UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

HUNTER: UNA PLATAFORMA DE REINGENIERÍA PARA JAVASCRIPT

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

DIEGO ARIEL ANDRÉS ORELLANA GUTIÉRREZ

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
FEDERICO OLMEDO BERÓN
JUAN ÁLVAREZ RUBIO

SANTIAGO DE CHILE
2019

Resumen

En la ingeniería de *software* resulta clave entender la estructura del código de un proyecto para poder implementar de forma efectiva las funcionalidades requeridas por el cliente. Sin embargo, es común en la industria que el desarrollador deba enfrentarse a bases de código que le son desconocidas y que no cuentan con una documentación adecuada; lo cual entorpece los ciclos de desarrollo y aumenta los costos. La situación en la cual se debe lidiar con código cuyos desarrolladores ya no se encuentran disponibles resulta especialmente crítica. En tal caso, el programador debe realizar un proceso de reingeniería que resulta complejo y demoroso para poder satisfacer las exigencias dadas por el negocio.

Con el objetivo de abordar dicha problemática, en este trabajo de memoria se construye un primer prototipo de HUNTER, una plataforma de reingeniería para JavaScript que permite al desarrollador explorar bases de código de forma visual e interactiva. Durante la fase de implementación se resolvieron una serie de desafíos técnicos que permitieron aplicar varios conocimientos enseñados a lo largo de la carrera sobre distintos tópicos tales como lenguajes de programación, patrones de diseño y construcción de interfaces de usuario.

Más allá de los retos técnicos enfrentados a lo largo de la confección de la herramienta, se espera que HUNTER pueda asistir al programador en la comprensión de proyectos de JavaScript con los cuales no está familiarizado. Para ello, durante la construcción de la plataforma, se llevaron a cabo pilotos con ingenieros en la industria con la finalidad de recabar *feedback* temprano y enfocar el desarrollo de HUNTER. Además, mediante experimentos realizados por otros investigadores, se han obtenido resultados que preliminarmente apuntarían a la utilidad de la herramienta para la exploración de proyectos de JavaScript desconocidos.

De esta forma, como resultado del presente trabajo de memoria, se obtiene una primera versión de HUNTER que facilita la comprensión de programas de JavaScript. Asimismo, se proponen diversas mejoras para esta plataforma, tales como la presentación de visualizaciones para *frameworks* específicos y su extensión a otros lenguajes. Por otro lado, se plantea la interrogante de si la plataforma pudiera no solamente ayudar a la comprensión de programas desconocidos, sino también a un mejor entendimiento de un sistema con el que el desarrollador ya estuviese familiarizado.

Agradecimientos

A mi familia por el apoyo económico brindado durante todos estos años.

A mi profesor guía Alexandre Bergel por su infinita paciencia y por su excelente motor de visualización *Roassal*. A Martín Días por todas las sesiones de programación codo a codo que tuvimos juntos y los inapreciables conocimientos de *Smalltalk* transmitidos en ellas. A Santiago Vidal por la confección de los experimentos de prueba de HUNTER. A Ronie Salgado por su increíble motor de resaltado de sintáxis en *Pharo*. A todo el equipo de *Intelligent Software Construction laboratory* (ISCLab) por el intercambio de ideas (y de cerveza).

A *Adere.so* y *BotCenter* por haberme permitido desenvolverme profesionalmente los últimos tres años y poner en práctica los conocimientos adquiridos en esta carrera.

A mis amigos de *El Culto* y *Los Mágicos* por todos los buenos momentos compartidos y el inconmesurable apoyo emocional entregado durante este proceso.

Finalmente, en particular, quiero agradecer a Francisco Madrid por haberme facilitado esta plantilla de LaTeX.

A todas las mujeres que me han rechazado. Ellas se lo han perdido.

Tabla de contenido

Índice de tablas

Índice de ilustraciones

Capítulo 1

Introducción

Uno de los problemas más recurrentes en el desarrollo de *software* es la degradación de la calidad del código a través del tiempo. Ya sea producto de cambios de requisitos o de rotación en el equipo de desarrollo, el código de las aplicaciones va sufriendo cambios que aumentan su complejidad, teniendo como consecuencia una mayor dificultad de comprensión para el desarrollador y una mayor facilidad para introducir *bugs*. Este fenómeno ya ha sido objeto de estudio dentro de la ingeniería de *software* [17].

Una situación más extrema corresponde al trabajo en proyectos en donde ya no hay ningún programador original y no existe más documentación que los comentarios del código, los cuales muchas veces pueden ser insuficientes para la comprensión del programa o, peor aún, pueden no estar al día y entregar información errónea. Este problema también ha sido estudiado ampliamente en la literatura [7, 11].

En ambos escenarios es preciso llevar a cabo (en mayor o menor medida) un proceso de reingeniería para adaptar el *software* a nuevos requisitos o para solucionar fallas en su operación. Lo anterior implica un proceso de comprensión del programa, de las partes que lo componen y de las formas en que se relacionan. Esto no solamente para encontrar los lugares del código responsables de un comportamiento particular, sino que también para evaluar las distintas formas de lograr los cambios deseados y estimar el tiempo en realizarlos.

De este modo, resulta esperable que existan herramientas que asistan al desarrollador en la realización del análisis anterior. En efecto, existen herramientas para entornos y lenguajes como Java [3], la plataforma .NET [19] y Visual Basic [4]. Éstas permiten, entre otras cosas, hacer un análisis de dependencias entre distintas partes del código y visualizar distintos tipos de métricas, tales como el número de líneas de código o la complejidad ciclomática a cierto nivel de granularidad (clase, método, etc.) del programa.

Por otro lado, JavaScript (también conocido como ECMAScript gracias al nombre del estándar que lo regula [16]) carece de herramientas comparables; a pesar de ser uno de los lenguajes más populares [1, 24] y ubicuos en la actualidad, y de encontrarse en aplicaciones *web* tanto del lado del navegador como de servidor [12] y en programas de escritorio [15]. Es por ello que en el presente trabajo de memoria se propone atacar esta problemática mediante

la construcción de HUNTER (acrónimo recursivo[1] de **H**UNTER **U**seful and **N**ice **T**ool for **E**CMAScript **R**eengineering): una plataforma de reingeniería de aplicaciones de JavaScript.

1.1. Justificación del trabajo

Como se mencionó anteriormente, JavaScript es uno de los lenguajes más populares hoy en día. Luego, es de esperarse que exista una gran cantidad de código JavaScript en producción que no se encuentre en el mejor estado para adaptarse a los cambios que deben producirse en el *software* dada la naturaleza dinámica de los problemas que resuelven. Más aún, dadas las tendencias que se dan en el ciclo de vida del *software* [17], se puede prever que la cantidad de código de tales características vaya en aumento. De este modo, la construcción de HUNTER es un trabajo con el potencial de beneficiar a muchos proyectos y aplicaciones en todo el mundo, sobre todo al tener en cuenta que aún no existe una plataforma de reingeniería para JavaScript.

Si bien no existen plataformas completas de reingeniería para JavaScript, sí hay proyectos que apuntan a una dirección similar, aunque resulten insuficientes en su alcance: JSCity [23], Plato [25] y Madge [14]. Un análisis más detallado de estas herramientas puede encontrarse en la Sección 2.2.

Asimismo, se debe tener en consideración que JavaScript se ejecuta en un gran número de ambientes distintos —cada uno con sus particularidades— y que se encuentra presente en una cantidad no menor de variantes y dialectos: ECMAScript 5, el mínimo denominador común en cuanto a funcionalidades de los ambientes de JavaScript actuales [28]; ECMAScript 2015, una revisión mayor al lenguaje que incorpora características significativas como el uso de clases e iteradores [8]; JSX, variante de JavaScript que permite expresar fragmentos de páginas *web* dentro del mismo lenguaje [10]; TypeScript, superconjunto de JavaScript que añade verificación estática de tipos [22]; entre otros. De este modo, es preciso acotar qué variedades y ambientes de JavaScript soportará HUNTER para llevar a cabo el proyecto en el transcurso de dos semestres.

Por otro lado, la construcción de una plataforma de reingeniería representa un desafío que otorga la oportunidad de aplicar muchos de los conocimientos adquiridos a lo largo de la carrera: el uso de conceptos de lenguajes de programación para realizar análisis estáticos como apoyo a la labor de reingeniería; el empleo de metodologías de programación y patrones de diseño para facilitar la elaboración del código de la plataforma; entre otros. Además es necesario utilizar herramientas y bibliotecas de terceros para realizar tareas tales como el análisis sintáctico de los archivos fuentes de JavaScript y la creación de visualizaciones interactivas.

Finalmente se debe complementar lo anterior mencionando que el proyecto de construcción de HUNTER tiene un alcance delimitado y resultados esperados claros. Existen precedentes con la existencia de plataformas con un objetivo similar para otros lenguajes —si bien no se

[1] Un acrónimo recursivo es un acrónimo que se refiere a sí mismo. Un ejemplo conocido es el del proyecto de *software* libre GNU (**G**NU is **N**ot **U**nix).

pretende alcanzar equivalencia ni correspondencia completa en cuanto a funcionalidades.

1.2. Objetivos

1.2.1. Objetivo General

El objetivo general de este trabajo de memoria consiste en la elaboración de un prototipo funcional de HUNTER: una plataforma de reingeniería de aplicaciones escritas en JavaScript. Esta plataforma deberá facilitar la comprensión del código de las aplicaciones (de las partes que lo componen y sus relaciones), especialmente para programadores que no estuvieron involucrados en el desarrollo. Por otra parte, la plataforma también deberá facilitarle a los desarrolladores la identificación de módulos y componentes más importantes en el proyecto.

1.2.2. Objetivos Específicos

- Realización de un análisis sintáctico de los archivos de código fuente en un proyecto de JavaScript, valiéndose potencialmente de otros programas o bibliotecas.
- Obtención de un modelo de las partes constituyentes de los programas (por ejemplo: clases, funciones, etc.) y las relaciones que se forman entre ellas a partir de la información sintáctica obtenida en la fase anterior.
- Generación de visualizaciones e interfaces gráficas que faciliten la comprensión del código.
- Validación de la utilidad de la plataforma con proyectos reales en JavaScript.

1.3. Metodología

1. Estudio previo
   En esta etapa se realizó un estudio previo sobre las herramientas, bibliotecas y programas existentes para la elaboración de la plataforma. También se delimitaron los dialectos y ambientes soportados de JavaScript en función de las herramientas existentes.
2. Obtención del árbol de sintaxis abstracto
   Con lo obtenido en la etapa anterior, se tuvo que construir una primera versión de la plataforma, la cual fue capaz de extraer el árbol de sintaxis abstracto (AST, por sus siglas en inglés) y lo dejó preparado para su posterior análisis y procesamiento.
3. Definición y extracción de metamodelos
   Se definieron los primeros metamodelos que representan las entidades más esenciales del lenguaje (tales como archivos, variables, funciones y clases) para un proyecto de JavaScript (por medio del AST obtenido en el paso previo).

4. Creación de visualizaciones
   Se definieron y crearon visualizaciones para ayudar al desarrollador a entender la información capturada en el modelo construido a partir del AST.
5. Validación y mejoras a los metamodelos y visualizaciones
   Una vez implementados los metamodelos y visualizaciones más básicas, se evaluaron posibles mejoras a realizar. Para ello se trabajó en conjunto con el profesor guía por medio de reuniones periódicas y también se recogieron sugerencias de otros alumnos tanto de pregrado como de postgrado y de ingenieros en la industria.
6. Estudio de bases de código reales con la plataforma
   Con el objetivo de validar la robustez y escalabilidad de HUNTER se realizaron estudios en bases de código reales de JavaScript.
7. Experimentos con desarrolladores
   Posteriormente se llevaron a cabo experimentos con otros desarrolladores para validar y entender mejor la manera en que HUNTER ayuda a la comprensión del código.
8. Escritura del documento de memoria
   Se trabajó en la redacción del documento de la memoria. Se efectuaron reuniones periódicas con el profesor guía para obtener *feedback*.

1.4. Resultados obtenidos

Como resultado de este proyecto se ha construido una aplicación que permite explorar proyectos de JavaScript de forma visual (la cual se encuentra disponible en `https://github.com/dorellang/hunter`). Si bien hay varios aspectos en los que la herramienta podría mejorar; HUNTER ha sido evaluado de forma positiva en el proceso de validación. De esta manera, puede afirmarse que el objetivo principal ha sido cumplido. Por último, cabe mencionar que el desarrollo de esta memoria no solamente ha abierto el espacio para un trabajo futuro en la herramienta misma, sino que también ha otorgado la oportunidad de que otros investigadores pudieran examinar de forma rigurosa el cómo HUNTER ayudaría a la comprensión de código.

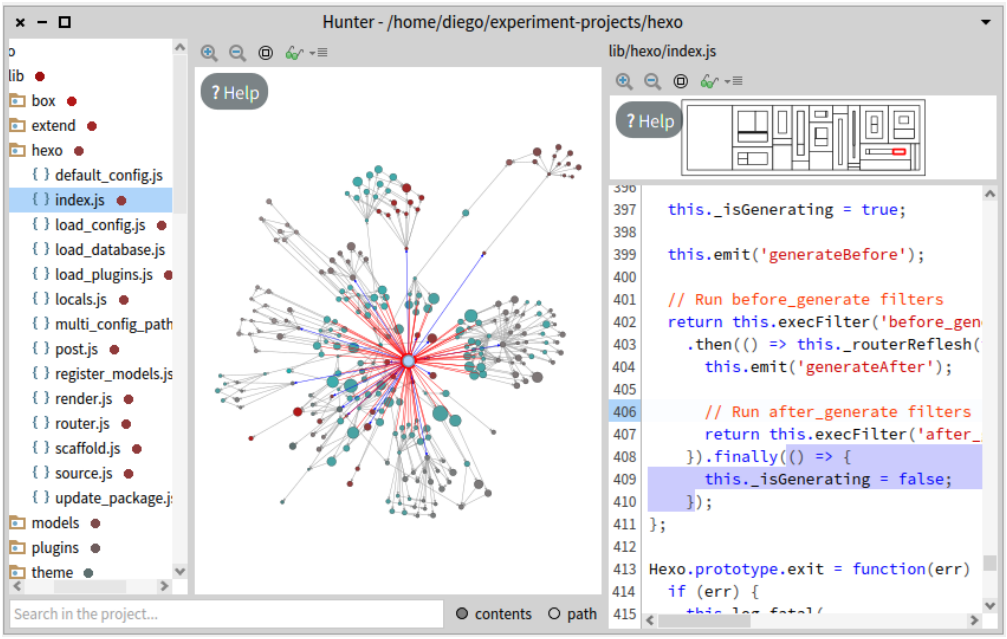

Figura 1.1: Captura de pantalla de HUNTER en funcionamiento.

1.5. Estructura del documento

Este informe posee la siguiente estructura. El Capítulo 2 presenta las aplicaciones ya existentes con un objetivo similar o relacionado a HUNTER, tanto para otros lenguajes como específicas para JavaScript. El Capítulo 3 describe el diseño de HUNTER, las tecnologías involucradas en su desarrollo y detalla el proceso de análisis realizado a los proyectos revisados en la plataforma. El Capítulo 4 documenta el proceso de construcción de la interfaz de usuario y las decisiones tomadas; además del resultado final y las visualizaciones construidas. El Capítulo 5 describe el proceso de validación seguido para evaluar la utilidad de HUNTER. Finalmente en el Capítulo 6 se recapitula brevemente sobre los resultados obtenidos, se discute las lecciones aprendidas y se propone posibles líneas de trabajo a futuro relacionadas con esta memoria.

Capítulo 2

Trabajos existentes

2.1. Estado del arte

Dentro de las herramientas de reingeniería halladas en general (no limitándose únicamente a JavaScript), éstas fueron las soluciones más completas encontradas:

- NDepend para la plataforma .NET
- VBDepend para Visual Basic
- JArchitect para Java
- CPPDepend para C/C++

Se observó que las capacidades ofrecidas entre las distintas herramientas eran similares, por lo que en esta sección se discutirán las características ofrecidas, en vez de enfocarse en las aplicaciones una por una.

2.1.1. Estimación de deuda técnica

Un escenario recurrente en el desarrollo de *software* es la decisión de resolver los requisitos del cliente con soluciones que —si bien logran responder en el corto plazo a los requerimientos— finalmente incurrirán en un costo de desarrollo adicional; ya sea al dificultar futuras modificaciones al programa o al no cumplir con los requisitos de calidad, escalibilidad o rendimiento que serán impuestos a futuro en el negocio. En 1992, Ward Cunningham acuñó el término de *deuda técnica* para referirse a estos costos [6]. Más aún, veinte años más tarde, en el 2012, se presentó la metodología SQALE[1] para medirlos numéricamente [18].

[1] **S**oftware **Q**uality **A**ssessment based on **L**ifecycle **E**xpectations. Es una metodología para medir la calidad de proyectos de *software*. Para ello define dos modelos: el SQALE Quality Model —para expresar las reglas con las que el código debe cumplir para no agregar *deuda técnica*, y que dependen del lenguaje de programación— y el SQALE Analysis Model —que normaliza y consolida las violaciones a las reglas para generar una medida númerica.

Las herramientas anteriormente mencionadas (con la excepción de VBDepend) permiten calcular la deuda técnica de proyectos de software mediante SQALE. En particular, JArchitect permite, entre otras cosas:

- Personalizar la medición de deuda técnica con fórmulas personalizadas en LINQ (acrónimo de *Language Integrated Query*, sintáxis similar a SQL integrada dentro de C# para la construcción de consultas).
- Comparar la deuda técnica actual contra la de una versión base del proyecto.
- Personalizar la presentación de la deuda técnica.

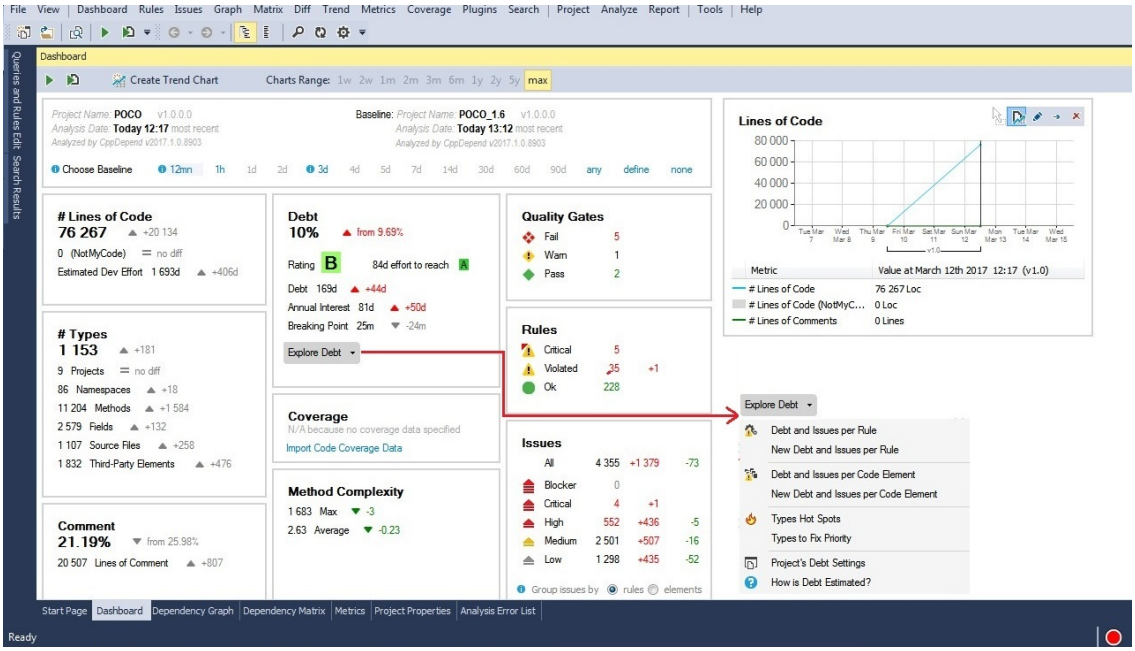

Figura 2.1: Dashboard de deuda técnica en JArchitect.

2.1.2. Métricas de calidad de *software*

Los programas mencionados también incluyen un amplio abanico de métricas de calidad de *software* que pueden ser usadas tanto en visualizaciones, así tambien como en búsqueda de código y en reglas de estilo para lanzar advertencias. Algunas de las métricas manejadas por estas herramientas son:

- Líneas de código efectivo
- Líneas de comentario
- Porcentaje de comentarios
- Número de métodos
- Número de variables de instancia
- Número de tipos
- *Coupling* aferente. Las dependencias entrantes; es decir, dado un paquete, la cantidad de clases en otros paquetes que dependen en las clases del primero.

- *Coupling* eferente. Las dependencias salientes; esto es, dado un paquete, la cantidad de clases ajenas a ese paquete en las que dependen las clases del mismo.
- Complejidad ciclomática. Introducida en 1976 por Thomas McCabe [20], es una medida de la complejidad de una función cuyo valor es proporcional a los puntos de decisión que posee (que en JavaScript, a *grosso modo*, corresponderían a la ocurrencia de construcciones sintácticas como `if`–`else`, `for`, `while`, `do`–`while` y de los operadores booleanos `&&` y `||`).

2.1.3. Análisis y visualización de dependencias

Asimismo estos programas permiten analizar las dependencias entre los distintos componentes de código de forma visual mediante grafos y matrices de dependencias, como se puede apreciar en la Figura 2.2 y Figura 2.3 respectivamente.

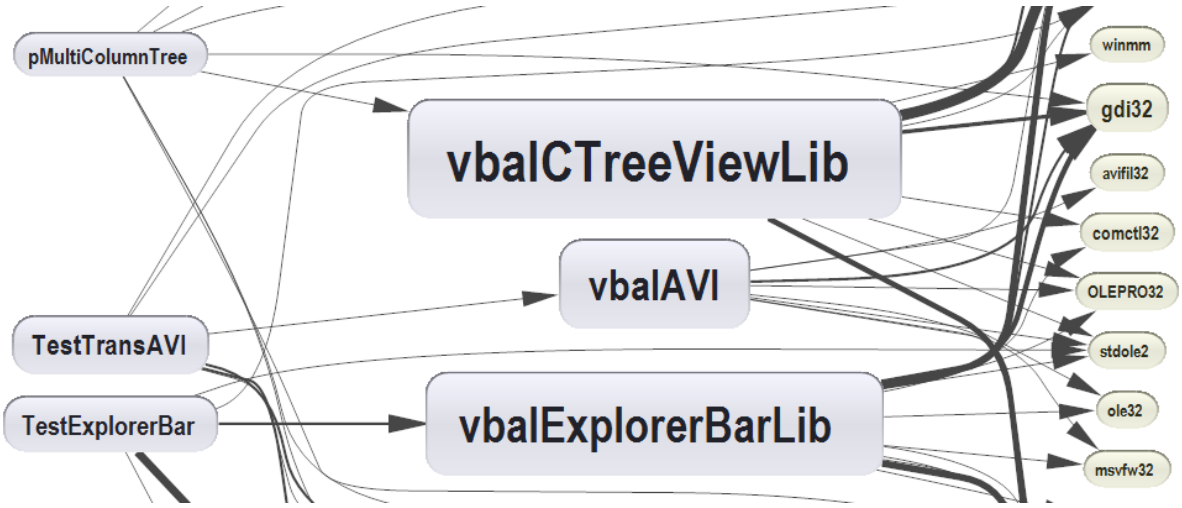

Figura 2.2: Fragmento del grafo de dependencias generado por VBDepend.

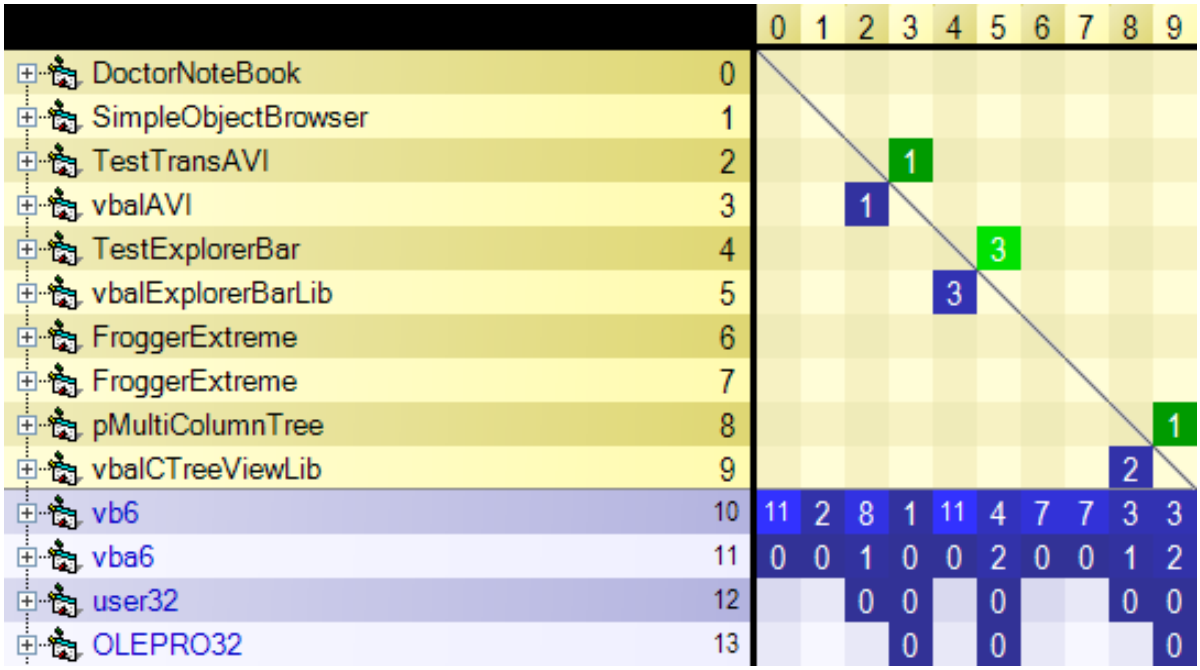

Figura 2.3: Matriz de dependencias generada por VBDepend.

2.1.4. Inclusión de *test coverage*

Los *tests* unitarios son de suma importancia al momento de trabajar en bases de código grandes. Estos permiten que el código evolucione de forma rápida, al reducir las posibilidades de introducir regresiones. De este modo, estas herramientas (con la excepción de VBDepend) poseen la capacidad de importar los datos de *test coverage* obtenidos al ejecutar la *suite* de *tests* unitarios. En particular, la Figura 2.4 muestra una visualización de los datos de *coverage* en NDepend.

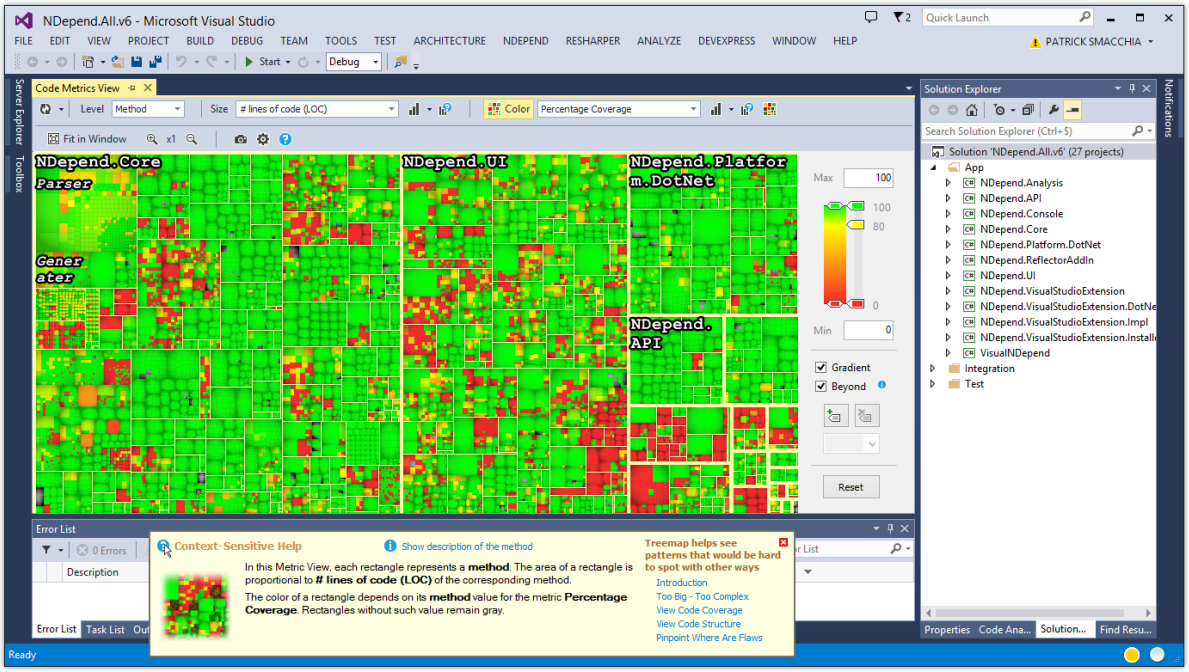

Figura 2.4: Uso de la visualización de TreeMap en NDepend para gráficar los porcentajes de *coverage* importados.

2.1.5. *Queries* de código

Todas estas herramientas poseen la capacidad de hacer consultas sobre el código usando la sintaxis CQLinq (Code Query over LINQ). Esta funcionalidad permite encontrar métodos, clases o funciones dentro del código que cumplen ciertas características o que poseen ciertas métricas, como se puede apreciar en la Figura 2.5. También esta capacidad permite crear reglas personalizadas para lanzar advertencias como se puede apreciar en la Figura 2.6.

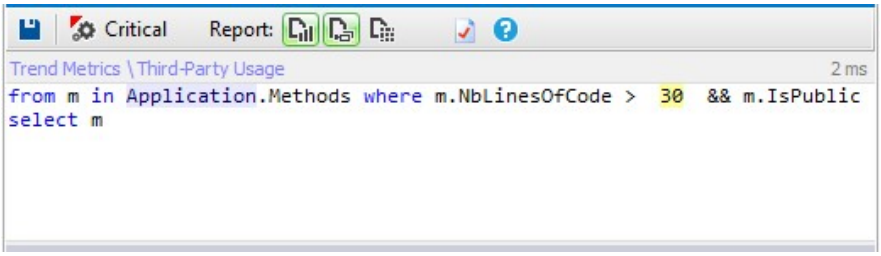

Figura 2.5: Query para hallar métodos públicos de más de 30 líneas en VBDepend.

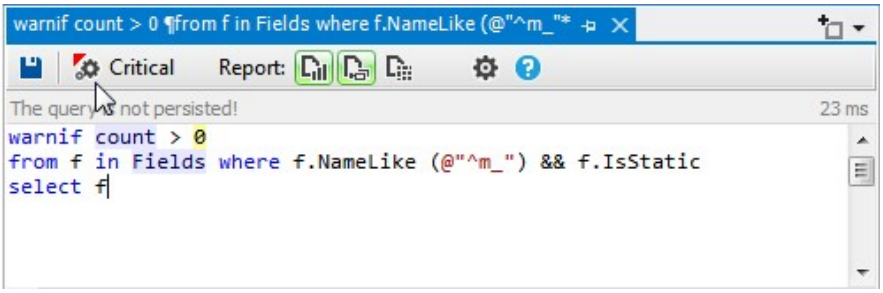

Figura 2.6: Definición de regla para advertir al usuario si no se respetan las convenciones de nombramiento de métodos estáticos en JArchitect.

2.1.6. Generación de reportes personalizados

Por otra parte, para facilitar la visión global del código de un proyecto, las plataformas de análisis mencionadas también soportan la generación de reportes personalizados en formato HTML. Las métricas y secciones a mostrar son configurables por el usuario. La Figura 2.7 ilustra esta característica.

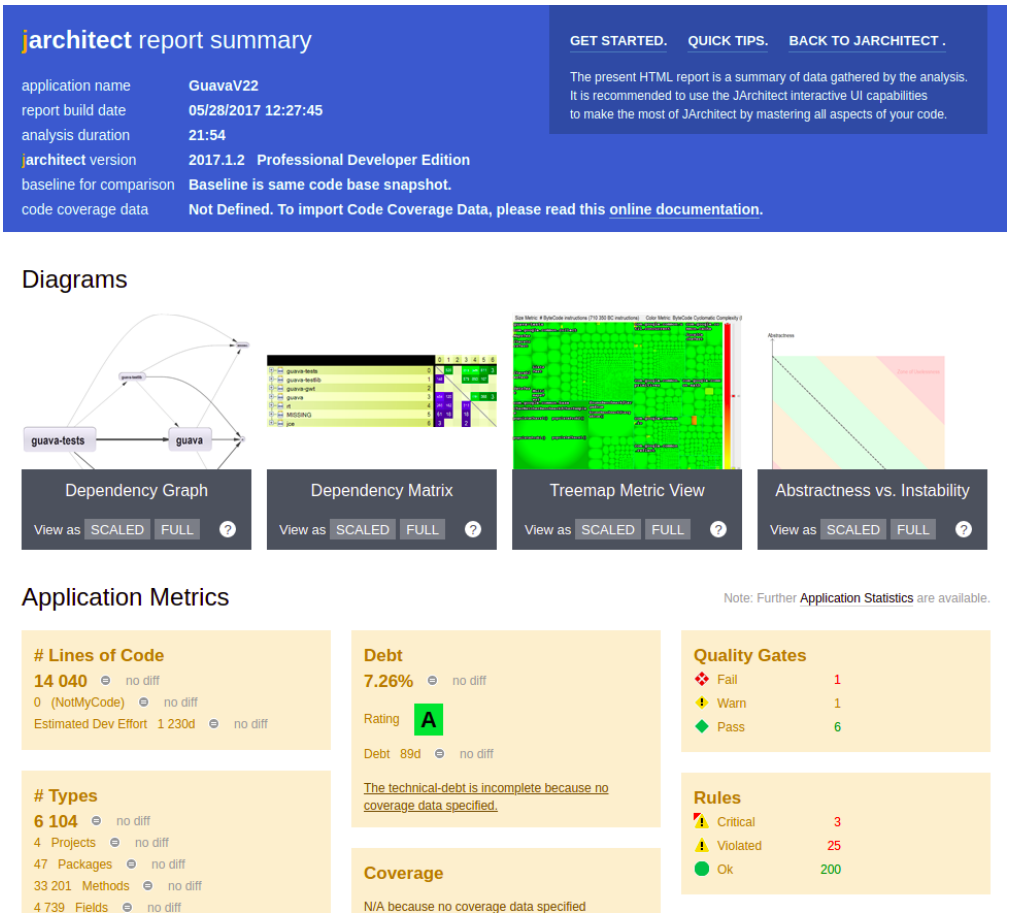

Figura 2.7: Captura de pantalla de reporte HTML generado en JArchitect.

2.1.7. Monitoreo de métricas en el tiempo

Finalmente, otra funcionalidad común es el monitoreo de la evolución de métricas de *software* a través del tiempo. La Figura 2.8 muestra esta característica en funcionamiento para NDepend.

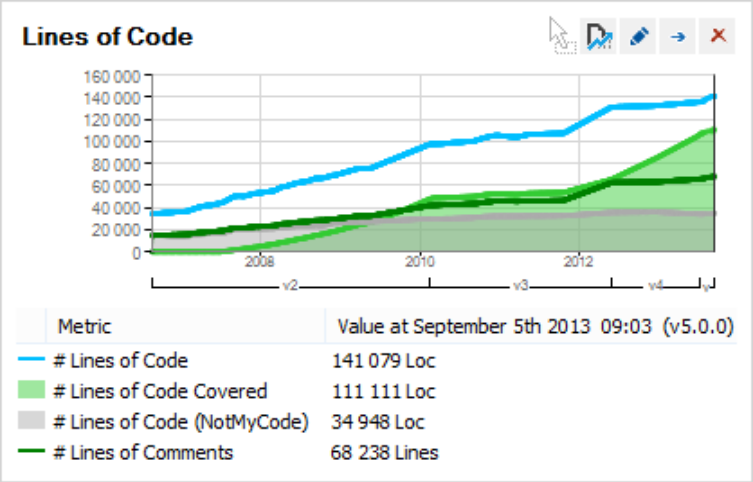

Figura 2.8: Visualización de la evolución de las lineas de código de un proyecto para la plataforma .NET a través del tiempo.

2.2. Herramientas disponibles para JavaScript

A pesar de lo sofisticado de las aplicaciones mencionadas en la sección anterior, al explorar las herramientas para JavaScript las opciones son mucho más acotadas. En particular en esta sección se revisarán las herramientas más notables encontradas por el autor en términos de su utilidad para la reingeniería de proyectos.

2.2.1. JSCity

JSCity es una versión para JavaScript de la visualización tridimensional de código llamada CodeCity [27] que aplica la metáfora de ciudad para representar las partes del programa: los directorios forman distritos, los archivos son subdistritos y las funciones son representadas como edificios; tal y como se aprecia en la Figura 2.9.

A pesar de la utilidad de la visualización generada —que permite identificar rápidamente funciones que deben ser refactorizadas—, ésta es rígida y poco flexible: no se pueden observar más métricas, ni más datos que la cantidad de variables y líneas de código de una función; en particular, no se pueden apreciar dependencias de ningún tipo. Otra limitación de la visualización es la imposibilidad de poder explorar el código fuente. Finalmente, cabe mencionar que JSCity tampoco posee soporte para la sintaxis y funcionalidades más nuevas añadidas a JavaScript en el estándar ECMAScript 2015 (también conocido como ECMAScript 6).

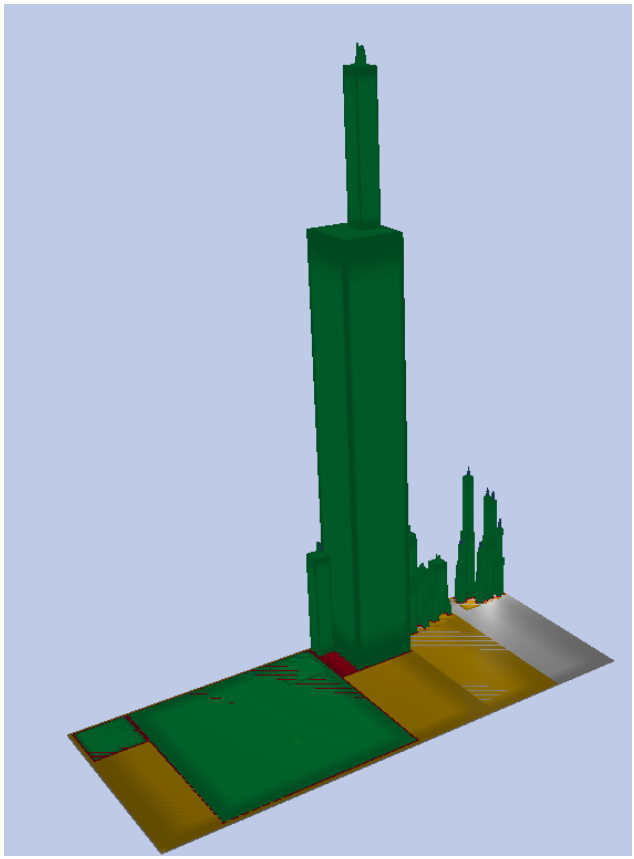

Figura 2.9: Visualización del proyecto ChartJS en JSCity. Los colores de los edificios indican el tipo de entidad al que corresponde el edificio (proyecto, carpeta, archivo o función). Por otra parte la altura de los edificios está correlacionada con las líneas de código y el área de la base con la cantidad de variables que posee la entidad.

2.2.2. Plato

Plato es un generador de reportes de calidad de código de un proyecto mediante el análisis estático de los ficheros fuentes. Dentro de sus características, permite obtener métricas como líneas de código y complejidad ciclomática para cada archivo y función. Además se integra con otras herramientas de análisis estático como JSHint y ESLint para encontrar posibles errores de estilo o de programación.

Dentro de sus ventajas se encuentra la posibilidad de incorporar fácilmente la generación de los reportes a un *pipeline* de integración continua; y que entrega tanto una vista global —con métricas del proyecto entero (ver Figura 2.10)— como otra local por archivo —con métricas sólo considerando ese fichero (ver Figura 2.11)—. También permite revisar el código fuente de la aplicación dentro del mismo reporte.

Sin embargo, Plato no permite (ni pretende) llevar a cabo análisis de dependencias entre módulos de un proyecto; los cuales son necesarios para la comprensión del código en la labor de reingeniería. Si bien presenta tanto métricas locales como globales del proyecto analizado, no facilita la visualización de la estructura lógica de la aplicación en cuestión.

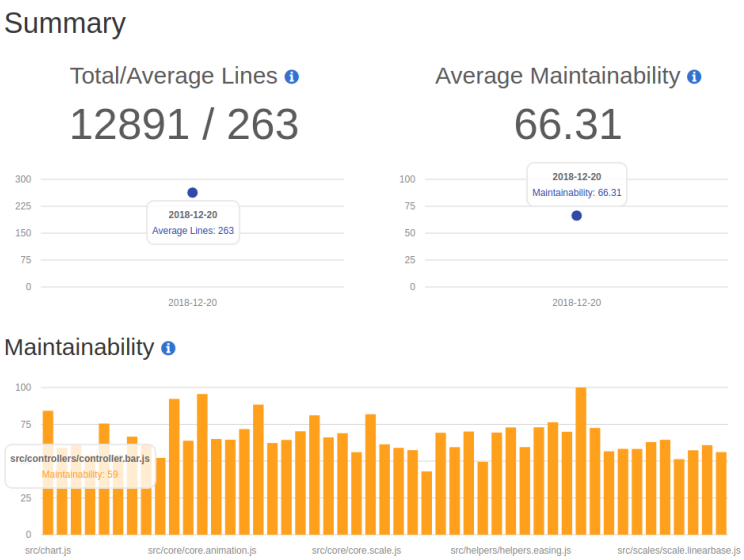

Figura 2.10: Vista general del reporte creado por Plato.

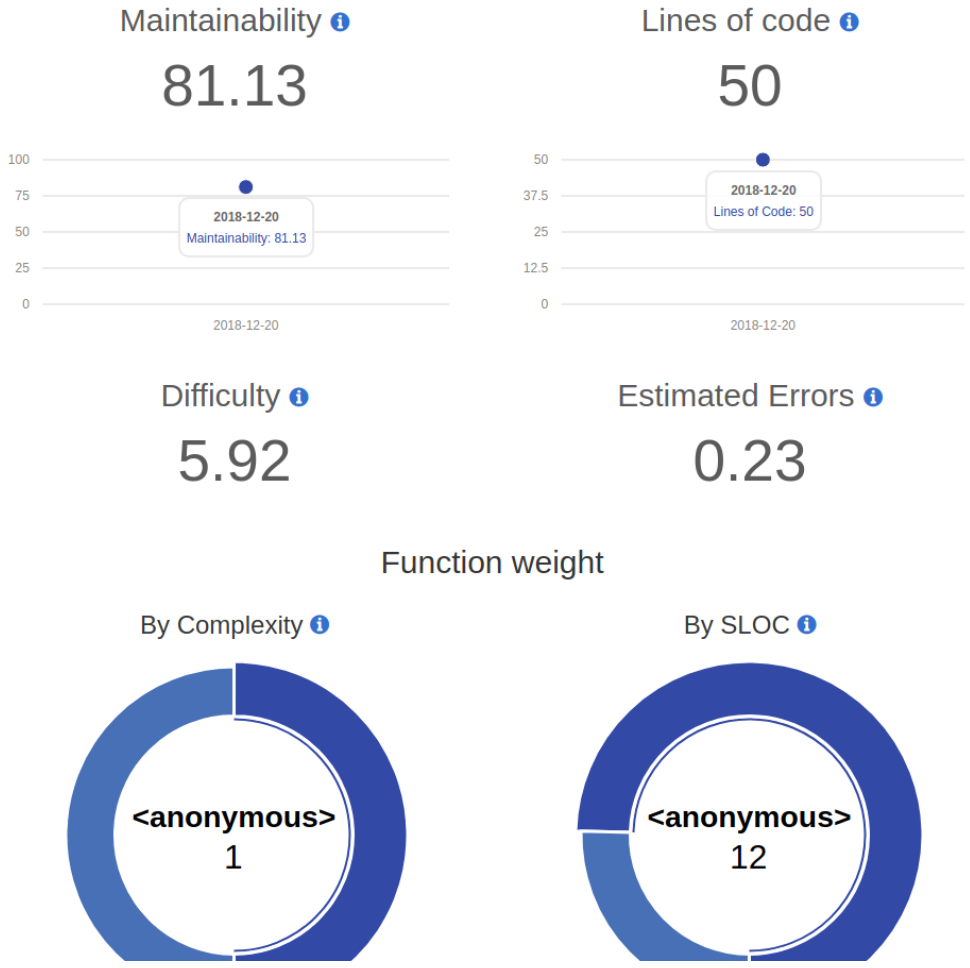

Figura 2.11: Vista por archivo del reporte creado por Plato.

2.2.3. Madge

Madge es una herramienta de análisis de dependencias para proyectos de JavaScript escrita para la plataforma de NodeJS. Madge está principalmente pensado para ser usado mediante línea de comandos. Así pues, la versión 3.3 de Madge permite llevar a cabo los siguientes análisis:

- Consultar los archivos dependientes de otro.
- Cantidad de dependencias por archivo.
- Generación de imagen archivo SVG que muestra visualmente el grafo de dependencias del proyecto.

Además Madge se integra con Webpack, herramienta extensible de *bundling* de módulos de JavaScript. El principal caso de uso de Webpack corresponde a la exportación de proyectos escritos en JavaScript a un conjunto fijo de *scripts* listos para ser incluídos en una página web (idealmente optimizados para una carga rápida). La integración de Madge con Webpack permite la correcta deducción de las dependencias incluso en aquellos proyectos que configuran Webpack con variantes no estándar de EcmaScript, o con lógicas personalizadas de *lookup* de archivos de código fuente dentro de una sentencia `import` o `require`.

Asimismo, adicionalmente a los comandos expuestos en la consola, Madge entrega una API de JavaScript disponible como una biblioteca de NodeJS; lo cual permitiría ocupar sus capacidades de análisis dentro de otras herramientas escritas en NodeJS.

Sin embargo, a pesar de las potentes capacidades de Madge, es claro que no pretende ser una plataforma de reingeniería completa; particularmente puede decirse que sólo expone funciones y comandos —las que, por separado, podrían a ayudar a un desarrollador a entender la estructura de un proyecto, pero que no constituyen un ambiente de reingeniería en sí mismo—. Además las visualizaciones que puede generar no son interactivas; lo que entorpece la exploración libre de un proyecto desconocido para un programador.

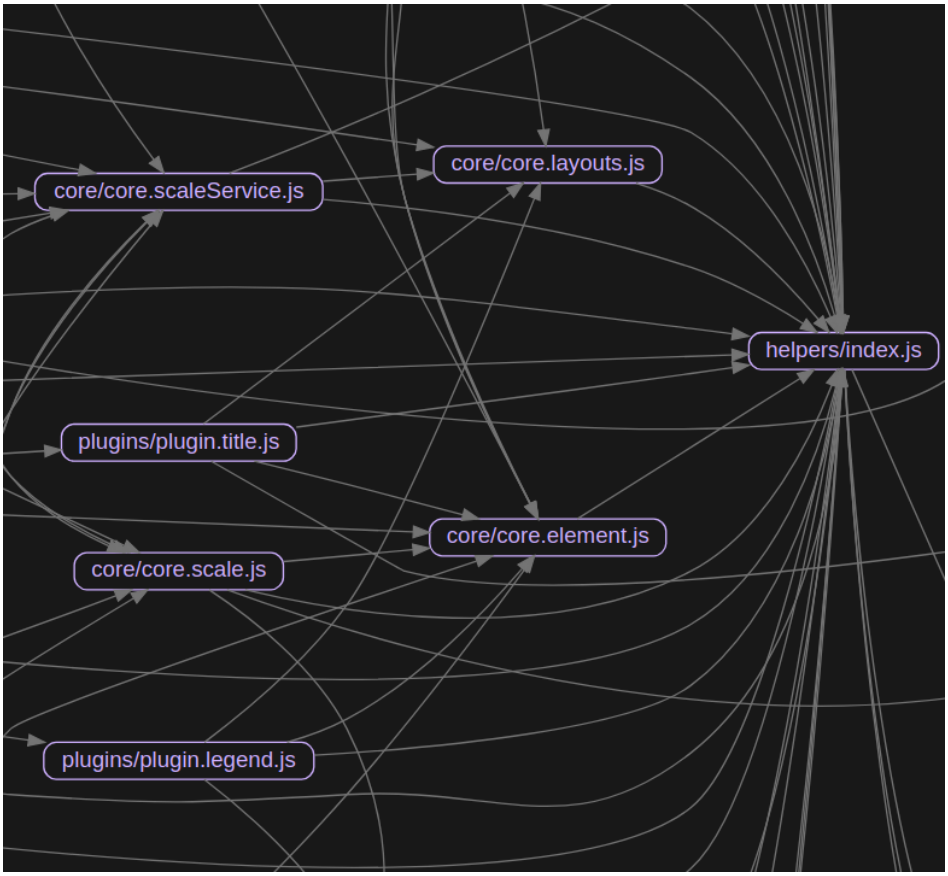

Figura 2.12: Porción del archivo de imagen vectorial SVG generado por Madge para el proyecto ChartJS.

2.3. Recapitulación

Se han revisado las aplicaciones ya existentes de análisis de proyectos de *software* que ayudan a mejorar el entendimiento del desarrollador sobre el programa en cuestión. Así, se concluye que, si bien existen plataformas muy sofisticadas y completas para inspeccionar *software* escrito en distintas plataformas como Java y .NET, ninguna de las alternativas encontradas para JavaScript constituye un ambiente de reingeniería completo.

Con ello y tomando en cuenta que:

- La metodología SQALE de estimación de deuda técnica no es trivial, tanto en su procedimiento, como en la selección de los patrones del código a penalizar.
- Implementar un sistema sólido de *queries* al estilo de CQLINQ utilizado en herramientas como NDepend resulta complejo.
- Existen indicios de que las métricas de calidad de *software* complejas no necesariamente entregan un valor adicional considerable por sobre la medición de líneas de código [21].
- Modelar las entidades del código (funciones, clases, etc.) a través del tiempo resulta complejo e infactible de llevar a cabo en el período de desarrollo dado para una memoria.
- El foco de este trabajo es el análisis estático de código.

Se ha decidido que HUNTER debe enfocarse principalmente en atacar la problemática de visualización de dependencias, siguiendo la línea de lo analizado en la Sección 2.1.3 (Análisis y visualización de dependencias) y en la Sección 2.2.3 (Madge); pero, a la vez, proponiendo un enfoque exploratorio que permita al usuario inspeccionar bases de código mediante la interacción con visualizaciones. En este sentido, se espera que HUNTER pueda ser una mejora concreta a las alternativas ya existentes para JavaScript.

Capítulo 3

Arquitectura de HUNTER

En este capítulo se describen a grandes rasgos los lenguajes y bibliotecas utilizados dentro de HUNTER. Además, se detallan las etapas del proceso que se lleva a cabo para ayudar al desarrollador en la labor de reingeniería; a saber:

1. Lectura de los archivos de JavaScript en un proyecto para generar un AST.
2. Realización de un análisis de varios pasos para la extracción de información y la construcción de un metamodelo a partir del AST generado en el paso anterior.
3. Presentación del metamodelo a través de una interfaz de usuario con visualizaciones interactivas.

Las interfaces y visualizaciones, al tratarse de tema complejo y extenso, se discutirán en detalle en el Capítulo 4.

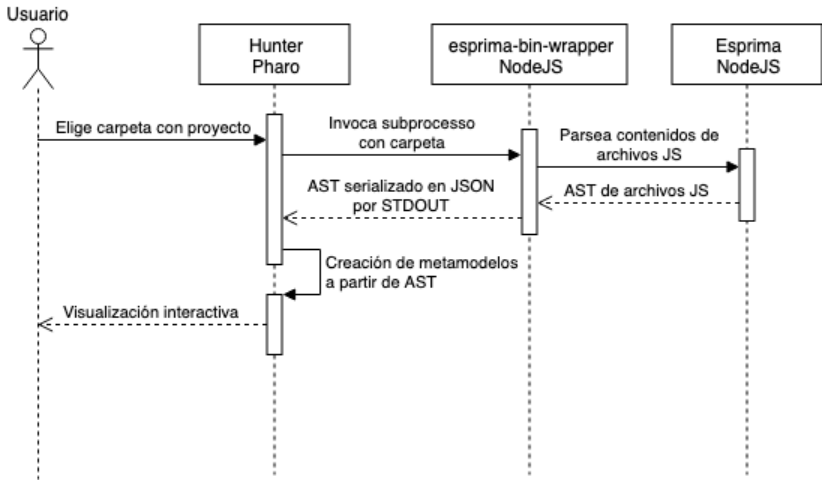

Figura 3.1: Diagrama de secuencias que muestra la arquitectura general de HUNTER. Los distintos componentes aparecen modelados en este diagrama como líneas de vida etiquetados con el nombre del componente y la plataforma en que se desarrollaron en una segunda línea.

3.1. Tecnologías usadas

En concordancia con lo visto en la Figura 3.1, dentro de HUNTER se han empleado varias herramientas, plataformas y bibliotecas. Particularmente se pueden distinguir dos grandes componentes de código: una escrita en Pharo Smalltalk y otra escrita en JavaScript, la cual se ejecuta en la plataforma de NodeJS.

3.1.1. Pharo

Para la construcción de la herramienta de HUNTER se ha decidido usar el ambiente de *live programming* de Pharo, un dialecto moderno de Smalltalk. Características tales como la inspección de cualquier expresión del lenguaje mientras se depura un error o la posibilidad de poder evaluar pequeños fragmentos de código dentro de un método mientras se edita han permitido agilizar el desarrollo de HUNTER e iterar rápidamente.

Por otra parte, para la publicación y versionamiento del proyecto se exporta el código mediante la herramienta Iceberg, la cual permite de forma transparente almacenar programas de Pharo en un repositorio del sistema de control de versiones de Git.

Asimismo, para el diseño de las visualizaciones se ha usado el *framework* para Pharo de visualizaciones de Roassal. Dicho *framework* cuenta con un lenguaje de dominio específico sencillo que permite construir visualizaciones de forma ligera y experimentar con ellas valiéndose del entorno de Pharo. En tanto para la construcción de la interfaz de usuario, se utilizó la biblioteca de Spec, la cual permite confeccionar interfaces gráficas de forma modular en Pharo, separando los elementos gráficos desplegados en componentes. Por otro lado, con el objetivo de mantener el estado de todos las componentes en sincronía, después de manejar alguna interacción del usuario, se elaboró un sistema de propagación de cambios. Más detalles sobre la construcción de la interfaz de usuario pueden encontrarse en el Capítulo 4.

Finalmente, para interactuar con la subaplicación escrita en NodeJS se ocupa la biblioteca de OSSubprocess [26], la cual permite invocar otros programas como subprocesos y obtener el resultado que imprimen en la salida estándar de forma asíncrona por medio de una interfaz de *stream* de caracteres.

3.1.2. NodeJS

La componente escrita en NodeJS corresponde principalmente a un *script* ejecutable. Este *script* recibe como parámetro de la línea de comandos uno o varios directorios a analizar e imprime en la salida estándar un archivo JSON con todos los archivos procesados con el AST respectivo si el *parsing* fue exitoso o un *stub* indicando error de haber problemas al procesarlo.

Para la construcción de este *script* se usaron las siguientes bibliotecas:

- Esprima para hacer *parsing* de los archivos fuente de JavaScript.
- Yargs para leer de forma fácil los parámetros de ejecución entregados al *script*.
- JSONStream para serializar los nodos del AST arrojados por Esprima. En una primera instancia se intentó con la función nativa de JavaScript `JSON.stringify()`. Sin embargo, la serialización fallaba con un error de memoria insuficiente al momento de procesar proyectos demasiado grandes. Esto se debió a que la función `JSON.stringify()` crea un *string* en memoria que representa el objeto serializado entero; en contraste con las funciones de JSONStream, las cuales permiten generar e imprimir la serialización JSON por segmentos.

3.2. Proceso de análisis sintáctico

3.2.1. Extracción del AST

Para lograr la extracción del AST de los programas de JavaScript se consideró en una primera instancia el uso del *parser* de JavaScript incluido dentro de SmaCC, un generador de *parsers* escrito en Pharo. Sin embargo el programa presentaba varias falencias que impedían su uso en la práctica:

- El nombre reservado para identificadores *catch* está permitido como el nombre de la propiedad de un objeto. En efecto la interfaz de promesas definida en el estándar ES2015 contiene un método con ese nombre [5]. El parser de SmaCC no manejaba este caso y fallaba en el código que usaba dicho método.
- La sintáxis de JavaScript permite la aparición en el código fuente de caracteres de espacio en blanco distintos de los caracteres ASCII usualmente utilizados (0x09 para tabulación horizontal y 0x20 para espacio). El parser incluido en SmaCC no maneja estos casos de forma adecuada y arrojaba un error al procesar el código fuente.

Si bien en teoría era posible modificar el *parser* de SmaCC, se terminó optando por otra solución ya existente para obtener resultados concretos cuanto antes y comenzar a iterar sobre ellos. Es por ese motivo que se ha decidido usar Esprima, un *parser* de JavaScript escrito en el mismo lenguaje; aunque resultara más difícil de integrar con el resto del programa escrito en Smalltalk.

Para lograr lo último, Esprima es invocado desde la componente de NodeJS de HUNTER, la cual a su vez es lanzada como subproceso mediante la biblioteca de OSSubprocess por la componente principal escrita en Pharo. Este subproceso escribe en la salida estándar un documento en formato JSON que luego es deserializado directamente con la biblioteca NeoJSON en objetos de Pharo, los cuales representan nodos del AST del proyecto.

La biblioteca Esprima ha resultado ser sólida y no ha presentado problemas para analizar el código de los proyectos con los que se ha experimentado hasta el momento.

3.2.2. Definición de Modelos y Análisis del AST

Una vez obtenido el AST del código de JavaScript, el siguiente paso es la extracción de los metamodelos que representan entidades dentro del código fuente para generar las visualizaciones. Actualmente la plataforma de HUNTER define los siguientes metamodelos:

- **Proyecto**. Corresponde al proyecto completo analizado en HUNTER. Modelado en la clase `HNProject`.
- **Programas**. Representa los archivos de código de JavaScript hallados en el proyecto en HUNTER. Modelado en la clase `HNProgram`.
- **Clases**. Clases de ECMAScript 2015 encontradas en el código, representadas por la clase `HNClass` dentro de HUNTER.
- **Funciones**. Representadas por la clase `HNFunction`.
- **Variables**. Representadas por la clase `HNVariable`.

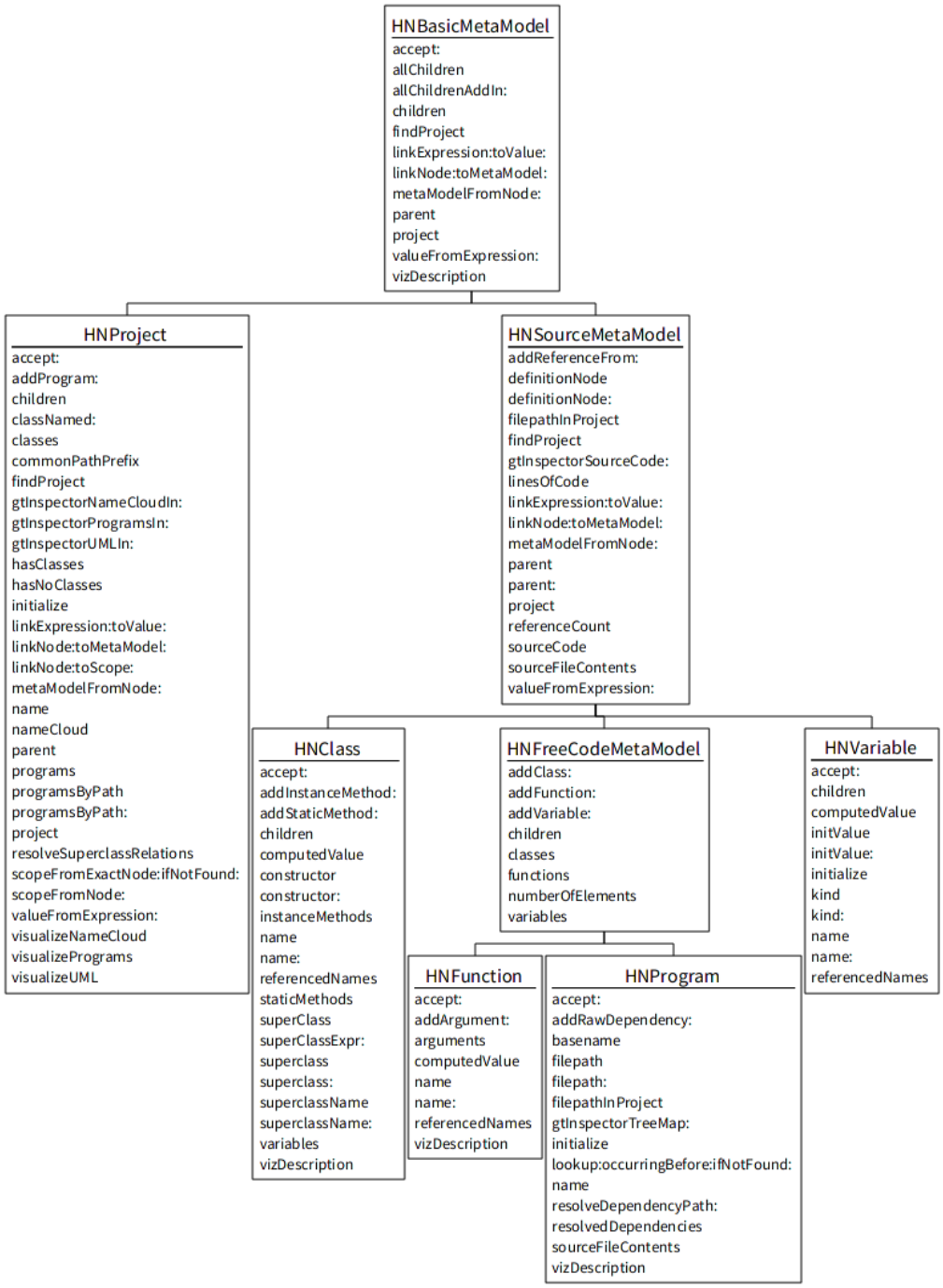

Figura 3.2: Diagrama UML de clases de los metamodelos incluyendo clases abstractas

Por otro lado, para extraer la información del AST necesaria para rellenar los metamodelos definidos se hace uso del patrón *visitor*. Dicho patrón permite desacoplar el AST de las operaciones que se realicen sobre él, facilitando la incorporación de una cantidad de arbitraria de procesadores de AST (denominados también *visitors*) que obtengan datos para alimentar los metamodelos según vaya siendo necesario.

Los *visitors* definidos actualmente son:

- **HNMetaModelBuilder**. Instancia los metamodelos con información básica para que

otros *visitors* puedan aumentar la información.

- **HNLHSPatternVisitor**. *Visitor* específico para recoger apareciendo al lado izquierdo de una declaración de variables.
- **HNScopeBuilder** Recorre el AST buscando contextos de declaración de variables.
- **HNIdentifierBinder** Asocia cada identificador en el código fuente con un contexto de declaración de variables obtenido en el paso anterior.
- **HNImportProcessor**. Visitor encargado de procesar las dependencias entre archivos, que luego son mostradas en la visualización de grafo de dependencias (detallada en la Sección 4.2.1). Cabe mencionar que las dependencias son capturadas a nivel de archivo y que, para su obtención, se identifican las llamadas a la función `require` y las ocurrencias de la construcción semántica de `import`.

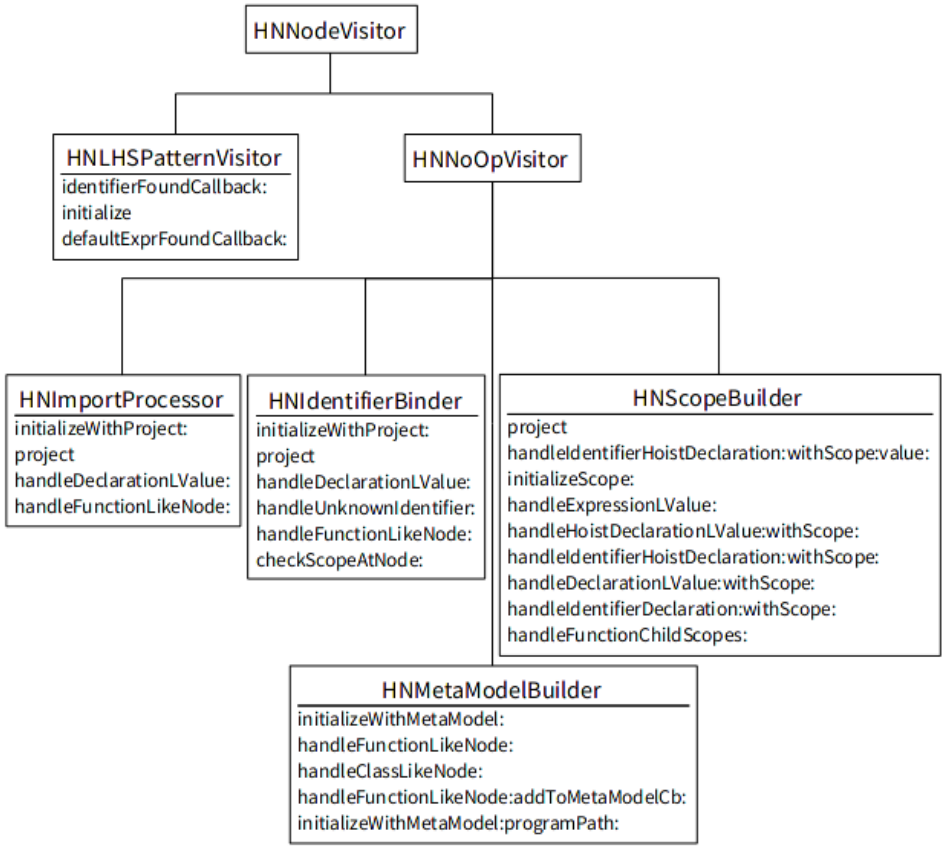

Figura 3.3: Diagrama UML de los visitors incluyendo clases abstractas. Los métodos encargados de visitar nodos del AST han sido excluidos por brevedad

Capítulo 4

Interfaz de usuario

4.1. Primeros acercamientos

El diseño de la interfaz de usuario de HUNTER fue principalmente un proceso iterativo, en el cuál se realizaron varias aproximaciones hasta alcanzar el diseño final. En esta sección se detallan las etapas previas la versión definitiva.

4.1.1. Inspector de elementos de Pharo

Al comienzo del desarrollo de HUNTER, se puso énfasis en la construcción de la infraestructura necesaria para realizar el análisis de los archivos de JavaScript, por lo cual la interfaz de usuario fue relegada a un segundo plano. En particular, esto fue posible dado que Pharo incluye un potente motor de inspección de elementos que, a pesar de que su principal caso de uso es facilitar la labor de *debugging*, su flexibilidad permitió bozquejar rápidamente una versión rudimentaria de la interfaz de HUNTER. Ésta facilitó la exploración gráfica del metamodelo generado en las fases anteriores de procesamiento, aunque presentara algunos problemas de usabilidad.

Concretamente para la construcción de este primer prototipo se hizo uso de la característica del inspector de poder definir pestañas personalizadas como métodos dentro de las clases de los objetos dentro de Pharo.

El funcionamiento de esta interfaz puede resumirse de la siguiente forma:

- Al abrir un proyecto se despliega una ventana con el inspector de Pharo. Esta contiene una serie de pestañas que permiten escoger la visualización general del proyecto a mostrar. Esto se ilustra en la Figura 4.1.

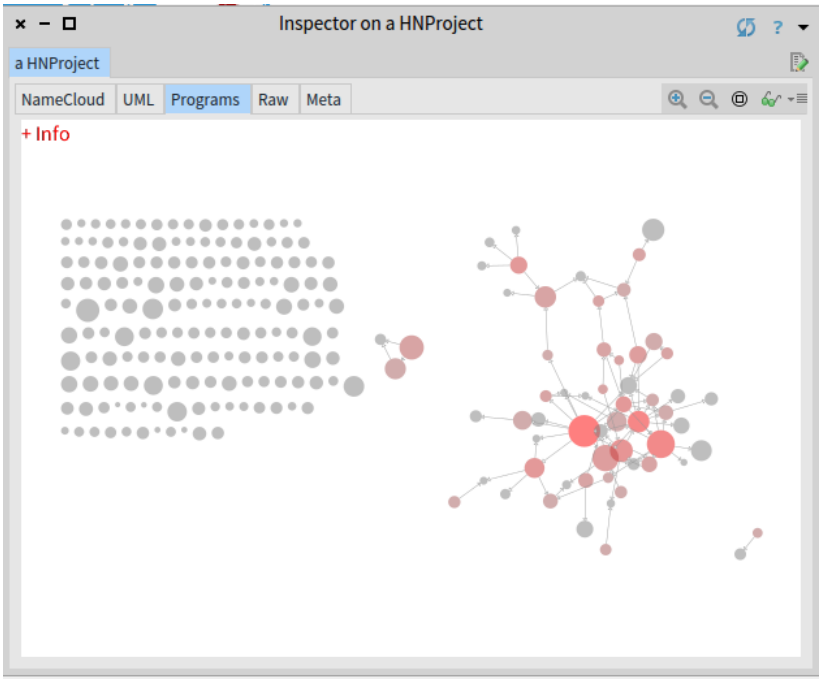

Figura 4.1: Primera versión de HUNTER que usa el inspector mostrando el grafo de dependencias.

- Al hacer *click* sobre un elemento de la visualización general se divide la pantalla en dos paneles: el panel izquierdo muestra la vista del inspector global que anteriormente abarcaba la ventana completa, mientras que el panel derecho despliega la vista del inspector del elemento *clickeado*. Esto puede apreciarse en la Figura 4.2.

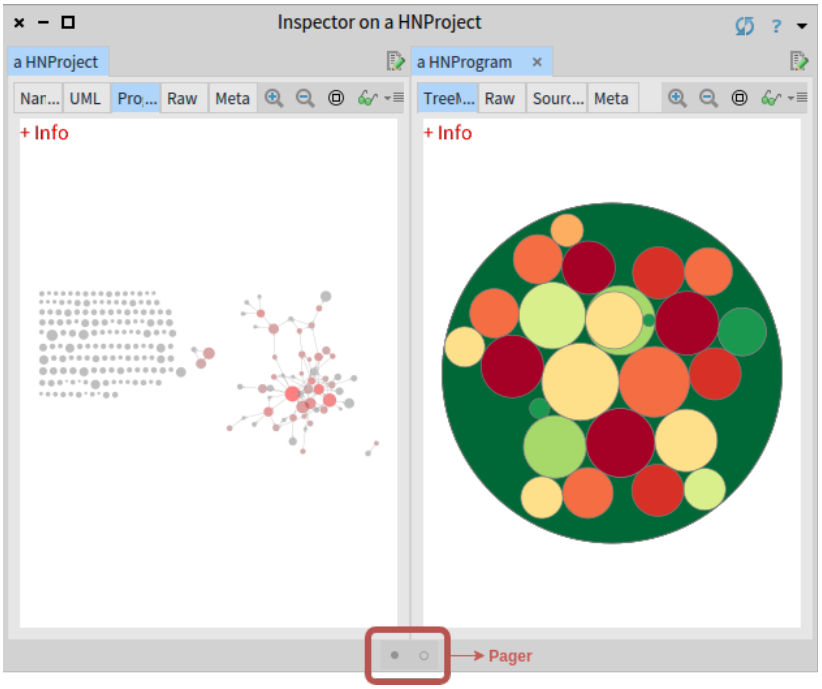

Figura 4.2: Primera versión de HUNTER que usa el inspector viendo el detalle de un archivo JS.

- A su vez, puede hacerse *click* en alguno de los elementos del panel derecho. Esto permite abrir un nuevo nivel en la ventana del inspector para, por ejemplo, inspeccionar el código fuente en particular de una clase o función, tal como puede verse en la Figura 4.3.

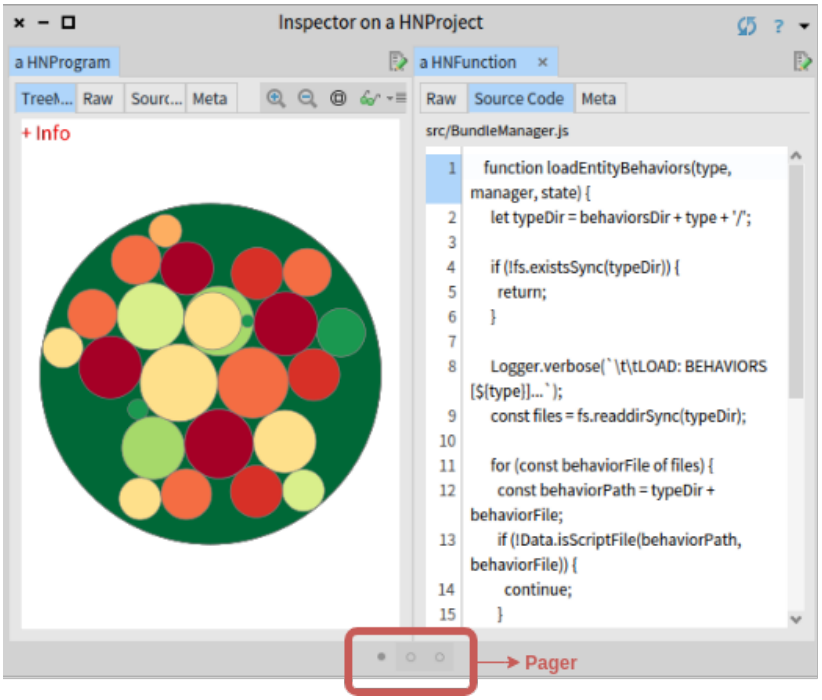

Figura 4.3: Primera versión de HUNTER que usa el inspector desplegando el código fuente de un archivo.

- Cabe mencionar que, por defecto, solamente se muestran los dos niveles más profundos del inspector. Esto puede cambiarse interactuando con el componente de *pager* hallado en la parte inferior de la ventana. Este control permite, en primer lugar, seleccionar qué niveles del inspector se muestran y, por otro lado, controlar la cantidad de niveles que aparecen desplegados en la ventana.

Lamentablemente, a pesar de la facilidad de implementación conseguida al usar la infraestructura del inspector de Pharo, pilotos realizados con ingenieros que ejercen en la industria pusieron en manifiesto que esta aproximación resultó ser insuficiente. En efecto, se detectaron inconvenientes tales como los siguientes:

- El uso del componente de *pager* que usa el inspector no resultó ser intuitivo para los sujetos. De forma tangible, después de alcanzar tres niveles de inspección ellos no podían volver a la vista general sin recibir instrucciones directas del entrevistador.
- Las pestañas de *Raw* y *Meta* distraían a los usuarios. Dentro de un contexto de *debugging* usual, estas pestañas permiten al programador inspeccionar las variables de instancia y obtener información acerca de la clase del objeto inspeccionado respectivamente. Sin embargo, dichas pestañas no mejoran la experiencia en la exploración visual de un proyecto de JavaScript en HUNTER.
- Expectativas no cumplidas en el flujo de navegación. En particular, se esperó que al abrir el detalle de un elemento desde el panel de inspección global de un proyecto se

obtuviera un *Treemap* mostrando sus subelementos. Y, por otra parte, se esperó que al seleccionar un subelemento en el *Treemap* de un elemento, se desplegara el código fuente de este subelemento. Lamentablemente, no existe una forma limpia y mantenible de implementar este comportamiento con la utilización del inspector.

En síntesis, se han identificado dificultades que surgen directamente del uso del inspector de elementos para presentar los metamodelos de HUNTER. Si bien el empleo de este mecanismo permitió construir una interfaz de usuario rápidamente, continuar con la aplicación de este enfoque no resolvería dichas problemáticas. Así pues, nace la necesidad de construir una interfaz gráfica propia con alguna de las herramientas disponibles en la plataforma de Pharo.

4.1.2. Interfaz de usuario en Glamour

De este modo, se tomó la decisión de desarrollar una nueva interfaz de usuario de HUNTER usando el *framework* de Glamour [2]. Glamour está especialmente pensado para construir interfaces gráficas denominadas *browsers*. Un *browser* se define como una interfaz de usuario concebida para la exploración visual de un modelo a través de un *flujo de navegación* determinado por el *browser* mismo. La documentación disponible sobre Glamour en The Moose Book [13] describe brevemente las nociones detrás de Glamour de la siguiente forma:

« En resumen, en Glamour existen cinco conceptos centrales: un *browser* (1) compuesto de *paneles* (2) que demarcan el lugar espacial en el cual distintos *objetos* (3) son *presentados* (4). Asimismo, el resultado de interactuar en la *presentación* (4) gráfica es *transmitido* (5) a otros paneles. »

Visualmente esta iteración sigue un esquema basado en una ventana con dos paneles: uno a la izquierda mostrando una visión global del proyecto; y uno a la derecha mostrando una visión en detalle de la entidad seleccionada. Este esquema es similar usado en la interfaz final en Spec (que se encuentra detallado en la Sección 4.2.1); aunque la implementación sea diferente.

Con ello, a priori se percibieron las siguientes ventajas de construir la interfaz gráfica de HUNTER en Glamour:

- Al ser Glamour una herramienta para la construcción de *browsers*, resulta razonable concebir a HUNTER como un *browser* sobre un proyecto escrito en JavaScript; en el cual hay que definir meramente el *flujo de navegación* y los componentes que conforman la interfaz.
- Una gran cantidad de ejemplos disponibles en el mismo entorno de Pharo.
- Una integración muy conveniente con el motor de visualización de Roassal, la cual permitía inspeccionar los elementos *clickeados* en la visualización sin la necesidad de escribir ningún código adicional.
- Posibilidad de reutilizar algunas vistas de la versión anterior, pues éstas se encontraban especificadas como pestañas del inspector de elementos de Pharo, las cuales deben ser escritas en Glamour.

Sin embargo, a pesar de la percepción inicial prometedora que se tuvo sobre Glamour, en la práctica se percibieron varias falencias que vale la pena mencionar:

- Una falta considerable de documentación al día. Toda la documentación hallada para obtener una idea clara sobre el funcionamiento del *framework* se extrajo de un capítulo de The Moose Book. Sin embargo, ésta se encontraba incompleta y en algunos casos desactualizada. Concretamente en ningún momento queda claro qué es un *puerto pasivo* o una *validación* y sólo se detallan tres de los diez *browsers* que existen en la versión actual.
- Los ejemplos que se incluían en Pharo, resultaban demasiado triviales en la práctica y no permitían resolver todas las inquietudes surgidas en el proceso de desarrollo; en particular, no se encontró ningún ejemplo que resolviera de forma acertada el problema de tener varios *browsers* anidados y que permitiera propagar los modelos seleccionados en un panel a través de esta jerarquía de *browsers*.
- Finalmente el enfoque de muy alto nivel ofrecido por Glamour y su complejidad conceptual inherente, en conjunto con los inconvenientes ya mencionados, causaron dificultades en el avance en la implementación de la nueva versión de la interfaz. Resultaba complicado determinar la forma correcta de utilizar la API de Glamour para la construcción de la interfaz de usuario de HUNTER.

A causa de estos problemas, se tomó la decisión de descartar la interfaz creada con Glamour y recomenzar su reconstrucción con la biblioteca Spec.

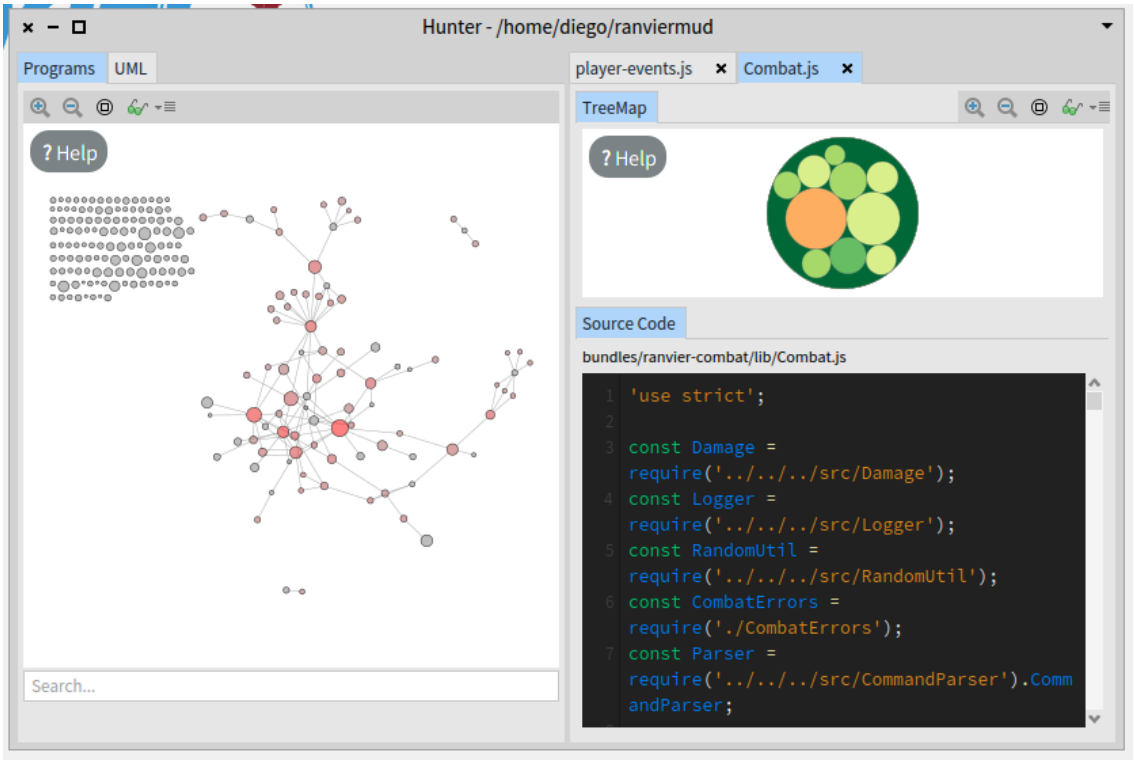

Figura 4.4: Captura de pantalla de la última versión de HUNTER escrita en Glamour.

4.2. Interfaz final en Spec

Dados los inconvenientes encontrados, se resolvió utilizar Spec para confeccionar la interfaz de usuario de HUNTER. Esta interfaz sería la utilizada en la versión final de HUNTER.

Citando la introducción del manual The Spec UI framework [9]:

« Spec es un *framework* para describir interfaces de usuario. Permite la construcción de una amplia variedad de interfaces; desde ventanitas con unos cuantos botones hasta herramientas complejas como un *debugger*. [...] »

De este modo, en contraste con Glamour —que facilita la construcción de un tipo particular de interfaces—, Spec es un *kit* de herramientas para la creación de interfaces gráficas de usuario en general. En efecto, si bien ambos *frameworks* facilitan el diseño modular de interfaces por medio de componentes reutilizables, Spec adopta un enfoque más minimalista y opta por no abordar el problema de la propagación de las interacciones, dejando al programador la tarea comunicar los componentes configurando los *callbacks* respectivos. Esta aproximación, en conjunto con una documentación completa y varios ejemplos *reales* de uso dentro del *software* incluido en Pharo, permitió abordar de forma efectiva el problema de la implementación de una interfaz usable para HUNTER.

Otra ventaja percibida de Spec por sobre Glamour fue su estructura de componentes basada en clases contra el estilo basado en *scripts* de Glamour. Ilustrativamente, Glamour está concebido como un motor de construcción de *browsers* mediante una API de *scripting*. Esto quiere decir que Glamour favorece principalmente la definición de la interfaz usuario, con su *layout* e interacciones entre componentes, en un único método; lo cual, si bien facilita el prototipado rápido de interfaces, también crea un ambiente propicio para la escritura de código difícil de mantener a largo plazo. En contrapartida, Spec impone una estructura clara basada en clases siguiendo las convenciones de la programación orientada a objetos; en particular, a cada componente le corresponde una clase que contiene métodos separados para las siguientes tareas: especificar el *layout*, inicializar el componente y definir las interacciones entre subcomponentes.

4.2.1. *Layout* final

Con ello, de acuerdo a la Figura 4.5, el *layout* de la interfaz final puede puede descomponerse de la siguiente manera:

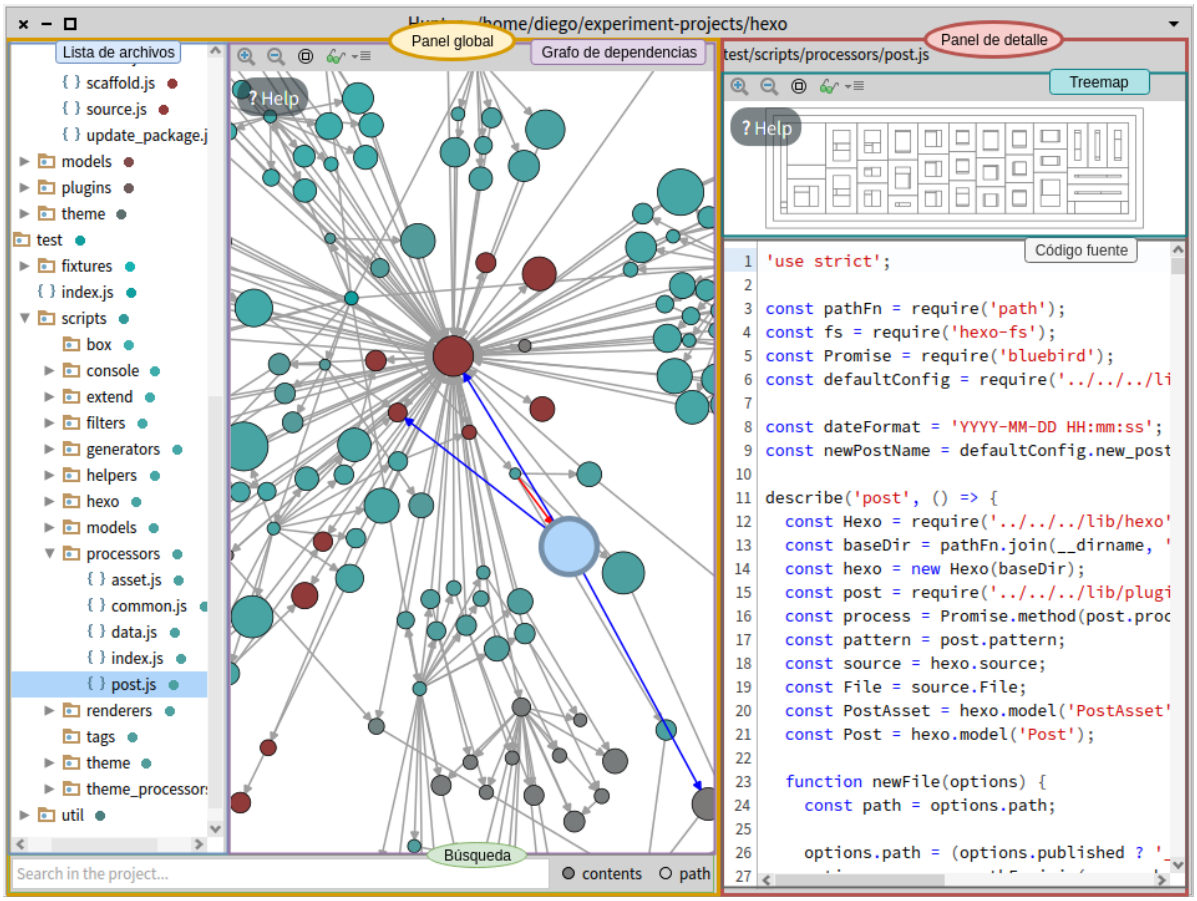

Figura 4.5: Layout de la interfaz final de HUNTER escrita en Spec.

- **Panel global**. Panel mostrado al lado izquierdo de la ventana de HUNTER. Este permite al usuario tener una visión global del proyecto; emulando un funcionamiento similar al de la mayoría de los IDEs y editores de texto modernos; los cuales despliegan a la izquierda un outline general del proyecto, y muestran a la derecha el archivo editado actualmente. En esta iteración de HUNTER, este panel contiene el grafo de dependencias, la lista expandible de archivos y una búsqueda en la parte inferior.
- **Lista de archivos**. Componente encargada de mostrar los archivos de JavaScript del proyecto actual. Éstos se hayan agrupados en carpetas, las cuales pueden ser expandidas o colapsadas. Al final del nombre de la carpeta o archivo se muestra un color; el cual es asignado por HUNTER de acuerdo a la carpeta que lo contiene en un primer nivel y que, en el caso de los archivos, corresponde con el color con el que se muestra en el grafo de dependencias. Esta característica permite entender gráficamente la estructura del proyecto visualizado. Asimismo, al hacer click sobre un archivo, éste se *selecciona* y aparece destacado en la del grafo de dependencias, además de desplegarse en la vista de detalle.
- **Grafo de dependencias**. Componente que despliega el grafo de archivos con sus dependencias. En esta vista cada archivo es modelado por un nodo circular, cuyo radio es proporcional a las líneas de código que posee. El color por otro lado depende de la carpeta de primer nivel que lo contiene. Este grafo además es interactivo, permitiendo al usuario descubrir el nombre de los archivos posando el *mouse* sobre su nodo correspondiente y *seleccionarlo* al hacer *click* sobre uno de ellos. Al momento de seleccionar

un archivo éste se muestra en el panel de detalle y se destaca en la lista de archivos.

- **Búsqueda**. Permite hacer búsqueda por texto tanto en el contenido de los archivos como en su ruta. Los archivos que calzan con la búsqueda se destacan en color amarillo tanto en la lista de archivos como en el grafo de dependencias. Además en el caso de la búsqueda por contenido, las ocurrencias también se destacan en la vista del código fuente.
- **Panel de detalle**. Panel mostrado al lado derecho de la ventana de HUNTER. Su objetivo es desplegar información del archivo *seleccionado* actualmente. Para ello, en primer lugar, muestra la ruta del archivo actual dentro del proyecto. Luego despliega un gráfico de *Treemap* mostrando las entidades (es decir, funciones o clases) que posee el archivo. Posteriormente presenta una componente con su código fuente.
- **Treemap**. Visualización de *treemap* que muestra las entidades que contiene el archivo *seleccionado* actualmente. Estas entidades pueden ser tanto funciones (ya sea anónimas o no) como clases. Al hacer *click* sobre uno de los elementos la vista del código fuente se desplaza hasta su ocurrencia y aparece destacado en ella.
- **Código fuente**. Muestra el código fuente del archivo seleccionado actualmente. Además del resaltado de sintaxis de JavaScript, aparecen destacados tanto la función o clase *seleccionada* actualmente como los resultados de la búsqueda.

Capítulo 5

Validación

5.1. Pruebas con ingenieros en la industria

A lo largo del desarrollo de HUNTER se llevaron a cabo una serie pilotos libres tanto con ingenieros en la industria como con otros alumnos e investigadores. Esto permitió guiar el desarrollo de la aplicación y enfocarlo hacia las características que aportaban más valor. Si bien durante la confección de la herramienta se realizaron pruebas periódicas con el profesor guía y otros investigadores asociados —lo cual facilitó la obtención de *feedback* temprano—, resultaron ser de particular importancia las sesiones con ingenieros en la industria. Estas entrevistas permitieron, por un lado, aportar una visión imparcial acerca del proyecto al no hayarse involucrados de ninguna forma en el desarrollo del mismo; y, por otro, aumentar el entendimiento del problema con una visión basada en la experiencia.

La metodología en todos estos pilotos fue la misma: permitir que los usuarios exploraran un proyecto mediante HUNTER, dando sólo instrucciones al ser consultado directamente por un asunto en concreto, o para indicar una característica que no había sido probada por el sujeto al notar un estancamiento en las acciones realizadas en HUNTER por el usuario. Esta manera de realizar los pilotos —que trató de reducir las intervenciones del entrevistador— permitió detectar problemas de usabilidad más allá de lo reportado por el entrevistado.

En consecuencia, en la construcción de HUNTER hubo dos fases en los cuales se realizaron pilotos con ingenieros. La primera, al encontrarse ya avanzada la interfaz de HUNTER realizada en el inspector de elementos de Pharo. Y la segunda, una vez terminada la obra gruesa de la interfaz escrita en Spec.

La primera ronda de entrevistas fue realizada especialmente con el objetivo de reducir incertidumbre sobre las futuras características a implementar en HUNTER. Aunque la infrastructura básica de HUNTER se encontraba ya construida —esto es, la fase de *parsing*, los objetos del AST y el metamodelo de las entidades dentro del código— hubo dudas en cuanto a si era realmente necesario implementar una infrastructura más potente que permitiera el cálculo de métricas más sofisticadas, y la deducción de información más compleja a partir del código fuente, tales como el seguimiento de referencias e invocaciones de funciones. Lo último

no es una labor trivial en JavaScript dado lo dinámico del lenguaje, y la falta de convenciones y estructuras dentro del código. Los resultados de los pilotos revelaron que aún era necesario pulir la experiencia de usuario de HUNTER. En concreto, tras la realización de dos pilotos con ingenieros se detectaron los siguientes problemas, que fueron corregidos en la siguiente iteración de HUNTER:

- Los usuarios no eran capaces de encontrar la leyenda dentro de las visualizaciones. Esto motivó un rediseño del botón que las despliega.
- Los sujetos no podían deducir que las visualizaciones eran interactivas, por lo cual se agregó la interacción de *highlighting* al pasar el *mouse* sobre los nodos graficados.
- El flujo de navegación impuesto por el inspector de Pharo no resultaba intuitivo.
- Se indicó que faltaba resaltar sintáxis en la sección de código fuente.

Finalmente en la segunda ronda de pilotos se entrevistaron tres ingenieros, dentro de las dificultades experimentadas fueron:

- Intutivamente los usuarios intentaban quitar la selección del archivo actual haciendo *click* en un espacio en blanco dentro de la visualización del grafo de dependencias.
- Un usuario encontró confuso el uso de las flechas en el grafo de dependencias. Lo cual sugiere que es preciso añadir dicha información en la leyenda de la visualización.
- La visualización de *treemap* por defecto posee una interacción que permite arrastrar las cajas con el *mouse*. Esta interacción es innecesaria y causó confusión en los sujetos al gatillarla por error.

Sin embargo, a pesar de los problemas identificados en esta entrevista, los usuarios calificaron positivamente la herramienta. En particular, apreciaron la forma gráfica de explorar *software* que otorgó HUNTER y la rápidez con la que les permitió ganar un mayor entendimiento acerca de la arquitectura lógica de aplicaciones.

5.2. Experimentos guiados por terceros

Adicionalmente a los pilotos detallados anteriormente por parte del autor para apoyar al desarrollo de HUNTER, otros investigadores han diseñado y realizado un experimento controlado para evaluar de forma más concreta la manera en que HUNTER ayuda a la comprensión de código. Si bien los experimentos no fueron diseñados ni organizados por el autor; los resultados preliminares resultan alentadores y agregan valor al presente trabajo en dos sentidos: en primer lugar, pone de manifiesto la relevancia de HUNTER al ser objeto de estudio; y en segundo, otorgan una validación tangible y, eventualmente, repetible de lo desarrollado en esta memoria. Es por ello que se introduce esta sección, en la cual se detallará brevemente el diseño experimental, y se analizarán los resultados obtenidos en una primera instancia.

Se entrevistaron 16 sujetos, a los cuales se dividió en cuatro grupos diferentes al azar. A cada uno de los grupos creados se les asignaron dos proyectos; uno para ser analizado en HUNTER y otro para ser analizado en Visual Studio Code.

La Tabla 5.1 detalla los proyectos que fueron analizados y sus características, mientras que la Tabla 5.2 detalla qué proyectos fueron asignados a qué grupos. Una tabla que muestra la URL de los repositorios de Git y los *commits* específicos puede hallarse en el apéndice A.

Proyecto	Número de archivos `.js`	Número de líneas de código
Local Forage	46	12154
Hexo	266	17411
es6-mario	44	1517
serverless	373	46757

Tabla 5.1: Proyectos analizados en el experimento y su tamaño en líneas de código y cantidad de archivos de JavaScript.

Proyecto/Herramienta	HUNTER	Visual Studio Code
Local Forage	Grupo A	Grupo B
Hexo	Grupo B	Grupo A
es6-mario	Grupo C	Grupo D
serverless	Grupo D	Grupo C

Tabla 5.2: Asignación de proyectos de JavaScript por herramienta.

Así, a cada uno de los integrantes de los grupos con sus respectivos proyectos a analizar tanto en HUNTER como en Visual Studio Code se les hizo utilizar las herramientas para responder preguntas de acuerdo a la siguiente plantilla:

- ¿Cuál es el archivo más invocado?
- ¿Cuál es el archivo JS que más invoca a otros archivos?
- ¿A cuántos archivos invoca el archivo X?
- ¿Por cuántos archivos es invocado el archivo Y?
- ¿Qué archivo JS tiene mayor cantidad de líneas de código?
- ¿Cuántos archivos no invocan ni son invocados por otros archivos JS?
- Identifique el *folder* que contiene la mayor cantidad de archivos JS (recursivamente).
- ¿Qué funciones llaman a la función Z definida en el archivo W?
- ¿Si muevo la función U definido en el archivo V, qué más debo mover?

Donde X, Y, Z, W, U y V son archivos y nombres de métodos o funciones distintos para cada proyecto. Con esto el diseño experimental permitiría establecer una comparación directa entre HUNTER y un editor de texto más tradicional como Visual Studio Code; pues habrían dos grupos de sujetos analizando el mismo proyecto y cumpliendo tareas idénticas, pero con herramientas distintas. Sin embargo, aún queda pendiente la fase de análisis de las grabaciones de uso de las herramientas para realizar esta comparación.

No obstante, se ha recabado *feedback* subjetivo de los participantes de los experimentos que permitiría establecer indicios acerca de la utilidad de HUNTER. En particular, tras el uso de ambas herramientas, los sujetos respondieron un cuestionario detallado en la Tabla 5.3, en

el cual deben valorar de 1 a 7 qué tanto concuerdan con una serie de aseveraciones positivas sobre HUNTER.

Asimismo, la Figura 5.1 muestra gráficamente la valoración promedio y la desviación estándar para cada una de las preguntas ordenadas de menor a mayor promedio.

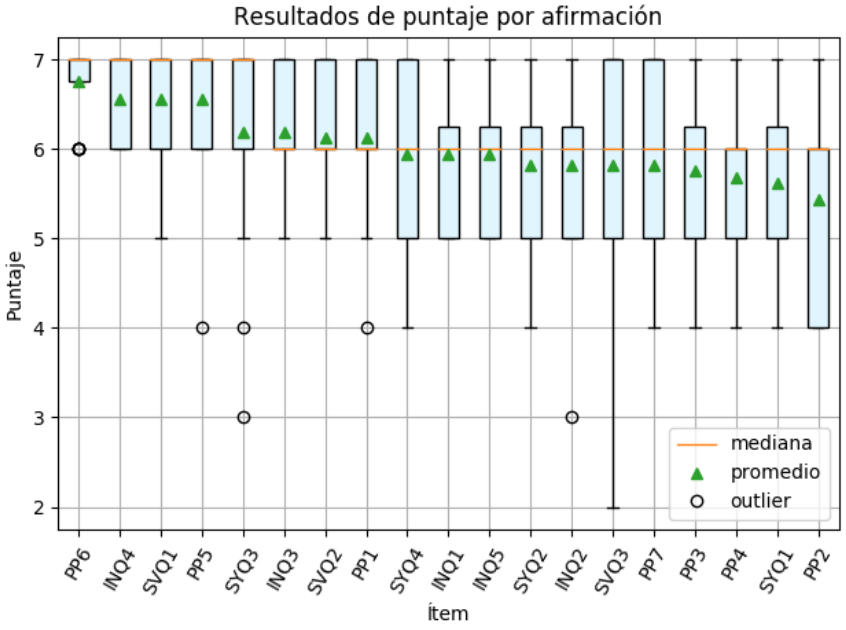

Figura 5.1: Diagrama de cajas y "bigotes" con los resultados del cuestionario subjetivo. Para cada *ítem*, las cajas encierran los valores entre el cuartil 1 y 3. Por otro lado, los "bigotes" se extienden desde los límites de las cajas hasta una distancia de $1{,}5(Q_3 - Q_1)$ unidades con Q_1 y Q_3 el primer y tercer cuartil respectivamente. Más allá de los límites de los "bigotes", los valores son considerados como *outliers* y son graficados individualmente.

De esta forma, se puede observar que, en general, la herramienta ha sido bien evaluada subjetivamente por los usuarios siendo el promedio general de todas las evaluaciones 6,0329, la mediana 6, con una desviación estándar de 0,9434.

Al analizar de forma más fina los resultados, se obtuvo que las cuatro aseveraciones mejor evaluadas fueron: "*PP6. La herramienta ayuda a la exploración.*" (promedio: 6,75, mediana: 7), "*INQ4. La información es confiable.*" (promedio: 6,5625, mediana: 7), "*PP5. El usar esta herramienta estimula mi curiosidad.*" (promedio: 6,5625, mediana: 7) y "*SVQ1. Se puede confiar en las funcionalidades provistas.*" (promedio: 6,5625, mediana: 7).

Por otro lado, las cuatro aseveraciones peor evaluadas fueron (de menor a mayor puntaje): "*PP2. Cuando interactúo con la aplicación no me doy cuenta de ningún ruido.*" (promedio: 5,4375, mediana: 6), "*SYQ1. Tiene un estilo y diseño apropiado.*" (promedio: 5,625, mediana: 6), "*PP4. El usar esta herramienta me da diversión en lo que estoy haciendo.*" (promedio: 5,6875, mediana: 6), "*PP3. El usar esta herramienta me hace disfrutar la tarea que estoy realizando.*" (promedio: 5,75, mediana: 6).

Con esto, a partir de este experimento organizado por terceros, puede concluirse que —de acuerdo a las percepciones subjetivas— HUNTER sí ayuda a explorar proyectos desconocidos, es confiable y estimula la curiosidad de los usuarios. Sin embargo, existe un espacio para mejorar la experiencia de usuario actual, principalmente mediante una interfaz con mejor estética. Si bien, puede ser tentador incluir dentro de los posibles aspectos a mejorar el componente de *engagement* referenciado en las afirmaciones *PP2*, *PP3*, *PP4*, no es evidente cómo mejorar este enfoque; especialmente al tomar en cuenta el principal objetivo de HUNTER: la presentación de información sobre código fuente de proyectos de JavaScript. En contrapartida, los datos apuntarían a que el rendimiento de la interfaz de HUNTER es suficiente para los usuarios, salvo casos particulares (gracias a la aseveración SYQ3, que posee mediana 7, pero que tiene 2 *outliers* que bajan su puntaje promedio, en virtud de la Figura 5.1).

Código	Afirmación	Promedio	Mediana	Desv. estándar
SYQ1	Tiene un estilo y diseño apropiado.	5,625	6	1,0878
SYQ2	La navegación de los datos es fácil.	5,8125	6	0,9106
SYQ3	La interfaz responde rápidamente.	6,1875	7	1,2230
SYQ4	Las funcionalidades ofrecidas son adecuadas para el tipo de aplicación.	5,9375	6	0,9287
INQ1	El contenido es suficiente para la información requerida.	5,9375	6	0,7719
INQ2	La información es precisa.	5,8125	6	1,047
INQ3	La información es oportuna.	6,1875	6	0,6551
INQ4	La información es confiable.	6,5625	7	0,5123
INQ5	La información se encuentra en un formato adecuado.	5,9375	6	0,7719
SVQ1	Se puede confiar en las funcionalidades provistas.	6,5625	7	0,6292
SVQ2	Da confianza a los usuarios reduciendo la incertidumbre.	6,125	6	0,7188
SVQ3	Provee detalles sobre demanda a los usuarios.	5,8125	6	1,2764
PP1	Cuando interactúo con la aplicación no me doy cuenta del paso del tiempo.	6,125	6	0,8062
PP2	Cuando interactúo con la aplicación no me doy cuenta de ningún ruido.	5,4375	6	1,0935
PP3	El usar esta herramienta me hace disfrutar la tarea que estoy realizando.	5,75	6	1
PP4	El usar esta herramienta me da diversión en la tarea que estoy realizando.	5,6875	6	0,8732
PP5	El usar esta herramienta estimula mi curiosidad.	6,5625	7	0,8139
PP6	La herramienta ayuda a la exploración.	6,75	7	0,4472
PP7	La herramienta potencia mi imaginación.	5,8125	6	1,0468

Tabla 5.3: Cuestionario subjetivo con aseveraciones sobre HUNTER. Los sujetos evalúan las afirmaciones de 1 a 7, donde 1 es *totalmente en desacuerdo* y 7 es *totalmente de acuerdo*. Incluye el promedio, mediana y desviación estándar de las valoraciones para cada *ítem*.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Dentro del contexto de este trabajo de memoria, se ha logrado construir una primera versión de HUNTER, una plataforma de reingeniería para JavaScript. Esta herramienta permite extraer las dependencias a nivel de archivo en proyectos de JavaScript y presentarlas de forma tal que facilita la comprensión de la arquitectura del *software* analizado; lo cual pudo ser corroborado en la práctica por el autor mediante pilotos informales con usuarios. Más aún, HUNTER ha sido calificado de forma positiva en estudios más formales realizados por otros investigadores.

Con respecto al estado del arte de herramientas para analizar proyectos de JavaScript, HUNTER representa una mejora al integrar tres aspectos de forma simultánea: presentación de una visión global de la aplicación analizada mediante un grafo de dependencias; capacidades de búsqueda textual destacando los resultados en la visualización; e inspección del código fuente de los componentes del *software*. De este modo, HUNTER es capaz de aumentar el entendimiento de proyectos de JavaScript de una forma distinta al resto de las herramientas ya existentes para dicho lenguaje.

Por otro lado, más allá de los resultados concretos obtenidos en este trabajo de titulación, el desarrollo de HUNTER ha sido una experiencia formativa que permitió poner en práctica una gran cantidad de conocimientos adquiridos a lo largo de la carrera. En particular el uso de las metodologías de orientación a objetos y patrones de diseño facilitó la construcción del *software* de forma escalable y modular, mientras que la realización de *tests* unitarios permitió guiar el desarrollo de ciertas funcionalidades y otorgar confianza a la hora de refactorizar el código. La aplicación de los dos puntos anteriores se vio facilitada al trabajar en Pharo, dado que dicho lenguaje ofrece un ambiente de programación especialmente propicio para el desarrollo incremental, permitiendo incluso la modificación del código de los programas que se encuentran en ejecución o que arrojan errores. Esto último fue una gran ayuda, pues permitió acortar el ciclo de *feedback* entre la escritura de una parte del programa y la corrección de sus *bugs*; lo cual se vio reflejado en una mayor productividad. Asimismo, la validación realizada con otros usuarios a lo largo del desarrollo permitió dirigir la confección de HUNTER en un

nivel superior y, con ello enfocar el desarrollo de HUNTER en lo que brindaba más valor. En este sentido, puede afirmarse que el presente trabajo de memoria fue una gran oportunidad para apreciar de forma práctica los beneficios del desarrollo ágil de *software*; y, a juicio del autor, resulta díficil concebir el haber obtenido resultados similares mediante un desarrollo de tipo cascada.

6.2. Trabajo futuro

No obstante, a pesar de haberse cumplido los objetivos propuestos, es preciso reconocer que existen varios aspectos que se encuentran sujetos a mejoras o que ameritan un estudio adicional.

En primer lugar, tanto en los pilotos libres como en los experimentos formales, pese a la sensación general positiva dejada tras el uso de HUNTER, los usuarios entregaron *feedback* que apuntaba a desperfectos menores y pequeños detalles de usabilidad dentro de la interfaz. De esta forma, se propone solucionar aquellas imperfecciones para futuras iteraciones de HUNTER.

Por otra parte, si bien los usuarios recibieron de forma positiva la visualización del grafo de dependencias a nivel de archivos, muchos de ellos manifestaron la utilidad de obtener un análisis similar a un nivel más fino: de clase o función; lo cual no pudo realizarse, pues dada la naturaleza dinámica de JavaScript, en conjunto con una falta de convenciones claras y prácticas heterogéneas en distintos proyectos, resultaba complejo determinar a ciencia cierta qué métodos u objetos eran invocados o referenciados.

Asimismo, si bien HUNTER funciona unicamente para proyectos de JavaScript, puede observarse que al menos las funcionalidades implementadas en este trabajo pueden ser adaptadas fácilmente para otros lenguajes. De esta forma, se plantea la posibilidad de construir visualizaciones interactivas similares para otros lenguajes, o, inclusive, extender HUNTER de modo que pueda ser, en mayor o menor medida, agnóstico al lenguaje concreto del proyecto analizado.

Finalmente, cabe mencionar que —aunque el caso de uso principal de HUNTER estudiado y abordado a lo largo de este trabajo de memoria fue el de ayudar al desarrollador en la comprensión de código de *software* desconocido— puede ser interesante explorar la forma en que HUNTER podría aumentar el entendimiento de proyectos de los cuales el programador ya posee un conocimiento previo.

Bibliografía

[1] Fabian Beuke. *Github Language Stats*. JavaScript es el lenguaje con más *pull requests* en el primer cuatrimestre de 2018. URL: https://madnight.github.io/githut/#/pull_requests/2018/1 (visitado 21-04-2018).

[2] Philipp Bunge y col. "Scripting Browsers with Glamour". Tesis de mtría. University of Bern, 2009.

[3] CoderGears. *JArchitect :: Java Static Analysis and Code Quality Tool*. URL: https://www.jarchitect.com/ (visitado 21-04-2018).

[4] CoderGears. *VBDepend :: VB6/VBA Static Analysis and Code Quality Tool*. URL: http://www.vbdepend.com/ (visitado 21-04-2018).

[5] Mozilla Contributors. *Promise.prototype.catch() | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch (visitado 06-09-2018).

[6] Ward Cunningham. "The WyCash Portfolio Management System". En: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*. OOPSLA '92. Vancouver, British Columbia, Canada: ACM, 1992, págs. 29-30. ISBN: 0-89791-610-7. DOI: 10.1145/157709.157715. URL: http://doi.acm.org/10.1145/157709.157715.

[7] Serge Demeyer, Stéphane Ducasse y Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[8] Ralf S. Engelschall. *ECMAScript 6: New Features: Overview and Comparison*. URL: http://es6-features.org/ (visitado 22-04-2018).

[9] Johan Fabri y Stéphane Ducasse. *The Spec UI framework*. 2017.

[10] Facebook. *JSX | XML-like syntax extension to ECMAScript*. URL: https://facebook.github.io/jsx/ (visitado 22-04-2018).

[11] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.

[12] Node.js Foundation. *Express - Node.js web application framework*. URL: http://expressjs.com/ (visitado 21-04-2018).

[13] Tudor Gîrba. *The Moose Book*. URL: http://themoosebook.org/book/index.html (visitado 23-12-2018).

[14] Patrik Henningsson. *pahen/madge - Create graphs from your CommonJS, AMD or ES6 module dependencies*. URL: https://github.com/pahen/madge (visitado 17-12-2018).

[15] Github Inc. *Electron | Build cross platform desktop apps with Javascript, CSS and HTML*. URL: https://electronjs.org/ (visitado 21-04-2018).

[16] ECMA International. *Standard ECMA-262*. URL: https://www.ecma-international.org/publications/standards/Ecma-262.htm (visitado 22-04-2018).

[17] M. M. Lehman. "Programs, life cycles, and laws of software evolution". En: *Proceedings of the IEEE* 68 (9 sep. de 1980), págs. 1060-1076.

[18] Jean-Louis Letouzey. "The SQALE method for evaluating Technical Debt". En: (jun. de 2012). DOI: 10.1109/MTD.2012.6225997.

[19] ZEN PROGRAM LTD. *Improve your .NET code quality with NDepend*. URL: https://www.ndepend.com/ (visitado 21-04-2018).

[20] Thomas J. McCabe. "A Complexity Measure". En: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, págs. 407-. URL: http://dl.acm.org/citation.cfm?id=800253.807712.

[21] Meine J.P van der Meulen y Miguel A. Revilla. "Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs". En: *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)* (nov. de 2007), págs. 203-208.

[22] Microsoft. *TypeScript - JavaScript that scales*. URL: https://www.typescriptlang.org/ (visitado 22-04-2018).

[23] Applied Software Engineering Research Group - Federal University of Minas Gerais. *aserg-ufmg/JSCity: Visualizing JavaScript source code as navigable 3D cities*. URL: https://github.com/aserg-ufmg/JSCity (visitado 22-04-2018).

[24] Stack Overflow. *Stack Overflow Developer Survey 2018*. Por sexto año consecutivo JavaScript aparece como el lenguaje más usado. URL: https://insights.stackoverflow.com/survey/2018/#most-popular-technologies (visitado 23-04-2018).

[25] Jarrod Overson. *es-analysis/plato: JavaScript source visualization, static analysis, and complexity tool*. URL: https://github.com/es-analysis/plato (visitado 22-04-2018).

[26] Mariano Martinez Peck. *OSSubprocess*. URL: https://github.com/pharo-contributions/OSSubprocess (visitado 23-12-2018).

[27] Richard Wettel y Michele Lanza. "CodeCity". En: *Proceedings of WASDeTT 2008 (1st International Workshop on Advanced Software Development Tools and Techniques)*. 2008.

[28] Juriy Zaytsev. *ECMAScript 5 Compatibility Table*. URL: http://kangax.github.io/compat-table/es5/ (visitado 22-04-2018).

Apéndice A

Proyectos analizados en HUNTER

A continuación se muestra en detalle la URL y commits de los proyectos analizados en los experimentos detallados en la Sección 5.2.

Proyecto	URL Commit
LocalForage	https://github.com/localForage/localForage.git 349a8743b08036078aaea9d88de89177345e5dc4
Hexo	https://github.com/hexojs/hexo.git e16f19af231d4400d4267f51aaf6e57f9c5a55b8
es6-mario	https://github.com/JuniorTour/es6-mario.git 9dfcf1f675f1c81ed0e1f72bc0376c3de1c06b50
serverless	https://github.com/serverless/serverless.git a9e243f60927c82a6447a0a9062c31e5eb94d934