

- Trabajo Final - Ingeniería de Sistemas -

Análisis de detección de Code Smells para el lenguaje JavaScript



- Facultad de Ciencias Exactas -
- UNICEN -

Alumna

Malavolta, Antonela

Director

Vidal, Santiago A.

Co-Directora

Marcos, Claudia A.

Índice

Capítulo 1: Introducción	7
1.1 Calidad del software	7
1.2 Catálogo de métricas para la detección de Code Smells	8
1.3 JavaScript	8
1.4 Análisis de Code Smells en JavaScript	10
1.4.1 vcomplex, una herramienta para obtener métricas para el lenguaje JavaScript.....	11
1.4.2 Identificación de Code Smells	12
1.5 Esquema general	12
 Capítulo 2: Mantenimiento de sistemas	 14
2.1 Calidad del software	14
2.2 Mantenimiento de software	15
2.3 Técnicas y estrategias de mantenimiento de software	17
2.4 Refactoring	18
2.5 Code Smells	19
2.6 Métricas	21
2.7 JavaScript	23
2.7.1 Variables	25
2.7.2 Funciones	27
 Capítulo 3: Trabajos relacionados	 29
3.1 Estimación de calidad de JavaScript	30
3.2 JSNOSE: detección de Code Smells en JavaScript	32
3.3 Uso de ejemplos para detectar Code Smells en JavaScript	34
3.4 Eliminación de Code Smells en JavaScript con Programación Orientada a Objetos (OOP)	36
3.5 Herramientas	38
3.5.1 escomplex	38
3.5.2 Gulp-complexity	39
3.6 Resumen	40
 Capítulo 4: Análisis de Code Smells	 42
4.1 Brain Method	42
4.1.1 Descripción	42
4.1.2 Detección	42
4.1.3 Análisis de las métricas en JavaScript	44
4.1.4 Conclusión Code Smell Brain Method	49
4.2 Data Class	49

4.2.1 Descripción	49
4.2.2 Detección	50
4.2.3 Análisis de las métricas en JavaScript	52
4.2.4 Conclusión Code Smell Data Class	58
4.3 God Class	58
4.3.1 Descripción	58
4.3.2 Detección	59
4.3.3 Análisis de las métricas en JavaScript	60
4.3.4 Conclusión Code Smell God Class	64
4.4 Intensive Coupling	65
4.4.1 Descripción	65
4.4.2 Detección	66
4.4.3 Análisis de las métricas en JavaScript	68
4.4.4 Conclusión Code Smell Intensive Coupling	73
4.5 Dispersed Coupling	73
4.5.1 Descripción	73
4.5.2 Detección	74
4.5.3 Análisis de las métricas en JavaScript	75
4.4.4 Conclusión Code Smell Dispersed Coupling	75
4.6 Resumen del análisis de Code Smells	76
Capítulo 5: Casos de estudios	78
5.1 Herramienta vcomplex	78
5.2 Descripción de los casos de estudio	79
5.3 Aplicación #1: Browserify	82
5.3.1 Detección de Code Smells y análisis de métricas	82
5.3.2 Brain Method	82
5.3.3 Intensive Coupling	86
5.3.4 Dispersed Coupling	87
5.4 Aplicación #2: Gulp	89
5.4.1 Detección de Code Smells y Análisis de Métricas	90
5.4.2 Brain Method	90
5.4.3 Intensive Coupling	92
5.5 Aplicación #3: Bower	93
5.5.1 Detección de Code Smells y análisis de métricas	93
5.5.2 Brain Method	94
5.5.3 God Class	100
5.5.4 Intensive Coupling	101
5.5.5 Dispersed Coupling	104
5.6 Resumen de los Casos de Estudio	106

Capítulo 6: Conclusión	108
6.1 Conclusiones finales	108
6.2 Contribuciones	109
6.3 Limitaciones	110
6.4 Trabajos futuros	111
Bibliografía	112

Índice de Figuras

1.1 Encuesta anual a desarrolladores por Stack Overflow	9
1.2 Flujo de ejecución para la detección de posibles Code Smells en JavaScript a partir de la herramienta vcomplex	11
2.1 Tipos de mantenimiento con su porcentaje de uso	16
2.2 Código ejemplo para conceptos básicos de JavaScript	26
3.1 Algoritmo de detección de code smells de la herramienta JSNOSE en JavaScript	34
3.2 Flujo general de [26]	35
3.3 Ejemplo de código JavaScript a refactorizar	37
3.4 Ejemplo de código JavaScript refactorizado	37
4.1 Estrategia de detección del Brain Method	43
4.2 Función draw en lenguaje JavaScript. Código ejemplo Brain Method	46
4.3 Función draw en lenguaje Java	47
4.4.1 Estrategia de detección del Code Smell Data Class	50
4.4.2 Estrategia de detección del Code Smell Data Class	51
4.5 Archivo js en el lenguaje JavaScript. Código ejemplo Data Class	54
4.6 Cálculo métrica CYCLO para código de ejemplo	58
4.7 Estrategia de detección del Code Smell God Class	59
4.8 Archivo js en el lenguaje JavaScript. Código ejemplo God Class	62
4.9 Ilustración del Code Smell Intensive Coupling	65
4.10.1 Estrategia de detección del Code Smell Intensive Coupling	66
4.10.2 Estrategia de detección del Code Smell Intensive Coupling	67
4.11 Archivo js en el lenguaje JavaScript. Código ejemplo Intensive Coupling	70
4.12 Ilustración del Code Smell Dispersed Coupling	73
4.13.1 Estrategia de detección del Code Smell Dispersed Coupling	74
4.13.2 Estrategia de detección del Code Smell Dispersed Coupling	75
5.1 Obtención de métricas con vcomplex, y cálculo de Code Smells	78
5.2 Ocurrencias de Code Smells en la aplicación Browserify	83
5.3 Función globalTr	84
5.4 Archivos que se usan desde la función globalTr del archivo index.js	86
5.5 Archivos js que se usan desde la función _createPipeline del archivo index.js	88
5.6 Función _createPipeline	89
5.7 Función handleArguments	91
5.8 Archivos js que se usan desde la función handleArguments del archivo gulp.js	92
5.9 Ocurrencias de Code Smells en la aplicación Bower	93
5.10 Archivo register.js que contiene la función register	97

5.11 Función <code>_checkout</code>	99
5.12 Archivos js que se usan desde la función <code>ensurePackage</code> del archivo <code>packages-svn.js</code>	102
5.13 Función <code>ensurePackage</code>	103
5.14 Archivos js que se usan desde la función <code>_readJson</code> del archivo <code>Project.js</code>	104
5.15 Función <code>_readJson</code>	105

Índice de Tablas

1.1 Resumen del análisis de Code Smells y métricas	11
2.1 Code Smells catalogados por Lanza y Marinescu	21
2.2 Métricas catalogadas por Lanza y Marinescu	23
3.1 Estructuras de control básicas y pesos	31
3.2 Código de ejemplo para el cálculo de la complejidad cognitiva (JCCM) en JavaScript	31
3.3 Comparación de métricas de complejidad	32
3.4 Code Smells que se detectan en el análisis que utiliza la herramienta JSNOSE	33
4.1 Resumen del análisis de Code Smells	76
4.2 Resumen del análisis de métricas	76
5.1 Umbrales para las métricas CYCLO, WMC y LOC	80
5.2 Umbrales para las métricas normalizadas	81
5.3 Umbrales para métricas con valores absolutos	81
5.4 Condiciones con métricas y umbrales para el Code Smell Brain Method	81
5.5 Condiciones con métricas y umbrales para el Code Smell God Class	81
5.6 Condiciones con métricas y umbrales para el Code Smell Intensive Coupling	82
5.7 Condiciones con métricas y umbrales para el Code Smell Dispersed Coupling	82
5.8 Valores de métricas de las ocurrencias del Code Smell Brain Method en la aplicación Browserify	83
5.9 Valores de métricas de las ocurrencias del Code Smell Intensive Coupling en la aplicación Browserify	86
5.10 Valores de métricas de las ocurrencias del Code Smell Dispersed Coupling en la aplicación Browserify	87
5.11 Valores de métricas de la ocurrencia del Code Smell Brain Method en la aplicación Gulp	90
5.12 Valores de métricas de la ocurrencia del Code Smell Intensive Coupling en la aplicación Gulp	92
5.13 Valores de métricas de las ocurrencias del Code Smell Brain Method en la aplicación Bower	96
5.14 Valores de métricas de las ocurrencias del Code Smell God Class en la aplicación Bower	100
5.15 Valores de métricas de las ocurrencias del Code Smell Intensive Coupling en la aplicación Bower	102
5.16 Valores de métricas de las ocurrencias del Code Smell Dispersed Coupling en la aplicación Bower	104

Capítulo 1

Introducción

1.1 Calidad del software

En ingeniería de software, calidad es "la totalidad de aspectos y características de un producto o servicio que tienen que ver con su habilidad de satisfacer las necesidades explícitas o implícitas" [4]. En los sistemas de software es muy importante centrar el desarrollo en su calidad [5,6]. Además, la calidad del software no se refiere únicamente a obtener un producto sin errores, sino también a que su código fuente se pueda extender, modificar y entender sin que ésto resulte un problema con el paso del tiempo [7].

A medida que los sistemas se hacen más grandes y complejos, una preocupación frecuente es la existencia de problemas de diseño estructurales que afecten la calidad del mismo. Dichos problemas de diseño, si no se tratan en etapas tempranas del desarrollo, producen problemas durante el mantenimiento y la evolución de los sistemas. Algunos de estos problemas se denominan Code Smells [2]. Un code smell es un síntoma en el código que ayuda a identificar un problema de diseño, permitiendo a los desarrolladores detectar fragmentos de código que deben reestructurarse para mejorar la calidad del sistema. La existencia y aumento de code smells generan una degradación de la calidad del software, impactando fuertemente sobre la comprensión, mantenibilidad y evolución del código fuente [8].

Si bien el uso del paradigma orientado a objetos ayuda a que los sistemas de software sean más flexibles, extensibles y comprensibles, y por lo tanto más fáciles de mantener [1], es muy común durante el proceso de desarrollo el uso de "malas prácticas" de programación. Estas malas prácticas (por ejemplo, una clase que engloba demasiada funcionalidad, un método que usa muchas variables o la generación de código duplicado) son ejemplos de code smells. No son errores, pero indican debilidades en el diseño que pueden complejizar el desarrollo o incrementar el riesgo de errores o fallos en el futuro.

Para mejorar el código y que sea mantenible y esté bien modularizado, se utiliza la técnica de refactoring [9]. La identificación de code smells en el código es el primer paso de esta técnica. Al refactorizar, se modifica la estructura interna de un sistema de software, con la finalidad de que sea más fácil de entender y modificar, sin alterar el comportamiento externo del mismo [10]. La tarea de refactorización es llevada a cabo por los desarrolladores como parte del proceso de desarrollo de software. Lo que se pretende es mejorar el código fuente de tal manera que el costo y el esfuerzo de realizar modificaciones sobre el sistema sea menor respecto a hacerlo sobre un sistema sin refactorizar [2].

1.2 Catálogo de métricas para la detección de Code Smells

Para el lenguaje Java, se encuentran definidos catálogos [1] [3], que por medio de métricas, permiten identificar code smells en el código fuente. A partir de las métricas y la identificación de los code smells, se detectan problemas en el código para poder resolverlos en una etapa posterior. Por lo tanto, los catálogos basados en métricas son útiles para mantener un orden y una estrategia clara para la detección de code smells en el lenguaje que se utilicen.

Lanza y Marinescu, presentan el catálogo de métricas para la detección de code smells en el lenguaje Java a partir de la definición de métricas [3]. Este catálogo, es muy importante para detectar los code smells que se pueden presentar en el código Java y que son muy comunes en el desarrollo. Además, el diseño de aplicaciones de gran tamaño es difícil debido a la complejidad intrínseca de los dominios modelados. A esta complejidad intrínseca, se añade otro factor que proviene de procesos de negocio, problemas organizativos, humanos y otros factores externos [3].

Actualmente, para lenguajes populares como JavaScript, no se han presentado catálogos.

1.3 JavaScript

JavaScript es uno de los lenguajes de programación más potentes e importantes en la actualidad [51]. Algunas de sus ventajas con respecto a otros lenguajes son su practicidad, utilidad y disponibilidad en cualquier navegador web. Se puede utilizar para la programación frontend, agregando mayor interactividad a la web, y también en los servidores web. El entorno de ejecución Node.js es la opción más utilizada para usar este lenguaje del lado del servidor.

El lenguaje JavaScript se rige bajo la especificación del estándar ECMAScript [4]. Dicho estándar define un lenguaje de tipos dinámicos inspirado en Java y otros lenguajes del estilo de C, y soporta algunas características de la programación orientada a objetos. Para la ejecución de los sistemas desarrollados bajo el estándar ECMAScript, se utiliza el entorno de ejecución Node.js. Este entorno, es el más popular y utilizado por los desarrolladores del lenguaje JavaScript. Además, Node.js adopta el concepto de módulos mediante la adopción de la especificación de módulos propuesta por CommonJS [17].

Según la encuesta que realiza el sitio web Stack Overflow, que es el más utilizado por la comunidad de desarrolladores para encontrar soluciones a problemas de programación en diferentes lenguajes, el lenguaje JavaScript es el más utilizado actualmente [11] (Figura 1.1).

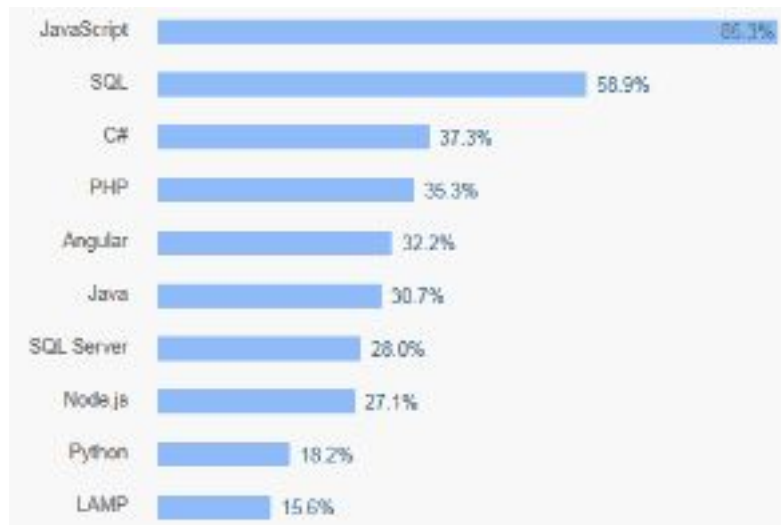


Figura 1.1 - Encuesta anual a desarrolladores por Stack Overflow.

Otras características importantes que se deben destacar del lenguaje JavaScript, que difieren de las características del lenguaje Java, son las siguientes:

- No define métodos, define los objetos y funcionalidades como una función (function). Otra manera de definir objetos es por medio del uso de prototipos.
- No define clases, sino que provee archivos js que contienen las propiedades y la funcionalidad representativa del mismo.
- Se definen dos ámbitos: local y global.
- Las variables se declaran mediante el uso de la palabra reservada var y serán locales a la función o al archivo js donde se declaren.
- Las variables pueden no declararse, en este caso se definen como variables globales al archivo js que las contiene.
- Las variables se pueden definir con this, en este caso también son globales y, además, pueden ser utilizadas desde otro archivo js.
- Para utilizar una función desde otro archivo js, se debe exportar el módulo que contiene la misma o la función directamente desde el archivo que la contiene.

JavaScript ha evolucionado en relación a la sintaxis, funcionalidad y características en general, pero hasta el momento no se ha consensuado en la comunidad un proceso o funcionalidad para abordar problemas de calidad de los sistemas de software relacionados con mantenibilidad. Por lo tanto, la detección de code smells en el lenguaje JavaScript es una problemática que aún no presenta demasiadas soluciones.

A pesar de que es muy importante la detección de code smells para lograr que el código sea mantenible y se eviten errores en el futuro, existen muy pocos trabajos relacionados a la detección de code smells en JavaScript.

1.4 Análisis de Code Smells en JavaScript

En este trabajo se propone el análisis de code smells para la detección de los mismos en el lenguaje JavaScript a partir de la creación de un catálogo basado en métricas, y considerando el catálogo de métricas propuesto por Lanza y Marinescu [1].

En el presente trabajo, se analizaron las métricas que se definen en la detección de code smells en el lenguaje Java, y se adaptaron hacia su uso en el lenguaje JavaScript. Para ello se tomó como base el catálogo propuesto por Lanza y Marinescu para el lenguaje Java. Debido a las diferencias entre los dos lenguajes, se identificó que algunos code smells se pueden aplicar sin modificaciones previas, otros son adaptados de acuerdo a las características de JavaScript y, otros smells, no tienen sentido en el contexto de JavaScript.

En el análisis que se realizó, se seleccionaron los code smells más significativos, en cuanto a su relación con el lenguaje JavaScript y de acuerdo a la posibilidad de realizar un buen análisis de los mismos, de la lista de code smells propuesta por Lanza y Marinescu (Tabla 2.1). Se detectaron las métricas que se utilizan en las estrategias de detección de los code smells que se seleccionaron, y a partir del análisis de cada uno de ellos, se determinó si era posible o no detectar el code smell en el lenguaje JavaScript. De esta manera, se obtuvo el catálogo de métricas para el lenguaje JavaScript que da lugar a la detección de code smells en el código que se implementa.

En la Tabla 1.1 se muestran los resultados que se obtuvieron luego del análisis que se realizó. En la misma se pueden ver los cinco code smells que se analizaron, y por cada uno se detalla: si es posible detectarlos en el lenguaje JavaScript, las métricas que se encuentran involucradas en la estrategia de detección de cada code smell y si es necesario, y posible, adaptarlas.

Code Smell	Detección	Métricas	Adaptación
Brain Method	Es posible detectar	LOC	No es necesario adaptarla
		CYCLO	No es necesario adaptarla
		MAXNESTING	No es necesario adaptarla
		NOAV	No es necesario adaptarla
Data Class	No es posible detectar	WOC	Se adapta
		NOAP	Se adapta
		NOAM	No se puede adaptar
		WMC	No es necesario adaptarla
God Class	Es posible detectar	ATFD	Se adapta
		WMC	No es necesario adaptarla
		TCC	No es necesario adaptarla
Intensive Coupling	Es posible adaptar	MAXNESTING	No es necesario adaptarla
		CINT	Se adapta
		CDISP	Se adapta
Dispersed Coupling	Es posible adaptar	MAXNESTING	No es necesario adaptarla

		CINT	Se adapta
		CDISP	Se adapta

Tabla 1.1: Resumen del análisis de Code Smells y métricas.

Luego de realizar el análisis correspondiente, para obtener las métricas necesarias para la detección de los code smells en el lenguaje JavaScript, se optó por implementar una herramienta que proporciona el valor de las métricas para luego reemplazar dichos valores en las estrategias de detección de los code smells, y así poder identificar posibles code smells en el código JavaScript.

1.4.1 vcomplex, una herramienta para obtener métricas para el lenguaje JavaScript

Al finalizar el análisis de code smells, y definir qué métricas se pueden utilizar en el lenguaje JavaScript, se implementó la herramienta de detección de métricas vcomplex, para obtener el valor de las mismas y detectar posibles code smells en el código de la aplicación a analizar.

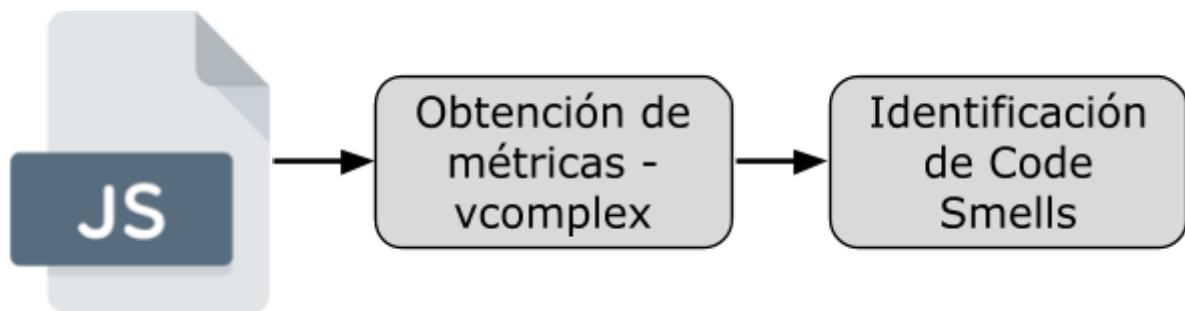


Figura 1.2: Flujo de ejecución para la detección de posibles Code Smells en JavaScript a partir de la herramienta vcomplex.

La Figura 1.2, muestra un esquema de ejecución para la detección de posibles code smells mediante el uso de la herramienta vcomplex. El esquema de ejecución, tiene como entrada los archivos js del sistema de software que se desea “medir”, y como salida la identificación de code smells a partir de los valores de métricas obtenidos al ejecutar la herramienta vcomplex.

En la detección de métricas, se analiza la sintaxis de cada archivo js, y se calculan las métricas para el mismo y para cada función que contiene. La evaluación de sintaxis, se realiza a partir de la interpretación de árboles de sintaxis que se obtienen con el analizador de ECMAScript, Esprima [18].

A partir de los valores de métricas que se obtienen, el siguiente paso es identificar los posibles code smells en el código mediante sus estrategias de detección, y definir si verdaderamente se trata de un code smell.

1.4.2 Identificación de Code Smells

Como se mencionó anteriormente, luego de ejecutar la herramienta vcomplex se identifican los posibles code smells, y a partir de los resultados que se obtienen se evalúa si el code smell detectado es realmente un code smell en el lenguaje JavaScript (Figura 1.2). Para ello se debe analizar el código que representa el problema y definir a partir de las características del code smell y del lenguaje si es o no un code smell.

En este trabajo se ejecutó la herramienta vcomplex para tres aplicaciones diferentes: Browserify [12], Gulp [19] y Bower [20]. Para cada una de las aplicaciones, se seleccionaron diferentes funciones y archivos js que se detectaron como posibles code smells. Se realizó el análisis de los mismos, y por cada uno se definió si verdaderamente se trata de un smell en el lenguaje JavaScript.

Por lo tanto, a partir del análisis de code smells en el lenguaje JavaScript y la definición de un catálogo basado en métricas para el mismo, se implementó la herramienta vcomplex para automatizar el proceso de búsqueda de code smells.

Las contribuciones del trabajo final son las siguientes: Se logró analizar code smells seleccionados del catálogo de métricas de Lanza y Marinescu [1] pensado para el lenguaje Java para definir si es posible o no su detección en el lenguaje JavaScript. Se implementó la herramienta vcomplex. Dicha herramienta realiza el proceso de detección automática de métricas en el lenguaje JavaScript. Además, se identifican posibles code smells reemplazando los valores de las métricas que se obtienen al ejecutar vcomplex en las estrategias de detección de los code smells.

1.5 Esquema general

El esquema general de este trabajo final está organizado de la siguiente manera:

En el Capítulo 2 se describen los conceptos relacionados a la evolución del software y el mantenimiento. En el mismo, se informa sobre los conceptos utilizados en el análisis que se realiza en este trabajo, como lo son los conceptos de refactorización, code smells y métricas. Además, se presentan conceptos básicos de JavaScript.

En el Capítulo 3 se presentan varios análisis relacionados con la detección de code smells a partir del uso de diferentes técnicas. Además, se detallan herramientas que obtienen el valor de métricas para la detección de code smells.

En el Capítulo 4 se presenta el análisis realizado de code smells para el lenguaje JavaScript. Se analiza cada code smell en particular, y se define si se pueden o no detectar en el lenguaje JavaScript a partir del análisis de sus métricas.

En el Capítulo 5 se analizan tres aplicaciones como casos de estudio. Para cada aplicación se obtienen los valores de las métricas a partir de la herramienta implementada, vcomplex, se detectan posibles code smells, y se analizan para identificar si verdaderamente se trata de un code smell. Por último, se desarrollan conclusiones de los resultados.

Finalmente, el Capítulo 6 resume las conclusiones, principales contribuciones, limitaciones y trabajos futuros.

Capítulo 2

Mantenimiento de sistemas

Este capítulo servirá de introducción al lector para que se familiarize con los conceptos utilizados a lo largo de este trabajo.

En el mismo, se introducirán conceptos básicos relacionados a la evolución de sistemas de software y los problemas que esto conlleva con el paso del tiempo. Se relacionarán los conceptos de code smells y métricas y, además, se introducirán conceptos relacionados al lenguaje JavaScript que se utilizaron en este trabajo.

2.1 Calidad del software

En la ingeniería de software, calidad es "la totalidad de aspectos y características de un producto o servicio que tienen que ver con su habilidad de satisfacer las necesidades explícitas o implícitas" [4]. En los sistemas de software es muy importante centrar el desarrollo en su calidad [5,6]. Además, la calidad del software no se refiere únicamente a obtener un producto sin errores, sino también a que su código fuente se pueda extender, modificar y entender sin que ésto resulte un problema con el paso del tiempo [7].

Al finalizar el desarrollo de un sistema y ponerlo en funcionamiento, su utilidad comienza a degradarse progresivamente debido a los cambios en el entorno. Dichos cambios, pueden causar modificaciones en los requerimientos, creación de nuevas funcionalidades y cambios en las reglas de negocios, entre otros. Los cambios, se implementan modificando los componentes del sistema existente y añadiendo nuevos componentes al sistema donde sea necesario [36].

Cuando un sistema evoluciona, éste se vuelve más complejo y se necesita de recursos adicionales para preservar y simplificar su estructura [21]. Si los sistemas no se adaptan ante los cambios en su entorno, su utilidad se verá afectada con el paso del tiempo [21]. El mantenimiento es una tarea primordial para evitar el envejecimiento del software y facilitar la evolución de los sistemas.

Los requerimientos evolucionan y es por eso que el sistema tiene que hacer lo propio para no quedar obsoleto [22]. Lehman en sus leyes hace referencia a esta problemática relacionada con el mantenimiento del software:

- Decremento de la calidad: La calidad de los sistemas software comenzará a disminuir a menos que dichos sistemas se adapten a los cambios de su entorno.

- Cambio continuo: Un sistema que se utiliza en un ambiente del mundo real debe cambiar o progresivamente será menos útil en ese ambiente.
- Complejidad creciente: A medida que un sistema evoluciona, su estructura se vuelve más compleja y se necesita de recursos adicionales para preservar y simplificar su estructura.
- Crecimiento continuo: La funcionalidad ofrecida por los sistemas tiene que crecer continuamente para mantener la satisfacción de los usuarios.

Generalmente, al momento de realizar la tarea de mantenimiento de un sistema, se hace difícil adaptarlo a los nuevos requerimientos debido a que el código es extenso y complejo [22]. Esta tarea, inevitablemente, provoca que el código deba ser modificado por diferentes motivos relacionados a los cambios en el entorno ya mencionados.

2.2 Mantenimiento de software

El mantenimiento de software es una de las tareas más importantes y costosas en el ciclo de desarrollo de software debido a que los sistemas son cada vez más complejos y están sujetos a constantes cambios [9].

El mantenimiento del software se define como:

- El proceso de modificación de un sistema de software o un componente luego de su entrega con el fin de corregir errores, mejorar la performance u otros atributos, o adaptarlo a un ambiente cambiante [23].
- Un producto de software recae en modificaciones de código y documentación asociada debido a un problema o la necesidad de una mejora. El objetivo es modificar el producto de software existente preservando su integridad [35].

Los cambios realizados al software pueden ser cambios sencillos para corregir errores de código, cambios más extensos para corregir errores de diseño o mejoras significativas para corregir errores de especificación o soportar nuevos requerimientos.

En [38] se definen las razones que motivan el mantenimiento y se clasifican en cuatro tipos:

- Adaptativo: Es la modificación del sistema realizada luego que éste fue entregado con el fin de mantenerlo operativo debido a un ambiente cambiante o que ya ha cambiado.
- Perfectivo: Es la modificación del sistema realizada luego que éste fue entregado con el fin de mejorar su performance o mantenimiento.

- **Correctivo:** Es la modificación del sistema, realizada luego que éste fue entregado, con el objetivo de corregir fallos. Por lo general, los errores de código son relativamente sencillos de corregir, los errores de diseño son más costosos ya que implican modificar varios componentes de los programas. Los errores de requerimientos son los más costoso de reparar debido a que puede ser necesario un rediseño extenso del sistema
- **Preventivo:** Es la modificación proactiva del sistema con el fin de evitar problemas en el futuro.

Lientz and Swanson [40] indican en su estudio que el mantenimiento perfectivo comprende alrededor del 50% de las modificaciones, el adaptativo el 25%, el correctivo el 20% y el preventivo solo el 5% (Figura 2.1).

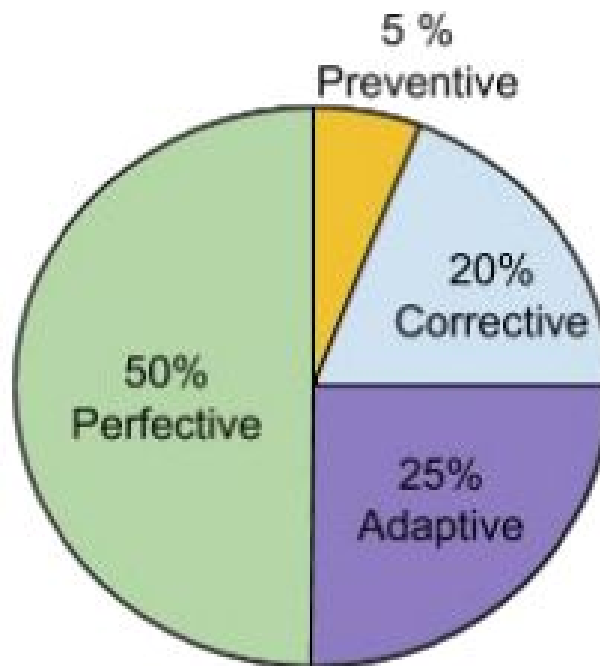


Figura 2.1: Tipos de mantenimiento con su porcentaje de uso.

Como se muestra en la Figura 2.1, los tipos de mantenimiento más utilizados en el desarrollo de software son el mantenimiento adaptativo y el perfectivo. Esto es así por dos razones: la primera es porque las tecnologías y el hardware cambian, lo cual lleva a que se tenga que adaptar el código a estos cambios, y la segunda razón es debido a la evolución del software que lleva a un cambio constante en los requerimientos aumentando así el mantenimiento perfectivo. El mantenimiento correctivo hace referencia a los bugs del sistema, que son reparados a medida que son encontrados. Y por último, el mantenimiento preventivo, que es el menos utilizado, ya que es difícil de prever las áreas donde es necesario hacer énfasis y tampoco es fácil la capacidad para prevenir bugs que todavía no sucedieron.

Un error que se comete usualmente, es que se piensa al software como un producto al que se le puede definir una etapa de finalización, y que se termina de construir en un momento dado. Se plantea en [39] que una primera versión es solo eso, una "primera versión" dentro de un proceso de evaluación continua.

Cuando el software evoluciona es importante identificar cuáles son los componentes estables y cuáles necesitan que se aplique un mantenimiento correctivo. En los componentes que se identifica que es necesario el mantenimiento, es necesario trabajar sobre la legibilidad, extensibilidad, correctitud y calidad ya que probablemente vuelva a ser modificado en un futuro. Si las sucesivas modificaciones no son efectuadas correctamente, el código se vuelve difícil de mantener, se introducen nuevos bugs y se vuelve más difícil adaptar el sistema a nuevos requerimientos.

Dada la importancia de la tarea de mantenimiento de software y su necesidad, es necesario proveer técnicas y estrategias que faciliten dicha actividad.

2.3 Técnicas y estrategias de mantenimiento de software

En la ingeniería de software se sabe que los cambios serán necesarios, ya sea en el presente o a futuro en la vida de los sistemas que se implementen. Actualmente, los enfoques tradicionales de la ingeniería de software en muchas ocasiones carecen de mecanismos que permitan anticipar los cambios.

Una de las estrategias que se ha propuesto tradicionalmente es diseñar para el cambio con lo cual se trata de prevenir los eventuales problemas que pueden surgir con la evolución del software [41]. Con este propósito algunas de las técnicas que se han utilizado son: ocultamiento de información, separación de concerns, interfaces, capas [42], herencia, polimorfismo, patrones de diseño [43], documentación actualizada [44].

No importa qué tan bien se diseñe un sistema, siempre pueden surgir modificaciones imprevistas que dificultan la tarea de mantenimiento. Por otro lado, diseñar un sistema desde cero no es algo que se da comúnmente en la ingeniería de software, sino que es algo que se da muy raramente. La mayoría de los ingenieros son asignados a mantener y/o evolucionar aplicaciones existentes. Tales sistemas sufren de problemas típicos, como documentación obsoleta, diseño complejo, mecanismos de parches, duplicación de código, componentes obsoletos, pérdida de los desarrolladores originales, etc. [45][46].

Una de las técnicas más utilizadas dentro del contexto de la programación orientada a objetos, es la técnica de refactoring. El refactoring lleva a un código a ser mantenible y extensible. Lo cual hace al sistema evolucionable y adaptativo ante los cambios de requerimientos. El

mantenimiento perfectivo y adaptativo están fuertemente ligados con el refactoring de un sistema.

2.4 Refactoring

Refactorizar se describe como el proceso de cambio de la estructura interna de un sistema de software, con el objetivo de que sea más fácil de entender y menos costoso de modificar, conservando el comportamiento externo del mismo. Es una técnica disciplinada para optimizar el código, y minimizar las posibilidades de introducir errores [8].

Esta técnica, que incrementa la legibilidad, la mantenibilidad y permite mejorar el diseño de software mediante la modularización, es llevada a cabo como parte del proceso de desarrollo de software por los desarrolladores. Con la misma, se mejora el código fuente de tal forma que el esfuerzo y costo de realizar modificaciones sobre el sistema sea menor respecto a hacerlo sobre un sistema sin refactorizar.

Debido a las características que tiene el paradigma orientado a objetos, a refactorización es la técnica utilizada para la reestructuración de código en sistemas orientados a objetos. Algunas de las actividades que se realizan con este objetivo son: mejorar atributos de calidad como la facilidad de comprender el sistema, reducir el código duplicado, mejorar la cohesión, reducir el acoplamiento, flexibilidad, mantenimiento, etc.

Se pueden resumir las actividades de la técnica de refactoring en los siguientes pasos:

1. Identificar los lugares del sistema que podrían ser refactorizados.
2. Determinar qué refactorings serán aplicados a cada uno de los lugares identificados previamente.
3. Verificar que la aplicación de los refactorings preserve el comportamiento del sistema.
4. Aplicar los refactorings.
5. Evaluar los efectos del refactoring realizado sobre los atributos de calidad del software (por ejemplo, complejidad, mantenimiento, etc.) o sobre los procesos (por ejemplo, productividad, costo, esfuerzo).
6. Mantener la consistencia entre el código refactorizado y el resto de los artefactos de software (por ejemplo, documentación, requerimientos, tests, etc.).

Como se menciona en el primer paso, el proceso de refactoring empieza con la identificación de oportunidades de refactorización. Fowler [8], sugiere la detección de code smells como una de las formas más eficaces para identificar dichas oportunidades. Los code smells indican la presencia de posibles problemas de diseño estructurales y dan lugar a la posibilidad de aplicar determinados refactorings.

2.5 Code Smells

Un code smell, también llamados bad smell, se refiere a cualquier síntoma que puede indicar que el código fuente presenta problemas [48], o como su nombre lo indica, tiene “mal olor”. Los code smells surgen debido a “malas prácticas” empleadas durante el proceso de desarrollo de software, como por ejemplo, métodos muy largos, código duplicado, clases muy grandes, métodos que engloban funcionalidad en exceso, etc. [49].

Un desarrollador puede intuir que su sistema contiene code smells cuando se encuentra [48]:

- **Código duplicado:** Este problema ocurre cuando en una misma clase se repite mucho código. El código duplicado lleva a programas difíciles de modificar. Para solucionarlo se deben eliminar las líneas que se repiten varias veces por ser exactamente iguales. También se deben eliminar las líneas de código que poseen estructuras similares.
- **Métodos muy largos:** Cuando los métodos son extremadamente largos se dificulta mucho su entendimiento, volviéndose difícil encontrar errores o agregar funcionalidad. Las clases que poseen este tipo de métodos se vuelven difíciles de mantener. En este caso, para solucionarlo, hay que dividir el código en porciones y extraerlas para crear métodos más pequeños, que sean más fáciles de comprender, reusar y mantener.
- **Clases muy grandes:** Este problema puede significar una mala división de responsabilidad de los objetos. Las clases con tamaño fuera de lo normal conducen a que un sistema sea complejo de entender y extender. Por lo tanto, se deben identificar las funcionalidades de la clase y ver si realmente dichas funcionalidades se encuentran relacionadas. Si no es así, se debe realizar una correcta división de la funcionalidad de la clase logrando clases más pequeñas, más entendibles y, por lo tanto, mantenibles.
- **Métodos que necesitan muchos parámetros:** Este problema provoca que un método sea poco mantenible, debido a que cuando hay un alto acoplamiento entre métodos, se vuelve inevitable pasar como parámetros muchas variables. Una solución es realizar una clase que contenga dichos parámetros y, luego, utilizar esa clase en el pasaje por parámetros.
- **Instrucciones Switch-Case:** Usualmente, una sentencia de este tipo, se repite en varios sitios del código, aunque en cada uno de ellos se realicen actividades distintas. Para solucionar este tipo de problemas, se aplica de diferentes formas el polimorfismo, para evitar que sea necesario usar este tipo de sentencia de control.

Los code smells, generalmente no son técnicamente incorrectos, no son errores, y no interfieren en el funcionamiento normal de un sistema de software. Sin embargo, aumentan el riesgo a fallos, afectan el diseño del sistema y hacen que el sistema se vuelva más lento en cuanto al desarrollo y mantenimiento del mismo [1].

Entre los catálogos de code smells más destacados, se encuentran los catálogos presentados por Fowler [8] y Lanza y Marinescu [1]. En la tabla 2.1 se muestra el resumen de los code smells presentes en el catálogo de Lanza y Marinescu, algunos de los cuales también se encuentran en el catálogo de Fowler.

Tipo de Code Smell	Descripción
God Class	Son clases que tienden a centralizar la inteligencia del sistema. Una God Class realiza demasiado trabajo por cuenta propia, delega sólo detalles menores a otras clases. Impacta en la reusabilidad y comprensibilidad del sistema
Feature Envy	Son métodos que están más interesados en los datos de otras clases que en los propios. Esto podría ser una señal de que el método está fuera de lugar y debe ser movido a otra clase.
Data Class	Son clases que almacenan datos, no tienen funcionalidad compleja, o directamente su funcionalidad es nula, y otras clases dependen en gran medida de sus datos. Es un síntoma de una mala encapsulación de datos y funcionalidad.
Brain Method	Es un método difícil de mantener o entender. Tienden a centralizar la funcionalidad de una clase, de la misma manera que una God Class centraliza toda la funcionalidad de todo un sistema.
Brain Class	Es una Clase compleja que tiende a contener una cantidad excesiva de responsabilidades, por lo general contiene varios métodos afectados por el code smell Brain Method. Se diferencia de God Class por no acceder a datos de otras clases de manera abusiva.
Significant Duplication	Hace referencia a código duplicado que resulte significativo. Esto quiere decir que sea suficientemente grande.
Intensive Coupling	Son métodos con demasiadas dependencias que se encuentran dispersas en una o pocas clases.
Dispersed Coupling	Son métodos con demasiadas dependencias que se encuentran dispersas en muchas clases. Los métodos invocados suelen ser muy simples.
Shotgun Surgery	Se trata de la situación inversa al Dispersed Coupling. Hace referencia a cuando el cambio en un método genera muchos pequeños cambios en otros métodos y clases.
Refused Bequest	Hace referencia a cuando, en una jerarquía, un padre define métodos protegidos que no son utilizados por sus hijos. Esto es un indicio de que algo es incorrecto en la clasificación de relación entre clases.
Tradition Breaker	Hace referencia a una clase que no especializa la clase de la cual hereda,

	y sólo agrega nuevos servicios que no dependen demasiado de la funcionalidad heredada. Esto es señal de que algo es incorrecto, ya sea con la interfaz que implementa la clase hijo o con la clasificación de la relación entre clases.
--	---

Tabla 2.1: Code Smells catalogados por Lanza y Marinescu.

Como se mencionó anteriormente, orientar la refactorización en la eliminación de code smells resulta beneficioso, por lo que es necesario realizar la previa detección de los mismos. Generalmente, la detección de code smells se realiza mediante la utilización de diversas métricas [1].

Las métricas se pueden usar en la práctica para ayudar en la caracterización de software, para evaluar su diseño y proporcionar las refactorizaciones adecuadas cuando hay problemas de diseño, code smells.

2.6 Métricas

Las métricas de software ayudan a caracterizar, evaluar y mejorar el diseño de sistemas [50]. Sin embargo, una métrica por sí sola no se considera suficiente para comprender y evaluar un sistema de software. Por lo tanto, es necesario que se entienda y se tenga en claro la correlación entre las diferentes métricas.

Para lograr que se trabaje de manera adecuada con las métricas, es importante interpretar sus valores correctamente. Para ello, se deben establecer umbrales para cada una de las métricas que se utilizan. El umbral es un valor con el que se determina el límite en el cual la métrica es válida, en caso contrario, se determina que la métrica posee un valor fuera del rango que se definió como normal o válido [1].

Para que las métricas sean útiles, si lo que se pretende es construir un sistema de software de alta calidad, se debe considerar la utilización de una estrategia de detección. Una estrategia de detección es una condición lógica compuesta, basada en métricas, por el cual se detectan fragmentos de diseño con propiedades específicas en el código fuente [1].

Lanza y Marinescu [1], proponen un catálogo de métricas en el lenguaje Java. Este catálogo es muy importante como guía para los desarrolladores en la búsqueda de problemas en el código de los sistemas que implementan, y por lo tanto muy utilizado en el entorno del desarrollo de sistemas de software. El catálogo de métricas es útil para mantener un orden y una estrategia clara para la detección de code smells en el lenguaje que se utilice.

En la Tabla 2.2 se detallan las métricas propuestas por Lanza y Marinescu.

Métrica	Descripción
AMWA - Average Method Weight	Complejidad media estática de todos los métodos de una clase.
ATFD (Access To Foreign Data)	Número de atributos de clases relacionadas a los que se accede directamente o mediante la invocación de métodos de acceso.
BOvR (Base Class Overriding Ratio)	Número de métodos de la clase que especializan a los métodos de la clase base, dividido el número total de métodos de la clase.
BUR (Base Class Usage Ratio)	Número de miembros de la herencia utilizados por la clase, dividido por el número total de miembros que se heredan.
CC (Changing Classes)	Número de métodos que son llamados por otras clases para definir sus métodos.
CDISP (Coupling Dispersion)	Número de clases en las que se definen los métodos de la clase, dividido la métrica CINT.
CINT (Coupling Intensity)	Número de operaciones distintas llamadas por la operación medida. Invocación de métodos de otras clases, a nivel de clase.
CM (Changing Methods)	Número de métodos distintos que llaman el método medido.
CYCLO (McCabe's Cyclomatic Number)	Número de trayectorias linealmente independientes a través de una operación.
FDP (Foreign Data Providers)	Número de clases en las que se definen los atributos a los que se acceden.
LAA (Locality of Attribute Accesses)	Número de atributos de la clase de definición del método, dividido por el número total de variables accedidas.
LOC (Lines of Code)	Número de líneas de código de una operación, incluyendo líneas en blanco y comentarios
MAXNESTING (Maximum Nesting Level)	Máximo nivel de anidamiento de las estructuras de control dentro de una operación.
NAS (Number of Added Services)	Número de métodos públicos de una clase que no son especializaciones de la clase heredada.
NOAM (Number of Accessor Methods)	Número de métodos de acceso (get's/set's) de una clase.
NOAV (Number of Accessed Variables)	Número de variables a las que se accede directamente desde la operación medida.
NOM (Number of Methods)	Número de métodos de una clase.
NOPA (Number of Public Attributes)	Número de atributos públicos de una clase.
NProtM (Number of Protected Members)	Número de atributos y métodos protected de la clase.
PNAS (Percentage of Newly Added Services)	Número de métodos de una clase que no son especializaciones de la clase heredada, dividido el número total de métodos.
TCC (Tight Class Cohesión)	Número relativo de pares de métodos de una clase que acceden al menos a un atributo en común de la clase.
WMC (Weighted Method Count)	La suma de la complejidad estática de todos los métodos de una clase (métrica CYCLO).

WOC (Weight Of a Class)	Número de métodos públicos "funcionales" dividido por el número total de atributos públicos de una clase.
-------------------------	---

Tabla 2.2: Métricas catalogadas por Lanza y Marinescu.

Como se mencionó anteriormente, tener un catálogo de métricas definido para el lenguaje de desarrollo que se utiliza, es útil para mantener un orden y orientar la búsqueda de code smells en el mismo a través de estrategias de detección definidas a partir de las métricas descritas. Actualmente, no se ha podido hallar la definición de un catálogo de métricas para el lenguaje JavaScript, siendo que sería de mucha utilidad y muy importante, debido al gran crecimiento del uso del lenguaje en el desarrollo de sistemas de software.

2.7 JavaScript

JavaScript es un lenguaje de programación de propósito general, y su uso no está restringido a los navegadores web. Este lenguaje de programación, fue diseñado para ser incorporado, y proporcionar capacidades de scripting a cualquier aplicación [51]. Su adopción global se produjo en un período de tiempo alarmantemente corto [52], y según la encuesta que realiza el sitio web Stack Overflow, que es el más utilizado por la comunidad de desarrolladores para encontrar soluciones a problemas de programación en diferentes lenguajes, el lenguaje JavaScript es el más utilizado en el mundo [11].

Entre las características más destacables del lenguaje se encuentran las siguientes [53] [54]:

- Es un lenguaje de programación orientado a objetos basado en la definición de prototipos.
- Es un lenguaje dinámico, en el sentido de que las variables en JavaScript no necesitan ser definidas antes de su uso. Los tipos de variables se resuelven dinámicamente durante la ejecución.
- Permite que las definiciones de funciones se modifiquen mientras se ejecuta el programa.
- El modelo de ejecución de JavaScript se basa en la interpretación del código fuente.

Con el objetivo de estandarizar JavaScript, en el año de 1997 se crea el comité TC39 en la ECMA (European Computer Manufacturers Association). A partir de ese momento, los estándares de JavaScript se rigen como ECMAScript [4], y se basan en la especificación que realiza dicho estándar del lenguaje. ECMAScript define un lenguaje de tipos dinámicos inspirado en Java y otros lenguajes del estilo de C, y soporta algunas características de la programación orientada a objetos mediante objetos basados en prototipos y pseudoclases.

Para la ejecución de los sistemas desarrollados en el lenguaje JavaScript, se utiliza el entorno de ejecución Node.js. Dicho entorno se encuentra basado en ECMAScript, y es el más popular y utilizado por los desarrolladores. De acuerdo a la información que brinda su página web, Node.js, se define de la siguiente manera [56]:

- Node.js es una plataforma construida encima del entorno de ejecución JavaScript de Chrome para fácilmente construir rápidas y escalables aplicaciones de red. Node.js usa un modelo de E/S no bloqueante dirigido por eventos que lo hace ligero y eficiente, perfecto para aplicaciones data-intensive en tiempo real.

Este estándar o familia de estándares define el entorno y las interfaces, así como utilidades comunes, que un sistema operativo debe soportar y hacer disponibles para que el código fuente de un programa sea portable, compilable y ejecutable, en diferentes sistemas operativos que implementan dicho estándar.

Entre las características que se mencionaron para el correcto funcionamiento de los sistemas desarrollados en JavaScript, falta mencionar un concepto muy importante para que el lenguaje conserve las características de encapsulación y dependencia que tanto se destacan en la programación orientada a objetos.

En JavaScript, siempre ha sido tarea del desarrollador asegurar que las dependencias se satisfagan al momento de ejecutar cada bloque de código, y que las mismas se carguen en el orden correcto. A medida que se escribe más código JavaScript en diferentes aplicaciones, la gestión de dependencias resulta más engorrosa. CommonJS [17] es una creciente colección de estándares que surgen de la necesidad de completar aspectos que no se consideran en la especificación de JavaScript. Entre estos aspectos se consideran la no utilización de JavaScript del lado del servidor o en aplicaciones de escritorio y, el de mayor incumbencia, la falta de un sistema de módulos [55]. Un módulo JavaScript es un código que de alguna manera es “auto contenido” y que expone una interfaz pública para ser usada. CommonJS se basa en el sistema de módulos síncrono, es decir, la carga de módulos se realiza por medio de un proceso que comienza por un módulo inicial. Al cargarse el módulo inicial, se cargaran todas sus dependencias y las dependencias de las dependencias hasta el máximo nivel de profundidad. Una vez que finalice la carga de todas sus dependencias, el módulo inicial estará cargado y comenzará a ejecutarse. Node.js incorpora el concepto de módulos siguiendo la especificación de CommonJS.

Existen dos conceptos esenciales para interactuar con los módulos: `require` y `exports`.

- **require** es una función que se puede usar para importar propiedades o funciones desde otro archivo js al ámbito actual. El parámetro pasado a `require` es el id del módulo.

- **exports** es un objeto especial, todo lo que es puesto en él se puede exportar como un elemento público (conservando el nombre de los elementos).

En JavaScript no se definen métodos, sino que se define un módulo u objeto como una función. Tampoco se definen clases como tales, por lo cual, algunos autores definen una función como una clase, mientras que otros definen una clase como un archivo js que contiene funciones. En este trabajo, se adopta la segunda definición en la cual se define un archivo js como si fuera una clase y las funciones que contiene el archivo js como métodos y/o definición de objetos. Por lo tanto, ante la adopción de esta notación, se usan los términos `require` y `exports` para utilizar funciones y propiedades de un archivo js en otro archivo js. Para que una variable o función se pueda acceder desde otro archivo js, se debe definir el `exports` de la variable o función en el archivo js que la contiene; y desde el archivo js que desea utilizarla, se debe realizar el `require` del archivo js que la contiene. De esta manera se define la “herencia simple” en JavaScript, sólo es de uso y de un único nivel.

2.7.1 Variables

Las variables en JavaScript se crean mediante la palabra reservada `var`, y se utilizan para almacenar y/o hacer referencias a otro valor. Dicha palabra reservada, se debe indicar al momento de declarar la variable. En JavaScript no es obligatorio inicializar las variables, se puede asignar un valor al declararla o posteriormente. Para utilizar las variables en el resto de instrucciones del script, sólo se debe indicar el nombre de la variable que se quiere utilizar. Sin embargo, también es posible utilizar variables sin haberlas declarado. Es decir, se pueden utilizar variables que no se hayan definido anteriormente con la palabra reservada `var`. Ante esta situación, se dice que JavaScript crea una variable global y le asigna el valor correspondiente.

El ámbito de una variable (llamado "scope" en inglés) es la zona del programa en la que se define la variable. JavaScript define dos ámbitos para las variables: global y local [57]. Si una variable se declara dentro de una función con la palabra reservada `var`, esta será local a la misma, y únicamente se podrá usar desde cualquier instrucción dentro de ella. Por el contrario, si se declara la variable dentro de la función sin la palabra reservada `var`, la misma será global al archivo js que contiene la función. A diferencia de las variables locales, las variables globales se declaran en cualquier parte del archivo js y por lo tanto puede ser utilizada desde cualquier parte incluso dentro de una función que contenga el archivo. Por lo tanto, si una variable se declara fuera de cualquier función, es una variable global y si se declara dentro de una función puede ser local o global dependiendo de si se declara o no con la palabra reservada `var`.

Además de definir variables mediante el uso de la palabra reservada `var`, se pueden definir variables mediante el uso de la palabra reservada `this`. Cuando se declara una variable de esta manera, no sólo se define que será global sino que también adquiere la cualidad de que podrá

ser utilizada desde otro archivo js. Se debe aclarar que si una variable se define con la palabra reservada `this` dentro de una función, la misma podrá ser accedida desde otro archivo js únicamente si se realiza el `export` de la función que la contiene.

```
var check = require('check-types'), report;

exports.analyse = analyse;

function analyse (ast, walker, options) {
  // TODO: Asynchronise

  var settings, currentReport, clearDependencies = true, scopeStack = [];

  check.assert.object(ast, 'Invalid syntax tree');
  check.assert.object(walker, 'Invalid walker');
  check.assert.function(walker.walk, 'Invalid walker.walk method');

  if (check.object(options)) {
    settings = options;
  } else {
    settings = getDefaultSettings();
  }

  // TODO: loc is moz-specific, move to walker?
  report = createReport(ast.loc);

  walker.walk(ast, settings, {
    processNode: processNode,
    createScope: createScope,
    popScope: popScope
  });

  calculateMetrics(settings);

  return report;

  ...
}
```

Figura 2.2: Código ejemplo para conceptos básicos de JavaScript.

Por ejemplo, en la Figura 2.2, se puede observar la variable `check` que se encuentra definida con la palabra reservada `var` y fuera de la función `analyse`. Dicha variable, es local al archivo js que la contiene y puede ser utilizada por cualquiera de las funciones que se definan dentro

del mismo. Otra característica de la variable `check` es que se encuentra inicializada, no así la variable `report`, que también se define local al archivo `js`, pero no se encuentra inicializada.

Otro ejemplo es la variable `settings`, que se define en la función `analyse` y por lo tanto es local a dicha función. Esta variable, se podrá utilizar únicamente en las instrucciones que se encuentran dentro de la función `analyse`. Si la variable `settings` se hubiera declarado sin la palabra reservada `var`, se definiría como una variable de ámbito global y podría ser utilizada por cualquier instrucción dentro del archivo `js` que contiene la función `analyse`. Otra manera de declarar esta variable, sería la de definirla utilizando la palabra reservada `this` en lugar de `var`. Si se hubiera declarado con la palabra reservada `this`, y como se realiza un `export` de la función, la variable `settings` hubiera podido ser utilizada desde cualquier archivo `js` que realice el `require` de la misma.

2.7.2 Funciones

Las funciones [57] en JavaScript se definen mediante la palabra reservada `function` delante del nombre de la función y son utilizadas, como en todos los lenguajes de programación donde se definen funciones, para contener y organizar información que se consulta o utiliza dentro de un programa generalmente repetitivas veces. Para contener las instrucciones de la función y definir su estructura, se utilizan las llaves, se abre la primer llave en el comienzo de la función y al final se utiliza la llave de cierre.

Para utilizar una función simplemente se llama por su nombre.

Las funciones en JavaScript pueden recibir parámetros, lo que difiere del resto de los lenguajes es la manera en la que se pueden definir y utilizar los mismos, como por ejemplo:

- Generalmente, en la mayoría de los lenguajes de programación, el número de argumentos que se pasa a una función debería ser el mismo que el número de argumentos que ha indicado la función. Sin embargo, en JavaScript no se produce ningún error si se pasan más o menos cantidad de argumentos de los indicados al declarar la función.
- El número de argumentos puede ser ilimitado, aunque si la cantidad es muy grande puede traer problemas de comprensión y complicar la llamada a la función.
- El orden de los argumentos es fundamental, ya que el primer dato que se indica en la llamada, será el primer valor que espera la función; el segundo valor indicado en la llamada, es el segundo valor que espera la función y así sucesivamente.

Como ya se mencionó anteriormente en JavaScript no existen los métodos, por lo tanto todo se define como una función que puede o no retornar un valor. Otra característica a mencionar,

muy importante y utilizada en este trabajo, es que para que se pueda utilizar una función desde otro archivo js la misma debe estar dentro del módulo que se exporta del archivo js que la contiene, o directamente se debe exportar la función. De esta manera se podrán acceder a las propiedades que se definen en la función si se definieron con la palabra reservada `this`, o al retorno de la misma.

En el código de ejemplo que se muestra en la Figura 2.2, se define la función `analyse`. Dicha función se encuentra definida mediante la palabra reservada `function` y su comienzo y final se encuentra delimitado por llaves. La función tiene tres parámetros: `ast`, `walker` y `options`. Como se mencionó anteriormente, la función puede recibir todos sus parámetros o parte de ellos, lo que hay que considerar es el orden. Por ejemplo, si a la función `analyse` se le indica un único parámetro, el mismo se corresponderá con el primer parámetro denominado `ast`. En este caso en particular la función retorna el valor `report`.

Otra consideración a destacar de la función `analyse` que se muestra en la Figura 2.2, es que se encuentra definido el `exports` de la misma. Por lo tanto, se podrá hacer uso de la funcionalidad de la función desde otro archivo js que realice su `require`. Sin embargo, hay que aclarar que las variables que se definen en la función no se podrán acceder desde otro archivo js que realice el `require`, ya que se encuentran definidas mediante la palabra reservada `var`.

Capítulo 3

Trabajos relacionados

El código fuente de un sistema puede cambiar a través del tiempo como consecuencia de cambios en los requerimientos que dan lugar a nuevas funcionalidades. La tarea de mantenimiento es fundamental ante los cambios en el entorno del sistema, y cuanto más mantenible sea el sistema que se implementa más fácil será esta tarea.

Para que el código sea entendible, el mantenimiento del mismo no sea complejo y costoso; y pueda evolucionar en el tiempo, es deseable que se encuentre libre de code smells. Las métricas son un instrumento muy importante para la detección de smells en el código. A partir de las mismas, se puede evaluar el cumplimiento de condiciones que llevan a la detección de un code smell.

Lanza y Marinescu [1], proponen un conjunto de once code smells junto a un catálogo de métricas, para detectar dichos smells en el código de un sistema de software desarrollado en lenguaje Java, y evitar problemas relacionados a la calidad y mantenibilidad del mismo.

En este capítulo, se presentan diferentes trabajos en los que se realiza el análisis de problemas en el código del sistema que afectan la mantenibilidad del mismo, haciendo énfasis en trabajos par JavaScript. Los trabajos presentados, pretenden mejorar la calidad de los sistemas principalmente mediante el análisis de técnicas de detección de smells a partir del uso de métricas u otros instrumentos que faciliten su mantenimiento.

Entre los puntos de análisis a destacar de los trabajos que se presentan, se encuentran:

- Análisis de la complejidad cognitiva del código JavaScript a partir de métricas.
- Detección de code smells utilizando métricas, y la técnica JNOSE.
- Ajuste de métricas para mejorar la detección de smells automáticamente.
- Eliminación de smells utilizando programación orientada a objetos.

Además, se presentan herramientas que realizan la detección de code smells, pero sin realizar un análisis previo.

El objetivo de este capítulo es destacar trabajos relacionados al análisis de code smells mencionando las características principales y destacando los aspectos importantes. En las primeras secciones se presentan diferentes análisis y/o técnicas de detección de smells; luego se presentan herramientas de detección de smells y, finalmente, se realiza un resumen del capítulo.

3.1 Estimación de calidad de JavaScript

Misra y Cafer [25], proponen una métrica de complejidad para evaluar los scripts del lenguaje JavaScript. La métrica propuesta es “JavaScript Cognitive Complexity Measure (JCCM, Medida de Complejidad Cognitiva de JavaScript)”, y tiene como objetivo evaluar la calidad de diseño de los scripts.

Su enfoque yace en el concepto de complejidad cognitiva, y en que, en la mayoría de las métricas existentes para evaluar código en JavaScript, no consideran las características cognitivas al calcular la complejidad del código. La complejidad cognitiva, se define como la capacidad, o el agobio mental, de las personas para comprender el código. Dichas personas pueden ser desarrolladores, testers, entre otros roles que tratan directamente con el código del sistema que se evalúa.

Para realizar el análisis de complejidad del código JavaScript, identifican cinco factores (métricas):

1. Tamaño en término de líneas de código: La complejidad de algunos programas dependen de la cantidad de líneas de código. Se incluyen, además, en el análisis de las líneas de código, todos los posibles factores que se supone que incrementan la complejidad de las mismas.
2. Número de variables distintas nombradas arbitrariamente (ANDV): Cuando se habla de la capacidad de comprensión del código, los nombres de las variables que se utilizan tienen un papel importante en la misma. Si el nombre de la variable representa de manera clara su significado, se vuelve más fácil entender el código.
3. Número de variables con un nombre significativo (MNV): Las variables con nombre significativo son más claras. En este punto se asigna el peso de variables significativas como una unidad.
4. Pesos cognitivos de estructuras de control básicas (BCS's): La complejidad de un sistema es directamente proporcional al peso cognitivo de las estructuras de control básicas. El peso cognitivo del software es la dificultad, o el tiempo y esfuerzo relativos para comprender el software modelado por medio del uso de estructuras de control básicas [37].

Las estructuras de control básicas son lógicas básicas que construyen bloques, a las cuales se les asigna un peso de acuerdo a la clasificación de fenómenos cognitivos propuesta por Wang [37]. Aunque este trabajo sigue un enfoque similar al propuesto por Wang [37], se presentan modificaciones. Por ejemplo, se incluye la estructura de control básica try-catch y se le asigna el peso 2, en función de su estructura. Las estructuras básicas y sus ponderaciones, se muestran en la Tabla 3.1. En dicha tabla,

además, se manifiesta que estructuras como lo son la secuencia, condición y bucles en JavaScript, tienen estructuras similares con otros lenguajes de programación.







Category	BCS	CWJ	Flow Graph
Sequence	sequence	1	
Condition	if-else	2	
	switch	2	
	go-to	2	
Loop	for	3	
	for...in	3	
	while/do...while	3	
	nested loop	3 ⁿ	
Functional Activity	function-call	2	
	alert/prompt/throw	2	
	event	2	
	recursion	3	
Exception	try...catch	2	

Tabla 3.1: Estructuras de control básicas y pesos.

- Número de operadores: Involucra a todo tipo de operadores que se relacionan directamente con las variables que se definen en el sistema. El número de operadores también aumenta la complejidad del código JavaScript. Son responsables del incremento del tamaño.

Para evaluar la complejidad cognitiva del código se evalúan cada una de las líneas que conforman el mismo. El ejemplo que se muestra en la Tabla 3.2, presenta 6 líneas de código, las cuáles son evaluadas una por una para obtener el valor de las métricas que representan la complejidad cognitiva.

Line No.	JavaScript Code	ANDV+MNV+ Operators	CWJ	JCCM
1	var i=0;	8	1	8
2	for (i=0;i<=5;i++)	10	3	30
3	{	0	1	0
4	document.write("The number is " + i);	3	1	3
5	document.write(" ");	1	1	1
6	}	0	1	0
	Total	-	-	42

Tabla 3.2: Código de ejemplo para el cálculo de la complejidad cognitiva (JCCM) en JavaScript.

Program No.	JCCM	ILOC	CC	Halstead			
				V	D	E	T
1	42	4	3	79	6	474	26
2	132	5	5	93	9	837	46
3	43	8	4	148	4	592	32

Tabla 3.3: Comparación de métricas de complejidad.

Una demostración destacable del análisis que se presenta en este trabajo, es que al comparar los valores de complejidad de todas las métricas de complejidad relacionadas, como la complejidad ciclomática, los valores de la JCCM son más altos (Tabla 3.3). Esto se debe a que la métrica JCCM representa valores de complejidad relacionados a todos los parámetros responsables de la complejidad. En este trabajo se define que, ninguna de las métricas más utilizadas son tan eficientes para evaluar la calidad de JavaScript, como JCCM.

A pesar del análisis que realiza de la complejidad del código JavaScript, y de la relación con otras métricas, en este trabajo, sólo se ven involucradas una cantidad menor de métricas que se evalúan para obtener un valor final de complejidad que no representa un problema puntual en el código como lo es un code smell.

Además, se debe destacar que el trabajo se centra en la complejidad de estructuras básicas de control, dejando de lado problemas que analizan otras características importantes de los sistemas como lo son Intensive Coupling o Dispersed Coupling, entre otros.

3.2 JSNOSE: detección de Code Smells en JavaScript

Fard y Ali [34], proponen una lista de code smells para aplicaciones basadas en JavaScript, y presentan la técnica JSNOSE para detectarlos. Dicha técnica, utiliza un algoritmo basado en métricas, y combina el análisis estático con el dinámico, para detectar smells en el código JavaScript. El enfoque que se define en este trabajo, que se encuentra implementado por la técnica JSNOSE, puede ser utilizado por los desarrolladores durante los ciclos de desarrollo y mantenimiento para detectar posibles code smells en aplicaciones basadas en JavaScript.

El trabajo expuesto propone las siguientes contribuciones:

- Propone una lista de code smells en JavaScript recopilados de diversos recursos de desarrollo web;
- Presenta un enfoque automatizado basado en métricas para detectar code smells en JavaScript;
- Implementa el enfoque en una herramienta llamada JSNOSE;
- Evalúa la efectividad de la técnica;

- Se evalúan 11 aplicaciones web utilizando JSNOSE para descubrir qué smells son más prevalentes.

La Tabla 3.4 presenta las métricas y los criterios que se utilizan en el enfoque de este trabajo para detectar code smells en aplicaciones JavaScript. Algunas de estas métricas y sus correspondientes umbrales se han propuesto y utilizado para detectar code smells en lenguajes orientados a objetos [27], [1], [28], [29], [30], y otras, son métricas propuestas en este trabajo para capturar otros code smells en relación a las características del lenguaje JavaScript que difieren de los lenguajes orientados a objetos.

Code smell	Level	Detection method	Detection criteria	Metric
Closure smell	Function	Static & Dynamic	$LSC > 3$	LSC: Length of scope chain
Coupling JS/HTML/CSS	File	Static & Dynamic	$JSC > 1$	JSC: JavaScript coupling instance
Empty catch	Code block	Static	$LOC(catchBlock) = 0$	LOC: Lines of code
Excessive global variables	Code block	Static & Dynamic	$GLB > 10$	GLB: Number of global variables
Large object	Object	Static & Dynamic	[30]: $LOC(obj) > 750$ or $NOP > 20$	NOP: Number of properties
Lazy object	Object	Static & Dynamic	$NOP < 3$	NOP: Number of properties
Long message chain	Code block	Static	$LMC > 3$	LMC: Length of message chain
Long method/function	Function	Static & Dynamic	[13], [30]: $MLOC > 50$	MLOC: Method lines of code
Long parameter list	Function	Static & Dynamic	[30]: $PAR > 5$	PAR: Number of parameters
Nested callback	Function	Static & Dynamic	$CBD > 3$	CBD: Callback depth
Refused bequest	Object	Static & Dynamic	[13]: $BUR < \frac{1}{3}$ and $NOP > 2$	BUR: Base-object usage ratio
Switch statement	Code block	Static	$NOC > 3$	NOC: Number of cases
Unused/dead code	Code block	Static & Dynamic	$EXEC = 0$ or $RCH = 0$	EXEC: Execution count RCH: Reachability of code

Tabla 3.4: Code Smells que se detectan en el análisis que utiliza la herramienta JSNOSE.

En la Figura 3.1 se muestra el método de detección de code smells del presente trabajo. El algoritmo es genérico en el sentido de que los procedimientos de detección, de smells estáticos y dinámicos, basados en métricas se pueden definir y usar para cualquier criterio de detección de smells. Dada una aplicación de JavaScript A, un tiempo t de rastreo máximo y un conjunto de criterios τ de code smell, el algoritmo genera un conjunto de code smells CS.

La evaluación que se realizó en este trabajo indica que lazy object, long method/function, closure smell, el acoplamiento entre JavaScript, HTML y CSS, y las variables globales excesivas son los code smells más frecuentes. Además, dicha evaluación, indica que existe una significativa correlación entre los tipos de smells y LOC, la complejidad ciclomática y el número de funciones y archivos de JavaScript.

El trabajo realiza un análisis interesante de la detección de code smells en JavaScript en relación a la lista de smells que propone y a la herramienta que logra la detección de los mismos, sin embargo, se debe destacar que tanto el análisis como la herramienta están orientados a la programación web en el lenguaje JavaScript.

Además, considerando los code smells que proponen Lanza y Marinescu [1], se puede deducir, a partir de los datos expuestos en la Tabla 3.4, que sólo uno es considerado en este trabajo y es el code smell Refused Bequest. Para dicho code smell en este trabajo, se utilizan dos métricas BUR y NOP, coincidiendo únicamente en la métrica BUR con el análisis realizado por Lanza y Marinescu. Las demás métricas que se utilizan en la estrategia de

detección del code smell en [1], no se definen en este trabajo, como por ejemplo AMW, WMC y NOM.

```

input : A JavaScript application  $A$ , the maximum exploration time
         $t$ , the set of smell metric criteria  $\tau$ 
output: The list of JavaScript code smells  $CS$ 

1  $CS \leftarrow \emptyset$ 
  Procedure EXPLORE() begin
2   while TIMELEFT( $t$ ) do
3      $CS \leftarrow CS \cup \text{DETECTINLINEJSINHTML}(\tau)$ 
4      $code \leftarrow \text{EXTRACTJAVASCRIPT}(A)$ 
5      $AST \leftarrow \text{PARSTOAST}(code)$ 
6      $\text{VISITNODE}(AST.root)$ 
7      $ASTinst \leftarrow \text{INSTRUMENT}(AST)$ 
8      $\text{INJECTJAVASCRIPTCODE}(A, ASTinst)$ 
9      $C \leftarrow \text{EXTRACTCLICKABLES}(A)$ 
10    for  $c \in C$  do
11       $dom \leftarrow \text{browser.GETDOM}()$ 
12       $\text{robot.FIREEVENT}(c)$ 
13       $new\_dom \leftarrow \text{browser.GETDOM}()$ 
14       $CS \leftarrow CS \cup \text{DETECTDYNAMICALLY}(\tau)$ 
15      if  $dom.HASCHANGED(new\_dom)$  then
16         $\text{EXPLORE}(A)$ 
17     $CS \leftarrow CS \cup \text{DETECTUNUSEDCODE}()$ 
18  return  $CS$ 

  Procedure VISITNODE( $ASTNode$ ) begin
19     $CS \leftarrow CS \cup \text{DETECTSTATICALLY}(node, \tau)$ 
20    for  $node \in ASTNode.getChildren()$  do
21       $\text{VISITNODE}(node)$ 

```

Figura 3.1: Algoritmo de detección de code smells de la herramienta JSNOSE en JavaScript.

3.3 Uso de ejemplos para detectar Code Smells en JavaScript

Este trabajo hace hincapié en las limitaciones de JSNOSE, en el cual las reglas que se usan para detectar code smells se definen manualmente, y puede ocurrir que la cantidad de posibles code smells para caracterizar manualmente con dichas reglas sea muy grande, lo que sería un problema para el usuario.

Shoenberger et al. [26], proponen realizar un ajuste automático de las reglas de detección de code smells en JavaScript para evitar la intervención humana, utilizando un conjunto de datos de code smells existentes. El trabajo se centra en hallar un subconjunto de objetos similares en una base de ejemplos, y luego usar Programación Genética para ajustar el umbral de métricas cuantitativas para maximizar la cobertura de code smells detectados en el subconjunto. Las métricas que se obtienen, se usarán posteriormente para detectar code smells en JavaScript.

La principal contribución del trabajo, se basa en la siguiente pregunta:

- ¿Cómo considerar el conocimiento existente en la detección de malas prácticas de programación en lenguajes orientados a objetos, en la detección de prácticas similares en JavaScript?

El flujo de trabajo general de este enfoque se compone de cuatro etapas principales (Figura 3.2):

1. Enumeración de entidades JavaScript: Se extraen de la aplicación y se analizan los archivos js con el enfoque de JSNOSE. El conjunto de entidades JS extraídas se envía a la calculadora de similitud.
2. Obtención de elementos similares: Se extraen elementos de código que son en su mayoría similares en términos de propiedades estructurales. Para ello se aplica una técnica de coincidencia inicial que evite la comparación exhaustiva.
3. Calibración de umbrales de métricas: El objetivo de este paso es actualizar los umbrales de las reglas de detección de JavaScript. Para hacerlo, primero se extraen los smells y se genera una regla de detección para cada uno. Luego, para cada regla que se genera, se someten al proceso de calibración las métricas utilizadas por JSNOSE. El proceso de ajuste, tiene como objetivo adaptar los umbrales de las métricas para maximizar el número de code smells detectados anteriormente. De esta manera, la regla adaptada se puede usar en una fase posterior para detectar code smells en JavaScript.
4. Detección de Code Smells: Finalmente, se retorna una regla de detección actualizada para cualquier tipo de smell conocido en el subconjunto de elementos del código. Esta regla se ejecuta para determinar si las entidades JavaScript, y sus propiedades, violan los umbrales de métricas actualizados.

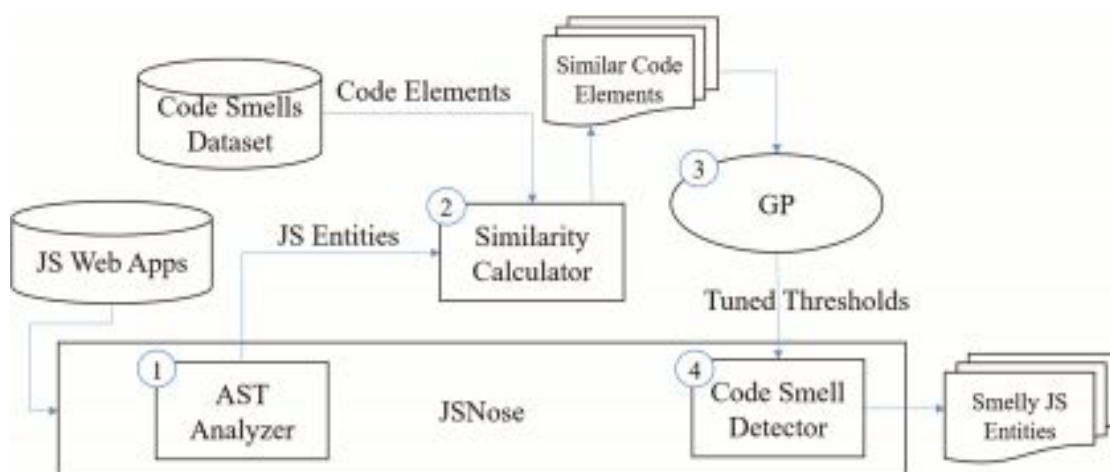


Figura 3.2: Flujo general de [26].

El trabajo presenta una detección novedosa de code smells en JavaScript utilizando ejemplos de code smells que se extraen de proyectos existentes en C++ y Java. La evaluación de este trabajo ha obtenido buenos resultados. Sin embargo, se debe destacar que los code smells que se consideran para la evaluación son limitados y, por lo tanto, a futuro va a ser necesario ampliar la base de métricas usadas para definir las reglas de detección. Por ejemplo, no se consideran métricas relacionadas al acoplamiento, la cohesión y la complejidad.

3.4 Eliminación de Code Smells en JavaScript con Programación Orientada a Objetos (OOP)

Wintz [31], propone definir una serie de code smells para ser identificados en el código y, a partir de los mismos, construir técnicas basadas en programación orientada a objetos para su refactorización. Wintz, aclara que no quiere quitar el dinamismo de JavaScript, sino aplicar una técnica suficiente para mantener el código para que sea entendible y mantenible.

En su análisis define tres code smells:

1. Conditional/Switch Statement Smell: Gran cantidad de condicionales que pueden manifestarse como estructuras if/else o como instrucciones switch.
2. Dictionary Object: Por ejemplo, se recupera un diccionario de una solicitud HTTP, y se comienza a replicar ese diccionario donde sea necesario.
3. Inappropriate Intimacy: Este code smell ocurre cuando un objeto se relaciona demasiado con otro objeto. Esta relación, provoca que sea muy difícil cambiar la estructura del objeto al que se accede. Como todo es un diccionario y no existe una noción nativa de miembros privados en Javascript, es una tendencia natural obtener las propiedades que necesita en lugar de solicitar el objeto utilizando un método de acceso.

Para el primer code smell que se menciona, se toma como ejemplo el código que se muestra en la Figura 3.3 que, a simple vista, parece inofensivo. En la mayoría de los casos, sin embargo, se puede ver que esta misma declaración de cambio aparece en muchos lugares, y que se vuelve problemática cuando comienzan a extenderse sobre diferentes archivos. Para este ejemplo particular, se muestra una condición representada por un texto y una lógica de validación de longitud en ambos casos. Si se quieren agregar nuevos componentes y/o validaciones se deberán actualizar ambas, o todas las que haya en el caso que se agreguen más condicionales.

Ante el ejemplo de código que se muestra en la Figura 3.3, se presentan dos alternativas, siendo la mejor la segunda alternativa (Figura 3.4). En donde se implementa utilizando una

clase abstracta más tradicional, creando subclases de los diferentes tipos. En JavaScript, dicho esquema se puede implementar usando herencia de prototipos.

```
post.prototype.verifyPostLength () {
  switch (this.type) {
    case 'twitter':
      return this.text.length <140;
      descanso;
    case 'facebook':
      return this.text.length <63206;
      descanso;
    // etc ...
  }
}
```

Figura 3.3: Ejemplo de código JavaScript a refactorizar.

```
// Función del archivo
Errors.js NotImplementedError () {
  this.name = "NotImplementedError";
  this.message = "Método abstracto no implementado por la subclase";
}

NotImplementedError.prototype = Error.prototype;

// Función de archivo post / post.js
Post () {}

Post.prototype.verifyLength = function () {
  throw new NotImplementedError ();
}

//
función de archivo post / TwitterPost.js TwitterPost () {}

TwitterPost.prototype = Object.create (Post.prototype);
TwitterPost.prototype.constructor = TwitterPost;

TwitterPost.prototype.verifyLength = function () {
  return this.text.length <48;
}
```

Figura 3.4: Ejemplo de código JavaScript refactorizado.

De esta manera, tiene un patrón reconocible para describir los métodos que se deben implementar en todas las funciones que representan subclases, y tiene una forma fácil de encontrar las implementaciones específicas del tipo. En objetos complejos con muchos comportamientos diferentes y posiblemente únicos, los beneficios de implementar este patrón se vuelven claros. Estas subclases son mucho más fáciles de administrar ante muchas instrucciones de cambio, o un objeto muy grande de "mapeo".

Para los code smells restantes se muestran ejemplos del mismo tipo, refactorizando el código de ejemplo que representa el problema.

Este trabajo, tiene como objetivo no dejar la refactorización para etapas posteriores al desarrollo. Lo que se propone es una guía de code smells que representan problemas comunes para que el desarrollador pueda reconocerlos fácilmente y evitarlos antes de que impidan que el sistema sea entendible y/o mantenible. Si bien la propuesta es válida, y ayuda a que el desarrollador tenga buenos hábitos de programación, más aún en un lenguaje dinámico como lo es JavaScript, en estos tiempos en el que la demanda de sistemas es muy exigente en cuanto a cantidad y tiempo de desarrollo, es difícil que el desarrollador adquiera este tipo de hábitos. Además, el desarrollador debe hacerlo manualmente sin el aporte de una herramienta. Otro problema que se presenta en este trabajo, es que los code smells que se definen son demasiado escasos.

Se considera además, luego del análisis realizado del trabajo que, de los tres code smells que se presentan, ninguno coincide con los code smells propuestos por Lanza y Marinescu [1].

3.5 Herramientas

Existen herramientas mediante las cuales se obtienen valores de métricas de software a partir de la evaluación de código JavaScript. Sin embargo, dichas herramientas no realizan el análisis de las métricas para la obtención de code smells, si no que, simplemente obtienen el valor de las mismas. La mayoría de las herramientas no definen estrategias de detección de code smells, ni métricas en relación a las mismas, sólo detallan el valor de una lista específica de métricas.

Este tipo de herramientas, se pueden comparar con la herramienta vcomplex que se utiliza como parte de este trabajo para obtener los valores de las métricas que se validaron al analizar las estrategias de detección de los code smells presentes en el Capítulo 4.

En esta sección se detallan dos herramientas del tipo de herramientas mencionadas.

3.5.1 escomplex

El objetivo de esta herramienta es realizar un análisis de complejidad de software en JavaScript [32]. Para realizar el análisis de complejidad, escomplex, recorre árboles de sintaxis que se obtienen mediante el uso de Esprima [18], que es un analizador de código JavaScript muy popular.

Esta herramienta fue creada en el 2012 por Phil Booth, que implementó las métricas de código fuente para el lenguaje JavaScript en el paquete npm llamado escomplex.

Actualmente la herramienta informa sobre:

- Líneas de código: tanto físicas (el número de líneas en un módulo o función) como lógicas (un recuento de las declaraciones imperativas).
- Número de parámetros: analizados estáticamente desde la firma de la función.
- Complejidad ciclomática: definida por Thomas J. McCabe en 1976, cuenta el número de ciclos en el gráfico de control de flujo del programa. Efectivamente, la cantidad de caminos distintos a través de un bloque de código.
- Densidad de complejidad ciclomática: propuesta como una modificación de la complejidad ciclomática por Geoffrey K. Gill y Chris F. Kemerer en 1991, esta métrica simplemente la reexpresa como un porcentaje de las líneas lógicas de código.
- Métrica de Halstead: definida por Maurice Halstead en 1977, estas métricas se calculan a partir del número de operadores y operandos en cada función.
- Índice de Mantenibilidad: definido por Paul Oman y Jack Hagemeister en 1991, esta es una escala logarítmica desde el infinito negativo hasta 171, calculado a partir de las líneas lógicas de código, la complejidad de ciclomática y el esfuerzo de Halstead.
- Dependencias: un recuento de las llamadas a CommonJS y AMD require. Analizado estáticamente desde la firma de la función, por lo que no se realiza ninguna contabilidad para las llamadas dinámicas donde una variable o función está ocultando la naturaleza de la dependencia.
- Densidad de primer orden: el porcentaje de todas las posibles dependencias internas que realmente se realizan en el proyecto.
- Costo de cambio: el porcentaje de módulos afectados, en promedio, cuando se cambia un módulo en el proyecto.
- Tamaño del núcleo: el porcentaje de módulos de los que dependen, y dependen en gran medida de otros módulos.

3.5.2 Gulp-complexity

Gulp-complexity [33] está basada en la herramienta escomplex que se describió en la sección anterior. Esta herramienta, se centra en el índice de mantenibilidad de los proyectos en JavaScript. El índice de mantenibilidad (IM) es un modelo de mantenibilidad de software que fue propuesto por Omán y Hagemeister en la Universidad de Idaho en 1991. Este modelo

consiste en un número de métricas calculadas fácilmente, y es capaz de predecir fácil y rápidamente la mantenibilidad de un producto de software.

Para realizar el cálculo del índice de mantenibilidad, se tienen en cuenta cuatro factores (métricas):

1. El porcentaje de Esfuerzo Halstead por módulo
2. El promedio de la complejidad ciclomática extendida por módulo
3. El porcentaje del número de líneas de código por módulo
4. El porcentaje de líneas de comentarios por módulo

El valor más importante del índice de mantenibilidad es la complejidad ciclomática. Por eso, al realizar el análisis de los datos que se obtienen luego de ejecutar la herramienta, se centran en el valor de esta métrica y, seguramente, si se revisan las funciones del archivo que contiene un valor de complejidad ciclomática alto, estas tengan varios condicionales.

Usar este tipo de índices, para el desarrollar es muy importante ya que permite obtener código altamente mantenible.

3.6 Resumen

Luego de presentar las características principales, y destacar los aspectos importantes de los diferentes trabajos que se mencionaron en este capítulo, se puede decir que no fue fácil hallar trabajos relacionados al trabajo realizado, debido a que, el análisis de código JavaScript en relación a la búsqueda de code smells, no es algo que esté muy estudiado en la actualidad. Los trabajos que se pueden encontrar son bastante recientes, y presentan análisis de diferentes code smells en el lenguaje JavaScript. En la mayoría de los casos el análisis se acota a unos pocos code smells.

Lo más avanzado y/o estudiado actualmente, son las herramientas que evalúan el código estático, y que el análisis que realizan está orientado al cumplimiento de los estándares de codificación.

También se pueden hallar herramientas del tipo descritas en la sección 3.5, que no realizan un análisis de code smells, sino que simplemente obtienen el valor de métricas en el código JavaScript. Este tipo de herramientas se comparó con la herramienta vcomplex que se implementó como parte de este trabajo para poder realizar el análisis de code smells a través de sus estrategias de detección. Pero es muy importante el destacar que no representan el análisis realizado en este trabajo.

Los demás enfoques que se presentaron en este trabajo, si bien realizan un análisis de problemas en en código JavaScript, ninguno propone un catálogo de métricas que se defina

para realizar el análisis code smells a partir de estrategias de detección. Hasta el momento, no se ha podido hallar un trabajo que realice dicho análisis, esto fue la motivación más importante del trabajo realizado.

La definición de un catálogo de métricas, como los definidos para el lenguaje Java que son de gran conocimiento en el mundo del desarrollo de software y muy utilizados por los desarrolladores, es necesaria para mantener un orden y una estrategia clara para la detección de code smells en el lenguaje que se utilicen. En este caso, sería sumamente necesario, debido al gran crecimiento del lenguaje a nivel mundial, que se defina un catálogo de métricas para el lenguaje JavaScript con el objetivo de mejorar el mantenimiento de los sistemas que se desarrollen en dicho lenguaje.

Capítulo 4

Análisis de Code Smells

En el presente capítulo, se realiza el análisis de los code smells que se definen para el lenguaje Java, con el objetivo de determinar si se pueden detectar, también, en el lenguaje JavaScript. El análisis, se centra en determinar si las estrategias de detección, que se definen a partir del uso de las métricas que presentan Lanza y Marinescu en su catálogo de métricas [1], se pueden usar o no en el lenguaje JavaScript, o si es necesario adaptar alguna de las métricas que utilizan para su cumplimiento.

En cada una de las Secciones que se presentan en el capítulo, se analiza la estrategia de detección de un code smell en particular:

- En la Sección 4.1, se analiza el Code Smell Brain Method.
- En la Sección 4.2, se analiza el Code Smell Data Class.
- En la Sección 4.3, se analiza el Code Smell God Class.
- En la Sección 4.4, se analiza el Code Smell Intensive Coupling.
- Finalmente, en la Sección 4.5 se analiza el Code Smell Dispersed Coupling.

4.1 Brain Method

El code smell Brain Method se caracteriza por centralizar la funcionalidad de una clase.

4.1.1 Descripción

Cuando se comienza a implementar un método se piensa en la funcionalidad básica del mismo, y a medida que se necesita nueva funcionalidad se va agregando. Si la funcionalidad que se agrega es demasiada, el método se vuelve un método grande y complejo. Además, a medida que pasa el tiempo, para el desarrollador, se vuelve difícil de entender, mantener y reutilizar debido a que la funcionalidad no es clara [1].

Debido a las características mencionadas del método, el costo de realizar un cambio ante la definición de nuevos requerimientos del sistema, que involucra al Brain Method, es alto. De la misma manera, es baja la probabilidad de reutilizar su funcionalidad para un nuevo componente del sistema.

4.1.2 Detección

Brain Method se identifica a partir de un conjunto de cuatro condiciones, cada una de ellas se representa a partir de las métricas definidas para su detección. La Figura 4.1, presenta la regla de detección del Brain Method propuesta por Lanza y Marinescu.

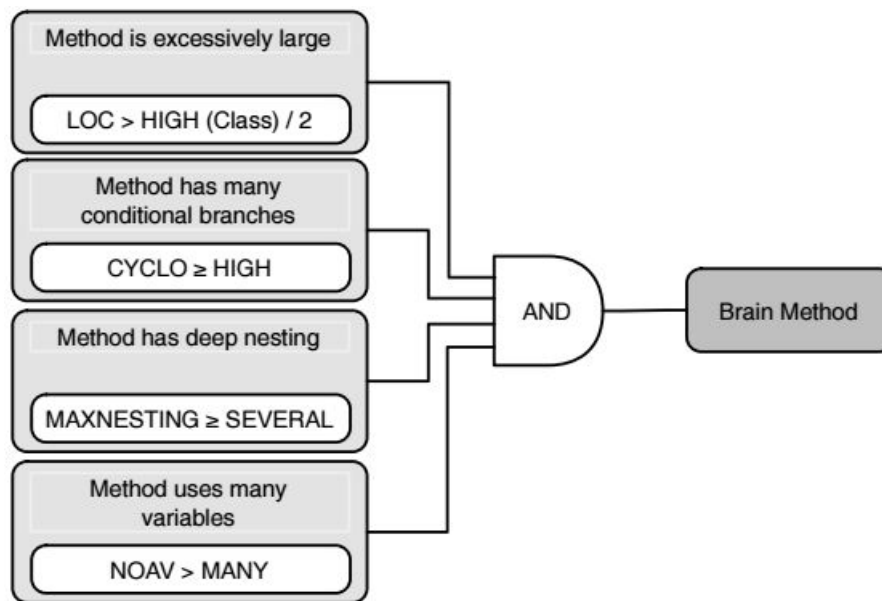


Figura 4.1 - Estrategia de detección del Brain Method.

Para que la estrategia de detección se cumpla, la primer condición que se presenta es que un método sea excesivamente largo. Se considera que los métodos largos son indeseables ya que afectan a la comprensibilidad y la capacidad de testing del código. Este tipo de métodos tiende a realizar y/o centralizar demasiada funcionalidad de una clase, y es probable que utilicen muchas variables y parámetros temporales, haciéndolos más propensos a errores.

La métrica que se utiliza para medir y determinar si un método es demasiado largo es LOC (Líneas de código) [5], esta métrica cuenta la cantidad de líneas de código del método. A partir del valor de métrica obtenido, la condición evalúa que el mismo sea mayor a un umbral definido para un método de una clase. Por lo tanto, se determina que si el recuento de líneas de código del método es mayor al umbral, el método es excesivamente grande.

La segunda condición, analiza si un método posee una cantidad alta de saltos condicionales. Para evaluar esta condición, se considera el uso excesivo de instrucciones que provocan un salto condicional en el flujo básico como lo son if, else, while, entre otras.

Para determinar que un método posee demasiados saltos condicionales se utiliza la métrica CYCLO (Complejidad Ciclomática), la cual proporciona una medición cuantitativa de la complejidad condicional de un programa. CYCLO, es una de las métricas de software de mayor aceptación, ya que ha sido concebida para ser independiente del lenguaje. Su objetivo principal es contar el número de caminos diferentes que existen dentro de un fragmento de código. Se utiliza el método del camino básico [2] para obtener una medida de la complejidad de un diseño procedimental, y utilizar esta medida como guía para la definición de una serie de caminos básicos de ejecución, diseñando casos de prueba que garanticen que cada camino se ejecuta al menos una vez.

En la tercer condición, se evalúa cuando un método posee demasiado anidamiento. Se considera que los métodos que poseen un nivel de anidamiento profundo para que se cumpla una condición y se ejecute una acción, no son intuitivos y son complejos para realizar pruebas y entender el código [1].

La métrica que se utiliza para medir si un método posee mucho anidamiento es MAXNESTING (Máximo nivel de anidamiento). Esta métrica considera el máximo nivel de anidamiento de las estructuras de control dentro de un método o función.

Por último, la cuarta condición determina si el método utiliza muchas variables. Se considera que, cuando el método utiliza más variables de las que una persona puede retener en su memoria a corto plazo aumenta el riesgo a introducir errores.

Se utiliza la métrica NOAV (Número de variables a las que accede el método) para determinar si un método utiliza demasiadas variables. Para dicha métrica se consideran variables locales, parámetros, atributos y variables globales, el recuento de estas variables comparado con un umbral determinado define si el método es un método que utiliza muchas variables o no.

Por lo tanto, al cumplirse cada una de las condiciones que se describen para la estrategia de detección presentada, se determina que el método es un Brain Method.

4.1.3 Análisis de las métricas en JavaScript

En esta sección, se analizan cada una de las cuatro métricas que se definen en la estrategia de detección de un Brain Method para determinar si son aplicables o no en el lenguaje JavaScript, si es necesario adaptarlas.

Como ejemplo para describir cada una de las métricas se considera el software Chart.js [10]. El mismo, se utiliza para realizar gráficos y se encuentra destinado a desarrolladores y/o diseñadores.

Para analizar las métricas presentes en la estrategia de detección de un Brain Method, se eligió la función draw que se muestra en la Figura 4.2. La función draw dibuja el bloque del título en el lienzo del gráfico, es decir, define el título del gráfico. La Figura 4.3 muestra el código del método draw en el lenguaje Java. El mismo tiene la misma funcionalidad descrita para la función draw en el lenguaje JavaScript.

Al observar la Figura 4.2, se puede ver que la función se define con el nombre que la identifica seguido de dos puntos, y el término function; para este caso en particular la función no recibe parámetros. También se observa que, en la declaración de variables se utiliza sólo la palabra reservada var sin ninguna restricción como poder ser el tipo o un modificador de

acceso (public, private y/o protected). Además, al utilizar la palabra reservada var la variable se define como local a la función.

En la Figura 4.3, donde se muestra el método draw en el lenguaje Java, se puede observar que la función se define detallando el ámbito, el tipo de retorno y el nombre del método. Además, los parámetros que recibe el método se encuentran tipados, así como también las variables que se declaran.

Al observar ambas figuras se puede diferenciar la manera en que se aplica la programación orientada a objetos en ambos lenguajes.

```
-: draw: function() { → CYCLO 1
1:   var me = this, //this hace referencia al título del gráfico → NOAV 1
2:     ctx = me.ctx, //ctx es el lienzo del gráfico → NOAV 2
3:     valueOrDefault = helpers.getValueOrDefault, → NOAV 3
4:     opts = me.options, → NOAV 4
5:     globalDefaults = Chart.defaults.global; → NOAV 5
6:
7:   if (opts.display) { → CYCLO 2 → MAXNESTING 1
8:     var fontSize = valueOrDefault(opts.fontSize, globalDefaults.defaultFontSize),
9:         → NOAV 6
10:    fontStyle = valueOrDefault(opts.fontStyle, globalDefaults.defaultFontStyle),
11:    → NOAV 7
12:    fontFamily = valueOrDefault(opts.fontFamily, globalDefaults.defaultFontFamily),
13:    → NOAV 8
14:    titleFont = helpers.fontString(fontSize, fontStyle, fontFamily), → NOAV 9
15:    rotation = 0, → NOAV 10
16:    titleX, → NOAV 11
17:    titleY, → NOAV 12
18:    top = me.top, → NOAV 13
19:    left = me.left, → NOAV 14
20:    bottom = me.bottom, → NOAV 15
21:    right = me.right, → NOAV 16
22:    maxWidth; → NOAV 17
23:
24:   ctx.fillStyle = valueOrDefault(opts.fontColor, globalDefaults.defaultFontColor),
25:   ctx.font = titleFont;
26:
27:   if (me.isHorizontal()) { → CYCLO 3 → MAXNESTING 2
28:     titleX = left + ((right - left) / 2); //Punto medio del ancho
29:     titleY = top + ((bottom - top) / 2); //Punto medio de la altura
30:     maxWidth = right - left;
31:   } else {
    titleX = opt.position === 'left' ? left + (fontSize / 2) : right - (fontSize / 2);
```

```

32:     titleY = top + ((bottom - top) / 2);
33:     maxwidth = bottom - top;
34:     rotation = Math.PI * (opt.position === 'left' ? -0.5 : 0.5);
35: }
36:
37: ctx.save();
38: ctx.translate(titleX,titleY);
39: ctx.rotate(rotation);
40: ctx.textAlign = 'center';
41: ctx.textBaseline = 'middle';
42: ctx.fillText(opts.text, 0, 0, maxwidht);
43: ctx.restore();
44: }
-: }

```

Figura 4.2 - Función draw en lenguaje JavaScript - Código ejemplo Brain Method

```

public Object draw(Graphics2D g2, Rectangle2D area, Object params) {
    if (this.content == null) {
        return null;
    }
    area = trimMargin(area);
    drawBorder(g2, area);
    if (this.text.equals("")) {
        return null;
    }
    ChartEntity entity = null;
    if (params instanceof EntityBlockParams) {
        EntityBlockParams p = (EntityBlockParams) params;
        if (p.getGenerateEntities()) {
            entity = new TitleEntity(area, this, this.toolTipText,this.urlText);
        }
    }
    area = trimBorder(area);
    if (this.backgroundPaint != null) {
        g2.setPaint(this.backgroundPaint);
        g2.fill(area);
    }
    area = trimPadding(area);
    RectangleEdge position = getPosition();
    if (position == RectangleEdge.TOP || position == RectangleEdge.BOTTOM) {
        drawHorizontal(g2, area); //Dibuja el título horizontal dentro del área especificada
    }
    else if (position == RectangleEdge.LEFT || position == RectangleEdge.RIGHT) {
        drawVertical(g2, area); //Dibuja el título vertical dentro del área especificada
    }
    BlockResult result = new BlockResult();
    if (entity != null) {
        StandardEntityCollection sec = new StandardEntityCollection();
        sec.add(entity);
        result.setEntityCollection(sec);
    }
}

```

```
    return result;  
}
```

Figura 4.3 - Función draw en lenguaje Java

Métrica LOC

La métrica a evaluar cuenta la cantidad de líneas de código de un método y se utiliza para definir si un método es demasiado largo.

En el lenguaje Java, para obtener el valor de la métrica LOC, se cuentan las líneas de código que contiene el método. Se considera, para realizar la cuenta, desde la línea donde se define el método hasta la llave que delimita el final del mismo.

Para evaluar la métrica LOC en el lenguaje JavaScript, se realiza el recuento de líneas de código de la función de la misma manera que se realiza para el lenguaje Java, considerando desde donde se define la función hasta la llave que delimita el final.

En ambos lenguajes descritos anteriormente, se utilizan llaves para delimitar el método/función sobre el cual se aplicará la métrica. Además, la definición de cualquiera de las operaciones nombradas se realiza con el nombre que las identifica. Por lo tanto, se concluye que se puede usar la métrica LOC en el lenguaje JavaScript sin necesidad de adaptarla.

En la función draw de Chart.js [10], se cuenta la cantidad de líneas de código desde la definición de la función, donde se encuentra el nombre que la identifica, hasta la llave de cierre que indica el final. Por cada una de las líneas de código se suma uno a la cantidad de líneas parcial. En la Figura 4.2 se muestra el código de la función con el número de línea al comienzo de la misma, para la función draw la métrica LOC tiene como valor 44.

Métrica CYCLO

La métrica CYCLO, evalúa si un método posee una cantidad alta de saltos condicionales.

En el lenguaje Java, la métrica CYCLO se obtiene utilizando la teoría de grafos. Para aplicar la teoría de grafos, se consideran los distintos caminos posibles que existen; dichos caminos se encuentran definidos por las sentencias condicionales presentes en el código que se analiza.

Para el análisis de la métrica en JavaScript, se debe destacar el hecho de que fue pensada para ser independiente del lenguaje [15], con lo cual se presume que la métrica sea totalmente válida en el lenguaje JavaScript. Al igual que en Java, en JavaScript también se utiliza la

teoría de grafos para calcular CYCLO y el anidamiento se determina de la misma manera, se utilizan sentencias condicionales y se encuentra delimitado por llaves.

De acuerdo a las características mencionadas, no es necesario adaptar la métrica CYCLO para su uso en el lenguaje JavaScript.

En la Figura 4.2 se marcan como comentarios, en color rojo a la derecha de la línea de código, las sentencias condicionales que se consideran para calcular la complejidad ciclomática de la función draw. Se comienza con el valor de métrica en uno, que es valor que se le da al flujo normal básico, y se suma de a uno por cada sentencia condicional presente en el código. Para este caso en particular las sentencias condicionales presentes en el código para evaluar esta métrica son las sentencias if's. Por lo tanto para la métrica CYCLO analizada, se obtuvo el valor 3.

Métrica MAXNESTING

La métrica MAXNESTING evalúa si el nivel de anidamiento en el código analizado es alto.

En el lenguaje Java, se consideran las sentencias condicionales para calcular la métrica MAXNESTING. A partir de las sentencias condicionales presentes en el código y el anidamiento de las mismas, se obtiene el valor de la métrica. Además, se considera para el cálculo de la métrica, que el anidamiento de las sentencias condicionales se ve reflejado en el código por el uso de llaves [16].

En JavaScript la métrica MAXNESTING se calcula considerando las sentencias condicionales presentes en el código y el nivel de anidamiento de las mismas. El anidamiento en el lenguaje JavaScript se define de la misma manera que en Java, mediante el uso de llaves como caracteres de delimitación.

A partir del análisis realizado para el cálculo de la métrica MAXNESTING en ambos lenguajes, se determina que no es necesario adaptar la métrica para su uso en el lenguaje JavaScript. Se llega a dicha conclusión debido a la similitud de ambos lenguajes para definir las sentencias condicionales y el anidamiento de las mismas.

En la Figura 4.2, se marcan las sentencias condicionales que se consideran para calcular el máximo nivel de anidamiento en la función draw que se utiliza como ejemplo. El valor que se obtuvo para la métrica analizada, considerando las sentencias condicionales if que se encuentran anidadas por medio de llaves, es 2.

Métrica NOAV

La métrica NOAV determina si un método utiliza demasiadas variables.

Para el lenguaje Java, se consideran variables globales, atributos, parámetros y variables locales. El ámbito de las variables para el cálculo de la métrica no se tiene en cuenta; una variable de ámbito local se considera de la misma manera que una variable de ámbito global o estático. Tampoco se considera el tipo de la variable para el cálculo de la métrica. La suma de todas las variables a las que usa y/o accede el método es el valor de la métrica NOAV.

Para el cálculo de la métrica NOAV en el lenguaje JavaScript, se tienen en cuenta las variables globales, los parámetros y las variables locales. La suma de las variables que utiliza la función es el valor que toma la métrica. JavaScript es un lenguaje no tipado y que sólo define dos ámbitos: global y local; sin embargo dichas características no afectan el cálculo de la métrica ya que no se considera el ámbito ni el tipo de las variables para el mismo.

Como para la métrica que se analiza sólo se considera el uso de variables sin considerar el ámbito y el tipo de las mismas, se concluye que no es necesario adaptar la métrica utilizada en el lenguaje Java para el lenguaje JavaScript. En ambos lenguajes, se debe contar la cantidad de variables que usa la función (o el método) que se está evaluando, y se debe comparar el resultado obtenido con el límite máximo de variables que puede utilizar la función (o el método) para definir si se está en presencia de un Brain Method.

En la Figura 4.2, se destacan las variables consideradas para calcular la métrica NOAV en el código de la función draw que se usa como ejemplo; a partir de las mismas, el valor de métrica es 17. Se consideran las variables declaradas dentro de la función, y las variables que se utilizan de otras funciones y/o módulos.

4.1.4 Conclusión Code Smell Brain Method

Luego de describir cada una de las métricas y realizar un análisis de las mismas para considerar si se deben o no adaptar para su uso en el lenguaje JavaScript, se llega a la conclusión de que no es necesario adaptar ninguna de las métricas presentadas por Lanza y Marinescu para la detección del Brain Method. Es decir, es posible aplicar las mismas métricas para detectar métodos complejos que centralizan la funcionalidad en un archivo js.

4.2 Data Class

El code smell Data Class, se centra en clases que sólo contienen atributos públicos y métodos de acceso (get's/set's) a los mismos.

4.2.1 Descripción

La falta de métodos funcionalmente relevantes puede indicar que los datos y comportamiento relacionados no están en la misma clase. Por lo tanto, este tipo de clases rompen con los

principios de un diseño orientado a objetos, debido a que permiten que otras clases vean y, posiblemente, manipulen sus datos.

En términos de atributos de calidad, el code smell analizado reduce la factibilidad de que un sistema sea mantenible, fácil de testear y comprensible. Esto se debe a que si se producen cambios en la Data Class, seguramente afecten otras clases que usan sus datos. Además, si dos clases usan alguno de los datos exactamente de la misma manera, aparecerá código duplicado, y en consecuencia, el esfuerzo que se requiere para probar el código aumentará en comparación a testear sólo un método que se encuentra en la Data Class. De la misma manera, la comprensibilidad de las clases se ve afectada debido a que la semántica de la funcionalidad se encuentra en la Data Class [6].

Por lo tanto, una solución posible para un Data Class se centraría en “mover” todos o parte de los métodos que realizan cálculos sobre los atributos, y se encuentran en otras clases, a la Data Class.

4.2.2 Detección

Las condiciones que se presentan para la detección del code smell Data Class, se basan en sus características; se buscan clases que no proporcionen casi ninguna funcionalidad, que contengan atributos públicos y que sólo definan métodos de acceso (get's/set's) sobre esos atributos.

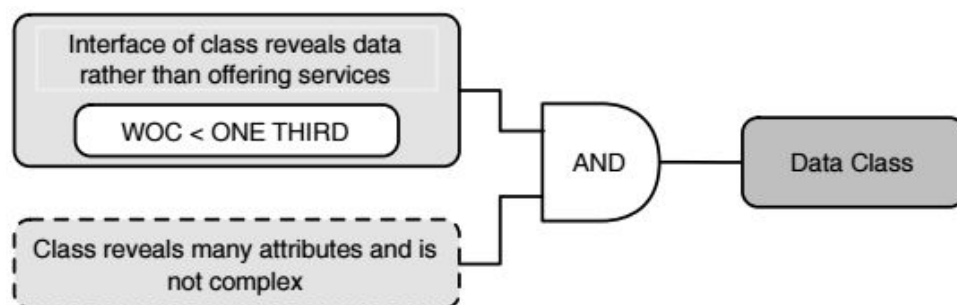


Figura 4.4.1 - Estrategia de detección del Code Smell Data Class

En la primer condición de la estrategia de detección para el code smell Data Class (Figura 4.4.1), se destaca que la interfaz de la clase está exponiendo datos en lugar de proporcionar servicios. Para esta condición se considera que los datos de la clase se exponen al ser definidos como públicos, de esta manera las demás clases pueden acceder a los mismos sin la necesidad de que se acuda a un método de acceso.

La métrica que se utiliza para determinar si una interfaz de clase revela datos en vez de ofrecer servicios, es la métrica WOC (peso de clase). Dicha métrica, calcula el número de métodos públicos "funcionales", dividido el número total de atributos públicos [3]. Por

métodos públicos funcionales se refiere a los métodos que no son métodos de acceso (get's/set's) de la clase [6].

A partir de los resultados que se obtienen, se evalúa si la interfaz de la clase contiene principalmente datos y métodos de acceso, en lugar de proveer servicios, y se determina el grado de funcionalidad de la misma. La interfaz de la clase debe contener sólo los métodos que implementan el comportamiento de la clase [6]. Cuanto menor sea el valor de la métrica WOC para una clase, más indicios de que la clase sea una Data Class.

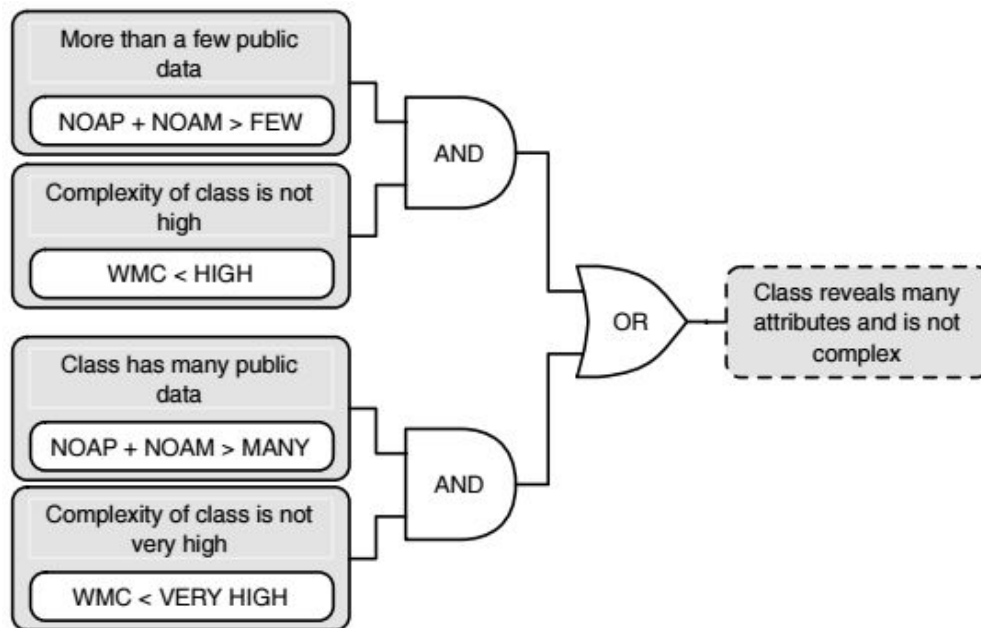


Figura 4.4.2 - Estrategia de detección del Code Smell Data Class

Para que se cumpla la condición de la estrategia de detección, donde se determina que la clase revela muchos atributos y no es compleja, se debe cumplir alguna de sus condiciones previas (Figura 4.4.2).

En el primer caso se definen dos condiciones, en la primer condición se determina si la clase contiene más de unos pocos atributos públicos, y en la segunda, se determina si la complejidad de la clase no es alta.

Para evaluar la primer condición, se consideran los atributos que se declaran en la clase como públicos, y los métodos de acceso de la misma. Las métricas que se utilizan para esta condición, son NOAP (Número de atributos públicos de una clase) y NOAM (Número de métodos de acceso - get's/set's - de una clase).

Al evaluar la segunda condición, se tiene en cuenta la complejidad que posee la clase. El número de métodos y la complejidad de los métodos involucrados es un indicador de cuánto tiempo y esfuerzo se requiere para desarrollar y mantener la clase. Además, es probable que

las clases con un gran número de métodos sean más específicas de la aplicación en particular a la que pertenecen, y para la cual fueron diseñadas, limitando la posibilidad de reutilización [7].

La métrica que se utiliza para obtener la complejidad de la clase es WMC. Dicha métrica, es la suma de las complejidades de todos los métodos de una clase.

Por lo tanto para que se cumpla el primer caso, si la clase no tiene una WMC (Complejidad estática de todos los métodos de la clase) alta, donde se presume que no se encuentren muchos datos públicos, se espera que la clase tenga más de unos pocos atributos públicos.

En el segundo caso también se definen dos condiciones, en la primer condición se considera que la clase tiene muchos atributos públicos, y en la segunda condición se determina si la complejidad de la clase no es muy alta.

Las métricas que se utilizan para que se cumplan las condiciones son las mismas que se describieron para el primer caso. Se diferencian en la comparación que se realiza de los valores obtenidos como resultado de las métricas. Se considera que la clase contiene muchos datos públicos y, al mismo tiempo la complejidad de la clase no sea demasiado alta debido a que una clase con complejidad muy alta no encaja conceptualmente en lo definido como una Data Class.

Por lo tanto, en el primer y segundo caso se hace una relación proporcional entre la complejidad de la clase y la cantidad de atributos públicos de la misma, considerando también los métodos de acceso, para definir si se está en presencia de una clase que revela muchos atributos y no es compleja. Si se determina lo descrito y la interfaz de la clase revela datos en lugar de definir funcionalidad, se está en presencia del code smell Data Class.

4.2.3 Análisis de las métricas en JavaScript

En esta sección, se realiza el análisis de las cuatro métricas que se definen para la estrategia de detección del code smell Data Class, se determina si son aplicables o no en el lenguaje JavaScript, y si es necesario adaptarlas.

Para analizar si las métricas del code Smell Data Class se pueden usar en el lenguaje JavaScript o si es necesario adaptarlas, se utiliza el mismo software de ejemplo, Chart.js [10], que en el análisis del code smell Brain Method. En este caso se considera el código del archivo js “core.element.js” que se muestra en la Figura 4.5.

```
var color = require('chartjs-color');  
-- Variable local al archivo js y global a las funciones y/o módulos que contiene
```

```

module.exports = function(Chart){
  var helpers = Chart.helpers;
  -- Variable local del módulo y global a las funciones que contiene

  function interpolate(start, view, model, ease) { → CYCLO 1 = 1
    var keys = Object.keys(model); -- Variable local a la función
    var i, ilen, key, actual, origin, target, type, c0, c1;
    -- Variables locales a la función

    for (i=0, ilen=keys.length; i<ilen; ++i){ → CYCLO 1 = 2
      key = keys[i];
      target = model[key];

      if (!view.hasOwnProperty(key)){ view[key] = target; } → CYCLO 1 = 3

      actual = view[key];

      if (actual === target || key[0] === '_'){ continue; } → CYCLO 1 = 4

      if (!start.hasOwnProperty(key)){ start[key] = actual; } → CYCLO 1 = 5
      origin = start[key];
      type = typeof(target);

      if (type === typeof(origin)){ → CYCLO 1 = 6
        if (type === 'string'){ → CYCLO 1 = 7
          c0 = color(origin);
          if (c0.valid){ → CYCLO 1 = 8
            view[key] = c1.mix(c0, ease).rgbString();
            continue;
          }
        } else if (type === 'number' && isFinite(origin) &&
          isFinite(target)){ → CYCLO 1 = 9
          view[key] = origin + (target - origin) * ease;
          continue;
        }
      }
      view[key] = target;
    }
  }

  Chart.elements = {};
  Chart.Element = function(configuration){ → CYCLO 2 = 1
    helpers.extend(this, configuration);
    this.initialize.apply(this, arguments);
  };

  helpers.extend(Chart.Element.prototype, {
    initialize: function(){ → CYCLO 3 = 1
      this.hidden = false;
    },
  },

```

```

    pivot: function() { → CYCLO 4 = 1
        var me = this; -- Variable local a la función
        if (!me._view) { me._view = helpers.clone(me._model); } → CYCLO 4 = 2
        me._start = {};
        return me;
    },

    transition: function(ease){ → CYCLO 5 = 1
        var me = this; -- Variable local a la función
        var model = me._model; -- Variable local a la función
        var start = me._start; -- Variable local a la función
        var view = me._view; -- Variable local a la función

        if (!model || ease === 1) { → CYCLO 5 = 2
            me._view = model;
            me._start = null;
            return me;
        }

        if (!view) { view = me._view = {} } → CYCLO 5 = 3

        if (!start) { start = me._start = {}; } → CYCLO 5 = 4

        interpolate(start, view, model, ease);
        return me;
    },

    tooltipPosition: function(){ → CYCLO 6 = 1
        return {
            x: this._model.x,
            y: this._model.y,
        };
    },

    hasValue: function() { → CYCLO 7 = 1
        return helpers.isNumber(this._model.x) && helpers.isNumber(this._model.y);
    }
});
Chart.Element.extend = helpers.inherits;
};

```

Figura 4.5: Archivo js en el lenguaje JavaScript. Código ejemplo Data Class

Métrica WOC

Esta métrica determina si en la interfaz de la clase se definen más atributos públicos que métodos públicos.

En el lenguaje Java para obtener el valor de la métrica que se analiza, se divide el número de métodos funcionales públicos de la clase por el número de atributos públicos de la misma. A partir del resultado que se obtiene de la división, se evalúa si al menos un tercio de la interfaz de la clase, está exponiendo datos en lugar de proporcionar servicios.

Para analizar la métrica en el lenguaje JavaScript, se debe tener en cuenta que no existen tipos como interfaces ni clases abstractas, por lo que no se puede definir una interfaz de la forma tradicional. En JavaScript sólo se permite la herencia simple, y no hay soporte incorporado para las interfaces. Sin embargo, JavaScript es un lenguaje flexible y trata de que se pueda seguir el diseño orientado a objetos a través del uso de funciones y prototipos; de la misma manera define una alternativa para que se implemente el concepto de interfaz.

A pesar de las características que se mencionaron del lenguaje, no es posible detectar si en realidad es una interfaz la función que se definió en un archivo js como tal, debido a que se declara de la misma manera que una función tradicional, y no hay una convención “universal” que se utilice al definir una interfaz entre todos los desarrolladores del lenguaje JavaScript.

Por lo tanto, a partir de lo descrito sobre las características del lenguaje JavaScript que se necesitan para evaluar la métrica WOC, que forma parte de la estrategia de detección del code smell Data Class, se concluye que no se podría adaptar la misma para el lenguaje JavaScript.

Sin embargo, al no haber manera de definir una interfaz en el lenguaje JavaScript, se podrían considerar las funciones no anidadas (métricas MAXNESTING de la función igual a uno) del archivo js para realizar el cálculo de la métrica, debido a que una función no anidada que se encuentre definida a partir del método export puede ser utilizada en cualquier otro archivo js que realice el import de la misma o del módulo que la contiene. De esta manera, se calcula la cantidad de funciones no anidadas, a las cuales se les realiza export, dividido la cantidad de variables públicas que contiene el archivo js, para obtener el valor de métrica WOC en el lenguaje JavaScript.

Métrica NOAP

La métrica NOAP, determina el número de atributos públicos de la clase.

Para evaluar la métrica en el lenguaje Java, se consideran los atributos que se definen en la clase como públicos. En Java un atributo se define como público, cuando se utiliza la leyenda public antes de que se defina el tipo y el nombre del mismo. Por lo tanto, el valor de métrica NOAP para el lenguaje Java es la suma de dichos atributos públicos.

En el lenguaje JavaScript, se debe considerar para analizar la métrica NOAP que no se definen ámbitos como público o privado, sino que se definen como global o local. A partir de esta diferenciación y mediante el uso de la palabra reservada `var`, se puede “simular” el uso de variables dentro de un archivo `js` como si se tratara de atributos públicos y privados en JavaScript.

Se debe aclarar que el uso de la variable `var` para el lenguaje JavaScript no es equivalente al uso de la palabra reservada `private` para un atributo de la clase en el lenguaje Java. El primer caso se refiere a una variable local en la función que fue declarada, y el segundo caso se refiere a una variable privada para la clase. Para realizar el análisis de la métrica en el lenguaje JavaScript, se consideran las variables en el archivo `js` que no posean la palabra reservada `var` y se definan mediante el uso de `this`, se encuentren o no declaradas dentro de una función del mismo.

Si se declara una variable dentro de una función con el prefijo `var`, se podrá acceder únicamente desde la función en la que fue declarada; por lo tanto, la variable se volvería local a la función. A diferencia de lo descrito, si delante de la variable no se agrega el prefijo `var`, la variable se considerará como una variable global y se podrá utilizar desde cualquier parte del módulo global que contenga la función. De la misma manera, se utiliza la palabra reservada `var` para definir el ámbito de una variable dentro de un `js` como atributo de la clase. Además, si un atributo dentro de una función se describe con `this`, podrá ser accedido directamente desde cualquier otro archivo `js`.

En la Figura 4.5, donde se muestra el código de ejemplo para el lenguaje JavaScript, se puede realizar una distinción entre los ámbitos, y por lo tanto el alcance, de las variables que se declaran. Por ejemplo, la variable “color” se declara con `var` dentro del archivo `js`, con lo cual se puede usar desde cualquier parte del mismo, pero su valor no puede ser accedido por otros archivos `js` sino es por medio de un método de acceso. Por lo tanto, la variable “color” es local al archivo `js` y se puede utilizar desde cualquier componente que se encuentre definido en el mismo; para dichos componentes la variable será de ámbito global.

Otro ejemplo es la variable “keys” que al ser definida con `var`, su valor se puede acceder únicamente desde dentro de la función donde se declaró; si la variable no se hubiera declarado con `var`, se consideraría global al archivo `js`. Si la variable se hubiese definido sin `var` y con `this`, y además se realiza el `export` de la función que la contiene, la misma se podría acceder desde otros archivos `js` que realicen el `require` del archivo `js`.

De acuerdo a lo descrito, se debe adaptar la métrica analizada. Se necesita considerar las variables globales al archivo `js` que no se definan con `var` y, además, las variables dentro de las funciones que posee el `js` que tampoco se hayan definido con `var`, considerando que en ambos casos se definan mediante el uso de `this`.

El análisis de la métrica NOAP, y la obtención del valor de la misma para el lenguaje JavaScript, es más complejo que para el lenguaje Java, se deben tener en cuenta más condiciones debido a la manera en la que se definen los ámbitos local y global en el lenguaje.

Métrica NOAM

Para la métrica NOAM, se consideran los métodos de acceso (get's/set's) de la clase.

En Java, se obtiene el valor de NOAM considerando los métodos que se definen con el prefijo get y set en el nombre del método; dichos métodos siempre se definen como públicos. El método get es un método que se utiliza como “consultor” debido a que su funcionalidad es retornar la información del atributo consultado; y el método set se utiliza como un método “modificador”, ya que permite que se modifique uno o varios de los atributos del objeto.

La cantidad de métodos que presenten las características descritas, será el valor de la métrica NOAV para el lenguaje Java.

En el lenguaje JavaScript, los métodos de acceso get y set se definieron en el estándar ECMAScript 5 del lenguaje, y conceptualmente son iguales a los métodos get y set en el lenguaje Java. Sin embargo, el uso de este tipo de métodos en JavaScript no es muy utilizado por los desarrolladores y, además, no es compatible con todos los navegadores [4]; por ejemplo, IE8 (Internet Explorer 8) no soporta este tipo de métodos.

Por lo tanto, no sería viable adaptar esta métrica para el lenguaje JavaScript debido a que, no se podría asegurar la obtención del valor de la misma para cada uno de los sistemas que se analicen.

La métrica analizada junto a la métrica NOAP, se utilizan para definir si la clase contiene muchos atributos públicos. Como no se puede obtener un valor fiable para la métrica NOAM, no se podría cumplir la condición descrita.

Métrica WMC

La métrica WMC se evalúa para obtener la complejidad de la clase, y es la suma de las complejidades de todos los métodos de la misma.

En el lenguaje Java, para el cálculo de la métrica WMC se considera la métrica CYCLO, la cual representa la complejidad funcional de un método considerando las sentencias condicionales del mismo. La suma de cada uno de los valores obtenidos, para cada método de la clase, es el valor de la métrica WMC.

La métrica CYCLO se analizó para el code smell Brain Method, donde se determinó que no es necesario adaptarla para su uso en JavaScript. Por lo tanto, como la métrica WMC es la suma de los valores obtenidos para la métrica CYCLO; se considera para cada una de las funciones del archivo js el valor de métrica CYCLO, y la suma total de dichos valores será el valor final para la métrica WMC del archivo js que se evalúa.

En la Figura 4.5, se destacan, como comentario a la derecha de la línea de código, las sentencias condicionales que se consideran para calcular la complejidad ciclomática del archivo js que representa la clase. En la Figura 4.6, se muestra el valor de métrica CYCLO para la clase de ejemplo.

$\text{CYCLO 1} + \text{CYCLO 2} + \text{CYCLO 3} + \text{CYCLO 4} + \text{CYCLO 5} + \text{CYCLO 6} + \text{CYCLO 7} = 9 + 1 + 1 + 2 + 4 + 1 + 1 = 19$

Figura 4.6 - Cálculo métrica CYCLO para código de ejemplo.

Por lo tanto, luego de que se analice la métrica WMC de la estrategia de detección del code smell Data Class, para determinar si es posible usar la misma en el lenguaje JavaScript o si es necesario adaptarla. Se determina que no es necesario adaptar la métrica WMC del code smell Data Class para el lenguaje JavaScript.

4.2.4 Conclusión Code Smell Data Class

Luego de describir cada una de las métricas y de realizar un análisis de las mismas para considerar si se deben o no adaptar para su uso en el lenguaje JavaScript, se llega a la conclusión que las métricas WOC y NOAP deben ser adaptadas y que la métrica WMC no se necesita adaptar. Además, la métrica NOAM, no se puede detectar en el lenguaje JavaScript. Por lo tanto, no es posible aplicar todas las métricas analizadas de la estrategia de detección del code smell Data Class para el lenguaje JavaScript.

4.3 God Class

El code smell God Class se refiere a clases que tienden a centralizar la inteligencia del sistema.

4.3.1 Descripción

Son clases que "saben o hacen demasiado"; generalmente, una God Class se crea por accidente como consecuencia de funcionalidad que se agrega gradualmente a una clase a lo largo de su evolución.

En un buen diseño orientado a objetos, la inteligencia de un sistema se distribuye uniformemente entre las clases de nivel superior.

Una God Class realiza demasiado trabajo por su cuenta, delegando solamente pequeños detalles a clases triviales y usando los datos de otras clases. Esto tiene un impacto negativo sobre la reutilización y la comprensibilidad de esa parte del sistema; por lo tanto, son un gran problema para el mantenimiento del mismo.

Además, este tipo de clases, rompen con el principio de la programación orientada a objetos que se refiere a que una clase debe tener una única responsabilidad.

4.3.2 Detección

El code smell God Class, se define a partir de la estrategia de detección propuesta por Lanza y Marinescu que se muestra en la Figura 4.7. Para la misma, se identifican tres condiciones cada una de las cuales se representa mediante el uso de métricas.

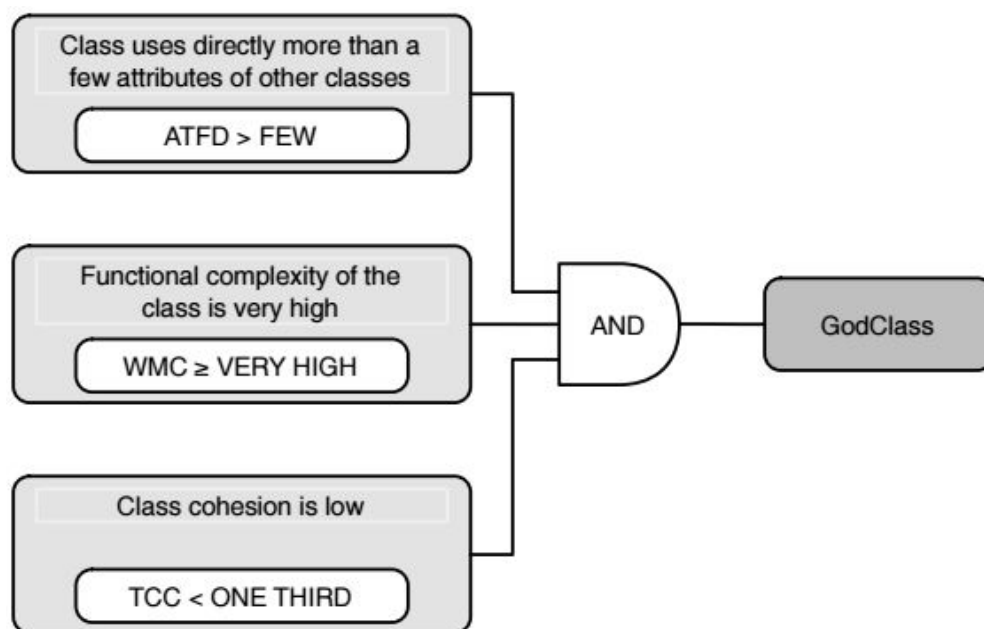


Figura 4.7: Estrategia de detección del Code Smell God Class.

En la primer condición que se presenta para la estrategia de detección del code smell God Class, se evalúa si la clase depende de más de unos pocos atributos de otras clases. La condición descrita, es el principal síntoma para detectar una God Class, ya que mide el grado de ruptura de la encapsulación de los datos. El término encapsulación de datos, se refiere al modo de ocultar información importante de los objetos, como lo son los atributos, pudiendo acceder a dicha información únicamente a través de métodos de acceso. Además, esta práctica de programación orientada a objetos, conserva la flexibilidad y ayuda al buen mantenimiento y a la extensibilidad de la clase.

La métrica que se presenta para cumplir con la condición descrita es ATFD (Acceso a datos foráneos), dicha métrica cuenta la cantidad de accesos desde la clase medida a atributos de

otras clases, ya sea directamente o por medio de métodos de acceso (get's/set's). El valor de métrica que se obtiene, se compara con el límite establecido como válido. Entonces, si el valor es mayor al límite, se cumple la condición; y por lo tanto se concluye que la clase utiliza demasiados atributos de otras clases.

La segunda condición se centra en determinar si la clase posee una complejidad funcional alta. Para determinarlo, se utiliza la métrica WMC que fue descrita para la estrategia de detección del code smell Data Class. Para esta condición en particular, se evalúa si el valor de WMC obtenido, es mayor o igual a un límite establecido como alto. Si la condición se cumple, se considera que la clase tiene una complejidad funcional alta.

Por último, se evalúa la tercer condición para que se cumpla la estrategia de detección. En dicha condición, se determina si la cohesión que posee la clase es baja. El término cohesión se refiere a la "relación" de los componentes del módulo; en este caso en particular, la cohesión de la clase se define en términos del número relativo de métodos conectados en la clase que se analiza. Los desarrolladores de software buscan tener sistemas de alta cohesión y bajo acoplamiento [8]. La métrica que se utiliza para la condición descrita es TCC (Estrecha cohesión de la clase), con la cual se mide el número relativo de pares de métodos de una clase que acceden al menos a un atributo en común de la misma. Una God Class realiza varias funcionalidades distintas que implican conjuntos disjuntos de atributos, lo cual provoca un impacto negativo en la cohesión de la clase. El límite impuesto indica que en la clase que se analiza, menos de un tercio de los pares de métodos tienen en común el uso del mismo atributo.

Con el cumplimiento de cada una de las condiciones que se describieron para la estrategia de detección, se determina que la clase es una God Class.

4.3.3 Análisis de las métricas en JavaScript

En esta sección, se realiza el análisis de las tres métricas que se definen en la estrategia de detección del code smell God Class, para determinar si son aplicables o no en el lenguaje JavaScript y si es necesario adaptarlas.

Para el análisis de las métricas del code smell God Class se usa el Software Chart.js [10], el mismo que se usó en los code smell que se analizaron previamente. En la Figura 4.8 se muestra el código de ejemplo del js Chart.js.

```
var Chart = require('./core/core.js')(); -- Se carga el módulo del archivo js

require('./core/core.helpers')(Chart);
require('./platforms/platform.js')(Chart);
require('./core/core.canvasHelpers')(Chart);
```

```

require('./core/core.element')(Chart);
require('./core/core.plugin.js')(Chart); -- Se llama al js que se analiza en el Figura 15
require('./core/core.animation')(Chart);
require('./core/core.controller')(Chart);
require('./core/core.datasetController')(Chart);
require('./core/core.layoutService')(Chart);
require('./core/core.tick.js')(Chart);
require('./core/core.scale')(Chart);
require('./core/core.interaction')(Chart);
require('./core/core.tooltip')(Chart);

require('./elements/element.arc')(Chart);
require('./elements/element.line')(Chart);
require('./elements/element.point')(Chart);
require('./elements/element.rectangle')(Chart);

require('./scales/scale.linearbase.js')(Chart);
require('./scales/scale.category')(Chart);
require('./scales/scale.linear')(Chart);
require('./scales/scale.logarithmic')(Chart);
require('./scales/scale.radialLinear')(Chart);
require('./scales/scale.time')(Chart);

// Controllers must be loaded after elements
// See Chart.core.datasetController.dataElementType
require('./controllers/controller.bar')(Chart);
require('./controllers/controller.bubble')(Chart);
require('./controllers/controller.doughnut')(Chart);
require('./controllers/controller.line')(Chart);
require('./controllers/controller.polarArea')(Chart);
require('./controllers/controller.radar')(Chart);

require('./charts/Chart.Bar')(Chart);
require('./charts/Chart.Bubble')(Chart);
require('./charts/Chart.Doughnut')(Chart);
require('./charts/Chart.Line')(Chart);
require('./charts/Chart.PolarArea')(Chart);
require('./charts/Chart.Radar')(Chart);
require('./charts/Chart.Scatter')(Chart);

//Loading built-it plugins
var plugins = [];

plugins.push(
    require('./plugins/plugin.filler.js')(Chart),
    require('./plugins/plugin.legend.js')(Chart),
    require('./plugins/plugin.title.js')(Chart)
);

Chart.plugins.register(plugins); -- Se usa la función register del archivo js exportado

module.exports = Chart; -- Se exporta la variable Chart
if (typeof window !== 'undefined'){

```

```
    window.Chart = Chart;  
}
```

Figura 4.8: Archivo js en el lenguaje JavaScript. Código ejemplo God Class.

Métrica ATFD

La métrica ATFD, cuenta la cantidad de accesos desde la clase medida a atributos de otras clases, ya sea directamente o por medio de métodos de acceso.

En el lenguaje Java, se obtiene el valor de métrica considerando el acceso a distintos atributos de otras clases desde la clase que se analiza. El acceso a los atributos de otras clases puede ser, directo en el caso que el atributo se encuentre definido como público, o a través de métodos de acceso (get's/set's) cuando el atributo se definió como privado.

La cantidad total de accesos a atributos de otras clases por alguno de los métodos descritos, será el valor final para la métrica ATFD.

En JavaScript, se puede considerar el acceso a los atributos de otros archivos js únicamente cuando los mismos son globales y se encuentran definidos como export. Para ser considerados globales, los atributos deben ser definidos con this (Sección 4.2.3 Análisis de las métricas en JavaScript).

Además, se debe considerar que el acceso a los atributos de otros archivos js por medio de métodos de acceso no se puede detectar de manera precisa en el lenguaje JavaScript (Sección 4.2.3 Análisis de las métricas en JavaScript), por lo cual no se podrá considerar el acceso a atributos de otros archivos js por medio de métodos de acceso.

Para poder utilizar los atributos de otras clases, JavaScript se basa en los conceptos de los términos export y require del entorno de ejecución Node js; este entorno, ejecuta código JavaScript haciendo uso de la Máquina virtual V8 de Google (el mismo entorno de ejecución para JavaScript que utiliza Google Chrome). Además, Node js implementa módulos basándose en la especificación de módulos de CommonJS [11]. Dichos módulos, se utilizan para organizar y crear la estructura de código del programa, y permiten definir límites en torno a la funcionalidad relacionada. Básicamente, CommonJS especifica que es necesario tener una función require para buscar dependencias, una variable exports para exportar contenido del módulo y un identificador de módulo (que describe la ubicación del módulo en cuestión) que se utiliza para requerir las dependencias [17].

Por lo tanto, el término export permite que se defina un módulo, en un archivo js, que podrá ser accedido desde otros archivos js por medio del uso de require.

Para el caso que se analiza, se considerará el uso de require para obtener el acceso a los atributos globales de otro archivo js. Dichos atributos se pueden acceder desde el archivo que se evalúa para obtener el valor de métrica ATFD, a partir del uso de export.

En el código de ejemplo de la Figura 4.8, se puede observar la definición de la variable Chart; la cual obtiene el valor de retorno del archivo js del que se hace el require. Además, al final del mismo, se puede ver que se utiliza el término export para indicar que la variable Chart podrá ser accedida desde otros archivos js.

La carga del módulo Chart se realiza mediante el uso de require; a partir del cual no sólo se carga el módulo sino que se obtiene el retorno del mismo. El valor de retorno que se obtiene, debido a que no se puede interactuar con el objeto directamente, generalmente se asigna a una variable, como sucede para este caso en particular.

Para acceder a los atributos globales de la clase Chart se deberá definir una variable que mediante el uso de new, cree una instancia del objeto como se muestra a continuación:

```
var Chart_obj = require('chart.js');  
var myChart = new Chart_obj({...});
```

Luego de la creación de la nueva instancia, se podrá acceder a los atributos globales de la misma a través del uso del nombre de la variable definida y el nombre del atributo al que se accede separados por punto (por ejemplo, mychart.NombreAtributo). En este caso en particular, se utiliza el punto entre ambos para indicar el acceso al valor del atributo.

Por lo tanto, a partir del análisis que se realizó para determinar si es posible usar la métrica ATFD de la estrategia de detección del code smell God Class en el lenguaje JavaScript, se concluye que es posible adaptar la métrica para su uso en el mismo. Sin embargo, se debe considerar para realizar la adaptación de la métrica, que se podrá contar la cantidad de atributos distintos de otros archivos js a los que se accede desde el archivo js que se analiza, pero únicamente cuando se accede directamente a los mismos. Se descarta de esta manera, la posibilidad de acceso a los atributos a través de métodos de acceso.

Debido a lo descrito anteriormente, el valor de métrica ATFD que se obtendrá para el lenguaje JavaScript, no será totalmente comparable al valor de métrica ATFD que se obtendría para el lenguaje Java.

Métrica WMC

La métrica WMC del code smell God Class, se centra en determinar si la clase posee una complejidad funcional alta.

Esta métrica se analizó para el code smell Data Class (Sección 4.2.3 Análisis de las métricas en JavaScript), donde se determinó cómo se obtiene su valor para el lenguaje Java y para el lenguaje JavaScript. Además, se mencionó el uso de la métrica CYCLO para su cálculo, la cual se analizó para el code smell Brain Method. En dicho análisis, se concluyó que no es necesario adaptar la métrica WMC para su uso en el lenguaje JavaScript.

Por lo tanto, a partir de lo descrito, y lo que se analizó previamente en los code smells mencionados, se puede concluir que no es necesario adaptar la métrica WMC para su utilización en la estrategia de detección del code smell God Class en el lenguaje JavaScript.

Métrica TCC

TCC (Cohesión de la clase) mide el número relativo de pares de métodos de una clase que acceden al menos a un atributo en común de la misma.

En Java para obtener el valor de TCC, se consideran los pares de métodos de la clase que acceden a un mismo atributo. Cuando esto ocurre se incrementa en uno el valor de la métrica. Una vez que se evaluaron cada uno de los pares de métodos de la clase, se obtiene el valor final para la métrica TCC.

El valor que se obtiene indica la cohesión existente entre los métodos de la clase; si este valor es menor a un cierto umbral se cumple la condición.

Para el análisis de la métrica en JavaScript, se deben considerar cada una de las funciones que se definen en el archivo js que representa la clase medida, sabiendo que en el lenguaje JavaScript no se presentan métodos sino que toda funcionalidad se define a través de la declaración de funciones. Se evalúa cada par de funciones para saber si acceden a un mismo atributo en común del archivo js; si es así, se modifica el valor de métrica aumentando el mismo. El valor final para la métrica TCC se obtendrá luego de evaluar cada par de funciones del archivo js, y se comparará con el umbral establecido para saber si se cumple la condición.

Por lo tanto, como no se presentan inconvenientes al definir funciones y atributos (propiedades) del archivo js, que representa una clase en el lenguaje JavaScript, se concluye que no es necesario adaptar la métrica TCC de la estrategia de detección del code smell God Class para su utilización en el mismo.

4.3.4 Conclusión Code Smell God Class

Luego de que se analizaran cada una de las métricas de la estrategia de detección del code smell God Class, para identificar si es posible su uso en el lenguaje JavaScript o si es

necesario adaptarlas, se concluye que es posible detectar el code smell en el lenguaje JavaScript a partir de las mismas.

Sin embargo se debe considerar que, como no se puede obtener la cantidad de accesos a atributos de otras clases a través de métodos de acceso en el lenguaje JavaScript, el valor final que se obtiene para la métrica ATFD no será igual que para Java.

4.4 Intensive Coupling

El code smell Intensive Coupling, se refiere a cuando una operación está intensamente acoplada a métodos de unas pocas clases (Figura 4.9).

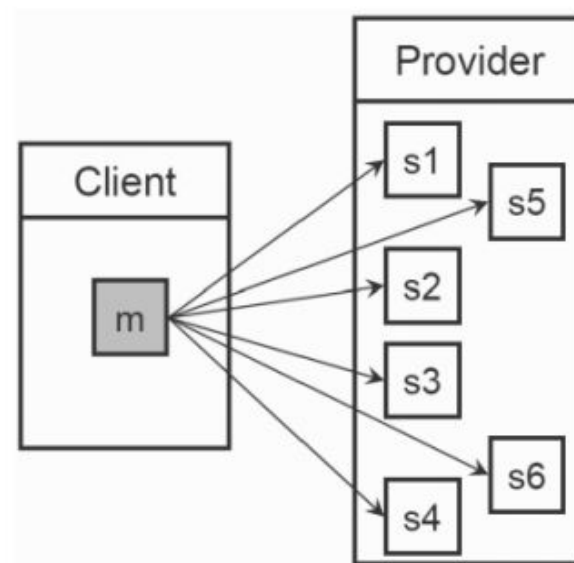


Figura 4.9: Ilustración del Code Smell Intensive Coupling.

4.4.1 Descripción

El code smell Intensive Coupling, se centra en el problema que se presenta cuando la comunicación entre una operación que se analiza, y las clases que le brindan servicios, es muy detallada; siendo estas clases una o muy pocas (Figura 4.9). Este tipo de relación provoca un acoplamiento intenso entre la operación que usa los servicios y las clases que brindan tales servicios.

En la mayoría de las ocasiones donde se presenta un acoplamiento intenso, las clases que proporcionan los muchos métodos invocados por la operación que se analiza no proporcionan un servicio en el nivel de abstracción requerido, es decir no presentan un servicio específico y entendible. En consecuencia, la comprensión de la relación entre las dos partes (es decir, el método del cliente y las clases que prestan servicios) se hace más difícil.

4.4.2 Detección

La estrategia de detección para el code smell Intensive Coupling descrita por Lanza y Marinescu, se encuentra definida por dos condiciones donde la primera se divide en cuatro condiciones más. En las condiciones que se presentan se definen tres métricas distintas, las cuales se utilizan para determinar si se cumplen o no dichas condiciones.

En la Figura 4.10.1, se muestran las condiciones que se deben cumplir para detectar el code smell Intensive Coupling. En la primer condición, se evalúa si el método llama a demasiados métodos de unas pocas clases no relacionadas, y en la segunda condición se evalúa si el método contiene pocos condicionales anidados. El término “clases no relacionadas” se refiere a clases que se encuentran en la misma jerarquía que la clase analizada.

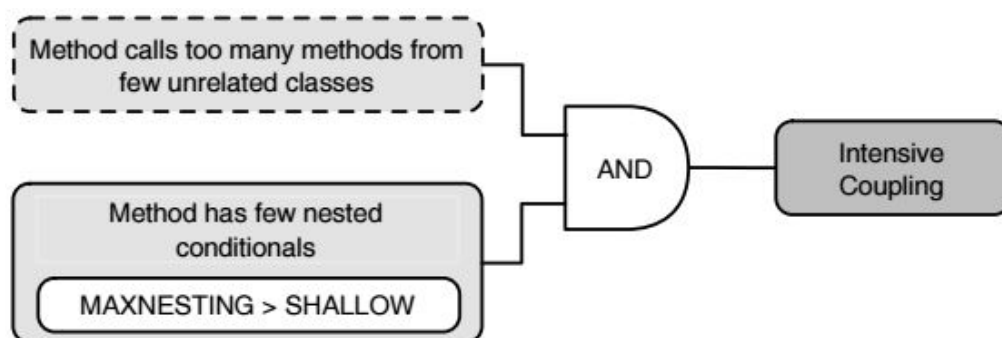


Figura 4.10.1: Estrategia de detección del Code Smell Intensive Coupling.

Para la segunda condición, se considera que una función que llama a muchos métodos, pero es “plana”, en términos del nivel de anidamiento de sus declaraciones, es menos compleja. En muchos casos, tales métodos son inicializadores (métodos constructores) o funciones de configuración que son más irrelevantes al momento de entender y mejorar la calidad de un diseño. Por lo tanto, se establece esta condición para que la función o método que se analiza contenga un nivel de anidamiento no trivial.

La métrica que se utiliza para evaluar la condición, es la métrica MAXNESTING (Máximo nivel de anidamiento) la cual obtiene el máximo nivel de anidamiento del método que se analiza. Esta métrica, se describió para la estrategia de detección del code smell Brian Method (Sección 4.1.2 Detección). En este caso en particular para que se cumpla la condición de la estrategia de detección, cuando se obtiene el valor de métrica, se compara si el mismo es mayor al límite que considera que los condicionales anidados son pocos.

La segunda condición que se detalla en la estrategia de detección del code smell Intensive Coupling, se define a partir de otras cuatro condiciones que se muestran en la Figura 4.10.2.

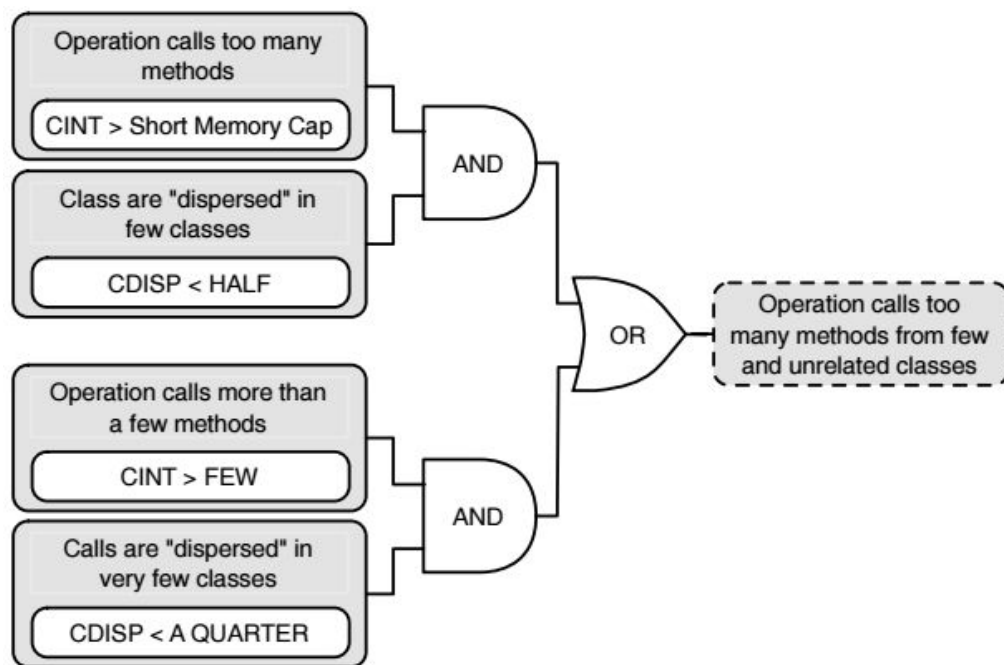


Figura 4.10.2: Estrategia de detección del Code Smell Intensive Coupling.

Las condiciones que se muestran en la Figura 4.10.2 se dividen en dos casos.

Para el primer caso se presentan dos condiciones, en la primera condición se controla si la operación llama a demasiados métodos de otras clases. Con demasiados se refiere a un número mayor que el número de elementos que pueden ser memorizados. La métrica que se utiliza para evaluar la condición es la métrica CINT (Intensidad de acoplamiento). Esta métrica, cuenta el número de operaciones distintas llamadas desde la operación medida. El valor de métrica que se obtiene se compara con un límite definido que se refiere a la capacidad de memoria, como ya se mencionó; si el valor es mayor al límite, se cumple la condición.

En la segunda condición, se evalúa si los métodos invocados por la operación tienen bajo grado de dispersión; esto quiere decir que los métodos invocados pertenecen a pocas clases. En promedio se invocan más de dos métodos de la misma clase proveedor.

La métrica para evaluar la condición, es la métrica CDISP (Dispersión de acoplamiento). Dicha métrica obtiene el número de clases en las que se definen las operaciones llamadas desde la operación medida, dividido por la métrica CINT. El valor que se obtiene para la métrica CDISP se evalúa para definir si la proporción de dispersión de los métodos invocados es inferior al 50%.

En el segundo caso que se muestra en la Figura 4.10.2, se presentan dos condiciones que son evaluadas a partir de las métricas descritas para el primer caso.

En la primer condición se considera la métrica CINT (Intensidad de acoplamiento) para evaluar si la operación llama a más de unos pocos métodos de otras clases; y en la segunda condición se evalúa si las clases a las que se accede son muy pocas. Para esta última condición se utiliza la métrica CDISP (Dispersión de acoplamiento).

Si se cumplen las condiciones de cada caso descrito se determina que la operación, la cual puede ser un método o una función independiente, llama a demasiados métodos de pocas clases no relacionadas. Además, si también se cumplen cada una de las condiciones que validan que el método contiene pocos condicionales anidados, se determina que se cumple la estrategia de detección que identifica al code smell Intensive Coupling.

4.4.3 Análisis de las métricas en JavaScript

En esta sección, se realiza el análisis de las tres métricas que se definen para la estrategia de detección del code smell Intensive Coupling, y se determina si es necesario adaptarlas para su utilización en el lenguaje JavaScript.

Para analizar la métrica en JavaScript, se tomará como referencia la Figura 4.8 en donde se encuentra el código de ejemplo que se utilizó para el code smell God Class, y la Figura 4.11, donde se muestra parte del código del archivo js “core.controller.js” del software Chart [10].

```
'use strict';

module.exports = function(Chart) {
  var helpers = Chart.helpers;
  var plugins = Chart.plugins;
  var platform = Chart.platform;

  Chart.types = {};
  Chart.instances = {};
  Chart.controllers = {};

  function initConfig(config) {
    config = config || {};
    var data = config.data = config.data || {};
    data.datasets = data.datasets || [];
    data.labels = data.labels || [];

    config.options = helpers.configMerge(
      Chart.defaults.global,
      Chart.defaults[config.type],
      config.options || {});
    return config;
  }

  function updateConfig(chart) {
    var newOptions = chart.options;
    if (newOptions.scale) {
```

```

        chart.scale.options = newOptions.scale;
    } else if (newOptions.scales) {
        newOptions.scales.xAxes.concat(newOptions.scales.yAxes).forEach(function(scaleOptions) {
            chart.scales[scaleOptions.id].options = scaleOptions;
        });
    }
    chart.tooltip._options = newOptions.tooltips;
}
function positionIsHorizontal(position) {
    return position === 'top' || position === 'bottom';
}
helpers.extend(Chart.prototype, /** @lends Chart */ {
    construct: function(item, config) {
        var me = this;
        config = initConfig(config);
        var context = platform.acquireContext(item, config);
        var canvas = context && context.canvas;
        var height = canvas && canvas.height;
        var width = canvas && canvas.width;

        me.id = helpers.uid();
        me.ctx = context;
        me.canvas = canvas;
        me.config = config;
        me.width = width;
        me.height = height;
        me.aspectRatio = height? width / height : null;
        me.options = config.options;
        me._bufferedRender = false;

        me.chart = me;
        me.controller = me;
        Chart.instances[me.id] = me;

        Object.defineProperty(me, 'data', {
            get: function() {
                return me.config.data;
            },
            set: function(value) {
                me.config.data = value;
            }
        });
        if (!context || !canvas) {
            console.error("Failed to create chart: can't acquire context from the given item");
            return;
        }
        me.initialize();
        me.update();
    },
    initialize: function() {
        var me = this;
        plugins.notify(me, 'beforeInit');
        helpers.retinaScale(me);
    }
});

```

```

    me.bindEvents();
    if (me.options.responsive) {
        me.resize(true);
    }
    me.ensureScalesHaveIDs();
    me.buildScales();
    me.initToolTip();
    plugins.notify(me, 'afterInit');
    return me;
},
...
});
Chart.Controller = Chart;
};

```

Figura 4.11: Archivo js en el lenguaje JavaScript. Código ejemplo Intensive Coupling.

Métrica MAXNESTING

La métrica MAXNESTING (Máximo nivel de anidamiento) ya se analizó para el code smell Brain Method. En dicho análisis, se describió cómo se obtiene su valor para el lenguaje Java y para el lenguaje JavaScript, y se llegó a la conclusión de que no es necesario adaptar la métrica para el lenguaje JavaScript.

Por lo tanto, se concluye que como ya se determinó que no es necesario adaptar la métrica para el lenguaje JavaScript en el code smell Brian Method, tampoco es necesario que sea adaptada para la estrategia de detección del code smell Intensive Coupling.

Métrica CINT

La métrica CINT se encarga de obtener la cantidad de operaciones distintas que se llaman desde la operación medida.

En el lenguaje Java, para obtener el valor de métrica CINT de la operación medida, se deben considerar las llamadas a métodos de otras clases. Dichas clases, deben ser clases no relacionadas a la clase que contiene la operación medida. El valor final para la métrica, será la cantidad total de operaciones distintas a las que se accede.

Para el análisis de la métrica en el lenguaje JavaScript se debe considerar que en el mismo no se define la herencia de clases; sólo se define el uso de archivos js desde otro a través del uso de export y require. Por lo tanto, no será posible detectar cuáles son las clases no relacionadas a la operación medida. Además, la operación sobre la cual se analiza el code smell Intensive Coupling podrá ser únicamente una función, debido a que en JavaScript no se definen métodos como tales; por lo tanto la función es la única unidad independiente que se presenta en el lenguaje.

De acuerdo a lo descrito para la métrica CINT de la estrategia de detección del code smell Intensive Coupling, se determina que es necesario adaptar la misma para su uso en el lenguaje JavaScript. Se debe considerar el uso de todos los archivos js debido a que no se pueden detectar “clases no relacionadas”. Además, se considerará a la función como única unidad independiente para obtener el valor de la métrica.

Por lo tanto, se debe analizar la función medida para determinar cuáles de las funciones que se acceden de la misma no pertenecen al archivo js en el cual se encuentra declarada. El valor final para la métrica CINT, será la cantidad total de funciones que no se encuentren definidas en el mismo archivo js que la función medida.

En la Figura 4.11, la cual se utiliza de ejemplo para el análisis de la métrica que se analiza, se muestra parte del código del archivo js “core.controller.js” del software Chart.js [10]. En el mismo se define la función initialize, la cuál será la función medida de ejemplo, y la que por lo tanto, se usará para obtener el valor de métrica CINT.

En dicha función, se realizan tres llamados a funciones que no pertenecen al archivo js donde se encuentra definida la función medida (initialize). Las llamadas son las siguientes:

1. `plugins.notify(me, 'beforeInit');`
2. `helpers.retinaScale(me);`
3. `plugins.notify(me, 'afterInit');`

Las demás llamadas a funciones que se realizan en la función initialize, se encuentran definidas en el archivo js “core.controller.js”, que es el mismo en el que se encuentra definida la función. Por lo tanto para la initialize, el valor de métrica CINT será igual a 3.

Métrica CDISP

La métrica CDISP obtiene el número de clases en las que se definen las operaciones llamadas desde la operación medida, dividido por la métrica CINT.

En Java para obtener el valor de la métrica CDISP, se evalúa una operación, que puede ser una función o un método, y se identifican las clases a las que pertenecen las funciones que se llaman desde la misma. La cantidad total de clases que se usan dividido el valor de la métrica CINT, será el valor final para la métrica CDISP.

Para obtener el valor de la métrica CDISP en JavaScript, se deben considerar las características descritas para el lenguaje en el análisis de la métrica CINT. En dicho análisis, se hace referencia a que no se define la herencia entre los archivos js, sino que se define el uso de archivos js en otros por medio de export y require. Además, se detalla que una

operación únicamente podrá ser una función, debido a que es la única unidad independiente que se presenta en el lenguaje JavaScript.

Por lo tanto, de acuerdo a lo descrito, se concluye que se debe adaptar la métrica CDISP que se presenta en la estrategia de detección del code smell Intensive Coupling para su uso en el lenguaje JavaScript. La cantidad de archivos js en los que se encuentran dispersas las funciones que se llaman desde la función medida, dividido el valor de métrica CINT, será el valor para la métrica CDISP en el lenguaje JavaScript.

En la Figura 4.11, se muestra parte del código del archivo js “core.controller.js” que se usará de ejemplo para el análisis de la métrica CDISP de la estrategia de detección del code smell Intensive Coupling. En dicho código, se encuentra definida la función initialize, la cual se analizó para obtener el valor de métrica la CINT, y la cual se analizará para obtener el valor de métrica la CDISP.

Para el análisis de la métrica CDISP se deben tener en cuenta las variables desde las que se realiza la llamada a las funciones que no pertenecen al mismo archivo js que la función medida. Las variables se pueden definir en el mismo archivo js que se define la función medida por medio de la palabra reservada var, y el uso de require para obtener el acceso a las funciones que se exportan desde el archivo js sobre el que se realiza el mismo; o por medio de una variable que se recibe por parámetro, la cual representa un archivo js como es el caso de la variable Chart para el archivo js de ejemplo.

La variable Chart del ejemplo, no sólo representa un archivo js sino que contiene funcionalidad de varios archivos js que también recibieron la variable por parámetro y “agregaron” funcionalidad a la misma a partir de la definición de nuevos objetos (Figura 4.8). Por lo tanto, cada variable distinta que se defina en el archivo js que contiene la función medida, y contenga la variable que se recibe por parámetro en el mismo, junto a la especificación del objeto particular al que se accede, se considerará como un acceso a un archivo js distinto.

Por ejemplo en las siguientes líneas, las cuales se muestran en la Figura 4.11, se define el acceso a tres archivos js distintos.

```
var helpers = Chart.helpers;  
var plugins = Chart.plugins;  
var platform = Chart.platform;
```

De los cuales se deben considerar a dos de ellos, ya que desde la función medida sólo se utilizan las variables helpers y plugins para acceder a funciones de otros archivos js. Las funciones que se llaman se especificaron en el análisis de la métrica CINT, las mismas eran tres llamadas a funciones de otros archivos js distintos al de la función medida. A partir de estas llamadas, se puede obtener el valor de la métrica CDISP, el cual será 2 dividido 3. El

primer valor hace referencia a la cantidad de archivos js distintos a los que se accede desde la función medida, y el segundo valor es la cantidad de funciones distintas a las que se accede desde la función medida y se calcularon para la métrica CINT, además dichas funciones se encuentran definidas en los archivos js que considerados para el primer valor.

4.4.4 Conclusión Code Smell Intensive Coupling

Luego de que se analizaran cada una de las métricas de la estrategia de detección del code smell Intensive Coupling, para identificar si es posible su uso en el lenguaje JavaScript o si es necesario adaptarlas, se determina que es necesario adaptar dos de las tres métricas que se presentan en la estrategia de detección. Luego de que se adapten cada una de las métricas mencionadas, es posible detectar el code smell Intensive Coupling para el lenguaje JavaScript.

4.5 Dispersed Coupling

El code smell Dispersed Coupling, se refiere al caso en el que una operación se encuentra ligada a muchas otras operaciones del sistema que se encuentran dispersas en muchas clases (Figura 4.12).

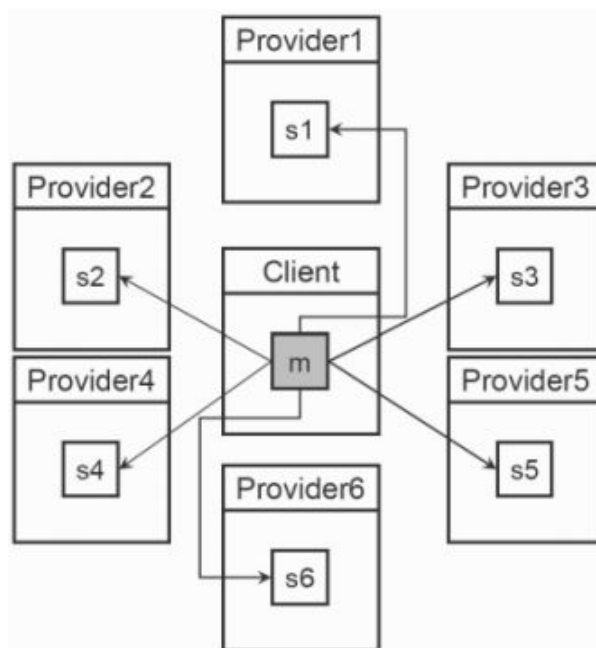


Figura 4.12: Ilustración del Code Smell Dispersed Coupling.

4.5.1 Descripción

Este code smell es lo complementario al code smell Intensive Coupling que se analizó anteriormente.

Se refiere al caso en el que una única operación se comunica con un número excesivo de clases proveedoras, por lo que la comunicación con cada una de las clases no es muy intensa, es decir, la operación llama a uno o unos pocos métodos de cada clase [1].

Las operaciones acopladas dispersamente conducen a efectos de ondulación no deseados, debido a que un cambio en un método acoplado dispersamente conduce potencialmente a cambios en todas las clases acopladas y, por lo tanto, dependientes.

4.5.2 Detección

La estrategia de detección para el code smell Dispersed Coupling descrita por Lanza y Marinescu, se encuentra definida por dos condiciones donde la primera se divide en dos condiciones más. Para que se cumplan las condiciones, se usan tres métricas: MAXNESTING, CINT y CDISP; estas métricas se usan en la estrategia de detección del code smell Intensive Coupling, con la diferencia de que para que se cumpla la estrategia de detección del code smell Dispersed Coupling, se consideran sólo aquellas operaciones que tienen una dispersión de acoplamiento alta.

En la Figura 4.13.1, se muestran las condiciones de la estrategia de detección del code smell Dispersed Coupling.

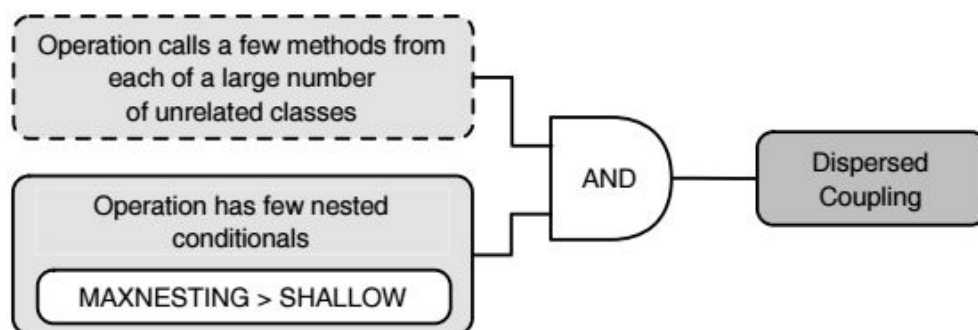


Figura 4.13.1: Estrategia de detección del Code Smell Dispersed Coupling.

La segunda condición de la estrategia de detección del code smell Dispersed Coupling, que se muestra en la Figura 4.13.1, es exactamente igual a la segunda condición de la estrategia de detección del code smell Intensive Coupling (Figura 4.10.1), en donde también se evalúa si la operación de llamada tiene un nivel de anidamiento no trivial, para asegurarse de que los casos irrelevantes (como las funciones de inicialización) se salteen.

La primer condición que se presenta para la estrategia de detección del code smell Intensive Coupling, donde se evalúa si la operación llama a unos pocos métodos de cada una de las muchas clases relacionadas, se encuentra definida por dos condiciones previas que se muestran en la Figura 4.13.2.

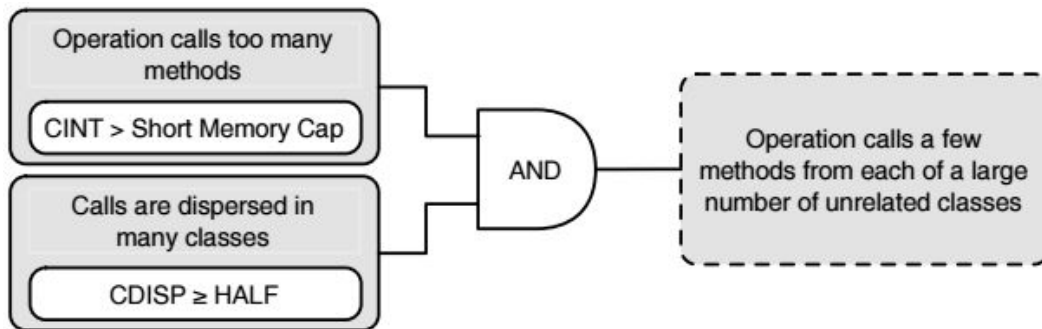


Figura 4.13.2: Estrategia de detección del Code Smell Dispersed Coupling.

En la primer condición de la Figura 4.13.2, se determina si la operación llama a muchos métodos de otras clases provocando de esta manera un acoplamiento intensivo; y en la segunda condición se evalúa si existe una gran dispersión entre las clases que contiene los métodos invocados.

Las métricas que se usan para cada una de las condiciones descritas, son las mismas que se definieron para la estrategia de detección del code smell Intensive Coupling.

4.5.3 Análisis de las métricas en JavaScript

En esta sección se realiza el análisis de las métricas de la estrategia de detección del code smell Dispersed Coupling, para definir si es posible su uso en el lenguaje JavaScript o si es necesario adaptarlas.

Como ya se mencionó, las métricas que se describen para la estrategia de detección del code smell Dispersed Coupling, también se utilizan en la estrategia de detección del code smell Intensive Coupling, para el cual ya se realizó el análisis de las mismas. En dicho análisis, se determinó que no es necesario adaptar la métrica MAXNESTING (Máximo nivel de anidamiento) para su uso en el lenguaje JavaScript; sin embargo, para el caso de las métricas CINT (Acoplamiento intensivo) y CDISP (Acoplamiento disperso), se determinó que es necesario adaptarlas.

4.5.4 Conclusión Code Smell Dispersed Coupling

Por lo tanto, se determina que se deben adaptar las métricas CINT y CDISP para que se cumpla la estrategia de detección del code smell Dispersed Coupling en el lenguaje JavaScript. Una vez que se adapten dichas métricas, es posible detectar el code smell para el lenguaje JavaScript. Como se puede ver, la resolución de este code smell se asemeja a a resolución del code smell Intensive Coupling, y claramente se puede deducir que se debe a que las métricas que se utilizan en las estrategias de detección de cada uno de los smells analizados, son las mismas.

4.6 Resumen del análisis de Code Smells

En la Tabla 4.1, se detalla por cada code smells que se analizó, si es posible o no detectarlos en el lenguaje JavaScript, y si es necesario adaptar alguna de las métricas.

Code Smell	Es posible detectar en JavaScript	Se deben adaptar métricas
Brain Method	SI	NO
Data Class	NO	-
God Class	SI	SI
Intensive Coupling	SI	SI
Dispersed Coupling	SI	SI

Tabla 4.1: Resumen del análisis de Code Smells.

Como se detalla en la Tabla 4.1, es posible detectar el code smell Brain Method en el lenguaje JavaScript, y no es necesario adaptar ninguna de sus métricas, debido a que las mismas se detectan de igual manera en ambos lenguajes, Java y JavaScript. En el caso del code smell Data Class, debido a las características que se consideran en el lenguaje Java para detectarlo, se concluyó que no es posible identificar dichas características en el lenguaje JavaScript, y que por lo tanto no es posible detectar el code smell en dicho lenguaje. Para los code smells God Class, Intensive Coupling y Dispersed Coupling, se define que es posible detectarlos en JavaScript, pero que es necesario adaptar sus métricas.

Al finalizar el análisis de cada code smell, se determinó que métricas se puede definir en el lenguaje JavaScript, cuáles no, y cuales de las que se pueden definir se deben adaptar (Tabla 4.2).

Métrica	Es posible definir en JavaScript	Se debe adaptar
LOC	SI	NO
CYCLO	SI	NO
MAXNESTING	SI	NO
NOAV	SI	NO
WOC	SI	SI
NOAP	SI	SI
NOAM	NO	-
WMC	SI	NO
ATFD	SI	SI
TCC	SI	NO
CINT	SI	SI
CDISP	SI	SI

Tabla 4.2: Resumen del análisis de métricas.

Como se puede ver en la Tabla 4.2, es posible definir en el lenguaje JavaScript casi la totalidad de las métricas que se utilizan en las estrategias de detección de los code smells que se analizaron.

La única métrica que no se puede definir en el lenguaje JavaScript es la métrica NOAM (Número de métodos de acceso - get's/set's - de una clase) que pertenece a la estrategia de detección del code smell Data Class, y se relaciona con los métodos de acceso get y set que se definen en el lenguaje Java, y que aún no se reconocen en todos los entornos de desarrollo del lenguaje JavaScript, como ya se mencionó anteriormente.

De las demás métricas que se muestran en la Tabla 4.2 y se pueden definir en el lenguaje JavaScript, hay un porcentaje similar de métricas que se deben adaptar con las que no. Esto quiere decir que hay características similares en el lenguaje Java y en el lenguaje JavaScript por lo que el trabajo de adaptación de métricas de un lenguaje a otro no es tan tedioso y es posible.

Por lo tanto, la definición de un catálogo de métricas en el lenguaje JavaScript en relación al catálogo de métricas definido por Lanza y Marinescu [1] para el lenguaje Java, es posible.

Capítulo 5

Casos de estudio

Con el objetivo de validar el análisis que se realizó para cada uno de los code smells presentes en el Capítulo 4, y para realizar el estudio de los diferentes casos, se implementa la herramienta vcomplex, y se analizan los resultados de su ejecución en tres aplicaciones diferentes.

5.1 Herramienta vcomplex

Vcomplex, realiza el análisis de complejidad de software desarrollado en lenguaje JavaScript a partir del informe de valores para diferentes métricas. En dicho análisis se evalúa la sintaxis del código de los archivos js que recibe la herramienta como entrada, para obtener, como salida, el valor de las métricas que son parte de las estrategias de detección de los code smells que se analizaron en el Capítulo 4. La evaluación de sintaxis, se realiza a partir de la interpretación de árboles de sintaxis que se obtienen con el analizador de ECMAScript, Esprima [18].

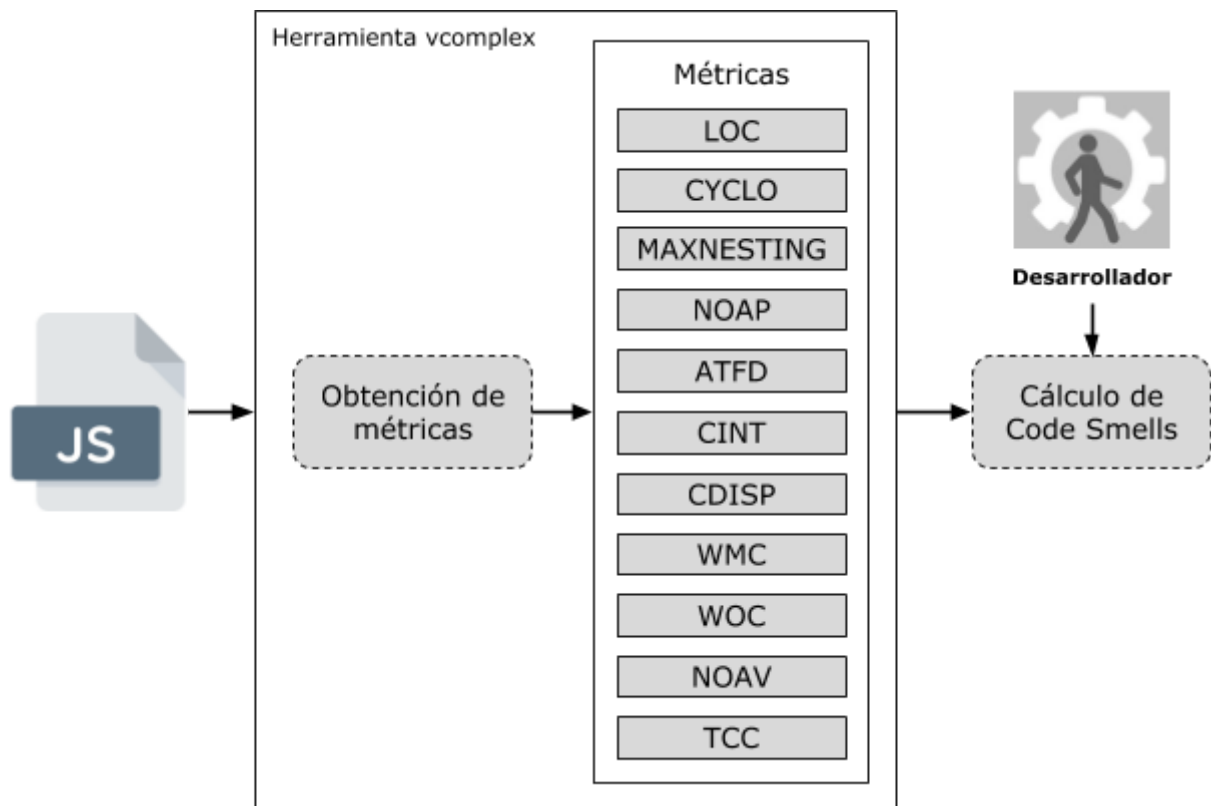


Figura 5.1: Obtención de métricas con vcomplex, y cálculo de Code Smells.

Para la implementación de la herramienta, se utilizó el entorno de desarrollo Eclipse Java EE IDE for Web Developers, versión Neon.3 Release 4.6.3. Luego de instalar el entorno, se

ejecuta por consola el script “npm install escomplex” para obtener las carpetas relacionadas a la librería que se utilizó para la implementación [13]. La versión de escomplex que se utilizó para este trabajo es la 2.0.0-alpha.

Como se muestra en la Figura 5.1, la herramienta vcomplex, tiene como entrada el archivo js que se analiza, y como salida el valor de las métricas que se obtienen al analizar el archivo de entrada. Una vez que se obtiene el valor de las métricas, se realiza el cálculo de los code smells. Para ello, se reemplaza el valor de las métricas en la estrategia de detección de cada una de los code smells que se deseen evaluar. Si se cumple la estrategia, se concluye que se está en presencia de un posible code smell en el archivo js que se analiza. La tarea del desarrollador será la de validar si verdaderamente se trata de un code smell en JavaScript.

Para obtener los valores de las métricas y detectar un problema en el código a partir de los mismos, se seleccionaron diferentes aplicaciones desarrolladas en el lenguaje JavaScript. Dichas aplicaciones se obtienen del top ten de paquetes más descargados desde el gestor de paquetes de JavaScript npm (Node Package Manager), el cual se caracteriza por ayudar a las personas que lo utilizan a administrar, distribuir y reutilizar paquetes, y a agregar dependencias de manera sencilla [11]. Desde la página de npm se presenta, para cada aplicación, un enlace al código que se encuentra en GitHub desde donde se puede realizar la descarga del mismo.

5.2 Descripción de los casos de estudio

Los casos de estudio son descritos de la siguiente manera: se hace una presentación del mismo, se realiza el análisis para la detección de problemas en el código que puedan llegar a ser definidos como un code smell, se detallan las métricas que son parte de la estrategia de detección del mismo y se describen los resultados obtenidos para determinar si el problema hallado es verdaderamente un code smell.

Las métricas se deben interpretar de manera adecuada, por lo que se deben establecer umbrales para cada una de las mismas determinando el límite en el cual es válida. Además, como las métricas por sí solas no se consideran suficientes para comprender y evaluar un sistema de software, se deben considerar las estrategias de detección que se presentaron en el Capítulo 4 para cada uno de los code smells que se analizaron. A partir de la correlación de las métricas presentes para cada estrategia de detección, se detectan los problemas presentes en el código que se analiza. Una vez que se detectan los problemas, code smells, se inspecciona el código fuente del code smell para determinar si el mismo es realmente un problema.

Los umbrales que se utilizan para las métricas que son parte de las estrategias de detección, se obtienen del análisis que realiza Lanza y Marinescu en donde se presentan las métricas junto a los valores definidos para cada uno de los mismos, o valores para umbrales que se utilizan

para más de una métrica de acuerdo a la condición de la estrategia de detección en la que se encuentran. Dichos valores, se obtuvieron desde una recolección de datos de una base estadística de 45 proyectos Java que se seleccionaron teniendo en cuenta su diversidad [1].

Se decidió utilizar los mismos umbrales que en Java, debido a que, al realizar el análisis de las métricas definidas en el lenguaje Java para determinar si era posible que se utilicen en el lenguaje JavaScript, se pudo percibir que para las métricas que se podían usar sin ser adaptadas, o las que se debían adaptar, la manera en la que se calculaban era similar en ambos lenguajes. Por ejemplo, el anidamiento es igual en ambos lenguajes, las líneas de código se consideran de la misma manera, entre otras. Sin embargo, queda pendiente para futuros trabajos de investigación, el cálculo de los umbrales específicos para el lenguaje JavaScript.

En la Tabla 5.1, se muestran los umbrales para la métrica CYCLO (Complejidad ciclomática), la métrica WMC (Complejidad de la clase) y, la métrica LOC (Líneas de código) de una clase y de un método.

Metric	Java			
	Low	Ave- rage	High	Very High
CYCLO/Line of Code	0.16	0.20	0.24	0.36
LOC/Method	7	10	13	19.5
WMC	5	14	31	47
LOC/Class	28	70	130	195

Tabla 5.1: Umbrales para las métricas CYCLO, WMC y LOC.

En el caso de la métrica MAXNESTING (Máximo nivel de anidamiento), el valor 0 (cero) se refiere a un método que no contiene ningún nivel de anidamiento, 1, 2 ó 3 significa que hay anidación en el método pero es dentro de lo normal y si el método tiene un nivel de anidamiento mayor a 3, se considera que el nivel de anidamiento del método es alto y, por lo tanto, el flujo de control se vuelve difícil [1].

Cuando se tengan valores normalizados para las métricas, como es el caso de la métrica TCC (Cohesión de la clase), se deben considerar los valores que se muestran en la Tabla 5.2; en donde se muestran los umbrales para las métricas normalizadas, y a partir de los cuales se puede obtener una semántica adecuada del valor de las mismas.

Por ejemplo, para la métrica TCC, cuanto menor sea el valor de la misma, menos cohesiva es la clase. Si se buscan encontrar clases cohesivas debe elegirse un valor inferior a la mitad (half), es decir un tercio (one-third) o un cuarto (one-quarter) [1]. Para los casos que se analizarán, se tendrá en cuenta como valor a comparar para el umbral de la métrica TCC, la cantidad total de pares de métodos que posee el archivo js para luego obtener el tercio de dicha cantidad.

Numeric Value	Semantic Label
0.25	ONE-QUARTER
0.33	ONE-THIRD
0.5	HALF
0.66	TWO-THIRDS
0.75	THREE-QUARTERS

Tabla 5.2: Umbrales para las métricas normalizadas.

Además de presentar valores para las métricas normalizadas, se asocian valores absolutos para los cuales se consideran valores entre 0 y 7 (Tabla 5.3); este último es el límite superior de la memoria a corto plazo humana [14].

Numeric Value	Semantic Label
0	NONE
1	ONE/SHALLOW
2 – 5	TWO, THREE/FEW/ SEVERAL
7 – 8	Short Memory Capacity

Tabla 5.3: Umbrales para métricas con valores absolutos.

Por lo tanto, a partir de los valores presentados para los umbrales, se muestran los valores definidos para las métricas de cada Estrategia de Detección en relación a los code smells que se pueden detectar en el lenguaje JavaScript (Tabla 5.4, 5.5, 5.6 y 5.7). En el caso de los valores de las condiciones que se muestran en la Tabla 5.6, para el cumplimiento de la estrategia de detección del code smell Intensive Coupling, se deben cumplir las condiciones 2 y 3 y/o las condiciones 4 y 5.

Brain Method

Número de condición	Condición
1	LOC > HIGH(Method)
2	CYCLO >= HIGH
3	MAXNESTING >= SEVERAL
4	NOAV > Short Memory Capacity

Tabla 5.4: Condiciones con métricas y umbrales para el Code Smell Brain Method.

God Class

Número de condición	Condición
1	ATFD > SEVERAL
2	WMC >= VERY HIGH
3	TCC < ONE THIRD

Tabla 5.5: Condiciones con métricas y umbrales para el Code Smell God Class.

Intensive Coupling

Número de condición	Condición
1	MAXNESTING > SHALLOW

2	CINT > Short Memory Capacity
3	CDISP < HALF
4	CINT > FEW
5	CDISP < A QUARTER

Tabla 5.6: Condiciones con métricas y umbrales para el Code Smell Intensive Coupling.

Dispersed Coupling

Número de condición	Condición
1	MAXNESTING > SHALLOW
2	CINT > Short Memory Capacity
3	CDISP >= HALF

Tabla 5.7: Condiciones con métricas y umbrales para el Code Smell Dispersed Coupling.

A partir de los umbrales que se definen para las métricas, las condiciones presentes en las estrategias de detección, y el relevamiento de métricas que se obtienen al ejecutar la extensión la herramienta vcomplex, se detectan diferentes code smells en las aplicaciones que se utilizan como casos de estudio. Además, para analizar los resultados, se inspecciona el código correspondiente a la detección del code smell, y se define si es en realidad un problema de diseño estructural para la aplicación.

5.3 Aplicación #1: Browserify

En esta primera evaluación, se analiza la aplicación Browserify [12]. Dicha aplicación es una herramienta de código libre que permite crear módulos en el cliente, utilizando la misma sintaxis que en Node (CommonJS); permitiendo requerir y exportar módulos, y manejar sus dependencias como en Node pero en el browser [12].

5.3.1 Detección de Code Smells y análisis de métricas

Los Code Smells que se hallaron para esta aplicación en particular fueron tres:

1. Brain Method.
2. Intensive Coupling.
3. Dispersed Coupling.

Para el primer code smell, se presentan quince ocurrencias en el código de la aplicación, para el segundo tres ocurrencias y para el último, respectivamente, se hallaron tres ocurrencias (Figura 5.2).

5.3.2 Brain Method

Como se puede ver en la Figura 5.2, el code smell que predomina en el código de la aplicación es Brain Method, esto se debe a que dicho smell es uno de los más frecuentes [1].

Esto quiere decir que Brain Method es el code smell del cual se obtienen mayores ocurrencias en los sistemas que se analizan, siendo de esta manera, uno de los problemas que más impacta sobre la mantenibilidad del sistema.

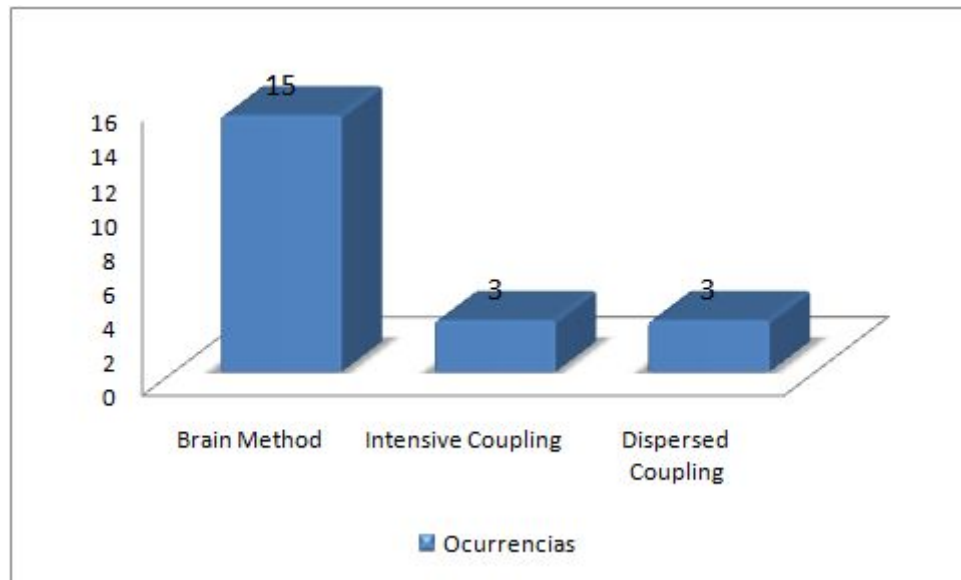


Figura 5.2: Ocurrencias de Code Smells en la aplicación Browserify.

En la Tabla 5.8, se muestran los valores de las métricas por las cuales se obtuvieron los code smells Brain Method en el código de la aplicación. En la columna Nombre de la función, se muestra el nombre de la función que se evaluó y, entre paréntesis, se detalla el archivo js al cual pertenece la misma.

Nombre de la función	LOC	CYCLO	MAXNESTING	NOAV
globalTr (index.js)	39	12	4	11
module.exports (bin/args.js)	245	15	4	41
<anonymous> - línea 134 (index.js)	24	11	2	14
<anonymous>.external (index.js)	57	4	2	19
<anonymous>.transform (index.js)	59	9	3	16
<anonymous>.plugin (index.js)	20	5	3	8
<anonymous>._createPipeline (index.js)	57	3	2	23
<anonymous>._createDeps (index.js)	151	6	8	38
<anonymous> - línea 464 (index.js)	32	11	4	12
Browserify (index.js)	66	11	2	19
<anonymous> - línea 678 (index.js)	48	9	2	17
<anonymous> - línea 692 (index.js)	27	7	3	12
<anonymous>.bundle (index.js)	34	5	2	11
<anonymous>.require (index.js)	91	19	3	21
<anonymous> - línea 171 (bin/args.js)	20	3	3	9

Tabla 5.8: Valores de métricas de las ocurrencias del Code Smell Brain Method en la aplicación Browserify.

Por ejemplo, para la función `globalTr` (Figura 5.3) que se referencia en la Tabla 5.8, se considera el valor 39 para la métrica LOC, 12 para la métrica CYCLO, 4 para la métrica MAXNESTING y 11 para la métrica NOAV. Estos valores son lo que se evalúan en las condiciones de la estrategia de detección del code smell Brain Method.

1. 39 (LOC) > 13
2. 12 (CYCLO) >= 2.4
3. 4 (MAXNESTING) >= 2 - 5
4. 11 (NOAV) > 7 - 8

Como dichos valores, al ser comparados con los umbrales definidos para cada una de las métricas, cumplen con las cuatro condiciones necesarias para que se valide la estrategia de detección, se considera que la función `globalTr` es un posible Brain Method en la aplicación Browserify.

```
function globalTr (file) { MAXNESTING = 1 CYCLO = 1
  if (opts.detectGlobals === false) return through(); CYCLO = 2

  if (opts.noParse === true) return through(); CYCLO = 3
  if (no.indexOf(file) >= 0) return through(); CYCLO = 4
  if (absno.indexOf(file) >= 0) return through(); CYCLO = 5

  var parts = file.split('/node_modules/');
  for (var i = 0; i < no.length; i++) { MAXNESTING = 2 CYCLO = 6 - 7
    if (typeof no[i] === 'function' && no[i](file)) { CYCLO = 8 - 9
      return through();
    }
    else if (no[i] === parts[parts.length-1].split('/')[0]) { MAXNESTING = 3 - 4 CYCLO = 10
      return through();
    }
    else if (no[i] === parts[parts.length-1]) { CYCLO = 11
      return through();
    }
  }
}

var vars = xtend({
  process: function () { return "require('_process')"},
}, opts.insertGlobalVars);

if (opts.bundleExternal === false) { CYCLO = 12
  vars.process = undefined;
  vars.buffer = undefined;
}

return insertGlobals(file, xtend(opts, {
  debug: opts.debug,
  always: opts.insertGlobals,
  basedir: opts.commondir === false
    ? '/'
    : opts.basedir || process.cwd()
  },
  vars: vars
));
}
```

Figura 5.3: Función `globalTr`.

En relación al valor de la métrica MAXNESTING, se puede observar que no es alto, y no representa un gran problema para entender el código. Sin embargo, a medida que pasa el

tiempo el desarrollador puede perder claridad ante lo que representa el anidamiento de las condiciones, y de qué es lo que se pretende evaluar del arreglo sobre el cual se calculan las mismas. Por lo cual se volvería necesario comentar el código indicando qué es lo que se evalúa en relación al arreglo.

Para la métrica CYCLO, se considera que hay una serie de condiciones al comienzo de la función `globalTr` que podrían afectar a la facilidad de lectura y entendimiento de la misma, debido a que no está claro cuál es la funcionalidad que representa cada una. Por otro lado, la métrica LOC, informa que la función `globalTr` es extensa lo cual hace que no sea claro el objetivo de la misma; y el valor de la métrica NOAV, es mayor a la capacidad que posee una persona para recordarlas, lo cual es un indicio a riesgos en el futuro.

Por lo tanto, si se consideran las características que se mencionaron de la función, y se tiene en cuenta que el archivo que contiene la misma cuenta con 807 líneas de código, se puede deducir que el método no se considera un método largo en relación al archivo, y que tampoco centraliza la funcionalidad del mismo. Con lo cual se concluye que la función `globalTr`, no es un Brain Method y es un falso positivo.

Otro ejemplo a considerar para el code smell Brain Method, es la función `module.exports` del archivo `args.js` (no se muestra el código de la misma por ser demasiado extenso).

Si se evalúan los valores de métricas, que se obtuvieron al ejecutar la herramienta, en la estrategia de detección, la misma se cumple indicando que `module.exports` es un posible Brain Method.

1. 245 (LOC) > **13**
2. 15 (CYCLO) >= **2.4**
3. 2 (MAXNESTING) >= **2 - 5**
4. 41 (NOAV) > **7 - 8**

Al realizar un análisis del código de la función, se observa que el anidamiento condicional que contiene no es significativo, lo cual hace que la misma no sea difícil de leer e interpretar condicionalmente. Sin embargo, engloba casi la totalidad del archivo que la contiene, y exporta dicha funcionalidad para que se pueda usar desde otro archivo que la requiera.

El archivo `args.js` contiene 270 líneas de código de las cuales 245 pertenecen a la función `module.exports`, más del 90% del total. Por lo tanto, considerando que si se reemplazan los valores de las métricas en la estrategia de detección del code smell que se analiza, esta se cumple; y además, que al realizar el análisis del código, la función `module.exports` centraliza casi la totalidad de la funcionalidad de la clase, se concluye que la misma es un Brain Method.

5.3.3 Intensive Coupling

En la Tabla 5.9, se muestran las funciones que se detectaron en la aplicación como posibles code smells Intensive Coupling.

Nombre de la función	MAXNESTING	CINT	CDISP
globalTr (index.js)	4	10	0.3
<anonymous>.require (index.js)	3	11	0.4545
next (test/externalize.js)	2	10	0.4

Tabla 5.9: Valores de métricas de las ocurrencias del Code Smell Intensive Coupling en la aplicación Browserify.

Se toma como ejemplo la función globalTr (Figura 5.3), para la cual la métrica MAXNESTING tiene como valor 4, la métrica CINT 10 y CDISP es igual a 0.3.

1. 4 (MAXNESTING) > 1
2. 10 (CINT) > 7 - 8
3. 0.3 (CDISP) < 0.5
4. 10 (CINT) > 2 - 5
5. 0.3 (CDISP) < 0.25

```
var mdeps = require('module-deps');
var depsSort = require('deps-sort');
var bpack = require('browser-pack');
var insertGlobals = require('insert-module-globals');
var syntaxError = require('syntax-error');

var builtins = require('./lib/builtins.js');

var splicer = require('labeled-stream-splicer');
var through = require('through2');
var concat = require('concat-stream');

var inherits = require('inherits');
var EventEmitter = require('events').EventEmitter;
var xtend = require('xtend');
var isArray = Array.isArray;
var defined = require('defined');
var has = require('has');
var sanitize = require('html-escape').sanitize;
var shasum = require('shasum');

var bresolve = require('browser-resolve');
var resolve = require('resolve');

var readonly = require('read-only-stream');

module.exports = Browserify;
inherits(Browserify, EventEmitter);

var fs = require('fs');
var path = require('path');
var relativePath = require('cached-path-relative')
```

Figura 5.4: Archivos que se usan desde la función globalTr del archivo index.js

Para que se cumpla la estrategia de detección del code smell Intensive Coupling basta con que se cumplan las condiciones 1, 2 y 3 ó las condiciones 1, 4 y 5; en este caso en particular, se cumplen las condiciones 1, 2 y 3. Por lo tanto, al cumplirse la estrategia de detección, se concluye que la función `globalTr` es un posible code smell Intensive Coupling en la aplicación Browserify.

Si se observa la Figura 5.4, se puede observar que se encuentran destacados los archivos js de los cuales se realizan llamadas a alguna, o varias, de sus funciones. Del total de los archivos js que se requieren, se usan funciones de menos del 14% de los mismos, lo cual es poco, e indica que las llamadas a funciones de otros archivos js, desde la función `globalTr`, se encuentran centralizadas en muy pocos archivos js. Entonces, se concluye que la función `globalTr` llama a demasiadas funciones de pocos archivos relacionados, lo cual representa un code smell Intensive Coupling en la aplicación Browserify.

5.3.4 Dispersed Coupling

En la Tabla 5.10, se presentan las dos funciones que se detectaron como posibles code smell Dispersed Coupling en la aplicación. Para las mismas, se consideran los valores de las métricas, que se obtuvieron al ejecutar la extensión de escomplex, comparados con los umbrales definidos en las tres condiciones que se presentan en la estrategia de detección del code smell Dispersed Coupling.

Nombre de la función	MAXNESTING	CINT	CDISP
<anonymous>._createPipeline (index.js)	2	8	0.875
<anonymous>._createDeps (index.js)	8	20	0.5
module.exports (bin/args.js)	2	12	0.8333

Tabla 5.10: Valores de métricas de las ocurrencias del Code Smell Dispersed Coupling en la aplicación Browserify.

Se toma como ejemplo la función `_createPipeline` (Figura 5.6), que se encuentra en el archivo `index.js` de la aplicación Browserify. Los valores de métricas que se obtuvieron para la misma fueron, 2 para la métrica MAXNESTING, 8 para CINT y 0.875 para CDISP.

1. $2 \text{ (MAXNESTING)} > 1$
2. $8 \text{ (CINT)} > 7 - 8$
3. $0.875 \text{ (CDISP)} \geq 0.5$

Al cumplirse las tres condiciones que se muestran para la estrategia de detección del code smell que se analiza, se concluye que la función `_createPipeline` del archivo `index.js`, es un posible code smell Dispersed Coupling para la aplicación Browserify.

En la Figura 5.6, se puede observar el valor de la métrica MAXNESTING a partir de las condiciones resaltadas. El valor de métrica no es alto, por lo cual se considera que la función

que se evalúa tiene pocos condicionales anidados lo cual es suficiente en la estrategia de detección que se evalúa. Para este caso en particular, no se considera el valor MAXNESTING de la función anónima que es parte de `_createPipeline`, ya que se ejecuta como una unidad individual para una condición en particular, y es parte de los parámetros de otra llamada. Si se desea considerar, se debe sumar el valor MAXNESTING de la misma, que se calcula en la herramienta vcomplex.

```
var mdeps = require('module-deps');
var depsSort = require('deps-sort');
var bpack = require('browser-pack');
var insertGlobals = require('insert-module-globals');
var syntaxError = require('syntax-error');

var builtins = require('./lib/builtins.js');

var splicer = require('labeled-stream-splicer');
var through = require('through2');
var concat = require('concat-stream');

var inherits = require('inherits');
var EventEmitter = require('events').EventEmitter;
var xtend = require('xtend');
var isArray = Array.isArray;
var defined = require('defined');
var has = require('has');
var sanitize = require('html-escape').sanitize;
var shasum = require('shasum');

var bresolve = require('browser-resolve');
var resolve = require('resolve');

var readonly = require('read-only-stream');

module.exports = Browserify;
inherits(Browserify, EventEmitter);

var fs = require('fs');
var path = require('path');
var relativePath = require('cached-path-relative');
```

Figura 5.5: Archivos js que se usan desde la función `_createPipeline` del archivo `index.js`.

Para la métrica CINT de la función `_createPipeline`, se obtuvo el valor 8, que indica que se está en el límite de cantidad de nombres de funciones que la memoria de una persona puede recordar, siendo un problema a futuro para el mantenimiento de la aplicación. Además, considerando el valor de la métrica CDISP, se puede percibir que, las funciones que se usan desde `_createPipeline` y pertenecen a otros archivos js, se encuentran dispersas en siete del total de archivos que se requieren en el archivo que contiene la función (Figura 5.5). Esto indica que la dispersión es alta, casi una función por archivo, y que si se realiza un cambio en la función se deberán revisar las llamadas a los siete archivos js, con probabilidad de que también se deban hacer cambios.

Por lo tanto, a partir de lo descrito, se puede afirmar que la función `_createPipeline` representa un code smell Dispersed Coupling en la aplicación Browserify.

```

Browserify.prototype._createPipeline = function (opts) { MAXNESTING = 1
  var self = this;
  if (!opts) opts = {};
  this._mdeps = this._createDeps(opts);
  this._mdeps.on('file', function (file, id) {
    pipeline.emit('file', file, id);
    self.emit('file', file, id);
  });
  this._mdeps.on('package', function (pkg) {
    pipeline.emit('package', pkg);
    self.emit('package', pkg);
  });
  this._mdeps.on('transform', function (tr, file) {
    pipeline.emit('transform', tr, file);
    self.emit('transform', tr, file);
  });

  var dopts = {
    index: !opts.fullPaths && !opts.exposeAll,
    dedupe: opts.dedupe,
    expose: this._expose
  };
  this._bpack = bpack(xtend(opts, { raw: true })); CINT = 1 - 2

  var pipeline = splicer.obj([ CINT = 3
    'record', [ this._recorder() ],
    'deps', [ this._mdeps ],
    'json', [ this._json() ],
    'unbom', [ this._unbom() ],
    'unshebang', [ this._unshebang() ],
    'syntax', [ this._syntax() ],
    'sort', [ depsSort(dopts) ], CINT = 4
    'dedupe', [ this._dedupe() ],
    'label', [ this._label(opts) ],
    'emit-deps', [ this._emitDeps() ],
    'debug', [ this._debug(opts) ],
    'pack', [ this._bpack ],
    'wrap', [ ]
  ]);
  if (opts.exposeAll) { MAXNESTING = 2
    var basedir = defined(opts.basedir, process.cwd()); CINT = 5
    pipeline.get('deps').push(through.obj(function (row, enc, next) { CINT = 6
      if (self._external.indexOf(row.id) >= 0) return next();
      if (self._external.indexOf(row.file) >= 0) return next();

      if (isAbsolutePath(row.id)) {
        row.id = '/' + relativePath(basedir, row.file); CINT = 7
      }
      Object.keys(row.deps || {}).forEach(function (key) {
        row.deps[key] = '/' + relativePath(basedir, row.deps[key]); CINT = 8
      });
      this.push(row);
      next();
    }));
  }
  return pipeline;
};

```

Figura 5.6: Función _createPipeline.

5.4 Aplicación #2: Gulp

En esta segunda evaluación, se analiza la aplicación Gulp [19]. Gulp, es un conjunto de herramientas que ayuda a automatizar tareas tediosas o que requieren mucho tiempo en el flujo de trabajo normal de desarrollo. Automatiza y mejora el flujo de trabajo. Se integra a los principales IDEs, y se utiliza actualmente con PHP, .NET, Node.js, Java, entre otras plataformas.

5.4.1 Detección de Code Smells y análisis de métricas

Los code smells que se hallaron para esta aplicación en particular fueron dos, Brain Method e Intensive Coupling. Para ambos code smells, se halló una única ocurrencia en el código de la aplicación. Además, la función involucrada en dichos code smells es la misma, `handleArguments`, que se encuentra en el archivo `gulp.js` de la carpeta `bin` de la aplicación.

5.4.2 Brain Method

En la Tabla 5.11, se muestran los valores de las métricas por las cuales se obtuvo el code smell Brain Method en el código de la aplicación.

Nombre de la función	LOC	CYCLO	MAXNESTING	NOAV
<code>handleArguments (bin/gulp.js)</code>	57	7	3	12

Tabla 5.11: Valores de métricas de la ocurrencia del Code Smell Brain Method en la aplicación Gulp.

Para la función `handleArguments` (Figura 5.7), se considera el valor 57 para la métrica LOC, 7 para la métrica CYCLO, 3 para la métrica MAXNESTING y 12 para la métrica NOAV. Estos valores son lo que se evalúan en las condiciones de la estrategia de detección del code smell Brain Method.

1. $57 (LOC) > 13$
2. $7 (CYCLO) \geq 2.4$
3. $3 (MAXNESTING) \geq 2 - 5$
4. $12 (NOAV) > 7 - 8$

Como dichos valores, al ser comparados con los umbrales definidos para cada una de las métricas, cumplen con las cuatro condiciones necesarias para que se valide la estrategia de detección, se considera que la función `handleArguments` es un posible Brain Method en la aplicación Gulp.

Para evaluar si la función representa o no un Brain Method en aplicación, se analizó en la funcionalidad de la misma y su impacto en el archivo al que pertenece. El archivo `gulp.js` que contiene la función `handleArguments`, cuenta con 218 líneas de código, y 17 funciones en las cuales se distribuye la funcionalidad que representa el mismo. Si las funciones fueran similares en cuanto a líneas de código, y por lo tanto funcionalidad, cada una debería tener en promedio 13 líneas de código. Sin embargo, la función que se analiza cuadruplica, y más, dicho promedio, ocupando casi el 30% del total de líneas del archivo. Además, `handleArguments`, es la función con más líneas de código en `gulp.js`. La siguiente función más extensa, en el archivo, tiene 40 líneas de código, las demás tienen en promedio 7 líneas de código.

```

function handleArguments(env) { MAXNESTING=1 CYCLO=1
  if (versionFlag && tasks.length === 0) { MAXNESTING=2 CYCLO=2
    gutil.log('CLI version', cliPackage.version);
    if (env.modulePackage && typeof env.modulePackage.version !== 'undefined') { MAXNESTING=3 CYCLO=3
      gutil.log('Local version', env.modulePackage.version);
    }
    process.exit(0);
  }

  if (!env.modulePath) { CYCLO=4
    gutil.log(
      chalk.red('Local gulp not found in'),
      chalk.magenta(tildify(env.cwd))
    );
    gutil.log(chalk.red('Try running: npm install gulp'));
    process.exit(1);
  }

  if (!env.configPath) { CYCLO=5
    gutil.log(chalk.red('No gulpfile found'));
    process.exit(1);
  }

  // Check for semver difference between cli and local installation
  if (semver.gt(cliPackage.version, env.modulePackage.version)) { CYCLO=6
    gutil.log(chalk.red('Warning: gulp version mismatch:'));
    gutil.log(chalk.red('Global gulp is', cliPackage.version));
    gutil.log(chalk.red('Local gulp is', env.modulePackage.version));
  }

  // Chdir before requiring gulpfile to make sure
  // we let them chdir as needed
  if (process.cwd() !== env.cwd) { CYCLO=7
    process.chdir(env.cwd);
    gutil.log(
      'Working directory changed to',
      chalk.magenta(tildify(env.cwd))
    );
  }

  // This is what actually loads up the gulpfile
  require(env.configPath);
  gutil.log('Using gulpfile', chalk.magenta(tildify(env.configPath)));

  var gulpInst = require(env.modulePath);
  logEvents(gulpInst);

  process.nextTick(function() {
    if (simpleTasksFlag) {
      return logTasksSimple(env, gulpInst);
    }
    if (tasksFlag) {
      return logTasks(env, gulpInst);
    }
    gulpInst.start.apply(gulpInst, toRun);
  });
}

```

Figura 5.7: Función handleArguments.

Considerando la cantidad de líneas del archivo gulp.js donde se define la función analizada, y de las demás funciones que se definen en el mismo, se deduce que la función handleArguments contiene la mayor parte de la funcionalidad del archivo gulp.j. Por lo tanto, de acuerdo a las características mencionadas, se concluye que la función handleArguments, es un Brain Method en el archivo gulp.js de la aplicación Gulp.

5.4.3 Intensive Coupling

En la Tabla 5.12, se muestran los valores de las métricas de la función `handleArguments` (Figura 5.7). Dicha función, es la única que se puede considerar como posible code smell Intensive Coupling en la aplicación Gulp, de acuerdo a los valores de métricas que se obtuvieron al ejecutar la herramienta `vcomplex`.

Nombre de la función	MAXNESTING	CINT	CDISP
<code>handleArguments (bin/gulp.js)</code>	3	23	0.1739

Tabla 5.12: Valores de métricas de la ocurrencia del Code Smell Intensive Coupling en la aplicación Gulp.

```
var gutil = require('gulp-util');
var prettyTime = require('pretty-hrtime');
var chalk = require('chalk');
var semver = require('semver');
var archy = require('archy');
var Liftoff = require('liftoff');
var tildify = require('tildify');
var interpret = require('interpret');
var v8flags = require('v8flags');
var completion = require('../lib/completion');
var argv = require('minimist')(process.argv.slice(2));
var taskTree = require('../lib/taskTree');
```

Figura 5.8: Archivos js que se usan desde la función `handleArguments` del archivo `gulp.js`.

Al ejecutar `vcomplex`, se obtuvo el valor 3 para la métrica MAXNESTING, 23 para CINT y 0.1739 para la métrica CDISP.

1. 3 (MAXNESTING) > 1
2. 23 (CINT) > 7 - 8
3. 0.1739 (CDISP) < 0.5
4. 23 (CINT) > 2 - 5
5. 0.1739 (CDISP) < 0.25

Como se mencionó al analizar la aplicación #1, para que se cumpla la estrategia de detección del code smell Intensive Coupling basta con que se cumplan las condiciones 1, 2 y 3 ó las condiciones 1, 4 y 5; en este caso en particular, se cumple la totalidad de las condiciones. Por lo tanto, por cualquiera de los dos caminos condicionales posibles, se cumple la estrategia de detección. Entonces, al cumplirse la estrategia de detección, se concluye que la función `handleArguments` es un posible code smell Intensive Coupling en la aplicación Gulp.

El valor de métrica CINT que se obtuvo, supera ampliamente el valor máximo de nombres de funciones que puede recordar una persona, lo cual hace que la función sea compleja en cuanto a mantenibilidad. Para el desarrollar se volvería cada vez más difícil, a medida que pasa el tiempo, recordar cuál es la funcionalidad que representa cada una y por qué fue referenciada. Además, dichas funciones se encuentran distribuidas en cuatro del total de doce

archivos que se usan desde el archivo que contiene la función que se analiza (Figura 5.8), lo cual se ve reflejado en el valor de métrica CDISP que indica que la dispersión de las funciones en otros archivos es baja.

Al ser demasiadas las llamadas a funciones de otros archivos (Figura 5.7), en relación a la cantidad de archivos que se utilizan desde `handleArguments`, se puede deducir que dichas llamadas se encuentran centralizadas en muy pocos de otros archivos que se requieren desde `gulp.js` y que, por lo tanto, se considera que la función `handleArguments` es un code smell Intensive Coupling en la aplicación Gulp.

5.5 Aplicación #3: Bower

En esta tercera evaluación, se analiza la aplicación Bower [20], que es un administrador de paquetes para la web.

Bower, hace un seguimiento de todos los paquetes que contiene el sitio web que se desarrolla para asegurarse que estén actualizados; sino es así, instala las versiones correctas de los paquetes que necesita y sus dependencias. Puede administrar componentes que contienen HTML, CSS, JavaScript, fuentes o incluso archivos de imágenes.

5.5.1 Detección de Code Smells y análisis de métricas

En la aplicación Bower, se hallaron cuatro code smells diferentes:

1. Brain Method
2. God Class
3. Intensive Coupling
4. Dispersed Coupling

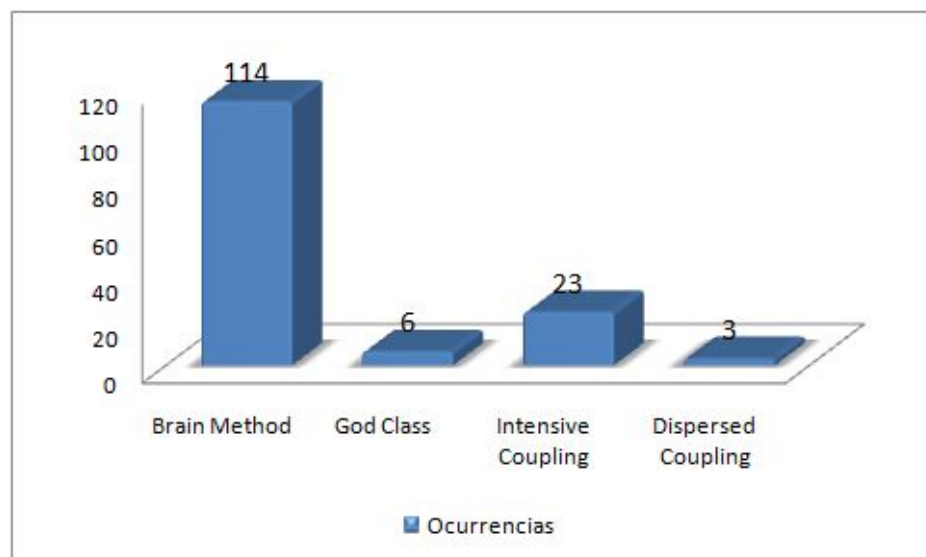


Figura 5.9: Ocurrencias de Code Smells en la aplicación Bower.

Para el primer code smell, Brain Method, se hallaron 114 ocurrencias, para God Class 6 ocurrencias, para Intensive Coupling 23, y por último, para el code smell Dispersed Coupling se encontraron 3 ocurrencias en el código de la aplicación (Figura 5.9).

5.5.2 Brain Method

En la Tabla 5.13, se muestran los valores de las métricas que se obtuvieron al ejecutar la herramienta vcomplex, y por las cuales se obtuvieron los code smells Brain Method en el código de la aplicación. Los valores corresponden a las 114 ocurrencias que se muestra en la Figura 5.9 para el code smell Brain Method.

Nombre de la función	LOC	CYCLO	MAXNESTING	NOAV
register (packages/bower-registry-client/lib/register.js)	52	3	2	11
<anonymous>._checkout (lib/core/resolvers/GitHubResolver.js)	71	4	2	16
setDefaults (lib/commands/init.js)	73	7	4	13
install (lib/commands/install.js)	21	4	2	10
list (lib/commands/list.js)	62	3	2	11
<anonymous> - línea 22 (lib/commands/list.js)	47	3	2	9
login (lib/commands/login.js)	102	4	3	15
search (lib/commands/search.js)	20	3	3	9
uninstall (lib/commands/uninstall.js)	32	3	2	11
unregister (lib/commands/unregister.js)	59	3	2	12
bump (lib/commands/version.js)	48	3	2	14
<anonymous> - línea 30 (lib/commands/version.js)	30	6	3	10
FsResolver (lib/core/resolvers/FsResolver.js)	17	3	2	8
GitHubResolver (lib/core/resolvers/GitHubResolver.js)	32	5	2	12
clean (lib/commands/cache/clean.js)	30	4	2	12
GitRemoteResolver (lib/core/resolvers/GitRemoteResolver.js)	25	4	2	11
<anonymous>._checkout (lib/core/resolvers/GitRemoteResolver.js)	52	4	2	10
<anonymous>._supportsShallowCloning (lib/core/resolvers/GitRemoteResolver.js)	54	7	2	11
<anonymous>._findResolution (lib/core/resolvers/GitResolver.js)	103	5	2	17
<anonymous> - línea 119 (lib/core/resolvers/GitResolver.js)	43	3	2	16
<anonymous> - línea 146 (lib/core/resolvers/GitResolver.js)	15	4	2	10
<anonymous> - línea 169 (lib/core/resolvers/GitResolver.js)	35	7	3	9
<anonymous>._savePkgMeta (lib/core/resolvers/GitResolver.js)	34	5	3	8
pluginResolverFactory (lib/core/resolvers/pluginResolverFactory.js)	309	3	2	34
<anonymous> - línea 138 (lib/core/resolvers/pluginResolverFactory.js)	30	5	2	10
<anonymous> - línea 98 (lib/core/resolvers/pluginResolverFactory.js)	40	4	4	12
<anonymous> - línea 107 (lib/core/resolvers/pluginResolverFactory.js)	24	4	2	10
<anonymous>._savePkgMeta (lib/core/resolvers/pluginResolverFactory.js)	28	4	2	9
<anonymous>._export (lib/core/resolvers/SvnResolver.js)	54	6	3	14
<anonymous>._findResolution (lib/core/resolvers/SvnResolver.js)	59	5	2	18
<anonymous> - línea 152 (lib/core/resolvers/SvnResolver.js)	43	3	2	17
<anonymous> - línea 179 (lib/core/resolvers/SvnResolver.js)	15	4	2	11
<anonymous> - línea 202 (lib/core/resolvers/SvnResolver.js)	23	5	2	10

<anonymous>._savePkgMeta (lib/core/resolvers/SvnResolver.js)	34	5	3	8
UrlResolver (lib/core/resolvers/UrlResolver.js)	18	3	2	10
<anonymous>._hasNew (lib/core/resolvers/UrlResolver.js)	47	4	2	10
<anonymous>._download (lib/core/resolvers/UrlResolver.js)	52	3	2	11
<anonymous>._parseHeaders (lib/core/resolvers/UrlResolver.js)	40	5	3	10
<anonymous>._extract (lib/core/resolvers/UrlResolver.js)	23	3	2	9
<anonymous> - línea 58 (lib/core/Manager.js)	15	3	2	8
<anonymous>.resolve (lib/core/Manager.js)	28	3	2	11
<anonymous>.install (lib/core/Manager.js)	95	3	2	32
<anonymous> - línea 181 (lib/core/Manager.js)	43	3	2	25
<anonymous>.toData (lib/core/Manager.js)	44	5	2	12
<anonymous>._onFetchSuccess (lib/core/Manager.js)	70	8	2	18
<anonymous>._onFetchError (lib/core/Manager.js)	24	4	2	10
<anonymous>._parseDependencies (lib/core/Manager.js)	85	3	2	21
<anonymous> - línea 455 (lib/core/Manager.js)	71	7	4	20
<anonymous> - línea 650 (lib/core/Manager.js)	30	4	2	11
<anonymous>._electSuitable (lib/core/Manager.js)	207	11	4	36
<anonymous> - línea 730 (lib/core/Manager.js)	31	7	4	9
<anonymous>._areCompatible (lib/core/Manager.js)	63	11	5	12
<anonymous> - línea 1065 (lib/core/Manager.js)	29	7	4	8
<anonymous> - línea 46 (lib/core//PackageRepository.js)	65	3	2	11
<anonymous> - línea 64 (lib/core//PackageRepository.js)	46	8	3	10
<anonymous>._extendLog (lib/core//PackageRepository.js)	25	5	2	8
<anonymous>.install (lib/core/Project.js)	93	4	2	26
<anonymous> - línea 83 (lib/core/Project.js)	33	3	2	14
<anonymous> - línea 87 (lib/core/Project.js)	19	7	3	11
<anonymous>- línea 45 (lib/core/Project.js)	32	3	2	12
<anonymous>.update (lib/core/Project.js)	93	3	2	19
<anonymous>.uninstall (lib/core/Project.js)	109	3	2	28
<anonymous> - línea 249 (lib/core/Project.js)	78	3	2	26
<anonymous> - línea 259 (lib/core/Project.js)	67	4	2	24
<anonymous>.walkTree (lib/core/Project.js)	41	6	3	10
<anonymous>.saveJson (lib/core/Project.js)	25	4	2	12
<anonymous> - línea 481 (lib/core/Project.js)	59	5	2	18
<anonymous> - línea 500 (lib/core/Project.js)	16	6	3	8
<anonymous> - línea 599 (lib/core/Project.js)	15	4	2	11
<anonymous> - línea 678 (lib/core/Project.js)	38	4	3	14
<anonymous> - línea 741 (lib/core/Project.js)	34	5	2	11
<anonymous>._restoreNode (lib/core/Project.js)	75	5	2	20
<anonymous> - línea 810 (lib/core/Project.js)	57	9	4	18
<anonymous> - línea 174 (lib/core/ResolveCache.js)	23	3	2	10
<anonymous> - línea 282 (lib/core/ResolveCache.js)	24	5	2	8
<anonymous> - línea 46 (lib/core/resolverFactory.js)	56	3	3	18
<anonymous> - línea 58 (lib/core/resolverFactory.js)	29	3	3	14
<anonymous> - línea 136 (lib/core/resolverFactory.js)	40	4	2	10
orderByDependencies (lib/core/scripts.js)	45	6	4	12
<anonymous> - línea 57 (lib/renderers/JsonRenderer.js)	59	8	2	15
standardRenderer (lib/renderers/StandardRenderer.js)	3	3	2	12
<anonymous>.error (lib/renderers/StandardRenderer.js)	36	9	2	10
<anonymous>._info (lib/renderers/StandardRenderer.js)	37	5	3	10

<anonymous> ._prefix (lib/renderers/StandardRenderer.js)	31	7	3	11
<anonymous> ._tree2archy (lib/renderers/StandardRenderer.js)	57	16	3	9
readOptions (lib/util/cli.js)	34	3	2	9
getWindowsCommand (lib/util/cmd.js)	25	4	2	9
<anonymous> - línea 88 (lib/util/cmd.js)	24	4	3	9
<anonymous> - línea 32 (lib/util/createLink.js)	15	3	2	8
<anonymous> - línea 44 (lib/util/download.js)	23	3	2	8
<anonymous> - línea 85 (lib/util/download.js)	35	3	2	13
extract (lib/util/extract.js)	66	4	2	15
<anonymous> - línea 17 (lib/util/readJson.js)	18	4	3	8
rc (packages/bower-config/lib/util/rc.js)	39	9	2	13
<anonymous> - línea 109 (packages/bower-config/lib/util/rc.js)	24	4	3	11
<anonymous> .create (packages/bower-config/test/helpers.js)	28	5	2	12
<anonymous> - línea 17 (packages/bower-json/lib/json.js)	43	3	2	8
getIssues (packages/bower-json/lib/json.js)	80	11	4	10
<anonymous> - línea 173 (packages/bower-json/lib/json.js)	22	7	3	8
find (packages/bower-json/lib/json.js)	35	3	2	12
findSync (packages/bower-json/lib/json.js)	25	4	2	8
<anonymous> .prompt (packages/bower-logger/lib/Logger.js)	58	3	2	12
<anonymous> - línea 91 (packages/bower-logger/lib/Logger.js)	30	3	2	8
<anonymous> .get (packages/bower-registry-client/lib/util/Cache.js)	48	4	2	8
<anonymous> .set (packages/bower-registry-client/lib/util/Cache.js)	30	4	2	12
doRequest (packages/bower-registry-client/lib/list.js)	46	3	2	14
<anonymous> - línea 29 (packages/bower-registry-client/lib/lookup.js)	20	4	2	9
doRequest (packages/bower-registry-client/lib/lookup.js)	58	3	2	16
<anonymous> - línea 94 (packages/bower-registry-client/lib/lookup.js)	31	7	2	8
clearCache (packages/bower-registry-client/lib/lookup.js)	19	3	2	8
doRequest (packages/bower-registry-client/lib/search.js)	46	3	2	15
unregister (packages/bower-registry-client/lib/unregister.js)	41	3	2	10
exports.command (test/helpers.js)	42	4	2	10
<anonymous> - línea 72 (Gruntfile.js)	141	7	2	22

Tabla 5.13: Valores de métricas de las ocurrencias del Code Smell Brain Method en la aplicación Bower.

Se toman como ejemplo, las dos primeras funciones que representan Brain Method en la aplicación Bower.

La primera función, que se muestra en el Figura 5.10, corresponde a la función register que se encuentra en el archivo register.js. Dicho archivo cuenta con 58 líneas de código, de las cuales 52 forman parte de la función register, lo que sería casi el 90% del total de líneas del archivo. En las líneas restantes, se declaran variables y se realiza el exports de la función require, para que la misma se pueda utilizar desde otros archivos js que la requieran.

Los valores de métricas, que se muestran en la Tabla 5.13 para la función register, y que se utilizan para evaluar la estrategia de detección del Code Smell Brain Method, son: 52 para la métrica LOC, 3 para la métrica CYCLO, 2 para la métrica MAXNESTING y 11 para la métrica NOAV.

1. 52 (LOC) > 13
2. 3 (CYCLO) >= 2.4
3. 2 (MAXNESTING) >= 2 - 5
4. 11 (NOAV) > 7 - 8

Como dichos valores, al ser comparados con los umbrales definidos para cada una de las métricas, cumplen con las cuatro condiciones necesarias para que se valide la estrategia de detección, se considera que la función register es un posible Brain Method en la aplicación Bower.

```
var parseUrl = require('url').parse;
var request = require('request');
var createError = require('./util/createError');

function register(name, url, callback) { MAXNESTING=1 CYCLO=1
  var config = this._config;
  var requestUrl = config.registry.register + '/packages';
  var remote = parseUrl(requestUrl);
  var headers = {};

  if (config.userAgent) { MAXNESTING=2 CYCLO=2
    headers['User-Agent'] = config.userAgent;
  }

  if (config.accessToken) { CYCLO=3
    requestUrl += '?access_token=' + config.accessToken;
  }

  request.post({
    url: requestUrl,
    headers: headers,
    ca: config.ca.register,
    strictSSL: config.strictSsl,
    timeout: config.timeout,
    json: true,
    form: {
      name: name,
      url: url
    }
  }, function (err, response) {
    // If there was an internal error (e.g. timeout)
    if (err) {
      return callback(createError('Request to ' + requestUrl + ' failed: ' + err.message, err.code));
    }

    // Duplicate
    if (response.statusCode === 403) {
      return callback(createError('Duplicate package', 'EDUPLICATE'));
    }

    // Invalid format
    if (response.statusCode === 400) {
      return callback(createError('Invalid URL format', 'EINFORMAT'));
    }

    // Everything other than 201 is unknown
    if (response.statusCode !== 201) {
      return callback(createError('Unknown error: ' + response.statusCode + ' - ' + response.body, 'EUNKNOWN'));
    }

    callback(null, {
      name: name,
      url: url
    });
  });
}

module.exports = register;
```

Figura 5.10: Archivo register.js que contiene la función register.

La función `register` representa casi en su totalidad la funcionalidad del archivo `register.js`. Sin embargo, la función no tiene tantas líneas de código en relación a otras funciones que también ocupan casi la totalidad del archivo. Como es el caso de la función `module.exports` del archivo `args.js`, que se tomó como ejemplo al evaluar la aplicación `Browserify`, que tiene 245 líneas de código en un archivo de 270 líneas. Además, el máximo nivel de anidamiento tampoco es alto, así como tampoco lo es la complejidad ciclomática, por lo cual la función no es compleja, y por lo tanto no es difícil entender su funcionalidad. Si bien podrían hacerse mejoras en la función para que sea aún más intuitiva, no representa grandes riesgos si se debe modificar en el futuro. A veces, es necesario que las funciones ocupen casi la totalidad del archivo, si el mismo no es extenso y no representa riesgos a futuro al tener que modificarlo.

Por lo tanto, en relación a lo anteriormente dicho, se puede concluir que la función `register` no representa un problema de gran importancia para la aplicación `Bower`, lo cual significa que no es un `Brain Method`, siendo un falso positivo.

La segunda función, que se toma como ejemplo es la función `_checkout` que se muestra en la Figura 5.11. Los valores de las métricas se muestran en la Tabla 5.13 y son: 71 para la métrica `LOC`, 4 para la métrica `CYCLO`, 2 para la métrica `MAXNESTING` y, por último, 16 para la métrica `NOAV`.

1. 71 (`LOC`) > 13
2. 4 (`CYCLO`) >= 2.4
3. 2 (`MAXNESTING`) >= 2 - 5
4. 16 (`NOAV`) > 7 - 8

Al comparar los valores de las métricas, que se obtuvieron para la función `_checkout` al ejecutar la herramienta `vcomplex`, con los umbrales definidos en la estrategia de detección del `code smell Brain Method`, se puede decir, que cumplen con las cuatro condiciones necesarias para que se valide la misma y que, por lo tanto, se considera que la función `_checkout` es un posible `Brain Method` en la aplicación `Bower`.

Al analizar los valores de las métricas por los cuales se cumplió la estrategia de detección del `code smell Brain Method`, se puede ver que la función no tiene un nivel de anidamiento alto ni una complejidad ciclomática elevada, por lo cual se podría decir que la misma, de acuerdo a estas métricas, no es compleja. Sin embargo si se tiene en cuenta la cantidad de líneas de código que tiene la función, en relación a la cantidad de líneas del archivo `js` que la contiene, `GitHubResolver.js`, podría considerarse que la función representa gran parte de la funcionalidad del mismo, y que si bien no tiene una complejidad definida por su nivel de anidamiento o su complejidad ciclomática, esto indicaría que la función es compleja y podría traer riesgos en su mantenibilidad a futuro.

```

GitHubResolver.prototype._checkout = function () { MAXNESTING=1 CYCLO=1
  var msg;
  var name = this._resolution.tag || this._resolution.branch || this._resolution.commit;
  var tarballUrl = 'https://github.com/' + this._org + '/' + this._repo + '/archive/' + name + '.tar.gz';

  var file = path.join(this._tempDir, 'archive.tar.gz');
  var reqHeaders = {};
  var that = this;

  if (this._config.userAgent) { MAXNESTING=2 CYCLO=2
    reqHeaders['User-Agent'] = this._config.userAgent;
  }

  this._logger.action('download', tarballUrl, { CYCLO=3
    url: that._source,
    to: file
  });

  // Download tarball
  return download(tarballUrl, file, {
    ca: this._config.ca.default,
    strictSSL: this._config.strictSsl,
    timeout: this._config.timeout,
    headers: reqHeaders
  }) CYCLO=4
  .progress(function (state) {
    // Retry?
    if (state.retry) {
      msg = 'Download of ' + tarballUrl + ' failed with ' + state.error.code + ', ';
      msg += 'retrying in ' + (state.delay / 1000).toFixed(1) + 's';
      that._logger.debug('error', state.error.message, { error: state.error });
      return that._logger.warn('retry', msg);
    }

    // Progress
    msg = 'received ' + (state.received / 1024 / 1024).toFixed(1) + 'MB';
    if (state.total) {
      msg += ' of ' + (state.total / 1024 / 1024).toFixed(1) + 'MB downloaded, ';
      msg += state.percent + '%';
    }
    that._logger.info('progress', msg);
  })
  .then(function () {
    // Extract archive
    that._logger.action('extract', path.basename(file), {
      archive: file,
      to: that._tempDir
    });

    return extract(file, that._tempDir)
    // Fallback to standard git clone if extraction failed
    .fail(function (err) {
      msg = 'Decompression of ' + path.basename(file) + ' failed' + (err.code ? ' with ' + err.code : '') + ', ';
      msg += 'trying with git..';
      that._logger.debug('error', err.message, { error: err });
      that._logger.warn('retry', msg);

      return that._cleanTempDir()
        .then(GitRemoteResolver.prototype._checkout.bind(that));
    });
  })
  // Fallback to standard git clone if download failed
  }, function (err) {
    msg = 'Download of ' + tarballUrl + ' failed' + (err.code ? ' with ' + err.code : '') + ', ';
    msg += 'trying with git..';
    that._logger.debug('error', err.message, { error: err });
    that._logger.warn('retry', msg);

    return that._cleanTempDir()
      .then(GitRemoteResolver.prototype._checkout.bind(that));
  });
};

```

Figura 5.11: Función _checkout.

El archivo GitHubResolver.js, tiene 145 líneas de código y contiene 9 funciones, por lo que cada una de las funciones debería tener alrededor de 16 líneas de código. La función _checkout que se evalúa, tiene 71 líneas, lo que sería casi la mitad del total de líneas del archivo, y superando ampliamente el promedio de líneas que debería tener. Como se mencionó anteriormente, la función representa gran parte de la funcionalidad del archivo GitHubResolver.js en comparación a la cantidad de líneas que tienen las demás funciones que

lo componen, dificultando su comprensión y por lo tanto la mantenibilidad. Por lo tanto, se puede deducir que la misma es un problema estructural para la aplicación Bower, lo que significa que la función `_checkout` es un Brain Method.

5.5.3 God Class

En la Tabla 5.14, se muestran las ocurrencias del code smell God Class en la aplicación que se analiza, junto a los valores de las métricas que se evalúan en la estrategia de detección de dicho code smell para determinar que el archivo js es una posible God Class. Se muestra el nombre del archivo y entre paréntesis la ruta de acceso al mismo dentro de la aplicación Bower.

Nombre del archivo	ATFD	WMC	TCC
resolveCache.js (test/core)	330	117	0
test.js (packages/bower-json/test)	66	86	0
urlResolver.js (test/core/resolvers)	171	89	0
resolverFactory.js (lib/core)	15	51	0
resolverFactory.js (test/core)	26	89	0
helpers.js (test)	38	58	3

Tabla 5.14: Valores de métricas de las ocurrencias del Code Smell God Class en la aplicación Bower.

Si se observan los valores de las métricas de los archivos que se muestran en la Tabla 5.14, se puede decir que el js más representativo para este code smell es el que tiene el mayor valor para las métricas ATFD y WMC, ya que, de esta manera, se tomaría como referencia el archivo que centraliza la mayor parte de la funcionalidad de la aplicación. Por lo tanto, se utiliza como ejemplo el archivo `resolveCache.js`.

Los valores de las métricas para el archivo `resolveCache.js`, que se obtuvieron al ejecutar la herramienta `vcomplex` y se utilizaron para validar la estrategia de detección del code smell God Class, son: 330 para la métrica ATFD, 117 para la métrica WMC y, por último, para la métrica TCC se obtuvo valor 0.

1. 330 (ATFD) > **2-5**
2. 117 (WMC) >= **47**
3. 0 (TCC) >= **3 (ONE-THIRD del total de pares de métodos del archivo js)**

Al evaluar el valor de las métricas del archivo, se observa que, la cantidad de atributos de otros archivos js a los que se accede es demasiado alta, superando ampliamente el umbral impuesto en la estrategia de detección para la métrica ATFD. Esto significa, que se está rompiendo el encapsulamiento de los datos, y aumentando el riesgo futuro ante cambios en alguno de los archivos que se usan desde `resolveCache`. Ya que, impacte o no, por cada archivo relacionado que se modifique, se debe verificar que no se vea afectado el mismo.

Además, de esta manera no se conserva la flexibilidad ni la extensibilidad del archivo, impactando directamente sobre su mantenibilidad.

Si se analiza la métrica WMC, también se puede observar que, de acuerdo al valor que tiene la misma, el archivo resolveCache presenta una complejidad funcional muy alta, en relación al umbral impuesto en la estrategia de detección. Esto hace que el archivo sea complejo, y por lo tanto se vean afectadas, directamente, la mantenibilidad y la extensibilidad del mismo.

En la aplicación, hay archivos que tienen mayor cantidad de líneas de código que el archivo que se analiza. Sin embargo, lo que hace particular al archivo resolveCache, es que, no tiene cohesión. Esto quiere decir que, los componentes del archivo no tienen relación, y que, cada uno de ellos representa una funcionalidad aislada, impactando directamente en la comprensión de la funcionalidad que representa el archivo en su totalidad, y dificultando su mantenimiento en el tiempo.

De acuerdo a lo mencionado, se puede concluir que el archivo resolveCache.js, que se encuentra en la dirección test/core de la aplicación Bower, es un code smell God Class en la misma. Se debe mencionar, que no se muestra el código del archivo en el documento, ya que el mismo es muy extenso.

5.5.4 Intensive Coupling

En la Tabla 5.15, se muestran los valores de las métricas de las 23 (veintitrés) ocurrencias que son posibles code smells Intensive Coupling en la aplicación que se analiza.

Nombre de la función	MAXNESTING	CINT	CDISP
ensurePackage (test/packages-svn.js)	2	13	0.3076
setDefaults (lib/commands/init.js)	3	11	0.4545
login (lib/commands/login.js)	3	15	0.4545
bump (lib/commands/version.js)	2	8	0.375
pluginResolverFactory (lib/core/resolvers/pluginResolverFactory.js)	2	31	0.2580
<anonymous>.resolve (lib/core/resolvers/pluginResolverFactory.js)	2	12	0.25
<anonymous> - línea 98 (lib/core/resolvers/pluginResolverFactory.js)	4	8	0.375
<anonymous>.install (lib/core/Manager.js)	2	22	0.3181
<anonymous>._electSuitable (lib/core/Manager.js)	4	16	0.25
<anonymous>._areCompatible (lib/core/Manager.js)	5	12	0.0833
<anonymous> - línea 136 (lib/core/resolverFactory.js)	2	16	0.25
<anonymous> - línea 14 (test/core/resolvers/fsResolver.js)	2	153	0.0671
<anonymous> - línea 12 (test/core/resolvers/gitFsResolver.js)	2	57	0.1578
<anonymous> - línea 129 (test/core/resolvers/gitFsResolver.js)	2	10	0.3
<anonymous> - línea 10 (test/core/resolvers/gitHubResolver.js)	2	59	0.1359

<anonymous> - línea 12 (test/core/resolvers/gitRemoteResolver.js)	2	100	0.1
<anonymous> - línea 455 (test/core/resolvers/gitRemoteResolver.js)	2	10	0.3
<anonymous> - línea 9 (test/core/resolvers/pluginResolverFactory.js)	2	59	0.1186
<anonymous> - línea 13 (test/core/resolvers/urlResolver.js)	2	284	0.0387
<anonymous> - línea 669 (test/core/resolvers/urlResolver.js)	2	21	0.1428
<anonymous> - línea 15 (test/core/resolverFactory.js)	2	159	0.0817
ensurePackage (test/packages.js)	2	11	0.3636
<anonymous> - línea 72 (Gruntfile.js)	2	23	0.3043

Tabla 5.15: Valores de métricas de las ocurrencias del Code Smell Intensive Coupling en la aplicación Bower.

Se toma como ejemplo la función ensurePackage (Figura 5.13), que se encuentra en el archivo packages-svn.js de la aplicación Bower. Los valores de métricas que se obtuvieron para la misma fueron, 2 para la métrica MAXNESTING, 13 para CINT y 0.3076 para CDISP.

1. 2 (MAXNESTING) > 1
2. 13 (CINT) > 7 - 8
3. 0.3076 (CDISP) < 0.5
4. 13 (CINT) > 2 - 5
5. 0.3076 (CDISP) < 0.25

Como ya se mencionó anteriormente al analizar el code smell Intensive Coupling en las aplicaciones #1 y #2, para que se cumpla la estrategia de detección del mismo se debe cumplir la condición uno, que involucra la métrica MAXNESTING y, además, se deben cumplir las condiciones 2 y 3 ó las condiciones 4 y 5. En esta caso en particular, para la función ensurePackage, se cumplen además de la condición 1, las condiciones 2 y 3. Por lo tanto, se puede decir que la misma es un posible code smell Intensive Coupling para la aplicación Bower.

```
var fs = require('../lib/util/fs');
var path = require('path');
var Q = require('q');
var semver = require('semver');
var mout = require('mout');
var rimraf = require('../lib/util/rimraf');
var mkdirp = require('mkdirp');
var chalk = require('chalk');
var cmd = require('../lib/util/cmd');
var packages = require('../packages-svn.json');
var nopt = require('nopt');
```

Figura 5.12: Archivos js que se usan desde la función ensurePackage del archivo packages-svn.js.


```

function ensurePackage(admin, dir) { MAXNESTING = 1
  var promise = new Q(); CINT = 1

  // If force is specified, delete folder
  if (options.force) { MAXNESTING = 2
    promise = promise.then(function () {
      return Q.nfcall(rimraf, admin); CINT = 2
    });

    promise = promise.then(function () {
      return Q.nfcall(rimraf, dir); CINT = 3
    });

    promise = promise.then(function () {
      throw new Error();
    });
  }
  // Otherwise check if .git is already created
} else {
  promise = Q.nfcall(fs.stat, path.join(dir, '.svn')); CINT = 4 - 5 - 6
}

// Only create if stat failed
return promise.fail(function () {
  // Create dir
  return Q.nfcall(mkdirp, dir) CINT = 7
  // Init svn repo
  .then(cmd.bind(null, 'svnadmin', ['create', admin], {})) CINT = 8
  // checkout the repo
  .then(cmd.bind(null, 'svn', ['checkout', pathToUrl(admin), dir], {})) CINT = 9
  // create directory structure
  .then(cmd.bind(null, 'svn', ['mkdir', 'trunk'], { cwd: dir })) CINT = 10
  .then(cmd.bind(null, 'svn', ['mkdir', 'tags'], { cwd: dir })) CINT = 11
  .then(cmd.bind(null, 'svn', ['mkdir', 'branches'], { cwd: dir })) CINT = 12
  // Commit
  .then(function () {
    return cmd('svn', ['commit', '-m"Initial commit."'], { CINT = 13
      cwd: dir,
      env: env
    });
  })
  .then(function () {
    return dir;
  });
});
}

```

Figura 5.13: Función ensurePackage.

En la Figura 5.12, se muestran los archivos de los cuales se realizan llamadas a métodos o propiedades desde ensurePackage. Estas llamadas se consideran para el cálculo de la métrica CINT. En la Figura 5.13, se encuentran marcadas dichas llamadas, y como se puede ver, son 13 y se encuentran distribuidas en 4 archivos distintos del total de 11 archivos que se usan desde el archivo package-svn.js que contiene la función. Esto quiere decir que las llamadas se encuentran distribuidas en menos de la mitad del total de archivos, lo cual provoca que la relación entre la función y los archivos js que se usan sea intensa, indicando un grado de acoplamiento alto. Principalmente, el acoplamiento de la función es con el archivo js cmd, al cual se realizan casi la mitad de las llamadas.

Por lo tanto, además de que los métodos a los que se llaman superan la capacidad de memoria de una persona, se llaman a más de cuatro métodos y/o propiedades de dos de los archivos js que se utilizan de un total de cuatro, lo cual indica que el acoplamiento es alto y que se está en presencia de un code smell Intensive Coupling en la aplicación Bower.

5.5.5 Dispersed Coupling

Para realizar el análisis del code smell Dispersed Coupling en la aplicación que se analiza, se muestran en la Tabla 5.16 las tres ocurrencias que se hallaron del mismo, junto a sus métricas.

Nombre de la función	MAXNESTING	CINT	CDISP
<anonymous>._readJson (lib/core/Project.js)	2	9	0.5555
<anonymous>._readInstalled (lib/core/Project.js)	2	8	0.5
<anonymous> - línea 34 (test/core/packageRepository.js)	2	13	0.6923

Tabla 5.16: Valores de métricas de las ocurrencias del Code Smell Dispersed Coupling en la aplicación Bower.

Se toma como ejemplo la función `_readJson` (Figura 5.15), que se encuentra en el archivo `Project.js` de la aplicación Bower. Los valores de métricas que se obtuvieron para la misma fueron, 2 para la métrica `MAXNESTING`, 9 para `CINT` y 0.5555 para `CDISP`.

1. $2 (\text{MAXNESTING}) > 1$
2. $9 (\text{CINT}) > 7 - 8$
3. $0.5555 (\text{CDISP}) \geq 0.5$

Al cumplirse las tres condiciones que se muestran para la estrategia de detección del code smell que se analiza, se concluye que la función `_readJson`, es un posible code smell Dispersed Coupling para la aplicación Bower.

```
var glob = require('glob');
var path = require('path');
var fs = require('../util/fs');
var Q = require('q');
var mout = require('mout');
var rimraf = require('../util/rimraf');
var endpointParser = require('bower-endpoint-parser');
var Logger = require('bower-logger');
var md5 = require('md5-hex');
var Manager = require('../Manager');
var semver = require('../util/semver');
var createError = require('../util/createError');
var readJson = require('../util/readJson');
var validLink = require('../util/validLink');
var scripts = require('../scripts');
var relativeToBaseDir = require('../util/relativeToBaseDir');
```

Figura 5.14: Archivos js que se usan desde la función `_readJson` del archivo `Project.js`.

En la Figura 5.14, se pueden observar todos los archivos que se usan desde el archivo Project.js que contiene la función que se analiza. Además, en dicha figura, se encuentran destacados los que se usan específicamente desde la función. Como se puede ver, los archivos que se usan desde la función son 5 de un total de 9 archivos, por lo cual la dispersión apenas supera la mitad de los mismos.

Si bien, la cantidad de operaciones que se llaman de otros archivos a las que se encuentra ligada la función _readJson (Figura 5.15), supera la capacidad de memoria de una persona, no es alta. Sino que, se supera dicho umbral por muy poco. Además, se debe destacar que el máximo nivel de anidamiento de la función tampoco es alto. Por lo tanto, las métricas CINT y MAXNESTING, no representan un gran riesgo a futuro en relación a la mantenibilidad del código.

```
Project.prototype._readJson = function () { MAXNESTING = 1
  var that = this;

  if (this._json) { MAXNESTING = 2
    return Q.resolve(this._json);
  }

  return Q.fcall(function () { CINT = 1
    // This will throw if package.json does not exist
    return fs.readFileSync(path.join(that._config.cwd, 'package.json')); CINT = 2 - 3
  })
  .then(function (buffer) {
    // If package.json exists, use it's values as defaults
    var defaults = {}, npm = JSON.parse(buffer.toString());

    defaults.name      = npm.name || path.basename(that._config.cwd) || 'root'; CINT = 4
    defaults.description = npm.description;
    defaults.main       = npm.main;
    defaults.authors    = npm.contributors || npm.author;
    defaults.license    = npm.license;
    defaults.keywords   = npm.keywords;

    return defaults;
  })
  .catch(function (err) {
    // Most likely no package.json so just set default name
    return { name: path.basename(that._config.cwd) || 'root' }; CINT = 5
  })
  .then(function (defaults) {
    return that._json = readJson(that._config.cwd, { assume: defaults, logger: that._logger }); CINT = 6
  })
  .spread(function (json, deprecated, assumed) {
    var jsonStr;

    if (deprecated) {
      that._logger.warn('deprecated', 'You are using the deprecated ' + deprecated + ' file');
    }

    if (!assumed) {
      that._jsonFile = path.join(that._config.cwd, deprecated ? deprecated : 'bower.json'); CINT = 7
    }

    jsonStr = JSON.stringify(json, null, ' ') + '\n';
    that._jsonHash = md5(jsonStr); CINT = 8
    return that._json = json;
  });
};
```

Figura 5.15: Función _readJson.

De acuerdo a lo que se mencionó anteriormente, al analizar la función _readJson que se usó como ejemplo para el code Ssell Dispersed Coupling, se concluye que, si bien se cumple la

estrategia de detección de dicho code smell, debido a que los valores de las métricas que se obtuvieron al ejecutar la herramienta vcomplex superan los umbrales determinados en la misma, la función no es un code smell Dispersed Coupling en la aplicación Bower.

5.6 Resumen de los casos de estudio

A partir del análisis que se realizó para cada una de las aplicaciones que se presentaron como casos de estudio, se puede concluir que, si bien los valores de las métricas que arroja la herramienta son certeros, no siempre quiere decir que si se cumple la estrategia de detección del code smell que se analiza se cumple que el código verdaderamente representa dicho code smell.

De los casos de estudio analizados, se puede observar que el code smell Brain Method es el que más se ha encontrado. Este code smell significa que los problemas que más se detectan en el código de los sistemas que se analizan, se relacionan con funciones extensas que poseen muchas líneas de código, y que además, presentan un nivel de anidamiento y una complejidad ciclomática considerable. Lo cual hace que las funciones sean difíciles de entender y mantener en el tiempo.

El code smell que menos se ha encontrado es el code smell God Class. Esto, de acuerdo a los resultados que se obtuvieron al analizar los casos de estudio y en relación al code smell Brain Method, indica que si bien los archivos js poseen funciones que se consideran como extensas, en general, la cohesión entre las mismas no es baja y la complejidad de la clase, en su totalidad, no llega a ser demasiado alta.

En relación a los code smell Intensive Coupling y Dispersed Coupling, se hallan ocurrencias en los sistemas que se analizan pero mucha menos cantidad que las del code smell Brain Method. Por lo general, la cantidad de ocurrencias de ambos smell se asemejan, sin embargo el code smell Intensive Coupling tiene tendencia a ser más encontrado. Lo que indica que, hay una tendencia a que las funciones usen funcionalidad de unos pocos archivos relacionados al archivo que la contiene.

En ocasiones, dependiendo cómo sea el contexto en general en el cual se encuentra el archivo o la función que se analiza, es necesario que algunos “problemas” no se consideren como tal, y no representen grandes riesgos a futuro.

Por lo tanto, la herramienta ratifica los problemas a casos específicos que luego deben ser evaluados por el desarrollador, para definir si verdaderamente son un problema de diseño en su aplicación, o si, en caso contrario, no presentan grandes riesgos a futuro relacionados con la calidad y/o mantenibilidad del código.

Finalmente, se puede decir que, a partir del análisis de code smells en el lenguaje JavaScript y la definición de un catálogo de métricas para el mismo, se pudo implementar la herramienta vcomplex para automatizar el proceso de búsqueda de code smells y los resultados que se obtuvieron fueron muy buenos.

Capítulo 6

Conclusión

6.1 Conclusiones finales

La identificación de code smells es muy importante para mejorar la calidad del software y prevenir problemas relacionados a la mantenibilidad y modificabilidad del sistema que se desarrolla. Un code smell, indica que una porción de código debe ser mejorada. No es considerado un error sino que se considera como un antipatrón de diseño. Para poder eliminar code smells, es necesario detectarlos. Para ello se presentan estrategias de detección que se construyen a partir de condiciones lógicas que se rigen por métricas y umbrales. Los umbrales son límites para validar las condiciones de acuerdo al valor de las métricas. Para mantener un orden y definir estrategias claras, se debe contar con un catálogo de métricas.

Lanza y Marinescu [1], presentan un catálogo de métricas para la detección de code smells en el lenguaje Java. Lo cual es un aporte muy interesante para la detección de problemas en dicho lenguaje, y para que los desarrolladores puedan validar su código considerando mantener la calidad del mismo.

El lenguaje JavaScript, actualmente, es uno de los más utilizados por los desarrolladores. Hasta el momento, no se ha definido un catálogo de code smells basado en métricas para el lenguaje JavaScript. Por lo cual, es muy importante la definición de un catálogo basado en métricas para la detección de code smells en el lenguaje JavaScript. Este trabajo final realiza un análisis en detalle de problemas de diseño en el lenguaje JavaScript para la definición de un catálogo de métricas en dicho lenguaje, considerando el catálogo de métricas definido por Lanza y Marinescu [1] para el lenguaje Java.

A partir del análisis realizado, se identificaron los code smells que se pueden detectar en el lenguaje JavaScript y las métricas necesarias para la detección de los mismos. Además, se identificaron las métricas que deben adaptarse para su definición en el lenguaje JavaScript, analizando en detalle las características que se deben considerar para adaptar la mismas.

De los code smells que se analizaron, sólo uno se concluyó que no puede ser detectado en el lenguaje JavaScript, se trata del code smell Data Class; y se debe a que no se puede definir la métrica NOAM, que se necesita para validar la estrategia de detección de dicho code smell, en el lenguaje JavaScript. Dicha métrica no puede definirse de manera adecuada debido a las características que presenta el estándar ECMAScript.

Una vez definidas las métricas necesarias para la detección de code smells en el lenguaje JavaScript, se implementó la herramienta vcomplex. Dicha herramienta obtiene los valores de las métricas necesarias para la identificación de posibles code smells.

Finalmente, la identificación de code smells se realiza utilizando los valores de las métricas que se obtuvieron al ejecutar vcomplex, para validar las estrategias de detección de dichos smells. Si se cumplen las condiciones de la estrategia de detección de un determinado code smell, se puede decir que se halló un posible code smell en el lenguaje JavaScript. La tarea del desarrollador, será la de definir si verdaderamente los posibles code smells que se identificaron representan problemas estructurales en el código.

Para analizar los resultados y determinar si el análisis de code smells fue positivo, se realizaron pruebas sobre tres aplicaciones diferentes. Para cada una de ellas se obtuvieron posibles code smells a partir de la ejecución de la herramienta vcomplex y del reemplazo de los valores de las métricas en las estrategias de detección. De los posibles code smells que se obtuvieron al validarse las estrategias de detección, se seleccionaron ejemplos de cada uno y se analizó si verdaderamente se trataba de un code smell en el lenguaje JavaScript. Para algunos casos, el code smell que se halló se trataba de un falso positivo, ya que al analizar las características y el entorno del código de ejemplo, se pudo observar que no se trataba verdaderamente de un problema en el código. Sin embargo, en otros casos, sí se pudo validar que se trataba de un code smell en el lenguaje JavaScript.

Por lo tanto, el análisis realizado fue positivo; ya que se acota la búsqueda de code smells en el lenguaje JavaScript a un grupo de posibles smells válidos a partir del cumplimiento de sus estrategias de detección. Siendo de esta manera un gran aporte para el mantenimiento y la extensión, entre otras características, que ayudan a crear sistemas de software de calidad, y para los desarrolladores del lenguaje JavaScript.

6.2 Contribuciones

La contribución más importante de este trabajo es el análisis de la detección de code smells en el lenguaje JavaScript, identificando las métricas que se deben adaptar para ello. Específicamente, se pueden detallar las siguientes contribuciones:

- Descripción de code smells en el lenguaje JavaScript: A partir del análisis que se realizó, se identificaron los code smells que pueden ser detectados en el lenguaje JavaScript. Se tomó como referencia los code smells propuestos por Lanza y Marinescu. De la misma manera se descartaron los code smells que no pueden definirse en el lenguaje JavaScript. Para el análisis realizado en este trabajo, sólo se halló que un único code smell no puede ser detectado en el lenguaje JavaScript.

- Descripción de métricas en el lenguaje JavaScript: Se describieron las métricas que se pueden utilizar en la detección de code smells en el lenguaje JavaScript, a partir de las métricas propuestas por Lanza y Marinescu en su catálogo de métricas. Se determinó además, qué métricas deben ser adaptadas para su utilización en dicho lenguaje.
- Implementación de la herramienta vcomplex: Se implementó una herramienta para hallar los valores de las métricas definidas para la detección de code smells en el lenguaje JavaScript.
- Extensión de la herramienta vcomplex: El diseño y la implementación de la herramienta es lo suficientemente flexible como para poder agregar nuevas métricas al análisis.

Este trabajo realiza un aporte para mejorar la calidad de un sistema implementado en JavaScript, en este caso relacionada con la modificabilidad y la mantenibilidad del mismo. A partir del análisis realizado y la herramienta vcomplex, se pueden hallar los problemas de diseño estructurales más destacables en el lenguaje JavaScript.

6.3 Limitaciones

A continuación se describen algunas limitaciones del análisis realizado:

- A partir de los resultados obtenidos de los casos de estudio analizados, se puede demostrar que si bien el análisis realizado junto con la herramienta vcomplex, ayudan a detectar síntomas de malas prácticas de diseño e implementación (code smells), no se puede evitar que se detecten falsos positivos. Esto se debe a que no es suficiente el análisis objetivo que se hace sobre las métricas. Por lo tanto, no se puede obviar la tarea de los desarrolladores, que deben revisar cada uno de los posibles smells detectados. Este problema también sucede en el lenguaje Java.
- En el análisis de code smells realizado, sólo se considera un conjunto reducido de smells. No se realiza el análisis de todos los code smells que proponen Lanza y Marinescu [1].
- Como para el análisis de code smells se consideró un conjunto de los smells propuestos por Lanza y Marinescu, las métricas que se analizaron son las métricas que se utilizan en las estrategias de detección de dichos code smells. Por lo tanto, no se analizó el total de métricas que proponen Lanza y Marinescu.

6.4 Trabajos futuros

Como sucede en cualquier proyecto de investigación, y como continuación de este trabajo, existen diversas cuestiones sobre las que es posible seguir trabajando. A continuación se detallan trabajos futuros que pueden realizarse como resultado de esta investigación o que, por exceder el alcance de este trabajo no se han podido tratar en detalle.

- Como se mencionó en las limitaciones de este trabajo, se podría ampliar el conjunto de code smells que se analizó. De esta manera, al incluir code smells, también se incluirían las métricas que no se analizaron.
- Se podrían considerar code smells propuestos por otros autores, ya que en este trabajo, únicamente se tomaron como referencia los code smells que se detectan a partir de las métricas definidas por Lanza y Marinescu en su catálogo.
- De la misma manera que se pueden agregar métricas en el análisis de code smells, también se podrían considerar e implementar otras métricas en la herramienta vcomplex.
- Como se encontraba fuera del alcance del trabajo, los valores de los umbrales que se utilizaron para validar las estrategias de detección, son los definidos por Lanza y Marinescu para la detección de code smells en el lenguaje Java. Por lo cual, sería interesante realizar un análisis en detalle de diferentes casos de estudio en el lenguaje JavaScript, para poder obtener valores más representativos de umbrales en dicho lenguaje.

Bibliografía

- [1] Michele Lanza, Radu Marinescu, “*Object-Oriented. Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*”, 2006.
- [2] Thomas J. McCabe, “*A complexity Measure*”, IEEE Transactions on Software Engineering, Vol. SE-2, NO.4, December 1976.
- [3] Radu Marinescu, “*Measurement and Quality in Object-Oriented Design.*”, PhD thesis, Department of Computer Science, Politehnica University of Timisoara, 2002.
- [4] ECMA International, “*Standard ECMA - 262, Language Specification*”, 6th Edition , June 2015.
- [5] D. P. Tegarden, S.D. Sheetz and D.E. Monarchi. “*Effectiveness of Traditional Metrics for Object-Oriented Systems.*” In Proceedings 25th Hawaii International Conference on System Sciences, Kauai, Hawaii, 1992
- [6] Radu Marinescu, “*Detecting Design Flaws via Metrics in Object-Oriented Systems*”, "Politehnica" University of Timisoara Department of Computer Science, Romania, August 2001.
- [7] Shyam R. Chidamber and Chris F. Kemerer, “*A metrics suite for object oriented design*”, IEEE Transactions on Software Engineering, June 1994.
- [8] James M. Bieman and Byung K. Kang. “*Cohesion and reuse in an Object-Oriented System.*” In Proceedings ACM Symposium on Software Reusability, April 1995.
- [9] Computing Curricula Series, “*Software Engineering 2004*”, Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, August 2004.
- [10] Open source HTML5 Charts for your website, <http://www.chartjs.org/>.
- [11] Node Package Manager, <https://www.npmjs.com/>.
- [12] Browserify, <http://browserify.org/>.
- [13] GitHub, <https://github.com/>.
- [14] Steven Pinker, “*How the Mind Works*”, W. W. Norton, 1997.
- [15] Wikipedia, Complejidad Ciclomática, https://es.wikipedia.org/wiki/Complejidad_ciclom%C3%A1tica.
- [16] Wikipedia, Anidamiento,

[https://es.wikipedia.org/wiki/Anidamiento_\(inform%C3%A1tica\).](https://es.wikipedia.org/wiki/Anidamiento_(inform%C3%A1tica))

[17] Common JS, <http://requirejs.org/docs/commonjs.html>.

[18] Esprima, <http://esprima.org/>

[19] Gulp, <https://gulpjs.com/>

[20] Brower, <https://bower.io/>

[21] M. M. Lehman and L.A. Belady, “*Program evolution: processes of software change*”, Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[22] M. M. Lehman and J.F. Ramil, “*An approach to a theory of software evolution*”, In Proceedings of the 4th international Workshop on Principles of Software Evolution (IWPSE '01), Vienna, Austria. ACM, 2001.

[23] IEEE. Standard 610.121990: Glossary of Software Engineering Terminology, volume 1. IEEE Press, 1999.

[24] Murphy Hill Emerson and Andrew P. Black. “*An interactive ambient visualization for code smells*”, Proceedings of the 5th international symposium on Software visualization, ACM, 2010.

[25] Sanjay Misra and Ferid Cafer, “*Estimating Quality of JavaScript*”, Department of Computer Engineering, Atilim University, Turkey, 2012.

[26] Ian Shoenberger, Mohamed Wiem Mkaouer and Marouane Kessentini, “*On the Use of Smelly Examples to Detect Code Smells in JavaScript*”, Department of Software Engineering, Rochester Institute of Technology, Rochester, USA and Department of Computer and Information Science, University of Michigan, Ann Arbor, USA, 2017.

[27] Y. Crespo, C. Lopez, R. Marticorena and E. Manso, “*Language independent metrics support towards refactoring inference*”, In 9th ECOOP Workshop on QAOOSE, volume 5, 2005.

[28] N. Moha, Y. G. Gueheneuc, L. Duchien and A. F. Le Meur, “*DECOR: A method for the specification and detection of code and design smells*”, IEEE Transactions on Software Engineering, 2010.

[29] M. J. Munro, “*Product metrics for automatic identification of “bad smell” design problems in Java source-code*”, In Proc. International Symposium Software Metrics, IEEE, 2005.

- [30] F. Simon, F. Steinbruckner and C. Lewerentz, “*Metrics based refactoring*”, In Proc European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2001.
- [31] Martin Wintz, “*Eliminating JavaScript Code Smells with OOP*”, CredibleCode, 2015.
- [32] Escomplex, <https://www.npmjs.com/package/escomplex>.
- [33] Gulp-complexity, <https://www.npmjs.com/package/gulp-complexity>.
- [34] Amin Milani Fard and Ali Mesbah, “*JSNOSE: Detecting JavaScript Code Smells*”, University of British Columbia Vancouver, BC, Canada, 2013.
- [35] Institute of Electrical and Electronics Engineers and Electronics Industry Association. Ieee/eia 12207 industry implementation of international standard iso/iec 12207 : 1995. Standard, March 1998.
- [36] I. Sommerville, “*Software Engineering*”, 7th ed. Addison Wesley, 2005.
- [37] Wang Y. and Shao J., “*A New Measure of Software Complexity Based on Cognitive Weights*”, Canadian Journal of Electrical and Computer Engineering, vol. 28, 2003.
- [38] R.S. Pressman, “*Software Engineering: A Practitioner's Approach*”, 3rd ed. McGrawHill, New York, NY, 1992.
- [39] Favre, JeanMarie, “*Languages evolve too! changing the software time scale*”, Principles of Software Evolution, Eighth International Workshop on. IEEE, 2005.
- [40] Bennett P. Lientz and E. Burton Swanson, “*Software Maintenance Management*”, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1980.
- [41] D.L. Parnas, “*Software aging*”, In Proceedings of the 16th international Conference on Software Engineering, Sorrento, Italy. IEEE Computer Society Press, 1994.
- [42] L. Bass, P. Clements and R. Kazman, “*Software Architecture in Practices*”, 2nd edition, Addison-Wesley Longman Publishing Co., Inc., 2003.
- [43] E. Gamma, R. Helm, R. Johnson and J. Vlissides, “*Design patterns - Elements of reusable object-oriented software*”, Professional Computing Series A. Wesley, 1995.
- [44] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord and J. Stafford, “*Documenting Software Architectures – Views and Beyond*”, A. Wesley, 2003.

- [45] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2005.
- [46] Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz, “*Object-Oriented Reengineering Patterns*”, Morgan Kaufmann, 2002.
- [47] M. Fowler, “*Refactoring: Improving the Design of Existing Code*”, Addison Wesley, 1999.
- [48] International Organization for Standardization, ISO 8402: 1994: Quality Management and Quality Assurance Vocabulary. International Organization for Standardization, 1994.
- [49] Van Emden, Eva and Leon Moonen, “*Java quality assurance by detecting code smells*”, Reverse Engineering, 2002. Proceedings. Ninth Working Conference on. IEEE, 2002.
- [50] Mark Lorenz and Jeff Kidd, “*Object-Oriented Software Metrics: A Practical Guide*”, Prentice-Hall, 1994.
- [51] David Flanagan, “*JavaScript: The Definitive Guide*”, 4th Edition, September 2001.
- [52] Douglas Crockford, “*JavaScript: The Good Parts*”, May 2008.
- [53] Noble, J., Taivalsaari and A., Moore, I., “*Prototype-Based Programming: Concepts, Languages and Applications*”, Springer Publishing Company, Singapore, 1999.
- [54] Tommi Mikkonen and Antero Taivalsaari, “Using JavaScript as a Real Programming Language”, October 2007.
- [55] Kris Kowal, “*Commonjs effort sets JavaScript on path for world domination*”, 2012. <https://arstechnica.com/information-technology/2009/12/commonjs-effort-sets-javascript-on-path-for-world-domination/>
- [56] Node.js, <https://nodejs.org/en/>
- [57] Javier Eguíluz Pérez, “*Introducción a JavaScript*”, 2008.