Universidad Zaragoza

Trabajo Fin de Máster

Estimación del riesgo de uso de librerías de terceros en el desarrollo de software

Estimation of the risk of using third-party libraries in software development

Autor

Freddy Hernán Martínez Quiñones

Director

Francisco Javier López Pellicer

Escuela de Ingeniería y Arquitectura

2024

Agradecimientos

A la Universidad de Zaragoza, por ofrecer un plan de estudios que permite conciliar la vida profesional y académica.

A los profesores del máster, quienes han mantenido un nivel académico alto mientras demostraban una notable comprensión de las necesidades de los estudiantes que también tienen responsabilidades profesionales.

Al profesor Javier, por su guía y orientación durante el desarrollo de este trabajo. Su acompañamiento fue esencial para definir y dar dirección a este proyecto.

A mi esposa Sara, quien me ha apoyado y acompañado en todos mis proyectos en los últimos casi diez años. Sin su paciencia y sacrificio, este logro no habría sido posible.

Resumen

El uso de librerías de terceros es una práctica común en el desarrollo de software, esta permite agilizar el proceso de construcción a costa de la pérdida de la propiedad del código. Esta falta de control conlleva riesgos que vale la pena evaluar antes de incluir una dependencia en un proyecto, los más evidentes son las posibles vulnerabilidades y exposiciones comunes (CVE's) en el código fuente, aunque existen otros riesgos menos evidentes que pueden afectar la viabilidad a largo plazo de una librería.

Por ese motivo, este trabajo aborda el desarrollo de una herramienta de apoyo en la selección de librerías de terceros, la cual permite obtener información relevante sobre las dependencias de un proyecto durante todo su ciclo de vida.

Para ello, la herramienta recopila información de los repositorios donde se aloja el código fuente y los registros dónde se publican las dependencias, compara las métricas obtenidas con los valores establecidos por el usuario para considerar una librería segura y emite mensajes de advertencia sobre los hallazgos, permitiendo así evaluar el estado de las dependencias del proyecto.

Índice general

Lista de Tablas

Lista de Figuras

1. Introducción

Una particularidad de la informática (que no se suele ver en otras ingenierías), es que para construir un producto de software, las piezas que se utilizan son también software. Clases, interfaces y funciones se agrupan para formar módulos, paquetes, librerías, frameworks y otros artefactos que se pueden publicar, compartir y reutilizar indefinidamente. Esta capacidad de reutilización permite evitar la duplicación de esfuerzos, siguiendo el conocido principio de "no re-inventar la rueda". Bajo esta premisa, cuando un desarrollador de software se enfrenta a un problema, suele ser más fácil buscar una solución existente que resolver el problema desde cero. Descargar e integrar una librería que da solución al problema se entiende como la vía más rápida y efectiva, incluso para funciones triviales [1].

Esta práctica tiene ventajas evidentes, como reducir el tiempo y los costes de desarrollo, además del aporte de flexibilidad para abordar nuevos requerimientos [2]. Sin embargo, también conlleva varios riesgos que, a largo plazo, pueden afectar negativamente al producto, al proyecto y/o al equipo. Este proyecto busca entender estos riesgos, darles visibilidad y ayudar a mitigarlos.

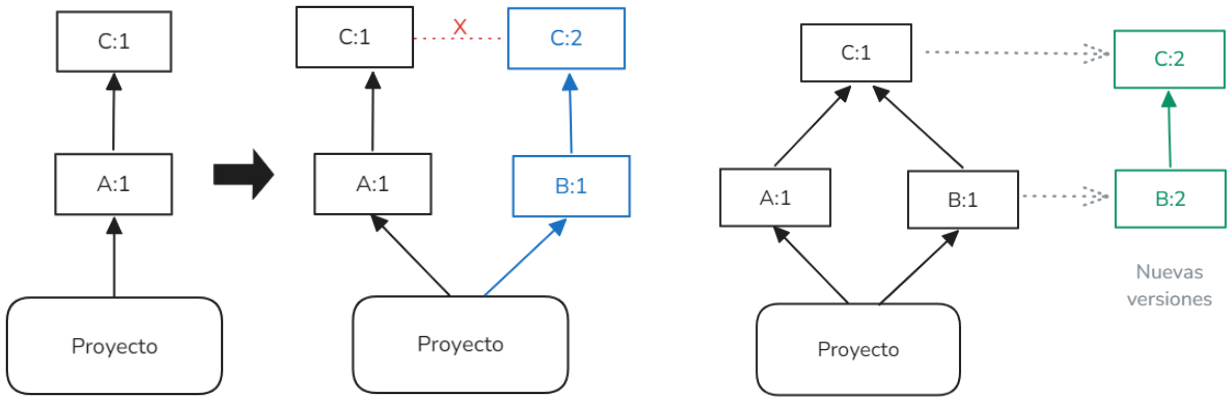
1.1. Riesgos asociados al uso de librerías de terceros

Al construir un producto de software, los desarrolladores frecuentemente recurren a librerías de terceros para agilizar el proceso. Estas librerías, conocidas como dependencias directas del proyecto, pueden incluir sus propias dependencias, que se denominan dependencias indirectas o transitivas. La cantidad de dependencias en un proyecto puede variar según el lenguaje de programación, pero en términos generales, un proyecto típico puede incluir decenas de dependencias directas y, a través de ellas, llegar a tener cientos de dependencias transitivas [3]. Estas dependencias, junto con el código propietario y cualquier herramienta utilizada para la creación y distribución del producto, pasan a hacer parte de lo que se conoce como la *cadena de suministro de software* [4].

El uso de librerías de terceros conlleva una serie de riesgos que pueden afectar tanto la calidad como la seguridad del proyecto. Estos riesgos abarcan desde conflictos de versiones entre dependencias transitivas hasta la inserción de código malicioso. Aunque muchos de estos problemas son comunes y están bien documentados, los desarrolladores no suelen actualizar ni verificar regularmente la seguridad de las dependencias en sus proyectos [5]. A continuación se exponen algunos ejemplos y se explica su relevancia:

- **Conflictos de versiones de dependencias transitivas:** ocurre cuando dos o más librerías usadas en un proyecto comparten una dependencia común, pero requieren versiones diferentes de esta [6]. Este escenario es problemático porque podría no existir una versión de la dependencia transitiva que incluya todas las características necesarias para que las demás librerías funcionen correctamente. Además, muchos gestores de paquetes no permiten la coexistencia de dos versiones diferentes de una misma dependencia en un proyecto, lo que puede provocar el mal funcionamiento de alguna de las librerías implicadas (aquellas que no disponen de la

versión correcta de su dependencia). Este tipo de conflicto se puede presentar al introducir una nueva dependencia, como se muestra en la figura 1.1a; aunque, a menudo ocurre cuando una dependencia directa se actualiza junto con sus propias dependencias, mientras que otras librerías en el proyecto que comparten esas dependencias no se actualizan. Esto puede bloquear la posibilidad de actualizar ciertas partes del proyecto, limitando la evolución del software. En la figura 1.1a se ilustra este escenario, la librería "B" no se puede actualizar hasta que la "A" tenga soporte para la última versión de la librería compartida "C".



(a) Conflicto causado por nueva dependencia directa. (b) Actualización bloqueada por conflicto.

Figura 1.1: Conflictos entre dependencias transitivas.

- **Inserción de vulnerabilidades y exposiciones comunes (CVE's):** hay dos escenarios principales en los que las dependencias pueden introducir CVE's en un proyecto. En primer lugar, un desarrollador podría incluir una librería que ya contenga vulnerabilidades conocidas; sin embargo, el escenario más común es la aparición de nuevas vulnerabilidades en librerías de terceros después de haber sido integradas en el proyecto. A través de la investigación exhaustiva y pruebas sistemáticas de diversas piezas de software, expertos en seguridad informática identifican y reportan constantemente nuevas vulnerabilidades, que pueden impactar una o varias de las dependencias de un proyecto, haciendo todo el proyecto vulnerable. Un caso muy conocido de este problema fue el descubrimiento de la vulnerabilidad `Log4Shell` de la librería `Log4j`, una librería de código abierto ampliamente utilizada en proyectos de Java [7]. Cuando este escenario se presenta, las únicas vías de solución son esperar a que los mantenedores resuelvan la vulnerabilidad y publiquen una nueva versión, o eliminar la dependencia del proyecto. Aunque en la mayoría de los casos los mantenedores corrigen las vulnerabilidades, se ha visto que estas persisten en los proyectos por falta de un adecuado manejo de las dependencias [5].
- **Eliminación de la librería de los registros públicos:** el Registro de softwares es el lugar donde se publican los paquetes y librerías listos para ser descargados y usados. Así como una persona u organización puede publicar una librería, puede también eliminarla del registro. Esto puede representar un problema grave para todos los proyectos que dependan de ella, ya que sería necesario reemplazarla o construir la funcionalidad que esta expone. Además, si la eliminación se realiza de forma repentina el proyecto puede quedar bloqueado, al no ser posible instalar dependencias, no es posible compilar ni ejecutar flujos de trabajo para el despliegue del producto. Un caso muy conocido de este escenario es el llamado "incidente de left-pad" [8], donde gran cantidad de proyectos alrededor del mundo se vieron afectados, incluso en compañías como Meta, Netflix y Spotify. Actualmente, algunos registros cuentan con políticas de eliminación de paquetes para ayudar a reducir estos escenarios.
- **Sabotaje de la librería por parte de sus mantenedores:** consiste en la introducción de código en una librería que provoca fallos o comportamientos no deseados en las aplicaciones

que dependen de ella. Esta práctica se ha hecho común recientemente como herramienta de protesta, popularizando los términos "hacktivismo" y "protestware" [9], llamando la atención de la comunidad para tratar este problema como un problema de seguridad. Un ejemplo de este tipo de protesta fue el caso de las librerías de JavaScript "colors" y "faker", cuyo propietario saboteó en 2022 a modo de protesta [10].

- **Ataque a la cadena de suministro de software:** este escenario se caracteriza por la inyección de código malicioso en un paquete de software con la intención de hacer vulnerables a los proyectos que dependen de este [11]. Este es un riesgo mucho mayor que el de la aparición de vulnerabilidades, ya que muchas veces estos ataques no son detectados por herramientas automáticas y el impacto económico puede llegar a ser enorme. En la revisión hecha por Ohm y su equipo [11] se analizan 174 paquetes con código malicioso que fueron usados en ataques reales entre noviembre de 2015 y noviembre de 2019. Un ejemplo reciente de este escenario fue el malware incorporado en la librería `node-ipc` en 2022, el cual sobrescribía los ficheros en el sistema del usuario [12], este estaba dirigido a sistemas en Rusia y Bielorrusia como acto de "hacktivismo" [9].

Estos riesgos ponen en evidencia que, para incluir una dependencia en un proyecto, no basta con que esta contenga la funcionalidad requerida, sino que el equipo de desarrollo necesita también saber que esta librería es mantenida de forma activa, y que puede confiar en su autenticidad e integridad a largo plazo.

Por otro lado, el mantenimiento continuo de las librerías implica que quienes las usan en sus proyectos deben estar al tanto de las actualizaciones, lo que lleva a otro problema común en el desarrollo de software: los equipos no mantienen las dependencias de sus proyectos actualizadas, porque perciben esta tarea como un esfuerzo extra[13]. Esta tendencia se convierte fácilmente en un círculo vicioso: a medida que un proyecto crece también lo hace su lista de dependencias, el riesgo de conflictos aumenta y mantener las dependencias actualizadas se hace cada vez más costoso, de modo que los desarrolladores se hacen más reacios a hacerlo. Es aquí donde adquiere relevancia el desarrollo de herramientas automáticas que ayuden a los desarrolladores a gestionar y mantener las dependencias de sus proyectos.

1.2. Técnicas y herramientas de detección de riesgos

A continuación se exploran algunas técnicas y herramientas que permiten gestionar y mitigar los riesgos asociados a componentes de terceros mencionados anteriormente. Para empezar, se mencionan aquellas técnicas enfocadas en la detección de vulnerabilidades y exposiciones comunes y, posteriormente, se mencionan algunas herramientas que sirven para mitigar otros de estos riesgos.

1.2.1. Detección de vulnerabilidades y exposiciones comunes

Para empezar, vale la pena mencionar la técnica `SAST` (Static Application Security Testing), que generalmente se enfoca en la detección de vulnerabilidades en el código propio del proyecto. Como el foco principal de este trabajo son los componentes externos de un proyecto, esta técnica en si misma puede resultar poco relevante. Sin embargo, sí pueden resultar interesantes aquellas herramientas que realicen análisis estático de código sobre las dependencias del proyecto, como se verá más adelante.

Por otro lado, las tecnologías `DAST` (Dynamic Application Security Testing) sí incluyen las dependencias del proyecto. Al realizar pruebas de "caja negra" simulando ataques sobre una versión funcional del sistema, pueden detectar vulnerabilidades y problemas de lógica en todas las partes que integran

el producto en tiempo de ejecución. Sin embargo, el principal problema de estas tecnologías en el contexto de este proyecto, es la incapacidad de señalar la fuente del problema detectado, lo que implica un esfuerzo adicional para determinar si el problema es del código propio o de una dependencia externa [14]. Otras desventajas de esta técnica son la dificultad para automatizar las pruebas [15] y la incapacidad de prevenir los riesgos, ya que estas herramientas se ejecutan al final del ciclo de desarrollo, cuando las vulnerabilidades ya están en producción.

Otra metodología que vale la pena mencionar es `IAST` (Interactive Application Security Testing), la cual combina técnicas de SAST y DAST para crear un mecanismo de análisis de vulnerabilidades rápido y altamente automatizado [16]. IAST ofrece grandes ventajas frente a SAST y DAST en términos de flexibilidad y velocidad, además, al tener un enfoque de "caja blanca" puede señalar en qué parte de la aplicación en ejecución se presenta una vulnerabilidad. Las principales desventajas de esta técnica son: la dificultad de implementación, las posibles incompatibilidades entre los agentes IAST y las tecnologías usadas en el proyecto y, la más importante, que el alcance de las pruebas está determinado por los casos de prueba que construyan los desarrolladores [17].

Para completar las técnicas de análisis de vulnerabilidades, se debe hablar del **Análisis de Composición de Software**. `SCA` (Software Composition Analysis) es un término general que engloba las metodologías y herramientas de seguridad que escanean los componentes de código abierto utilizados en un proyecto, con la intención de evaluar la seguridad, el cumplimiento de las licencias y la calidad del código [18]. Esta técnica adopta herramientas de SAST para analizar de forma estática el código de las dependencias del proyecto. Actualmente existe una gran oferta de herramientas con capacidades de SCA, lo que muestra la creciente preocupación de la comunidad por mantener segura la cadena de suministro de software [4]. Entre las más conocidas se puede encontrar herramientas como Snyk, OWASP dependency-check y Dependabot, que en términos generales, permiten detectar vulnerabilidades en dependencias tanto directas como transitivas, pueden ejecutar auditorías de seguridad, generar reportes y sugerir o aplicar actualizaciones para resolver estos problemas, mejorando la seguridad global del proyecto.

Un listado exhaustivo de herramientas de análisis de vulnerabilidades se encuentra en la documentación de la fundación OWASP [19], donde se agrupan según sus capacidades y las técnicas que aplican (SAST, DAST, IAST y/o SCA). Todas estas técnicas y herramientas son de gran utilidad para mantener un proyecto libre de vulnerabilidades y exposiciones comunes. Sin embargo, estas herramientas dejan de lado muchos de los riesgos mencionados anteriormente.

1.2.2. Detección de otros riesgos

Si bien existe una amplia gama de técnicas y herramientas enfocadas en la detección de vulnerabilidades y exposiciones comunes, no existe una oferta de herramientas similar que permita gestionar riesgos como el conflicto de versiones o el sabotaje de una librería. Aún así, a continuación se presentan las herramientas relacionadas que se han podido encontrar.

Algunos gestores de paquetes tienen la capacidad de señalar conflictos entre dependencias transitivas, por ejemplo, el comando npm audit permite analizar dependencias de proyectos de JavaScript, mientras que en Python es posible usar pip check. Además, estos incorporan funcionalidades de auditoría que les permiten detectar vulnerabilidades en dependencias y resolverlas (cuando es posible) actualizando versiones.

También es posible encontrar algunas iniciativas individuales que buscan proporcionar una visión más amplia sobre la gestión de dependencias. Por ejemplo, el paquete libs-inspector genera un reporte con la descripción y sugerencias de actualización de las dependencias del proyecto. Mientras que el paquete deps-updater actualiza automáticamente todos los paquetes obsoletos. Estas herramientas

son útiles para mantener las dependencias actualizadas y evitar la introducción de riesgos debido a versiones obsoletas.

Por último, un enfoque más innovador lo presenta el proyecto DEAN [20], el cual se enfoca en el análisis automático de riesgos mediante la evaluación de métricas relacionadas con los repositorios donde se aloja el código fuente, en vez del análisis estático del código. Este análisis busca una visión más amplia de los riesgos asociados con la introducción de dependencias, planteando la posibilidad de estimar algunos riesgos en base a las interacciones de los usuarios con el repositorio de un proyecto, con la intención de abordar situaciones como el abandono de ese proyecto. Fue este proyecto el que sirvió como inspiración para el desarrollo de la herramienta propuesta en este trabajo.

1.3. Objetivos

El objetivo de este proyecto es desarrollar una herramienta de código abierto que facilite a la comunidad de desarrolladores la gestión y mantenimiento de las dependencias en sus proyectos de software. Esta herramienta debe brindar información relevante sobre las dependencias, permitiendo a los usuarios tomar acciones preventivas ante riesgos que no se abordan con otras herramientas ya establecidas. Estos riesgos son el abandono de la librería, su eliminación del registro público, el sabotaje de la librería y la introducción intencionada de código malicioso (ataque a la cadena de suministro).

Para ello, se propone un enfoque basado en el análisis de métricas de la interacción humana con los proyectos, tomando como modelo la idea planteada en DEAN para establecer indicadores que permitan estimar los riesgos mencionados. Se dará prioridad a los componentes de código abierto, siguiendo la metodología de Análisis de Composición de software (SCA).

Los objetivos específicos para el desarrollo de la herramienta incluyen:

- **Definición de métricas e indicadores:** teniendo en cuenta los riesgos que se desean abordar, identificar y seleccionar un conjunto de métricas claras y cuantificables. Estas deben permitir la detección de señales (indicadores) que evidencien la presencia de estos riesgos.
- **Identificación de fuentes de datos:** investigar y seleccionar las fuentes de datos necesarias para obtener la información requerida por las métricas definidas.
- **Diseño de una arquitectura flexible y extensible:** diseñar y desarrollar una arquitectura de sistema que facilite la extensión y la incorporación de nuevas funcionalidades, como nuevos indicadores y fuentes de datos, sin requerir un rediseño significativo de la herramienta.
- **Implementación de una versión inicial de la herramienta:** construir una primera versión de la herramienta que incorpore las funcionalidades principales. Esta debe ser fácil de instalar y usar, minimizando las barreras para su adopción. Además, debe incluir una interfaz que permita a los desarrolladores obtener informes detallados y comprensibles, que les permita analizar fácilmente sus proyectos.

1.4. Alcance

El alcance inicial definido para este proyecto, siguiendo los objetivos planteados, incluye:

- La primera versión de la herramienta será implementada en JavaScript y se enfocará en dependencias de este mismo lenguaje. El soporte para otros lenguajes queda fuera del alcance

de este proyecto. Esta decisión está fundamentada en diversos motivos: en primer lugar, aprovechar la familiaridad con el lenguaje y su ecosistema; por otro lado, la gran comunidad de desarrolladores de JavaScript puede facilitar la obtención de retroalimentación con respecto a la utilidad de la herramienta; por último, los proyectos de JavaScript suelen tener una gran cantidad de dependencias, tanto directas como transitivas, comparados con otros lenguajes de programación [3].

- Integrar datos tanto del registro público donde se exponen estas dependencias como del repositorio donde se aloja el código fuente, en este caso, se habla de `npm` y `GitHub` para artefactos de JavaScript. Se escogen estas fuentes de datos porque el análisis de composición de software se enfoca en componentes de código abierto, por lo que analizar librerías publicadas en registros privados o alojadas en repositorios privados tendría poca relevancia.

- Identificar y evaluar al menos cuatro indicadores, que hagan uso de distintos datos de las fuentes mencionadas y que sean relevantes a la hora de determinar si una librería corre el riesgo de ser abandonada, saboteada o eliminada del registro público. Se determinó que esta es la cantidad mínima para demostrar que el sistema es suficientemente flexible para incorporar otros indicadores en el futuro. Además, se puede considerar una cantidad adecuada de dimensiones para tomar decisiones relacionadas con la incorporación de dependencias, mitigando el impacto de los sesgos que se puedan generar en alguna de estas dimensiones. Si se puede sacar la misma conclusión de varios indicadores, esta conclusión es más fuerte que una basada en uno solo indicador.

- La herramienta no debe requerir procesos de compilación ni la descarga de software adicional para ser usada. Para reducir en lo posible las barreras para incluir esta herramienta en el ciclo de vida de proyectos reales, esta debe estar lista para ser usada tras su descarga. Para el caso de desarrolladores de JavaScript, esto significa poder aprovechar las herramientas de desarrollo habitual: `Node.js` y `npm`.

- Se debe implementar al menos un sistema de reporte, siendo la consola de comandos la primera elección para mostrar los resultados de la evaluación de indicadores. Se decide iniciar con esta interfaz porque es un medio típico en el que muchas herramientas muestran sus hallazgos, permite mostrar información en tiempo real y se integra bien dentro de muchos procesos automáticos. Sin embargo, el sistema debe ser fácil de extender a otros medios de reporte, como la generación de archivos `HTML` o `JSON`.

1.5. Metodología

Dado que este trabajo se planteó como un proyecto de software experimental para el desarrollo del prototipo, se propuso desde el inicio el uso de una metodología iterativa, basada en pequeños ciclos de análisis, diseño, desarrollo y pruebas, lo que permite abordar el proyecto de forma incremental, utilizando como hitos los objetivos específicos mencionados anteriormente. Las metodologías iterativas de desarrollo de software permiten la mejora gradual y continua de un producto de manera práctica y eficiente [21]. Estas técnicas han demostrado ser efectivas a lo largo del tiempo [22] y resultan particularmente útiles para el desarrollo de prototipos [23], ya que permite comprender mejor diferentes aspectos del sistema, como sus requerimientos y las compensaciones entre diferentes estrategias de diseño [23], y promueve la creación de un producto fácilmente modificable [21]. Además, para efectos de este trabajo, esta práctica permite revisar y discutir los avances al final de cada iteración.

1.6. Organización de la memoria

A continuación se describe en detalle el proceso que llevó al desarrollo de la herramienta propuesta. En el capítulo 2 se explica el análisis que permitió la identificación de datos relevantes, la definición de métricas y la elaboración de indicadores para la evaluación de riesgos. Posteriormente, en el capítulo 3, se sintetiza el proceso de diseño que llevó a la definición de un software suficientemente simple y flexible, es decir, fácil de leer y de extender. En el capítulo 4 se detalla cómo se abordaron los requerimientos durante el desarrollo de la herramienta. Después, en el capítulo 5 se explica cuáles fueron las pruebas funcionales realizadas para validar la herramienta y se muestran ejemplos de los resultados obtenidos. Por último, en el capítulo 6 se explica el aporte del trabajo y se valora el cumplimiento de los objetivos.

2. Análisis

Como se explicó en el capítulo 1, la incorporación de dependencias introduce riesgos que pueden afectar la estabilidad y seguridad de un proyecto a largo plazo, por lo que se necesitan herramientas que permitan mitigar esos riesgos sin impactar negativamente el flujo de trabajo de los equipos de desarrollo. Además, la mayoría de las herramientas de Análisis de Composición de Software disponibles en el mercado se enfocan en la detección de vulnerabilidades, tal como se expone en la sección 1.2.1, dejando de lado escenarios como el abandono de una librería, el sabotaje de la misma o su eliminación del registro público.

Estos últimos escenarios son particularmente difíciles de predecir, ya que van más allá de la inspección del código fuente de las librerías. Sin embargo, existen patrones en la interacción humana con los proyectos que pueden servir como indicadores de estos riesgos. Por ejemplo, si un proyecto tiene un solo mantenedor y una base de usuarios pequeña, es mucho más propenso a ser abandonado o eliminado que si tiene un equipo de mantenimiento robusto y una comunidad de usuarios grande y activa. Este tipo de patrones es lo que este proyecto busca identificar y analizar, para poder brindar a los desarrolladores una visión más amplia del estado de las dependencias de un proyecto.

A continuación, se explican el enfoque con el que se abordó el problema de detección de riesgos, se presentan los conceptos de métricas e indicadores y se describe la relación entre estos. Posteriormente, se profundiza en el concepto de métrica y se presenta una lista de métricas candidatas para la creación de indicadores. A continuación, se habla de las fuentes de datos y de la información extraída para la creación de esas métricas. En la sección 2.4 se profundiza en los indicadores y se expone la lista de indicadores generados a partir de los datos recuperados. Después se habla del sistema de evaluación definido para esos indicadores, para presentarlos a los usuarios de forma clara y coherente. En la sección 2.7 se hace una breve mención al planteamiento de arquitectura del sistema y, por último, se enumeran los requisitos del sistema identificados durante el proceso de análisis.

2.1. Enfoque del problema

Como se mencionó anteriormente, se desea brindar información a los desarrolladores sobre los riesgos asociados a las dependencias de sus proyectos. Además, esa información debe complementar la que ofrecen las herramientas de análisis de composición de software disponibles en el mercado. Para lograrlo, se propone un análisis de las dependencias desde una perspectiva global, que permita anticipar escenarios como el abandono o el sabotaje de la librería.

Para determinar si una librería es confiable, se debe comenzar por identificar una serie de dimensiones o parámetros que permitan distinguir las librerías confiables de las que no lo son. Estas dimensiones pueden ser cuantitativas (como la frecuencia de publicación de nuevas versiones de la librería) o cualitativas (por ejemplo, si el propietario del proyecto es o no una compañía conocida), y son lo que en adelante se llamarán **métricas**. Una vez definidas las métricas, se deben identificar las fuentes

de datos que permiten extraerlas; este proceso puede ser iterativo, ya que puede ocurrir que no se encuentren datos para generar una métrica en particular, pero en su lugar se identifiquen otras métricas a partir de los datos disponibles.

Una vez se han hallado los datos y se dispone de un conjunto de métricas, se debe definir la forma de evaluar las librerías en esas dimensiones para poder identificar aquellas que suponen un riesgo alto; es aquí donde entra el concepto de **indicador**. Los indicadores representan la evaluación de la librería en una o varias de las dimensiones y deben destacar comportamientos relevantes para poder discriminar entre librerías. Teniendo los resultados de estas evaluaciones, se pueden generar reportes que permitan a los usuarios tomar decisiones informadas sobre el mantenimiento de las dependencias de su proyecto. En la figura 2.1 se ilustra el flujo de información desde la fuente de datos hasta la creación del reporte.

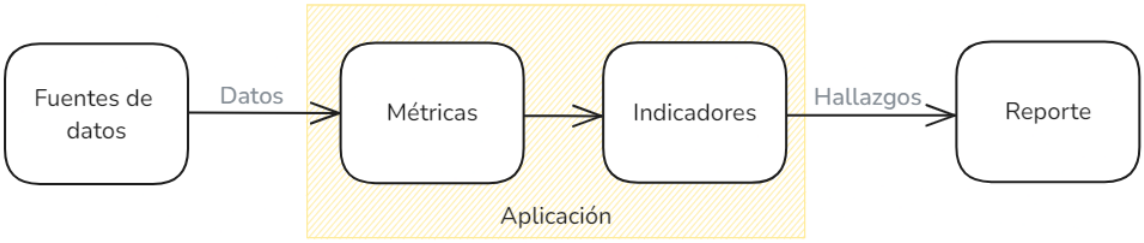


Figura 2.1: Flujo de información para la toma de decisiones.

2.2. Métricas

Las métricas corresponden a aquellas características, tanto cuantitativas como cualitativas, que se pueden utilizar para evaluar la calidad o fiabilidad de una librería. Estas se obtienen de los datos extraídos de diversas fuentes y pueden medir tanto la actividad del equipo de mantenedores como la interacción de la comunidad. Por ejemplo, a partir de los datos de *fechas de publicación de las diferentes versiones de una librería* se puede obtener la métrica *frecuencia promedio de publicación*, que permite medir la frecuencia con la que el equipo que mantiene una librería publica nuevas versiones de la misma.

Métricas como el *número de mantenedores*, la *frecuencia promedio de publicación* de nuevas versiones y la *cantidad de descargas semanales* de la librería, son cuantificables y permiten evaluar la librería en estas dimensiones. Algunas de ellas corresponden a los datos sin ningún procesamiento, mientras que otras se obtienen de hacer cálculos simples sobre los datos recogidos. Al evaluar métricas como estas, se pueden identificar patrones relevantes para detectar riesgos. Por ejemplo, un proyecto con bajo *número de mantenedores*, baja *frecuencia promedio de publicación* y pocas *descargas semanales* puede correr riesgo de abandono. Por el contrario, un proyecto con una comunidad activa puede ser un indicio de mayor capacidad de respuesta frente a desafíos como la detección de bugs o el descubrimiento de vulnerabilidades.

Por otra parte, métricas como el *tipo de propietario del repositorio* son cualitativas, pero también pueden ayudar a discriminar entre librerías. Por ejemplo, no da la misma "confianza" usar una librería desarrollada por una persona desconocida que una hecha por una empresa como Google o Meta.

En la tabla 2.1 se presentan todas las métricas que se consideraron relevantes para estimar el nivel de salud de un proyecto. En el anexo A muestra esta lista de forma más detallada, explicando la importancia de cada parámetro para la estimación de riesgos.

Tipo	Fuente	Métrica
Mantenimiento	Registro	Número de versiones
		Tiempo de vida del proyecto
		Frecuencia media de publicación de nuevas versiones
		Tiempo transcurrido desde la última publicación
		Última versión estable
	Repositorio	Número de mantenedores
		Tipo de propietario del proyecto
		Número de Issues abiertos
		Número de Issues cerrados
		Tiempo de vida de los Issues
		Número de Pull Requests activas
		Número de Pull Requests cerradas
		Tiempo de vida de las Pull Requests
		Porcentaje de salud del repositorio
Comunidad	Registro	Número de descargas semanales desde el registro
		Número de proyectos dependientes
	Repositorio	Número de estrellas en el repositorio
		Número de Forks del repositorio
		Número de observadores

Tabla 2.1: Lista de métricas candidatas.

2.3. Fuentes de datos

Una vez identificados los valores de interés, el paso a seguir fue la investigación de las fuentes de datos para obtener esta información. Tal como se define en la sección 1.4, se trabajó con npm y GitHub, siendo el primero el gestor de paquetes y el registro público más conocido y utilizado para proyectos de JavaScript; mientras que GitHub, es la plataforma de hospedaje de repositorios de código fuente más popular en la comunidad, donde se aloja una gran cantidad de proyectos de código abierto.

Ambas plataformas cuentan con API's públicas y herramientas de línea de comandos, de donde es posible extraer información de interacción de los usuarios con los proyectos. Sin embargo, no todos los datos identificados en la sección anterior están disponibles públicamente. Algunos de estos datos solo se pueden obtener a través de las aplicaciones clientes de cada plataforma, es decir, directamente en la web de npm o de GitHub (por ejemplo, la lista de proyectos dependientes de una librería). Lo que implica que haría falta aplicar técnicas de `web scraping`[1] para recuperar esta información. En

[1]Proceso que usa bots y otras herramientas automáticas para extraer contenido de páginas web.

la tabla 2.2 se detalla la información que fue posible recuperar de cada fuente y se relaciona con las métricas extraídas:

Fuente	**Dato**	**Métrica**
CLI de npm	Listado de versiones y fechas	Número de versiones
		Tiempo de vida del proyecto
		Frecuencia media de publicación de nuevas versiones
		Última versión estable
		Tiempo transcurrido desde la última publicación
API Rest de npm	Descargas semanales	Número de descargas semanales
API Rest de GitHub	Issues abiertos	Número de Issues abiertos
	Número de estrellas	Número de estrellas en el repositorio
	Número de Forks	Número de Forks del repositorio
	Número de observadores	Número de observadores
	Tipo de propietario	Tipo de propietario del proyecto
	GitHub Community Profile	Porcentaje de salud del repositorio

Tabla 2.2: Fuentes de datos y métricas extraídas.

Si bien no fue posible extraer todos los parámetros de interés mencionados en la sección 2.2, se puede decir que la información obtenida es suficientemente amplia como para elaborar indicadores relevantes en la evaluación de la salud y seguridad de un proyecto.

2.4. Indicadores

Un indicador es una afirmación sobre la librería que se confirma evaluando una o varias métricas, esta evaluación debe arrojar valores significativos que permitan saber cómo se comporta la librería en cada dimensión. Por ejemplo, el indicador **"librería se publica con frecuencia"** se determina al evaluar la métrica *"frecuencia promedio de publicación"* para determinar si la librería en cuestión se publica de forma continua. De forma similar, las métricas *"número de descargas por semana"* y *"número de estrellas en el repositorio"* pueden ser evaluadas para dar respuesta al indicador **"librería es popular"**, determinando así la percepción que tiene la comunidad de una librería.

Es importante aclarar que estos indicadores no son etiquetas definitivas que determinen de manera absoluta la seguridad o la viabilidad a largo plazo de una dependencia. Por ejemplo, una librería que no ha recibido actualizaciones en un período significativo de tiempo podría estar en riesgo de ser abandonada, o podría ser que el proyecto ha alcanzado un nivel de madurez y estabilidad suficientes para no requerir cambios frecuentes. En algunos casos, los mantenedores deciden congelar el código y limitan los cambios a la solución de problemas de seguridad, rechazando nuevas características en favor de mantener el alcance original del proyecto.

De manera similar, un proyecto con un solo mantenedor activo no necesariamente indica una alta probabilidad de abandono, el mantenedor podría ser suficientemente dedicado y capaz de sostener el proyecto en el tiempo. Por lo tanto, la evaluación de riesgos debe ser vista en contexto y complementada con un análisis de la naturaleza y el historial del proyecto.

Aún así, aunque no se pueda predecir con certeza el futuro de una librería basándose únicamente en estos indicadores, estos pueden proporcionar una visión más amplia y permitir una evaluación más completa de los riesgos del uso de librerías de terceros. De esta forma, los equipos de desarrollo pueden tomar decisiones más informadas sobre qué dependencias integrar en sus proyectos.

De las métricas expuestas en la sección 2.2, y teniendo en cuenta la información recuperada de las fuentes, se extrajeron los siguientes indicadores:

- **Librería publicada recientemente:** verificar si la última publicación de la librería se ha hecho en un lapso de tiempo definido.
- **Librería publicada frecuentemente:** evaluar si la frecuencia media de publicación de nuevas versiones es suficientemente alta.
- **Es un proyecto de larga vida:** evaluar la fecha de creación del proyecto, para saber si lleva tiempo suficiente a disposición de la comunidad.
- **Es descargada frecuentemente:** analizar la cantidad de descargas semanales desde el registro, se espera que este número sea lo más alto posible.
- **Repositorio destacado:** utilizar el número de estrellas que ha recibido el repositorio para determinar si es valorado por la comunidad.
- **Repositorio con demasiados Issues abiertos:** verificar que la cantidad de Issues sin cerrar sea bajo, de otro modo, puede indicar una baja capacidad de respuesta de los mantenedores.
- **Repositorio clonado repetidamente:** analizar la cantidad de Forks para estimar el interés que tiene la comunidad en extender el proyecto.
- **Tiene suficientes observadores:** evaluar el número de observadores, como señal de garantía y respaldo por parte de comunidad.
- **Propietario confiable:** confirmar si el tipo de propietario del repositorio es el deseado.
- **Repositorio saludable:** verifica que el porcentaje de salud del repositorio es adecuado.
- **Versión usada es la última estable:** consiste en verificar que la versión de la librería que se usa en un proyecto es la última versión estable publicada.

Estos indicadores pueden dar una visión global sobre el estado de salud de un proyecto, proporcionando indicios significativos en la toma de decisiones respecto al uso de librerías de terceros.

2.5. Evaluación de indicadores

Para permitir la valoración del riesgo a los equipos de desarrollo, los indicadores deben presentar información significativa de forma fácil de interpretar. Por este motivo, se plantearon varios enfoques de evaluación y presentación de indicadores.

La primera aproximación formulada fue usando un sistema de calificaciones para cada dimensión, de modo que cada indicador se presenta como un resultado en una escala, por ejemplo de 0 a 1. De esta forma se estandarizarían los resultados, permitiendo hacer más procesamiento sobre ellos si se

desea. Un problema de este enfoque radica en la disparidad de los datos, mientras que el número de forks de un repositorio a lo largo de toda su existencia va de cero a algunos cientos, la cantidad de descargas de una librería puede llegar a decenas de millones por semana. Como ninguno de estos valores tiene un límite superior, para normalizar habría que escoger arbitrariamente un valor máximo para cada dimensión. El segundo problema, quizá el más importante, es la interpretación que pueda dar un usuario a estos valores.

El enfoque escogido fue utilizar umbrales para evaluar cada dimensión, generando etiquetas para cada resultado. Por ejemplo, con dos umbrales se pueden obtener tres estados: `OK`, `ADVERTENCIA` y `ALERTA`, dependiendo donde se encuentre el atributo evaluado respecto a estos umbrales. Esta metodología limita la capacidad para hacer otros cálculos con los resultados de la evaluación, pero facilita la presentación de resultados, permitiendo generar reportes mucho más fáciles de interpretar.

Vale la pena mencionar que establecer los umbrales de evaluación no es una tarea trivial. Por ejemplo, al evaluar la popularidad de una librería según las descargas, no hay un número mágico que permita separar las librerías populares de las que no lo son, dependiendo de factores como el campo de aplicación, la cantidad de descargas semanales que tiene una librería considerada popular puede cambiar varios órdenes de magnitud. Por ejemplo, alguien podría decir que tanto lodash (el paquete de utilidades de JavaScript) como p5 (la versión en JavaScript de `Processing`), son igualmente populares, sin embargo, el primero tiene algo más de 5×10^7 descargas semanales, mientras que el segundo tiene apenas unas 2×10^4.

Por este motivo, se determinó que el mejor enfoque para mostrar los resultados es usar etiquetas y permitir que el usuario defina los umbrales para cada indicador. De este modo la evaluación se ejecuta de acuerdo a sus consideraciones personales y las necesidades del proyecto, mientras que las etiquetas permiten crear un reporte más limpio y fácil de leer. Así mismo, puede resultar útil ofrecer a los usuarios la capacidad de escoger los indicadores que consideren más importantes: distintos equipos pueden querer usar criterios de evaluación diferentes, por lo que evaluar solo los indicadores necesarios no solo ahorra poder computacional, sino que también contribuye a generar un reporte más enfocado.

2.6. Interfaz de usuario

En la sección 1.3 se estableció como objetivo hacer que la herramienta sea fácil de instalar y usar, con la intención de minimizar las barreras para su adopción. Con esto en mente, se consideraron dos opciones para la interacción con la herramienta: construir una interfaz gráfica (GUI) o permitir su uso mediante interfaz de línea de comandos (CLI). Ambas opciones son viables: frameworks como Electron facilitan la construcción de aplicaciones de escritorio en JavaScript, mientras que la creación de una herramienta ejecutable desde la línea de comandos se logra mediante algunos archivos de configuración.

Si bien las interfaces gráficas tienen la ventaja de ser más intuitivas para la mayoría de los usuarios finales, la interfaz de línea de comandos es una herramienta importante para los desarrolladores y administradores de sistemas [24]. Para estos usuarios, la CLI permite interacciones más eficientes y la creación de trabajo personalizados a través de scripts [25]. Además, en el contexto de este proyecto, una herramienta CLI tiene múltiples ventajas en comparación con una GUI:

- **Rendimiento y tamaño del proyecto:** una solución de tipo CLI es mucho más ligera y consume menos recursos, ya que los frameworks de JavaScript para aplicaciones de escritorio deben incorporar un navegador en la aplicación.

- **Facilidad de descarga:** una herramienta CLI se puede publicar como un paquete de JavaScript en el registro público de npm, permitiendo su descarga con un simple comando. Por otro lado, una aplicación de escritorio tendría que publicarse en plataformas como Microsoft Store o incluir el ejecutable en el repositorio del proyecto, lo que lo haría más difícil de encontrar y descargar.
- **Facilidad de integración:** las herramientas de consola de comandos son fáciles de integrar en flujos de trabajo automáticos, por ejemplo, se pueden ejecutar antes de integrar cambios en el código fuente de un proyecto o durante procesos de despliegue de una aplicación. Por este motivo, muchas herramientas de desarrollo y casi todos los proveedores de servicios en la nube ofrecen interfaces de línea de comandos [26].
- **Tiempo de desarrollo:** el uso de frameworks como Electron puede ralentizar el proceso debido a la curva de aprendizaje que requiere, además de aumentar la complejidad del desarrollo por la estructura, reglas y artefactos que aporta.
- **Riesgos asociados a dependencias:** la inclusión de un framework como Electron puede introducir riesgos adicionales, como los que este proyecto busca ayudar a mitigar.

En cuanto a la configuración de la herramienta para selección de indicadores y umbrales, como se definió en la sección 2.5, un enfoque ampliamente aceptado para herramientas CLI es el uso de archivos con formato `JSON` o `YAML`. Estos dos formatos proporcionan un mecanismo de intercambio de datos legible para humanos [27] y se usan en diversas herramientas de desarrollo; por ejemplo, Visual Studio Code, Webpack y el mismo npm usan el formato `JSON`, mientras que herramientas como Docker y Kubernetes usan `YAML`. Por lo general, estos archivos se crean en la misma ubicación donde se ejecuta la herramienta CLI o se permite indicar la ruta al archivo mediante un parámetro de ejecución.

2.7. Alternativas arquitecturales

Para definir la arquitectura del sistema, se debe partir de los objetivos planteados y las necesidades identificadas. Para empezar, uno de los objetivos propuestos en la sección 1.3 fue construir un sistema flexible, que permita extender y/o sustituir sus componentes con facilidad. Por otra parte, del análisis realizado hasta el momento se sabe que, para permitir la valoración del riesgo de una librería, el sistema debe: recuperar información de esa librería de diversas fuentes de datos (como GitHub y npm), recibir datos de configuración del usuario (para seleccionar indicadores y umbrales), evaluar los indicadores y, por último, generar el reporte con el cual el usuario podrá tomar decisiones con respecto a la librería evaluada.

A partir de estas necesidades es posible identificar los módulos principales del sistema: la entrada de datos del usuario, la entrada de datos de las librerías, el módulo central para la evaluación de los indicadores y un último módulo encargado de la salida de datos en forma de reporte. Para este proyecto resulta relevante tener flexibilidad para cambiar los módulos de entrada de datos de las librerías y de salida del reporte. Teniendo la capacidad de extender y/o reemplazar estos módulos, el sistema podría trabajar con diversas fuentes de datos para dar soporte a otros lenguajes de programación o generar reportes en formatos diversos según las necesidades del usuario.

Con esto en mente, se hace patente la necesidad de seguir un estilo de diseño con bajo acoplamiento entre componentes. Por este motivo, se consideraron las siguientes arquitecturas candidatas:

- **Arquitectura hexagonal:** también conocida como Arquitectura de Puertos y Adaptadores, es un patrón de arquitectura que busca crear sistemas desacoplados y altamente mantenibles.

La idea central es separar la lógica de negocio de las dependencias externas del sistema (como bases de datos, herramientas de pruebas, interfaz de usuario y aplicaciones externas). Para lograrlo, el sistema central interactúa con los componentes externos mediante puertos (protocolos o interfaces que definen como usar la aplicación) y adaptadores (implementaciones que satisfacen el contrato del puerto) [28]. Esto facilita la sustitución de estos componentes externos sin alterar la lógica central del sistema, facilitando la incorporación de nuevas funcionalidades.

- **Clean Architecture**: es una aproximación que intentar integrar diversas arquitecturas, como la Arquitectura Hexagonal o la llamada Onion Architecture, en una única idea viable. La idea central es crear sistemas débilmente acoplados mediante la separación de responsabilidades, separando el software en capas siguiendo lo que Robert C. Martin define como "la regla de dependencia" [29].
- **Arquitectura de Microkernels:** en ocasiones considerada como "múltiples arquitecturas hexagonales", es un modelo centrado en compartir los recursos del sistema entre varios servicios. En esta arquitectura el desarrollo se centra en el core de la aplicación, el cual debe permitir a los usuarios añadir funcionalidad mediante plug-ins. Esta arquitectura es especialmente adecuada para sistemas que requieren alta extensibilidad [30].

Estas arquitecturas ofrecen ventajas significativas en términos de modularidad, sin embargo, la elección también depende de otros factores como la complejidad de implementación. Los Microkernels, por ejemplo, requieren una infraestructura bastante compleja, que podría ser innecesaria para el alcance de este proyecto. Las arquitecturas Hexagonal y Clean Architecture son conceptualmente muy similares; de hecho, algunos autores muestran que al incorporar otros patrones de arquitectura (como MVVM y EBI) en la arquitectura Hexagonal es posible obtener implementaciones válidas de Clean Architecture [28].

Para evitar caer en el error de hacer sobre-ingeniería, se determinó que el mejor enfoque era diseñar y construir versiones simples de cada parte del sistema e ir desacoplándolas gradualmente, siguiendo los lineamientos de la arquitectura hexagonal, que es la arquitectura más sencilla que cumple con las necesidades del proyecto. Si fuera necesario, esta arquitectura podría evolucionar hacia algo más sofisticado en el futuro.

2.8. Requerimientos del sistema

Para completar el análisis, se definieron los casos de uso y los requerimientos mínimos para la versión inicial del sistema. Para que esta primera versión resultara de utilidad a sus usuarios, se plantearon dos casos de uso:

- **Análisis de una librería individual:** un usuario utiliza la herramienta para obtener información valiosa sobre una librería en particular, para determinar si es segura antes de incluirla en su proyecto.
- **Análisis de todas las dependencias de un proyecto:** un usuario incluye la herramienta como parte del proceso de desarrollo, para estar alerta ante posibles riesgos que puedan surgir en las dependencias de su proyecto. La herramienta debe poder identificar las dependencias directas del proyecto para analizarlas y generar reportes de valor para el usuario.

Teniendo presentes estos casos de uso y todo el análisis expuesto en este capítulo, se pueden definir los requerimientos del sistema. A continuación se enumeran los diferentes requerimientos, tanto funcionales como no funcionales, identificados a lo largo del proceso de análisis.

2.8.1. Requerimientos funcionales

Estos describen las funciones de la herramienta; es decir, las entradas, comportamientos y salidas que debe tener el sistema para cubrir todos los casos de uso. De las secciones 2.5 y 2.6 se sintetizan los siguientes requerimientos funcionales:

- La herramienta debe ser ejecutable mediante línea de comandos, tal como se define en la sección 2.6, y debe permitir evaluar una librería o todas las dependencias de un proyecto en una misma ejecución, según la necesidad del usuario.
- El usuario debe poder seleccionar los indicadores que desea evaluar, un usuario de la herramienta puede estar interesado en tan solo un subconjunto de los indicadores identificados anteriormente.
- El usuario debe poder establecer los umbrales para los indicadores que desea evaluar. Si bien se pueden definir valores por defecto para cada indicador, el usuario debe ser capaz de modificar esos valores cuando lo vea conveniente, para que el sistema se adapte a sus necesidades.
- El usuario debe poder configurar uno o varios mecanismos de parada para el proceso de evaluación, basados en las etiquetas definidas en la sección 2.5. La herramienta debe permitir detener el proceso de evaluación de una librería en cualquiera de los siguientes casos:
  - *Se alcanza un número de alertas:* el usuario puede definir el número máximo de alertas que puede generar una librería, al llegar a esa cantidad se detiene la evaluación para esa librería.
  - *Se alcanza un número de advertencias:* similar a las alertas, se detendría la evaluación de indicadores para cualquier librería que genere esa cantidad advertencias.
  - *Estado de indicador crítico es diferente de "ok":* el usuario puede configurar una lista de indicadores que deber dar como resultado "ok", si alguno de estos indicadores da un resultado diferente, se detiene la evaluación para esa dependencia.
  - *Estado de "alerta" en indicadores críticos:* similar a la condición anterior, el sistema debe detener la ejecución para la dependencia en evaluación si alguno de los indicadores de la lista genera una alerta.
- Se debe generar un reporte con los resultados de la evaluación, para la primera versión de la herramienta el reporte se presenta en la consola de comandos, manteniendo la misma interfaz de comunicación con el usuario. Este reporte debe tener un formato que facilite su lectura.

2.8.2. Requerimientos no funcionales

Estos requisitos, también llamados atributos de calidad del sistema, especifican características, restricciones o condiciones de funcionamiento, mantenimiento o instalación del sistema. Para la versión inicial de la herramienta, se han identificado algunos requisitos, enfocados en el rendimiento y la extensibilidad de la herramienta:

- Consultar solo los datos necesarios para los indicadores escogidos por el usuario. Si, por ejemplo, los indicadores seleccionados por un usuario no requieren información de GitHub, entonces no se debe consultar esta fuente. Esto ahorra tiempo de procesamiento y ancho de banda, además de que algunas fuentes pueden tener límites en las peticiones a sus API's públicas (como es el caso de GitHub).

- El sistema debe ser fácil de extender para incluir otras fuentes de datos, sin afectar el funcionamiento existente. De este modo se permite dar soporte a otros lenguajes de programación.
- La herramienta debe permitir la implementación de otros sistemas de reporte, diferentes al de consola de comandos, sin afectar el funcionamiento existente. De esta forma, se podrían generar reportes en archivos o enviar los resultados de la evaluación directamente a otras herramientas.
- La herramienta no debe requerir la instalación de software adicional, en cambio, debe funcionar con las herramientas básicas que suele tener un desarrollador que trabaja en JavaScript, como la herramienta de línea de comandos de npm. Esto reduce la fricción para el uso de la herramienta por parte de la comunidad de desarrollo.

3. Diseño

Como se ha mencionado anteriormente, para el desarrollo de esta herramienta se buscó elaborar un diseño flexible y extensible, a la vez que fácil de entender y mantener. Esta no es una tarea trivial, ya que la complejidad de un sistema tiende a crecer en función de la flexibilidad requerida para el mismo. Es común que ingenieros y desarrolladores hagan uso de arquitecturas sobredimensionadas o patrones de diseño innecesarios al intentar lidiar con flexibilidad. Por este motivo, se optó por empezar por un diseño global simple, que se fue diseccionando en componentes más específicos a medida que se incluía funcionalidad. Esta metodología permitió que el diseño evolucionara de manera orgánica y adaptativa, evitando inyectar complejidad accidental.

Inicialmente, se identificaron las partes clave del sistema a nivel global. Las áreas principales identificadas fueron: el **mecanismo de extracción de datos**, encargado de obtener la información de las fuentes; el sistema central o **"core" de la aplicación**, encargado de orquestar la evaluación; y el **sistema de reporte**, encargado de mostrar los resultados al usuario. Teniendo claros los módulos que componen el sistema, se puede definir la forma en que interactúan siguiendo la arquitectura hexagonal, tal como se determinó en la sección 2.7.

3.1. Arquitectura del sistema

Bajo el enfoque de la arquitectura hexagonal, el sistema central es el que contiene la lógica del negocio, mientras que los otros dos módulos se encargan de lidiar con elementos externos a la aplicación, como son las fuentes de datos y la interfaz del reporte.

Para que el sistema central pueda dirigir el flujo de trabajo, este debe poder hacer uso de los otros módulos, pero no puede depender de la implementación de estos. Afortunadamente, el principio clave de la arquitectura hexagonal es que el núcleo de la aplicación no tenga dependencias externas. Esta independencia se logra creando puertos que definen la forma en que el sistema central interactúa con los componentes externos, junto con adaptadores específicos para los componentes de extracción de datos o de generación de reportes. En la figura 3.1 se ilustran los tres módulos principales del sistema siguiendo el patrón de arquitectura hexagonal.

El punto de inicio y la configuración de la aplicación son detalles que no se especifican en la arquitectura hexagonal [28], ya que son detalles de implementación más que decisiones de diseño. Estos detalles, junto con la entrada de datos del usuario, se ilustran en la figura 3.1 como información de arranque (Bootstrapping) de la aplicación.

A continuación se profundiza en el detalle de cada uno de los tres módulos principales identificados, señalando las interfaces que cumplen el papel de puertos siguiendo el estilo de arquitectura escogido.

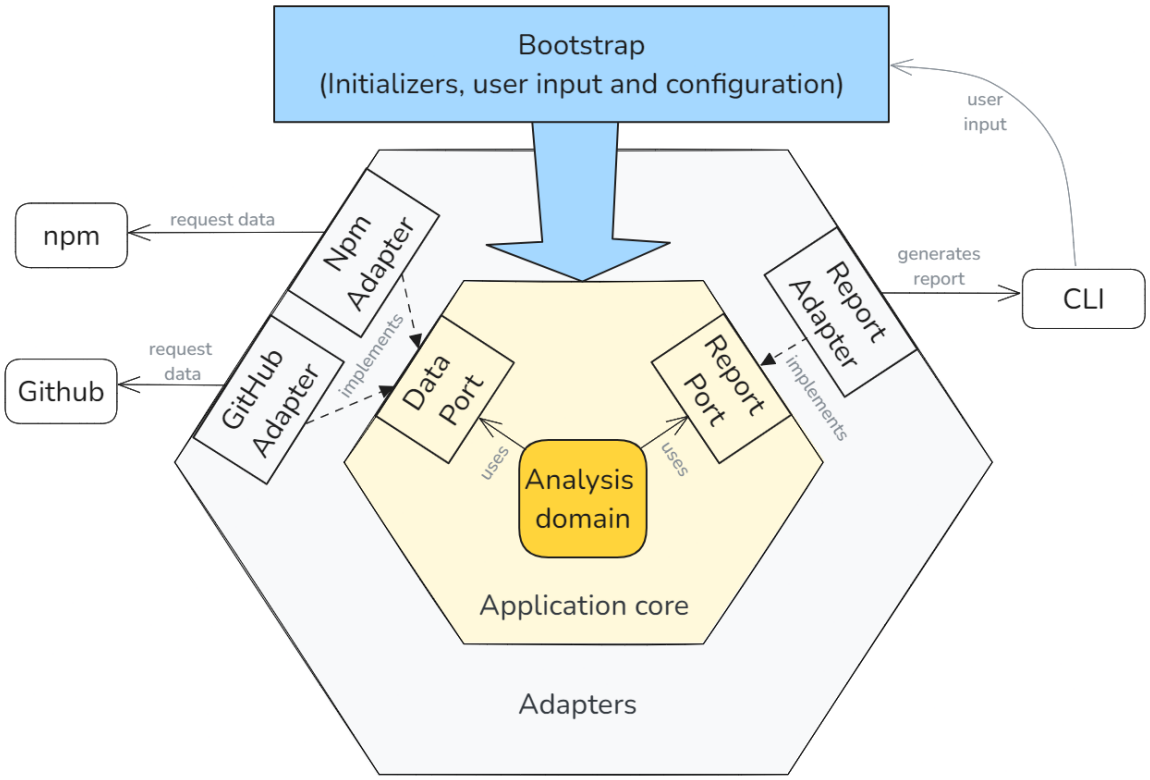


Figura 3.1: Componentes principales del sistema bajo el enfoque de la arquitectura hexagonal.

3.2. Sistema central

Como su nombre lo indica, es el módulo principal de la herramienta, responsable de orquestar el proceso de evaluación de indicadores. Para permitir al usuario seleccionar y configurar los indicadores de evaluación, se definieron tres piezas que componen esta parte de la aplicación: el indicador, el registro y el ejecutor.

3.2.1. Interfaz Indicador

Para empezar, se definió una interfaz común para todos los indicadores. Cada indicador definido en la sección 2.4 corresponde a una clase que implementa esta interfaz, de modo que es posible interactuar con cualquiera de ellos de forma indistinta. En la figura 3.2 se ilustra la interfaz con algunas implementaciones concretas.

Cada indicador implementa un método llamado "evaluar", que recibe una instancia de la `Librería` y retorna el resultado de la evaluación. La evaluación de cada indicador se realiza comparando uno o varios atributos de la librería con los umbrales definidos para el mismo; mientras que el resultado incluye un estado ("ok", "advertencia" o "alerta", como se explicó en la sección 2.5) y un mensaje explicativo asociado al estado.

El indicador expone la lista de propiedades que requiere de la Librería para ejecutar la evaluación, lo que permite consultarlos mediante el sistema de extracción de datos en caso de no tenerlos. Además, cada indicador tiene valores por defecto para los umbrales, pero también cuentan con un método que permite sobrescribir estos valores (`setThresholds`), de este modo se puede configurar los indicadores según el criterio o necesidades del usuario.

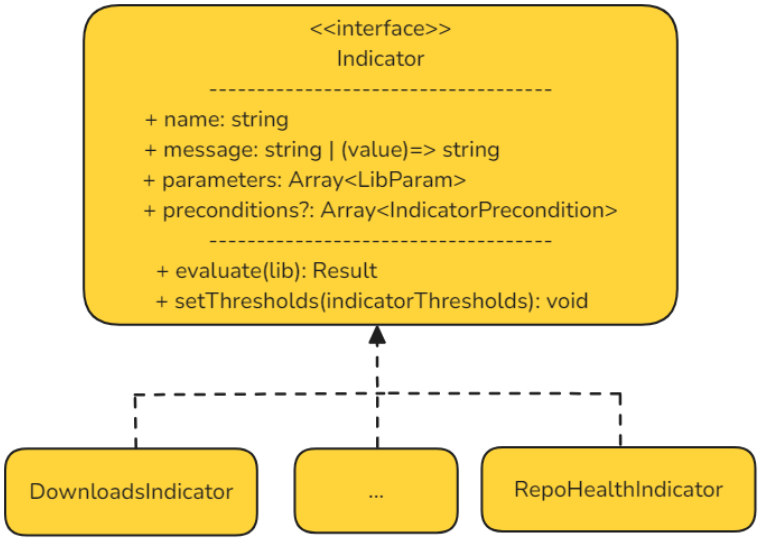


Figura 3.2: Interfaz "Indicador" con ejemplos de implementaciones concretas.

Adicionalmente, los indicadores pueden tener pre-condiciones, que permiten determinar si la evaluación de un indicador debe llevarse a cabo según los resultados de otros indicadores previamente evaluados. Por ejemplo, un indicador que analiza datos de GitHub solo puede ser evaluado si se ha confirmado previamente que la dependencia tiene un repositorio público asociado.

3.2.2. Componente de Registro

Este componente se creó para gestionar la lista de indicadores. Este utiliza una estructura de tipo diccionario (vector asociativo) para mapear cada indicador con su identificador único, lo que facilita su búsqueda al momento de verificar parámetros, confirmar pre-condiciones o evaluar el indicador.

El registro es también el responsable de confirmar que las pre-condiciones se cumplan antes de la evaluación de un indicador, verificando los resultados de indicadores precedentes o ejecutando su evaluación cuando es necesario. Además gestiona la configuración definida por el usuario, limitando la evaluación a los indicadores escogidos y sobrescribiendo los umbrales de estos indicadores cuando el usuario suministra estos datos.

Adicionalmente, el sistema permite al usuario definir condiciones de parada para la evaluación de indicadores de una dependencia. Estas condiciones de parada aseguran que, si se cumple alguno de los criterios, la evaluación se detiene para evitar procesamiento innecesario. La comprobación de si alguna condición de parada se cumple también la hace el registro tras obtener el resultado de cada indicador.

Como se puede ver, este componente es una pieza clave en la ejecución de la herramienta. En la figura 3.3 se pueden observar los métodos expuestos por la clase Registro para cumplir las funcionalidades descritas.

3.2.3. Componente Ejecutor

Se puede decir que el ejecutor es la pieza principal del sistema, ya que es el responsable orquestar la evaluación. Este recibe como argumentos: la lista de librerías que se van a evaluar, una instancia del `BuilderDirector` y la instancia del registro. En la figura 3.3 se observa cómo se integra el ejecutor con los demás componentes de esta parte del sistema.

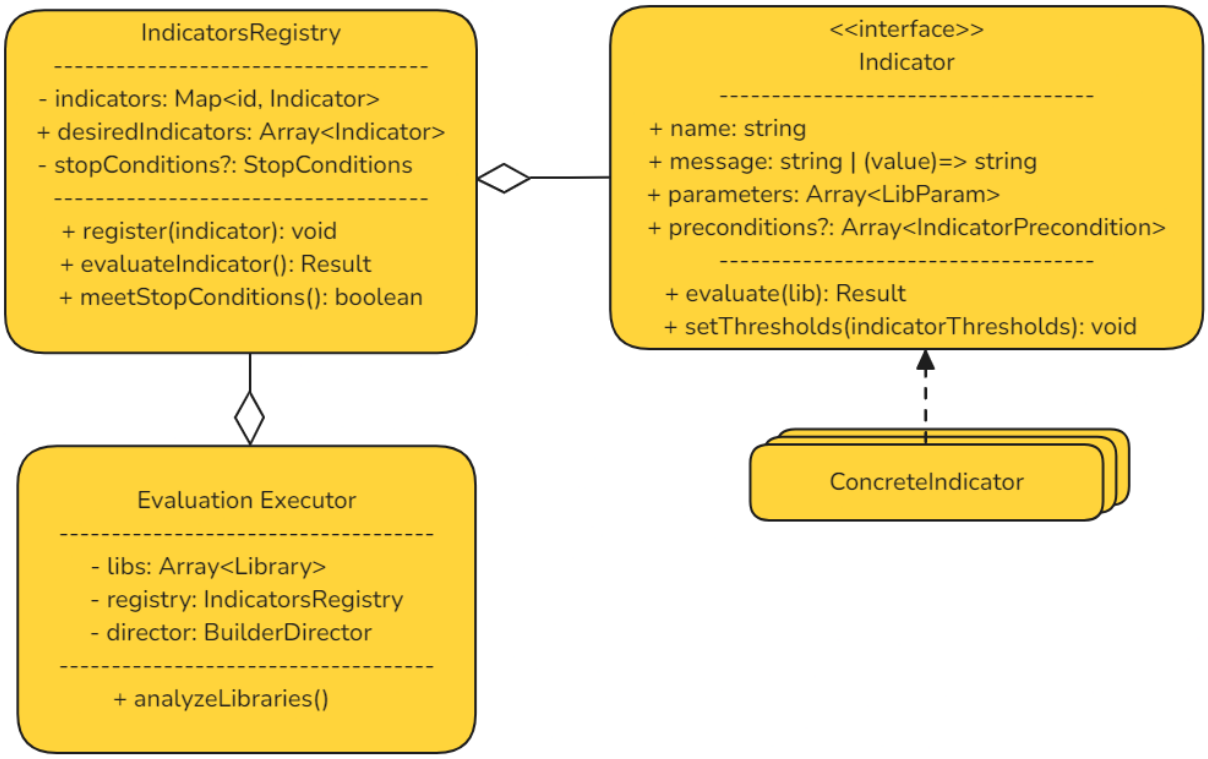


Figura 3.3: Sistema central de la herramienta, encargado de orquestar la evaluación.

Para cada librería en la lista, el ejecutor debe evaluar uno a uno los indicadores escogidos hasta cumplir una condición de parada o completar todos los indicadores. Para evaluar un indicador, el ejecutor utiliza el registro para comprobar las propiedades que requiere de la librería para evaluar ese indicador, si la instancia de la librería no tiene esa propiedad, delega en el director la tarea de obtenerla de la fuente de datos correspondiente. Cabe señalar que al consultar una fuente se llenan todos los atributos obtenibles de esa fuente, no solo el atributo específico requerido, de ese modo una fuente solo se debe consultar una vez por cada librería. Una vez ha sido obtenida la información necesaria, el ejecutor hace uso del registro para efectuar la evaluación del indicador y, posteriormente, confirmar si se cumple alguna condición de parada.

3.3. Mecanismo de extracción de datos

Como su nombre lo indica, esta parte de la aplicación se encarga de recuperar la información requerida para la evaluación de indicadores. Este módulo debe ser capaz de producir un objeto cuyos atributos correspondan con las métricas que requiere el sistema central para ejecutar la evaluación. Además, como se ha mencionado anteriormente, se debe poder reemplazar con facilidad los elementos que lo componen, para poder incorporar otras fuentes de datos en el futuro.

3.3.1. Componente Builder

Para poder extraer datos de diferentes fuentes, se aplicó un enfoque similar al del patrón de diseño Builder, creando una clase responsable de la obtención de datos por cada fuente. Todas ellas deben cumplir con el mismo contrato, permitiendo reemplazar una implementación por otra dependiendo de la fuente de datos que se desea utilizar. La diferencia de este enfoque con el patrón Builder radica en que cada implementación no crea una instancia diferente de un objeto con una interfaz común, sino

que varios Builders añaden diferentes partes a una misma instancia compartida, el objeto `Librería`. Por ejemplo, una instancia de Builder construida para extraer datos de npm (`NpmBuilder`) puede trabajar en conjunto con una implementación enfocada en GitHub (`GithubBuilder`) para completar los datos de una librería de JavaScript. Esta primera aproximación se ilustra en la figura 3.4.

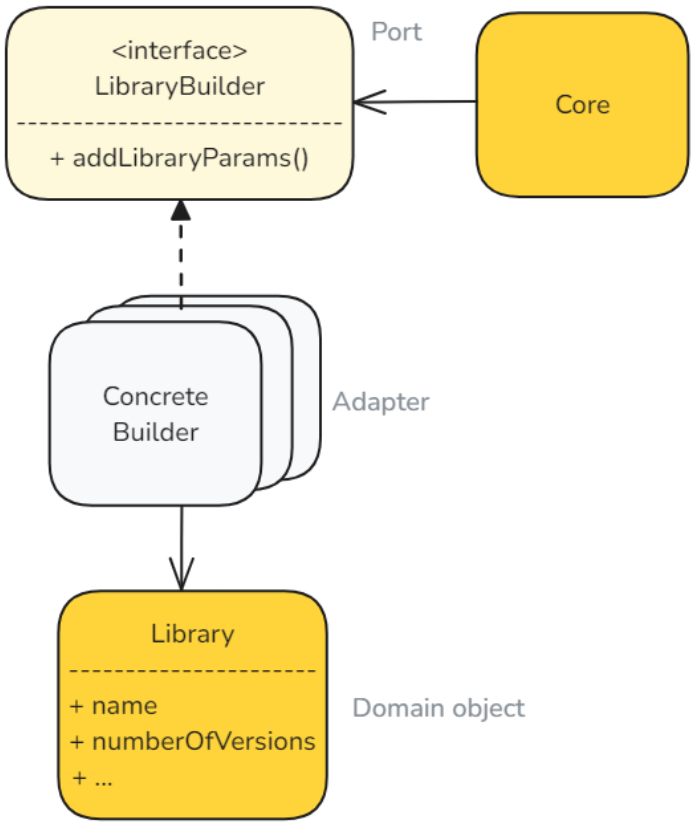


Figura 3.4: Primera aproximación del mecanismo de extracción de datos.

En este caso, la interfaz Builder funciona como puerto, para que los componentes del core puedan interactuar con los adaptadores (las diferentes implementaciones de builder) sin conocer los detalles de su implementación. Aún así, en la figura 3.4 hace falta una pieza que permita saber relacionar las implementaciones de diferentes adaptadores con los lenguajes de programación a los que se da soporte.

3.3.2. Componente Director

Para permitir la extensión a otros lenguajes de programación, con sus respectivas fuentes de datos, se agregó al diseño la interfaz `BuilderDirector`. Además, este elemento da la capacidad de recuperar datos bajo demanda (como se definió en los requerimientos no funcionales 2.8). Las clases que implementan esta interfaz relacionan cada parámetro de la librería con la fuente de datos de donde se extrae; asimismo, relacionan cada fuente de datos con el Builder responsable de consultarla. De este modo, una instancia de `BuilderDirector` puede dar soporte a un lenguaje de programación y consultar cada fuente independientemente, utilizando los Builders que tiene asignados. En la figura 3.5 se observa la estructura de este sistema.

Este enfoque permite extender el mecanismo fácilmente a otros lenguajes de programación: para dar soporte a un nuevo lenguaje, solo se necesita crear los Builders correspondientes para sus fuentes de datos y el BuilderDirector que los administra, cumpliendo con las interfaces definidas. De este modo, el usuario puede escoger el lenguaje de programación al inicio de la ejecución, se instancia el

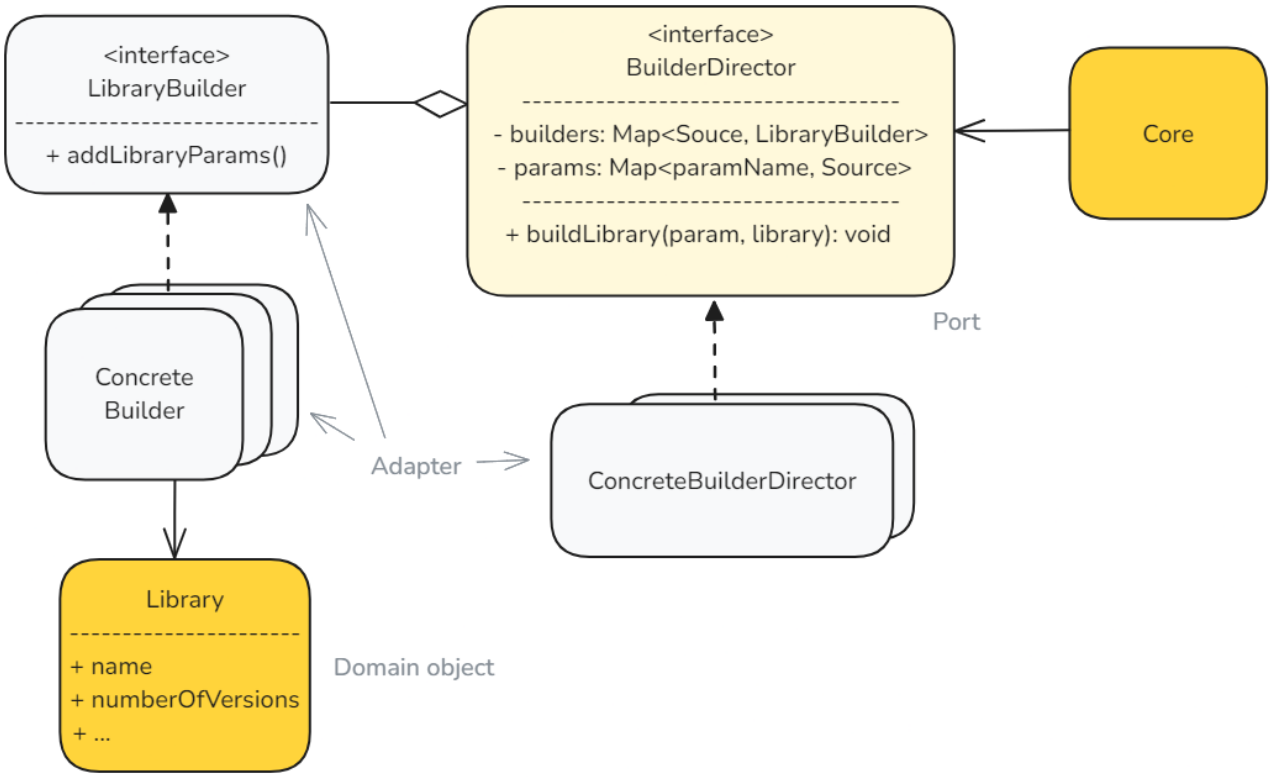


Figura 3.5: Mecanismo de extracción de datos.

director correspondiente y se le asignan los Builders necesarios para trabajar con las fuentes de datos del lenguaje escogido, luego se agrega la instancia del director al sistema central para que haga uso de estos componentes. Este proceso es transparente para el sistema central, que interactúa con los diferentes directores a través de una interfaz común. De esta manera, el sistema está preparado para incorporar nuevas fuentes de datos y lenguajes de programación con un esfuerzo mínimo.

Con este nuevo diseño se puede decir que la interfaz BuilderDirector actúa como puerto, creando una capa de abstracción adicional en la arquitectura. En este caso, el adaptador de extracción de datos (para un lenguaje de programación específico) es todo el conjunto integrado por un director concreto y sus Builders.

3.4. Sistema de reporte

Como su nombre lo indica, es el encargado de presentar los resultados de la evaluación al usuario. Para presentar un reporte de forma estructurada y ordenada, los resultados de los indicadores para cada dependencia se almacenan en estructuras de tipo diccionario dentro de la clase `ResultsStore`. De este modo se evitan problemas de condiciones de carrera si la evaluación se ejecuta de forma concurrente.

Además, se definió una interfaz llamada "Contexto", la cual pretende dar soporte a diferentes tipos de reportes. Las clases que implementen esta interfaz deben exponer un método para recibir los resultados y generar el reporte correspondiente. La interfaz Contexto actúa como puerto, permitiendo al core interactuar con diferentes adaptadores de reporte. En la figura 3.6 se observan los componentes descritos.

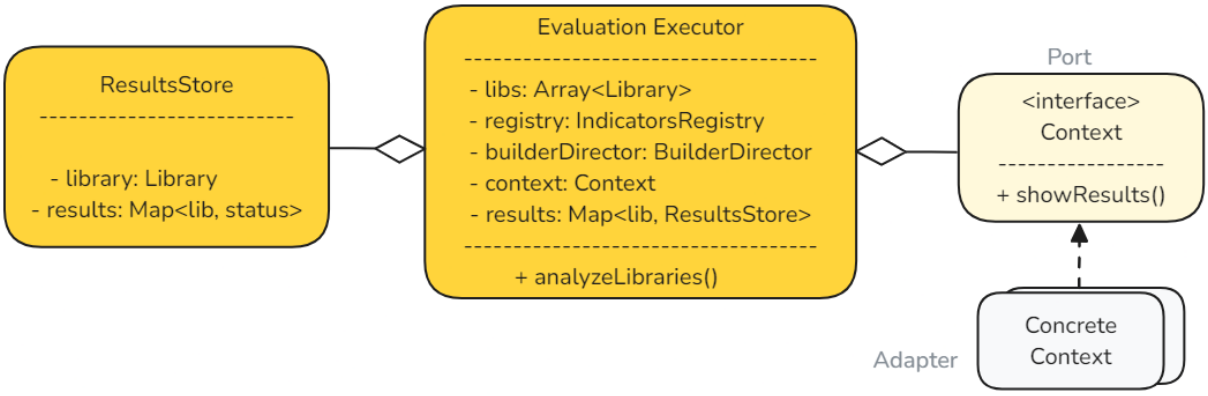


Figura 3.6: Componentes involucrados en la generación de reportes.

Este diseño permite extender el sistema a nuevos formatos de reporte, de modo que el usuario pueda escoger el deseado al iniciar la ejecución. Se crea la instancia del adaptador según la elección del usuario y el ejecutor interactúa con cualquiera de ellas de forma indistinta.

3.5. Resultado final

En la figura 3.7 se ilustra el diseño global de la aplicación, obtenido integrando los componentes principales explicados a lo largo de esta sección. Además de mostrar la relación entre componentes según la arquitectura hexagonal, en este diagrama se señala el proceso en el que intervienen. Cabe aclarar que hay numerosos componentes auxiliares que no se muestran en este diagrama, estos intervienen en diferentes partes flujo del programa ayudando en la creación de instancias, el mapeo de datos, la ejecución de comandos y peticiones http, entre otras funciones; estos elementos serán mencionados más adelante en los detalles de implementación.

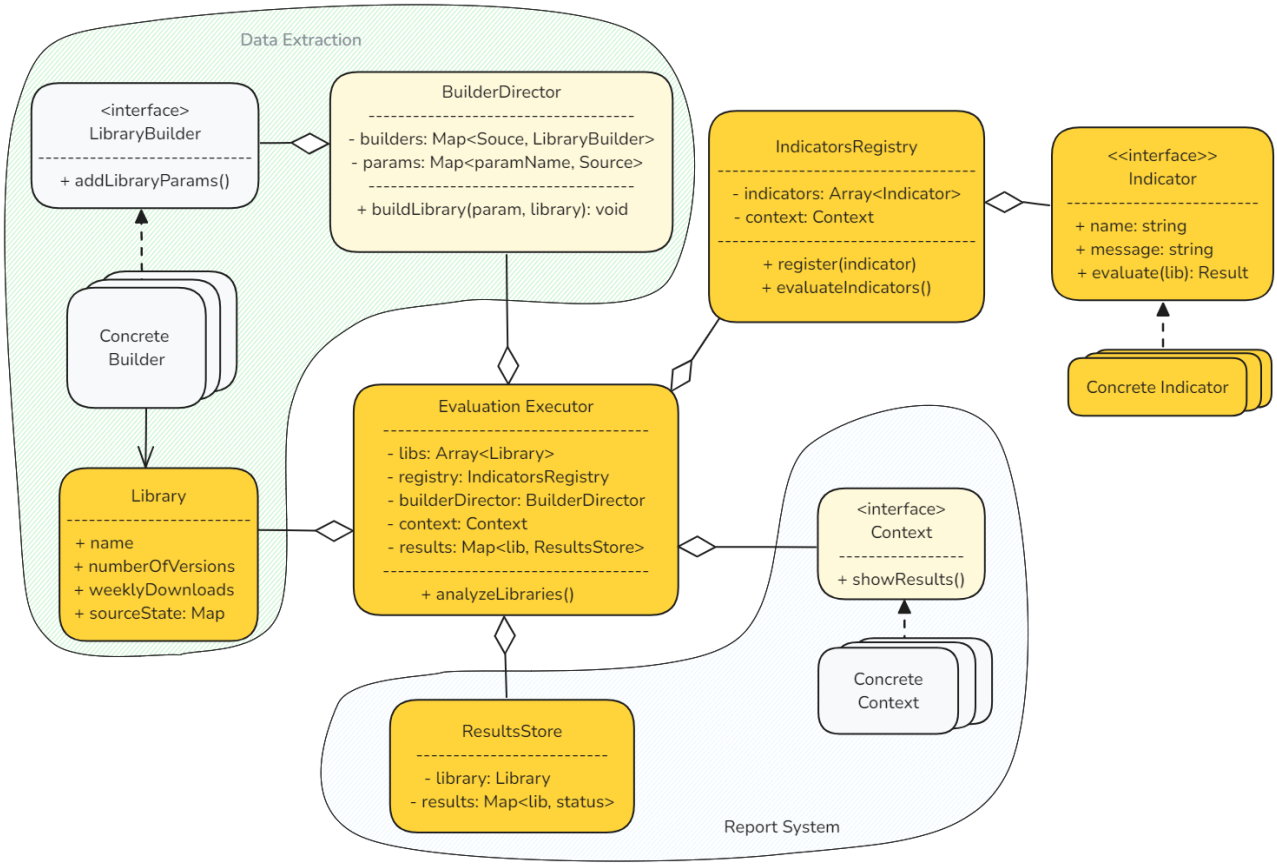


Figura 3.7: Interacción entre los componentes principales de la aplicación.

4. Desarrollo

En esta sección se describe el proceso de construcción de la herramienta, detallando la implementación de los elementos principales del sistema descritos en el capítulo 3 y añadiendo algunos otros detalles importantes para el funcionamiento de la herramienta. Al final de este capítulo se hace mención a las pruebas unitarias y otras buenas prácticas de desarrollo que se intentaron seguir durante el proceso de construcción de la herramienta.

4.1. Sistema central

En la sección D se explicaron los componentes principales que integran este módulo, sin embargo, vale la pena mencionar otros elementos que intervienen en el control del flujo de la aplicación y dan soporte a esos componentes principales. Estos son algunos de los detalles mencionados en la sección 3.5 que no cubre la descripción de arquitectura.

Para empezar, se debe mencionar la función encargada de la interacción mediante interfaz de línea de comandos. Esta se ejecuta al inicio del flujo para obtener las opciones de ejecución escogidas por el usuario, como el lenguaje de programación o el formato del reporte.

Además, vale la pena recordar que en la sección 2.8 se definieron dos casos de uso para la herramienta: analizar una sola librería o la lista completa de dependencias de un proyecto. En el primer caso, el usuario debe proporcionar el nombre y la versión de la librería; en el segundo caso, la herramienta busca y lee el archivo `package.json` para extraer las dependencias del proyecto, existe una función encargada de esta tarea. En el anexo B se muestran más detalles de esta función y de como extender la funcionalidad a otros lenguajes.

Una vez se tiene la lista de dependencias con sus versiones, se crean tantas instancias de Librería como corresponda. El objeto `Librería` es un modelo de dominio que se ha mencionado en el capítulo de diseño y que vale la pena analizar en más detalle. En el fragmento de código 1 se observa la implementación del modelo que se hizo para la primera versión de la herramienta; esta se ha hecho incluyendo explícitamente los atributos que usan los indicadores implementados, se hizo así con la intención de aprovechar las funcionalidades de comprobación de tipos que brinda `Typescript`, aunque se podría cambiar en el futuro para añadir atributos de forma dinámica.

Los primeros tres atributos del objeto (líneas 2 a 4) son los requeridos para la identificación y manipulación de cada instancia, los dos primeros corresponden al nombre y la versión de la librería que se desea analizar, mientras que el tercero es un vector asociativo que permite saber cuáles fuentes han sido consultadas anteriormente. Los atributos de las líneas siete y ocho corresponden al nombre del repositorio y el nombre del propietario; estos se obtienen de npm y permiten consultar en GitHub. El resto de atributos corresponden a las métricas con las cuales se va a realizar la evaluación.

```typescript
export type Library = {
  name: string;
  usedVersion: string;
  sourceStatus: Map<string, LibSourceStatus>;
} & Partial<{
  // Properties from npm
  repoName: string;
  repoOwner: string;
  numberOfVersions: number;
  weeklyDownloads: number;
  lastVersion: string;
  lastVersionDate: Date;
  lifeSpan: number;
  releaseFrequency: number;
  // Properties from github
  repoOpenIssues: number;
  repoStars: number;
  repoForks: number;
  repoObservers: number;
  repoOwnerType: string;
  repoHealth: number;
}>;
```

Listing 1: Definición del modelo Librería

Después de crear las instancias de la librería, se crea el registro de indicadores explicado en la sección 3.2.2. Para ello se utilizan un par de funciones auxiliares, encargadas de crear la instancia del registro, obtener el archivo de configuración del usuario, registrar los indicadores y establecer los valores obtenidos del archivo de configuración. El archivo de configuración debe tener el nombre `scout.config.json` y es completamente opcional, en caso de usarse le permite al usuario escoger los indicadores que le resultan de interés, definir los umbrales que considera apropiados para cada indicador e incluso definir las condiciones de parada para la evaluación de una librería. En caso de que el usuario no proporcione el archivo, el sistema usa la lista de indicadores con los umbrales definidos por defecto. En el bloque de código 2 se muestra un ejemplo de este archivo de configuración, en este se observa la lista de indicadores deseados (líneas 3 a 13), el umbral que el usuario desea modificar (líneas 16 a 19) y las condiciones de parada (líneas 22 y 23).

Tras crear el registro, se crea todo el mecanismo de extracción de datos, seguido del contexto de reporte y el ejecutor. Este último recibe las instancias del registro, el director, la colección de librerías que se van a evaluar y el contexto para mostrar los resultados, tal como se explica en la sección D. Luego, el hilo de ejecución inicia el proceso de evaluación mediante el método "analizar librerías" del ejecutor.

Para optimizar el proceso de evaluación de múltiples librerías, este se realiza de forma secuencial dentro de los indicadores de una librería, pero de forma concurrente entre librerías; es decir, el sistema no espera a terminar el análisis de una librería para empezar la siguiente. De esta forma, el sistema puede estar evaluando los indicadores de una librería mientras espera la respuesta de un servicio externo para obtener los datos de otra librería.

```
{
  "indicators": [
    "is-last-version",
    "was-released-recently",
    "is-released-frequently",
    "is-downloaded-frequently",
    "is-long-living-project",
    "is-starred-repo",
    "has-open-issues",
    "has-forks",
    "has-enough-observers",
    "repo-owner-type",
    "is-healthy-repo"
  ],
  "thresholds": {
    "is-released-frequently": {
      "warningThreshold": 30,
      "alertThreshold": 90
    }
  },
  "conditions": {
    "mustBeOk": ["was-released-recently"],
    "maxAlerts": 2
  }
}
```

Listing 2: Ejemplo de configuración usando el archivo `scout.config.json`

4.2. Mecanismo de extracción de datos

Su desarrollo inició de forma paralela al análisis, ya que fue necesario experimentar con las fuentes de datos para aprender a interactuar con ellas y determinar cual era información disponible. Inicialmente se crearon funciones simples para extraer datos mediante la herramienta de línea de comandos de npm. Posteriormente, se crearon funciones para realizar peticiones HTTP tanto a la API de GitHub como a la de npm. Los datos obtenidos son los que se listan en la sección 2.3.

Después de construir los servicios, es decir, las funciones encargadas de extraer datos sin procesar de cada fuente, se construyeron lo Builders que se describen en la sección 3.3.1. Para dar soporte a JavaScript se construyeron en total 4 de estas clases, uno por cada fuente: línea de comandos de npm, API de npm, API de GitHub y API para el perfil de la comunidad de GitHub. Cada Builder usa un servicio para extraer los datos que le corresponden y procesarlos, si es necesario, para obtener las métricas, que se almacenan como atributos del objeto Librería.

Por último se construyó el Director, para hacer uso de los Builders como se explica en la sección 3.3.2. En el anexo B se muestra como se instancia un Director y se explica qué componentes se deben añadir para dar soporte a otros lenguajes.

4.3. Sistema de reporte

Se comenzó por mostrar los resultados de evaluación y otros mensajes directamente en consola de comandos a medida que se iban generando. Sin embargo, este enfoque tenía dos problemas evidentes: la imposibilidad de extender a otras formas de reporte y la incompatibilidad con la ejecución concurrente de la evaluación. La implementación de los componentes `ResultsStore` y `Context`, expuestos en la sección 3.4, resolvió estos problemas.

Para esta primera versión se implementaron dos versiones del contexto: la primera muestra los resultados en la consola de comandos, tal como se definió en los objetivos iniciales del proyecto, mientras que la segunda permite guardarlos en un archivo con formato HTML. Este segundo adaptador se hizo con la intención de demostrar la flexibilidad del sistema.

Cuando el usuario inicia el programa, se le pide seleccionar el tipo de reporte deseado. El sistema crea una instancia de la clase de reporte escogida y hace uso de esta para presentar los resultados al terminar la evaluación. En el anexo C se explica como crear nuevas implementaciones del contexto para crear otros tipos de reporte, por ejemplo, en formato JSON.

4.4. Buenas prácticas de desarrollo

La herramienta se desarrolló como un proyecto de ámbito profesional, siguiendo lo que comúnmente se conoce como buenas prácticas de desarrollo de software. Entre otras cosas, se aplicaron los principios de diseño de software[1], se siguieron las convenciones de nombres de TypeScript[2] y se integró ESLint para el análisis estático de código, garantizando así un código libre de errores comunes y acorde a las mejores prácticas de estilo de código.

Además, se utilizó Git como sistema de control de versiones para gestionar el código fuente del proyecto, este se encuentra en un repositorio público en GitHub. Git permite hacer un seguimiento detallado de los cambios realizados durante el desarrollo, lo que facilita la identificación y corrección de errores. También se adoptó la especificación de "Conventional Commits", la cual ayuda a mantener la coherencia de los mensajes de `commit`[3] para tener un historial de cambios claro y comprensible.

El proyecto también incluyó pruebas unitarias, implementadas con Jest, para asegurar la funcionalidad de los componentes individuales y facilitar el mantenimiento del código a lo largo del tiempo. Esta práctica está diseñada para mantener el proyecto libre de errores a medida que este crece y evoluciona.

Aunque en un principio se consideró el uso de la metodología "Test Driven Development"[4], este enfoque se hizo difícil de aplicar por la naturaleza experimental de la herramienta. A pesar de ello, se procuró mantener una cobertura de pruebas unitarias razonable, como se muestra en la figura 4.1, para garantizar la fiabilidad y estabilidad del código.

[1] Los principios de diseño son un conjunto de directrices que se siguen para escribir código limpio, mantenible y extensible. La mayoría de ellos usan acrónimos como regla mnemotécnica, entre los más conocidos están `SOLID`, `DRY` y `YAGNI`.

[2] Es la nomenclatura recomendada por Google para TypeScript, esta promueve la consistencia y la legibilidad del código.

[3] Un commit es una operación que guarda el estado de un proyecto en un momento específico, junto con un mensaje descriptivo que explica los cambios introducidos.

[4] El desarrollo guiado por pruebas es una práctica de desarrollo en la la pruebas se escriben primero con la intención de producir código de más calidad en menos tiempo.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	82.91	75.6	86.66	83.1	
core/executor	63.46	62.5	68.75	61.22	
executor.ts	55.88	50	85.71	54.54	43-60,71-76
resultsStore.ts	77.77	100	55.55	75	20,28-32,46
core/indicators	100	100	100	100	
indicators.types.ts	100	100	100	100	
core/registry	93.33	80.76	100	97.43	
registry.ts	93.33	80.76	100	97.43	118
models	76.47	0	100	76.47	
builderDirector.ts	76.47	0	100	76.47	20,23,32-33
util	97.36	100	92.85	97.29	
asyncExec.ts	100	100	100	100	
constants.ts	92.3	100	50	91.66	15
httpGet.ts	100	100	100	100	
readFileAsync.ts	100	100	100	100	

Figura 4.1: Reporte de la cobertura de pruebas unitarias de la herramienta.

5. Pruebas funcionales de la herramienta

En este capítulo se detalla el proceso de validación funcional de la herramienta desarrollada. El objetivo de estas pruebas era garantizar que la herramienta funciona correctamente en distintos escenarios y cumple con los requisitos establecidos (sección 2.8.1). Para ello, se realizaron pruebas en varios entornos y con diferentes usuarios, asegurando que la herramienta se comporta de manera fiable y que tiene el impacto deseado en los proyectos.

El capítulo se divide en cuatro secciones: en la primera, se describen las pruebas locales realizadas durante el desarrollo para verificar el correcto funcionamiento de la herramienta en un entorno controlado; además, se explica el proceso de instalación y uso de la misma. La segunda sección aborda la validación de la herramienta con proyectos reales, evaluando su impacto en situaciones de uso práctico. En la tercera sección se analiza el desempeño de la herramienta comparando su impacto con los de otras soluciones existentes en el mercado, verificando su capacidad para complementar y mejorar los análisis de composición de software. Finalmente, la última sección describe la validación realizada con usuarios externos, recogiendo sus comentarios y ajustando la herramienta para mejorar la experiencia del usuario.

5.1. Pruebas locales y uso de la herramienta

Durante el desarrollo, se realizaron pruebas locales para evaluar las funcionalidades del proyecto durante cada iteración. Este se puede compilar y ejecutar localmente, teniendo la posibilidad de evaluar sus propias dependencias, como se observa en la figura 5.1. En esta figura se muestra un ejemplo del reporte en consola, una de las dos opciones de reporte mencionadas anteriormente, donde se observan los mensajes de alerta (color amarillo), advertencia (color rojo) e incluso condiciones de paradas (color violeta) generadas para cada dependencia.

En la sección 1.4 se indica que la herramienta debe ser fácil de descargar y usar, esta debe ser ejecutable desde la consola de comandos aprovechando las herramientas habituales de un desarrollador de JavaScript. Se definió de este modo con el objetivo de reducir barreras en el uso y llegar a la mayor cantidad de usuarios posibles. Por este motivo, la herramienta se publicó como un paquete de JavaScript en el registro público de npm. El único requisito para descargarla y ejecutarla es tener instalados `Node.js` y `npm`, preferiblemente una versión con soporte activo.

La instalación de la herramienta se puede realizar tanto en un proyecto específico como de forma global, siendo la segunda opción la recomendada. Para instalar el paquete de forma global se utiliza el comando `npm i -g deps-scout`. Una vez instalado, solo se debe ejecutar el comando `scout` para iniciar la interacción mediante línea de comandos.

Al iniciar, el programa solicita al usuario seleccionar el lenguaje de programación, como se ve en la figura 5.2a. A continuación, se solicita escoger el formato del reporte, como se muestra en la figura 5.2b. Cabe recordar que actualmente la herramienta solo tiene soporte para JavaScript, los

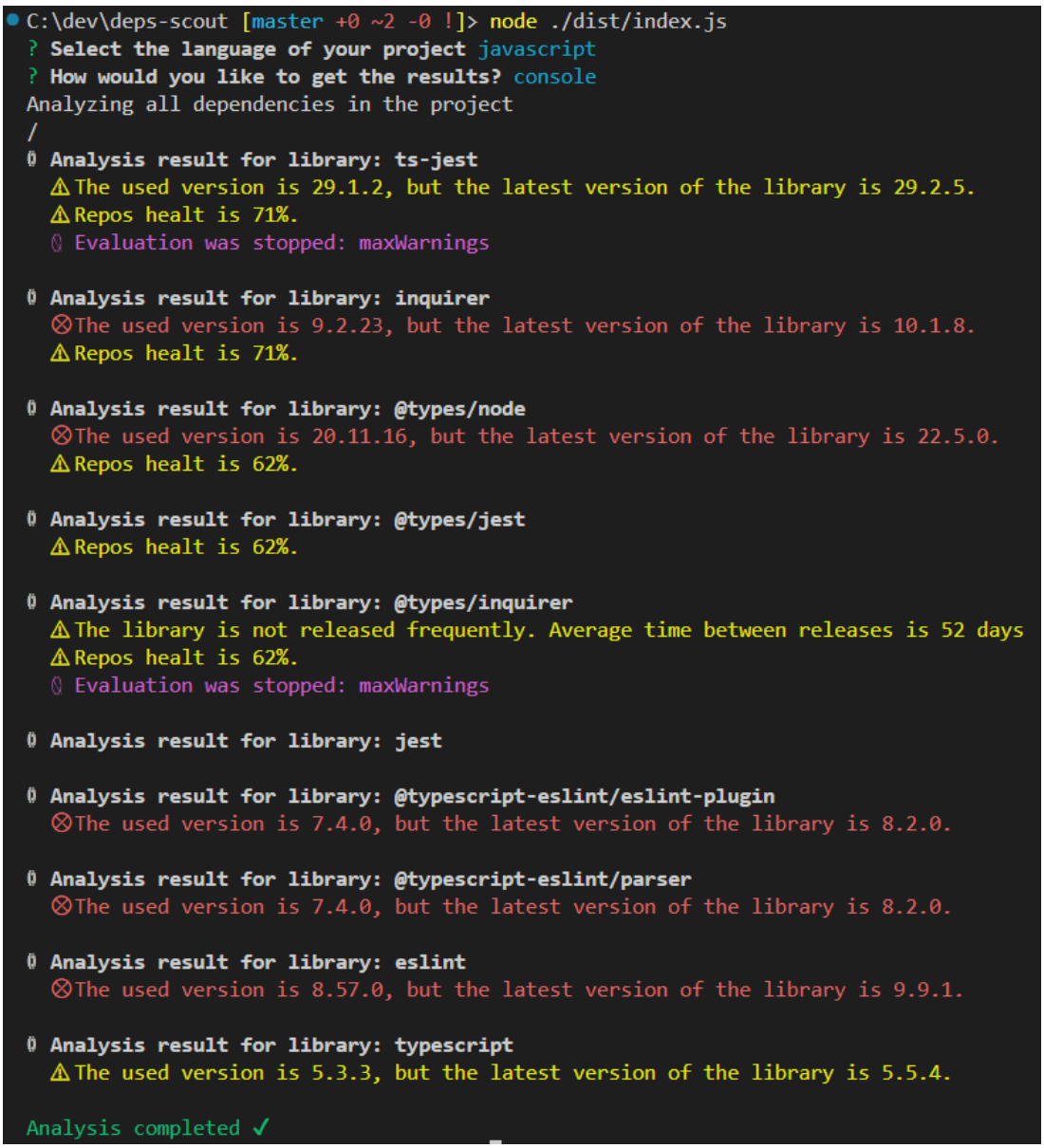
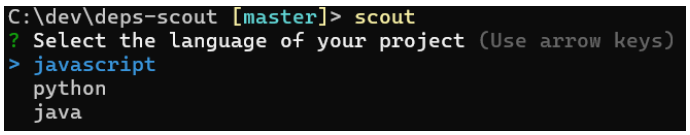
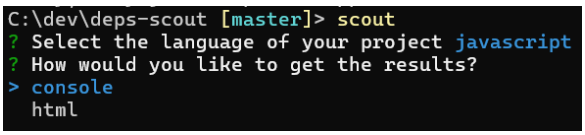


Figura 5.1: Reporte en consola ejecutando el proyecto durante desarrollo.

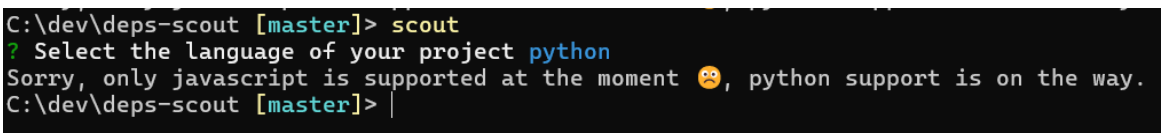
otros lenguajes que se muestran son solo marcadores (para dar soporte en el futuro), si el usuario selecciona un lenguaje no soportado se muestra un mensaje de error, como se ve en la figura 5.2c.



(a) Selección de lenguaje.



(b) Selección de tipo de reporte.



(c) Mensaje de error al seleccionar un lenguaje no soportado.

Figura 5.2: Interacción con la herramienta.

5.2. Validación con otros proyectos

Para validar la utilidad de la herramienta, además de las pruebas realizadas después de incluir cada funcionalidad, se diseñaron algunos experimentos enfocados en verificar que la herramienta se comporta de la forma esperada y que tiene un impacto positivo en los proyectos donde se usa. Para ello, se ejecutó la herramienta en diversos proyectos tanto personales como empresariales, intentando detectar librerías con alto riesgo de abandono, eliminación o sabotaje.

En las pruebas fue posible identificar librerías con pocas descargas, estrellas y observadores, o con muchos Issues y Pull Requests sin atender, lo que puede ser indicador de alto riesgo y vale la pena analizar en profundidad. Por ejemplo, en la figura 5.3 se observa el resultado del análisis de la librería nodemailer-stub, encontrada en un proyecto personal, la cual obtuvo varios mensajes de alerta. Al indagar en el registro y el repositorio, se encontró que esta tenía pocos observadores, estrellas y descargas, a la vez que mucho tiempo sin ser actualizada, por lo que se convirtió en candidata para ser reemplazada.

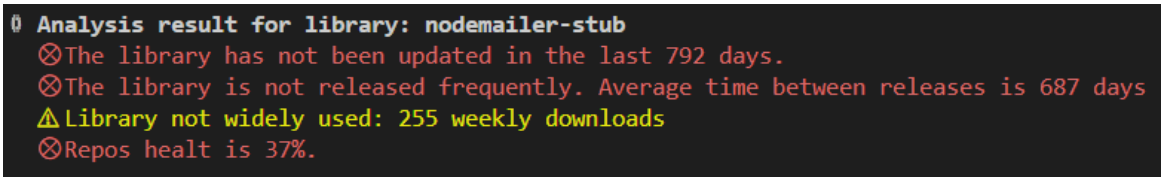


Figura 5.3: Ejemplo de librería con pocas descargas y mucho tiempo sin actualizar.

Además de las múltiples librerías con mucho tiempo sin modificaciones, se llegó a encontrar una librería abandonada hace más de 5 años en uno de los proyectos empresariales analizados. La librería en cuestión es ng-simple-slideshow y el proyecto donde se encontró es una aplicación web que utiliza Angular como framework de desarrollo. En la imagen 5.4 se observa el reporte generado para esta librería.

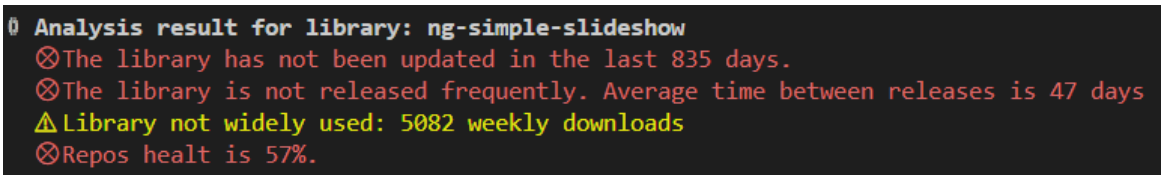


Figura 5.4: Librería abandonada por los mantenedores.

Si bien en la imagen se señala que la librería no ha sido actualizada en algo más de dos años, al revisar en detalle la información que ofrece npm se encontró que la última versión estable había sido publicada en 2019, lo que reveló que la fecha actualización no necesariamente coincide con la última publicación. Además, al revisar la documentación de la librería se encontró en su última versión estaba diseñada para trabajar con las versiones 4 a 7 de Angular. Sin embargo, la última versión estable de Angular a la fecha de ejecutar la prueba era la 17, dejando en evidencia un problema de bloqueo por conflicto de dependencias (como se ilustra en la figura 1.1b). Tras ese descubrimiento, se sustituyó esta librería por una con soporte vigente y se actualizaron las dependencias bloqueadas, como Angular.

5.3. Validación con otras herramientas de SCA

Uno de los indicadores que resultó tener más impacto de lo esperado fue el de **"última versión"**, este alerta al usuario cuando la versión que está usando de una dependencia no es la versión estable más reciente. Este indicador permite a los desarrolladores mantener actualizadas las dependencias de sus proyectos, incluyendo así ajustes a posibles errores o vulnerabilidades corregidas en esas dependencias.

Para validar el impacto de esta funcionalidad, se compararon los resultados del análisis de otras herramientas de SCA antes y después de aplicar las actualizaciones sugeridas por la herramienta. Por ejemplo, se utilizaron tanto el comando de auditoría de npm como la extensión de Red Hat Dependency Analytics para buscar vulnerabilidades comunes antes y después de actualizar o reemplazar las librerías señaladas por la herramienta. Por ejemplo, en las figuras 5.5 y 5.6 se observan los resultados del análisis de las tres herramientas antes y después de actualizar las dependencias señaladas.

En la figura 5.5a se observa parte del reporte generado por la herramienta, en donde se señalan algunas dependencias del proyecto que deben ser actualizadas. Al ejecutar las herramientas de auditoría en este mismo proyecto, estas advirtieron de la presencia de 3 vulnerabilidades en la cadena de suministro, como se observa en las figuras 5.5b y 5.5c. Con esta información, se procedió a actualizar las librerías señaladas, para posteriormente volver a ejecutar las tres herramientas. En la figura 5.6 se observa que tras realizar las actualizaciones, se reducen los riesgos por vulnerabilidades conocidas señalados por las otras herramientas.

5.4. Validación con otros usuarios

Una vez publicada una versión estable de la herramienta con las funcionalidades básicas definidas en el alcance, esta se compartió en diferentes medios con otros desarrolladores y equipos de trabajo, para que fuera probada y valorada de forma práctica, en busca de retroalimentación que permita identificar fallos y posibles mejoras. Estas pruebas son importantes para confirmar la utilidad y relevancia de la herramienta para los usuarios finales.

La recepción inicial parece haber sido positiva, la facilidad con la que se descarga y ejecuta la herramienta permitieron que muchas personas la probaran, aunque no muchas dieron realimentación que resultara de utilidad para seguir mejorándola. Los comentarios recibidos se enfocaron en mejorar dos aspectos: la documentación, explicando como usar la herramienta, y el reporte, para el cual se sugería utilizar un formato más fácil de visualizar y donde los resultados quedaran guardados. En cuanto a la documentación, la versión actualizada se puede observar en el repositorio del proyecto, mientras que para el reporte, se desarrolló la versión en formato HTML.

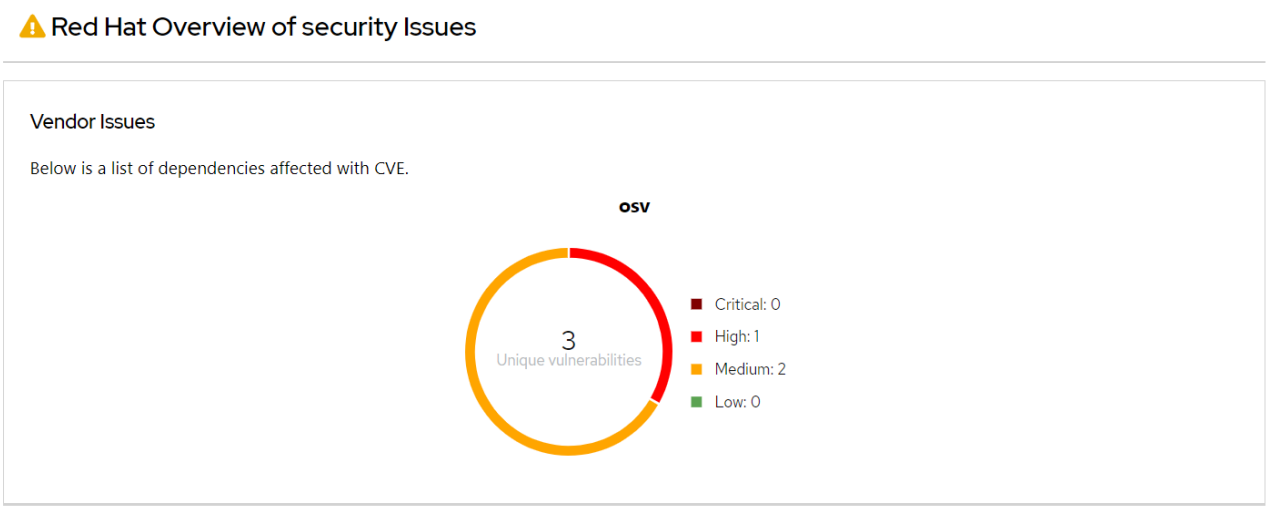
En la figura 5.7 se muestra este otro tipo de reporte. El archivo HTML se guarda en la ruta donde se ejecuta la herramienta. En este se observan las dependencias evaluadas como filas de la tabla, mientras que los indicadores evaluados se ven como columnas de la misma. Los resultados para cada indicador y dependencia están codificados por colores, lo que permite al usuario identificar rápidamente aquellos que han resultado en "alerta" o "advertencia".

```
Analysis result for library: express
  ⚠ The used version is 4.18.3, but the latest version of the library is 4.19.2.
  ⚠ The library is not released frequently. Average time between releases is 18 days

Analysis result for library: supertest
  ⊗ The used version is 6.3.4, but the latest version of the library is 7.0.0.
  ⊗ The library is not released frequently. Average time between releases is 64 days
```

(a) Mensajes de advertencia generados por la herramienta.

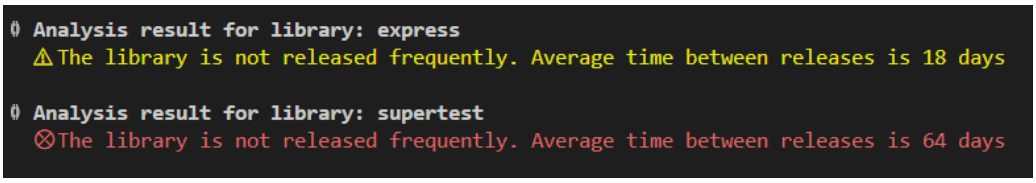
```
C:\dev\NodeJs_Learning\UdemyNodeTDD [master ↑1 +1 ~1 -0 | +0 ~2 -0 !]> npm audit
# npm audit report

braces  <3.0.3
Severity: high
Uncontrolled resource consumption in braces - https://github.com/advisories/GHSA-grv7-fg5c-xmjg
fix available via `npm audit fix`
node_modules/braces

express  <4.19.2
Severity: moderate
Express.js Open Redirect in malformed URLs - https://github.com/advisories/GHSA-rv95-896h-c2vc
fix available via `npm audit fix`
node_modules/express

tar  <6.2.1
Severity: moderate
Denial of service while parsing a tar file due to lack of folders count validation - https://github.com/advisories/GHSA-f5x3-32g6-xq36
fix available via `npm audit fix`
node_modules/tar

3 vulnerabilities (2 moderate, 1 high)
```

(b) Resultado del análisis de vulnerabilidades de npm.

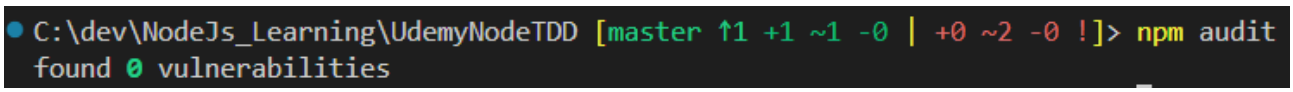


(c) Resultado del análisis de vulnerabilidades de la herramienta de Red Hat.

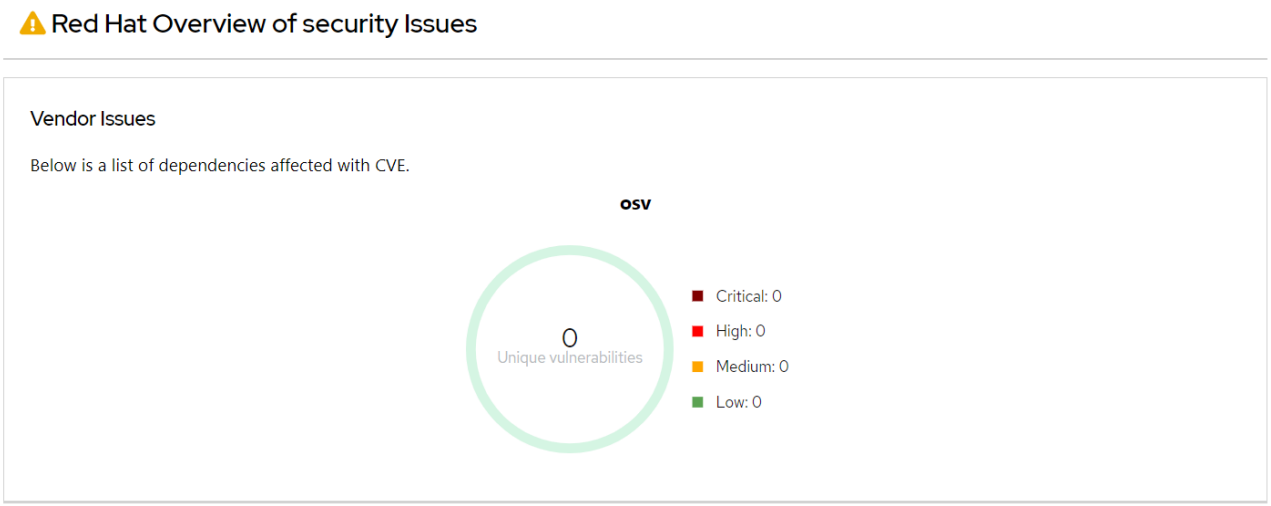
Figura 5.5: Resultados iniciales del análisis de las tres herramientas SCA.



(a) Mensajes de advertencia generados por la herramienta en un proyecto personal.



(b) Resultado del análisis de vulnerabilidades de npm.



(c) Resultado del análisis de vulnerabilidades de la herramienta de Red Hat.

Figura 5.6: Resultados del análisis tras la actualización de las dependencias del proyecto.

Scout Results

Dependency \ Indicators	is-last-version	is-same-major-version	is-same-minor-version	was-released-recently	is-released-frequently	is-downloaded-frequently	is-long-living-project	is-starred-repo	has-open-issues	has-forks	has-enough-observers	repo-owner-type	is-healthy-repo	Stop reason
@types/jest	Ok	-	-	The library has not been updated in the last 97 days.	-	-	-	-	-	-	-	-	-	Indicator was-released-recently must be ok
@types/inquirer	Ok	-	-	The library has not been updated in the last 97 days.	-	-	-	-	-	-	-	-	-	Indicator was-released-recently must be ok
ts-jest	The used version is not the latest released version of the library.	Ok	Ok	Ok	Ok	Ok	Ok	Ok	Repository has 88 open issues.	Ok	Projects repository has 34 observers.	Repo owner type is 'User'.	Repos healt is 71%.	-
@types/node	The used version is not the latest released version of the library.	Ok	The minor version of the used library is different from the latest version.	Ok	Ok	Ok	Ok	Ok	Repository has 705 open issues.	Ok	Ok	Ok	Repos healt is 62%.	-
inquirer	Ok	-	-	Ok	Ok	Ok	Ok	Ok	Repository has 188 open issues.	Ok	Ok	Repo owner type is 'User'.	Repos healt is 71%.	-
eslint	The used version is not the latest released version of the library.	The used major version is 8, but the latest version of the library is 9.	-	Ok	Ok	Ok	Ok	Ok	Repository has 77 open issues.	Ok	Ok	Ok	Ok	-
@typescript-eslint/eslint-plugin	The used version is not the latest released version of the library.	Ok	The minor version of the used library is different from the latest version.	Ok	Ok	Ok	Ok	Ok	Repository has 433 open issues.	Ok	Ok	Ok	Ok	-
@typescript-eslint/parser	The used version is not the latest released version of the library.	Ok	The minor version of the used library is different from the latest version.	Ok	Ok	Ok	Ok	Ok	Repository has 433 open issues.	Ok	Ok	Ok	Ok	-
jest	Ok	-	-	Ok	Ok	Ok	Ok	Ok	Repository has 333 open issues.	Ok	Ok	Ok	Ok	-
typescript	The used version is not the latest released version of the library.	Ok	The minor version of the used library is different from the latest version.	Ok	Ok	Ok	Ok	Ok	Repository has 5789 open issues.	Ok	Ok	Ok	Ok	-

Figura 5.7: Reporte en formato de tabla en HTML.

6. Conclusiones

El mantenimiento de las dependencias de un proyecto es tan importante como el mantenimiento del código fuente. Aún así, a menudo los desarrolladores solo prestan atención a sus dependencias en dos momentos clave: cuando las integran en el proyecto y cuando surgen problemas. Esta práctica puede llevar a riesgos significativos, ya que las dependencias pueden evolucionar, ser abandonadas o volverse inseguras con el tiempo. Por este motivo, es necesario el desarrollo de herramientas que faciliten el mantenimiento y la gestión de dependencias a lo largo de todo el ciclo de vida de los proyectos, necesidad que intenta abordar la herramienta desarrollada en este proyecto.

A continuación se valora el cumplimiento de objetivos, se exponen las posibilidades de continuación del trabajo y, para finalizar, se realiza una reflexión personal sobre el proceso de desarrollo y el impacto de la herramienta obtenida.

6.1. Objetivos alcanzados

Se desarrolló y publicó un paquete de JavaScript que está disponible para su descarga desde el registro público de npm bajo el nombre deps-scout. Tal como se muestra en las pruebas del capítulo 5, esta herramienta permite analizar las dependencias de proyectos en JavaScript, evaluando diversos indicadores relacionados con el mantenimiento y la interacción de la comunidad entorno a dichas dependencias. Al considerar estos aspectos, "deps-scout" genera reportes que brindan información relevante para el mantenimiento y la seguridad de los proyectos, complementando la información proporcionada por otras herramientas de Análisis de Composición de Software (SCA), que suelen centrarse en la detección de CVE's.

La herramienta es configurable, lo que permite a los usuarios personalizar el análisis según su criterio y necesidades. En particular, los usuarios pueden seleccionar qué indicadores desean evaluar, ajustar los valores con los cuales se evalúa cada uno de estos indicadores, e incluso establecer condiciones de parada para el análisis de cada dependencia (siguiendo el ejemplo de configuración 2 de la sección 4.1). A la fecha de escribir este documento, la herramienta cuenta con 11 indicadores, que se encuentran documentados tanto en el repositorio del proyecto como en el registro de npm.

Otra característica destacable de esta herramienta es la facilidad con la que se puede integrar en el flujo de trabajo de los equipos de desarrollo. Su fácil instalación y uso permiten que sea adoptada rápidamente sin impactar los procesos existentes, ofreciendo valor inmediato con un esfuerzo mínimo, permitiendo a los desarrolladores centrarse en la construcción de nuevas funcionalidades. Esto quedó demostrado al realizar las pruebas con otros usuarios, comentadas en la sección 5.4.

Además, la herramienta ha sido diseñada con una arquitectura flexible, lo que facilita la incorporación de nuevos formatos de reporte, así como dar soporte a otros lenguajes de programación y fuentes de datos adicionales. De igual forma, se pueden incluir nuevos indicadores, tal como se detalla en el anexo D.

6.2. Trabajo futuro

La arquitectura flexible de la herramienta presenta oportunidades significativas para su futura expansión y adaptación. Su diseño no solo permite dar soporte para otros lenguajes de programación con sus respectivos gestores de paquetes y fuentes datos, sino que también abre la puerta a diversos usos adicionales. Por ejemplo, los datos recopilados por la herramienta podrían servir como base para otros análisis, estudios de investigación o para desarrollar nuevas funcionalidades que aprovechen estos datos en contextos distintos.

Por otra parte, aunque la herramienta ha sido bien recibida por los desarrolladores que la han probado, es importante reconocer que aún está en las etapas iniciales de su adopción. Como se menciona en la sección 5.4, la realimentación por parte de los usuarios es fundamental para que la herramienta alcance todo su potencial. A medida que más equipos comiencen a usarla y compartir sus experiencias, se abrirán oportunidades para refinar y expandir sus características. La retroalimentación de la comunidad será esencial para la evolución de la herramienta, asegurando que sea relevante y útil en un entorno de desarrollo de software que está en constante cambio.

6.3. Reflexión personal

En términos generales, me siento muy satisfecho con el desarrollo y el resultado de este proyecto. Dado que está estrechamente relacionado con mi área profesional, considero que la inversión de tiempo y esfuerzo ha sido muy valiosa (ver Anexo E) para detalles del cronograma de tareas). El proceso de análisis me permitió no solo ampliar mi visión sobre la problemática de la gestión de dependencias, sino también tomar mayor conciencia de los riesgos asociados a la cadena de suministro de software. La comprensión de estos riesgos y la manera en que se pueden mitigar es un aprendizaje valioso que ahora intento compartir con otros desarrolladores e ingenieros. Además, ahora cuento con una herramienta que pienso utilizar regularmente para verificar que mis proyectos están seguros y evitar que se degraden con el paso del tiempo.

Por otro lado, el proceso de diseño y construcción me permitió profundizar en conceptos de diseño de software que considero fundamentales para mi crecimiento profesional. Además, el enfoque global que se le dio al proyecto me permitió recordar y reafirmar la importancia de abordar el software como la ingeniería que es, algo que contrasta con la realidad de muchos proyectos, donde los desarrolladores toman decisiones basados en tendencias, opiniones o anécdotas antes que en experimentos y datos reales.

En cuanto a la herramienta, considero que tiene un gran potencial y puede ser un complemento valioso para la gestión de dependencias de software. Esta puede trabajar en conjunto con otras herramientas en busca de mantener segura la cadena de suministro de software de cualquier proyecto. No obstante, soy consciente de que su adopción no será un proceso fácil, ya que el mantenimiento de las dependencias no se percibe como una necesidad en muchos proyectos. Tal como se vio en la sección 5.2, es común encontrar proyectos con dependencias obsoletas o abandonadas hace años, que pueden dificultar la actualización del resto de dependencias debido a problemas de conflictos de versiones. Aunque esta herramienta puede ser una solución efectiva para evitar tales escenarios, su verdadero impacto dependerá en gran medida de un cambio de mentalidad en la comunidad de desarrolladores. Fomentar una cultura que valore el mantenimiento activo y la actualización continua de dependencias es un reto que debemos abordar si queremos mejorar la seguridad y estabilidad de nuestros proyectos.

Glosario

Cadena de suministro de software Conjunto de componentes, herramientas, y procesos necesarios para desarrollar, mantener y distribuir un producto de software. Incluye tanto el código propietario como las dependencias directas e indirectas (transitivas) y cualquier otra herramienta utilizada en el ciclo de vida del software.

Commit Registro de cambios realizado en un repositorio de control de versiones como Git, que captura el estado del proyecto en un momento específico junto con un mensaje que describe dichos cambios.

Conventional Commits Convención para escribir mensajes de commit que sigan un formato estructurado y predefinido, facilitando la comprensión del historial de cambios y la automatización de versiones.

CVE (Common Vulnerabilities and Exposures) Sistema de identificación de vulnerabilidades de seguridad conocidas en software, que permite a los desarrolladores y usuarios estar informados sobre problemas que pueden comprometer la seguridad de un sistema.

Dependencia directa Librería de terceros que se incluye explícitamente en un proyecto de software para aprovechar sus funcionalidades.

Dependencia transitiva Librería que es indirectamente incluida en un proyecto de software, como resultado de incluir otra librería que a su vez depende de esta.

ESLint Herramienta de análisis estático de código que se utiliza para identificar y corregir problemas en el código fuente de JavaScript y TypeScript, ayudando a mantener un código limpio y libre de errores comunes.

Fork Copia de un repositorio que se crea en la cuenta de un usuario, permitiéndole experimentar, modificar o ampliar el proyecto, sin afectar el repositorio fuente.

Issue Reporte que se deja en un proyecto para discutir un problema, una idea o una tarea pendiente.

Jest Marco de pruebas unitarias para JavaScript y TypeScript, utilizado para asegurar que las funciones individuales del código se comporten de la manera esperada.

Librería En el contexto de este proyecto, la palabra librería se usa para referirse a cualquier artefacto de código que pueda ser publicado, reutilizado e intercambiado. Es decir, este término abarca paquetes, módulos, librerías, middlewares y frameworks.

Node.js Entorno de ejecución de JavaScript ampliamente utilizado, tanto para el desarrollo como para la ejecución de proyectos en servidores.

Pruebas unitarias Tipo de prueba de software que verifica el funcionamiento de componentes individuales (o unidades) del código, asegurando que cada parte funcione correctamente de manera aislada.

Pull Request Solicitud de incorporación de cambios. Permite a los desarrolladores proponer cambios en el código u otros documentos del proyecto. Esta puede ser revisada y aprobada por los mantenedores de un proyecto para incluir nuevas características, solucionar errores o incorporar mejoras de otro tipo.

Registro de software Es la plataforma donde se publican los paquetes de código listos para ser descargados y usados dentro de otros proyectos.

repositorio Aalmacenamiento donde se guarda y gestiona el código fuente de un proyecto de software junto con su historial de cambios.

Referencias

[1] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid y E. Shihab, «Why do developers use trivial packages? an empirical case study on npm,» en *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ép. ESEC/FSE 2017, Paderborn, Germany: Association for Computing Machinery, 2017, págs. 385-395, ISBN: 9781450351058. DOI: 10.1145/3106237.3106267. dirección: https://doi.org/10.1145/3106237.3106267.

[2] O. Nierstrasz y T. Dirk Meijler, «Research directions in software composition,» *ACM Computing Surveys*, vol. 27, n.º 2, págs. 262-264, 2022. DOI: 10.1145/210376.210389.

[3] Github Blog, *Keep your dependencies secure and up-to-date with GitHub and Dependabot*, 2019. dirección: https://github.blog/2019-01-31-keep-your-dependencies-secure-and-up-to-date-with-github-and-dependabot/ (visitado 30-06-2024).

[4] Github, *How developer-first supply chain security helps you ship secure software fast*, 2022. dirección: https://resources.github.com/security/supply-chain-security/ (visitado 15-06-2024).

[5] G. A. A. Prana et al., «Out of sight, out of mind? How vulnerable dependencies affect open-source projects,» *Empirical Softw. Engg.*, vol. 26, n.º 4, jul. de 2021, ISSN: 1382-3256. DOI: 10.1007/s10664-021-09959-3. dirección: https://ink.library.smu.edu.sg/sis_research/6048/.

[6] Google Cloud, *What is a diamond dependency conflict?* 2019. dirección: https://jlbp.dev/what-is-a-diamond-dependency-conflict (visitado 05-06-2024).

[7] Instituto Nacional de Ciberseguridad, *Log4Shell: análisis de vulnerabilidades en Log4j*, 2022. dirección: https://www.incibe.es/incibe-cert/blog/log4shell-analisis-vulnerabilidades-log4j (visitado 12-06-2024).

[8] Wikipedia, *npm left-pad incident*. dirección: https://en.wikipedia.org/wiki/Npm_left-pad_incident (visitado 15-06-2024).

[9] F. Massacci et al., «"Free" as in Freedom to Protest?» *IEEE Security & Privacy*, vol. 20, n.º 5, págs. 16-21, 2022. DOI: 10.1109/MSEC.2022.3185845.

[10] Sonatype, *npm libraries 'colors' and 'faker' sabotaged in protest by their maintainer—What to do now?* 2022. dirección: https://www.sonatype.com/blog/npm-libraries-colors-and-faker-sabotaged-in-protest-by-their-maintainer-what-to-do-now (visitado 15-06-2024).

[11] M. Ohm, H. Plate, A. Sykosch y M. Meier, «Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks,» en *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini y N. Neves, eds., Cham: Springer International Publishing, 2020, págs. 23-43, ISBN: 978-3-030-52683-2.

[12] Arstechnica, *Sabotage: Code added to popular NPM package wiped files in Russia and Belarus*, 2022. dirección: https://arstechnica.com/information-technology/2022/03/sabotage-code-added-to-popular-npm-package-wiped-files-in-russia-and-belarus (visitado 15-06-2024).

[13] R. G. Kula, D. M. German, A. Ouni, T. Ishio y K. Inoue, «Do developers update their library dependencies?» *Empir Software Eng*, vol. 23, págs. 384-417, 2018. dirección: https://doi.org/10.1007/s10664-017-9521-5.

[14] Opentext, *What is Dynamic Application Security Testing (DAST)?* Dirección: https://www.opentext.com/what-is/dast#:~:text=Dynamic%20Application%20Security%20Testing%20(DAST)%20is%20the%20process%20of%20analyzing,like%20a%20malicious%20user%20would. (visitado 26-07-2024).

[15] OWASP, *DevSecOps Guideline - v-0.2, Interactive Application Security Testing*. dirección: https://owasp.org/www-project-devsecops-guideline/latest/02c-Interactive-Application-Security-Testing (visitado 26-07-2024).

[16] Y. Pan, «Interactive Application Security Testing,» en *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, 2019, págs. 558-561. DOI: 10.1109/ICSGEA.2019.00131.

[17] S. Journey, *SAST vs DAST vs IAST*. dirección: https://www.securityjourney.com/post/sast-vs-dast-vs-iast#:~:text=IAST%20Disadvantages,the%20assistance%20of%20a%20specialist. (visitado 26-07-2024).

[18] Synopsys, *What is software composition analysis (SCA)?* 2024. dirección: https://www.synopsys.com/glossary/what-is-software-composition-analysis.html (visitado 15-07-2024).

[19] OWASP, *Free for Open Source Application Security Tools*, 2024. dirección: https://owasp.org/www-community/Free_for_Open_Source_Application_Security_Tools (visitado 27-06-2024).

[20] J. F. Barcelona Auría, «DEAN: Analizador de dependencias estático para Evaluación de Riesgos,» thesis, Universidad de Zaragoza, Zaragoza, España, 2022.

[21] V. R. Basil y A. J. Turner, «Iterative enhancement: A practical technique for software development,» *IEEE Transactions on Software Engineering*, vol. SE-1, n.º 4, págs. 390-396, 1975. DOI: 10.1109/TSE.1975.6312870.

[22] C. Larman y V. R. Basili, «Iterative and incremental developments: a brief history,» *Computer*, vol. 36, n.º 6, págs. 47-56, 2003. DOI: 10.1109/MC.2003.1204375. dirección: https://www.craiglarman.com/wiki/downloads/misc/history-of-iterative-larman-and-basili-ieee-computer.pdf.

[23] N. M. Goldman y K. Narayanaswamy, «Software evolution through iterative prototyping,» *International Conference on Software Engineering*, págs. 158-172, 1992. dirección: https://dl.acm.org/doi/pdf/10.1145/143062.143109.

[24] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama y M. Prabaker, «Field studies of computer system administrators: analysis of system management tools and practices,» en *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, ép. CSCW '04, Chicago, Illinois, USA: Association for Computing Machinery, 2004, págs. 388-395, ISBN: 1581138105. DOI: 10.1145/1031607.1031672. dirección: https://www.researchgate.net/publication/220879050_Field_studies_of_computer_system_administrators_Analysis_of_system_management_tools_and_practices.

[25] S. R. Murillo y J. A. Sánchez, «Empowering Interfaces for System Administrators: Keeping the Command Line in Mind when Designing GUIs,» en *Proceedings of the XV International Conference on Human Computer Interaction*, ép. Interacción '14, Puerto de la Cruz, Tenerife, Spain: Association for Computing Machinery, 2014, ISBN: 9781450328807. DOI: 10.1145/2662253.2662300. dirección: https://doi.org/10.1145/2662253.2662300.

[26] H. Sampath, A. Merrick y A. Macvean, «Accessibility of Command Line Interfaces,» en *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ép. CHI '21, Yokohama, Japan: Association for Computing Machinery, 2021, ISBN: 9781450380966. DOI:

10.1145/3411764.3445544. dirección: https://dl.acm.org/doi/pdf/10.1145/3411764.3445544.

[27] AWS Cloud Comparisons, *What's the Difference Between YAML and JSON?* Dirección: https://aws.amazon.com/compare/the-difference-between-yaml-and-json/ (visitado 08-08-2024).

[28] R. Nunkesser, «Using Hexagonal Architecture for Mobile Applications,» en *International Conference on Software Technologies*, vol. 17, 2022, págs. 113-120, ISBN: 978-989-758-588-3. DOI: 10.5220/0011075100003266. dirección: https://www.scitepress.org/Papers/2022/110751/110751.pdf.

[29] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st. USA: Prentice Hall Press, 2017, cap. The Clean Architecture, ISBN: 0134494164.

[30] M. Richards, *Software Architecture Patterns*. O'Reilly Media, Inc, 2022, cap. Microkernel Architecture, ISBN: 9781098134273.

Anexos

A. Lista de métricas detallada

A continuación se expone la lista de métricas candidatas, explicando su importancia y señalando la fuente dónde se esperaba obtener este valor.

- **Métricas relacionadas con el mantenimiento de la librería:**
  - **Número de versiones:** la cantidad de versiones que han sido publicadas a lo largo de la vida del proyecto. Se puede pensar que un proyecto con un elevado número de versiones puede ser un proyecto activo. Se debe poder obtener del registro público donde se expone la librería.
  - **Tiempo de vida del proyecto:** el tiempo transcurrido entre la primera publicación de la librería y la fecha actual. Se puede esperar que un proyecto con tiempo de vida largo tenga menos probabilidades a ser abandonado. Se debe obtener del registro.
  - **Frecuencia media de publicación de nuevas versiones:** corresponde al tiempo promedio que transcurre entre una publicación y otra. Una frecuencia de publicación alta puede indicar un nivel de actividad alto. Se obtiene de dividir el número de versiones entre el tiempo de vida del proyecto.
  - **Tiempo transcurrido desde la última publicación:** si ha transcurrido mucho tiempo desde la última publicación, puede ser un indicio de que el proyecto ha sido abandonado. Este parámetro se obtiene de la diferencia entre la fecha actual y la fecha de la última publicación.
  - **Número de mantenedores:** son las personas que aportan de forma activa al mantenimiento de la librería y la construcción de nuevas características. Si el número de mantenedores es elevado, se puede considerar que hay menos riesgo de que se incluya código malicioso, se elimine o se abandone el proyecto. Se debe obtener del repositorio.
  - **Tipo de propietario del proyecto:** el propietario de un repositorio puede ser una empresa o persona. En términos generales, se puede preferir que el propietario sea una empresa, por el respaldo que esto puede suponer para el proyecto. Este dato se debe obtener del repositorio donde se aloja el código fuente.
  - **Número de Issues abiertos:** corresponde a la cantidad de reportes que se crean para discutir bugs, ideas de mejora o tareas pendientes. Una cantidad alta de issues abiertos en un periodo corto de tiempo puede ser síntoma de una comunidad e interesada del proyecto, aunque también puede indicar que algo está yendo mal en el mismo. Este dato se debe obtener del repositorio.
  - **Número de Issues cerrados:** es la cantidad de reportes que se cierran. Se espera que todos los issues que se abren se cierren una vez la discusión lleva a una solución del bug o a la implementación de la idea que se estaba abordando. Idealmente, se espera que el número de issues cerrados sea cercano al de issues abiertos, si es así, es señal de buena

capacidad de reacción por parte del equipo que mantiene el proyecto. Este dato se debe obtener del repositorio.

- **Tiempo de vida de los Issues:** además de cerrar los issues, es importante el tiempo que tardan en ser resueltos. Un tiempo de vida bajo indica que los mantenedores responden rápidamente a los desafíos que se puedan encontrar los usuarios de una librería. Este dato se debe obtener calculando el tiempo transcurrido entre la apertura y el cierre de cada issue que ha sido cerrado.
- **Número de Pull Requests activas:** es la cantidad de solicitudes de incorporación de cambios abiertas por la comunidad. En un proyecto de código abierto, cualquier usuario puede contribuir abriendo una Pull Request (PR), ya sea para corregir un defecto o para incorporar nueva funcionalidad. El equipo de mantenedores debe revisar estas solicitudes y aprobarla, rechazarla o solicitar modificaciones según consideren necesario. Un número alto de PR's abiertas en un periodo de tiempo puede significar un alto interés de la comunidad en el proyecto, aunque también puede indicar una baja capacidad de respuesta por parte de los mantenedores. Este parámetro se debe obtener del repositorio.
- **Número de Pull Requests cerradas:** al igual que con los issues, es importante que las PR's se cierren, ya sea porque los cambios han sido aceptados e integrados, o porque han sido rechazados. Lo ideal es que el total de las solicitudes de cambio se resuelvan, si es así, es señal de un nivel saludable de actividad por parte de la comunidad y los mantenedores. Este dato se debe extraer del repositorio.
- **Tiempo de vida de las Pull Requests:** al igual que con los issues, es interesante saber cual es el tiempo medio que tarda una PR en ser resuelta. Un tiempo bajo es una buena señal de la capacidad de reacción por parte de los mantenedores. Este dato se obtiene de la diferencia entre la fecha de creación y cierre de cada Pull Request.
- **Porcentaje de salud del repositorio:** es un parámetro que se puede obtener del perfil de la comunidad de GitHub. Está relacionado con la presencia de documentación básica en el repositorio.

- **Métricas relacionadas con la comunidad:**
  - **Número de descargas semanales desde el registro:** el número de descargas de una librería se puede extrapolar al número de usuarios y de proyectos que dependen de esta. Un número elevado de descargas puede significar un mayor soporte para la librería, al haber un número elevado de usuarios interesados en mantener el proyecto vivo y saludable. Este valor se deber recuperar del registro público.
  - **Número de estrellas en el repositorio:** las estrellas son una forma de mostrar "aprecio" por un proyecto, a la vez que permite guardar proyectos de interés para hacer seguimiento sin recibir notificaciones de la actividad del repositorio. Un número alto de estrellas en un repositorio refleja un interés elevado por parte de otros usuarios. Este parámetro se debe extraer del repositorio.
  - **Número de Forks del repositorio:** número de veces que el proyecto ha sido copiado para trabajar de forma paralela en otras características. Si el proyecto tiene un alto número de "forks", puede indicar una relevancia alta para la comunidad. Este dato se debe extraer del repositorio público.
  - **Número de proyectos dependientes:** en algunos registros, como npm, es posible encontrar la lista de proyectos que dependen de una librería en particular. Un elevado

número de proyectos dependientes implica un grupo proporcional de usuarios interesados en mantener el proyecto saludable. Este dato se debe poder extraer del registro.

- **Número de observadores:** cantidad de usuarios que reciben notificaciones de todas la actividad en el repositorio, como issues, Pull Requests y cambios en el código fuente. Un número alto de observadores puede servir como garantía de que el proyecto tiene un buen respaldo y una alta capacidad de respuesta. Este parámetro se debe buscar en el repositorio.

B. ¿Cómo dar soporte a otros lenguajes?

Para dar soporte a otros lenguajes, se necesita primero identificar las fuentes de datos y, posteriormente, construir tres elementos en la herramienta: los **Builders** para obtener información de las fuentes identificadas; el **BuilderDirector** con el cual trabaja el sistema central; y por último, la función encargada de detectar las dependencias del proyecto.

La complejidad de un Builder depende de la cantidad de datos que obtiene de su fuente y la complejidad del cálculo de las métricas para construir la librería. Aunque se puede decir que, en general, son clases muy sencillas que implementan un único método público definido por la interfaz. A continuación se muestra la interfaz y un ejemplo de implementación usado en la herramienta.

```
export interface LibraryBuilder {
  addLibraryParams(lib: Library): Promise<void>;
}

async function getNpmDownloads(libName: string) {
  try {
    const downloads = await httpGet(npmDownloadsUrl(libName));
    return JSON.parse(downloads) as NpmDownloads;
  } catch (e) {
    throw new Error(`Error getting downloads for ${libName}`);
  }
}

export class NpmDownloadsBuilder implements LibraryBuilder {
  async addLibraryParams(library: Library) {
    const npmDownloads = await getNpmDownloads(library.name);
    if (npmDownloads) library.weeklyDownloads = npmDownloads.downloads || 1;
  }
}
```

En cuanto a los Directores, como la única diferencia entre instancias para diferentes lenguajes son sus atributos (la implementación es idéntica, solo cambian los vectores que asocian fuentes y Builders), se optó por crear una única clase e inicializar estos atributos a través del constructor. En el bloque de código a continuación como se muestra cómo se instancia el BuilderDirector para JavaScript.

```
function createJavascriptBuilder() {
  const jsParams = new Map<keyof Library, Source>();
  jsParams.set("repoName", NPM);
```

```
  jsParams.set("repoOwner", NPM);
  jsParams.set("numberOfVersions", NPM);
  jsParams.set("weeklyDownloads", NPM_DOWNLOADS);
  jsParams.set("lastVersion", NPM);
  jsParams.set("lastVersionDate", NPM);
  jsParams.set("lifeSpan", NPM);
  jsParams.set("releaseFrequency", NPM);
  jsParams.set("repoOpenIssues", GITHUB);
  jsParams.set("repoStars", GITHUB);
  jsParams.set("repoForks", GITHUB);
  jsParams.set("repoObservers", GITHUB);
  jsParams.set("repoOwnerType", GITHUB);
  jsParams.set("repoHealth", GITHUB_COMUNITY);

  const jsBuilders = new Map<Source, LibraryBuilder>();
  jsBuilders.set(NPM, new NpmBuilder());
  jsBuilders.set(NPM_DOWNLOADS, new NpmDownloadsBuilder());
  jsBuilders.set(GITHUB, new GithubBuilder());
  jsBuilders.set(GITHUB_COMUNITY, new GithubCommunityBuilder());

  return new BuilderDirector(jsParams, jsBuilders);
}
```

Para extender a Java, por ejemplo, habría que crear tantos Builders como fuentes hagan falta para completar los atributos de la librería. Después, habría que añadir estos Builders al director en la función que se muestra a continuación. De este modo, la función `createBuilderDirector` estaría preparada para crear directores de Java y de JavaScript.

```
function createJavaBuilder() {
  const javaParams = new Map<keyof Library, Source>();
  /* @TODO: Map labrary attributes to data sources */
  const javaBuilders = new Map<Source, LibraryBuilder>();
  /* @TODO: Implement Java builders and add to map */
  return new BuilderDirector(javaParams, javaBuilders);
}

export function createBuilderDirector(language: string) {
  if (language === "javascript") {
    return createJavascriptBuilder();
  }
  if (language === "java") {
    return createJavaBuilder();
  }
  throw new Error(`Language ${language} not supported`);
}
```

Por último, para poder analizar todas las dependencias de un proyecto, hace falta una función que permita leer el archivo de configuración según el lenguaje al que se esté dando soporte. Por ejemplo, actualmente la herramienta cuenta con una función que lee el archivo `package.json` y extrae las dependencias de un proyecto de JavaScript, como se muestra en el bloque de código a continuación. Para dar soporte a Java, siguiendo el ejemplo de antes, haría falta construir la función encargada de leer el archivo `pom.xml` del proyecto para extraer las dependencias.

```
async function getJavascritDeps() {
  const packageJson = await readFileAsync("./package.json", "utf8");
  const parsedPackageJson = JSON.parse(packageJson);
  const allDeps = {
    ...parsedPackageJson.dependencies,
    ...parsedPackageJson.devDependencies,
  } as Record<string, string>;
  for (const dep in allDeps) {
    allDeps[dep] = allDeps[dep].replace("^", "").replace("~", "");
  }
  return allDeps;
}

export async function getProjectDeps(language: string) {
  if (language === "javascript") {
    return await getJavascritDeps();
  }
  if (language === "java") {
    /** @TODO: add function to read Java deps */
  }
  throw new Error(`Language ${language} not supported`);
}
```

C. ¿Cómo crear otros reportes?

Para crear otros formatos e interfaces, basta con crear una nueva implementación de la interfaz Contexto explicada en la sección 3.4. El contexto debe tener un método para mostrar errores y otro para mostrar los resultados, tal como se muestra en el bloque de código a continuación.

```
1 interface ExecutionContext {
2   showResults: (
3     results: Map<string, ResultsStore>,
4     indicators?: string[]
5   ) => void | Promise<void>;
6   showError: (error: unknown) => void;
7 }
```

El método para mostrar los resultados recibe dos argumentos: el vector asociativo (`Map`) que contiene los resultados para cada librería analizada (línea 3) y la lista de indicadores que fueron evaluados (línea 4). La lista de indicadores se envía aparte porque puede que algunos resultados no contengan todos lo indicadores, esto debido a las condiciones de parada o a posibles errores en la ejecución. Esto se observa en a figura 5.7, donde algunas librerías tienen espacios en gris señalando que esos indicadores no fueron evaluados.

Para crear un nuevo reporte, se puede recorrer el vector asociativo extrayendo los resultados de cada librería y, luego, recorrer la lista de indicadores analizados para cada librería, extrayendo el resultado de cada uno de ellos del `ResultsStore`, si es que existe.

En el reporte en formato HTML, por ejemplo, se utilizan estos dos bucles para crear la tabla, el primero permite crear una fila por cada librería, mientras que el segundo crea la celda con el resultado para cada indicador. En el repositorio del proyecto se pueden encontrar las implementaciones tanto del reporte en HTML como del reporte en consola, que pueden servir de guía a la hora de implementar una clase para un nuevo formato.

D. ¿Cómo añadir nuevos indicadores?

Para añadir un nuevo indicador, se deben crear una clase que implemente la interfaz Indicador explicada en la sección . Esta interfaz se muestra a continuación.

```
1 interface Indicator {
2   name: string;
3   evaluate: (lib: Library) => IndicatorResult;
4   message: string | ((...data: unknown[]) => string);
5   parameters: Array<keyof Library>;
6   preconditions?: Array<IndicatorPrecondition>;
7   setThresholds?: (thresholds: IndicatorThresholds) => void;
8 }
```

Las clases que implementan esta interfaz necesitan: un nombre (línea 2) para ser identificadas en el registro, una lista de parámetros (línea 5) que permite a la herramienta saber si dispone de la información necesaria para la evaluación o si debe consultarla, una función de evaluación (línea 3) que recibe la instancia de la librería y retorna el resultado y, por último, el mensaje (línea 4) que permitirá mostrar al usuario el resultado de la evaluación en un formato legible. Las pre-condiciones (línea 6) son opcionales, este atributo permiten señalar los indicadores que deben ser evaluados antes que el indicador en cuestión; el método `setThresholds` también es opcional, este permite establecer los umbrales que selecciona el usuario mediante el archivo de configuración, es especialmente útil para aquellos indicadores que utilizan métricas cuantificables.

Si el nuevo indicador usa métricas que no se están incluyendo actualmente como propiedades del objeto librería, se debe primero añadir el atributo al modelo de librería (bloque de código 1) y luego modificar el sistema de extracción de datos para obtener esas nuevas métricas. Si se trata de extraer esos datos de nuevas fuentes podría hacer falta añadir nuevos Builders, tal como se muestra en el anexo B.

En el repositorio del proyecto se encuentran los indicadores desarrollados hasta ahora, estos pueden servir de ejemplo para las nuevas implementaciones. En el fichero de indicadores de actividad del repositorio, por ejemplo, se encuentran varias implementaciones que usan características cuantitativas como cualitativas.

E. Cronograma del proyecto

A continuación se exponen las tareas principales que se llevaron a cabo en este proyecto. Asimismo, en la figura E.1 se presenta (a groso modo) la distribución temporal de estas tareas.

- Definición inicial del proyecto: identificación del problema y definición de objetivos.
- Análisis de antecedentes y revisión bibliográfica: se llevó a cabo una búsqueda y revisión inicial de documentación relacionada con el problema. Se incluyeron incidentes, herramientas y estudios relacionados con el manejo de dependencias de terceros.
- Identificación de métricas y fuentes de datos: consistió en el análisis presentado en el capítulo 2.
- Diseño y desarrollo de la herramienta: como se expuso en los capítulos 3 y 4, el sistema cuenta con 3 módulos principales, los cuales se diseñaron, desarrollaron y probaron de forma iterativa y, hasta cierto punto, paralela. La distribución de tiempo se muestra de forma aproximada para cada uno de ellos en la figura E.1.
- Validación y ajuste de la herramienta: consistió en el proceso de pruebas con otros proyectos, usuarios y herramientas, que se exponen en el capítulo 5.
- Documentación del trabajo: consiste principalmente en la elaboración de esta memoria.

Figura E.1: Cronograma del trabajo realizado.