiene.

La traducción de cada símbolo operador a un fragmento NFA se detalla en la sección 6.11.6.

NodeSyntax Con los cambios mencionados recientemente, las estructuras de los nodos (*NodeStructure*) propuestas en la sección 6.10 se tranforman en descriptores de la sintaxis de los nodos, por lo que reciben el nombre de *NodeSyntax*. Por lo tanto, los *NodeSyntax* son las expresiones regulares (RegExp) que describen la sintaxis de los tipos de nodos mediante la utilización de los nuevos símbolos implementados.

Retomando el caso de los nodos de tipo *LambdaExpression* de Java, su *NodeSyntax* se define como se muestra en el fragmento de código 7, donde los métodos *child*, *seq*, *zeroOrMore* y *or* son métodos que sirven para simplificar la lectura de los constructores de cada uno de los símbolos mencionados anteriormente, y que se definen según lo mostrado en el fragmento de código 8.

```
1  define(ASTLambdaExpression.class,
2    or(
3      child(ASTVariableDeclaratorId.class),
4      child(ASTFormalParameters.class),
5      seq(LPAREN,
6        child(ASTVariableDeclaratorId.class),
7        zeroOrMore(COMMA, child(ASTVariableDeclaratorId.class)),
8        RPAREN
9      )
10   ),
11   LAMBDA,
12   or(child(ASTExpression.class), child(ASTBlock.class))
13 );
```

Código 7: Sintaxis del nodo *LambdaExpression* de Java utilizando los nuevos símbolos

Notemos que los fragmentos de código 6 y 7 son muy similares. Esto es una gran ventaja, no sólo en cuanto a lo que implica la mantenibilidad del código sino también la facilidad con la que se pueden definir las sintaxis de los nodos utilizando los nuevos símbolos, puesto que en principio basta con observar las definiciones realizadas en JavaCC y reescribirlas con los símbolos correspondientes.

Una vez definidas las sintaxis, las mismas no son convertidas o **com-**

```
public abstract class AbstractNodesSyntax<T extends AbstractNode>
  ↪ implements NodesSyntax<T> {
  // ...
  protected final ChildSE child(final Class<? extends T> validChildClass)
  ↪ {
    return ChildSE.newInstance(validChildClass);
  }

  protected ZeroOrOneSym zeroOrOne(final Symbol... symbols) {
    return ZeroOrOneSym.newInstance(symbols);
  }

  protected ZeroOrMoreSym zeroOrMore(final Symbol... symbols) {
    return ZeroOrMoreSym.newInstance(symbols);
  }

  protected OneOrMoreSym oneOrMore(final Symbol... symbols) {
    return OneOrMoreSym.newInstance(symbols);
  }

  protected AlternationSym or(final Symbol... symbols) {
    return AlternationSym.newInstance(symbols);
  }

  protected ConcatenationSym seq(final Symbol... symbols) {
    return ConcatenationSym.newInstance(symbols);
  }
  // ...
}
```

Código 8: Métodos para simplificar la lectura y el uso de los símbolos operandos

piladas de expresiones regulares a NFAs sino hasta que se realiza alguna modificación sobre el tipo de nodo al que representan. Es decir, siguiendo con el ejemplo, hasta que no se realiza alguna modificación sobre algún nodo de tipo *ASTLambdaExpression*, el NFA equivalente a la expresión regular que representa la sintaxis de este tipo de nodo no es generado.

Es importante destacar que al realizar una única traducción $NodeSyntax \rightarrow NFA$ por cada tipo de nodo, y sólo en caso de que se modifique algún nodo de ese tipo, se logra una mejora en la cantidad de memoria y procesamiento requerida para llevar a cabo la modificación de un nodo, en particular en lo que concierne a sucesivas modificaciones sobre un nodo ya modificado y también a nodos que no fueron modificados aún (dado que estos últimos no

están consumiendo memoria almacenando información adicional).

Además, una cuestión de vital utilidad es que, dado que la traducción a NFA se realiza sólo si se modifica un nodo de ese tipo, no es necesario tener definidas las sintaxis de todos los nodos de un lenguaje, sino sólo de aquellos que se deseen manipular. Por ejemplo, para definir la sintaxis de los nodos *LambdaExpression* de Java (ver fragmento de código 7) no es necesario definir la sintaxis de ninguno de los nodos allí mencionados. Esto hace por demás atractiva la arquitectura propuesta, sobre todo en lo que conscierne a su adopción por parte de nuevos lenguajes.

6.11.3. Flujo de manipulación

Para mostrar un flujo completo de manipulación, que involucra esencialmente modificar un nodo y que esta modificación se traduzca en los cambios correspondientes sobre el código fuente subyacente (es decir, para que los cambios sobre los nodos se mapeen en los cambios sobre la cadena de tokens), analizaremos un ejemplo concreto. Para continuar con el mismo curso, trabajaremos con un nodo de tipo *LambdaExpression* de Java.

Supongamos que en algún lugar del código fuente se tiene la siguiente expresión lambda:

```
1 (a,Sample.this,c) -> { ; }
```

Código 9: Expresión lambda en el código fuente.

Esta expresión es parseada a un nodo de tipo *ASTLambdaExpression* por PMD (que por detrás utiliza *JavaCC* [11] para este propósito). La jerarquía de nodos que se produce a partir de este parseo es la que se muestra en el código 10.

Una vez obtenida la instancia de nodo (supongamos que en una variable de nombre *lambdaExpression*), se puede realizar cualquiera de las modificaciones explicadas en la sección 6.10.

En particular, supongamos que lo que queremos hacer es removerle el

[11] https://javacc.org/

```
>LambdaExpression
> VariableDeclaratorId
> VariableDeclaratorId
> VariableDeclaratorId
> Block
>  BlockStatement
>   Statement
>    EmptyStatement
```

Código 10: Jerarquía de nodos a partir del parseo de la expresión lambda mostrada en el fragmento de código 9

primer hijo al *LambdaExpression*; es decir, según el código 9, vamos a remover el *VariableDeclaratorId*: *a*.

Para ello, lo que hacemos es ejecutar la sentencia:

```
lambdaExpression.removeChild(0);
```

Internamente, y de manera común para todos los nodos, previo a realizar cualquier modificación sobre los hijos de un nodo (ya sea inserción, remoción o reemplazo), se ejecuta el método *AbstractNode.syncRequired* (ver código 11):

Notar que lo que hace este método, desde la perspectiva del nodo que lo ejecuta, es básicamente:

- Marcar que necesito sincronizarme y que necesito revisar a mis hijos, por si ellos tienen cambios.
- Almacenar mi estructura, si es que no tengo ninguna.
- Marcar toda la rama del AST desde la raíz hasta mí para que sepan que tengo un cambio.

Dado que se trata de la primera modificación que se realiza sobre **esta instancia de nodo** *LambdaExpression*, el campo *structure* es *null*, por lo que se ejecuta la línea 10 del fragmento de código 11, donde lo que se hace es: 1) obtener la sintaxis para el tipo de nodo correspondiente (en este caso, *LambdaExpression*); 2) pedirle a dicha sintaxis que escanee la instancia de nodo actual.

```java
public abstract class AbstractNode implements Node {
  // ...
  public final void syncRequired() {
    if (selfSyncRequired && childrenSyncRequired) {
        return;
    }

    // First time scan if needed.
    if (structure == null) {
        structure = getNodeSyntax().scan(this);
    }

    // Mark me as needing sync.
    selfSyncRequired = true;
    // My children may need sync too.
    childrenSyncRequired = true;

    // Mark all my branch to inform that I need to be synced.
    Node currAncestor = parent;
    while (currAncestor != null) {
        AbstractNode aCurrAncestor = requireAbstractNode(currAncestor);
        if (aCurrAncestor.childrenSyncRequired) {
            return;
        }
        aCurrAncestor.childrenSyncRequired = true;
        currAncestor = currAncestor.jjtGetParent();
    }
  }
  // ...
}
```

Código 11: Método *AbstractNode.syncRequired*

El método *scan* (ver código 12) le pide al parser correspondiente que parsee **completamente** la instancia de nodo dada para obtener la estructura correspondiente.

```
1  public class NodeSyntaxImpl<T extends AbstractNode> implements
    ↪  NodeSyntax<T> {
2    // ...
3    public Structure scan(final T node) {
4      return ensureParserPresent().parseComplete(node);
5    }
6    // ...
7  }
```

Código 12: Método *NodeSyntaxImpl.scan*

Algunas preguntas que pueden surgir de esto son:

1. ¿Cuál es el parser correspondiente a la sintaxis de este nodo? ¿Cómo se generó?
   La respuesta a esta pregunta se encuentra en la sección 6.11.6.

2. ¿Qué significa parsear **completamente** un nodo? ¿Qué otras formas de parsear un nodo existen?
   La respuesta a esta pregunta se encuentra en la sección 6.11.7.

Una vez que la estructura fue generada por primera vez, lo único que resta es completar la operación de manipulación solicitada, que en el caso que estamos analizando es una operación de remoción. El detalle de cómo funciona esta operación (y toda otra que sea propia de manipulación de hijos de un nodo) se explica en la sección 6.11.9.

Notemos que hasta este momento no hubo ningún cambio generado en la cadena de tokens que reflejen los cambios realizados sobre las características del nodo, puesto que lo único que hicimos fue:

1. Cargar la estructura inicial (es decir, la original) del nodo sobre el cual se está realizando la manipulación

2. Realizar la remoción propiamente dicha

Habiendo ya realizado la remoción, procedamos a imprimir el nodo *LambdaExpression*. Para reflejar estos cambios en la cadena de tokens, previo a la

utilización de la cadena, se debe *sincronizar* la nueva información del nodo en la cadena de tokens usando como guía la estructura e información anterior del nodo, tal como se especificó superficialmente en la sección 6.11.2.

Es por este motivo que en el método *AbstractNode.print*, previo a la impresión de los tokens, se realiza una llamada al método *AbstractNode.syncIfRequired* (ver fragmento de código 13).

```
1  public abstract class AbstractNode implements Node {
2    // ...
3    public void print() {
4      syncIfRequired();
5      System.out.println(GenericTokens.stringify(firstToken, lastToken));
6    }
7    // ...
8  }
```

Código 13: Método *AbstractNode.print*

Básicamente, lo que se hace en el método *AbstractNode.syncIfRequired* (ver fragmento de código 14), desde el punto de vista del nodo, es:

1. Revisar si alguno de mis hijos necesita ser sincronizado, si así fue configurado. Una vez terminado, marcar que ya finalicé.
2. Sincronizar mis tokens, en caso de que así sea necesario, para que reflejen mis propiedades, utilizando para ello mi sintaxis.

Notar que la sincronización se realiza solamente en caso de que se haya marcado como necesaria, factor que también contribuye a la eficiencia de la arquitectura implementada.

En el ejemplo que estamos analizando, tanto *childrenSyncRequired* como *selfSyncRequired* son verdaderos debido a la ejecución el método *AbstractNode.syncRequired* (código 11) tal como se mencionó previamente. Dado que los hijos no recibieron ninguna modificación en nuestro ejemplo, podemos obviar la primera parte del método *AbstractNode.syncIfRequired* y concentrarnos directamente en la sincronización del nodo *LambdaExpression* (líneas 300-303).

Lo que se hace entonces es obtener la sintaxis correspondiente a los nodos de tipo *LambdaExpression* para con ella realizar la sincronización correspon-

```java
public abstract class AbstractNode implements Node {
  // ...
  private void syncIfRequired() {
    if (childrenSyncRequired) {
      if (children != null) {
        for (Node child : children) {
          requireAbstractNode(child).syncIfRequired();
        }
        childrenSyncRequired = false;
      }
    }

    if (selfSyncRequired) {
      structure = getNodeSyntax().sync(this);
      selfSyncRequired = false;
    }
  }
  // ...
}
```

Código 14: Método *AbstractNode.syncIfRequired*

diente, actualizando finalmente la estructura del nodo con la retornada luego de la sincronización.

Analizando el método *NodeSyntaxImpl.sync* (ver fragmento de código 15), podemos ver que lo que se hace es:

1. Parsear **los hijos** del nodo utilizando el mismo parser que el utilizado al ejecutar el método *NodeSyntaxImpl.scan* (ver código 12). Notar, sin embargo, que el parseo es diferente al del *scan*, dado que en aquella ocasión se había realizado un parseo **completo** del nodo.

2. Seleccionar una única nueva estructura de entre todas las estructuras posibles.

3. Sincronizar la información del nodo utilizando la nueva estructura seleccionada.

4. Retornar esta nueva estructura.

Antes de continuar, cabe aclarar que:

```java
public class NodeSyntaxImpl<T extends AbstractNode> implements
    NodeSyntax<T> {
    // ...
    public Structure sync(final T node) {
        final Set<Structure> possibleNewStructures =
    ensureParserPresent().parseChildren(node);
        final Structure newStructure =
    pickNewStructure(node.getStructure(), possibleNewStructures);
        newStructure.sync(node);
        return newStructure;
    }
    // ...
}
```

Código 15: Método *NodeSyntaxImpl.sync*

- En la sección 6.11.7 se explica con mayor detalle el funcionamiento del parser.
- En la sección 6.11.8 se explica con mayor detalle el funcionamiento del mecanismo de sincronización.
- La selección de la nueva estructura entre todas las posibles actualmente funciona de la siguiente manera:
  - Si la estructura anterior está entre las candidatas como la nueva estructura, se elige esta misma estructura.
  - Sino, se elige la estructura candidata con menor longitud (aquella que esté compuesta por la menor cantidad de elementos estructurales). En caso de haber dos con la menor longitud, se elige alguna de ellas de manera arbitraria.

  Es importante destacar que esta política de decisión es fácilmente adaptable a cualquier otra que se considere pertinente en el futuro.

Con todo esto, lo que se hizo fue sincronizar las propiedades del nodo *LambdaExpression* en la cadena de tokens para reflejar sus propiedades actuales, realizando los cambios mínimos necesarios para ello, y se eligió la nueva estructura del nodo, que contiene su información actualizada.

Finalmente, la cadena de tokens que se imprime en pantalla es la siguiente:

```
1         (Sample.this,c) -> { ; }
```

Código 16: Expresión lambda luego de remover el primer hijo.

En resumen, los pasos realizados durante un flujo de manipulación son los que se muestran en la figura 3. Notar el uso de *manipulationMethod* como nombre de método de manipulación genérico. Notar también el uso de *NodeClass* en lugar de *AbstractNode* para denotar que el flujo puede ser implementado en cualquier tipo de nodo. En particular, este flujo así como se muestra, está implementado para los métodos de manipulación: *insertChild*, *setChild*, *removeChild* y *remove* de la clase *AbstractNode*.

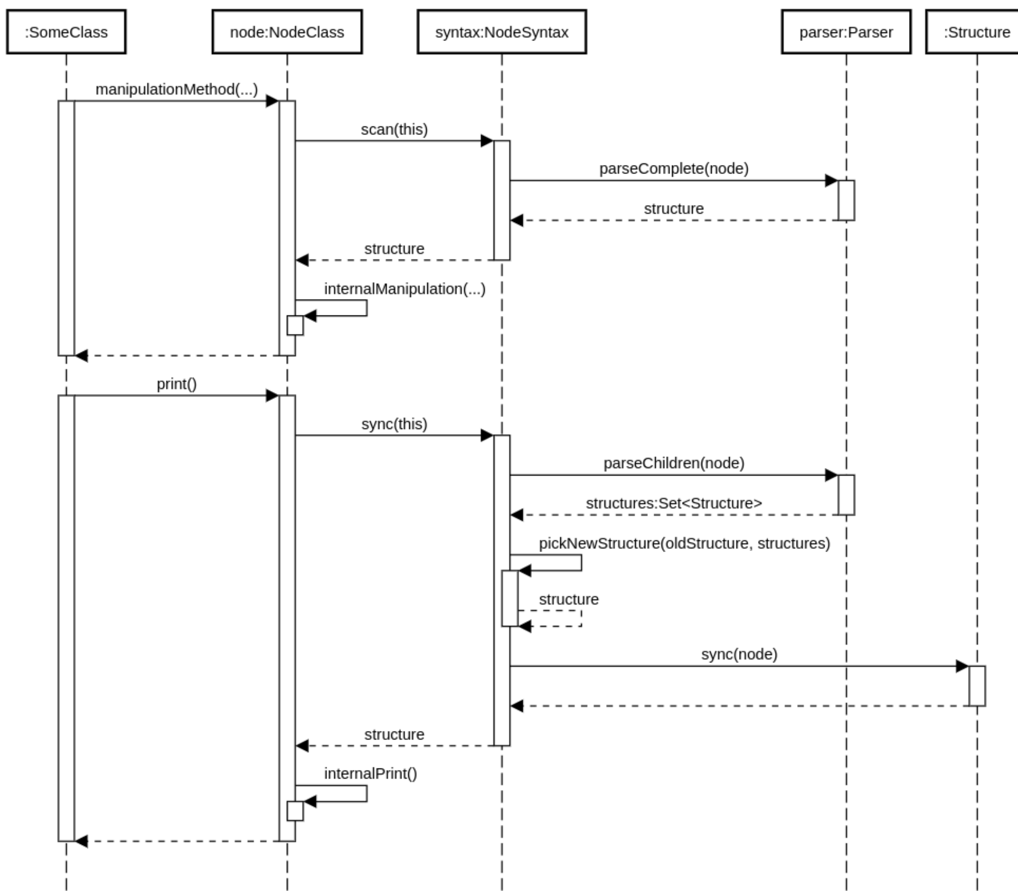

Figura 3: Diagrama de secuencia de un flujo de manipulación genérico.

6.11.4. Flujo de creación

La arquitectura diseñada, al igual que la de la sección 6.10, soporta la posibilidad de crear nodos dinámicamente. Esto se logra aprovechando la sintaxis definida de cada tipo de nodo, utilizando de manera transparente el NFA compilado que almacena el parser correspondiente.

A manera descriptiva, es oportuno mencionar que el parser en este caso no necesita *parsear* un nodo para generar su estructura correspondiente, sino que basta con recorrer el NFA subyacente para que genere todas las estructuras del nodo posibles en base a sus tokens. Es decir, se recorre el NFA generando todas las estructuras compuestas exclusivamente de tokens o características propias del nodo, que no dependen de los hijos. La explicación detallada del funcionamiento de este mecanismo se presenta en la sección 6.11.7.

Para analizar un ejemplo práctico de este flujo, supongamos que tenemos nuevamente la expresión lambda del fragmento de código 9. Supongamos que lo que queremos hacer es remover todos los parámetros de la expresión lambda. Notemos que según la gramática de los nodos *LambdaExpression* definida en JavaCC (ver fragmento 6), que coincide con la sintaxis definida utilizando los nuevos símbolos (ver fragmento 7), para que no haya parámetros, estos nodos deben tener como primer hijo un nodo de tipo *FormalParameters* sin ningún hijo de tipo *FormalParameter*.

Para lograr esto, lo que se debe hacer es:

1. Remover todos los hijos de tipo *VariableDeclaratorId* del nodo *LambdaExpression* (llamémoslo *lambdaExpression*)
2. Crear un nodo de tipo *FormalParameters* (llamémoslo *newFPs*)
3. Insertar el nodo *newFPs* como primer hijo de *lambdaExpression*

Estos sencillos pasos son igual de fáciles de realizar con la arquitectura propuesta. El código 17 muestra cómo realizarlos utilizando el mecanismo propuesto e implementado.

Luego de llamar al método del fragmento de código 17, basta con llamar al método *print* del nodo *lambdaExpression* (ver fragmento de código 13 para que se realice la sincronización de los cambios realizados, es decir, para que se remuevan los tokens correspondientes a los 3 hijos removidos (y los tokens

```java
private static void removeAllVDIDs(final ASTLambdaExpression
    lambdaExpression) {
  // Step 1
  lambdaExpression.removeChild(2);
  lambdaExpression.removeChild(1);
  lambdaExpression.removeChild(0);
  // Step 2
  final ASTFormalParameters newFPs = JavaNodesSyntax.getInstance()
    .getNodeSyntax(ASTFormalParameters.class).newInstance();
  // Step 3
  lambdaExpression.addChild(0, newFPs);
}
```

Código 17: Expresión lambda luego de remover el primer hijo.

extra que existían por ellos, que son las comas que los separaban). Con esto, la cadena de tokens que se imprime en pantalla es la siguiente:

```
() -> { ; }
```

Código 18: Expresión lambda luego de remover todos los hijos de tipo VariableDeclaratorId e insertar un FormalParameters.

Analicemos lo que se hace en la líneas 7 y 8 del fragmento de código 17.

1. Se obtiene la instancia donde se define la sintaxis de todos los nodos de Java. Para ver en mayor detalle de qué se trata esta estructura, referirse a la sección 6.11.5.

2. Se obtiene la sintaxis de los nodos de tipo *FormalParameters*.

3. Se crea una nueva instancia de un nodo de este tipo (es decir, de tipo *FormalParameters*).

Nos concentraremos en la creación de la instancia. La implementación de esta funcionalidad se exhibe en el fragmento de código 19.

Analicemos paso a paso lo que se hace, siguiendo la numeración detallada en los comentarios del método.

```java
public class NodeSyntaxImpl<T extends AbstractNode> implements
    NodeSyntax<T> {
  // ...
  public final <C extends Node> T newInstance(final C... children) {
    // 1. Create the new node.
    final T newNode = nodeMetaInfo.newNode();
    if (newNode == null) {
      return null;
    }

    // 2. Set first/last token as emptyTokens.
    final GenericToken emptyToken = GenericTokens.newEmptyToken();
    newNode.jjtSetFirstToken(emptyToken);
    newNode.jjtSetLastToken(emptyToken);

    // 3. Set its structure so it can be lately synced.
    newNode.setStructure(scanNew(newNode));

    // 4. Insert its children.
    for (int childIndex = 0; childIndex < children.length; childIndex++) {
      final C child = children[childIndex];
      newNode.addChild(childIndex, child);
    }

    // 5.Activate the sync required flag in case none children have been
    //  added (i.e.: image node only).
    newNode.syncRequired();

    // 6. Return the new node with all the given children :D
    return newNode;
  }
  // ...
}
```

Código 19: Método *NodeSyntaxImpl.newInstance*

1. El nodo es creado utilizando la misma idea de *NodeMetaInfo* diseñada en la arquitectura anterior. Para más detalle sobre esto, referirse a la sección 6.10.3.

2. Dado que el nodo creado no tiene tokens, lo que se hace es setearle el primer y el último token como vacíos. Esto permite realizar las operaciones de manipulación de tokens de manera transparente a si son vacíos o no, ofreciendo una enorme ventaja por sobre el uso de valores null para representar la ausencia de valor.

3. Se genera la nueva estructura del nodo que, como mencionamos previamente, estará conformada solamente por elementos que representen tokens. Esto se verá con un poco más de detalle a continuación.

4. Se insertan los hijos pasados como argumento, en el orden en que fueron pasados. Notar que se utiliza directamente el método de manipulación *addChild*, por lo que el manejo de la lógica de sincronización explicado en la sección 6.11.3 está automáticamente incorporado en este aspecto.

5. En caso de que ningún hijo haya sido insertado, se activa la sincronización para el nodo recientemente creado, de manera que se puedan realizar las manipulaciones pertinentes una vez que el mismo sea retornado.

6. Se retorna el nodo creado.

El método *scanNew* invocado en la línea 16 del fragmento de código 19 se muestra en el fragmento 20. Notar que lo que se hace es básicamente elegir una de todas las estructuras posibles que resultan de parsear el nuevo nodo, con la política de parseo *new*, tal y como se mencionó al comienzo de esta sección. Recordar que el funcionamiento del parser se explica con mayor detalle en la sección 6.11.7.

Con esto, tenemos un panorama general del uso y funcionamiento del flujo de creación de nodos. En la figura 4 se muestra un diagrama de secuencia de las principales interacciones explicadas en este apartado. Notar el uso de *LanguageNodesSyntax* como clase genérica donde se define la sintaxis de los nodos del lenguaje (genérico) *Language* (esto se explica más en detalle, como se dijo, en la sección 6.11.5). Notar también el uso de *NodeClass* para denotar que la creación puede aplicarse a cualquier clase de nodo cuya syntaxis esté definida.

```
1 public class NodeSyntaxImpl<T extends AbstractNode> implements
   ↪ NodeSyntax<T> {
2     // ...
3     private Structure scanNew(final T node) {
4         return pickNewStructure(null,
   ↪ ensureParserPresent().parseNew(node));
5     }
6     // ...
7 }
```

Código 20: Método *NodeSyntaxImpl.scanNew*

6.11.5. Definición de NodeSyntax

NodesSyntax Recordemos que dos de las premisas mencionadas en la sección 4 para el diseño de la arquitectura eran su facilidad de adaptación a nuevos lenguajes y su mantenibilidad. Con esto en mente, se decidió implementar un mecanismo que permitiera que cada lenguaje defina la sintaxis de cada uno de sus nodos de manera centralizada.

Este mecanismo consiste en una interfaz llamada *NodesSyntax* que posee una implementación abstracta llamada *AbstractNodesSyntax*. Nos concentraremos en esta última clase, en particular, en lo mostrado en el fragmento de código 21.

Para implementar la arquitectura de manipulación de código para un nuevo lenguaje, una de las pocas cosas que deben hacerse es extender esta clase abstracta, y en ella definirse la sintaxis de todos los nodos mediante la utilización del método *define*.

Al llamar a *AbstractNodesSyntax.define* (líneas 18-20 del fragmento de código 21), básicamente se almacena en un mapa la clase, cuya sintaxis está siendo definida, como clave y la sintaxis de esa clase como valor. Notemos que, según lo que se puede ver allí, **la sintaxis de un nodo está compuesta por todos los símbolos pasados como argumento junto con la metainformación de la clase del nodo al que corresponde esta sintaxis**, que es la que permite obtener nuevas instancias de este tipo, tal y como se explica en la sección 6.11.4. Para más informacion sobre la clase *NodeMetaInfo*, referirse a la sección 6.10.3.

Un ejemplo de definición de sintaxis se puede observar en el fragmento de código 7. Cabe aclarar que este código se encuentra dentro de la clase

```java
// T is the base type of all nodes syntax
// C is a specialized type of node syntax of type T
public abstract class AbstractNodesSyntax<T extends AbstractNode>
    ↪ implements NodesSyntax<T> {
    private final Map<Class<? extends T>, NodeSyntax<? extends T>>
    ↪ syntaxByNode;
    private final NodesMetaInfo<T> nodesMetaInfo;

    protected AbstractNodesSyntax(final NodesMetaInfo<T> nodesMetaInfo) {
        syntaxByNode = new HashMap<>();
        this.nodesMetaInfo = nodesMetaInfo;
    }

    @Override
    public final <C extends T> NodeSyntax<C> getNodeSyntax(final Class<C>
    ↪ nodeClass) {
        //noinspection unchecked // We are storing keys of Class<C> with
    ↪ values of NodeSyntax<C>.
        return (NodeSyntax<C>) syntaxByNode.get(nodeClass);
    }

    protected <C extends T> void define(final Class<C> nodeClazz, final
    ↪ Symbol... symbols) {
        syntaxByNode.put(nodeClazz,
    ↪ NodeSyntaxImpl.newInstance(metaInfo(nodeClazz), symbols));
    }

    // ...
}
```

Código 21: Código principal de la clase *AbstractNodesSyntax*

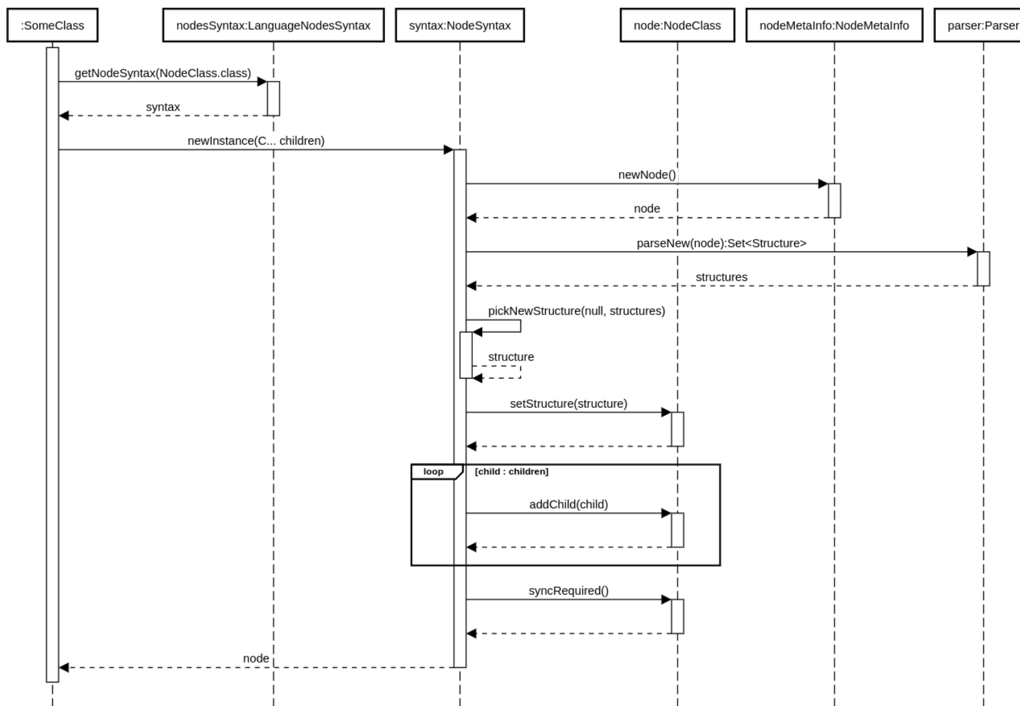

Figura 4: Diagrama de secuencia de un flujo de creación genérico.

JavaNodesSyntax, que es la extensión de la clase *AbstractNodesSyntax* especializada para Java.

Una vez definida la sintaxis de los nodos de un lenguaje, utilizando el método *AbstractNodesSyntax.getNodeSyntax* es posible acceder a las sitaxis definidas, con tan solo especificar la clase de la sintaxis deseada.

Todo esto permite que acceder y utilizar la sintaxis de cada tipo de nodo sea muchísimo más sencillo. De hecho, notemos que se pide que cada tipo de nodo sepa responder al método *getNodeSyntax* (ver por ejemplo los fragmentos de código 11 y 14), con el objetivo de poder obtener la sintaxis de los nodos independientemente de su tipo y lenguaje, para finalmente hacer uso de la funcionalidad de manipulación de código.
Dada la forma en que está implementado el sistema de definición de la sintaxis de los nodos, para utilizar este sistema en un nuevo lenguaje basta con implementar el método *getNodeSyntax* en un ancestro común a todos los nodos de un lenguaje, puesto que lo único que se necesita conocer de manera concreta es cuál es el contenedor de todas las sintaxis.

Por ejemplo, en el caso de los nodos de Java, su ancestro en común es la

clase *AbstractJavaNode*, por lo que basta con definir el método *getNodeSyntax* como se muestra en el fragmento de código 22 para poder utilizar la arquitectura de manipulación de código sobre todos los nodos de Java que hayan definido su sintaxis en la clase *JavaNodesSyntax*.

```
1  public abstract class AbstractJavaNode extends AbstractNode implements
    ↪  JavaNode {
2    // ...
3    protected NodeSyntax<? extends AbstractJavaNode> getNodeSyntax() {
4      return JavaNodesSyntax.getInstance().getNodeSyntax(this.getClass());
5    }
6    // ...
7  }
```

Código 22: Método *AbstractJavaNode.getNodeSyntax*

Notar que la misma definición puede realizarse para cualquier lenguaje, reemplazando *JavaNodesSyntax* por la clase correspondiente al lenguaje deseado.

NodeSyntax Observemos brevemente que los métodos declarados en esta interfaz e implementados en la clase *NodeSyntaxImpl* son los explicados en detalle en las secciones 6.11.3 y 6.11.4.

```
1  public interface NodeSyntax<T extends AbstractNode> {
2    <C extends Node> T newInstance(C... children);
3    Structure scan(T node);
4    Structure sync(T node);
5  }
```

Código 23: Interfaz *NodeSyntax*

En el fragmento de código 23 se puede observar la definición de la interfaz *NodeSyntax*. Notemos que las implementaciones de esta interfaz (en particular, la única que existe que es *NodeSyntaxImpl*) tienen la responsabilidad de la administración del flujo de sincronización para actualizar la cadena de tokens de manera que represente el estado actual de los nodos que componen el árbol AST. Por lo tanto, esta estructura tiene un rol importante en el funcionamiento de la arquitectura propuesta, siendo su principal tarea la correcta administración del parser y los métodos que este ofrece para lograr la sincronización entre nodos y tokens.

La sintaxis de un nodo, correspondiente a una instancia concreta de *NodeSyntax* (que en el caso implementado es una instancia de *NodeSyntaxImpl*), almacena internamente:

1. La lista de símbolos que definen la sintaxis del nodo que representan.
2. La metainformación del nodo que representan, que se utiliza exclusivamente al momento de ejecutar el método *newInstance*, tal como se muestra en la sección 6.11.4.
3. El parser correspondiente a los símbolos, instanciado una única vez y sólo en caso de que alguno de los métodos de la interfaz haya sido llamado. Esto denota nuevamente el comportamiento *lazy* de la arquitectura propuesta, mejorando la eficiencia de procesamiento y de memoria durante su uso.

La instanciación del parser consiste esencialmente en pedir una nueva instancia de parser, pasando como argumento el símbolo que *wrappea* a los símbolos de la sintaxis en cuestión. En el caso de *NodeSyntaxImpl*, eso se realiza en el método *ensureParserPresent()*. El proceso de generación del parser se explica en detalle en la sección 6.11.6, mientras que las diferentes modalidades de uso del parser se detallan en la sección 6.11.7.

6.11.6. Parser: Instanciación

El parser necesita para ser instanciado recibir solamente el primer símbolo que representa la sintaxis del nodo para el cual el parser podrá ser utilizado. Esto se puede ver en el fragmento de código 24.

Notar que, esencialmente, el parser es el NFA obtenido o compilado a partir de la expresión regular (RegExp) que se representan a partir de la instancia de símbolo pasada como argumento. Por ejemplo, en el caso de los nodos de Java de tipo *LambdaExpression*, según la sintaxis definida en el fragmento de código 7, el símbolo que se utiliza para la instanciación del parser y para la compilación del NFA subyacente es de tipo *AlternationSym*.

Recodemos que como se menciona en la sección 6.11.2, cada símbolo implementado tiene que tener la capacidad de traducirse en un fragmento del NFA final. Esto involucra no sólo traducirse a sí mismo, sino también saber

```java
public class Parser {
  private final State startState;

  private Parser(final State startState) {
    this.startState = startState;
    DebugUtils.dump(startState);
  }

  /** Build new instance (and all its parser states) based on the given
   ↪  symbol. */
  public static Parser newInstance(final Symbol symbol) {
    return new Parser(compileNFA(symbol));
  }

  // ...

  private static State compileNFA(final Symbol symbol) {
    // True value means 'mandatory = true'.
    return symbol.toNFAFragment(true).close();
  }
}
```

Código 24: Clase *Parser* - Métodos de instanciación

cómo vincular correctamente cada uno de los fragmentos NFA correspondientes a los símbolos que contiene. En particular, un hecho importante en esta traducción es identificar los estado que componen a los fragmentos de NFA que son *mandatory*, es decir, aquellos estados por los que el parser deberá pasar obligatoriamente para llegar a un estado final.

La traducción de los símbolos operadores implementados en sus respectivos fragmentos NFA se muestra en las figuras 5, 6, 7, 8, 9. La notación utilizada es la siguiente:

- **s** representa el estado creado durante la traducción del símbolo al fragmento NFA (esta creación puede no ser necesaria)
- **fi** representa el fragmento NFA del i-ésimo símbolo contenido por el símbolo que se está traduciendo. En caso de que haya solamente un símbolo contenido, su fragmento NFA correspondiente se etiqueta como **f**
- los círculos de líneas punteadas representan estados anteriores que se conectan con el primer estado del fragmento NFA que está siendo tra-

ducido.

- cada círculo sin etiqueta de líneas continuas representa uno de los posibles estados de salida del nuevo fragmento NFA. Estos círculos son solamente dibuados para reflejar la existencia de múltiples estados de salida, pero NO corresponden a estados de salida concretos (el o los estados de salida concretos son aquellos que idealmente se conectarían a este círculo)

- las transiciones entre el estado **s** y cada uno de los fragmentos **fi** (o **f**) son transiciones λ, mientras que las transiciones desde los fragmentos **fi** (o **f**) hacia sus correspondientes estados de salida dependen de la evaluación de transiciones realizadas dentro de cada uno de estos fragmentos

Notemos que en la figura 10, el fragmento del símbolo contenido se denota tanto como **fM** y **fN**. Notemos que la diferencia entre ambos es que **fM** es el fragmento **f** representado como *mandatory* (esta es la parte de *one* del símbolo) mientras que **fN** es el fragmento **f** representado como *no mandatory*. La figura 9 muestra el fragmento NFA del mismo símbolo sin marcar esta diferencia.

En cuanto a los símbolos operandos implementados por default (*ChildSE* y *TokenSE*), sus fragmentos NFA corresponden solamente a un estado, con la particularidad de que las transiciones a los estados siguientes ya no son λ, sino que la determinación de cuál es el estado siguiente, si es que debe haber alguno, depende esencialmente del tipo de parseo que se esté realizando y del matcheo de la unidad de lectura correspondiente a ese parseo. Al final de esta sección se puede encontrar una explicación más detallada de la compilación de los símbolos operandos en los estados del NFA. Para ver en más detalle los tipos de parseo posibles, junto con sus unidades de lectura, referirse a la sección 6.11.7.

Una vez que se compilan todos los fragmentos NFA, lo que se hace es *cerrar* el fragmento contenedor más grande, creando así un estado final al que se llega a través de transiciones λ desde todos los estados de salida de este fragmento. De esta forma, se obtiene el NFA que utilizará internamente el parser, según el tipo de parseo pedido.

Para reforzar lo recientemente explicado, apliquémoslo al ejemplo del nodo *LambdaExpression*. Siguiendo la sintaxis detallada en el fragmento de código 7, el NFA que generará el parser es el detallado en la figura 11. En este

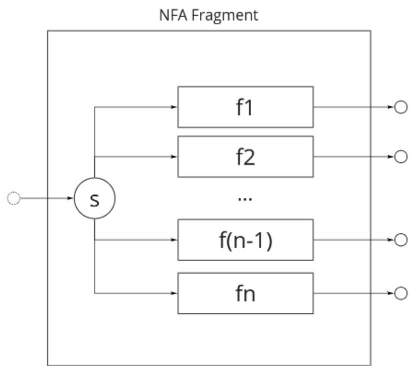

Figura 5: *AlternationSym* NFA Fragment

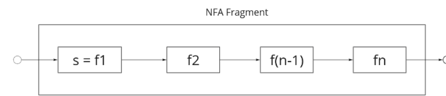

Figura 6: *ConcatenationSym* NFA Fragment

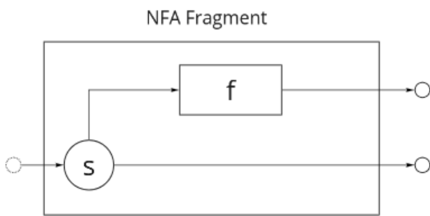

Figura 7: *ZeroOrOneSym* NFA Fragment

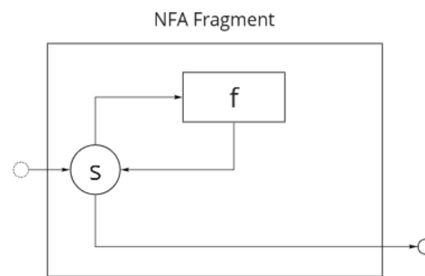

Figura 8: *ZeroOrMoreSym* NFA Fragment

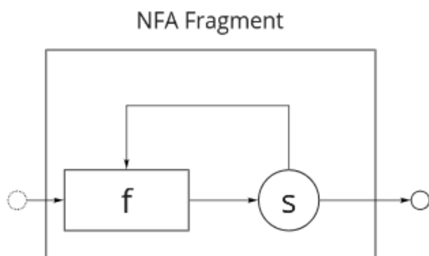

Figura 9: *OneOrMoreSym* NFA Fragment

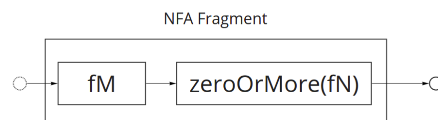

Figura 10: *OneOrMoreSym* NFA Fragment (with mandatory)

caso, el estado inicial se denota con un nodo completamente negro, mientras que el estado final se denota con un nodo dibujado con doble circunferencia.

Notar que en este NFA el único estado que es *mandatory* es el representado por la flecha de la expresión lambda, dado que para el resto de las partes de la expresión se tienen diferentes posibilidades de elección. Notar también que se marcan los respectivos fragmentos con líneas punteadas, por lo que se puede observar con claridad cómo es que se da la composición total de todos los fragmentos.

Símbolo Operando → NFA Fragment Cabe recordar que la clasificación de *símbolo operando*, a nivel código, es meramente una cuestión de nombres. La interfaz que funciona como *marker* para estos objetos es *OperandSym*. Los únicos símbolos operandos existentes en la implementación actual son aquellos que permiten saber si deben matchear o no un determinado valor parseado según el criterio de parseo, y en caso de tener que matchear, decir si efectivamente matchean o no el valor dado. Estas operaciones son declaradas en la interfaz *MatcherSym* que extiende de *OperandSym*.

Cada elemento estructural (*StructureElement* o *SE*) se comporta además como un *MatcherSym*, por lo que debe implementar su propia lógica de matcheo. Esta lógica será ejecutada por el estado que lo represente dentro del NFA cuando se alcance dicho estado (la ejecución de esta lógica se puede ver en la sección 6.11.7).

Recordemos que para el caso de los símbolos operadores, todos los estados que se creaban eran, llamémoslos, de tipo λ o *LambdaState*s, dado que al llegar a ellos, las transiciones a todos los estados siguientes eran realizadas sin ninguna validación. Sin embargo, para el caso de los símbolos operandos, las transiciones tienen distintos criterios de decisión para seleccionar el conjunto de estados de salida, por lo que podríamos llamarlos *MatchingState*s.

Dentro de los *MatchingState*, tenemos dos tipos de comportamiento:

- Sin loops (*SingleMatchingState*): Una vez que se produce un matcheo, no se esperan más valores parseados a consumir por ese estado, por lo que se pasa a los estados de salida correspondientes.
- Con loops (*ReentrantMatchingState*): Una vez que se produce un matcheo, no se sabe si habrá más valores a parsear que puedan ser consumidos por este estado, por lo que el mismo estado se pone como estado

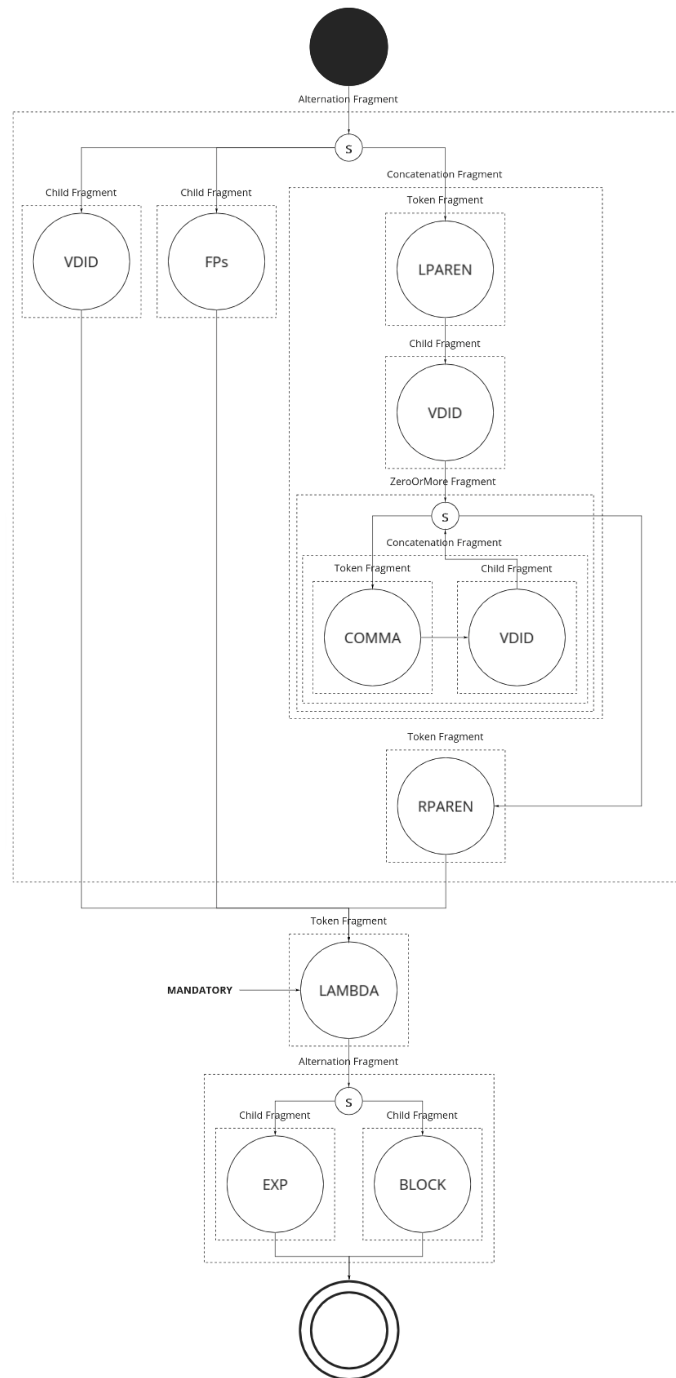

Figura 11: NFA del parser de los nodos de tipo *LambdaExpression*

de salida, y en el siguiente valor parseado se revisará si le corresponde o no revisarlo.

El funcionamiento específico y ejemplificado de cada uno de estos estados se puede ver en la sección 6.11.7.

Repasando un poco antes de continuar: los elementos estructurales implementan la interfaz *MatcherSym* (extiende de *OperandSym*), por lo que ellos pueden ser tratados como símbolos operandos. Además, todos los *SE* se compilan a un NFA Fragment que tiene un único estado de tipo *MatchingState*, que puede ser: *SingleMatchingState* (sin loops) o *ReentrantMatchingState* (con loops).

Dada esta última característica (es decir, que todos los *SE* se compilan a un NFA Fragment que tiene un único estado de tipo *MatchingState*), se cuenta con una clase abstracta *AbstractSE* que crea el fragmento NFA y le pide a las clases que la extiendan que implementen qué tipo de matching state desean (ver el fragmento de código 25).

```
1  public abstract class AbstractSE implements StructureElement,
    ↪  ParsedValueMatcher {
2    // ...
3    @Override
4    public NFAFragment toNFAFragment(final boolean areStatesMandatory) {
5      final MatchingState thisState = newMatchingState(areStatesMandatory);
6      final NFAFragment thisFragment = NFAFragment.newInstance(thisState);
7      thisFragment.addOutStates(thisState.getOutStates());
8      return thisFragment;
9    }
10
11   protected abstract MatchingState newMatchingState(boolean
    ↪  areStatesMandatory);
12   // ...
13 }
```

Código 25: Método *AbstractSE.toNFAFragment*

Para evitar código duplicado, se crearon dos clases que extienden de *AbstractSE*: *AbstractSingleMatchingStateSE* y *AbstractReentrantMatchingStateSE*, las cuales solamente implementan el método *newMatchingState*, creando (como resulta evidente del nombre) un *SingleMatchingState* y un *ReentrantMatchingState* respectivamente.

Habiendo explicado esto, lo único que queda por mencionar es que los *StructureElements* implementados (*ChildSE* y *AbstractTokenSE*) ambos extienden de *SingleMatchingState*. Como se dijo, el funcionamiento particular de la lógica de matcheo se explica en la sección 6.11.7.

En la figura 12 se puede observar el diagrama de clases que involucra a las clases mencionadas en esta parte.

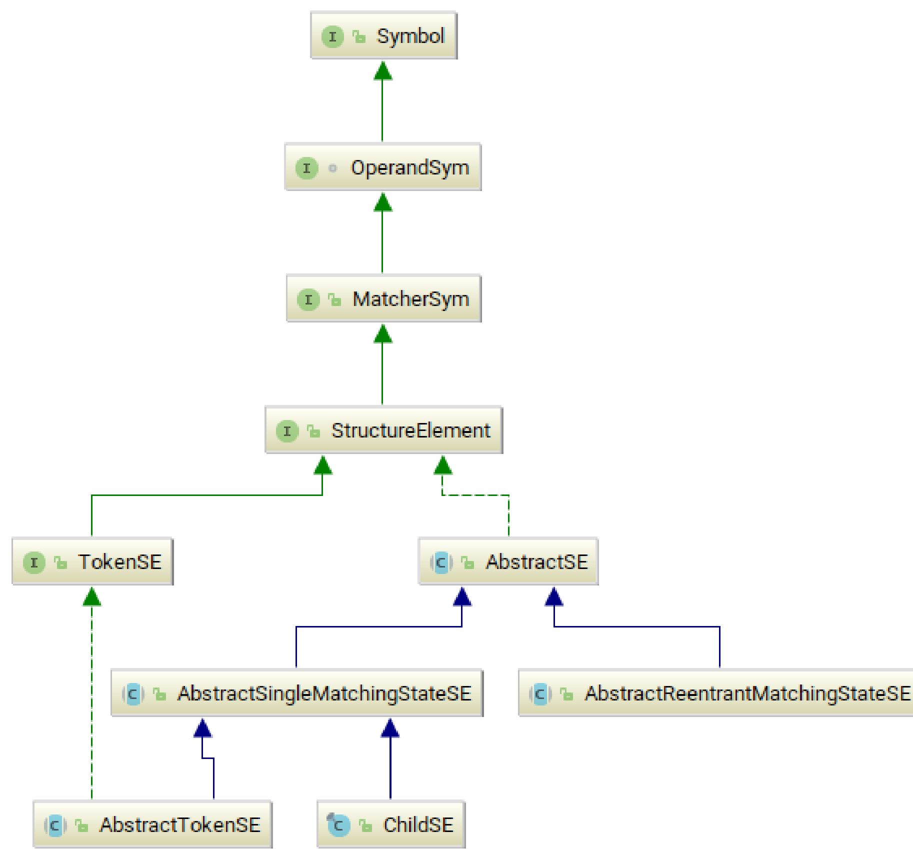

Figura 12: Diagrama de clases [12] de los *StructureElement* implementados en el paquete *pmd-core*.

[12] La notación utilizada en este diagrame de clase puede verse en: https://www.jetbrains.com/help/idea/class-diagram-toolbar-and-context-menu.html#d44243e211

6.11.7. Parser: Funcionamiento

Habiendo construido el NFA subyacente al parser (esto se detalla en la sección 6.11.6), resta entender cómo puede ser utilizado.

En las secciones 6.11.3 y 6.11.4 se muestran tres usos posibles del parser.

1. Parseo completo de un nodo (*parseComplete*): se utiliza para determinar la estructura original de un nodo, previo a ser manipulado
2. Parseo de los hijos (y las propiedades) de un nodo (*parseChildren*): se utiliza para determinar la nueva estructura de un nodo, luego de haber sido manipulado (las manipulaciones pueden ser tanto a nivel hijo como a nivel propiedades)
3. Parseo de un nodo nuevo (*parseNew*): se utiliza para construir la estructura de un nodo recientemente creado.

Cada modo de parseo requiere una lógica de lectura distinta del nodo, lo que lleva asociado también un uso distinto del NFA subyacente al parser. En definitiva, el uso del NFA varía de una u otra forma según la lógica de lectura que esté siendo utilizada.

La forma de leer el nodo está a cargo de implementaciones concretas de la interfaz *ParseContext*, que son (respectivamente, siguiendo el orden de la enumeración anterior): *CompleteParseContext*, *ChildrenParseContext* y *GenerateParseContext*. Cada contexto de parseo o *ParseContext* sabe cómo leer al nodo de forma de responder a los métodos de la interfaz, que son los que se muestran en el fragmento de código 26, según la política de parseo (*ParsePolicy*) que corresponda. Notar que exite una implemntación abstracta de *ParseContext*, llamada *AbstractParseContext*, donde se implementan los métodos necesarios para consumir las características del nodo de la manera correcta (esto implica, como se verá a continuación, consumir los tokens y los hijos de los nodos una única vez y de manera progresiva).

Antes de continuar, consideremos lo siguiente. ¿Por qué es necesario consumir de a unidades regulares y no dejar que cada estado del NFA consuma alguna parte del nodo que necesita consumir (una propiedad, un token o un hijo)? La respuesta a esta pregunta es clave en el desarrollo y comprensión de todo lo que sigue, por lo que nos concentraremos en responderla de manera

```java
public interface ParseContext {
  boolean hasNext();
  ParsedValue parseNext();
  ParsePolicy getParsePolicy();
}
```

Código 26: Interfaz *ParseContext*

detallada. En primer lugar, si cada estado consumiera del nodo la parte que necesita, entonces cabrían 2 posibilidades:

1. Mantener N contextos de parseos diferentes, uno por cada estado actual al momento de ejecutar el parser.
2. Mantener un contexto, lo que implica que el parser puede recorrer solamente un camino a la vez. En este caso, es necesario implementar una lógica de backtracking adecuada.

Ambos casos presentan serios problemas, no sólo en términos de eficiencia computacional y espacial, sino también en lo que respecta a mantenibilidad de código por la cantidad de líneas extras necesarias para implementarlas.

Es por esto que decidimos buscar una solución a este problema. En la web [2], se propone utilizar y se explica con un buen nivel de detalle el método introducido por Ken Thompson[12] para procesar expresiones regulares, comparándolo con otros algoritmos que procesan expresiones regulares y mostrando la alta escalabilidad y eficiencia del método de Thompson NFA, sobre todo debido al no uso de bactracking para procesar el input.

Si se observa con cuidado, la lógica implementada al construir los símbolos de las expresiones regulares (sección 6.11.2) y al realizar la compilación de los símbolos en el NFA (sección 6.11.6) está fuertemente basada en lo explicado en este sitio, es decir, en Thompson NFA.

Para evitar el uso de backtracking, la idea es, en términos sencillos, avanzar de un set de estados a otro, y siempre consumiendo *una unidad del input* una única vez. Una vez que se terminó de consumir todo el input, si entre los estados actuales hay algún estado final, entonces el input es válido. Para más detalles del funcionamiento de esta lógica ver [2].

Si bien este algoritmo es mucho más eficiente y sencillo de implementar

que otras alternativas, el problema ahora pasa a definir cuál debería ser la unidad de input, es decir, la unidad de lectura para parsear a un nodo. Seguro que un nodo está compuesto por tokens, pero si se tomara esta unidad de lectura, reconocer a sus hijos sería al menos costoso, porque habría que matchear todos los tokens que se parsean con cada uno de los tokens del próximo hijo sin consumir para decir que efectivamente matchean. Además de que es costoso, lo que suceda dentro del contexto del hijo de un nodo no debiera preocuparnos a nivel algorítmico cuando estamos parados en el nodo padre.

Luego, debe considerarse una unidad de lectura que permita, uniformemente, leer *de a hijos* cuando se reconoce una región de tokens del próximo hijo que aún no se escaneó (reconocer la región sin recorrer todos los tokens exhaustivamente), mientras que se debe leer *de a tokens* cuando el token no pertenece a una región de un hijo. Esta unidad de lectura uniforme es la interfaz *ParsedValue*, cuyas implementaciones concretas son: *ChildParsedValue* y *TokenParsedValue*, respectivamente a los casos explicados anteriormente.

¿Cómo reconocer las regiones de los hijos sin recorrer todos sus tokens exhaustivamente? Dado que se garantiza, incluso luego de correr el algoritmo de sincronización mostrado superficialmente en la sección 6.11.3, que existe un único camino de tokens para llegar a un token destino (el camino puede ser más o menos largo, depende del nodo en el que se comience a recorrerlo), para determinar la región de tokens de un hijo basta con comparar el token actual que se está por consumir con el primer token del próximo hijo a consumir.

- En caso de que sean iguales, sabemos que todos los tokens que siguen, hasta el último token de ese hijo, son los que vienen después, por lo que se parsea una unidad de lectura de tipo hijo. Es decir, se instancia un *ChildParsedValue* que contiene al hijo parseado y a la región de tokens que comprende el hijo, que es una *TokenRegion* que va desde el primer hasta el último token del hijo, inclusive.
- En caso de que sean distintos, se parsea una unidad de lectura de tipo token. Es decir, se instancia un *TokenParsedValue* que contiene al token parseado y una *TokenRegion* unitaria (el token de comienzo y finalización de la región es el mismo token).

¿Y para parsear las propiedades de un nodo? Dado que las propiedades no son hijos, pueden verse como una secuencia de tokens (o hijos y tokens) que guardan cierto valor semántico dentro del nodo que los contiene, para un

lenguaje dado. De esta forma, para procesar propiedades, bastaría crear un estado que *consuma unidades de lectura hasta que encuentre una unidad que no le corresponde procesar, momento en el cual deriva esta tarea al estado siguiente, si correspondiera.* Este es un ejemplo particular de cuándo una implementación de *StructureElement* puede utilizar un *ReentrantMatchingState* que lo represente en el NFA del parser. Para más información sobre tipos de estados, referirse a la sección 6.11.6. Su uso se explicará también con más detalle más adelante en esta misma sección.

Implementar este sistema de estructuras genéricas (unidades de lectura y contextos de parseo, según cada nodo y la necesidad por la que se está ejecutando el parser) permite que el algoritmo del parser sea implementado de manera transparente a estas cuestiones. El fragmento de código 27 muestra este algoritmo.

Transcripción Para explicar lo que sucede en este método y la terminología usada, es oportuno volver a explicar de dónde surgió la idea (ya fue mencionado oportunamente en la sección 6.11.1, aunque con otro propósito).

Como se menciona en la sección donde se presenta la idea (6.11.2) y en las secciones donde se muestran los flujos de manipulación y creación (6.11.3 y 6.11.4 respectivamente), lo que permite el parser es construir la estructura correspondiente a un nodo según su información (y según el propósito de construir esta estructura, es decir, según la política de parseo).

Notemos que el proceso biológico de transcripción del ADN (ver [1]) funciona básicamente de la siguiente forma: una proteína se posiciona sobre una secuencia promotora, y junto con la ayuda de otras proteínas, comienza a leer cada base de una de las dos tiras del ADN, generando otra tira con las bases complementarias que va leyendo, hasta llegar a una secuencia de parada, momento en el cual libera la tira que fue generada.

Lo que tratamos de hacer fue imitar este proceso: dado un nodo, desde su primer token (secuencia promotora dentro de toda la cadena de tokens del programa) hasta su último token (secuencia de parada), lo que se hace es generar una estructura a partir de su información. A este proceso lo llamamos transcripción.

En nuestro caso, el nodo actual es el ADN, y los *ParsedValue*s son las bases (según el proceso de transcripción biológico).

```
1   public class Parser {
2     // ...
3     private Set<Structure> parse(final ParseContext parseContext) {
4       Set<StructureTranscription> currTranscriptions = new HashSet<>();
5       currTranscriptions.add(StructureTranscription.newInstance(startState,
    ↪  Structure.newBuilder()));
6       while (parseContext.hasNext()) {
7         currTranscriptions = step(parseContext.getParsePolicy(),
    ↪  parseContext.parseNext(), currTranscriptions);
8         if (currTranscriptions.isEmpty()) { // There's no matching state =>
    ↪  this node has no matching structure.
9             throw new IllegalStateException("Node structure is invalid: not
    ↪  matching the given tokens.");
10        }
11      }
12
13      // One final step to consume epsilon transitions.
14      currTranscriptions = step(parseContext.getParsePolicy(), null,
    ↪  currTranscriptions);
15
16      final Set<Structure> availableStructures = new HashSet<>();
17      for (final StructureTranscription transcription : currTranscriptions)
    ↪  {
18        if (transcription.hasFinished()) {
19            availableStructures.add(transcription.get());
20        }
21      }
22
23      return availableStructures;
24    }
25    // ...
26  }
```

Código 27: Método *Parser.parse*

¿Con qué elementos se genera la estructura de la transcripción? En lugar de utilizar las bases complementarias biológicas, lo que se usan son los *StructureElements* mencionados en las secciones 6.11.2 y 6.11.6, y explicados en la sección 6.10.1, pero *wrappeados* junto con otra información en una clase denominada *SEInfo*. Esta información extra se utiliza para marcar: 1) si el elemento estructural actual es o no obligatorio; 2) La región de tokens del nodo que le corresponde.

Estas estructuras son las que permiten mantener la información del nodo, "segmentada.en regiones de tokens, asociada a elementos estructurales, que luego serán utilizadas para realizar la sincronización de las nuevas propiedades de los nodos en la cadena de tokens, de manera de garantizar que las regiones internas a modificar de un nodo sean sólo las necesarias (es decir, aquellas en las que se hayan producido modificaciones).

Análisis del método *Parser.parse* (fragmento de código 27) Habiendo aclarado de dónde surgió la idea del parser y el proceso de transcripción, ahora es más evidente la nomenclatura utilizada a lo largo de la implementación mostrada. Lo que se hace en el algoritmo es básicamente (en orden):

1. Cargar la transcripción estructural (*StructureTranscription*) inicial.
2. Mientras haya unidades de parseo para consumir (según la lógica implementada por el contexto de parseo), avanzar un *paso* a lo largo del NFA, utilizando la política y el valor de parseo junto con el set de todas las transcripciones estructurales actuales. Dar este *paso*, modifica, crea o remueve las transcripciones estructurales según cómo se haya avanzado en ese paso. En caso de que no haya ninguna transcripción posible, es evidente que el nodo actual está mal formado con respecto a la sintaxis provista, puesto que no hay ningún camino en el NFA subyacente que permita representarlo.
3. Una vez consumidas todas las unidades de parseo, se terminan de consumir todas las transiciones lambda del NFA desde los estados actuales (que están almacenados internamente en los *StructureTranscription*) dando un *paso* a lo largo del NFA, pasando como valor de parseo *null*.
4. Se generan y retornan sólo las estructuras a partir de las transcripciones realizadas, pero sólo de aquellas transcripciones que hayan logrado

llegar hasta un estado final (es decir, que sean estructuras completas y válidas según la información del nodo).

Notar que el método *Parser.step* lo que hace es ejecutar el mismo paso sobre cada una de las trancripciones actuales, lo que en definitiva consiste en ejecutar un *paso* o *paso de transcripción* desde el estado actual subyacente a cada transcripción utilizando la política y el valor de parseo provistos, y almacenando los elementos estructurales en el *builder* asociado a esa transcripción (ver fragmento de código 28 y 29 respectivamente). Cabe destacar que a partir de realizar un paso de transcripción sobre un estado pueden surgir múltiples transcripciones estructurales; en dicho caso, cada una de ellas contiene un *builder* de la estructura que es exactamente igual al original (es decir, al que se utiliza al ejecutar la línea *aState.transcribe(...)*), con la excepción de los nuevos elementos estructurales agregados al ejecutar efectivamente las transiciones en el NFA subyacente.

La lógica de transcripción de cada estado depende de su implementación específica (es decir, del tipo de estado del que se trate). A continuación analizaremos las implementaciones *SingleMatchingState* y *ReentrantMatchingState*, que son los dos estados posibles para representar símbolos operandos en el NFA, según se detalla en la sección 6.11.6. Para más información sobre los otros tipos de estados implementados, consultar en el código fuente la interfaz *State* y el resto de sus implementaciones.

```
1  public class Parser {
2    // ...
3    private Set<StructureTranscription> step(final ParsePolicy policy,
4                                             final ParsedValue value,
5                                             final
   ↪  Set<StructureTranscription> transcriptions) {
6        final Set<StructureTranscription> nextTranscriptions = new
   ↪  HashSet<>();
7        for (final StructureTranscription transcription : transcriptions) {
8            nextTranscriptions.addAll(transcription.step(policy, value));
9        }
10       return nextTranscriptions;
11   }
12   // ...
13 }
```

Código 28: Método *Parser.step*

```java
public class StructureTranscription {
  private final State aState;
  private final Structure.Builder structureBuilder;
  // ...
  public Set<StructureTranscription> step(final ParsePolicy policy, final
   ParsedValue value) {
    return aState.transcribe(policy, value, structureBuilder);
  }

  public boolean hasFinished() {
    return EndState.getInstance().equals(aState);
  }

  public Structure get() {
    return structureBuilder.build();
  }
}
```

Código 29: Método *StructureTranscription.step*

MatchingStates*: *SingleMatchingState* y *ReentrantMatchingState
Nos enfocaremos particularmente en la implementación del método *transcribe*, para analizar la lógica de matcheo realizada por cada una de estos tipos de estados.

SingleMatchingState Como se menciona en la sección 6.11.6, las instancias de *SingleMatchingState* representan estados en el NFA que no tienen loops, es decir, que una vez que se produce un matcheo, no se esperan más valores parseados a consumir por ese estado, por lo que se pasa (en caso de que se produzca ese matcheo) a los estados de salida correspondientes.

El fragmento de código 30 muestra la implementación del método *transcribe* de esta clase. Lo que se hace en este método, desde el punto de vista del estado, es:

1. Revisar si, según la política de parseo y el valor dados, debería intentar matchear el valor dado. En caso de que no, delego inmediatamente esta tarea a todos mis posibles estados de salida.
   Por ejemplo, si se está ejecutando el método *parseChildren* del parser, la *ParsePolicy* será *ParsePolicy.CHILD_ONLY*, por lo que las instancias de *SingleMatchingState* que tengan como *StructureElement* asociado a una especialización que extienda de *AbstractTokenSE* delegarán la

llamada (ver la implementación de esta clase en el código fuente).
En cambio, si se ejecuta el método *parseComplete* del parser, la *ParsePolicy* será *ParsePolicy.COMPLETE*, por lo que estas mismas instancias intentarán matchear el valor dado.

2. Si debemos matchear el valor, y este valor no está definido (es decir, nos están pidiendo realizar una transición *lambda*), no podremos realizar la evaluación del matcheo, por lo que no ejecutamos ninguna lógica y nos retornamos a nosotros mismos como estado de salida, dado que no podemos decidir el matcheo por sí o por no. Es decir, al no contar con transiciones lambda, si nos piden que nos movamos a traves de este tipo de transiciones, la salida es no moverse del estado actual.

3. Luego de los chequeos anteriores, nos fijamos si efectivamente matcheamos o no. No matchear significa que no existen transiciones a otros estados usando el valor parseado y el criterio establecido por el *StructureElement* subyacente. Por eso, en caso de que no se produzca el matcheo, retornamos una colección vacía (en definitiva, este camino no lleva a ningún lado).

4. Si matcheamos, lo que hacemos es agregar nuestro *StructureElement* a la estructura que está siendo construida y luego retornamos una copia de esta estructura en construcción junto a cada uno de nuestros estados de salida alcanzados a través del valor matcheado.

En particular, cabe mencionar que todos los *ChildSE* e implementaciones concretas de *AbstractTokenSE* se compilan a fragmentos NFA que tienen este tipo de estado como representante. Para más información sobre el proceso de compilación de *StructureElement* a NFA Fragment, referirse a la sección 6.11.6. Para ver el diagrama de clases de estos elementos estructurales, ver la figura 12.

ReentrantMatchingState Como se menciona en la sección 6.11.6, las instancias de *ReentrantMatchingState* representan estados en el NFA que tienen loops, es decir, aquellos en los que en caso de producirse un matcheo, no se sabe si habrá más valores a parsear que puedan ser consumidos por este estado, por lo que el mismo estado se vuelve a poner como estado de salida, y en el siguiente valor parseado se revisa si le corresponde o no matchear el nuevo valor parseado.

```
1  public class SingleMatchingState extends AbstractNonEndState implements
    ↪  MatchingState {
2    private final StructureElement seMatcher;
3    private final boolean isMandatory;
4    // ...
5    public Set<StructureTranscription> transcribe(final ParsePolicy policy,
    ↪  final ParsedValue value, final Structure.Builder structureBuilder) {
6      if (!seMatcher.shouldMatch(policy, value)) {
7        structureBuilder.append(SEInfo.newInstance(seMatcher, isMandatory));
8        return delegateTranscribe(policy, value, structureBuilder);
9      }
10
11     if (value == null) { // Epsilon or Lambda transition.
12       return
    ↪  Collections.singleton(StructureTranscription.newInstance(this,
    ↪  structureBuilder));
13     }
14
15     if (!seMatcher.doesMatch(value)) {
16       return Collections.emptySet();
17     }
18
19     final Set<State> outStates = getOutStates().get();
20     if (outStates.isEmpty()) {
21       return Collections.emptySet();
22     }
23
24     structureBuilder.append(SEInfo.newInstance(seMatcher, isMandatory,
    ↪  value.getTokenRegion()));
25
26     final Set<StructureTranscription> nextStates = new
    ↪  HashSet<>(outStates.size());
27     for (final State outState : outStates) {
28       nextStates.add(StructureTranscription.newInstance(outState,
    ↪  structureBuilder.duplicate()));
29     }
30
31     return nextStates;
32   }
33   // ...
34 }
```

Código 30: Método *SingleMatchingState.transcribe*

El fragmento de código 31 muestra la implementación del método *transcribe* de esta clase. Lo que se hace en este método, desde el punto de vista del estado, es:

1. Si es la primera vez que entramos al estado, lo que hacemos es agregar nuestro elemento estructural subyacente con una región vacía.
2. Si no debemos matchear el valor parseado según la política de parseo, delegamos la transcripción a nuestros estados de salida.
3. Si debemos matchear el valor parseado, pero nos están solicitando una transición *lambda*, al igual que en el caso del *SingleMatchingState*, no podemos decidir si debemos hacer un loop más (en caso de matcheo) o si debemos continuar hacia nuestros estados de salida, por lo que nos retornamos a nosotros mismos como estado de salida (es decir, no hacemos ningún progreso sobre el NFA).
4. Si no matcheamos con el valor que nos pasan, lo que hacemos es delegar la transcripción a nuestros estados de salida, puesto que ya no es nuestra responsabilidad consumir este valor. En este caso, nos comportamos como una transición lambda saliendo del loop explicado anteriormente.
5. Si matcheamos, lo que hacemos es extender la región que nos representa en la estructura que está siendo construída, y luego nos retornamos a nosotros mismos como siguiente estado, puesto que es posible que el próximo valor a parsear también nos corresponda a nosotros, según lo explicado anteriormente.

En particular, cabe mencionar que este tipo de estado es utilizado como representante en el NFA por implementaciones de propiedades particulares para los lenguajes, que son conjuntos de *tokens* y/o *children* que guardan sentido semántico solamente dentro de ese lenguaje como propiedades de los nodos a los que pertenecen. En el código fuente implementado, este es el caso de los *StructureElements*: *ArraySE* y *ModifiersSE* para Java.

6.11.8. Sincronización

En el fragmento de código 15 mostramos que la sincronización de las características del nodo en la cadena de tokens es realizada utilizando una estructura generada por el parser (seleccionada de entre todas las posibles

```java
public class ReentrantMatchingState extends AbstractNonEndState implements
    MatchingState {
  private final StructureElement seMatcher;
  private final boolean isMandatory;
  // ...
  @Override
  public Set<StructureTranscription> transcribe(final ParsePolicy policy,
      final ParsedValue value, final Structure.Builder structureBuilder) {
    SEInfo thisSEInfo = structureBuilder.peekLastSEInfo();
    if (thisSEInfo == null || !thisSEInfo.getSE().equals(seMatcher)) {
      thisSEInfo = SEInfo.newInstance(seMatcher, isMandatory);
      structureBuilder.append(thisSEInfo);
    }

    if (!seMatcher.shouldMatch(policy, value)) {
      return delegateTranscribe(policy, value, structureBuilder);
    }

    if (value == null) { // Epsilon or Lambda transition.
      return
          Collections.singleton(StructureTranscription.newInstance(this,
          structureBuilder));
    }

    if (!seMatcher.doesMatch(value)) {
      return delegateTranscribe(policy, value, structureBuilder);
    }

    thisSEInfo.extendRegion(value.getTokenRegion());
    return Collections.singleton(StructureTranscription.newInstance(this,
        structureBuilder));
  }
  // ...
}
```

Código 31: Método *ReentrantMatchingState.transcribe*

con el criterio que se explica en el texto que rodea al fragmento de código mencionado).

En esta sección nos adentraremos en este proceso de sincronización. Para ello, comenzaremos por revisar el método *sync* de la clase *Structure*, que se puede observar en el fragmento de código 32.

```
1  public class Structure {
2    // ...
3    public void sync(final AbstractNode node) {
4      final SyncContext syncContext = SyncContext.newInstance(node);
5      for (final SEInfo seInfo : structureInfo) {
6        seInfo.sync(syncContext);
7      }
8      syncBoundTokens(node);
9    }
10   // ...
11 }
```

Código 32: Método *Structure.sync*

Como se puede ver, la sincronización entonces consiste básicamente en generar un contexto de sincronización, y sincronizar individualemente cada uno de los *SEInfo* que componen la estructura, utilizando este contexto. Recordar que los *SEInfo* son los elementos estructurales con la información extra de si son *mandatory* o no y la región de tokens que les corresponde en el nodo.

Luego de esto, lo que se hace es sincronizar los *bound* tokens del nodo. Esto es: si hubo modificaciones en el primer y/o último token del nodo que se acaba de sincronizar, se llama al padre de este nodo para que actualice su campo de primer y/o último token, y así sucesivamente hasta llegar a la raíz (utilizando este criterio de actualización). Esto es necesario puesto que existen casos en que los campos de primer y/o último token de un nodo son los mismo que los de su padre, y si se actualiza el hijo sin actualizar al padre, entonces la información del padre queda desactualizada, y la cadena de tokens del código fuente original pasa a ser inválida. Para más información sobre este método, consultar el código fuente.

Antes de continuar, cabe destacar que es en la lógica de sincronización donde debería realizarse cualquier mejora que pretenda hacerse en cuestiones relacionadas al proceso de traducción de manipulaciones sobre el AST a código fuente (en definitiva, a cambios en los tokens que representan el código

fuente).

SyncContext El contexto de sincronización permite, de manera transparente, ir recorriendo las propiedades del nodo y su estructura anterior, para que, a partir de esta información, cada *SEInfo* pueda sincronizarse a la región de tokens que le corresponda realizando sólo los cambios mínimos necesarios. Además, se encarga de ir ensamblando cada una de las regiones de tokens de la nueva estructura, lo que permite centralizar y hacer exclusiva la manipulación de la cadena de tokens a esta clase. Para ello, cuando se inicializa el contexto de sincronización, lo que se hace es remover la región de tokens correspondiente al nodo de la cadena de tokens del código fuente, para luego, a medida que se van sincronizando las regiones de cada uno de los *SEInfo*, ir insertando dichas regiones en la cadena de tokens del código fuente. Para más información sobre la clase *SyncContext*, consultar el código fuente.

SEInfo.sync Analicemos ahora la sincronización de los *SEInfo*, que se exhibe en el fragmento de código 33. Notemos que el flujo de sincronización varía dependiendo del campo *isMandatorySE* del *SEInfo*, que denota la obligatoriedad o no de un *StructureElement* del tipo subyacente en la estructura del nodo.

```
1  public class SEInfo {
2    private final StructureElement structureElement;
3    private final boolean isMandatorySE;
4    private TokenRegion tokenRegion;
5    // ...
6    public void sync(final SyncContext syncContext) {
7      // Sync & Recognize our tokens region in the tokens chain
   ↪  accordingly.
8      if (isMandatorySE) {
9        syncMandatory(syncContext);
10     } else {
11       syncOptional(syncContext);
12     }
13   }
14   // ...
15 }
```

Código 33: Método *SEInfo.sync*

La diferencia radical entre la sincronización *mandatory* y la *optional* con-

siste en la forma en que se consume la estructura anterior del nodo (a través del *SyncContext*). Para ver la implementación de ambos métodos, consultar el código fuente. A continuación se explican cada uno de los métodos.

- *syncMandatory*: Cuando el *SEInfo* es *mandatory*, sabemos que nuestro *StructureElement* se encuentra también presente en la estructura anterior, independientemente de cómo esté formada. Si recorremos ambas estructuras (la nueva y la vieja) desde el comienzo hasta el final, es seguro que ambas tendrán la misma cantidad de elementos estructurales obligatorios, y que además estarán en el mismo orden en ambas estructuras.
  Aprovechando este conocimiento, cuando se sincorniza un *SEInfo* de tipo *mandatory*, lo que se hace es consumir los *SEInfo* de la estructura anterior hasta encontrar el próximo *mandatory*. En ese momento, el nuevo elemento estructural se sincroniza con el viejo y con las propiedades que requiera del nodo (que también son accedidas a través de la instancia de *SyncContext* para poder manejarlas de manera centralizada).
- *syncOptional*: Cuando el *SEInfo* es **no** *mandatory* (es decir, *optional*), no tenemos la certeza de si nuestro *StructureElement* se encuentra también presente en la estructura anterior. Lo que se hace en este caso es pedirle al *SyncContext* que nos de el próximo *SEInfo* de la estructura anterior y marcarlo como *usado* sólo en caso de que las regiones de tokens de ambos elementos estructurales coincidan (es decir, que no haya habido cambios en la región subyacente).

El encargado de realizar la sincronización en cualquiera de los dos casos es el *StructureElement* subyacente al *SEInfo*, que recibe:

1. El contexto de sincronización, al cual le puede solicitar información particular del nodo (como por ejemplo, el próximo hijo a sincronizar).
2. El *StructureElement* del *SEInfo* de la estructura anterior.
3. La *TokenRegion* del *SEInfo* de la estructura anterior.

Esta información es utilizada para realizar la sincronización con la lógica que corresponda en cada caso.

Analicemos por ejemplo, los casos de implementación del método *sync* para los *StructureElements*: *ChildSE* y *AbstractTokenSE*.

ChildSE.sync En el fragmento de código 34 podemos observar la implementación (simplificada, puesto que el código contiene chequeos que a los fines prácticos de esta explicación son irrelevantes) de la sincronización para el caso de los elementos estructurales de tipo *Child*. Notar que lo que se hace es simplemente crear una nueva *TokenRegion* en caso de que la región del elemento estructural de la estructura anterior no coincida con la esperada (que es la región delimitada por el hijo en cuestión). Esta nueva región estará delimitada entonces por el primer y último token del hijo en cuestión, y es la que luego inserta el *SEInfo* a través del *SyncContext* en la cadena de tokens (ver método *SyncContext.insert* en el código fuente).

```
1  public final class ChildSE extends AbstractSingleMatchingStateSE {
2    // ...
3    public TokenRegion sync(final SyncContext syncContext, final
   ↪  StructureElement oldSE, final TokenRegion oldTokenRegion) {
4      final AbstractNode child = syncContext.nextChild();
5
6      final GenericToken childFirstToken = child.jjtGetFirstToken();
7      final GenericToken childLastToken = child.jjtGetLastToken();
8      final GenericToken oldFirstToken = oldTokenRegion.getFirstToken();
9      final GenericToken oldLastToken = oldTokenRegion.getLastToken();
10     if (this.equals(oldSE) && childFirstToken.equals(oldFirstToken) &&
   ↪  childLastToken.equals(oldLastToken)) {
11       return oldTokenRegion; // We are representing the exact same
   ↪  element.
12     }
13
14     // We need to create a new region for the matching child tokens.
15     return TokenRegion.newInstance(childFirstToken, childLastToken);
16   }
17   // ...
18 }
```

Código 34: Método *ChildSE.sync*

AbstractTokenSE.sync En el fragmento de código 35 podemos observar la implementación de la sincronización para el caso de los elementos estructurales de tipo *Token*. Notar que lo que se hace es simplemente crear una nueva *TokenRegion* sólo en caso de que así se requiera: si los *StructureElement* y los tokens en sí (sin *SpecialTokens*, es decir, comentarios, espaciados, etc.) coinciden, entonces reutilizamos la misma región, dado que estamos representando la misma región del código. Al igual que en el caso de *ChildSE.sync* (y que el resto de las implementaciones de *StructureElement.sync*),

la región retornada por este método es la que luego inserta el *SEInfo* a través del *SyncContext* en la cadena de tokens (ver método *SyncContext.insert* en el código fuente).

```
1  public abstract class AbstractTokenSE extends
    ↪  AbstractSingleMatchingStateSE implements TokenSE {
2    // ...
3    public TokenRegion sync(final SyncContext syncContext, final
    ↪  StructureElement oldSE, final TokenRegion oldTokenRegion) {
4      // We are representing the same token => reuse old region :D
5      if (this.equals(oldSE) &&
    ↪  oldTokenRegion.stringifyWithoutSpecial().equals(getImage())) {
6        return oldTokenRegion;
7      }
8      // Create a new region with a new own token.
9      final GenericToken newToken = newToken();
10     return TokenRegion.newInstance(newToken, newToken);
11   }
12   // ...
13 }
```

Código 35: Método *AbstractTokenSE.sync*

6.11.9. Manipulación de hijos de nodos

Para permitir la manipulación de los hijos de un nodo de manera genérica, se implementaron los métodos *addChild*, *setChild*, *removeChild* y *remove*. En todos ellos, lo que se hace (tal como se mencionó en la sección 6.11.3) es primeramente, validar la correctitud de los parámetros; luego, se llama al método *syncRequired* y finalmente se realiza la manipulación solicitada.

Los métodos fueron agregados a la interfaz *Node*, y obedecen el comportamiento y la firma de los métodos *add*, *set*, *remove* de la interfaz *List* de Java[13].

A continuación se explica cada uno de ellos.

1. Existen dos firmas del método *addChild*: *addChild(int, Node)* y *addChild(Node)*. El primero inserta en el índice especificado el nodo hijo

[13]https://docs.oracle.com/javase/10/docs/api/java/util/List.html

dado, shifteando a la derecha todos los otros hijos que aparezcan después (y actualizando sus campos correspondientes), mientras que el segundo simplemente llama al primero con el índice del tamaño del array de hijos (es decir, apendea el nodo dado como último hijo). El máximo índice permitido es igual al número de hijos del nodo sobre el cuál se está ejecutando la inserción; esto se pide de esta forma para garantizar que no queden posiciones del array de hijos en null.

2. La firma del método *setChild* es *setChild(int, Node)*, donde el máximo índice permitido es igual al número de hijos menos uno del nodo sobre el cual se está ejecutando el método. Lo que se hace en este caso es reemplazar el nodo hijo que está en la posición indicada por el índice (primer parámetro) por el nodo hijo pasado por parámetro.

3. La firma del método *removeChild* es *removeChild(int)*, donde el máximo índice permitido es igual al número de hijos menos uno del nodo sobre el cual se está ejecutando el método. Lo que se hace en este caso es remover el nodo hijo que está en la posición indicada por el índice dado, shifteando a la izquierda todos los otros hijos que aparezcan después (y actualizando sus campos correspondientes).

4. La firma del método *remove* es *remove()*, donde lo que se hace es sencillamente ejecutar el método *removeChild* sobre el padre del nodo sobre el que se ejecuta el *remove*.

6.11.10. Utilización en un nuevo lenguaje

Para poder utilizar la arquitectura propuesta e implementada en un nuevo lenguaje, se deben realizar los siguientes pasos.

1. Extender la clase *NodesMetaInfo* y en ella definir la información necesaria para construir nuevos nodos del lenguaje correspondiente, teniendo en cuenta lo especificado en la sección 6.10.3.

2. Extender la clase *AbstractNodesSyntax*, y en ella definir la sintaxis de los nodos del lenguaje correspondiente, teniendo en cuenta lo especificado en la sección 6.11.5. Considerar la utilización de la clase *JavaNodesSyntax* implementada en el código fuente de la propuesta como guía para realizar este paso.

De ser necesario, se deben definir nuevos símbolos que permitan representar las semánticas del lenguaje. Para ello, tener en cuenta lo mencionado en la sección 6.11.11.

3. Implementar el método *getNodeSyntax* en un ancestro común a todos los nodos del lenguaje, tal y como se muestra en la sección 6.11.5 y en el fragmento de código 22.

4. Generar la estructura original del nodo previo a su manipulación a través de la ejecución del método *AbstractNode.syncRequired* en caso de que el método de manipulación no lo ejecute aún, tal y como se muestra en la sección 6.11.3.

5. Ejecutar el método *AbstractNode.syncIfRequired*, tal y como se muestra en la sección 6.11.3, para sincronizar las manipulaciones sobre los nodos en la cadena de tokens del código fuente, previo a su utilización.

6.11.11. Definición de nuevos ***StructureElement*s**

A la hora de definir nuevos elementos estructurales (que en definitvas son símbolos, según lo especificado en la sección 6.11.6 y en la figura 12) para poder utilizar la arquitectura en un nuevo lenguaje, tener en cuenta los siguientes items.

- Extender de una de las dos clases de *StructureElement*: *AbstractSingleMatchingStateSE* o *AbstractReentrantMatchingStateSE* según lo especificado en la sección 6.11.6. También es posible extender de *StructureElement* o *AbstractSE* directamente, teniendo en cuenta los detalles pertientes que se marcan en la sección mencionada sobre la cuestión de la representación de los elementos estructurales en el NFA subyacente al parser.

- Implementar los métodos *shouldMatch* y *doesMatch* para que obedezcan la lógica deseada, según lo explicado en la sección 6.11.7 (que depende del estado que representa al *StructureElement* en el NFA).

- Implementar el método *sync* para realizar la sincronización de las propiedades del nodo (accedidas a través del *SyncContext* y el resto de los parámetros del método) teniendo en cuenta los detalles de funcionamiento mencionados en la sección 6.11.8.

Por último, considerar la utilización de las implementaciones realizadas para Java de los símbolos de modificadores, imágenes y arrays (ver las clases *ModifiersSE*, *ImageSE* y *ArraySE* en el código fuente de la propuesta), así como también las implementaciones de los *StructureElement* genéricos: *ChildSE* y *AbstractSE*.

6.12. Anexo

Esta sección muestra implementaciones no incluidas como parte del release final de PMD pero no necesariamente desestimables. Todas las funcionalidades incluidas en esta sección fueron cortadas en su desarrollo debido a lo explicado en la sección 6.6.

Otro punto importante a mencionar que ninguno de los cambios mencionados posee Breaking API Changes, pudiendo entrar en una minor release.

6.12.1. Conversión de AST a Texto

La ultima opción que se consideró para traducir el AST a código fuente, mencionada en la sección 6.4, es recorrer un AST desde su primer hasta su último token, ambos campos existentes en la clase **AbstractNode**. Se considera el nodo a traducir como el raíz y no es necesariamente el nodo raíz para cualquier AST de ese lenguaje.

Para separar la lógica de representación de un AST se decidió no modificar la clase **AbstractNode** que posee todo el comportamiento genérico de la mayor parte de los nodos de todo lenguaje; de esta manera se permite implementar diferentes formas de representar un AST, incluyendo opciones de formato, sin que se requiera hacer una modificación sobre los nodos. Se creó una clase de utilidades **SourceCodeWriter**[14] que permite, a partir de un nodo, obtener su representación en texto. Ésto se logra recorriendo los tokens obteniendo en cada instancia la imagen del token y el puntero al próximo token. A su vez, se decidió incluir todos los tokens especiales (por ejemplo espacios o sus equivalentes, comentarios); como los tokens especiales están guardados de manera inversa, es decir, cada token especial apunta a su token previo, se deben almacenar los mismos en un stack y, finalmente, consumirlo obteniendo su imagen correspondiente.

También se implementó en la misma clase un salvado del AST a archivo que espera un nodo y el path al cual guardar. Para evitar sobreescribir el archivo ya existente previo a la finalización del AST, se decidió escribirlo en un archivo temporal y luego moverlo. Esto evita perder el archivo original en caso de una falla en la escritura.

[14] `https://bit.ly/2KmEzJG`

Luego durante el procesamiento de cada archivo en PMD, en el método *processFiles* de la clase **PMD**, se agregó una condición en la cuál si para el archivo a analizar está guardado en la cache, el análisis incremental está activado, es decir, se está utilizando la cache, y los autofixes están activados por flag de la línea de comandos, se desactivan, ya que en esta branch del proyecto no está implementado la cache para almacenar autofixes. Por otro lado, si el análisis incremental está activado pero ese archivo no fue procesado previamente, entonces se continua con el flujo normal de autofixes, es decir, se obtiene la representación en texto del AST al final del procesamiento y se guarda a un archivo.

La implementación completa se encuentra en el siguiente link: `https://github.com/gibarsin/pmd/tree/node-to-source-code`

6.12.2. Aplicación de fixes en el flujo de PMD

La implementación completa se encuentra en el siguiente link: `https://github.com/gibarsin/pmd/tree/store-rule-violation-fix`.

Independientemente de cómo se modelen los fixes en una clase, se requiere de un aplicador de fixes sobre el AST. Para ello se creó la clase **ASTAutoFixes** que almacena todos los fixes en una cola y los aplica cuando le es solicitado, en particular, cuando se terminan de procesar todas las violaciones de una regla; esto es visible en la siguiente línea de la clase **RuleSet**: `https://github.com/gibarsin/pmd/blob/store-rule-violation-fix/pmd-core/src/main/java/net/sourceforge/pmd/RuleSet.java#L503`. Como puede ocurrir que se estén revisando violaciones para diferentes archivos de manera concurrente, se decidió utilizar un mapa con el par *(Filename, FixesQueue)*. De esta manera se permite aplicar fixes a diferentes AST de manera concurrente.

6.12.3. Reporte de una violación con fix

La implementación completa compartida por la de aplicación de los fixes, se encuentra en el siguiente link: `https://github.com/gibarsin/pmd/tree/store-rule-violation-fix`. En PMD, al encontrar una violación, se realiza un reporte del mismo a través de un método heredado por todas las reglas que detectan violaciones, llamado *addViolation*. De manera tal que todas

las reglas posean una manera de reportar una violación con un fix, se agregaron métodos que poseen el mismo comportamiento que sus contrapartes sin arreglos, pero además se terminan reportando a la clase **ASTAutoFixes**, mencionado en la sección anterior de este anexo.

La clase abstracta **RuleViolationFix** se utilizó como medio para implementar el resto del flujo, pero no está modelado en esta branch el cómo se aplican los fixes al AST.

6.12.4. Cache/Análisis Incremental para fixes

Para almacenar los fixes que le corresponden a cada archivo en la cache, se utilizó la implementación de diff match patch de Google[15] que posee un funcionamiento similar al diff de Git. En particular se utilizó la generación de lo que son llamados Patches: a partir de dos textos, estos se comparan y se devuelve una lista de modificaciones que se poseen entre los dos. La idea es, guardar estos patches serializados para que en una futura ejecución de PMD, esos sean deserializados y aplicados al archivo original, en caso de que este no haya sido modificado.

Se agregaron dos métodos a la clase **AbstractAnalysisCache**: *getPaches* y *setPatches* en los cuales se obtienen los patches de un archivo indicado y se guardan los patches deseados de un archivo en la cache, respectivamente. En esta implementación no se conoce qué fix se corresponde con qué patch, sino que es la diferencia entre códigos luego de aplicados todos los fixes posibles para todas las violaciones de código.

Para evitar la inserción de dependencias adicionales, se decidió copiar la clase de utilidades del repositorio y realizar los cambios sintácticos necesarios para que posean sentido semántico en PMD.

6.12.5. Versionado

Debido a la cantidad de análisis y propuestas desechas según los problemas que se presentaban al momento de integrar las implementaciones a PMD, se había decidido pensar una propuesta sobre anotación de cambios experimentales o inestables, que intuitivamente permitiría introducir

[15] `https://github.com/google/diff-match-patch`

interfaces o métodos públicos pero de tal forma que introduzca la flexibilidad de poder realizar cambios sobre las mismas, aún en producción. Esta iniciativa ya había sido discutida previamente a nuestra dedicación (`https://github.com/pmd/pmd/issues/995`). Si bien se planteó una solución, debido a los objetivos reformulados (ver sección 6.6), se decidió no presentar la solución a la herramienta.

A continuación se presentan las anotaciones. También se puede encontrar el código en el siguiente link: `https://bit.ly/2MBDU3D`.

Experimental Nuevas APIs marcadas con esta anotacióna nivel clase o método están sujetos a cambios en cualquier tipo de release (patch, minor o major) o merge con el code base (branch *master* en el proyecto principal) y pueden ser modificados de cualquier forma (incluido pero no extensivo a comportamiento y firma) o incluso ser removidos.

Se recomienda de hacer uso de la API solamente si sos un desarrollador de esta y/o de la funcionalidad para las cuales puede utilizarse (i.e. consumidores de la API).

Si se decide que una API tenga su anotación removida, luego del release o merge al code base que incluye esa remoción no se puede volver a agregar, y un major release de la herramienta es necesaria para la aprobación de cualquiera de los cambios a ella.

Se recomienda realizar un cambio de anotación a Beta para asegurar a los consumidores de la API que esta no será modifcada o borrada, pero pueden presentar inestabilidades cuando es usada.

Como consecuencia de esta definición, cualquier cambio a esta API mientras su anotación exista no es un Breaking API change, significando que cualquier consumo, incluso aunque este consumidor no haga uso de la anotación, tendrá que ajustarse internamente a esos cambios.

Beta Nuevas APIs marcadas con esta anotación a nivel de clase o método se saben que no son estables cuando son utilizadas (propenso a bugs, issues conocidos, no puede ser probado completamente por depender de otra funcionalidad aún no introducida). Esto sirve como una palabra de advertencia de que utilizar esta API puede incurrir en bugs del código del consumidor.

Una API no necesita haber poseido la anotación previa de **Experimental** para utilizar esta anotación.

Si se decide que esta anotación puede ser removida, luego de un release o merge del codebase que incluye esa remoción, la anotación no puede ser agregada nuevamente a la API, implicando que la API es considerada estable.

Esta anotación debe ser utilizada con un criterio justificado y juzgado por otros contribuidores o dueños. Esto significa que no debe ser utilizado como motivo de vaguedad para realizar testing, por ejemplo.

Utilizar **Experimental** en vez de esta anotación si se sabe que la API puede cambiar de cualquier forma.

7. Conclusiones

Luego de haber estado más de un año realizando este proyecto, lo primero que destacamos es la experiencia adquirida en lo relacionado a la dificultad de idear e implementar una solución adaptada a un proyecto open source y en constante desarrollo y mantenimiento. Esto se explica sobre todo por el hecho de haber afrontado tantas dificultades para llegar a la arquitectura final, pasando innumerables veces por etapas de frustración, de desorientación, de tener que tirar más de tres cuartos del trabajo realizado (cuando no todo) para seguir investigando hasta finalmente lograr proponer una arquitectura que cumpliera con las premisas estipuladas al comienzo del proyecto y modificadas a lo largo del mismo (ver sección 4).

Otra experiencia adquirida fue la capacidad de analizar grandes proyectos con poca guía o sin guía siquiera, con el objetivo de entender el funcionamiento de una arquitectura determinada, de por qué se modelaron las cosas de cuál o tal forma, para luego poder extraer sus ideas fundamentales y con ellas construir la solución que resuelva la problemática que nos concierne. Este fue el caso de todas las herramientas analizadas a lo largo del proyecto, entre las que se mencionan en el informe: PMD, Eclipse, Intellij IDEA, JavaParser.

Además, si bien esto ocurrió principalmente durante la propuesta 1, se adquirió experiencia en la realización de aportes a una herramienta open source, con todo lo que eso implica: evitar breaking API changes, tratar de modificar la menor cantidad de código posible para no romper otras funcionalidades, acatar los procesos de contributing y review, etc.

Finalmente, y no por ello menos importante, nos llevamos una gran sorpresa al darnos cuenta de que muchos de los conocimientos adquiridos a lo largo de la carrera fueron puestos en práctica en este proyecto, permitiendo así el desarrollo de las diferentes etapas del mismo y pudiendo alcanzar los objetivos reformulados. Incluso, nos sorprendimos a nosotros mismos al aplicar conceptos de otros campos (como el proceso de transcripción del ADN) que también fueron adquiridos en la universidad. En este sentido, las materias que sentimos que nos permitieron realizar este proyecto, y sin las cuales hubiese sido prácticamente imposible su desarrollo, son (el orden es irrelevante): Programación Imperativa; Programación Orientada a Objetos; Estructura de Datos y Algoritmos; Ingeniería de software I y II; Autómatas, Teoría de Lenguajes y Compiladores; Introducción a la Bioinformática;

Analisis y Verificación de Modelos y Códigos.

8. Trabajo Futuro

A continuación se enumeran los items de los objetivos iniciales que en los objetivos reformulados no fueron realizados (aunque fueron tomados en cuenta al momento de diseñar la solución):

- Integrar la propuesta 5 (sección 6.11) a PMD (a través del proceso de PRs, agregando la suite de tests correspondiente).
- Agregar el flujo de uso de manipulación para soporte de fixes por reglas (ver anexo en la sección 6.12). Esto implica principalmente:
  - Actualizar la tabla de símbolos que mantiene PMD durante su ejecutción.
  - Agregar soporte para el uso de fixes a través del plug-in de PMD para Eclipse.
  - Agregar soporte de uso de métodos de manipulación a través de mecanismo similar a XPath.
  - Agregar sistema para actualizar reportes según los arreglos realizados.
  - Soportar análisis incremental (cache) para autofixes.
- Agregar soporte para dar estilo al nuevo código.

Referencias

[1] DNA Learning Center. *Transcription (Advanced) - 3D Animation.* URL: `https://www.dnalc.org/resources/3d/13-transcription-advanced.html`.

[2] Russ Cox. *Regular Expression Matching Can Be Simple And Fast.* URL: `https://swtch.com/~rsc/regexp/regexp1.html`.

[3] Andreas Dangel. *Página Oficial, PMD: Don't Shoot The Messenger.* URL: `https://pmd.github.io/`.

[4] Eclipse Foundation. *Document.java, Eclipse Documentation (versión Mars 4.5)*. URL: https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjface%2Ftext%2FDocument.html.

[5] Eclipse Foundation. *Página Oficial, Eclipse*. URL: https://www.eclipse.org/.

[6] Gonzalo Ibars Ingman. *TextEdit Design Decision. Eclipse Community Forums*. Oct. de 2017. URL: https://www.eclipse.org/forums/index.php/m/1773658/msg_1773658.

[7] Jetbrains. *Página Oficial, Intellij IDEA*. URL: https://www.jetbrains.com/idea/.

[8] Jetbrains. *PsiJavaFile.java, Código Fuente (commit 9559f7f)*. Jun. de 2016. URL: https://upsource.jetbrains.com/idea-ce/file/idea-ce-d00d8b4ae3ed33097972b8a4286b336bf4ffcfab/java/java-psi-api/src/com/intellij/psi/PsiJavaFile.java.

[9] Jetbrains. *Quick Fix, Intellij Platform SDK DevGuide*. URL: http://www.jetbrains.org/intellij/sdk/docs/tutorials/custom_language_support/quick_fix.html.

[10] N. Smith, D. van Bruggen y F. Tomassetti. *JavaParser: Visited*. Leanpub, oct. de 2017.

[11] N. Smith, D. van Bruggen y F. Tomassetti. *Página Oficial, Java Parser*. URL: http://javaparser.org/.

[12] Ken Thompson. «Programming Techniques: Regular Expression Search Algorithm». En: *Commun. ACM* 11.6 (jun. de 1968), págs. 419-422. ISSN: 0001-0782. DOI: 10.1145/363347.363387. URL: http://doi.acm.org/10.1145/363347.363387.