UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: CodeCaption – Una herramienta para realizar Code Review distribuido
Autores: Nahuel Alejandro Aparicio
Director: Dr. Federico Balaguer
Codirector: –
Asesor profesional: –
Carrera: Licenciatura en Sistemas

Resumen

La tesina detalla el desarrollo de una herramienta para realizar Code Review distribuido llamada CodeCaption. El desarrollo fue realizado en Pharo Smalltalk. La herramienta permite al desarrollador agregar comentarios al código fuente de un proyecto dentro del IDE, y que estos puedan ser vistos, revisados y/o modificados por todos los desarrolladores del mismo. El objetivo de este trabajo es mejorar la comunicación de un equipo de desarrollo de software con revisiones de código asíncronas y remotas. Está especialmente orientado a equipos de desarrollo remotos donde sus miembros se encuentran en diferentes ubicaciones y husos horarios.

Palabras Claves

Code Review, CodeCaption, Equipo Remoto, Pharo, Smalltalk, Abstract Syntax Tree, Git, STON, Prueba de Usuario

Conclusiones

Se logró un desarrollo que representa un primer prototipo funcional de una herramienta de revisión de código dentro de un IDE. Cumple los objetivos planteados en el trabajo, logrando permitir a los desarrolladores realizar revisiones de código de forma remota y asíncrona sin necesidad de uso de software externo. La herramienta pasó por una prueba de usuario donde logró facilitar y acortar los tiempos en la tarea de revisión de código entre desarrolladores. El proyecto sirve como base para una herramienta robusta de revisión de código entre pares dentro del IDE Pharo.

Trabajos Realizados

Al principio se analizaron las herramientas disponibles dentro de Pharo y las tecnologías posibles para lograr el guardado y sincronización de revisiones de código. Luego, se planteó un diseño tanto para la interfaz de la herramienta como para sus distintos componentes y la interacción entre ellos. A partir de este momento se comenzó la implementación de la herramienta desde Pharo Smalltalk. Se extendió la funcionalidad del IDE y se desarrolló la creación, guardado, resolución y sincronización de las revisiones. Se hizo uso de la implementación de Abstract Syntax Trees de Pharo para crear las revisiones, STON para el guardado y Git para la sincronización. Por último se realizaron Pruebas de Usuario para medir usabilidad, satisfacción y mejoras posibles a la herramienta.

Trabajos Futuros

Como posibles trabajos futuros se puede considerar mejorar la experiencia del usuario. Al no ser el enfoque principal, y teniendo en cuenta las limitadas herramientas provistas por el IDE, en las pruebas se notó que la experiencia de usuario se puede mejorar notoriamente. También se pueden extender las funcionalidades para agregar más interacciones a las revisiones como comentarios, reacciones, etc. Otro posible trabajo futuro es el de implementar un análisis de comentarios mediante el uso de Inteligencia Artificial. Esto podría identificar estados de ánimo en las revisiones, errores comunes, palabras claves y demás estadísticas de interés.

Fecha de la presentación: DICIEMBRE 2022

CodeCaption – Una herramienta para realizar Code Review distribuido

Tesina de Grado para el título de Licenciado en Sistemas de la Universidad Nacional de La Plata, Facultad de Informática.

Autor: Nahuel Alejandro Aparicio

Director: Dr. Federico Balaguer

Esta tesina está dedicada a mi novia, familia y amigos. Cada uno brindó el apoyo necesario para seguir adelante y nunca bajar los brazos.

También quiero dar un agradecimiento especial al director Dr. Federico Balaguer quien se ofreció a dirigir la tesina. Sin su apoyo, consejo, y guía no hubiera sido posible el desarrollo de la misma.

Índices

Índice General

Lista de Figuras

Capítulo 1. Introducción

1.1 Objetivos del proyecto

El objetivo principal de este trabajo consiste en mejorar la comunicación de un equipo de desarrollo de software con revisiones de código asíncronas y remotas. En especial para el caso de equipos de desarrollo remotos donde sus miembros se encuentran en diferentes ubicaciones y husos horarios. Se busca facilitar la tarea de los programadores para realizar comentarios y revisiones de código sin la necesidad de utilizar otro tipo de software complementario. Para lograr esto, se desarrolló una herramienta integrada en el entorno de desarrollo Pharo Smalltalk que permite realizar Code Reviews y que estas sean visibles e interactuables por todo el equipo de desarrollo involucrado. La herramienta permite al desarrollador:

- Crear una revisión de código
  - Añadir una revisión a una porción del código en el entorno de desarrollo.
  - La revisión creada hace referencia a la estructura sintáctica del programa seleccionada para Code Review.
- Visualizar una revisión de código
  - Visualizar las revisiones del proyecto creadas dentro del editor de código del entorno.
  - Las revisiones estarán visibles junto al código que hacen referencia.
- Modificar una revisión de código
  - Modificar una revisión de código creada en el entorno de desarrollo.
  - Cambiar contenido, añadir información.
- Resolver una revisión de código
  - Marcar una revisión de código como resuelta.
  - Un desarrollador podrá marcarla como resuelta cuando se considere que lo indicado ya fue solucionado en el código.
- Integrar las revisiones al repositorio Git[1] del proyecto
  - Integrar todas las revisiones al repositorio del proyecto para garantizar su disponibilidad remota.

1.2 Motivación

Actualmente en la industria del Desarrollo de Software es de suma importancia el trabajo remoto en equipo. Un gran porcentaje de los proyectos de programación se basan en desarrollos realizados por equipos distribuidos en distintas partes físicas del mundo [1], esta tendencia se va acrecentando cada vez más, y se ha visto potenciada a partir del año 2020 por

[1] Software de control de versiones cuyo propósito es llevar registro de los cambios en archivos de computadora incluyendo coordinar el trabajo que varias personas realizan sobre archivos compartidos en un repositorio de código.

la pandemia global provocada por el virus COVID-19 [2][3]. Sin embargo con el trabajo distribuido y remoto suelen aparecer nuevas problemáticas que no estaban presentes en el trabajo “on site”.
Nos encontramos entonces, en una situación donde distintos programadores desarrollan al mismo tiempo sobre un mismo código, por lo que la coordinación y comunicación es clave para lograr un desarrollo correcto sin complicaciones. Evitar interferencias entre los desarrolladores, retrasos y errores puede ahorrar costos considerables. Es así que vemos la necesidad de sincronizar los equipos de desarrollo para que cada uno realice su parte sin problemas y sobre todo, puedan revisar mutuamente el trabajo realizado para encontrar problemas o mejoras en el desarrollo [4].

Este trabajo se enfoca en la revisión mutua de código asíncrona y remota entre desarrolladores. Hoy en día vemos que el desarrollo en un equipo remoto se suele sincronizar mediante: el uso de herramientas de versionado de código y herramientas de comunicación grupales, (con propósito más general). El problema ocurre cuando un desarrollador realiza una revisión de código a una sección hecha por otro miembro del equipo. Para lograr esto, debería buscar el código a revisar en el repositorio y ver en la herramienta de versionado quién realizó el cambio, para entonces comentarle en el canal de comunicación la revisión concreta de su código. Incluso esto se haría más complejo si ni siquiera se comparte huso horario entre los mismos, provocando retrasos en la comunicación. La alternativa de usar los comentarios propios del código, (del tipo “*// This is a comment*”) no es suficiente para poder realizar este tipo de tareas, ya que forman parte estructural del código y su utilidad está enfocada en la comprensión y entendimiento del mismo.
Este proceso de revisión puede tornarse muy lento y tedioso rápidamente en proyectos muy grandes con muchos desarrolladores trabajando en conjunto por lo que surge la necesidad de unir la comunicación con el desarrollo en un mismo contexto y entorno [5].

Es necesaria entonces una herramienta que permita realizar estas revisiones de código colaborativas en el mismo entorno de desarrollo, sincronizando y distribuyendo las mismas, dando visibilidad y disponibilidad a interactuar con ellas en todos los entornos distribuidos del equipo. Esto nos permite poder integrar la comunicación (revisiones sobre el código) con el contexto de desarrollo (edición de código) de un proyecto sin necesidad de canales separados. También encontramos de utilidad una herramienta como ésta en otros ámbitos como el educativo, utilizando las revisiones de código como correcciones y recomendaciones al desarrollo de un estudiante por parte del docente dentro del mismo entorno, limitando los canales de comunicación al uso puramente educativo.

1.3 Organización de la Tesis

El trabajo está organizado en 5 capítulos principales.
El primer capítulo es introductorio al trabajo de la tesina. El capítulo expone los objetivos, la motivación detrás del desarrollo y explica la organización del documento.

El segundo capítulo proporciona el marco teórico detallando definiciones referentes a los temas abordados durante el desarrollo del trabajo.

El tercer capítulo detalla todo lo referente al diseño de la herramienta desarrollada en el trabajo. Explica en detalle cómo se diseñó, y también explica las particularidades de cada decisión tomada durante el proceso de diseño. El objetivo de este capítulo es explicar conceptual y estructuralmente el desarrollo del trabajo con el uso de gráficos acordes, sin llegar a adentrarse en tecnologías específicas ni en la implementación.

El cuarto capítulo relata todos los aspectos de la implementación de la herramienta. Luego de explicar el diseño, en este capítulo se cuenta en detalle el proceso de implementación de la herramienta, desde el comienzo del desarrollo, pasando por todas las particularidades sobre las tecnologías utilizadas hasta tener un desarrollo completo. Aquí se ve detallado el entorno de desarrollo sobre el cual se implementa la herramienta y las tecnologías usadas para lograr el cometido.

El quinto capítulo es sobre las pruebas de usuario subsiguientes a la implementación de la herramienta. Se detallan las distintas pruebas realizadas, en qué consisten, cómo se realizaron, los involucrados, y qué resultado se obtienen.

El último y sexto capítulo es un capítulo final donde se plasman las conclusiones a las que se llegó durante el desarrollo del trabajo. Luego del análisis, diseño, implementación y pruebas se arman las conclusiones en base a los resultados obtenidos y las experiencias ocurridas, también dejando sentados los posibles trabajos futuros para realizar con la herramienta.

Capítulo 2. Antecedentes y Conceptos

El foco del trabajo es proveer las herramientas que permitan agilizar los procesos para las revisiones de código remotas y asíncronas en un equipo de desarrollo de software.
Las revisiones de código son un instrumento muy importante para ayudar a mejorar la calidad del código en una organización. Ayudan a detectar errores y/o mejoras en una etapa temprana del desarrollo para evitar sorpresas durante la etapa productiva del software.

En este capítulo se detallan conceptos teóricos relacionados con la calidad de software, la calidad del código fuente y cómo garantizar la misma, y revisiones de código. También se explican otros conceptos técnicos relacionados al análisis estático del código y su importancia en el diseño de la herramienta CodeCaption.

2.1 Calidad de Software y Métricas de Calidad de Código Fuente

Al consumir un bien o servicio, casi siempre buscamos que se adecúe a nuestras exigencias, es decir estamos en busca de un producto de *calidad*. La calidad de software puede definirse como el grado en el que un sistema, componente, o proceso satisface las necesidades de un requerimiento, es decir que cumple con las necesidades o expectativas del usuario [6]. Además, de acuerdo al estándar de la IEEE sobre Ingeniería de Software [7], la calidad de software se puede resumir en una serie de atributos de un sistema de software que se definen como:

- El grado en el que un software o proceso cumple con las expectativas o necesidades del usuario o cliente.
- El grado en el que un software o proceso cumple con los requerimientos especificados.
- La calidad contempla todas las características y funcionalidades de un producto o actividad en relación a la satisfacción de los requerimientos dados.

Para garantizar la ya definida calidad del software, recurrimos al concepto de Aseguramiento de la Calidad o Quality Assurance. El Aseguramiento de la Calidad es la parte del manejo de la calidad enfocado en proveer la seguridad suficiente que los requerimientos de calidad serán cumplidos. Es decir que consta de un seguimiento y monitoreo en los métodos, actividades y productos para cumplir con cierta normativa de calidad. [8].

Cada proyecto de software necesita tener un código eficiente, fácil de entender, de modificar y de expandir en un futuro. Una forma de implementar el aseguramiento de la calidad es garantizando que el código fuente del software en cuestión cumpla estas afirmaciones, y esto se puede lograr siguiendo ciertas métricas de calidad de código fuente.

Una forma de categorizar estas métricas de código es en: [9][10]

- Métricas de Calidad Código Cualitativas
- Métricas de Calidad Código Cuantitativas

2.1.1 Métricas de Calidad Código Cualitativas

Las métricas de calidad de código cualitativas se consideran como mediciones subjetivas de los encargados de un proyecto para definir qué significa que el código es de calidad.

Eficiencia

La eficiencia del código es medida por la cantidad de recursos consumidos en la creación y ejecución del mismo, así como en la velocidad en la que se ejecuta. Es primordial no usar más recursos de lo necesario y no generar problemas de performance al usuario.

Extensibilidad

Aquí se mide el grado en el que un software puede ser modificable o extendible en un futuro sin crear demasiadas complicaciones al código existente. El código debería contemplar ciertas posibles futuras funcionalidades o expansiones y no realizar sólo la tarea específica que se requiere en el momento. Esto puede ahorrar mucho tiempo de desarrollo y costos en el futuro.

Buena Documentación

Una buena documentación garantiza mayor legibilidad y facilidad al entendimiento del código, no sólo para otros desarrolladores, incluso para el autor. La corrección de errores o el agregado de nuevas funcionalidades se hace mucho más ágil al contar con una buena documentación del código, lo cual también ayuda a la mantenibilidad del software.

Mantenibilidad

Define qué tan fácil es realizar cambios en el código existente y los riesgos que conlleva realizar estos cambios. Cuanto menos mantenible sea el código, más tiempo llevará realizar cambios. Generalmente cuando un software tiene demasiadas líneas de código sin una buena organización la mantenibilidad es perjudicada.

Claridad

Un código se considera claro cuando al leerlo se entiende perfectamente qué hace y no es ambiguo. Esto es importante desarrollando en equipo, sobre todo, para que cualquiera pueda entender la función del código escrito con facilidad.

Legibilidad y Formateo de Código

Esta métrica apunta a respetar los estándares de código del lenguaje para mejorar su legibilidad, como puede ser usar una indentación correcta, dividir líneas largas en varias partes, evitar “if”s anidados, etc.

Comprobabilidad (Testability)

Software con tests bien definidos e implementados ayudan a asegurar que el código no contenga errores y sobre todo, agiliza esta verificación de errores con cada cambio que se agregue al mismo.

2.1.2 Métricas de Calidad Código Cuantitativas

Las métricas cuantitativas ayudan mediante fórmulas y algoritmos a identificar un código de calidad.

Puntos de Función Ponderados (Weighted Micro Function Points)

Esta métrica consta de un algoritmo que parsea y divide al código fuente en pequeñas funciones. Luego se aplican métricas de complejidad a estas funciones para al final generar un resultado general. Se analizan comentarios, estructura del código, cálculos aritméticos, y camino de flujo de control.

Métricas de Complejidad de Halstead

Estas son mediciones que analizan el vocabulario del código, la longitud, volumen, dificultad, esfuerzo, y tiempo de testeo. Estas mediciones proveen un estimativo de la complejidad del código analizado.

Complejidad Ciclomática

Es una métrica que define la complejidad estructural de un programa. Esto lo logra contando los diferentes caminos independientes que tiene cada porción del código fuente. Los métodos con una complejidad ciclomática muy alta (mayor a 10) son más difíciles de entender y generalmente contienen defectos que pueden ser mejorados.

2.2 Metodologías Ágiles y Extreme Programming

Para lograr un uso eficiente de la herramienta en un ambiente de trabajo remoto puede ayudar en gran medida el uso de metodologías ágiles como Extreme Programming en el equipo. Esto se haría buscando cierto orden y estructura sin perder agilidad en el trabajo en equipo, sobre todo priorizando la constante comunicación y sincronización entre los desarrolladores del proyecto.

En las primeras décadas de la programación había una visión muy difundida de que la forma más adecuada para lograr un mejor software era mediante una cuidadosa planeación del proyecto, aseguramiento de calidad formalizada y el uso de métodos de análisis, así como procesos de desarrollo de software rigurosos y controlados. Este tipo de planeación de proyectos de desarrollo estaba pensado desde un concepto más ligado a la ingeniería y para equipos grandes en grandes compañías. Cuando se intentaba aplicar a equipos y organizaciones pequeñas los costos eran enormes y se invertía más tiempo en diseñar el sistema, que en el desarrollo y la prueba del programa.
A partir de esto surgen las metodologías ágiles, que se apoyan universalmente en un enfoque incremental para la especificación, el desarrollo y la entrega del software. Son adecuados para el diseño de aplicaciones con requerimientos cambiantes. Tienen la intención de entregar con

lo más rápido posible el software operativo a los clientes, quienes entonces propondrán requerimientos nuevos para incluir en posteriores iteraciones del sistema. [11]

Una de estas metodologías ágiles es la Programación Extrema o Extreme Programming (XP), que fue formulada por Kent Beck, que como su nombre lo dice, se enfoca en llevar al extremo algunas de las prácticas ya existentes en la programación. Está diseñada para equipos pequeños de desarrollo donde los requerimientos no son muy claros y/o sufren cambios constantes. [12]
Los requerimientos son expresados junto con el cliente en forma de Historias de Usuario[2] y estas se dividen en una serie de tareas a completar. Por cada una de estas tareas los programadores desarrollan, escriben y realizan tests, reestructuran el código y discuten y evalúan las funcionalidades entre programadores y clientes constantemente. [11]

Una práctica principal de XP es la de Programación en Pares o Pair Programming. Los desarrolladores involucrados en implementar las tareas propuestas, trabajan en pares, y cada uno comprueba el trabajo del otro. Además, también ofrecen apoyo para que se realice siempre un buen trabajo. Incluso, otras prácticas de XP indican que los desarrolladores trabajan en todas las áreas del sistema, de manera que no se crean islas de experiencia, ya que todos los desarrolladores se responsabilizan por todo el código. Cualquiera puede cambiar cualquier función y se logra un sentido de pertenencia con el producto final.

Existe entonces una constante comunicación entre los desarrolladores, tanto para desarrollar, planificar, evaluar y testear los componentes que hacen a las tareas. Por lo que resulta vital que estas vías de comunicación sean lo más rápidas y fáciles de usar posible.

CodeCaption apunta a agilizar estas tareas entre los desarrolladores y mejorar la comunicación entre los mismos cuando no se encuentran dentro del mismo espacio físico ni temporal. Las revisiones de código en el contexto de XP son una herramienta importante al desarrollar en sincronía con otros programadores. La posibilidad de realizar revisiones de código directamente en el mismo contexto del desarrollo contribuye a hacer el Pair Programming más ágil y más cercano a lo que sería el mismo en un escenario no remoto.

2.3 Revisión de Código

Una de las actividades más utilizadas en XP que es de gran utilidad para asegurar la calidad del código fuente son las revisiones de código, que sirven de foco de atención en este trabajo.

Una revisión de código (Code Review o también Peer Review) es una actividad de Quality Assurance donde una o más personas revisan con atención un programa de computadora principalmente mediante la lectura del código fuente del mismo. La revisión puede ocurrir durante o después de la implementación del código, aunque es necesario que al menos una de

[2] Una Historia de Usuario representa una funcionalidad requerida por el cliente, expresada desde la perspectiva del usuario en el sistema.

las personas encargadas de la revisión (llamados revisores) no haya participado en el desarrollo del código a revisar. Esto es importante ya que un desarrollador que no conoce el código puede abstraerse y analizar sin ningún prejuicio, ni omisión por conocimiento previo. Es así que al desarrollador no involucrado en el desarrollo del código le será más fácil encontrar aspectos a corregir y/o mejorar.

La función principal de la revisión de código es contribuir a garantizar la calidad del software, y para lograrlo se suelen lograr distintos objetivos importantes:

- Mejor calidad de código.
    - Aumentar la legibilidad, mantenibilidad, uniformidad, entre otros aspectos de calidad.
- Encontrar defectos.
    - Hallar vulnerabilidades, mejoras de performance o correcto uso del código reutilizable.
- Transferencia de conocimiento.
    - Lograr la transferencia del conocimiento adquirido en el desarrollo por el autor hacia los revisores.
    - No sólo el revisor logra aprender y conocer mejor el código del implementador, sino que también ambos comparten sus conocimientos técnicos y mejoran así sus habilidades como programadores.
- Incremento de responsabilidad mutua.
    - Tanto el autor como los revisores se sienten con la responsabilidad y propiedad del desarrollo.
    - Esto permite que más personas puedan atender cualquier necesidad que surja relacionada con la porción de código revisada.
- Encontrar mejores soluciones al problema.
    - Hallar caminos de solución distintos que pueden ser más simples y/o más fáciles de mantener a futuro.
- Cumplir con normas de calidad como ISO/IEC.
    - Muchas organizaciones incluyen a las revisiones de código como parte de los procesos certificados por ISO/IEC, por lo que su realización es obligatoria y vital para la organización.

La realización de revisiones de código en contextos de trabajo en equipo se vuelven más una necesidad que un agregado de calidad al software producido. Sólo entre un 10% a un 35% de las revisiones no han requerido de cambios en el código. Estos cambios en el código a partir de revisiones son muy frecuentes para sistemas de larga vida que requieren alta mantenibilidad y evolucionar en el tiempo. [13]

Las revisiones de código deberían integrarse con los procesos existentes del equipo de desarrollo. Por ejemplo, si un equipo utiliza un esquema de flujo de tareas, se inicia la revisión luego de que el código haya sido escrito y se hayan ejecutado satisfactoriamente los test automatizados, pero antes de hacer el merge hacia la versión correspondiente. Así, el tiempo

de la revisión se usa para detectar problemas que las máquinas no ven y se evita corromper el flujo normal del desarrollo [14].

La ejecución de revisiones de código tienen una gran importancia en mejorar la calidad del software, más aún en organizaciones grandes o en crecimiento donde el código crece indiscriminadamente y su mantenimiento llega a ser tedioso, y sobre todo muy costoso.

Sin embargo, las ventajas de implementar las revisiones aplican a todo tipo de organizaciones. La transferencia de conocimiento, por ejemplo, nos garantiza que una nueva funcionalidad del código no sea propiedad única de quien la desarrolló, sino que varias personas compartan esa propiedad y sobre todo, que cualquiera pueda responder rápidamente cuando dicha funcionalidad falle y se necesite revisar y aplicar correcciones. Ésto es muy cierto sobre todo en equipos de trabajo que cambian de miembros regularmente, haciendo la transferencia de conocimiento una necesidad [15]. El autor del código puede tomarse un descanso cuando quiera libremente y confiar en que sus compañeros participantes de la revisión atiendan ante un fallo.

Las revisiones de código se pueden usar también como herramienta de aprendizaje para las nuevas incorporaciones de una organización. Miembros más experimentados pueden conducir una revisión con nuevos miembros explicando el funcionamiento del código e interiorizando a los mismos a distintas prácticas de desarrollo de la organización. También puede ocurrir que la mirada nueva de un miembro recién llegado pueda observar alguna mejora o detectar algún fallo que el resto no haya encontrado al estar condicionados por el desarrollo continuo dentro de la organización.

2.4 Análisis de Código Fuente y Abstract Syntax Trees (AST)

En las revisiones de código se trabaja justamente con código de software, que es analizado y manipulado en pos de garantizar la calidad. Sin embargo, las máquinas no comprenden el código de la misma forma que lo hacen los seres humanos. Las máquinas necesitan de compiladores e interpretadores que traducen el código escrito por humanos a lenguajes entendibles por una máquina.

Los compiladores de código generalmente pasan por las siguientes etapas [16]: Análisis Léxico, Análisis Sintáctico, Análisis Semántico, Generación intermedia de Código, Generación de Código. El interés de este trabajo recae principalmente sobre las primeras 3 etapas. En estas se identifican las palabras claves del lenguaje, se analiza la sintaxis ,y la semántica y consistencia del código para encontrar errores o posibles mejoras.

En el análisis sintáctico del código se genera la estructura de Árbol de Sintaxis Abstracta o Abstract Syntax Tree (AST). Un AST es la representación del código fuente del programa como un árbol de nodos donde cada nodo representa una parte específica del código. A medida que crecen los niveles del árbol (se acerca a las hojas), los nodos representan estructuras más pequeñas del código, llegando hasta las constantes o variables. Por ejemplo, un nodo de bajo

nivel podría ser la definición de un método mientras que uno del más alto nivel podría ser el valor de una asignación a una variable [17][18].

A continuación se detalla un código básico de ejemplo en Smalltalk y cómo queda el AST resultante luego del parseo por parte del compilador.

```
getRaceResults: aGrandPrix

    | racePositions |
    racePositions := aGrandPrix drivers.

  "Calcular resultados"
  ^ racePositions shuffle.
```

Figura 1. Ejemplo del parseo de código hacia un AST

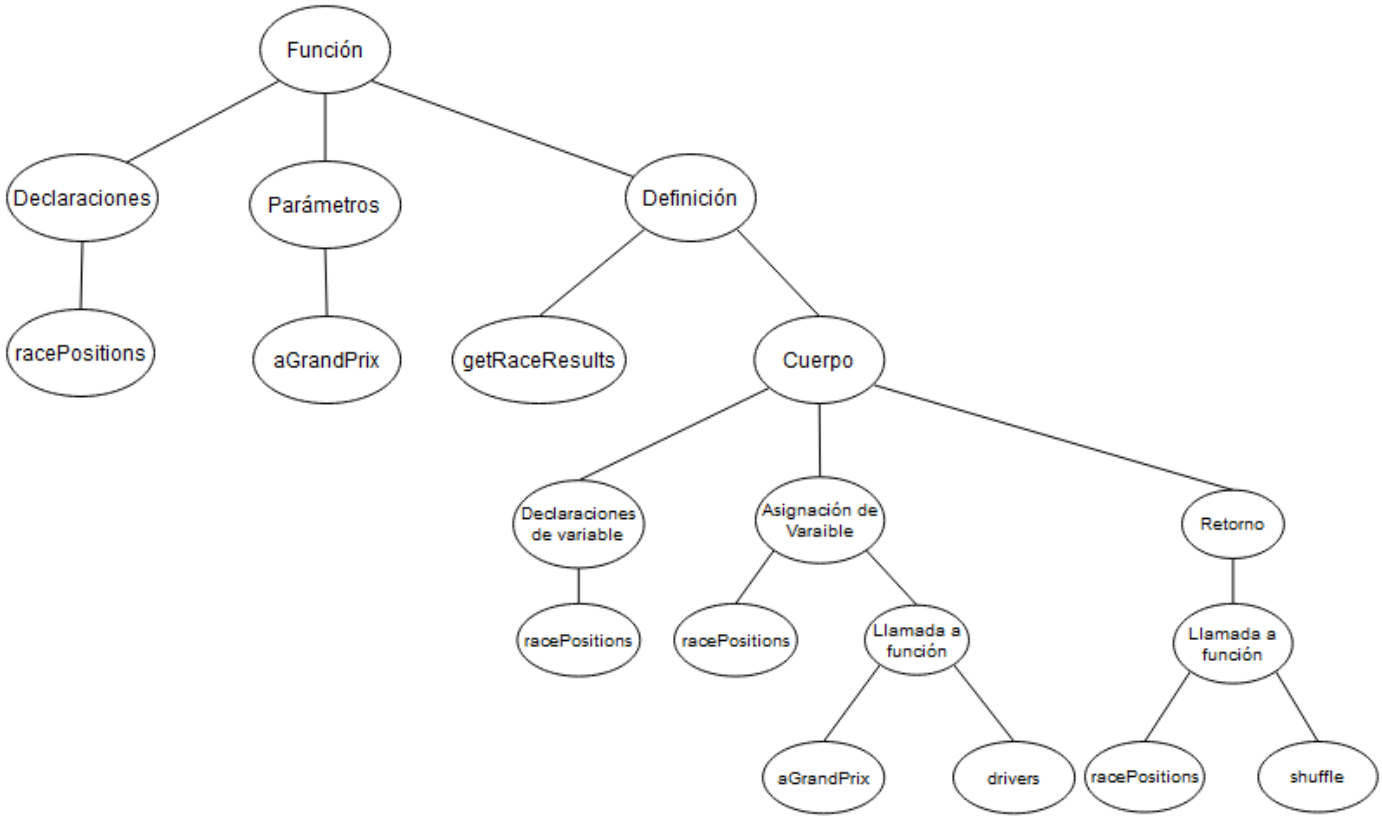


Figura 2. Ejemplo de árbol abstracto sintáctico del código de un método.

En el proceso de compilación el AST tiene como función ser la base desde donde se realizarán los análisis sintácticos y semánticos del programa. Para un software como un compilador, el análisis es mucho más eficiente y correcto si trabaja con estructuras como árboles sintácticos, ya que entiende perfectamente cómo recorrer un árbol y no tanto como recorrer una secuencia de caracteres compleja. Si se detecta una estructura errónea dentro del árbol (faltante de

nodos o en lugares incorrectos) quiere decir que el código tiene errores sintácticos, por lo que no puede ser ejecutado.

En la figura siguiente hay un código con un error sintáctico al realizar una asignación de la variable *racePositions* pero no asignarle un valor correcto.

```
getRaceResults: aGrandPrix

    | racePositions |
    racePositions :=

   "Calcular resultados"
   ^ racePositions shuffle.
```

Figura 3. Ejemplo del parseo de un código erróneo sintácticamente

Al intentar generar un AST correcto para el código, el compilador se encontraría con que falta un nodo para que la secuencia de sea correcta:

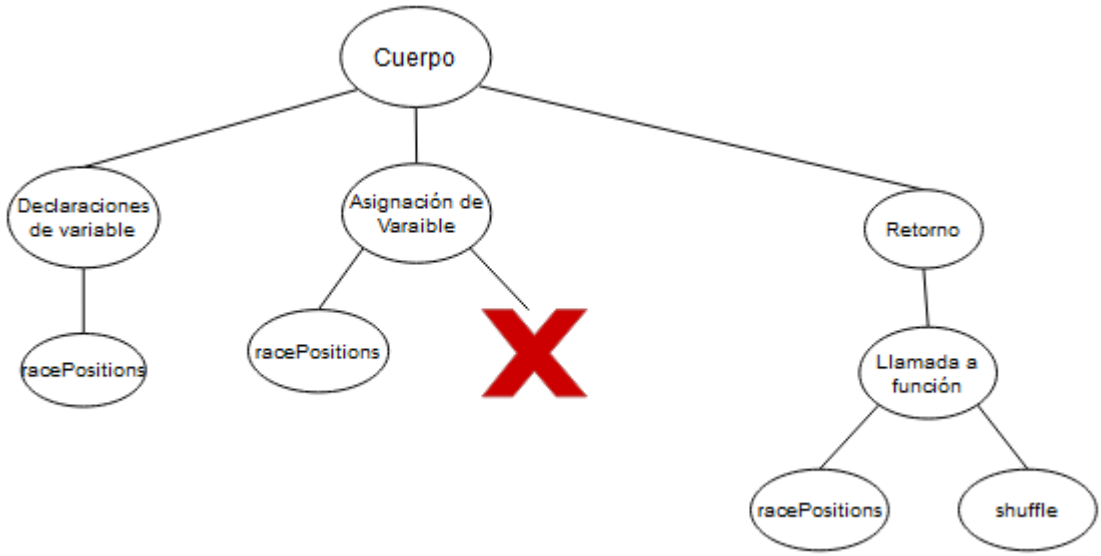


Figura 4. AST que no se pudo parsear de un código erróneo sintácticamente.

Como falta el nodo en la asignación de la variable, el AST no se puede completar por lo que tampoco se puede completar correctamente el Análisis Semántico del código. Entonces el compilador indica que el código es erróneo.

Esta introducción a los AST es de interés porque en el trabajo se utilizan al realizar comentarios en una revisión de código. Esto quiere decir que cuando un revisor realice un comentario sobre una porción de código, no lo hará sobre la secuencia de caracteres correspondiente a esa porción, sino que se aplicará directamente al nodo del árbol que representa a esa porción.

2.5 Soluciones existentes

Dentro de la industria del software existen algunas herramientas existentes similares a la propuesta por este trabajo que buscan ayudar a hacer más eficiente y cómodas las revisiones de código en un equipo de trabajo. Principalmente hablaremos de dos alternativas en cuanto a la forma en la que implementan la herramienta de Code Review; una busca integrar el Code Review con el IDE como CodeCaption y la otra alternativa se basa en el sistema de pull request[3] mediante un servicio web separado para hacer las revisiones.

La primera alternativa es implementada por el plugin UpSource de JetBrains. UpSource es un producto de JetBrains que por sí solo provee toda una aplicación para realizar todo tipo de seguimiento a los cambios de un proyecto versionado, ver diferencias en el código, revisar issues, agregar code reviews y comentarios simples en el código. La aplicación provee muchas herramientas para facilitar el trabajo en equipo pero además se puede incluir dentro del IDE mediante un plugin específico para cualquiera de los IDE de JetBrains (IntelliJ Idea, PhpStorm y más).
Con este plugin es posible entonces integrar las herramientas provistas por UpSource con el IDE en el que escribimos el código del proyecto. Así, tenemos una interfaz gráfica que emula a la versión web de UpSource pero dentro del entorno de desarrollo, evitando la incomodidad y frustración de tener que cambiar de aplicaciones para las revisiones de código [5]. Este enfoque es igual a lo que hace CodeCaption, integrando las revisiones de código dentro del IDE para evitar usar más de una aplicación a la vez.
Sin embargo, esta alternativa requiere tener corriendo un servidor de UpSource para poder guardar y sincronizar todas las revisiones de código y seguimientos al código del proyecto. En el caso de CodeCaption esto se hace utilizando la herramienta de versionado del proyecto, registrando las revisiones como archivos dentro del repositorio. Por esto mismo, no se requiere instalar ningún tipo de servidor ni software adicional.

Para la otra alternativa de herramientas de revisión de código tenemos varias opciones. La mayoría de estas opciones son aplicaciones web que se integran con herramientas de versionado para permitir a los usuarios realizar revisiones sobre el código del repositorio.

Una de estas herramientas es Azure DevOps. Se trata de una aplicación que nos permite revisar los pull request que existen dentro del proyecto, visualizar los cambios realizados y agregar comentarios a los mismos. A diferencia de UpSource, Azure DevOps como otras herramientas no permiten crear revisiones de código proactivamente, sino que nos posibilita agregar comentarios en los pull request existentes. Así, por ejemplo, no podríamos agregar revisiones de código hasta que el desarrollador de una funcionalidad no la complete y realice un pull request en el repositorio.
Sin embargo, es posible integrar la funcionalidad descrita al IDE de Microsoft Visual Studio Code (VSCode) mediante un plugin (llamado extensión dentro del VSCode) de Azure DevOps.

[3] Un pull request es una solicitud del propietario de una rama git con nuevos cambios para que dichos cambios se revisen y se combinen (merge) con la rama principal productiva del proyecto (generalmente master).

Allí entonces tenemos la posibilidad de crear nuevas pull request o revisar las existentes y agregar comentarios al código para así realizar una code review dentro del contexto del IDE [19].

El mismo escenario ocurre en los servicios web basados en git como GitLab y GitHub. Ambas son plataformas web de desarrollo colaborativo para mantener el código versionado de un proyecto en la web. Contienen todas las herramientas necesarias para tener un versionado completo de un proyecto, poder trabajar en equipo remotamente, revisar cambios, volver atrás cambios y demás.

Con respecto a las revisiones de código github permite que un desarrollador luego de realizar un pull request pueda requerir una revisión por parte de un miembro del equipo. Allí dicho miembro puede revisar los cambios y realizar comentarios de ser necesario, y/o directamente aprobar la review para que se concrete el pull request y se haga merge con la rama principal del código. También es posible agregar comentarios en los commits del proyecto, en una vista donde se visualizan los cambios en líneas de código, aunque dichos comentarios no necesitan ser revisados para que los cambios se concreten. [20]
En el caso de GitLab es casi idéntico al de GitHub, se permite crear pull request (llamadas merge request en este caso) para revisar el código a mergear posibilitando notificar a miembros específicos. Una vez creado el merge request los miembros pueden dejar comentarios en los cambios de código del mismo, que se van actualizando si nuevos cambios fueron subidos al repositorio. Si la persona indicada para revisar los cambios cree que está todo correcto se procede con el merge, sino puede solicitar más cambio mediante los comentarios o directamente cerrar el merge request. [21]

En estas últimas tres alternativas se integran las funcionalidades de revisiones de código a la herramienta de versionado y su repositorio del proyecto. Ya sea mediante comentarios en los pull request o en los archivos en sí, es posible revisar código remota y asíncronamente entre los colaboradores como sucede en CodeCaption. La diferencia con este último es que estas aplicaciones no cuentan las revisiones como nuevos cambios en los archivos del repositorio del proyecto, y el historial y seguimiento de las mismas dependen del servidor de la herramienta de versionado (GitHub, GitLab, etc). También la gran mayoría de este tipo de aplicaciones no tiene una integración directa con el IDE, sino que son aplicaciones web aparte.

2.6 CodeCaption en el Cumplimiento de las Métricas

Con la herramienta a desarrollar en este trabajo se buscará ayudar al desarrollador a poder cumplir con las métricas mencionadas en la sección 2.1.2 más fácilmente. El hecho de trabajar en un equipo remoto y poder dejar comentarios dentro del mismo código visibles para todos los integrantes del mismo hace más fácil unificar conceptos para cumplir con las métricas. Resulta entonces fácil para el revisor indicar en qué parte del código se requiere un cambio que haga que el mismo cumpla con lo mencionado, tanto como para legibilidad, mantenibilidad, reducir complejidad, etc.

Capítulo 3. Diseño

El diseño de una aplicación es una etapa clave en el proceso de construcción de la misma. Aquí se analiza y plantea cómo atacar la problemática, logrando así un plano estructurado como base desde donde comenzar a trabajar en la etapa de implementación.

3.1 Objetivos del diseño de la herramienta

El diseño de la herramienta CodeCaption se basa en 2 objetivos principales:

- Brindar la posibilidad de crear, visualizar y manipular revisiones de código como comentarios dentro del entorno de desarrollo (IDE[4]).
- Lograr que estas revisiones se sincronicen con todos sus cambios en el repositorio remoto del proyecto. Esto va a permitir que las revisiones sean vistas y manipulables por todos los miembros del equipo de desarrollo.

Con esto se logra una herramienta de gran valor para el desarrollador y sus pares al poder realizar revisiones de código remotamente sin necesidad de siquiera cambiar de aplicación.

CodeCaption es en términos prácticos una extensión del IDE Pharo, utiliza las herramientas y framework provistos por el mismo para crear una herramienta propia con una interfaz gráfica simple que permita al desarrollador realizar las revisiones de código. Quizás el aspecto más importante de la interfaz gráfica que añade CodeCaption al entorno es la posibilidad de seleccionar una línea o múltiples partes del código real del proyecto y agregar un comentario de revisión a esa porción, integrando fuertemente la programación con la revisión. Esta funcionalidad está inspirada en la herramienta Google Docs, que es un software de procesamiento de texto pero que cuenta con una opción para agregar comentarios dinámicamente a porciones del texto redactado para que todos los involucrados en el documento lo puedan ver. Estos comentarios pueden ser respondidos, eliminados o resueltos y son de gran utilidad para trabajos en equipo, ya que nos permite comentar una porción específica del texto dentro del mismo entorno de redacción, sin necesidad de una herramienta adicional.

La unificación de estos conceptos no solo es conveniente para quien está desarrollando el proyecto y quiere realizar comentarios sobre el código del mismo. También permite a los involucrados tener un orden mayor sobre los distintos cambios y solicitudes de cambios presentes durante el proceso de desarrollo, logrando mejorar el entendimiento del proceso.

[4] Un entorno de desarrollo integrado o entorno de desarrollo interactivo, en inglés Integrated Development Environment (IDE), es un programa informático que provee servicios integrales al desarrollador o programador para que pueda realizar con facilidad un desarrollo de software.

3.2 Diseño de la mecánica de uso

La herramienta de CodeCaption se ejecuta dentro del contexto de un IDE, por lo tanto hay que considerar la integración entre ella y el IDE para su diseño. Teniendo en cuenta esto entonces se pueden comenzar a pensar las diferentes funcionalidades de la herramienta y cómo estas funcionalidades se podrán utilizar dentro del contexto del IDE.

3.2.1 Uso de la interfaz gráfica

Como se mencionó en la primera sección de este capítulo, un objetivo principal del diseño de CodeCaption es poder agregar y manipular revisiones de código. Por eso, es vital poder agregar revisiones dentro de la misma vista de edición de código, identificando la porción de código a revisar y seleccionándola para agregar nuestro comentario. A continuación se detalla mediante bocetos el diseño de la interfaz gráfica de la herramienta dentro del IDE.

La siguiente figura presenta un método dentro del IDE que va a ser utilizado como ejemplo a lo largo de este capítulo.

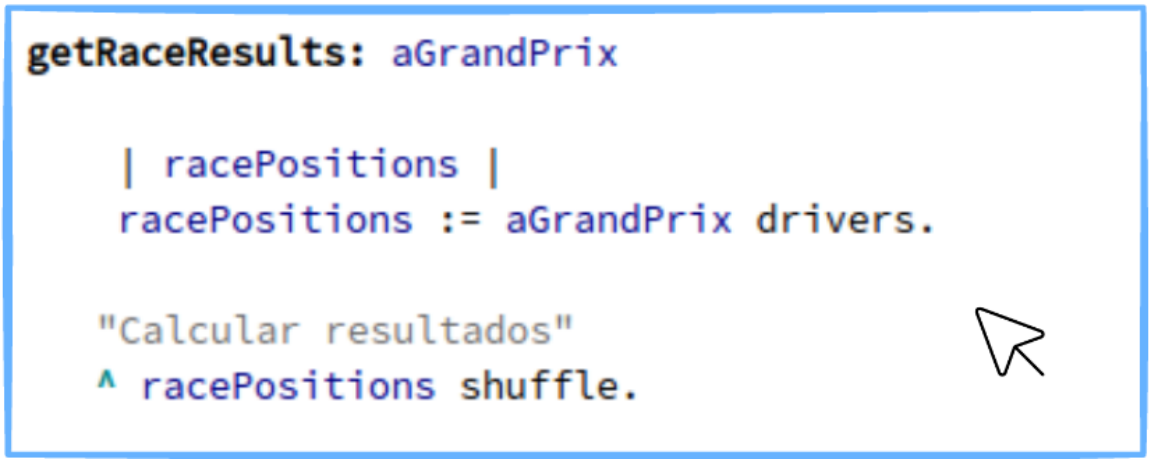


Figura 5. Código Fuente de un método en Smalltalk

El código de la figura comprende una función que calcula los resultados de un Gran Premio (carrera de automóviles). La lógica es muy simple, solo realiza un ordenamiento al azar de los diferentes pilotos participantes del Gran Premio, y toma ese orden como resultado a retornar.

Es probable que un desarrollador revisando el código quiera que esa lógica sea mejorada y esté más cerca a lo que sería una simulación de una carrera real. Es decir que a cambio de realizar un orden azaroso, se calcule los resultados en base a las diferentes habilidades de los pilotos y demás factores que inciden en el rendimiento de los mismos en la carrera. Entonces al analizar el código, el revisor selecciona la sección que calcula los resultados y llama a agregar un comentario:

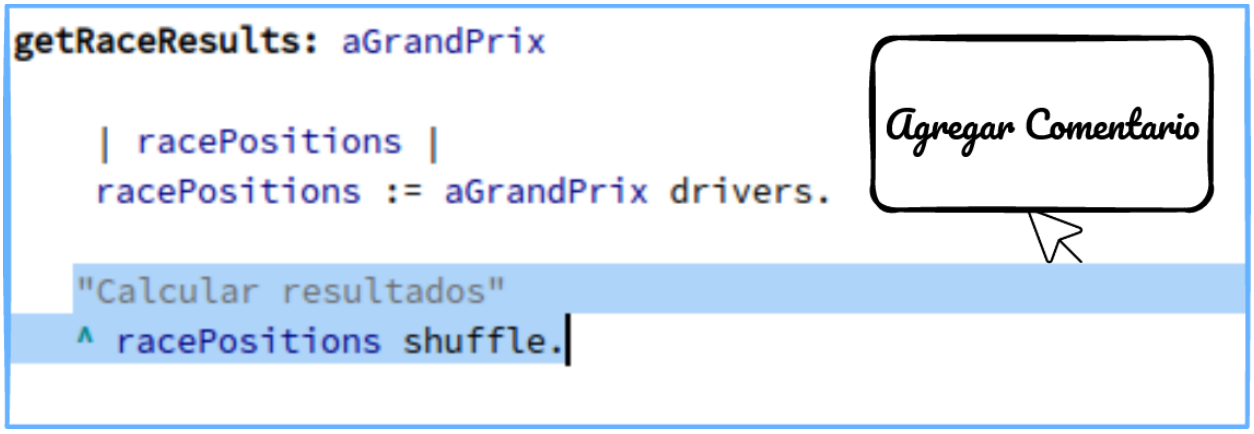


Figura 6. Boceto de una selección de código a revisar en el entorno.

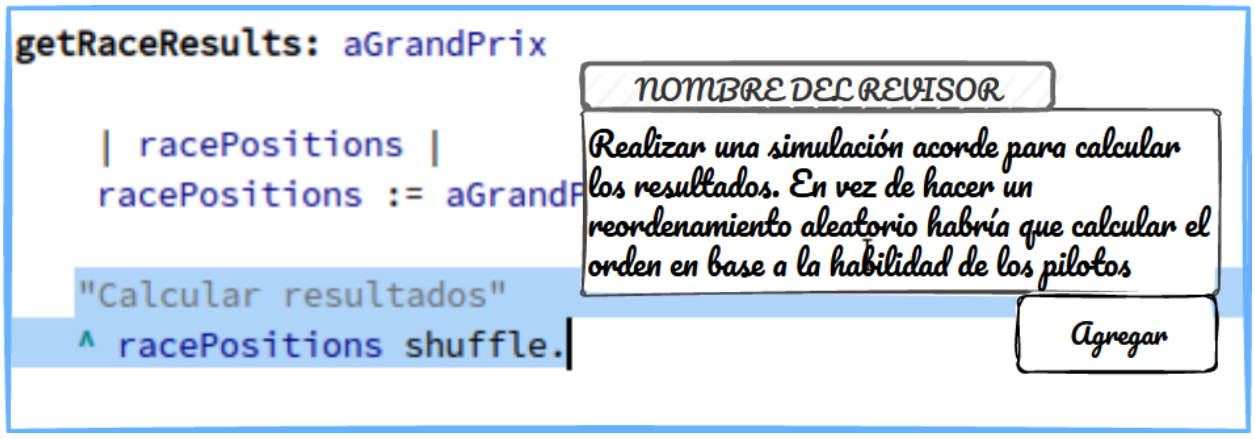


Figura 7. Boceto de redacción y agregado del comentario de la revisión de código dentro del entorno.

En este paso la herramienta debería asociar el comentario que el revisor está a punto de agregar con la porción de código seleccionada.

Entonces la porción de código ahora tiene asociada la revisión con su comentario. En este ejemplo se muestra la revisión con el comentario en texto, el autor y un botón para “resolver” la revisión.

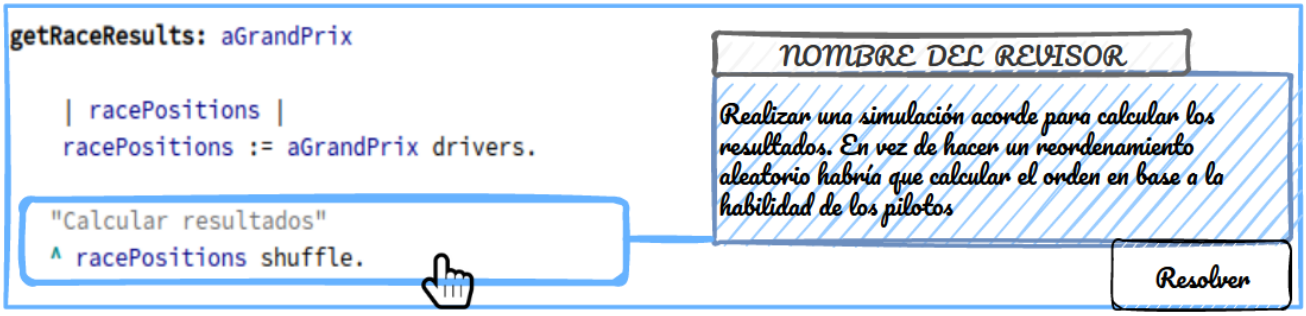


Figura 8. Boceto de visualización de la revisión de código una vez creada con la referencia al código en cuestión.

El recuadro de la izquierda como dijimos anteriormente nos indica que esa porción de código es a la que se le ha hecho la revisión mostrada en la derecha.

Se plantea ahora el escenario donde el desarrollador realiza una modificación y altera el código luego de creada la revisión. Sin embargo, la modificación no es sobre la porción de código revisada, sino que se aplica sobre el mismo método pero en una sección distinta. Esto se demuestra en la siguiente figura, donde se modifica el código anterior al cálculo de los resultados, pero este último se mantiene igual.

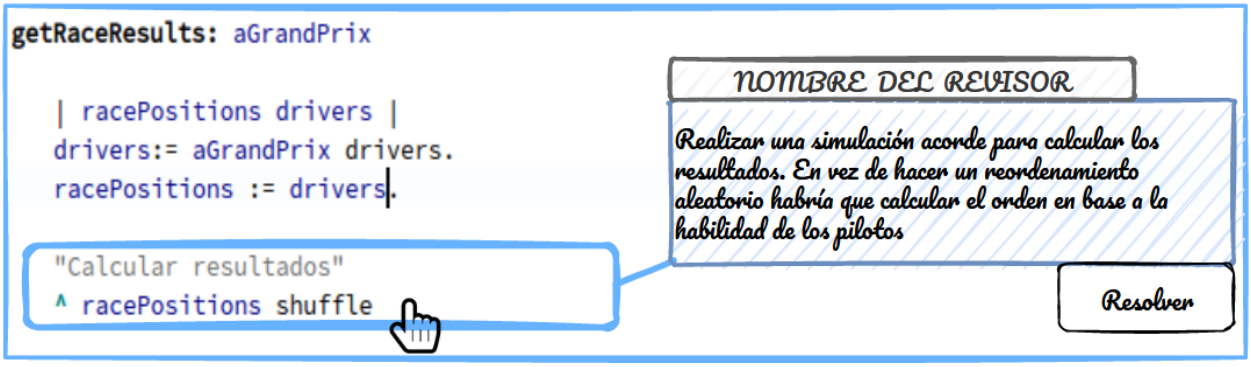


Figura 9. Boceto de visualización de la revisión luego de cambios en el código del método ajenos a la revisión.

Ahora bien, el caso en el cual se modifica el código al que la revisión hace referencia presenta un escenario distinto. El comentario de la revisión entonces ahora es sobre un código que ya no está presente en el proyecto. Si el código no está presente se pierde la referencia, así que debemos actualizarla de alguna manera. En este punto se determina automáticamente que la revisión no quede asociada a ninguna sección específica, sino que quede asociada al método que contenía el código alterado como se muestra en la figura.

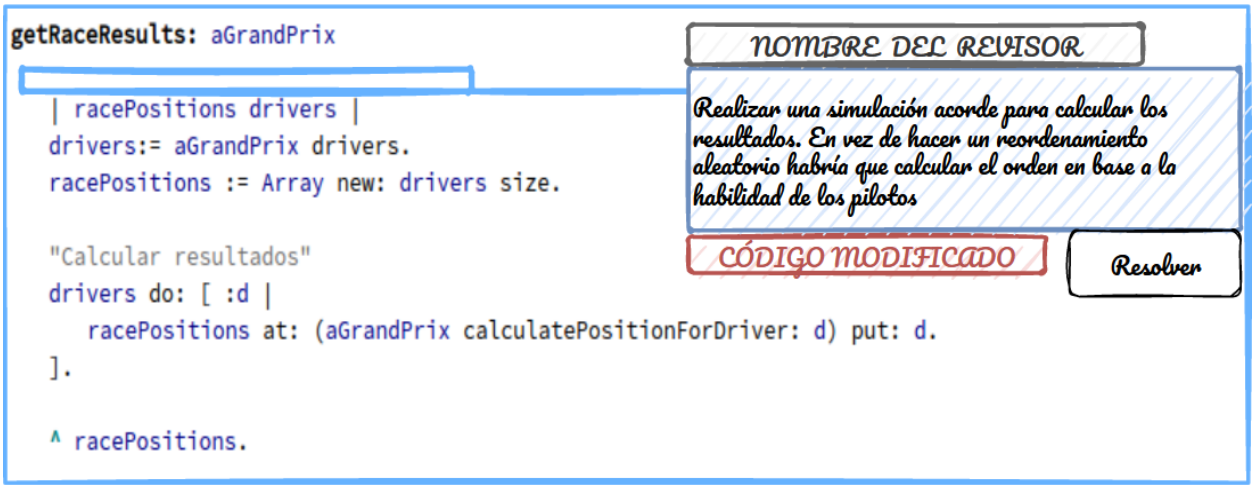


Figura 10. Boceto de visualización de la revisión luego de cambios en el código referenciado por la misma.

Esto puede significar que el autor del código original leyó el comentario dejado por el revisor y se puso a trabajar en corregir la lógica y agregar el código necesario. En el caso en cuestión se ve como luego de las modificaciones ahora se recorren los pilotos de la competición y se llama a un método aparte para calcular la posición final en el gran premio, realizando un cálculo más parecido a una simulación y no dejarlo al azar.

En este punto el autor del código puede resolver la revisión mediante el menú de interacción que aparece al ver la misma. Esto hará que la revisión desaparezca del contexto de la función en el IDE y se marque como resuelta. La revisión no se elimina por completo, quedarían sus cambios asociados a la herramienta de versionado usada en el proyecto pero ya no se verá en el IDE como pendiente para corregir. Al resolver la revisión luego de los cambios, el IDE se verá ahora sin ninguna revisión pendiente a considerar.

```
getRaceResults: aGrandPrix

   | racePositions drivers |
   drivers:= aGrandPrix drivers.
   racePositions := Array new: drivers size.

   "Calcular resultados"
   drivers do: [ :d |
      racePositions at: (aGrandPrix calculatePositionForDriver: d) put: d.
   ].

   ^ racePositions.
```

Figura 11. Código resultante de la modificación del código revisado.

3.2.2 Cambios de las estructuras de datos en el uso de la herramienta

En la mecánica de uso de la herramienta suceden distintas transformaciones en las estructuras de datos utilizadas por la herramienta para su funcionamiento.
En el capítulo 2 fue explicado el concepto de los Abstract Syntax Trees (AST), además de su importancia en el desarrollo y el uso que se le da en este proyecto. Cuando se escribe código en el entorno de desarrollo, por detrás se está realizando un parseo y armado de un AST que representa a este código. El AST resultante del cuerpo del método de ejemplo utilizado en esta sección (figura 1) es algo del estilo:

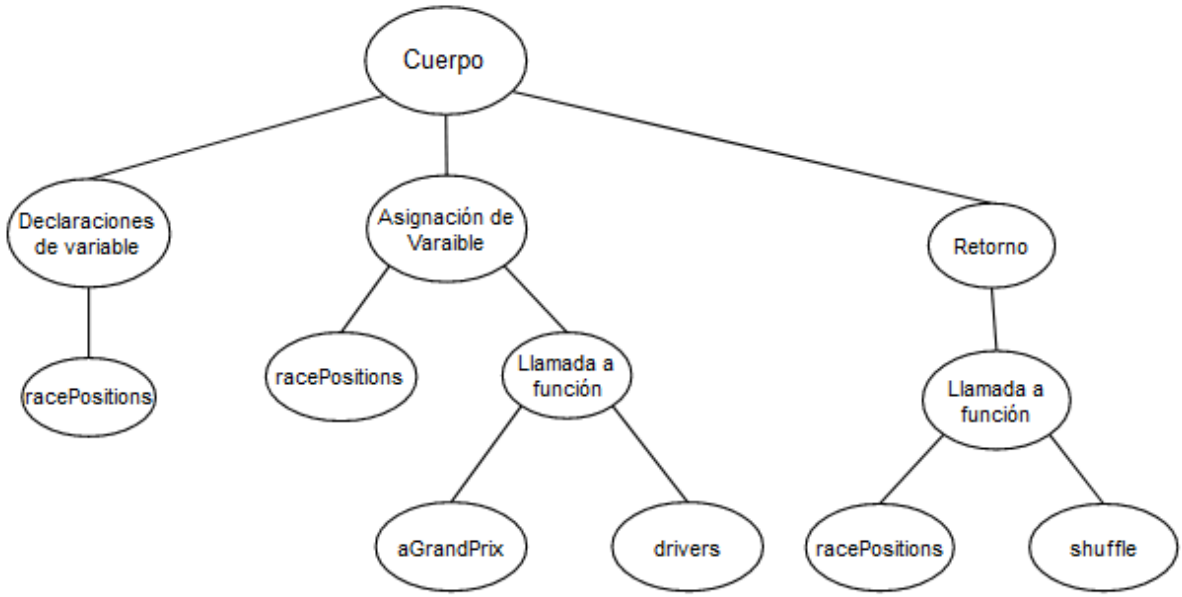


Figura 12. AST del método de ejemplo utilizado en la sección.

En la figura anterior se puede observar cómo se dividen las distintas partes del método en hijos del nodo cuerpo del mismo. En este caso se separan las declaraciones de variables del resto de las sentencias del método y la sentencia que interesa para el ejemplo es la de retorno. Cuando el revisor realiza un comentario al código seleccionando la sección del mismo donde se calculan y retornan los resultados (figura 6), el sistema identifica a esa sección con el nodo de retorno del AST y asocia la revisión CodeCaption creada a ese nodo. Cabe recalcar que este proceso no implica simplemente asociar el comentario con el contenido textual del código sino que debe hacer referencia al nodo retorno del AST que comprende al método.

Por consiguiente, la revisión de CodeCaption recién creada asocia el comentario con el último hijo del nodo cuerpo del AST, el nodo de retorno:

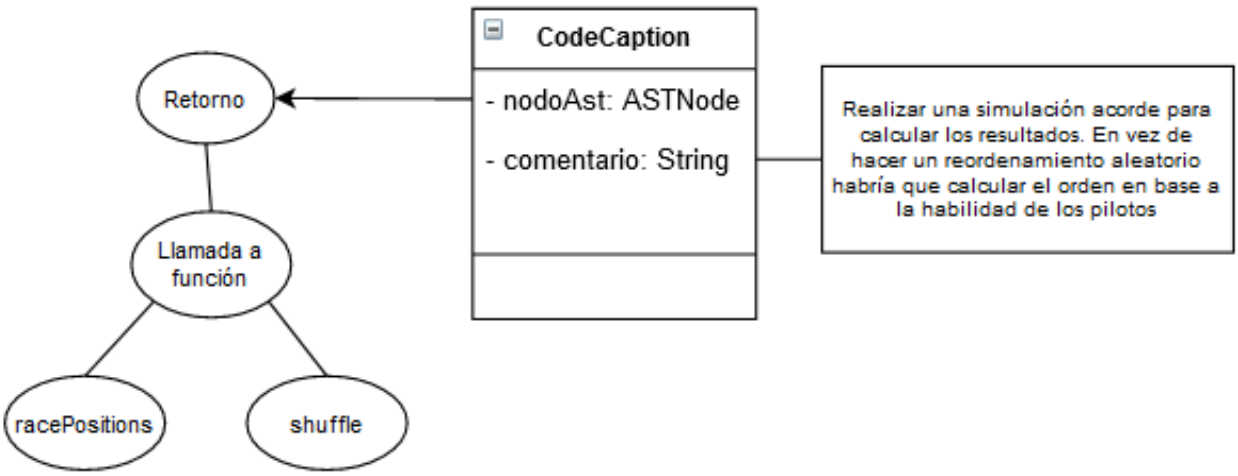


Figura 13. Ejemplo de cómo se asocia la revisión CodeCaption al nodo del AST.

Este será entonces el resultado de la creación del nuevo CodeCaption, con la referencia al nodo retorno del AST de la función y el comentario ingresado por el usuario.

Si el desarrollador realiza modificaciones al código del método pero deja intacto el código referenciado por la revisión (figura 9), la misma no sufrirá ningún cambio. Como se utiliza una asociación de nodo AST con la revisión y no otra metodología como podrían ser los números de línea o directamente el texto del código. Mientras el nodo en cuestión no sea alterado no importa si el resto del código del método lo es, la referencia de la revisión hacia el nodo queda intacta.

Las modificaciones del AST producto de este cambio de código entonces sería la siguiente.

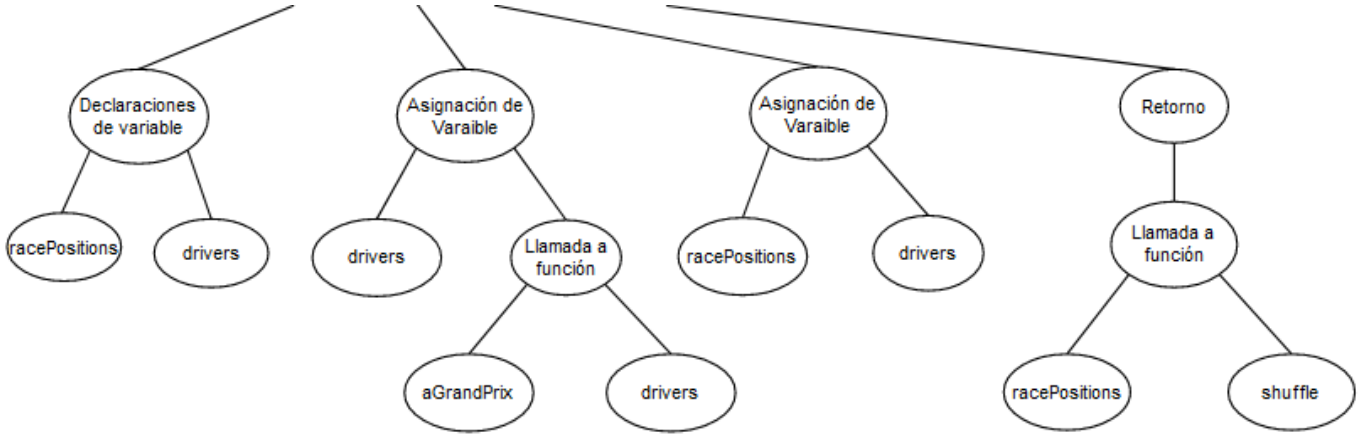


Figura 14. AST resultante de la modificación de código del método ajeno a la revisión.

Aquí se puede apreciar como el nodo de interés para la revisión queda intacto, ahora solo tiene hermanos distintos pero su contenido no cambió, así que la referencia se mantiene correctamente.

En el escenario en que el desarrollador realice modificaciones al código referenciado por la revisión (figura 10), ocurre que la referencia de la revisión termine siendo alterada. Las modificaciones del código resultan en un cambio más profundo del AST, cuyo nodo de retorno, así como más nodos del cuerpo, sufrieron modificaciones. La siguiente figura detalla al nodo retorno luego de las modificaciones y qué ocurre con la revisión CodeCaption.

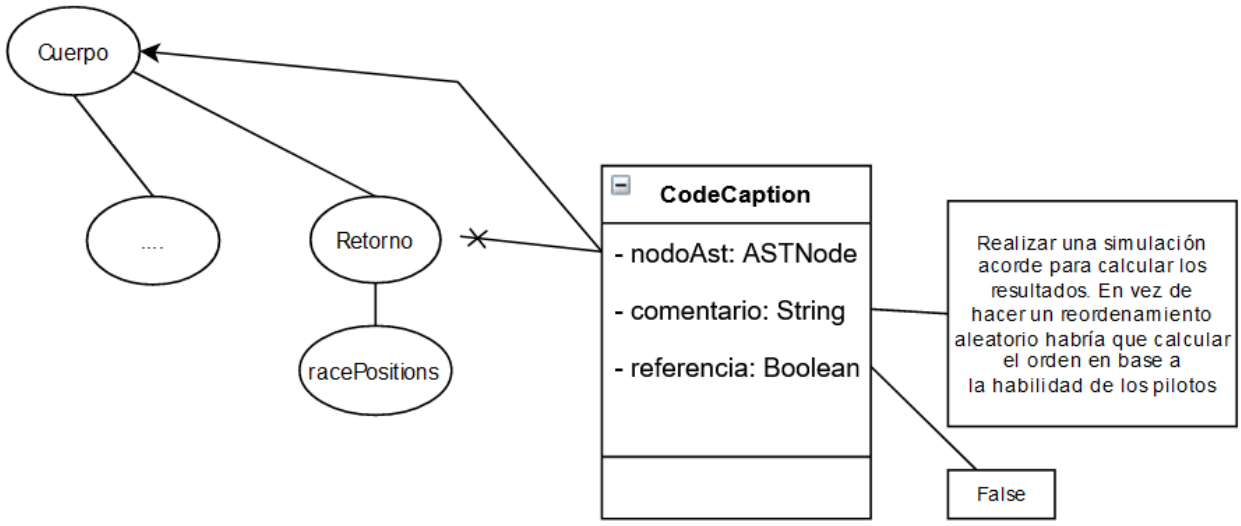


Figura 15. AST del nodo retorno luego de las modificaciones y cambios en la revisión CodeCaption.

En este último caso el nodo de interés ha sido modificado, por lo que se pierde la referencia correcta al nodo desde CodeCaption. Al perderse la referencia, la revisión CodeCaption ahora hará referencia al nodo método que contiene el nodo retorno que representaba la sección de código revisada, es decir que ahora simplemente hará referencia al cuerpo entero del método e informa que la referencia original se ha perdido.

Luego, al realizarse la resolución de la revisión (figura 11), la misma queda eliminada del sistema, sin ningún código referenciado ni comentario.

3.3 Casos de Uso

A continuación se detallan cuatro casos de usos posibles de la herramienta en forma de historias de usuario. En estas, dos colaboradores de un proyecto desarrollan y luego revisan el código desarrollado, realizando todas las tareas con la herramienta. Todo este proceso ocurre dentro del entorno de desarrollo Pharo Smalltalk, sin necesidad de software complementario.

3.3.1 Revisor analiza el código de un desarrollador en el proyecto

Un desarrollador llamado Pedro realiza una modificación al código del proyecto agregando un nuevo método (*race*) en una clase existente (*GrandPrix*). En este ejemplo simple, el método simula una carrera de coches, y calcula los resultados finales de los pilotos.

```
race
    | drivers resultArray |
    resultArray:= OrderedCollection new.
    drivers:= teams flatCollect: [ :t | t drivers ].
    raceResults:=  drivers shuffled.
```

Figura 16. Código de ejemplo a revisar.

Luego de realizar la modificación, Pedro sube los cambios al repositorio del proyecto para sincronizar con el resto del equipo. En este punto una revisora llamada Paula entra al entorno de desarrollo y hace una actualización del repositorio para obtener los últimos cambios que Pedro agregó. Al revisar el código Paula encuentra que, a pesar de no tener errores, el método debería retornar los resultados de la carrera en lugar de guardarlos en una variable de instancia (*raceResults*). Entonces Paula resalta el código donde se guardan los resultados en la variable de instancia y agrega un CodeCaption desde el menú contextual.

Al solicitar el agregado de una revisión CodeCaption se habilita una ventana para introducir el comentario. Paula entonces deja sentada la revisión con un texto explicativo y llama a agregar el CodeCaption. Una vez agregado, automáticamente la herramienta guarda la información en un archivo en disco dentro del directorio del proyecto y realiza un *commit*[5] para confirmar la operación. Cuando Paula suba al repositorio remoto sus cambios locales, el nuevo CodeCaption estará disponible para el resto del equipo cuando actualicen sus proyectos locales en el entorno.

Hasta este punto, la línea de tiempo del caso de uso quedaría de la forma detallada en la siguiente figura.

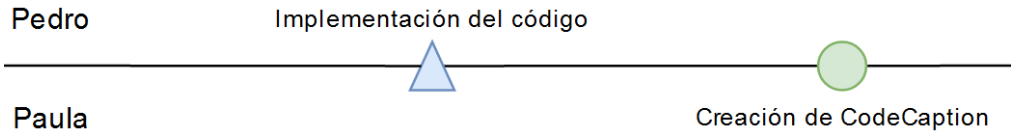


Figura 17. Línea temporal de la creación de un CodeCaption.

3.3.2 Desarrollador recibe revisión y realiza correcciones

El desarrollador Pedro actualiza su proyecto localmente sincronizando con el servidor remoto del repositorio y encuentra nuevos cambios. Dentro de estos cambios se encuentran los cambios en las revisiones CodeCaptions, que pueden visualizarse dentro del entorno al presionar un botón.

Dentro de las revisiones, Pedro encuentra una revisión creada por la revisora Paula al código de un método que estuvo desarrollando (figura 13). Al seleccionar la revisión de Paula, se abre la descripción de la misma en una nueva ventana donde se ve la revisión con el texto y la parte del código fuente que fue revisada. Allí Pedro puede leer el comentario dejado por Paula y decidir si es necesario modificar el código. El comentario de Paula indica que no es necesario guardar los resultados en una variable de instancia, con sólo retornar los resultados inmediatamente está correcto. Pedro entonces decide proceder a realizar la modificación en el código, expresada en la figura.

[5] Operación de programa informático que indica que un conjunto de cambios "tentativos, o no permanentes" se conviertan en permanentes.

```
race
    | drivers |
    drivers:= teams flatCollect: [ :t | t drivers ].
    ^ drivers shuffled
```

Figura 18. Código de ejemplo luego de la modificación.

Luego de la corrección del código sugerida en el CodeCaption por parte de Pedro, quedaría una línea de tiempo final del estilo representado en la siguiente figura.

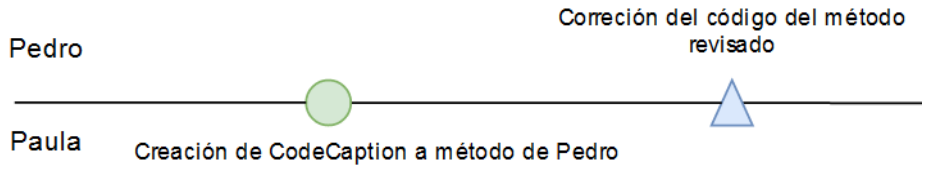


Figura 19. Línea temporal de la corrección del código revisado en un CodeCaption

3.3.3 Revisor resuelve la revisión luego de aplicarse las correcciones

La revisora Paula actualiza su proyecto local con el repositorio remoto y se encuentra con distintos cambios. Dentro de estos cambios nota que ha sido modificado un método creado por el desarrollador Pedro, al que ella le había hecho una revisión. Al analizar los cambios de Pedro, Paula determina que la corrección es válida, por lo que la revisión CodeCaption ya no es necesaria. Incluso, en este escenario la referencia al código de la revisión ya se perdió, porque el código referenciado fue modificado por Pedro. Entonces, ahora la revisión hace referencia al método entero.

Para solucionar esto, abre la revisión que había creado sobre ese método para luego llamar a resolverla. La revisión CodeCaption entonces queda ahora resuelta, y cuando Paula vuelva a subir los cambios al repositorio remoto, quedará también actualizado el CodeCaption para el resto del equipo. En el caso de la resolución, se elimina la revisión de la lista de revisiones del proyecto para que queden visibles solamente las que todavía no fueron revisadas.

Las operaciones de corrección del código y la resolución del CodeCaption quedan registradas en el repositorio del proyecto. Esto nos permite tener conocimiento de, además de los cambios en el código, las revisiones creadas con la herramienta, sus cambios y su posterior resolución, quedando así borrada del proyecto.

Con las correcciones al código aplicadas y la posterior resolución de la revisión, queda una línea temporal expresada en la figura.

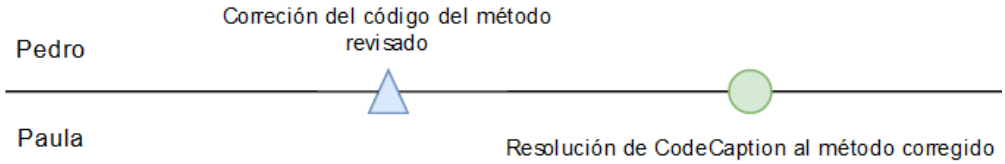


Figura 20. Línea temporal de la corrección de código y resolución de una revisión CodeCaption.

3.3.4 Desarrollador decide no aplicar correcciones a partir de revisión

Un escenario posible en el uso de la herramienta puede ser el de no realizar los cambios sugeridos por la revisión.

El revisor Pedro decide agregar un comentario al código creado por una desarrolladora que tiene un nombre de variable que cree que podría mejorarse.

```
race
    | drivers |
    drivers:= teams flatCollect: [ :t | t drivers ].
    ^ drivers shuffled
```

Figura 21. Código comentado por el revisor con el nombre de la variable seleccionada.

Aquí el revisor crea una revisión CodeCaption indicando que el nombre de la variable “drivers” podría cambiarse a un nombre más acorde como “driverPositions”.

Cuando la desarrolladora Paula actualiza su entorno y recibe la revisión, decide que el cambio sugerido por el revisor es innecesario ya que no es más que una diferencia de nomenclatura sin importancia real para el entendimiento del método. Por ende, procede a realizar la resolución de la revisión sin implementar ninguna modificación al código, quedando una línea temporal expresada en la figura:

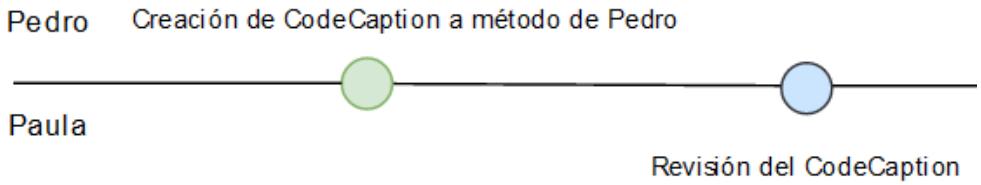


Figura 22. Línea temporal de la resolución de un CodeCaption sin modificación de código.

3.4 Flujo e Interacción entre los Objetos del Diseño

En este apartado se procede a explicar el flujo de los distintos procesos presentes en la herramienta CodeCaption y cómo interactúan los objetos entre sí para lograr un funcionamiento correcto. La explicación siguiente con diagramas de interacción se realiza en base a las operaciones realizadas en los casos de uso mencionados en el apartado anterior, enfocando específicamente en el agregado y la resolución de una revisión.

Para los distintos casos de uso explicados en el punto anterior, en los diagramas de interacción se pueden definir los siguientes objetos:

- Interfaz del IDE
    - Funcionalidad gráfica de la interfaz del entorno de desarrollo en el cual trabaja CodeCaption. Se interactúa con los distintos menúes para agregar o listar las revisiones de código, todo dentro del mismo editor del entorno.
- Code Caption Manager
    - Lógica propia del manejo interno de CodeCaption.
    - Instancia las clases necesarias para poder cargar, guardar y operar con las revisiones de código.
- Parser de CodeCaption
    - Encargado de leer, interpretar y cargar las revisiones de código de un proyecto hacia la herramienta dentro del IDE.
- Interfaz gráfica de CodeCaption
    - Funcionalidad gráfica de la interfaz del entorno de desarrollo perteneciente a CodeCaption. Contiene las ventanas de agregado, edición, listado y demás sobre las revisiones de código de un proyecto.
- Almacenador de CodeCaption
    - Funciona como la inversa del parser. Se encarga de guardar las nuevas revisiones de código y/o modificar o borrar las existentes en disco.
- Herramienta de versionado
    - Herramienta de versionado de código externa al IDE y CodeCaption donde se registran todos los cambios en las revisiones. Detecta los cambios en el archivo en disco de CodeCaption, y mediante comunicación desde el IDE se versionan los cambios para subirlos al repositorio remoto.

3.4.1 Diagrama de interacción para el agregado de un nuevo Code Caption

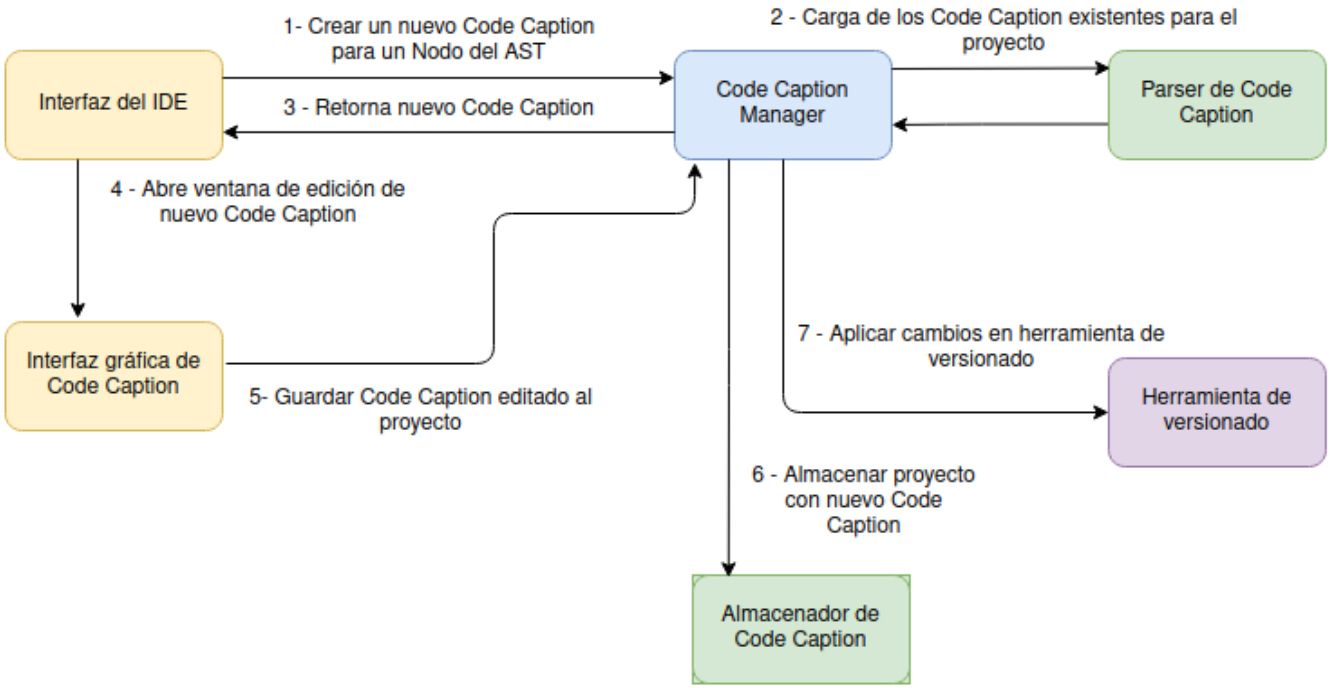


Figura 23. Diagrama de interacción en el agregado de una revisión de código CodeCaption

En el diagrama de interacción para el agregado de un nuevo CodeCaption el primer paso se da desde la interfaz del IDE llamando a agregar una nueva revisión.
Allí la interfaz cede el control al manager de CodeCaption para crear todas las estructuras necesarias para la nueva revisión, incluyendo el nodo de AST al que hace referencia la misma.
Antes de presentar al desarrollador con la vista para agregar el comentario a la revisión se llama al parser a revisar si ya existen revisiones de código presentes en disco para este proyecto. De ser así las carga y la nueva revisión que se crea se agregará como una revisión más para este proyecto junto a las existentes.

Ya teniendo toda la estructura preparada y cargadas de disco las revisiones existentes del proyecto, la interfaz del IDE llama a la interfaz de CodeCaption para mostrar la ventana de creación de la revisión. Aquí se presenta con un editor de texto para agregar el comentario y demás información necesaria para la revisión.
Cuando el desarrollador llame a guardar la nueva revisión se carga correctamente la nueva estructura de la revisión y se llama al almacenador de CodeCaption para guardar la misma en disco.

El almacenador codifica la nueva revisión junto con las ya existentes del proyecto y guarda el resultado en un archivo en disco. Luego, se llama a registrar los cambios de las revisiones en

disco en la herramienta de versionado del proyecto. Una vez registrados los cambios se completa el proceso de agregado de una revisión de código CodeCaption.

3.4.2 Diagrama de interacción para la resolución de un Code Caption

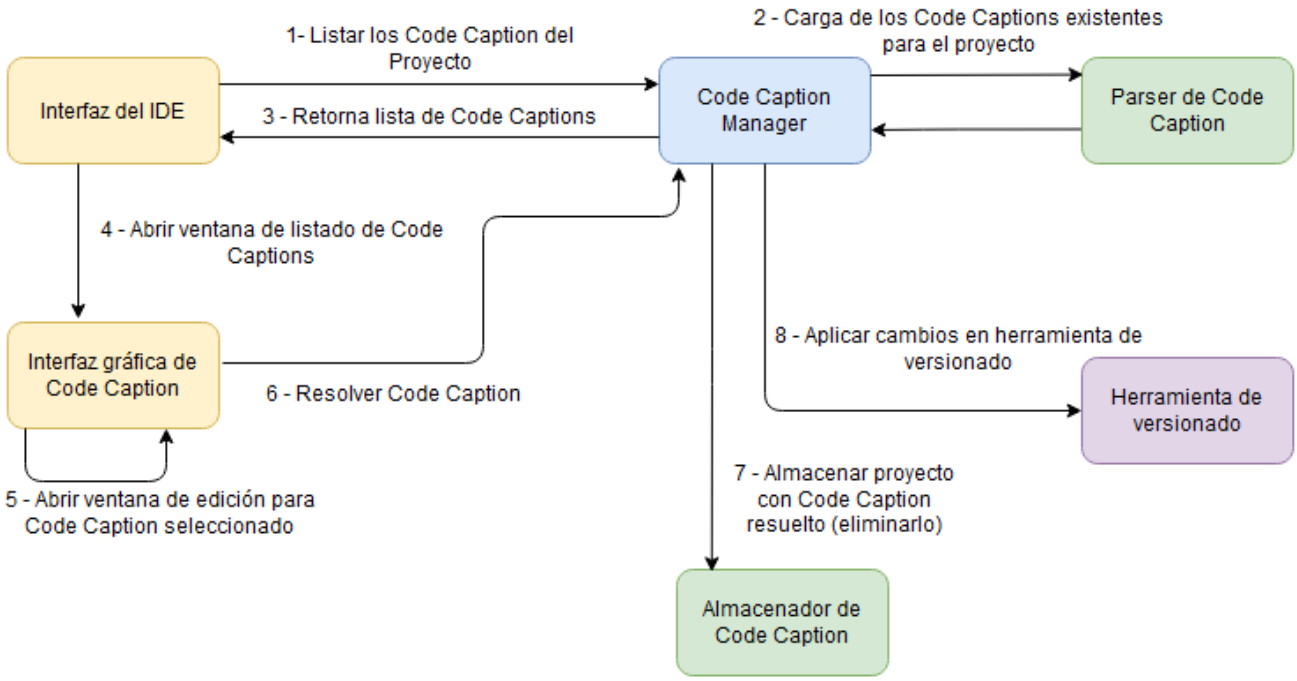


Figura 24. Diagrama de interacción en la resolución de una revisión de código CodeCaption

En el diagrama de interacción para la resolución de una revisión CodeCaption el primer paso se da desde la interfaz del IDE llamando a listar las revisiones presentes en el proyecto.

Ya obtenidas las revisiones de disco, se instancia la interfaz gráfica de CodeCaption abriendo la ventana con el listado de las mismas. En este punto el usuario selecciona la revisión que desea resolver del listado y se abre la ventana de edición de revisión, donde se puede visualizar en su totalidad la misma y está disponible el botón para resolver.

Al llamar a resolver, la interfaz comunica la acción al manager de CodeCaption que aplica la resolución, eliminando la revisión de la lista del proyecto. Eliminada la revisión, se obtiene la nueva lista de revisiones pendientes del proyecto y se llama al almacenador para guardarlas en disco.

A partir de este punto el proceso es igual a lo que ocurre en el final del diagrama anterior: el almacenador codifica y guarda las revisiones en disco para que luego la herramienta de
Luego toma el mando el manager de CodeCaption para inicializar las revisiones de código del proyecto y llamar al parser a que cargue las revisiones desde disco. En caso de que existan,

las carga y las retorna hacia el manager y hacia la interfaz del IDE.versionado registre los cambios.
Así se finaliza el proceso de resolución de una revisión de código CodeCaption.

3.5 Diseño de CodeCaption dentro de Pharo

En este apartado se muestra el diseño final de la aplicación dentro del entorno de desarrollo Pharo Smalltalk.
Para integrar la herramienta con el entorno, es necesario contar con una interfaz gráfica con la que el usuario pueda interactuar. Dicha interfaz tiene que ser una extensión de la interfaz gráfica ya existente del entorno, para que cambiar entre realizar tareas de desarrollo y tareas de revisión de código sea algo transparente para el usuario.

CodeCaption se integra con el entorno mediante el framework Spec2 [22] para la versión 9.0 de Pharo Smalltalk. Spec brinda al desarrollador la posibilidad de poder construir sus propias herramientas dentro de Pharo, desde pequeñas interfaces con poca funcionalidad a herramientas complejas y de gran alcance como un debugger[6]. Incluso el debugger de Pharo está desarrollado con Spec, así como otras herramientas incluidas por el entorno como Iceberg (para gestionar repositorios git dentro de Pharo), Change Sorter (revisar cambios en la imagen) o Critics Browser (que permite realizar un análisis de código de las clases).

Usando las herramientas que provee Pharo, CodeCaption implementa una interfaz que se divide en:

- Una interfaz propia de la herramienta con ventanas personalizadas para la manipulación de las revisiones.
- Una extensión de la interfaz de Pharo para poder acceder a la interfaz de la herramienta desde las funcionalidades propias del entorno.

En los siguientes capítulos se explican en detalle estas dos secciones de la interfaz del trabajo.

3.5.1 Interfaz de CodeCaption

Como ya fue dicho en el capítulo anterior, la interfaz principal de CodeCaption dentro de Pharo está creada bajo el framework de Spec, logrando una herramienta gráfica corriendo dentro del contexto del entorno de desarrollo. Esta interfaz consta de dos secciones principales, la ventana del agregado/editado de un CodeCaption y la ventana de listado de los CodeCaption presentes en el proyecto, que se detallan a continuación.

3.5.1.1 Agregado y/o editado de CodeCaption

La ventana de agregado de un CodeCaption es la parte de la interfaz de la herramienta donde el revisor agrega el comentario a una porción del código elegida para revisar. En la siguiente

[6] Programa de computadora que asiste en la detección y corrección de errores en otros programas.

figura se muestra una ventana ejemplo de agregado, con la explicación debajo de cada sección.

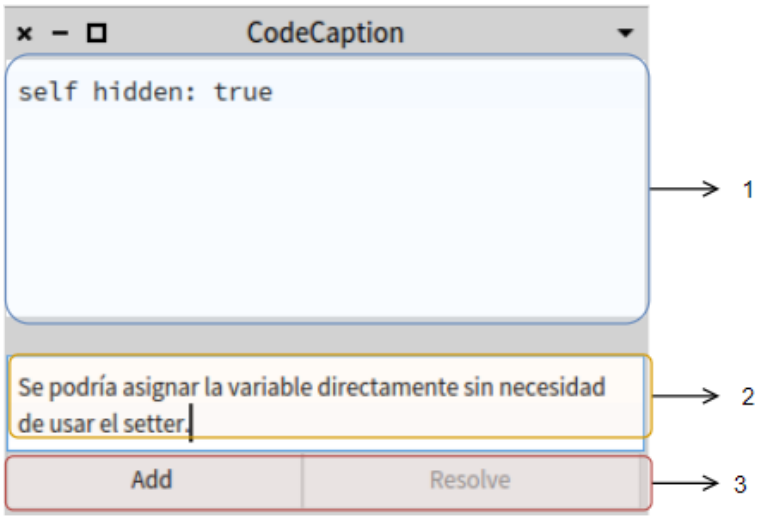


Figura 25. Ventana de Agregado de un CodeCaption.

La ventana se divide en tres secciones principales:

1. Bloque de código del método a revisar. Aquí aparece el código al que el revisor le está agregando un comentario, puede ser una sentencia corta como en este caso o todo un bloque código entero de un método.
2. Área de texto para el comentario. Aquí el revisor debe ingresar el texto del comentario que desea hacerle al código en cuestión.
3. Área de acciones. En este recuadro se encuentran los botones ‘Add’ para agregar el nuevo CodeCaption al proyecto y el ‘Resolve’ para resolver el mismo. En el caso actual del agregado de un nuevo CodeCaption el botón para resolver está deshabilitado ya que todavía no fue creado.

Al ingresar un comentario y presionar el botón ‘Add’ el CodeCaption es creado definitivamente y agregado al proyecto.

La ventana en la edición de un CodeCaption creado anteriormente es idéntica a la de agregado, con la excepción de que ahora se cargará el comentario ingresado en la creación junto con la posibilidad de resolver la revisión.

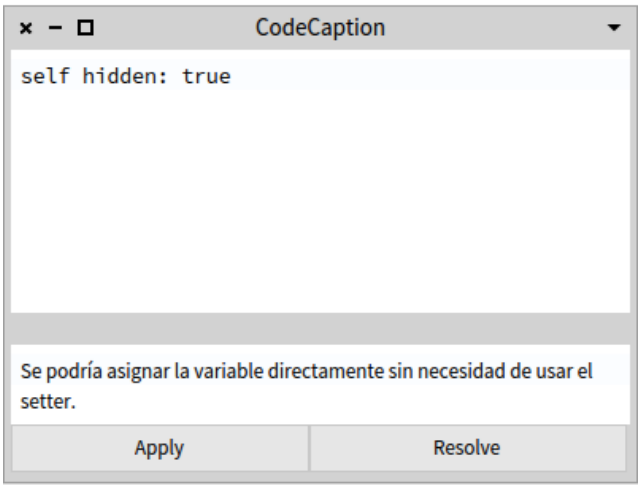


Figura 26. Ventana de edición de un CodeCaption.

Al presionar el botón 'Resolve', el CodeCaption es resuelto y satisfactoriamente eliminado de la lista de revisiones pendientes del proyecto.

3.5.1.2 Listado de CodeCaptions

La ventana de listado de CodeCaptions es una tabla donde se pueden visualizar todos los CodeCaptions presentes en el proyecto que estemos trabajando dentro de Pharo. Al abrir la ventana se muestra el listado con diferentes columnas de información y la posibilidad de abrir cualquiera de los CodeCaption.

En la siguiente figura se muestra un ejemplo de listado de CodeCaption con la información de cada uno entre las distintas columnas. Debajo de la figura se explica qué significa cada sección de la ventana.

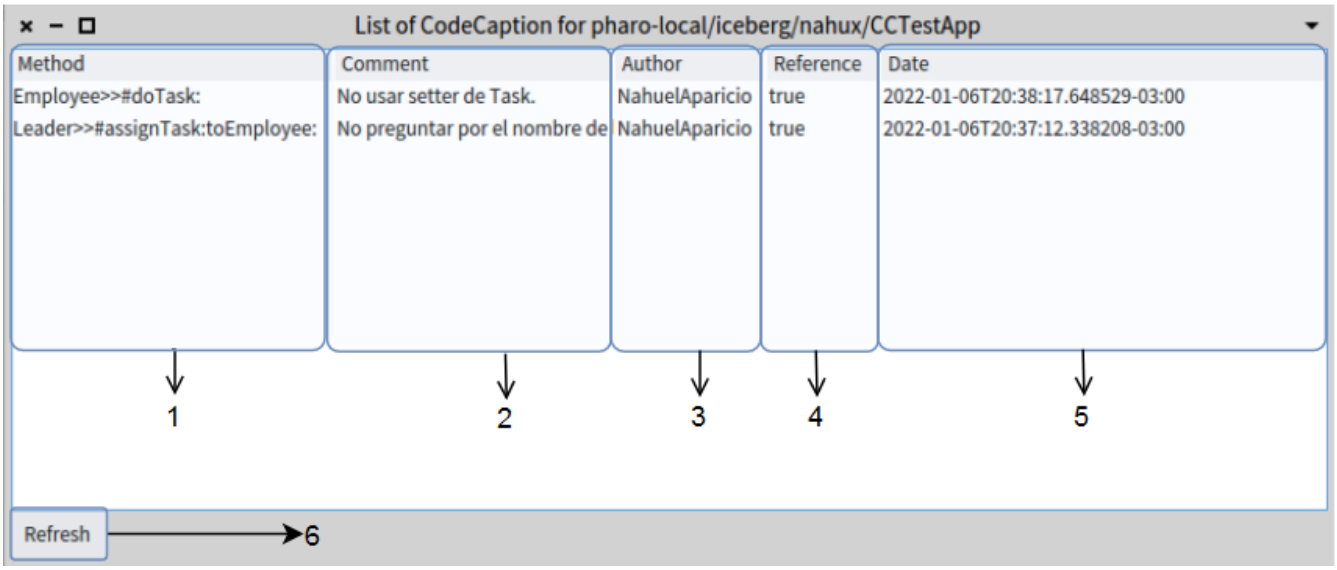


Figura 27. Ventana de listado de CodeCaption.

Explicación de las secciones de la vista del listado de CodeCaptions:

1. **Method:** Método en el que se encuentra el código revisado. Muestra la clase que contiene al método también.
2. **Comment:** Comentario realizado por el revisor en el CodeCaption al código revisado. Es el texto explicativo ingresado por el revisor al analizar el código.
3. **Author:** Autor material de la revisión CodeCaption. Refleja automáticamente el nombre configurado en el entorno por el revisor.
4. **Reference:** Indica si el código referenciado no se ha modificado o eliminado desde la creación del CodeCaption. De haber sido alterado el valor sería *false* en vez de *true*.
5. **Date:** Indica la fecha de creación de la revisión CodeCaption por parte del revisor.
6. **Botón de Refresh:** Este botón permite al usuario recargar la tabla de revisiones para ver si alguna sufrió alguna modificación luego de que se abriera la ventana.

Si el usuario selecciona uno de los CodeCaption listados puede visualizar la ventana de edición de CodeCaption detallada en la sección anterior.

También, el usuario puede abrir un menú de contexto al seleccionar un CodeCaption listado. Esto muestra un botón "*Browse*" que le permite navegar hacia el código del método referenciado en la revisión:

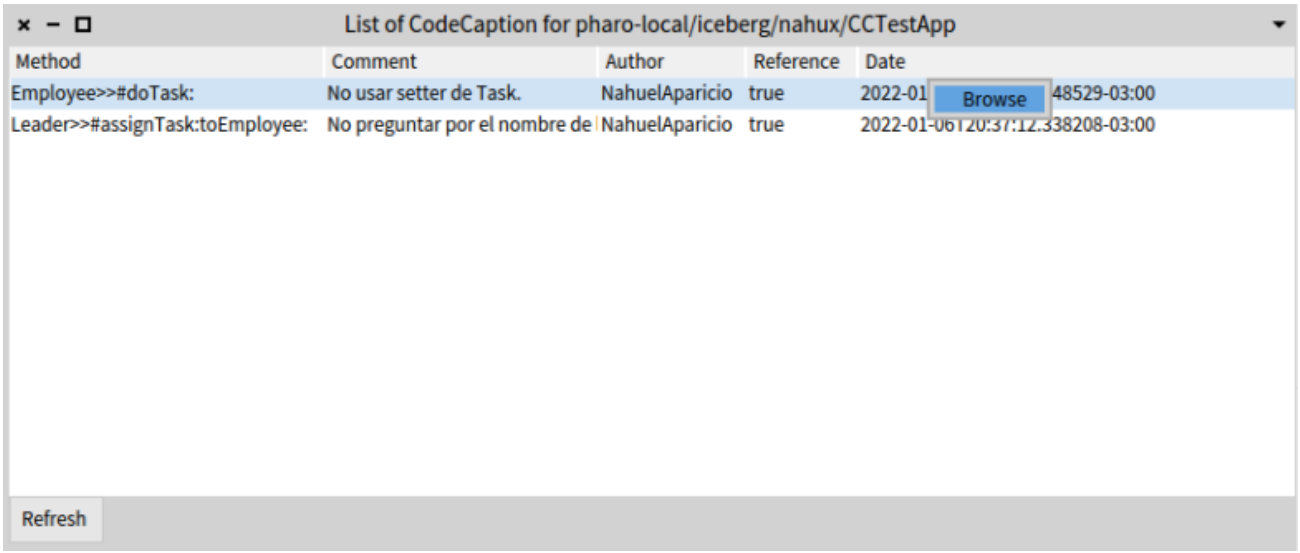


Figura 28. Ventana de listado de CodeCaption con menú de contexto.

3.5.2 Extensión de la interfaz de Pharo

Para poder integrar definitivamente la herramienta a Pharo es necesario extender las funcionalidades de la interfaz gráfica propia del entorno.
En el caso de CodeCaption hay dos puntos en los que se extiende la interfaz de Pharo: la selección de código a revisar y la apertura del listado de los CodeCaptions presentes dentro del proyecto del código que estamos viendo.

3.5.2.1 Selección de código a revisar

El revisor para crear una revisión CodeCaption tiene que indicar qué código del proyecto es el que quiere revisar. Para esto, la herramienta se integra con la interfaz de Pharo permitiendo que al seleccionar una porción de código dentro de un método se pueda crear una nueva revisión desde un menú contextual. En la siguiente figura se detalla el escenario.

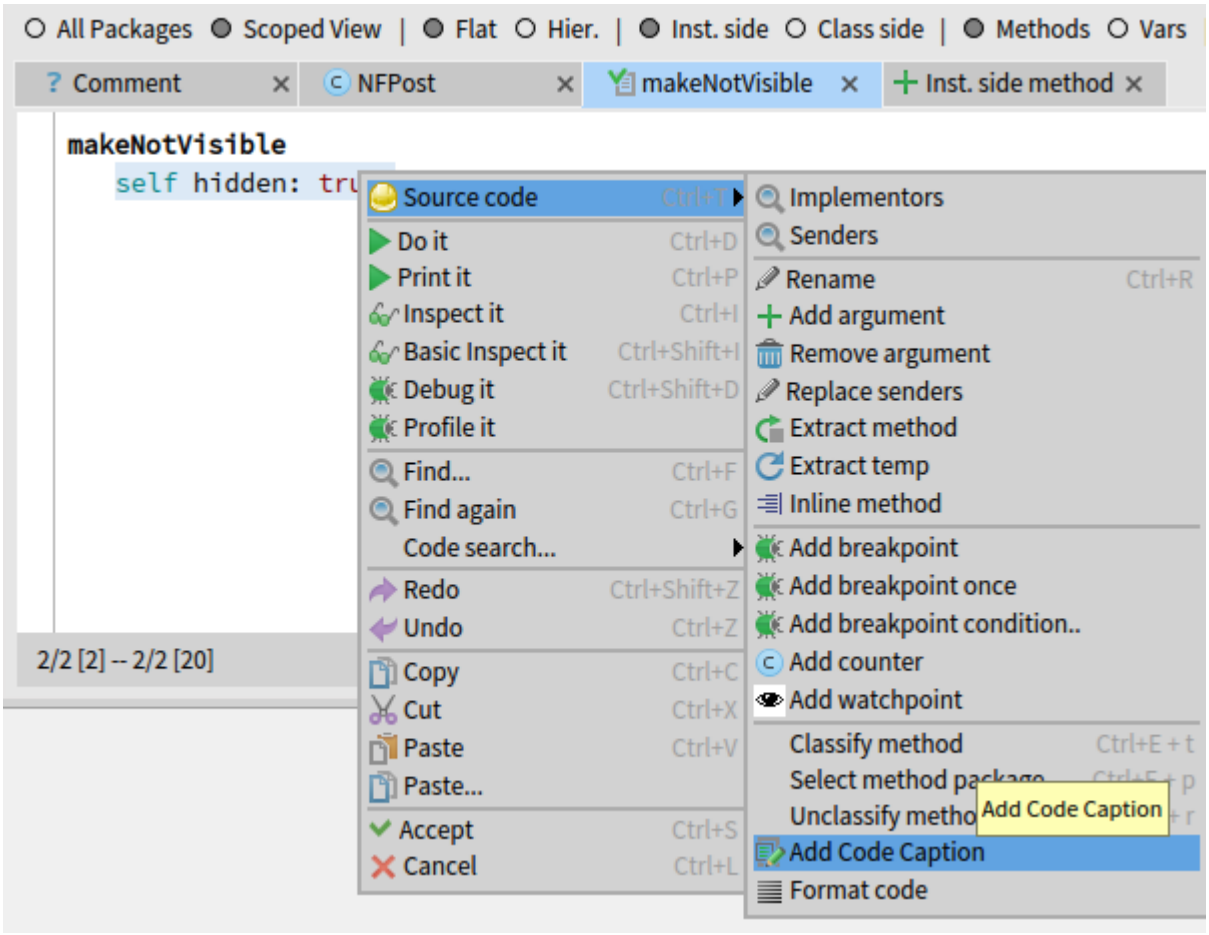


Figura 29. Botón para agregar revisión CodeCaption a partir de una selección de código.

El revisor señala el código que desea revisar y agregar un comentario, y mediante el menú contextual del entorno selecciona la opción “Add Code Caption” para crear una revisión. Aquí lo que sucede es que se ha extendido la funcionalidad del menú contextual del entorno para agregar esta nueva opción y enviarle a la interfaz de CodeCaption el código seleccionado.

Al seleccionar la opción de ‘Add Code Caption’ el usuario revisor verá la ventana de agregado que se detalla en el apartado anterior de la interfaz (figura 25). Esta funcionalidad integrada dentro del mismo espacio de edición de código del entorno hace que la tarea de revisión sea simple y eficiente para el revisor que se encuentra analizando el código del proyecto.

3.5.2.2 Apertura de listado de CodeCaptions

Para poder ver los CodeCaptions que fueron creados y están todavía presentes en un proyecto se requiere extender otra sección de la interfaz gráfica de Pharo.
Al igual que en el caso de la selección de código para el agregado de un nuevo CodeCaption, en este caso el usuario tiene que estar en la vista de edición de código del entorno. El usuario al estar en la vista de edición de código del entorno está añadiendo, modificando o simplemente observando código dentro de un proyecto de Pharo, por lo que la herramienta puede identificar a qué proyecto solicitarle las revisiones.

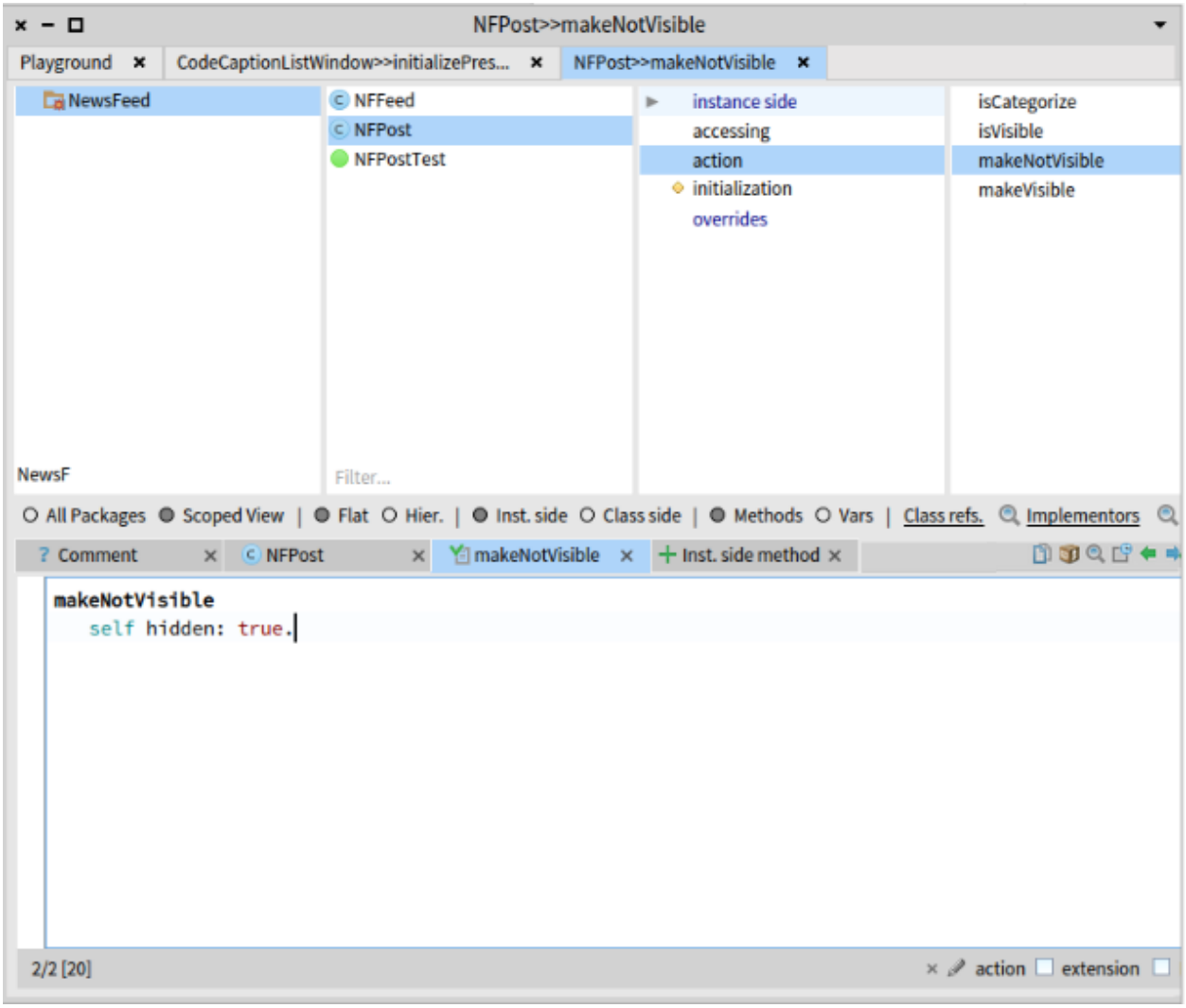


Figura 30. Vista de edición del código de un método en Pharo.

La figura anterior representa la vista de edición de código de Pharo dentro de la clase “NFPost” del paquete “NewsFeed” que forma parte de un proyecto. Allí se está visualizando el código del método “makeNotVisible” que simplemente llama a un método setter de la misma clase.

Si el usuario hace click derecho en el paquete que que contiene a la clase y por consiguiente al método que estamos visualizando, verá un menú de contexto con acciones sobre el paquete. En estas acciones se incluye el botón para listar todas las revisiones CodeCaption presentes para el paquete de Pharo.

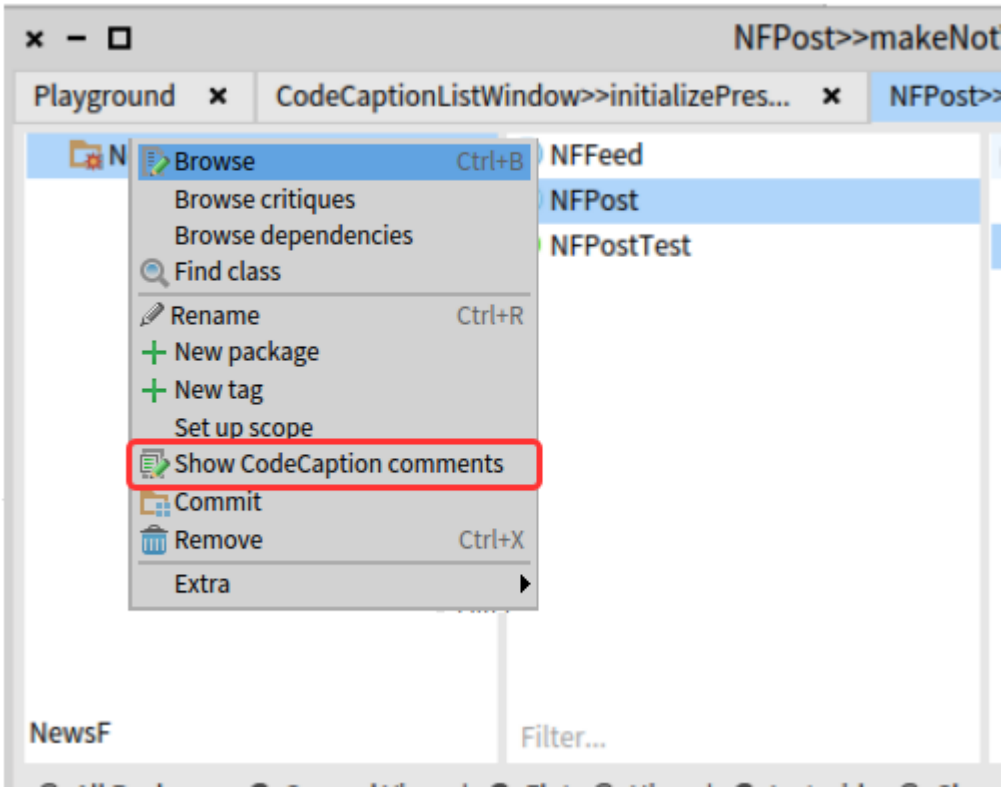


Figura 31. Botón para listar todas las revisiones de código del proyecto en Pharo.

Aquí se encuentra el botón de listado de CodeCaption, señalado con el recuadro rojo. Al presionar este botón, el usuario podrá ver el listado de CodeCaptions del proyecto mediante la ventana de listado detallada en el apartado de la interfaz de CodeCaption (figura 28).

Al ejecutarse el botón, se cargan todos los CodeCaption sin resolver presentes del proyecto para que el usuario pueda verlos y editarlos si así lo desea.

3.6 Desventajas del Diseño

Durante el desarrollo del trabajo fueron apareciendo algunas posibles características que no terminaron siendo incluidas en el diseño del mismo. Se trata de algunas mejoras o distintos enfoques que quedaron fuera del scope al no ser parte primordial del foco de la tesina, pero podrían agregar valor a la herramienta.

Uno de estos faltantes es que no existe un “flujo de trabajo” para una revisión de CodeCaption. Es decir, que la vida de la revisión en un proyecto no está delimitada por reglas ni caminos a seguir. Cualquier involucrado en el proyecto con permisos en el repositorio y acceso al entorno de desarrollo puede crear, modificar y resolver la revisión que le parezca. No existe ningún tipo de rol que indique quien pueda crear y/o modificar, quien pueda resolver, etc.

En este mismo sentido, también está limitada la posibilidad de agregar algún tipo de información extra a las revisiones, sólo se permite un comentario por creador de revisión. Cualquier usuario del proyecto está imposibilitado de dejar comentarios adicionales en la revisión, Por ejemplo, alguien podría necesitar realizar alguna consulta o explicar el pensamiento detrás del desarrollo del código revisado, entre otros ejemplos.

Tampoco es posible ver dinámicamente los CodeCaptions presentes en el código que el desarrollador está viendo, (cómo un tooltip al pasar por encima de un código con revisión asociada). La única posibilidad de observar qué revisiones están presentes en el proyecto que el usuario tiene abierto en el IDE es mediante el botón que muestra el listado de dichas revisiones. Esto se debe a una dificultad técnica del entorno que resulta en tener que realizar un desarrollo complejo y demandante para poder completar esta funcionalidad, que no aporta demasiado al foco del trabajo.

Capítulo 4. Implementación de la herramienta

4.1 Introducción a la Implementación

El desarrollo de CodeCaption está completamente realizado en el entorno de programación Pharo Smalltalk. El lenguaje utilizado por el entorno es por consiguiente Smalltalk, que cubre un 100% del código fuente de la herramienta.

Al ser una herramienta para el mismo entorno en el que se desarrolla, hace mucho uso de las funcionalidades ya existentes e incluso se implementan extensiones de código específicas para la interfaz del entorno. La comunicación de la herramienta con servicios externos al entorno consta básicamente de la lectura/escritura de las revisiones de código en archivos en disco y en la integración con la herramienta de versionado Git. Esta última es clave para poder mantener un historial de los cambios de las revisiones y para poder sincronizarlas entre todos los colaboradores del proyecto y que no queden sólo en la máquina local del usuario.

La implementación de la herramienta se puede dividir en 4 partes generales:

- Interfaz Gráfica de la Herramienta:
    - Incluye las ventanas propias de CodeCaption para ver y manipular revisiones de código.
    - También incluye las extensiones a la interfaz propia de Pharo Smalltalk para poder acceder a las funcionalidades de CodeCaption.
- Parseo y Almacenamiento de las revisiones en disco:
    - Esta sección es clave para el guardado y la recuperación de las revisiones de código. Se define el uso del formato de datos tipo STON para el guardado en disco.
    - Contiene toda la lógica del parseo del archivo de revisiones STON así como su almacenamiento en disco a partir de los objetos Smalltalk.
- Manejo de las revisiones de código y cómo se integran con los ASTs de Pharo:
    - Aquí se encuentra la lógica para la creación, manipulación y estructura de datos de las revisiones de código.
    - Las revisiones de código de la herramienta dependen de los ASTs que representan el código Smalltalk de Pharo. Por esta razón, se incluye la lógica para leer y recorrer los nodos que representan el código revisado en los ASTs.
- Integración con Repositorio Git:
    - Aquí se encuentra la porción de código encargada de integrar las revisiones CodeCaption de un proyecto con su repositorio Git asociado.
    - Esto permite poder versionar todas las revisiones mediante la creación y modificación del archivo en disco.

En los siguientes capítulos se ahondará sobre la estructura de la implementación con la explicación del diagrama de clases de la herramienta, y luego se explicará en detalle la implementación de las secciones mencionadas en el párrafo anterior.

4.2 Diagrama de Clases

El diagrama de clases del proyecto se encuentra dividido en unas secciones que agrupan distintas clases que en conjunto implementan una funcionalidad o funcionalidades de una misma temática. Por ejemplo, para las funcionalidades de las ventanas de la interfaz gráfica del proyecto, se agrupan las clases involucradas en la sección ”UI”. Estas secciones a su vez se plasman en la implementación como etiquetas o “tags” dentro de Pharo.

Las distintas secciones son:

- UI
    - Renderizado y funcionalidad de las ventanas de la interfaz gráfica. Comprende a la ventana del listado de los CodeCaption como la de agregar/editar.
- UI Context
    - Funcionalidad de los botones del menú contextual de Pharo para poder abrir las ventanas de CodeCaption. Incluye el botón agregar un nuevo CodeCaption que aparece en el editor de código y el botón listar, que aparece arriba del editor.
- Caption
    - Lógica de CodeCaption, con los proyectos, la integración con Git, el uso de los nodos AST de Pharo y la carga y almacenamiento de los CodeCaption en disco.
- Extension
    - Extensiones a clases preexistentes en Pharo. Básicamente agrega la funcionalidad para guardar en disco a las instancias de clases ajenas a CodeCaption pero que son necesarias para el funcionamiento.

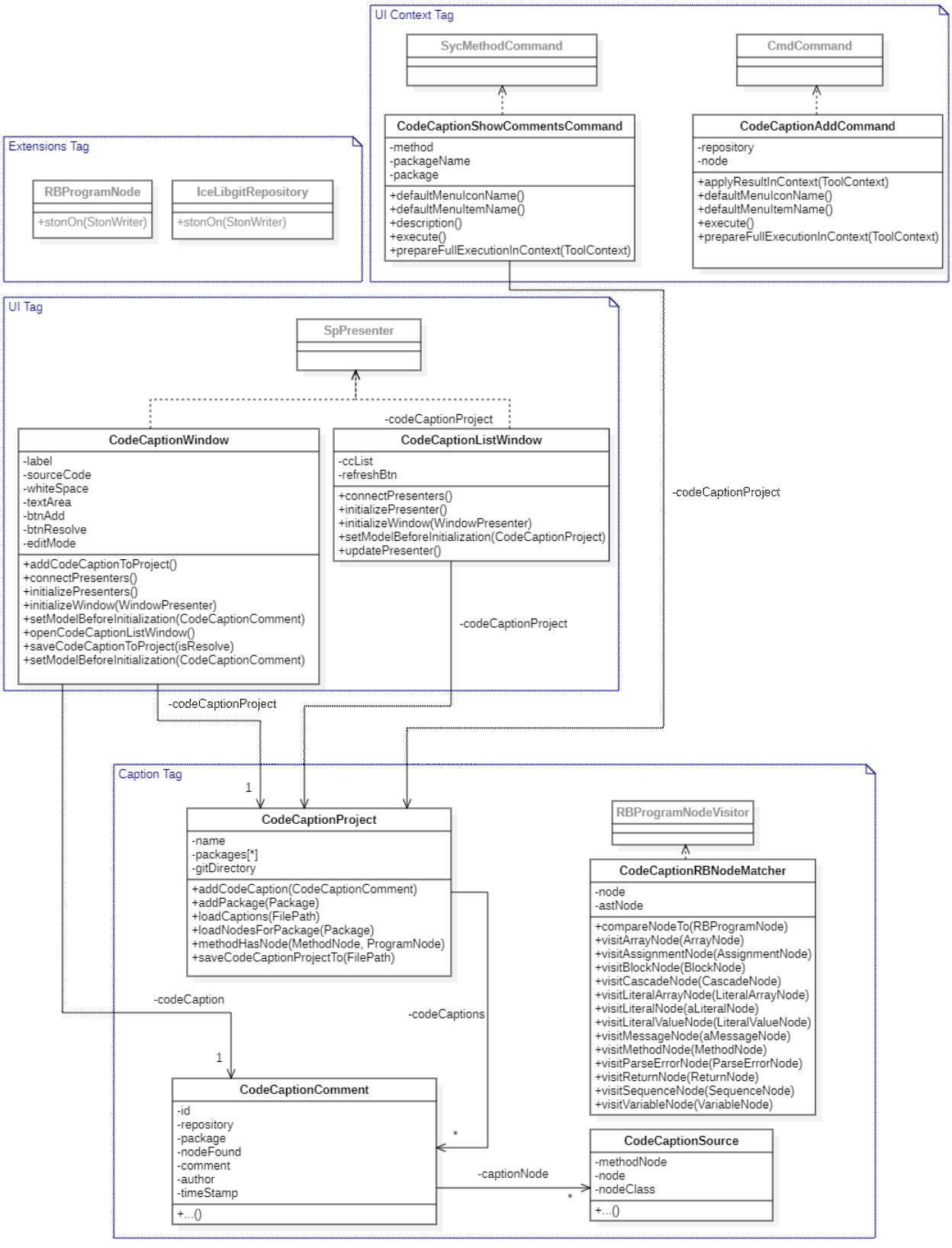


Figura 32. Diagrama de Clases de CodeCaption dividido en secciones.

4.3 Clases por Secciones

A continuación se detallan todas las clases del código del proyecto organizadas por secciones (representadas como “tags” dentro de Pharo). Por cada clase se explica brevemente su importancia dentro del funcionamiento de la herramienta.

4.4 Caption

CodeCaptionComment

Una revisión de código Code Caption.
Es la clase que se instancia cuando los usuarios agregan una nueva revisión en el entorno, tiene toda la información correspondiente de la revisión: el nodo referenciado (CodeCaptionNode), la revisión, el estado, el autor e información relevante para el correcto guardado como el repositorio y el paquete al que pertenece.
No contiene mucha lógica de negocio pero es clave para la representación de un CodeCaption, es en definitiva lo que el usuario crea, guarda y se muestra en la interfaz de usuario del entorno.

CodeCaptionSource

Es la clase que vincula un CodeCaption con el nodo del AST al cual se aplica. Dicho vínculo requiere de tres elementos: clase, método y nodo AST.
Hace uso de la estructura Abstract Syntax Tree (AST) implementada por Pharo para representar dinámicamente el código del entorno. Tiene el nodo AST propio del código referenciado (por ejemplo el nodo de asignación de una variable), el nodo AST del método que contiene al nodo anterior (incluso pueden ser los mismos si se referencia todo el método) y la clase a la que pertenece el método.

CodeCaptionProject

Un proyecto de CodeCaption para un proyecto Pharo en un repositorio git controlado por Iceberg. Su función principal es administrar todas las revisiones de código del proyecto.
Es el punto de entrada a CodeCaption, la primera clase a instanciar cuando se quiere crear o cargar un proyecto del mismo, encargándose también de guardarse y cargarse usando un archivo en disco.

CodeCaptionRBNodeMatcher

Es una clase que implementa el rol visitor en el patrón de diseño homónimo, usado para encontrar un nodo de referencia de un CodeCaption en el AST del código fuente del proyecto. Gracias a dicho visitor podemos avisarle a la revisión de CodeCaption si el nodo al que hace referencia sigue presente sin modificaciones en el código o fue alterado en alguna forma.

4.5 UI Context

CodeCaptionAddCommand

Extensión de la interfaz de Pharo para el botón en el contexto del editor de código de un método en Pharo para agregar una nueva revisión de CodeCaption. Al seleccionar el botón se abre la ventana de agregado de una nueva revisión de CodeCaption.

CodeCaptionShowCommentsCommand

Extensión de la interfaz de Pharo para el botón que lista todos los CodeCaption en un paquete. Se encuentra por encima de la edición de código de un método en Pharo, entre el resto de los botones disponibles para aplicar acciones relacionadas con el método. Al hacer click en el botón se abre la ventana con el listado de todas las revisiones de CodeCaption guardadas para el proyecto, en caso de no haber ninguna se muestra un mensaje indicando dicha situación.

4.6 UI

CodeCaptionListWindow

Ventana gráfica con la lista de todas las revisiones de CodeCaption de un proyecto.
Lista cada CodeCaptionComment en forma de tabla con la información de cada uno, permitiendo ingresar a editar cada uno de ellos seleccionando la fila correspondiente.

CodeCaptionWindow

Ventana de agregado y/o edición de una revisión de CodeCaption.
Tiene una caja de texto no editable con el código fuente del nodo al que el CodeCaption hace referencia, y debajo tiene una caja de texto para agregar o editar el texto propio de la revisión. Las acciones permitidas son las de guardar el CodeCaption con los datos ingresados o si se quiere, resolverlo.

4.7 Extensions

Extensiones a clases ya presentes en Pharo, agregan código a clases ya existentes para el correcto funcionamiento de CodeCaption. Estas extensiones en Pharo sirven para poder agregarle funcionalidad a una clase ya existente en el entorno.

RBProgramNode

Implementación de método **stonOn:** *stonWriter* (Definida en la clase Object) para poder exportar el AST del nodo referenciado en el CodeCaption a un archivo de notación de objetos de tipo STON.

IceLibgitRepository

Implementación de método **stonOn:** *stonWriter* (Definida en la clase Object) para poder exportar el repositorio git (objeto que lo representa) como propiedad de CodeCaptionComment, a un archivo de notación de objetos STON.

4.8 Interfaz gráfica

La interfaz gráfica de CodeCaption está implementada en su totalidad con las herramientas, frameworks y librerías provistas por Pharo Smalltalk.

Tal como se especifica en la sección 3.5 Diseño de CodeCaption dentro de Pharo, se puede dividir la implementación de la interfaz gráfica de CodeCaption en dos secciones:

- Por un lado, la herramienta hace uso de la implementación interna de Pharo de los elementos de su interfaz. Específicamente se extiende la funcionalidad agregando más botones a los presentes en la vista de edición de código para poder integrar la herramienta al entorno de desarrollo mediante los mismos
- Por otro lado, la herramienta también hace uso del framework de interfaces gráficas Spec 2.0 [22] que permite crear una gran variedad de ventanas gráficas para el usuario dentro de Pharo. Este framework posibilita la creación de las ventanas gráficas de agregado/edición de revisión y de listado de todas las revisiones de código de un proyecto.

4.8.1 Botones Comandos de Pharo Smalltalk

En el caso de la extensión de los botones de la edición de código de Pharo se implementa la herencia de la clase *CmdCommand* que identifica un “comando” a ejecutar desde la interfaz del entorno. Los botones de la vista de edición de código de Pharo que nos permiten realizar distintas acciones dentro del entorno son precisamente “comandos”:

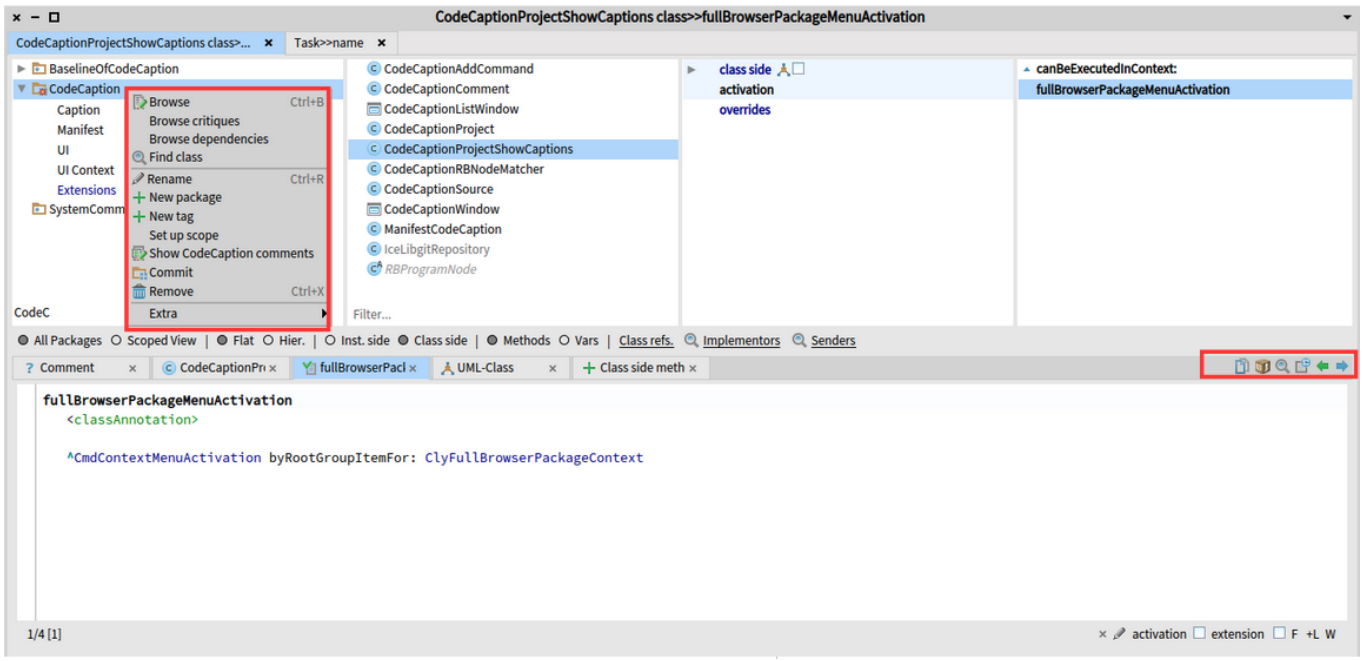


Figura 33. Vista de edición de código de Pharo Smalltalk con los botones "comandos" encuadrados en rojo.

Para implementar uno de estos botones, se debe heredar de la case *CmdCommand* especificando al instanciar la clase dónde y en qué contexto debe aparecer el botón. En el caso de CodeCaption se implementan dos comandos distintos: *CodeCaptionAddCommand* (para agregar una revisión de código) y *CodeCaptionShowCommentsCommand* (para listar las revisiones de código de un proyecto).

4.8.1.1 Agregar y/o Editar un CodeCaption (CodeCaptionAddCommand)

Esta clase representa al botón de contexto del entorno para crear y/o editar una revisión de código CodeCaption dentro de la ventana de edición de código de un método en Pharo:

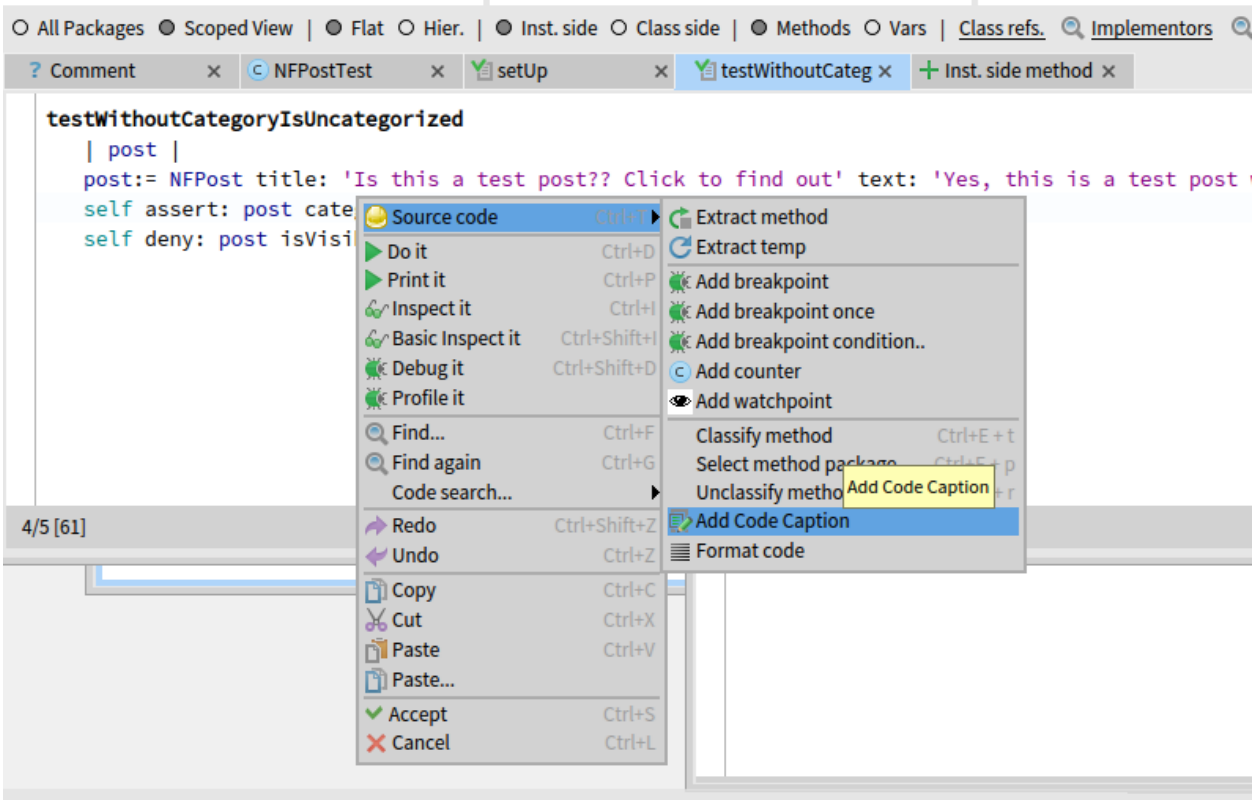


Figura 34. Vista de edición de código de un método con el botón de agregado de revisión CodeCaption en el menú de contexto.

Para que el comando se muestre en esa sección del menú de contexto es necesario especificar un método de clase de "activación" del comando en el contexto adecuado. En este caso se implementa el método *sourceCodeMenuActivation* que especifica que el botón del comando será visible dentro del menú de contexto "Source Code" en la vista de edición de código de un método de Pharo.

```
sourceCodeMenuActivation
    <classAnnotation>

    ^SycSourceCodeMenuActivation byRootGroupItemOrder: 100000 for: ClyMethodSourceCodeContext
```

Figura 35. Código del método de clase sourceCodeMenuActivation para la clase CodeCaptionAddCommand.

También para CodeCaption es necesario que dicho botón solo aparezca en el caso que se esté editando el código de un método de una clase que esté incluida dentro de un paquete perteneciente a algún repositorio de Iceberg. Para ésto se implementa el método de clase *canBeExecutedInContext: aToolContext* donde se retorna si el método en cuestión es parte de un repositorio de Iceberg o no:

```
canBeExecutedInContext: aToolContext
    ^ (Smalltalk globals includesKey: #IceLog)
        and: [ (IceRepository registeredRepositoryIncludingPackage: aToolContext lastSelectedMethod package) notNil
```

Figura 36. Código del método de clase canBeExecutedInContext para la clase CodeCaptionAddCommand.

En caso de que el método sea de un paquete ajeno a Iceberg el botón de agregado de CodeCaption no estará visible para el usuario, ya que la herramienta no sabría dónde guardar la revisión en disco.

Con respecto a los métodos de instancia de la clase se implementan los que definen el ícono y texto del comando entre otros. aunque los más importantes son:

- *prepareFullExecutionInContext: aToolContext*
    - Indica la lógica antes de la ejecución del comando y tiene como parámetro el contexto en el que se ejecuta.
- *execute*
    - Realiza la ejecución efectiva del comando (abrir la ventana de agregado de revisión).

El método *prepareFullExecutionInContext: aToolContext* se encarga de tomar el método siendo revisado, el repositorio Iceberg correspondiente y el nodo AST seleccionado por el usuario antes de llamar al comando.

```
prepareFullExecutionInContext: aToolContext
    | method packageName |
    super prepareFullExecutionInContext: aToolContext.

    "Set Iceberg repository"
    method := aToolContext lastSelectedMethod.
    packageName := method package name.
    repository := CodeCaptionProject getRepositoryForPackageName: packageName.
    repository repositoryDirectory
        ifNil: [ self inform: 'No repository was found for this package!' ]
        ifNotNil: [ "Set AST node"
            aToolContext selectedSourceNode.
            node := aToolContext selectedSourceNode ]
```

Figura 37. Código del método prepareFullExecutionInContext: aToolContext de la clase CodeCaptionAddCommand.

El método *execute* por otro lado contiene la lógica que se ejecutará al presionar el botón de agregado de revisión y tiene como función instanciar la ventana gráfica para el agregado de una revisión de código enviando como parámetro una instancia de *CodeCaptionComment* con los datos obtenidos en el método *prepareFullExecutionInContext: aToolContext*:

```
execute
    repository repositoryDirectory
        ifNotNil: [ (CodeCaptionWindow
                on:
                    ((CodeCaptionComment
                        newWithNode: node
                        package: node methodNode methodClass package
                        methodNode: node methodNode
                        repository: repository) nodeFound:true)) openWithSpec ]
```

Figura 38. Código del método execute de la clase CodeCaptionAddCommand.

Al terminar su ejecución se visualizará la ventana de agregado de una nueva revisión de código.

4.8.1.2 Listar CodeCaptions (CodeCaptionShowCommentsCommand)

Esta clase representa al botón de contexto del entorno que lista todas las revisiones de código dentro de un proyecto.

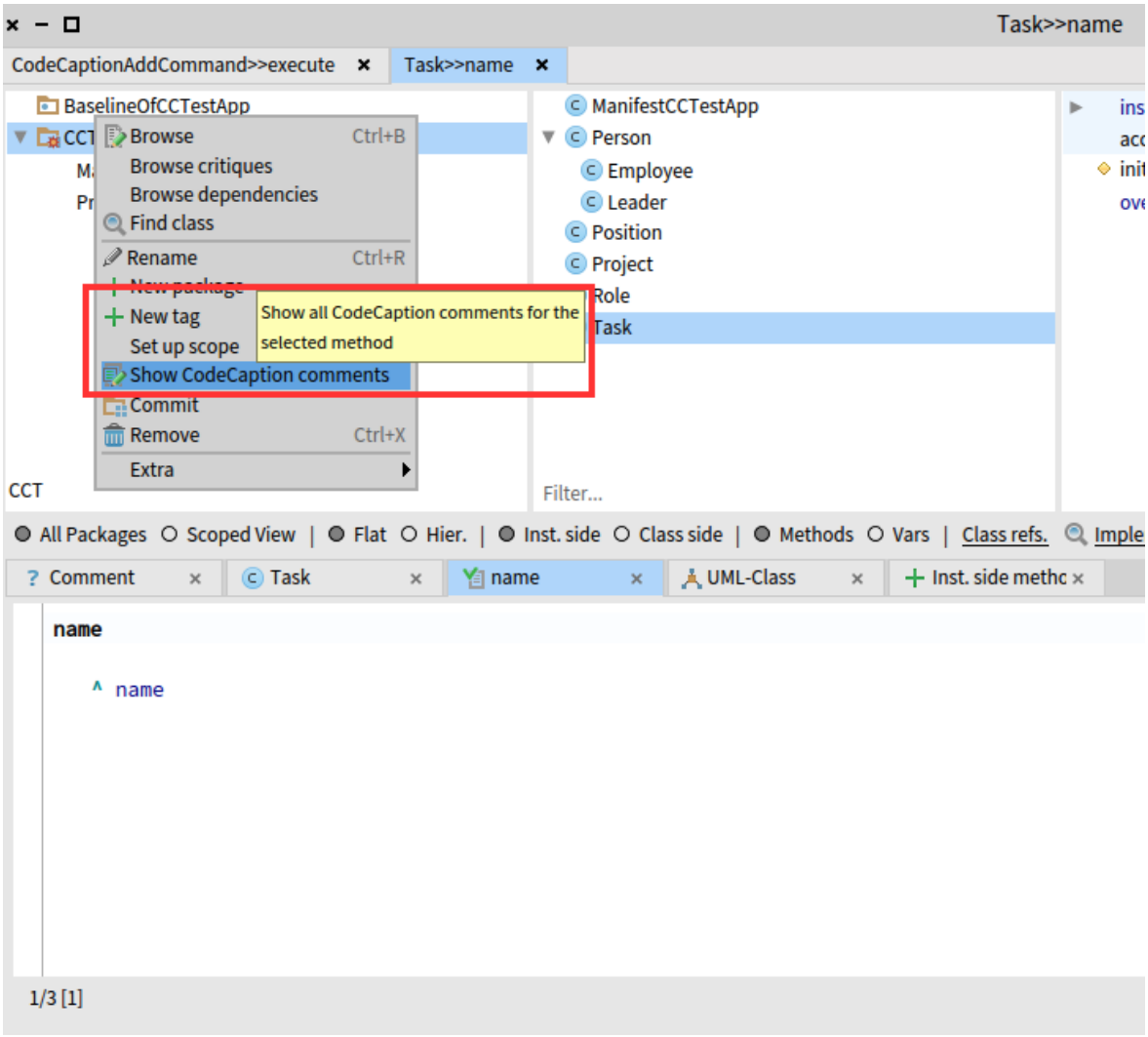


Figura 39. Vista de edición de código de Pharo Smalltalk con el botón para listar las revisiones de código encuadrado en rojo.

Para que el comando sea visible en el menú de contexto al seleccionar un paquete Pharo, es necesario especificar un método de clase de "activación" del comando en el contexto adecuado. Por esto se implementa el método *fullBrowserPackageMenuActivation* que especifica que el botón del comando será visible junto con el resto de los botones presentes por en el menú de contexto de un paquete Pharo.

```
fullBrowserPackageMenuActivation
    <classAnnotation>

    ^CmdContextMenuActivation byRootGroupItemFor: ClyFullBrowserPackageContext
```

Figura 40. Código del método de clase fullBrowserPackageMenuActivation para la clase CodeCaptionShowCommentsCommand.

Al ejecutar el botón se deben buscar las revisiones de código presentes en el proyecto al cual pertenece el paquete seleccionado por el usuario. En caso de que el usuario haya seleccionado un paquete ajeno a Iceberg el botón de listado de revisiones CodeCaption muestra un mensaje de advertencia indicando que no existen revisiones de código para para el paquete actual.

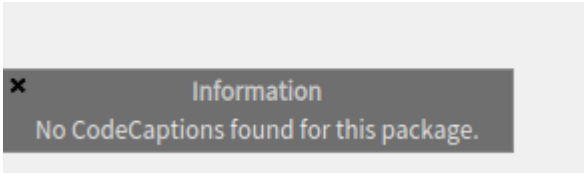


Figura 41. Mensaje informativo de la herramienta indicando que no hay revisiones de código para el paquete seleccionado.

Con respecto a los métodos de instancia de la clase, al igual que el comando de agregado de CodeCaption y resto de los comandos, se implementan los que definen el ícono y texto, y los métodos *prepareFullExecutionInContext: aToolContext* y *execute*.

El método *prepareFullExecutionInContext: aToolContext* es el que tiene el procesamiento principal de la lógica del comando. Tiene como función identificar el repositorio Iceberg al que pertenece la clase y paquete del método que se está visualizando, para luego llamar a instanciar a *CodeCaptionProject* cargando las revisiones de código para el proyecto en caso de existir.

```
prepareFullExecutionInContext: aToolContext
    super prepareFullExecutionInContext: aToolContext.
    package := aToolContext lastSelectedPackage.

    "Get repository from context"
    repository := CodeCaptionProject getRepositoryForPackageName: package name.
    repository isNotNil
        ifTrue: [ repository repositoryDirectory isNotNil
                ifTrue: [ codeCaptionProject := CodeCaptionProject
                        loadProject: (CodeCaptionProject codeCaptionsDirectoryForRepository: repository)
                        packages: repository workingCopy packages ] ]
```

Figura 42. Código del método prepareFullExecutionInContext: aToolContext de la clase CodeCaptionShowCommentsCommand.

Una vez instanciado el proyecto, el mismo se guarda como una variable de instancia del comando, que luego será usada en el método *execute*. El método *execute* no hace más que abrir la ventana gráfica tipo tabla con todas las revisiones presentes para el proyecto en caso de existir (*CodeCaptionListWindow*), o mostrar el mensaje de que no existen revisiones para el paquete.

```
execute
    codeCaptionProject isNotNil
        ifFalse: [ self inform: 'No CodeCaptions found for this package.' ]
        ifTrue: [ (CodeCaptionListWindow on: codeCaptionProject) openWithSpec ]
```

Figura 43. Código del método execute de la clase CodeCaptionShowCommentsCommand.

4.8.2 Ventanas Gráficas

La interfaz gráfica propia de CodeCaption está implementada mediante el framework de interfaces gráficas Spec 2.0. El framework permite la creación de una gran variedad de ventanas gráficas dentro de Pharo y en la herramienta se implementa una ventana interactiva de tipo formulario (para el agregado/edición de una revisión) y una ventana de tipo tabla (para la lista de revisiones de código de un proyecto).

4.8.2.1 Agregado/Editado de CodeCaption (CodeCaptionWindow)

La vista de agregado y/o edición de una revisión de código CodeCaption consta de la clase *CodeCaptionWindow* que representa una ventana gráfica con los siguientes elementos:

- Área de texto con el código del método seleccionado para la revisión.
- Input de texto para agregar el comentario de la revisión.
- Sección de botones para guardar la revisión o resolver en caso que se trate de una revisión existente.

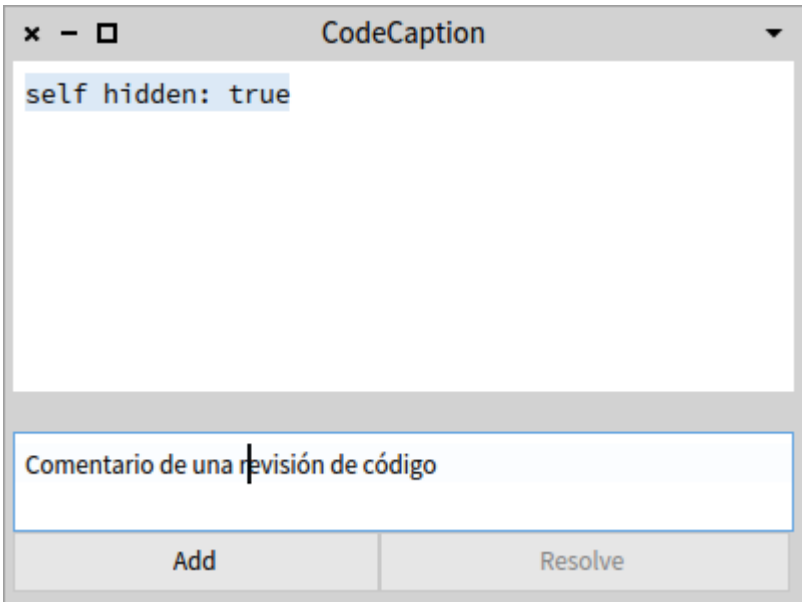


Figura 44. Ventana de agregado/edición de revisión CodeCaption

La clase *CodeCaptionWindow* hereda de la clase presetenter de Spec 2.0 *SpPresenter* que se usa para armar ventanas gráficas personalizables dentro de Pharo. Para instanciar una

ventana y especificar el diseño de la misma es necesario implementar el método de clase *defaultSpec.*

```
defaultSpec
    ^ SpBoxLayout newVertical
        add: #sourceCode;
        add: #whiteSpace height: 20;
        add: #textArea height: 50;
        add:
            (SpBoxLayout newHorizontal
                add: #btnAdd;
                add: #btnResolve;
                yourself) height: 30;
        yourself.
```

Figura 45. Código del método de clase defaultSpec de la clase CodeCaptionWindow para instanciar una ventana.

Como se puede ver en la figura, en este método se definen las distintas partes de la ventana con la posición y tamaño de cada una de ellas y se las matchea con una variable de instancia mediante los símbolos #. Por ejemplo la primera sección será el área de texto que contiene el código a revisar, y se matchea como #sourceCode para indicar que la variable de instancia sourceCode definirá su forma y estilo.

Para instanciar la ventana solo hace falta llamar al método de clase *CodeCaptionWindow >> on: aCodeCaptionProject* enviándo el proyecto como parámetro. Luego para abrir efectivamente la ventana se debe llamar el método *openWithSpec.*

```
(CodeCaptionWindow on: selection selectedItem) openWithSpec
```

Figura 46. Ejemplo de llamado a la clase CodeCaptionWindow para abrir una ventana de agregado/edición de revisiones CodeCaption.

Una vez definida la instancia de la ventana es necesario implementar los siguientes métodos de instancia para poder renderizar correctamente la misma en Pharo:

- setModelBeforeInitialization: aCodeCaptionComment
- *initializeWindow: aWindowPresenter*
- *initializePresenters*
- *connectPresenters*

setModelBeforeInitialization: aCodeCaptionComment

Este método recibe como parámetro una instancia *CodeCaptionComment* que representa a la revisión de código y se encarga de la lógica necesaria para que la revisión sea accesible dentro de la ventana y se pueda mostrar y guardar correctamente.

```
setModelBeforeInitialization: aCodeCaptionComment
   | package repository |
   super setModelBeforeInitialization: aCodeCaptionComment.
   codeCaption := aCodeCaptionComment.
   editMode := codeCaption comment ifNil: [ false ] ifNotNil: [ true ].
   package := codeCaption package.

   "TODO: add error/warning when repository repositoryDirectory is nil. It needs a repository"
   repository := IceRepository registry
      detect: [ :each | each includesPackageNamed: package name ].

   codeCaptionProject := CodeCaptionProject
      loadProject: (CodeCaptionProject codeCaptionsDirectoryForRepository: repository)
      packages: (repository workingCopy packages).
   codeCaptionProject
      ifNil: [ codeCaptionProject := CodeCaptionProject new.
         codeCaptionProject
            name: repository name;
            addPackage: package;
            gitDirectory: 'pharo-local/' , repository repositoryDirectory path pathString ]
```

Figura 47. Código del método setModelBeforeInitialization de la clase CodeCaptionWindow.

Lo primero que hace es inicializar el modelo de la ventana gráfica que usará para renderizarse y luego llena las variables de instancia necesarias para la misma. Asigna la nueva o existente revisión de código, establece si es edición o agregado de una nueva, asigna el paquete, repositorio y llama a cargar el *CodeCaptionProject* o crea uno nuevo para la revisión en caso de no existir para el paquete indicado.

initializeWindow: aWindowPresenter

En este método se especifica la lógica para instanciar la ventana gráfica de Spec, en este caso solo se especifica el título.

```
initializeWindow: aWindowPresenter
   aWindowPresenter title: 'CodeCaption'.
```

Figura 48. Código de método initializeWindow: de la clase CodeCaptionWindo

initializePresenters

Para este método se necesita especificar todo lo relacionado a los elementos gráficos que componen la ventana. Se instancian las distintas partes con su forma, orden, estilo y los datos asociados en base al modelo de datos cargado en *setModelBeforeInitialization*.

```
initializePresenters
    "Initialize widgets"
    label := self newLabel.
    sourceCode := self newCode.
    sourceCode withoutSyntaxHighlight.
    whiteSpace := self newNullPresenter.
    textArea := self newText.
    codeCaption comment ifNotNil: [ textArea text: codeCaption comment ].
    btnAdd := self newButton.
    btnResolve:= self newButton.
    btnResolve label: 'Resolve'.

    "Setting up Widgets"
    sourceCode text: codeCaption captionSource node newSource; enabled: false; withScrollBars.
    textArea autoAccept: true; withScrollBars.
    editMode
        ifTrue:[
            btnAdd label: 'Apply'.
        ]
        ifFalse:[
            btnAdd label: 'Add'.
            btnResolve disable
        ].

    "Layout order"
    self focusOrder
        add: sourceCode;
        add: whiteSpace;
        add: textArea;
        add: btnAdd;
        add: btnResolve.
```

Figura 49. Código del método initializePresenters de la clase CodeCaptionWindow.

Cada elemento gráfico de la ventana se denomina “widget” y se instancian usando los métodos propios del *SpPresenter.* En el caso del código fuente a revisar, se llama al método *newCode* que instancia un área de texto de código fuente. Lo mismo se hace para cada widget presente en la vista y también se le especifican distintas propiedades como el orden dentro de la ventana, los textos y los datos asociados. Para el código fuente se le asigna como dato el código que el usuario seleccionó al llamar a agregar una nueva revisión de código.

connectPresenters

En este método se especifica la lógica asociada a cada acción requerida por los elementos interactuables de la ventana como los botones. En este caso tenemos los botones para guardar o resolver la revisión.

```
connectPresenters
    btnAdd
        action: [
            self addCodeCaptionToProject.
            self saveCodeCaptionToProject: false.
            self openCodeCaptionListWindow.
            self window close ].
    btnResolve
        ifNotNil: [
            btnResolve
                action: [
                    self codeCaptionProject resolveCodeCaption: self codeCaption.
                    self saveCodeCaptionToProject: true.
                    self openCodeCaptionListWindow.
                    self window close ] ]
```

Figura 50. Código del método connectPresenters de la clase CodeCaptionWindow.

Para el botón de guardado de revisión (btnAdd) se define como acción la lógica:

1. Agregar la revisión recién creada o editada al proyecto CodeCaption. Esto asigna el texto ingresado por el usuario como comentario de la revisión y le asigna como autor a dicho usuario para luego llamar a que *CodeCaptionProject* agregue la revisión a su lista.
2. Llamar a guardar el proyecto CodeCaption a disco, persistiendo así el guardado de la revisión.
3. Llamar a abrir la ventana del listado de las revisiones del proyecto CodeCaption así el usuario puede observar la revisión guardada entre las demás del proyecto.
4. Cerrar la ventana actual de agregado/editado de la revisión.

Para el botón de resolver la revisión (btnResolve) se define como acción la lógica:

1. Indicar al proyecto CodeCaption que contiene la actual revisión a que la resuelva enviándola como parámetro.
2. Llamar a guardar el proyecto CodeCaption a disco, persistiendo el proyecto sin la revisión que se acaba de resolver.
3. Llamar a abrir la ventana de listado de las revisiones del proyecto CodeCaption así se observan las revisiones presentes sin la actual recién resuelta.
4. Cerrar la ventana actual de agregado/editado de la revisión.

El método *saveCodeCaptionToProject: isResolve* llamado por las acciones en *connectPresenters* llama a *CodeCaptionProject* para que se guarde a sí mismo y le indica el autor del cambio y si el cambio fue un guardado de una revisión o su resolución.

```
saveCodeCaptionToProject: isResolve
   | repository commitText |
   commitText := isResolve
      ifTrue: [ 'Resolved CodeCaption: ' ]
      ifFalse: [ 'Saved CodeCaption: ' ].
   commitText := commitText , '"' , codeCaption comment , '"'.
   repository := IceRepository registry
      detect: [ :each | each includesPackageNamed: codeCaption package name ].
   codeCaptionProject
      saveCodeCaptionProjectTo: (CodeCaptionProject codeCaptionsDirectoryForRepository: repository)
      withText: commitText
      andAuthor: codeCaption author
```

Figura 51. Código del método saveCodeCaptionToProject de la clase CodeCaptionWindow.

4.8.2.2 Listado de Revisiones de código CodeCaption (CodeCaptionListWindow)

La vista de listado de las revisiones de código CodeCaption consta de la clase *CodeCaptionListWindow* que representa una ventana gráfica de tipo tabla con todas las revisiones de un proyecto.

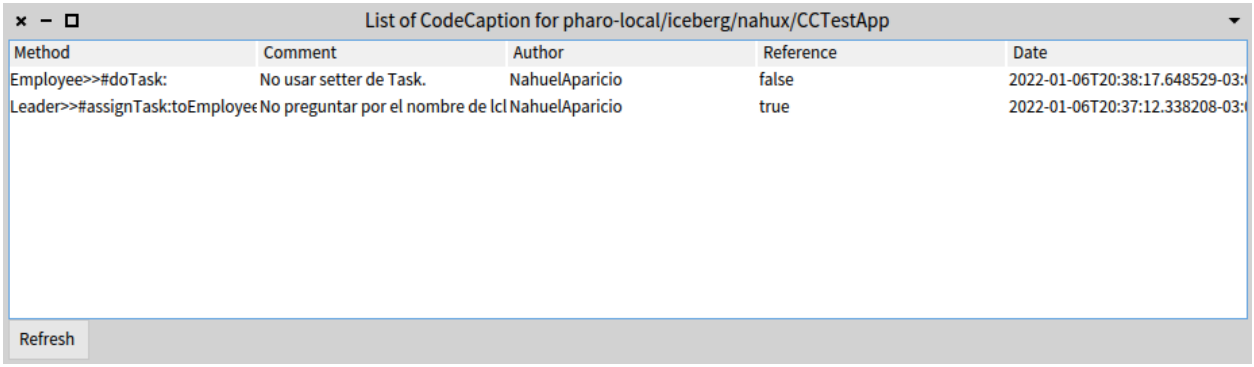


Figura 52. Ventana del listado de revisiones CodeCaption.

De cada revisión se muestran en forma de columna los campos *reference* (para indicar si el código no fue modificado luego de la creación de la revisión), *method* (el método donde se encuentra el código revisado), *comment* (el comentario de la revisión), *author* (el usuario autor de la revisión y *date* (fecha de creación de la revisión).

También se agrega un botón “Refresh” para que cuando se haga click, se recargue la tabla con las revisiones por si ocurrió algún cambio en las mismas o en el código al que referencian y se necesite actualizar la información de la tabla.

La clase *CodeCaptionListWindow,* (al igual que la de la vista de agregado/edición de una revisión) hereda de la clase presetenter de Spec 2.0 *SpPresenter* que se usa para armar ventanas gráficas personalizables dentro de Pharo. Para instanciar una ventana y especificar el diseño de la misma es necesario implementar el método de clase *defaultSpec.*

```
defaultSpec
    ^SpBoxLayout newVertical
        add: #ccList;
        yourself
```

Figura 53. Código del método de clase defaultSpec de la clase CodeCaptionListWindow-

Como se ve en la figura, en este método se definen las distintas partes de la ventana y se las matchea con una variable de instancia mediante los símbolos #. En este caso, tenemos sólo la sección que será la tabla que contiene todas las revisiones del proyecto, que se matchea como #ccList para indicar que la variable de instancia ccList definirá su forma y estilo.

Para instanciar la ventana solo hace falta llamar al método de clase *on: aCodeCaptionProject* enviándo el proyecto como parámetro. Luego para abrir efectivamente la ventana se debe llamar el método *openWithSpec.*

```
openCodeCaptionListWindow
    (CodeCaptionListWindow on: (codeCaptionProject)) openWithSpec.
```

Figura 54. Ejemplo de llamado a la clase CodeCaptionListWindow para abrir una ventana de listado de revisiones CodeCaption.

Una vez definida la instancia de la ventana es necesario implementar los siguientes métodos de instancia para poder renderizar correctamente la misma en Pharo:

- setModelBeforeInitialization: aCodeCaptionComment
- *initializeWindow: aWindowPresenter*
- *initializePresenters*
- *updatePresenter*
- *connectPresenters*

setModelBeforeInitialization: aCodeCaptionProject

Este método recibe como parámetro una instancia *CodeCaptionProject* que representa al proyecto CodeCaption con las revisiones de código y asigna su variable de instancia *codeCaptionProject* con el parámetro. También procesa todas las instancias de sí misma, es decir de la *CodeCaptionListWindow*, y si encuentra una ventana igual con el mismo proyecto, la cierra. Esto evita que se abran varias ventanas iguales en el entorno.

```
setModelBeforeInitialization: aCodeCaptionProject
    super setModelBeforeInitialization: aCodeCaptionProject.
    self class
        allInstancesDo: [ :w |
            w codeCaptionProject
                ifNotNil: [ w codeCaptionProject name
                        ifNotNil: [ w codeCaptionProject name = aCodeCaptionProject name
                                ifTrue: [ w delete ] ] ] ].
    codeCaptionProject := aCodeCaptionProject
```

Figura 55. Código del método setModelBeforeInitialization de la clase CodeCaptionListWindow.

initializeWindow: aWindowPresenter

En este método se especifica la lógica para instanciar la ventana gráfica de Spec, en este caso solo se especifica el título. En dicho título se encuentra el directorio local del repositorio donde se encuentra el proyecto con las revisiones a modo de información para el usuario.

```
initializeWindow: aWindowPresenter
    aWindowPresenter title: 'List of CodeCaption for ', codeCaptionProject gitDirectory .
```

Figura 56. Código del método initializeWindow: de la clase CodeCaptionListWindow.

initializePresenters

Para este método se necesita especificar todo lo relacionado a los elementos gráficos que componen la ventana. Se instancian las distintas partes con su forma, orden, estilo y los datos asociados en base al modelo de datos cargado en *setModelBeforeInitialization,* (el proyecto en la variable *codeCaptionProject*).

```
initializePresenters
    "Initialize the window layout"
    btnRefresh := self newButton.
    btnRefresh label: 'Refresh'.
    ccList := self newTable.
    ccList
        addColumn: (SpStringTableColumn title: 'Method' evaluated: #fullMethodName);
        addColumn: (SpStringTableColumn title: 'Comment' evaluated: #comment);
        addColumn: (SpStringTableColumn title: 'Author' evaluated: #author);
        addColumn: (SpStringTableColumn title: 'Reference' evaluated: #nodeFound);
        addColumn: (SpStringTableColumn title: 'Date' evaluated: #timeStamp);
        activateOnDoubleClick;
        beResizable;
        contextMenu:
            (SpMenuPresenter new
                addItem: [ :item |
                    item
                        name: 'Browse';
                        icon: (self iconNamed: #browse);
                        action: [
                            self browseMethod: ccList selectedItem
                        ]
                ]);
        whenActivatedDo:
                [ :selection | (CodeCaptionWindow on: selection selectedItem) openWithSpec ].
```

Figura 57. Código del método initializePresenters de la clase CodeCaptionListWindow

Cada elemento gráfico de la ventana se denomina “widget” y se instancian usando los métodos propios del *SpPresenter.* En este caso existen dos: la tabla con las revisiones de código y el botón “Refresh”.
Para el widget de la tabla se especifica que el elemento es una tabla y se definen las columnas a mostrar. Además, se define a qué campo de cada variable de CodeCaptionComment de la colección *ccList* se matchea cada columna. Para el widget del botón Refresh se indica que es un botón y se le asigna una etiqueta.
También se define un menú de contexto a aparecer cuando se haga click derecho en una fila de la tabla correspondiente a una revisión. Desde este menú de contexto se puede seleccionar el botón “browse” que llevará al usuario al código del método al que la revisión hace referencia.
Por último se especifica que al hacer doble click en una fila se activa el código especificado en *whenActivatedDo:* que abre la ventana de edición de la revisión seleccionada.

connectPresenters

En este método se especifica la lógica asociada a cada acción requerida por los elementos interactuables de la ventana. En este caso solo está el botón de Refresh de la tabla.

```
connectPresenters

    | repository |
    btnRefresh action: [
        repository := CodeCaptionProject getRepositoryForPackageName:
                          codeCaptionProject packages anyOne name.
        repository isNotNil ifTrue: [
            repository repositoryDirectory isNotNil ifTrue: [
                codeCaptionProject := CodeCaptionProject
                                          loadProject:
                                          (CodeCaptionProject
                                              codeCaptionsDirectoryForRepository:
                                              repository)
                                          packages: repository workingCopy packages ] ].
        self updatePresenter ]
```

Figura 58. Código del método connectPresenters de la clase CodeCaptionListWindow

Se especifica la lógica a ejecutar al clickear el botón. Esta lógica toma el proyecto CodeCaption asociado a la tabla de revisiones y vuelve a parsear todas las revisiones de disco para actualizar la información.

updatePresenter

Este método se ejecuta para inicializar la ventana en el entorno, y en este caso particular se usa para llenar la tabla de revisiones en la variable de instancia *ccList* obteniendo las revisiones del proyecto.

```
updatePresenter
    ccList items: codeCaptionProject codeCaptions asOrderedCollection.
```

Figura 59. Código del método updatePresenter de la clase CodeCaptionListWindow

4.9 Almacenamiento de CodeCaption

Para el almacenamiento de las revisiones de código se implementa un guardado en disco, todas las modificaciones de CodeCaption en un proyecto quedan reflejadas en un archivo de texto dentro de la carpeta del repositorio del mismo. Esto significa que cada vez que se modifiquen los CodeCaption se modificará dicho archivo y se verá reflejado el cambio en el git del proyecto.

El contenido del archivo guardado en disco no es un archivo de texto plano común, utiliza el formato STON (Smalltalk Object Notation). STON es un formato de intercambio de datos legible para el humano que nos permite serializar objetos Smalltalk. Está influenciado fuertemente en el formato JSON (Javascript Object Notation) para serializar objetos Javascript e incluso es compatible con él. Sin embargo, resulta difícil poder serializar objetos Smalltalk de cierto

tamaño y complejidad en formato JSON ya que este está limitado a representar listas y mapas y no permite conceptos como objetos y/o clases de distinto tipo ni estructuras circulares como grafos.

En este proyecto es utilizada la librería que nos proporciona STON para serializar y deserializar los CodeCaption en Pharo Smalltalk. En el caso que la librería no pueda serializar el objeto Smalltalk se necesita implementar el método *stonOn***:** *stonWriter* dentro de la clase afectada, indicando cómo generar el texto que represente a la clase. Justamente en este proyecto se implementa este método en dos clases de Pharo Smalltalk: RBProgramNode y IceLibgitRepository. En la primera la herramienta le indica a STON que guarde el resultado del método *dump* de la misma clase que genera una expresión literal que recrea el objeto. En la segunda es usado el método *asString* de la clase Object que genera un texto simple representando a la clase.

4.9.1 Guardado de CodeCaption

Como ya fue mencionado anteriormente, para guardar un CodeCaption en disco es utilizado el formato STON y su librería que nos provee las herramientas para serializar nuestros objetos. En definitiva, lo que se guarda en el archivo STON es una instancia de la clase CodeCaptionProject, que agrupa todos los CodeCaption de un proyecto.

A continuación se detalla el proceso de almacenamiento al agregar una nueva revisión a un proyecto. Lo primero que un usuario revisor de la herramienta debe hacer es identificar la línea de código que se quiere revisar y agregar el comentario de texto acorde. Cuando el usuario abre la ventana de agregado, se crea una instancia de *CodeCaptionProject* (en caso de no existir ya), y se la inicializa con una colección vacía de CodeCaptions. Luego el usuario llama a guardar la revisión con su comentario, por lo que se agrega el nuevo CodeCaption al proyecto y se serializa y guarda la instancia de CodeCaptionProject al disco:

```
btnAdd
    action: [
        self addCodeCaptionToProject.
        self saveCodeCaptionToProject: false.
```

Figura 60. Código asociado al botón agregar en la ventana de agregado de CodeCaption.

El método *saveCodeCaptionProject: isResolve* contiene la lógica de detectar a qué repositorio pertenece el método y clase que contienen el código revisado. Primero arma el texto del commit correspondiente, ya sea si se guarda una revisión o se resuelve. Luego con la información del repositorio al que pertenece el código revisado, obtiene la ubicación de guardado del archivo STON en el disco, dentro de la carpeta con el código fuente del repositorio del proyecto. Con esa ubicación, llama a *saveCodeCaptionProjectTo: aFilePath* para crear o modificar el archivo STON con el nuevo CodeCaption:

```
saveCodeCaptionToProject: isResolve
   | repository commitText |
   commitText := isResolve
      ifTrue: [ 'Resolved CodeCaption: ' ]
      ifFalse: [ 'Saved CodeCaption: ' ].
   commitText := commitText , '"' , codeCaption comment , '"'.
   repository := IceRepository registry
      detect: [ :each | each includesPackageNamed: codeCaption package name ].
   codeCaptionProject
      saveCodeCaptionProjectTo: (CodeCaptionProject codeCaptionsDirectoryForRepository: repository)
      withText: commitText
      andAuthor: codeCaption author
```

Figura 61. Código del método de guardado de las revisiones CodeCaption de un proyecto.

Ya dentro del método de guardado se hace uso de la librería STON para obtener la serialización de la instancia *CodeCaptionProject*, con la llamada a: *STON toStringPretty: self*. Con esto ya se obtiene el texto formateado que representa a la instancia y sólo queda guardar dicho texto en el archivo en disco:

```
saveCodeCaptionProjectTo: aFilePath withText: aCommentText andAuthor: anAuthorName
   | ston workingDir fstream fullRepositoryPath commitText |
   ston := STON toStringPretty: self.
   workingDir := FileSystem disk workingDirectory.
   (workingDir / aFilePath) ensureCreateFile.
   fstream := (workingDir / aFilePath) writeStream.
   fstream nextPutAll: ston.
   fstream close.
```

Figura 62. Código que hace el guardado de las revisiones CodeCaption de un proyecto en un archivo en disco dentro de la carpeta del repositorio.

Al terminar el proceso, queda un archivo en el repositorio en disco del proyecto dentro de la carpeta *CodeCaptions* con el contenido:

```
CodeCaptionProject {
    #name : 'pharo-code-caption',
    #codeCaptions : Set [
        CodeCaptionComment {
            #id : 414030592,
            #repository : 'IceLibgitRepository(pharo-code-caption)',
            #package : RPackage {…
            },
            #captionNode : CodeCaptionNode {…
            },
            #nodeFound : true,
            #comment : Text {
                #string : 'Esta variable no es necesaria',
                #runs : RunArray {…
                }
            },
            #resolved : false,
            #author : 'NahuelAparicio'
        }
    ],
    #packages : Set [
        @4
    ],
    #gitDirectory : 'pharo-local/iceberg/nahux/pharo-code-caption'
}
```

Figura 63. Contenido de ejemplo de un archivo STON con las revisiones de código de un proyecto.

Como paso final, la herramienta se comunica con la terminal del Sistema Operativo corriendo el entorno Pharo para agregar el cambio del archivo a git ejecutando los comandos “add” y “commit”. Esto se explica en detalle en el capítulo Integración con Repositorio Git más adelante. Hecho el “commit” solo quedaría realizar un “push” al repositorio remoto para que la sincronización quede completa y disponible para el resto del equipo.

4.9.2 Carga de CodeCaption

Cuando se tiene almacenado un archivo de CodeCaption dentro de la carpeta del repositorio del proyecto, la herramienta puede detectar que existen CodeCaption para ese proyecto y cargarlos. La detección ocurre cuando se quiere agregar un nuevo CodeCaption o cuando se quiere listar los CodeCaption disponibles de un proyecto (si los hay).

En el caso del agregado de un nuevo CodeCaption es necesario cargar el objeto *CodeCaptionProject* del disco si existe, ya que es necesario agregar el nuevo CodeCaption a la lista del proyecto. En el caso de listar los CodeCaption del proyecto se debe cargar

CodeCaptionProject para poder obtener la información de las revisiones ya cargadas en el archivo en disco.

Para poder realizar la carga se llama al método de clase *loadProject: aFilePath packages: icePagackes* de la clase *CodeCaptionProject*. Este método no hace más que cargar el contenido del archivo especificado en *aFilePath* para los distintos paquetes del repositorio e intentar instanciar la clase con dicha información.
En un principio se intenta leer el archivo de disco, en caso de no existir se asume que no existen CodeCaption creados para dicho proyecto. En el caso de que sí exista, se lee el contenido del archivo y se llama a: *STON fromString: fstream contents*. El método fromString de la librería hace la deserialización del texto instanciando la clase *CodeCaptionProject*. Sin embargo, luego de instanciar la clase es necesario llamar por cada uno de los paquetes del repositorio a uno de sus métodos: *loadNodesForPackage: aPackage*, ya que la serialización de un CodeCaption es limitada por la complejidad de los nodos AST del código de Pharo que contiene y porque también es posible que el código referenciado en el archivo de disco haya cambiado en el código real del entorno.

```
loadProject: aFilePath packages: icePackages
   | working_dir fstream project |
   working_dir := FileSystem disk workingDirectory.
   (working_dir / aFilePath) exists
      ifTrue: [
         fstream := (working_dir / aFilePath) readStream.
         project := STON fromString: fstream contents.
         "Load all CodeCaptions for all project packages"
         icePackages do: [ :p | project loadNodesForPackage: (RPackage named: p packageName) ].
         ^ project
      ]
      ifFalse: [ ^ nil ]
```

Figura 64. Código del método loadProject: aFilePath packages: icePackages

El método *loadNodesForPackage: aPackage* recorre todos los *CodeCaptionComment* de *CodeCaptionProject* y se encarga de instanciar correctamente los nodos AST comparándolos con las instancias concretas del entorno Pharo (utilizando el patrón Visitor, explicado en el capítulo 4.10.2), si la comparación entre el real y el almacenado da igual se instancia normalmente, sino se establece un Flag para indicar que el código fue modificado luego de la creación del CodeCaption.

```
loadNodesForPackage: aPackage
   | allMethods foundMethod |
   allMethods := aPackage methods.
   codeCaptions
      do: [ :c |
         (RPackage organizer packageNamed: (c package name)) == aPackage ifTrue: [
            c captionSource node isString
               ifTrue: [ c captionSource node: (Object readFrom: c captionSource node) ].
            c captionSource methodNode isString
               ifTrue: [ c captionSource
                  methodNode: (Object readFrom: c captionSource methodNode) ].
            foundMethod := nil.
            allMethods
               do: [ :pm |
                  "Check if method and class pairing from CodeCaption File exists in the source code"
                  (pm methodNode selector = c captionSource methodNode selector
                     and: pm methodNode methodClass = c captionSource nodeClass)
                     ifTrue: [ foundMethod := pm methodNode ] ].
            foundMethod
               ifNil: [ c nodeFound: false ]
               ifNotNil: [ c captionSource methodNode: foundMethod.
               "Search the CodeCaptionComment node in the methodNode to see if it was modified"
                  c nodeFound: (self method: foundMethod hasNode: c captionSource node) ]
         ].
      ]
```

Figura 65. Código del método loadNodesForPackage: aPackage que instancia los nodos AST de las porciones de código revisadas.

Luego de terminar la ejecución de carga del proyecto se retorna la instancia de *CodeCaptionProject* con todas las revisiones de código presentes en el proyecto para poder ser utilizado desde la UI de la herramienta.

4.10 Integración con los AST de Pharo Smalltalk

En el entorno de desarrollo Pharo Smalltalk existe la filosofía de que todo es un objeto, y además todo el código de su implementación es accesible desde el mismo entorno. Esto hace que se puedan explorar todas las clases que hacen al funcionamiento del entorno e incluso sea posible modificar su funcionamiento.

Para este trabajo interesa sobre todo poder manipular la representación del código Smalltalk que genera Pharo mismo en su entorno. La representación del código dentro de Pharo se realiza mediante Árboles de Sintaxis Abstracta (AST) y su implementación se puede observar en su totalidad en el entorno. La herramienta CodeCaption necesita interactuar con dicha representación del código que un desarrollador va creando en el entorno para poder revisarlo.

La interacción entre el AST y la herramienta ocurre principalmente en dos partes, en la estructura de datos de cada revisión al crear una nueva revisión de código, y en el parseo de

las revisiones guardadas y comparación del código del entorno con la referencia de cada revisión.

4.10.1 Referencia a nodo del AST en las revisiones

Como se explica en el capítulo 4.4 - CodeCaptionSource, cada revisión CodeCaption contiene en su estructura de datos la referencia hacia el nodo (del AST) que representa la sección del código revisado. Esta es la forma elegida para poder saber exactamente qué porción del código es la que se está revisando, ya que permite una implementación más simple y fácil de la herramienta al no tener que manipular el texto plano del código.

Para acceder a la ventana de creación de una nueva revisión de código es necesario que el revisor seleccione el código a revisar y haga clic en el botón para crear una revisión. En este momento se ejecuta el código de inicialización del botón que prepara los datos para crear una nueva revisión vacía:

```
prepareFullExecutionInContext: aToolContext
    | method packageName |
    super prepareFullExecutionInContext: aToolContext.

    "Set Iceberg repository"
    method := aToolContext lastSelectedMethod.
    packageName := method package name.
    repository := CodeCaptionProject getRepositoryForPackageName: packageName.
    repository repositoryDirectory
        ifNil: [ self inform: 'No repository was found for this package!' ]
        ifNotNil: [ "Set AST node"
            aToolContext selectedSourceNode.
            node := aToolContext selectedSourceNode ]
```

Figura 66. Código de inicialización del botón para abrir ventana de agregado de una revisión CodeCaption

Este código recibe del entorno de Pharo información como el paquete, método y el nodo seleccionado (*aToolContext selectedSourceNode)*. Con estos datos el código de ejecución del botón de agregado genera una nueva revisión vacía y llama a abrir la ventana de creación de revisión para que el revisor deje su comentario:

```
execute
    repository repositoryDirectory
        ifNotNil: [ (CodeCaptionWindow
                on:
                    ((CodeCaptionComment
                        newWithNode: node
                        package: node methodNode methodClass package
                        methodNode: node methodNode
                        repository: repository) nodeFound:true)) openWithSpec ]
```

Figura 67. Código de ejecución del botón para abrir ventana de agregado de una revisión CodeCaption

La clase CodeCaptionWindow es llamada entonces para que se encargue de dibujar la ventana de creación de revisión y recibe ya una revisión vacía con toda la información de qué parte del código se está revisando.

Al guardar la revisión con su comentario, la misma se almacena en disco, (capítulo 4.9.1) conteniendo la información del nodo del AST que representa al código revisado, al igual que el resto de las revisiones almacenadas.

4.10.2 Uso del patrón Visitor en el AST al realizar el parseo de las revisiones

En el capítulo 3.2.2 se explica en detalle los distintos cambios que sufre la representación AST cuando se modifica un código en Pharo Smalltalk y también qué sucede con la revisión CodeCaption que hace referencia a un nodo de dicho AST.

El proceso de control sobre los cambios en los nodos del AST que están referenciados por alguna revisión es ejecutado cuando se hace el parseo de las revisiones desde el disco (capítulo 4.9.2). En el parseo de las revisiones se recorre cada revisión verificando a qué lugar del código corresponde cada una (combinación paquete - clase - método - nodo AST del código del método). Si se encuentra el paquete, clase y el método dentro del código de Pharo Smalltalk lo necesario para cargar correctamente la revisión es verificar si la porción de código revisado (nodo del AST) existe también.

Para verificar la existencia del nodo en el AST actual del método de Pharo se utiliza el patrón Visitor. Esto se hace en el método *method: aMethodNode hasNode: anRBProgramNode*:

```
method: aMethodNode hasNode: anRBProgramNode
    | visitor |
    "Check if both are the same node (same method node)"
    (anRBProgramNode isMethod) ifTrue: [(aMethodNode body = anRBProgramNode body) ifTrue: [ ^ true ]].
    visitor := CodeCaptionRBNodeMatcher newWithNode: anRBProgramNode.
    aMethodNode acceptVisitor: visitor.
    visitor astNode ifNotNil: [ ^ true ] ifNil: [ ^ false ]
```

Figura 68. Implementación de método para verificar si un nodo del AST se encuentra dentro de un método

Este método recibe el nodo AST del método y de la porción de código revisada.
Primero verifica si los nodos recibidos son iguales, es decir, que la porción de código revisada equivale a todo el método. De ser así retorna verdadero ya que el nodo revisado existe y es todo el método.

Si los nodos no son iguales, se recurre a utilizar el patrón Visitor mediante la creación de una instancia de la clase *CodeCaptionRBNodeMatcher,* que es el visitor que necesitamos. Luego enviamos el visitor hacia el nodo del método usando su método *acceptVisitor: visitor*.

La clase *CodeCaptionRBNodeMatcher* hereda de la clase *RBProgramNodeVisitor* que es provista por Pharo y no es más que un visitor abstracto para poder recorrer los nodos AST del código de Pharo Smalltalk. La clase *CodeCaptionRBNodeMatcher* tiene dos variables de instancia, una que representa al nodo referenciado en la revisión (node) y la otra al nodo equivalente encontrado en el AST del código de Pharo (astNode), si no se encuentra esa variable queda nula.
La clase padre *RBProgramNodeVisitor* tiene un método *visitX*, siendo X el tipo de nodo posible del AST, (como *ArrayNode*) por cada tipo de nodo distinto que existe en un AST de Pharo. Por esto, se debería heredar de esta clase e implementar el método del tipo de nodo que se quiera obtener y realizar la lógica correspondiente para que el visitor funcione correctamente:

Figura 69. Lista (incompleta) de los métodos disponibles en la clase RBProgramNodeVisitor para implementar

En el caso de CodeCaption, como se trabaja con cualquier porción del código de un método, habría que implementar el *visitX* para todos los tipos de nodos disponibles dentro de un nodo método. La alternativa más simple en cambio, es implementar el método *visitNode:* que se ejecuta para cualquier tipo de nodo AST que esté dentro del nodo al que se le envía el visitor.

```
visitNode: anASTNode
    self node newSource = anASTNode newSource
        ifTrue: [ astNode := anASTNode  ]
        ifFalse: [ super visitNode: anASTNode  ]
```

Figura 70. Implementación del método visitNode en CodeCaptionRBNodeMatcher

Gracias al uso del patrón Visitor, la implementación en la herramienta resulta muy sencilla. Se compara el código del nodo referenciado en la revisión CodeCaption con el nodo actual del recorrido del visitor, si son iguales, se asigna la variable de instancia astNode con el nodo encontrado, sino se sigue con el recorrido del árbol.

Luego, cuando termina la ejecución se retorna al método *method: aMethodNode hasNode: anRBProgramNode* que lo único que debe hacer es preguntar si existe un nodo en la variable de instancia astNode del visitor. Si existe quiere decir que el nodo referenciado en la revisión existe en el código de Pharo por lo que se instancia la revisión CodeCaption sin problemas. Si la variable es nula quiere decir que se ha cambiado el código referenciado por lo que se instancia la revisión con una advertencia de que la referencia al código original se ha perdido. La revisión entonces queda con una variable de instancia *reference* en false, ya que la referencia se ha perdido producto de una modificación del código posterior a la creación de la revisión. El revisor y/o desarrollador puede en este caso leer el código del método para verificar que el cambio de código realiza lo que la revisión pedía y resolver definitivamente la revisión del proyecto.

4.11 Integración con Repositorio Git

Una parte clave de la herramienta es poder hacer recibir y enviar las revisiones entre revisor y desarrollador de un proyecto sin tener que usar otro canal de comunicación por fuera del entorno de desarrollo. Para poder realizar esta tarea, se sincronizan las distintas revisiones de código CodeCaption entre los distintos colaboradores de un proyecto mediante el uso de la herramienta de versionado Git. Esto permite poder compartir las revisiones sin necesidad de establecer ningún canal de comunicación aparte ni tener ningún servidor escuchando las actualizaciones de las revisiones. Lo único que se necesita es que el proyecto utilice Git como herramienta de versionado y esté sincronizado con algún repositorio remoto.

El hecho de utilizar Git no solo permite a la herramienta sincronizar remotamente los cambios sino que también deja un historial de los mismos sin tener que implementar ninguna funcionalidad de auditoría y/o logueo. Cuando un CodeCaption es creado o modificado se registra dicho evento en Git mediante un “commit”, que tiene toda la información del cambio realizado. Si se quisiera saber quien resolvió una revisión CodeCaption que ya no está presente en el proyecto solo alcanza con analizar el log de commits de Git y allí se encontrará el commit que refleje el evento de resolución.

Git es una herramienta de control de versiones que funciona en base a archivos en un proyecto. Por esta razón, si Git detecta que hubo cambios en un proyecto quiere
decir que alguno de los archivos del mismo sufrió alguna modificación, se borró o es un archivo nuevo. La herramienta CodeCaption entonces utiliza almacenamiento de las revisiones en archivos, como es explicado en el capítulo Almacenamiento de CodeCaption. De esta forma queda asegurado que cada vez que se cree, modifique o se elimine (resuelva) una revisión Git detectará el cambio, por lo que CodeCaption hará un commit de ese cambio para confirmarlo y poder sincronizar con el repositorio remoto.
CodeCaption guarda todas las revisiones de código de un proyecto en un solo archivo .STON dentro de una carpeta /CodeCaptions en el directorio raíz del repositorio para facilitar el proceso de carga y descarga de cada CodeCaption en Pharo. Cuando hay cambios en las revisiones de un proyecto Git detecta que el archivo de ese proyecto fue modificado.

4.11.1 Utilización de Git dentro de Pharo mediante Iceberg

Para utilizar Git dentro de Pharo Smalltalk existe una herramienta del entorno llamada Iceberg. Esta herramienta permite al desarrollador poder versionar el código smalltalk de su proyecto con Git y poder sincronizarlo con su repositorio remoto.
Iceberg puede usarse tanto como para un proyecto propio donde es posible sincronizar con el repositorio remoto descargando y subiendo cambios, como para proyectos de terceros que se pueden utilizar como librerías en nuestro proyecto.

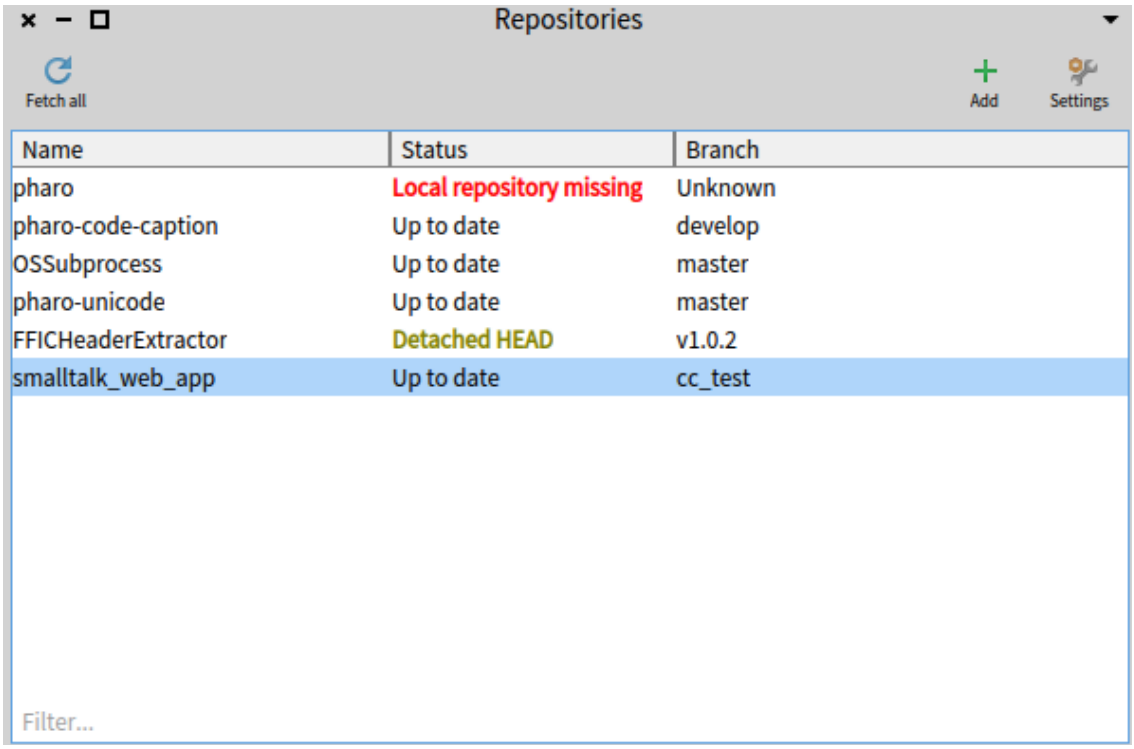


Figura 71. Ventana de ejemplo de Iceberg en Pharo.

La figura anterior muestra una ventana de ejemplo de los repositorios cargados por Iceberg en nuestra imagen de Pharo. Aquí se detallan el nombre de cada repositorio, su estado actual (si está actualizado o requiere alguna acción) y la rama de Git en la que se encuentra.

CodeCaption sincroniza las revisiones mediante Git y más específicamente hace uso de Iceberg para trabajar con los repositorios del entorno. Esto hace necesario el uso de Iceberg para versionar nuestro proyecto si se quiere utilizar CodeCaption en él. Cuando se llama a crear una nueva revisión de código en un método, la herramienta se encarga de identificar primero el paquete al cual pertenece el método y la clase en la que se está trabajando. Una vez identificado el paquete, se procede a revisar si dicho paquete pertenece a algún proyecto cuyo repositorio se encuentra en Iceberg. De no encontrar ningún repositorio para el paquete, se muestra un mensaje de advertencia informando al usuario que no es posible crear la revisión de código allí.

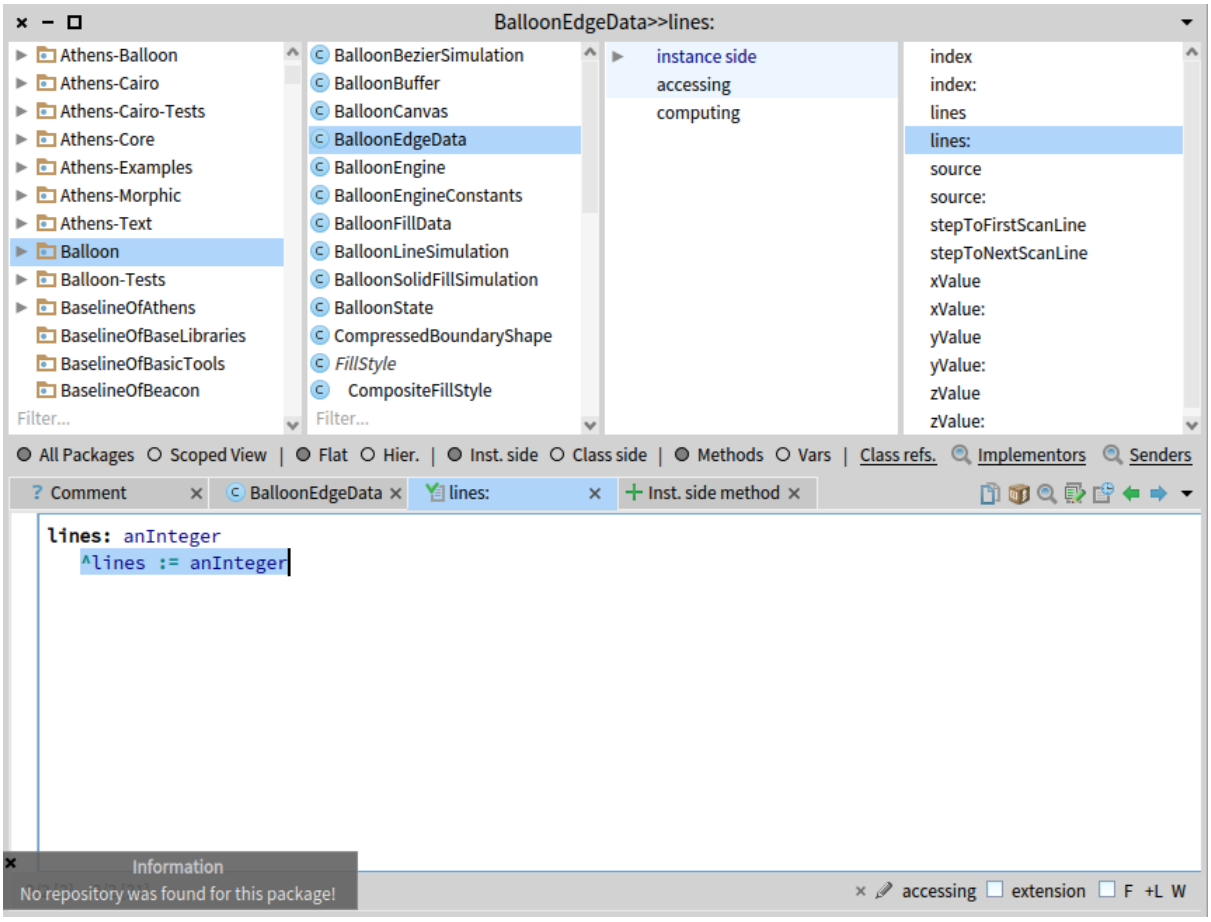


Figura 72. Ventana de edición de Pharo con mensaje de advertencia que no es posible crear una revisión en un paquete que no tiene un repositorio Iceberg asociado.

La figura anterior detalla esta situación. Se quiere crear una revisión para un paquete “Baloon” que forma parte del código base de Pharo y por ende no está asociado a ningún repositorio de Iceberg por lo que se muestra la advertencia en la esquina inferior izquierda.

Esta detección de repositorios por paquete actual se implementa de la forma siguiente.

```
getRepositoryForPackageName: aPackageName
    ^ (IceRepository registry)
        detect: [ :each | each includesPackageNamed: aPackageName ] ifNone: [ ^ nil ].
```

Figura 73. Código de método que detecta el repositorio al que pertenece el paquete al que pertenece el código que el usuario está revisando.

El código expresado en la figura anterior utiliza la clase *IceRepository* de Iceberg que contiene todos los repositorios cargados en la imagen. Con esta clase recorre todos los registros que contiene para detectar el repositorio que incluya el paquete que el desarrollador está trabajando. Ya con el repositorio encontrado se puede asociar la nueva revisión a un proyecto y crear o cargar (si ya existían revisiones) el archivo con las revisiones presentes del proyecto.

Una vez creada la revisión CodeCaption, queda modificado o creado el archivo con las revisiones del proyecto. Esto deja un cambio sin seguimiento en el git del proyecto, por lo que sería necesario agregar el archivo y su cambio a git y realizar un commit.

4.11.2 Ejecución de comandos Git mediante la terminal

Para poder realizar la tarea del trackeo de los cambios, CodeCaption utiliza los comandos Git desde la terminal del sistema operativo, que serían los comandos: *git add ARCHIVO* y *git commit -m "Mensaje de commit"*.
Desde Pharo existe la posibilidad de correr comandos de la terminal del sistema operativo de manera controlada mediante una librería llamada OSSubprocess[7]. Esta librería permite la ejecución de cualquier tipo de comando del sistema operativo host del usuario, solo hace falta especificar el comando a ejecutar y sus argumentos.

Cuando se realiza una modificación en las revisiones CodeCaption de un proyecto se modifica el archivo del mismo que contiene dichas revisiones. Esto hace que sabiendo el proyecto, se puede identificar el directorio local del repositorio y el archivo de las revisiones (dentro de la carpeta *CodeCaptions* en el directorio raíz del repositorio). Solo basta con realizar un *git add* a dicho archivo y luego un *git commit* para que el cambio quede registrado correctamente en git.

[7] Librería OSSubprocess de Pharo para ejecutar procesos del sistema operativo del usuario: https://github.com/pharo-contributions/OSSubprocess. NOTA: No permite la ejecución de comandos bajo sistemas no Unix (Windows).

```
OSSUnixSubprocess new
   command: '/usr/bin/git';
   arguments:
      (Array
         with: '-C'
         with: fullRepositoryPath
         with: 'add'
         with: 'CodeCaptions/');
   runAndWait.
OSSUnixSubprocess new
   command: '/usr/bin/git';
   arguments:
      (Array
         with: '-C'
         with: fullRepositoryPath
         with: 'commit'
         with: '-m'
         with: commitText);
   run.
```

Figura 74. Código de ejecución de comandos git add y git commit en Pharo.

En la figura anterior se observa cómo se realizan los comandos mencionados con la librería de Pharo. Por cada comando se instancia la clase OSSUnixSubprocess y se especifica el path ejecutable de git junto con los argumentos. En ambos casos se pasan los argumentos *-C* junto con el directorio del repositorio para especificar dónde ejecutar el comando. Luego en el primer caso se pasa el argumento "*add"* más el directorio *CodeCaptions/* para agregar el archivo de revisiones y en el segundo caso se pasa el argumento "*commit"* y *-m* junto con el mensaje del commit. El mensaje del commit dependerá si se crea o modifica una nueva revisión o si fue resuelta. En el primer caso el mensaje será del estilo: *[CodeCaption] Saved CodeCaption: "Comentario de Prueba" by: "Autor"* y en el segundo caso será del estilo: *[CodeCaption] Resolved CodeCaption: "Comentario de Prueba" by: "Autor"*. En ambos casos se especifica la acción junto con el texto del comentario y el autor de dicha revisión. Luego de realizado el commit, el repositorio git del proyecto queda limpio y listo para sincronizar con el repositorio remoto.

Para realizar la sincronización con el repositorio remoto el desarrollador debe utilizar la herramienta iceberg. Allí puede traerse los cambios mediante la acción "pull" y subir los cambios (como pueden ser las nuevas revisiones ya commiteadas por CodeCaption) mediante la acción "push".

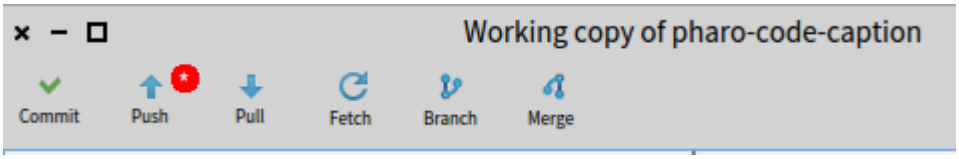


Figura 75. Ventana de iceberg con las acciones para sincronizar con el repositorio remoto.

Capítulo 5. Prueba de usuario

En los capítulos anteriores se explicó en detalle el diseño e implementación de la herramienta CodeCaption, en este capítulo se va a definir una prueba de uso de la herramienta con distintos usuarios.

5.1 Objetivos de la prueba

Esta prueba de usuario tiene como objetivos poder medir e identificar:

- La usabilidad de la herramienta.
- La satisfacción del usuario en el uso de la herramienta.
- Posibles mejoras y nuevas funcionalidades para la herramienta.

Es de interés del trabajo poder obtener de esta prueba una medición positiva en cuanto a la usabilidad y satisfacción en el uso de la herramienta.

5.2 Tipo de prueba

La prueba de usuario consta del uso de la herramienta y posterior respuesta de preguntas mediante una encuesta. Los usuarios son instruidos sobre cómo instalar y utilizar la herramienta para que puedan experimentar las funcionalidades que la componen para luego contestar una serie de preguntas en formato encuesta.

La prueba es asíncrona con usuarios remotos. No es necesaria la presencialidad ni una sincronización entre quien lleva a cabo la prueba y el usuario. La prueba es enviada al usuario mediante un link a un documento de Google con las instrucciones y la encuesta a contestar. Los usuarios pueden realizar la misma en el momento que quieran sin un límite de tiempo, y lo pueden hacer remotamente donde quieran.

Para poder demostrar correctamente su uso y aplicar la prueba, se creó una aplicación Smalltalk básica en un repositorio git público. Esta aplicación contiene al menos 5 code smells intencionales para que el usuario pueda encontrarlos y comentarlos con el uso de la herramienta. En el documento se explica el procedimiento de la prueba iniciando con la instalación del IDE y la aplicación.

5.3 Usuarios de la prueba

Los usuarios que realizaron la prueba tienen un perfil de desarrolladores calificados que estudiaron en la Facultad de Informática de la Universidad Nacional de La Plata. Por esto cuentan con un conocimiento del uso de Pharo Smalltalk y su lenguaje.
Se realizaron pruebas a 5 personas distintas del entorno del autor de esta tesina que encajan con el perfil mencionado. Se elige esta cantidad debido a restricciones de tiempo, la prueba es

solamente una prueba inicial para poder medir mínimamente la usabilidad y satisfacción de la herramienta. A cada uno de estos usuarios se les envía el documento con la prueba, que se explica en la siguiente sección (5.4 Definición de la prueba).

5.4 Definición de la Prueba

En la prueba de usuario entonces se definen ciertos pasos a seguir, divididos en 3 partes. Luego de la prueba se le indica al usuario responder una mínima encuesta de 4 preguntas en relación a su experiencia con la herramienta en la prueba. Toda esta definición con sus instrucciones se encuentran en un Google Doc que se le provee al usuario. El usuario también deberá escribir en las secciones indicadas del documento para realizar correctamente la prueba.

5.4.1 Parte 1: Instalación de la aplicación y herramienta, y búsqueda de code smells

Instalación del entorno Pharo Smalltalk (de ser requerido) y de la aplicación de prueba. La aplicación de prueba tiene un script de instalación (baseline) que instala automáticamente Code Caption y sus dependencias en el entorno.
Luego de la instalación se instruye al usuario a explorar el código de la aplicación de prueba y encontrar al menos 3 posibles code smells en el mismo.
Cada code smell encontrado en la aplicación debe ser anotado en el documento Google Doc que se le provee al usuario. Esta modificación del contenido en definitiva genera una entrada en el historial del documento con la modificación y la fecha y hora que será clave para el análisis de la prueba.

5.4.2 Parte 2: Creación de revisiones de código con la herramienta

Ahora se le indica al usuario crear una revisión de código Code Caption dentro de Pharo por cada code smell encontrado en la parte anterior. Para esto se le explica el proceso de creación de cada revisión y de listado de las mismas en el entorno.
Para finalizar se le pide al usuario ingresar en el Google Doc una captura de pantalla de las revisiones de código creadas dentro de la herramienta. La captura va a contener entre otras cosas: el comentario, a qué clase y método hace referencia y un timestamp con la fecha y hora de la creación.

5.4.3 Parte 3: Visualización de revisiones de código creadas por otro desarrollador.

En la última parte se prueba la funcionalidad de la herramienta para visualizar las revisiones de código creadas por otro colaborador del proyecto en el repositorio.

Para lograr esto se requiere que el usuario cambie de rama de git dentro del entorno a una rama que contiene varias revisiones de código para cada code smell intencionalmente agregado en la aplicación de prueba.
Ya en la rama correspondiente del proyecto de prueba, el usuario puede visualizar todas las revisiones creadas por otra persona en el entorno. Esto le muestra al usuario el final del proceso en cómo la herramienta se comporta en el uso dentro de un equipo de trabajo, con distintos colaboradores en distintos sitios físicos desarrollando y revisando código.

5.4.4 Encuesta

Al final de la prueba el usuario tiene unas preguntas para responder en cuanto a la experiencia que tuvo con el desarrollo de la prueba que sirven para medir la usabilidad de la herramienta, la satisfacción del usuario y posibles mejoras a considerar.
Las preguntas de la encuesta son las siguientes:

1. ¿Qué tal te pareció la experiencia del uso de la herramienta? ¿Te resultó fácil de usar? *Opciones*: Muy Fácil de Usar - Fácil de Usar - Normal - Difícil de Usar - Muy Difícil de Usar
2. ¿Qué tan bien se compara en términos de rapidez y facilidad de uso la realización de code review usando la herramienta con respecto a hacerlo manualmente? *Opciones*: 1 al 10, (siendo 1 mucho peor que Manual, 5 iguales y 10 mucho mejor que Manual)
3. ¿Usas una herramienta de este tipo dentro de un IDE para realizar code reviews cotidianamente? ¿Si no usas actualmente, te gustaría usar una herramienta de este tipo? *Opciones*: Me encantaría, Me da Igual, No usaría.
4. ¿Qué te gustaría mejorar o añadir a la herramienta?

Una vez respondidas las preguntas de la encuesta, la prueba para el usuario finaliza.

5.5 Análisis

Como se definió en la sección 5.1 Objetivos de la prueba, la prueba de usuario busca medir la usabilidad, satisfacción del usuario e identificar posibles mejoras a realizar al desarrollo. En este capítulo se toman los datos provistos por los usuarios en el documento de la prueba, tanto la encuesta como los pasos anteriores, analizando las respuestas y los tiempos para sacar las conclusiones.

5.5.1 Usabilidad

En esta prueba se mide la usabilidad de la herramienta haciendo foco en la eficiencia y en la eficacia de la misma. Para medir la eficiencia se miden los tiempos que le toma al usuario identificar los code smells en la aplicación de prueba con y sin la herramienta. Para la eficacia en cambio, se le encuesta al usuario sobre qué tán fácil y rápido le resultó el uso de la herramienta en la prueba.

Duración de Identificación de Code Smells (eficiencia)

En esta sección se analiza la duración de la identificación de los Code Smells de la aplicación de prueba por parte del usuario (Parte 1 y Parte 2 de la prueba). Lo importante para este trabajo es analizar y comparar el tiempo que le llevó a los usuarios la identificación sin usar la herramienta, (ingresando el comentario sobre el code smell en el documento de Google) con el tiempo que llevó usando la herramienta.

Sin el uso de CodeCaption

Para poder realizar la medición en la identificación sin la herramienta se utiliza la funcionalidad del historial del documento de Google. Es decir que se analiza el historial de cambios del documento (que tiene fecha y hora por cada cambio) sobre la porción del mismo donde el usuario debe escribir los comentarios sobre los code smells encontrados.

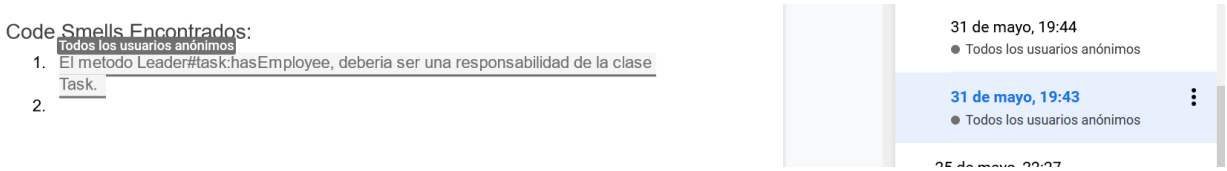


Figura 76. Ejemplo de historial de cambios del documento en la identificación de code smells del usuario.

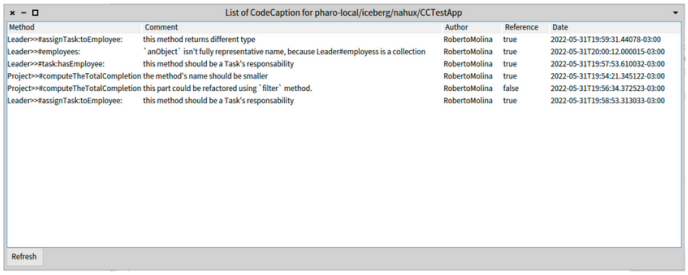
Se detecta en el historial de cambios desde el momento que el usuario encontró e ingresó el primer code smell hasta que ingresó el último encontrado al documento. La diferencia de minutos entre el primero y el último es el dato que se necesita para comparar con el uso de la herramienta.

Con el uso de CodeCaption

Para poder realizar la medición en la identificación con el uso de la herramienta se le requiere al usuario hacer un uso regular de CodeCaption, agregando los comentarios a los code smells encontrados dentro del IDE y de la herramienta. Cada revisión dentro de la herramienta se genera con la fecha y hora que fue creada. Por esto, se hace el mismo análisis que se hizo con el historial del documento en la sección anterior, pero ahora se hace con una captura de pantalla de la lista de las revisiones creadas por el usuario (conteniendo la fecha y hora de cada una en la captura).

Captura de pantalla con las revisiones creadas:

Pegar aquí la captura de pantalla con las revisiones de código creadas para cada code smell.



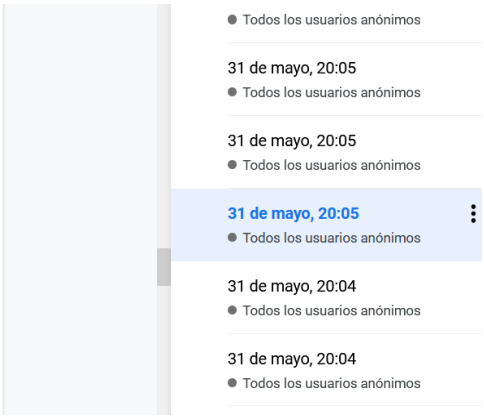


Figura 77. Ejemplo de captura de pantalla con la lista de revisiones para los code smells identificados en la prueba usando la herramienta.

Al igual que en la sección anterior se toma como dato los minutos transcurridos desde la primera revisión de code smells creada en la herramienta hasta la última.

Comparación de tiempos

La identificación de code smells sin el uso de la herramienta, escribiendo la revisión de cada uno en el documento de google tomó en promedio **12 minutos** para todos los usuarios. En cambio la identificación de los mismos dentro de la herramienta tomó en promedio unos **6,5 minutos**.

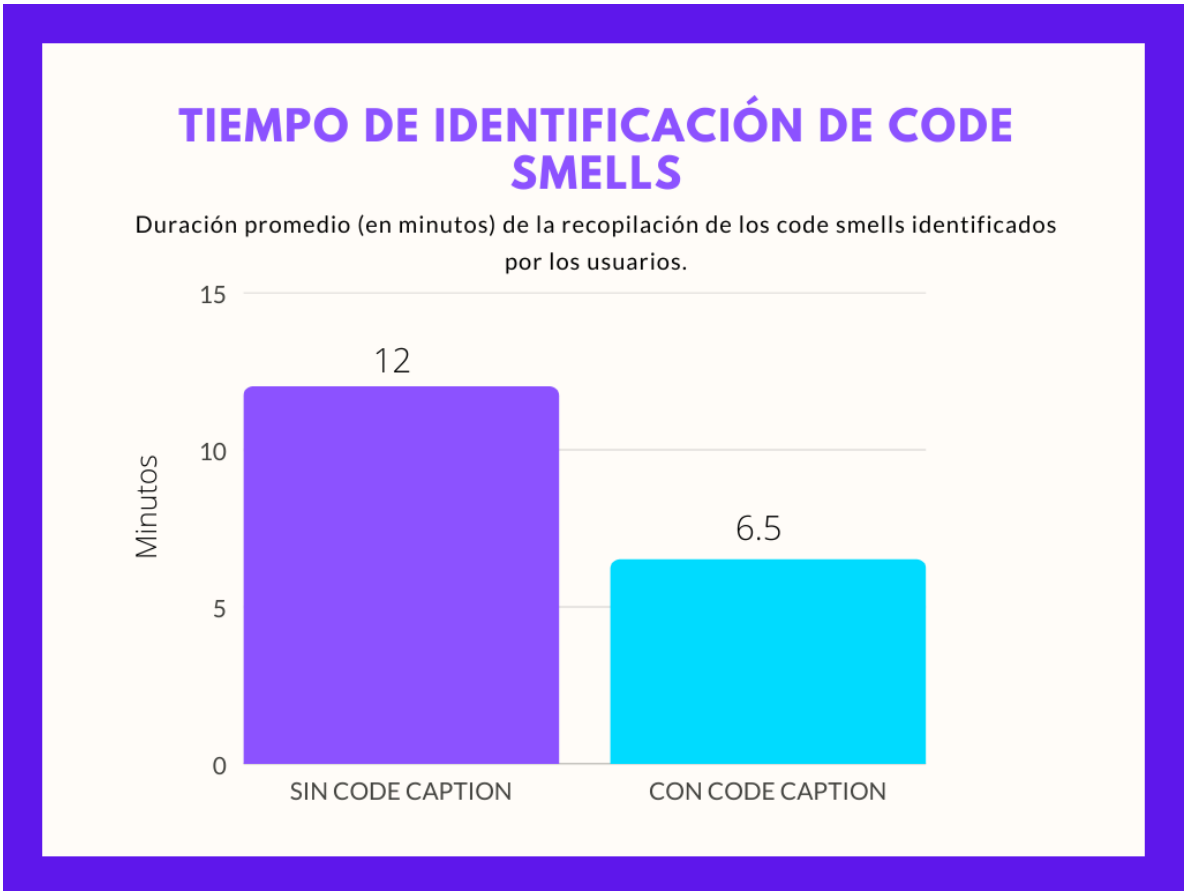


Figura 78. Comparativa gráfica del tiempo transcurrido para la identificación de code smells con el uso de la herramienta y sin la misma.

Se puede ver que con la herramienta el tiempo que tarda en promedio un usuario de la prueba es un poco más de la mitad que el tiempo que tarda sin el uso de la misma. Esto se puede explicar en tener tanto el código como las revisiones en el mismo contexto del IDE y no tener que usar un programa externo.

Encuesta sobre rapidez y facilidad de uso (eficacia)

En esta sección se analiza la respuesta a la pregunta 2 de la encuesta. La pregunta busca que el usuario indique qué tan fácil y rápido le resultó la experiencia con la herramienta comparada con la realización de la misma tarea sin la herramienta. La pregunta es:

- ¿Qué tan bien se compara en términos de rapidez y facilidad de uso la realización de code review usando la herramienta con respecto a hacerlo manualmente?

Teniendo como opciones: 1 al 10, (siendo 1 mucho peor que Manual, 5 iguales y 10 mucho mejor que Manual).

Las respuestas a la pregunta son las siguientes:

Figura 79. Histograma detallando el porcentaje de respuestas a la pregunta 2 de la encuesta sobre la facilidad y rapidez del uso de la herramienta por sobre el uso manual.

La mayoría está de acuerdo en que la experiencia desde la herramienta para la revisión de código es mejor que hacerlo mediante un documento externo.
Sin embargo, el grado de facilidad y rapidez entre los usuarios varía. El puntaje de la herramienta por sobre el uso manual más bajo contestado (6) fue seleccionado por un 40% de los usuarios, al igual que el medio (8) y un 20% eligió el mejor puntaje (10).

Entre los comentarios adjuntos a la respuesta encontramos por ejemplo:

- *"Es mejor que manualmente. Aunque quizás necesita alguna que otra funcionalidad para ser perfecto"*
- *"Está buena la posibilidad de realizar los code review desde el IDE/Workspace debido a que está todo en el mismo lugar, no es necesario usar otra herramienta o abrir el código desde otra plataforma."*

5.5.2 Satisfacción del usuario

La satisfacción del usuario con la herramienta es una métrica clave aunque más difícil de medir, es difícil de cuantificar y puede estar influenciada por la subjetividad del usuario.

Para poder realizar una medición se depende de una retroalimentación del usuario en cuanto a su experiencia con el uso de la herramienta. Por esto, se definen dos preguntas de la encuesta para poder medir la satisfacción de los usuarios, la pregunta 1 y 3.

Pregunta 1:

La pregunta es:

- ¿Qué tal te pareció la experiencia del uso de la herramienta? ¿Te resultó fácil de usar? *Opciones*: Muy Fácil de Usar - Fácil de Usar - Normal - Difícil de Usar - Muy Difícil de Usar

Esta pregunta solicita directamente al usuario a calificar la facilidad de la herramienta, y agregar un comentario sobre qué le pareció la experiencia en el uso. Con respecto a la facilidad, los usuarios contestaron:

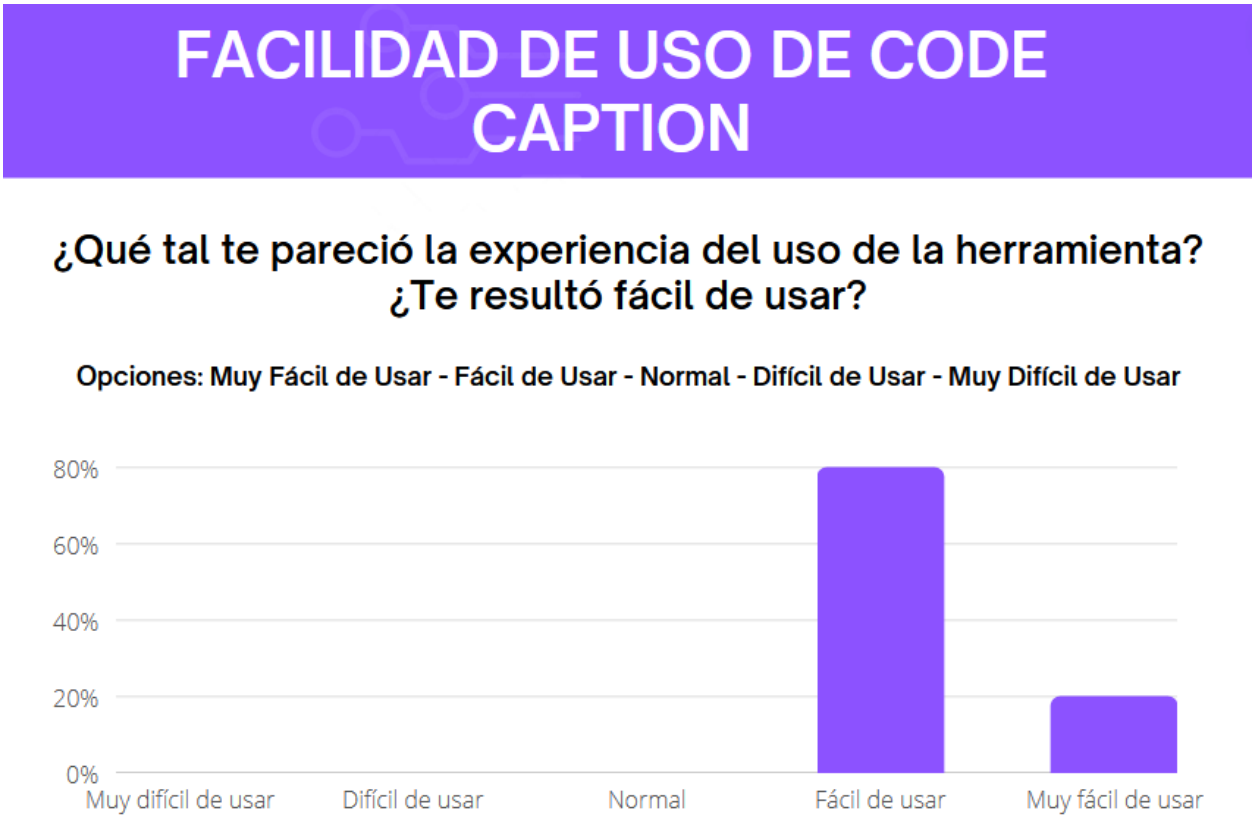


Figura 80. Histograma detallando el porcentaje de respuestas a la pregunta 1 de la encuesta sobre la facilidad de uso de la herramienta.

Como se puede apreciar en la figura anterior, el 80% de los usuarios de la prueba consideraron que la herramienta fue fácil de usar, mientras que el 20% consideró que la herramienta fue muy fácil de usar. Este dato resulta muy positivo para la herramienta ya que ningún usuario de los encuestados consideró que la herramienta no fuera fácil de usar, nadie tuvo problemas para instalarla y entender rápidamente cómo se usa.

Pregunta 3:

La pregunta es:

- ¿Usas una herramienta de este tipo dentro de un IDE para realizar code reviews cotidianamente? ¿Si no usas actualmente, te gustaría usar una herramienta de este tipo? *Opciones*: Me encantaría, Me da Igual, No usaría.

Esta pregunta no consulta directamente al usuario sobre su satisfacción al usar la herramienta pero hace referencia a un concepto que termina influyendo en la misma. La pregunta consulta al usuario si hace uso o usaría de herramientas para hacer code review y que se ejecuten dentro del contexto de un IDE, específicamente si usaría una herramienta de este estilo cotidianamente como programadores.
A esto los usuarios contestaron:

Figura 81. Histograma detallando el porcentaje de respuestas a la pregunta 3 de la encuesta sobre el uso de herramientas de code review dentro del contexto del IDE.

De las respuestas se aprecia que a un 80% de los encuestados le encantaría usar una herramienta de code review dentro del contexto del IDE y a un 20% le da igual. Esto ayuda a

reafirmar la mejora en la rapidez y comodidad que significa usar una herramienta de code review dentro del IDE sin tener que usar software externo.

Dentro de los encuestados que eligieron la opción *me encantaría*, la mayoría aclara que ya usa una herramienta del estilo cotidianamente, como sugiere un usuario:

- *"Si, cuando tengo la posibilidad trato de usar las herramienta de code review de VSCode."*

5.5.3 Mejoras a realizar

La última pregunta de la encuesta tiene como función consultar a los usuarios sobre posibles mejoras o nuevas funcionalidades a agregar a la herramienta. Al ser una pregunta abierta no es posible hacer una medición cuantitativa pero podemos identificar puntos en común en los comentarios de los usuarios para analizar nuevos cambios a la herramienta.

La principal cuestión a mejorar en la que coinciden los usuarios es en la visibilidad y navegabilidad de las revisiones por sobre el código.
En la herramienta actualmente para ver las revisiones creadas se requiere abrir una ventana aparte dentro del IDE con todas las revisiones del proyecto actual. Aunque es cierto que al hacerle *click derecho* a cada revisión se abre un menú donde se permite navegar directamente hacia el código revisado esto no es muy claro y tampoco lo suficientemente práctico.

También incluso al estar observando el código de un método que tiene una revisión no hay ninguna indicación de que existe verdaderamente allí una revisión sin tener que abrir el listado de revisiones. Una sugerencia de un usuario indica usar las "advertencias" provistas por el IDE para indicar que existe allí una revisión.

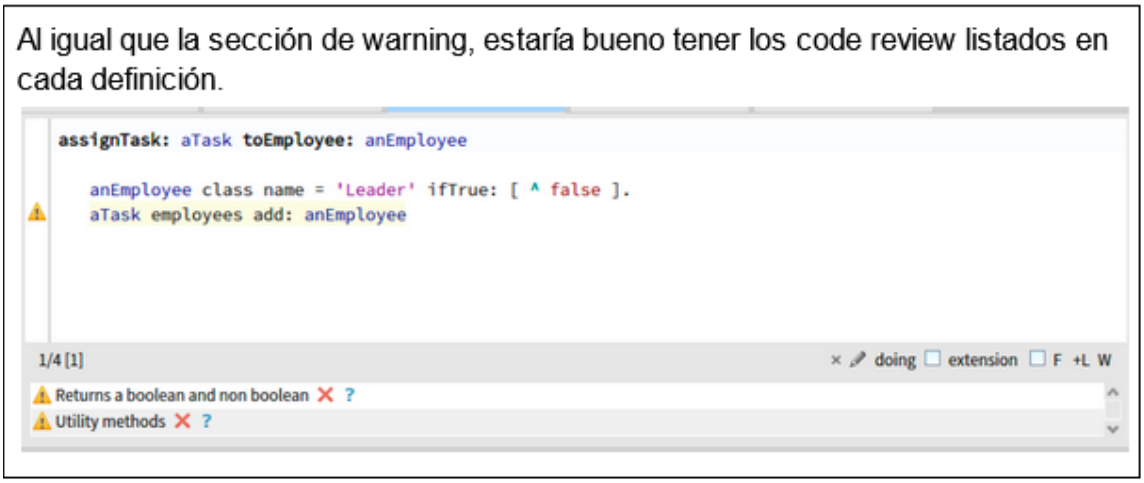


Figura 82. Respuesta a la pregunta 4 de la encuesta con una sugerencia de un usuario sobre cómo se podrían mostrar las revisiones de código de mejor forma dentro del IDE.

Capítulo 6. Conclusiones y Trabajo Futuro

6.1 Conclusiones

Durante el transcurso del trabajo se desarrolló un prototipo de herramienta para realizar revisiones de código remotas y asíncronas: CodeCaption. El diseño de la herramienta fue pensado como un plugin dentro del contexto de un IDE, para evitar el uso de programas externos por parte de los desarrolladores al revisar código.

Code Caption corre dentro del IDE Pharo Smalltalk, y a la vista del usuario funciona como una extensión de las funcionalidades del mismo. Permite a los usuarios aplicar revisiones de código dentro del IDE, ya sea tomando el rol de revisor o de desarrollador con su código siendo revisado.
Cada revisión es creada por un usuario directamente en la vista de edición de código del IDE, con un comentario hacia una porción específica del código de un proyecto. A su vez, la revisión se integra al repositorio Git del proyecto, para lograr la sincronización entre los colaboradores y dejar un historial de las revisiones en el versionado del repositorio. Se logra así que todos los involucrados en el proyecto puedan ver las revisiones que fueron creadas e interactuar con las mismas.

Se decidió incursionar en el desarrollo de este tipo de herramientas ya que nos encontramos en un contexto global donde el trabajo remoto ha tomado una relevancia y crecimiento muy grande. Es por esto que también es de vital importancia agilizar, en este contexto, las actividades de las metodologías ágiles como lo son las revisiones de código. Permitir a los desarrolladores realizar revisiones de forma remota y asíncrona ayuda a agilizar el proceso.
También resultó de gran interés técnico para el trabajo aplicar dichas revisiones de código directamente en la representación de AST del código de un proyecto. Esto explica en parte por qué se eligió específicamente el IDE Pharo Smalltalk, ya que todo el código que lo compone y su representación en forma de AST está a la vista del desarrollador para investigar, depurar y modificar. Además, ayudó a la elección del IDE el hecho que no abundan soluciones similares para el mismo, y también a que el autor de la tesina se encuentra familiarizado con el lenguaje.

En un principio, el trabajo comenzó con una investigación acerca de los Abstract Syntax Tree (AST) y su representación en Pharo. Desde el primer momento, el objetivo era poder enlazar de alguna manera un comentario de texto a un nodo del AST de Pharo que representa a una porción de código en Smalltalk. Aunque puede resultar compleja, la representación del AST en Pharo es lo suficientemente clara y concisa, por lo que este enlace se encontró relativamente rápido. Se decidió que sería indicado implementar el patrón Visitor para ubicar el nodo comentado fácilmente, aunque este sufriera modificaciones o cambie de sitio en el árbol.
Luego se continuó con la investigación sobre cómo implementar un plugin de Pharo. Para esto se decidió el uso de la librería Spec [22], que permite al desarrollador crear ventanas gráficas dentro del IDE con la funcionalidad que sea necesaria. También fue necesario investigar cómo

integrar dichas ventanas a las funcionalidades propias del IDE, para permitir crear revisiones de código e interactuar con las mismas. Esta fase de la implementación resultó compleja ya que no existe documentación completa sobre cómo extender las funcionalidades, así que hubo que recurrir a analizar el código fuente mismo de Pharo para entender el flujo de trabajo del IDE. Incluso, debido a esta complejidad, hubo que aceptar algunas limitaciones en cuanto a la experiencia de usuario de la herramienta, como el poder ver las revisiones “en tiempo real” mientras se edita el código o mejorar el comportamiento de los menús de contexto.
Se logró finalmente integrar con la interfaz gráfica de Pharo y obtener el AST del código con el que el usuario interactúa, junto con demás información del proyecto. Con esta representación fue posible permitir al usuario agregar un comentario, (revisión de código) y almacenar dicha revisión con un formato de texto tipo STON con todas las revisiones dentro del directorio raíz del proyecto. Luego se realizó la integración con el Git del proyecto dentro de Pharo mediante Iceberg para poder integrar el archivo de las revisiones al repositorio. Esta integración no resultó fácil debido a que Iceberg no contempla archivos que no son parte del código de Smalltalk como parte del repositorio. Por esto se decidió utilizar una librería que se comunica con la consola del Sistema Operativo para correr comandos Git.
Para ver las revisiones creadas dentro de Pharo, se implementó el parseo del archivo de texto con las mismas. En este paso hubo algunas complicaciones, ya que fue necesario implementar el patrón Visitor junto con una lógica un poco compleja. Esta lógica logra obtener nuevamente la representación AST del código del proyecto de Pharo. Luego encuentra en el mismo, el nodo comentado en la revisión guardada en el archivo de texto. Este parseo del nodo AST en forma de archivo a la representación en memoria del IDE tomó tiempo y requirió de ayuda por parte de la comunidad de Pharo.

Al final del desarrollo se optó por realizar pruebas de usuario con el fin de comprobar el correcto funcionamiento de la herramienta, usabilidad y conformidad de los usuarios con la misma. Se decidió por pruebas de usuario para aprender de la importancia de las mismas en el desarrollo de aplicaciones. Fue de gran utilidad para encontrar muchas posibilidades de mejora y puntos a destacar.

A fin de cuentas, se obtuvo un desarrollo, que representa un primer prototipo funcional de una herramienta de revisión de código dentro de Pharo. Cumple los objetivos planteados en el trabajo, logrando permitir a los desarrolladores realizar revisiones de código de forma remota y asíncrona sin necesidad de uso de software externo. El proyecto sirve como base para una herramienta robusta de revisión de código entre pares dentro de Pharo. Una herramienta que logre escalar dentro del constante crecimiento de equipos de desarrollo remotos.

6.2 Trabajo Futuro

En este apartado se listan los posibles trabajos futuros a partir de la tesina.

- Mejora de experiencia de usuario. A partir de la propia experiencia y de la de los usuarios de las pruebas, se destacó que uno de los puntos más débiles del desarrollo es la experiencia de usuario. Existe una limitación a las herramientas que ofrece Pharo para el desarrollo de interfaces de usuario pero se puede afirmar que la experiencia es mejorable. Principalmente se ve esto en la forma en la que el usuario puede ver las revisiones de código de un proyecto, ahora es posible mediante un listado de todas las revisiones del proyecto en general. Lo ideal sería brindarle la posibilidad al usuario de ver si existen revisiones en el código que está editando (mostrar una ventana pequeña de la revisión al pasar el mouse por el código por ejemplo.

- Desarrollo de un servicio para alojar las revisiones. Desarrollar un servicio que permita alojar las revisiones de código en una base de datos y sincronizar con el mismo mediante una API desde el IDE, en lugar de depender de Git para estas tareas es una posible alternativa.

- Agregar más interacciones a las revisiones. Las revisiones de código solo permiten ser creadas, modificadas y ser resueltas; esto se hizo para simplificar el desarrollo de la tesina. Sin embargo, podrían agregarse funcionalidades que hagan de la experiencia algo más “social”, como reacciones con emojis, respuestas, likes y dislikes, etc.

- Análisis de comentarios mediante el uso de Inteligencia Artificial. Se podría analizar el estado de ánimo de los comentarios, determinando por ejemplo qué porcentaje de “positivo” o “negativo” tiene el comentario. Luego esto sería mostrado estadísticamente y se podrían sacar conclusiones en cuanto a cómo y qué tan bien se están aplicando las revisiones, que tan efectivas pueden llegar a ser, etc.

Bibliografía

[1] Filev, Andrew. (2013). Expansion of remote teams: What drives it forward, and how is it shaping the future of project management. PM world journal, 2, 3

[2] Tekla S. Perry (Julio 2020) Will the Tech Workplace Ever Be the Same Again?

[3] TeamBlind (2020) Blind's coronavirus and WFH survey. https://www.teamblind.com/post/Blinds-coronavirus-and-WFH-survey-6mMVM2aY

[4] Greiler, M., Bird, C., Storey, M. A., MacLeod, L., & Czerwonka, J. (2016). Code reviewing in the trenches: Understanding challenges, best practices and tool needs.

[5] Maria Khalusova (2016) Peer Code Review from IDE. https://blog.jetbrains.com/upsource/2016/05/02/peer-code-review-from-ide/

[6] Amran Hossain, Dr. Md. Abul Kashem, Sahelee Sultana (Abril 2013). “Enhancing Software Quality Using Agile Techniques”. IOSR Journal of Computer Engineering

[7] IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology 610.12

[8] Organización Internacional de Normalización. Sistemas de gestión de la calidad (ISO 9000). 2015

[9] Duecode (2020) Code Quality Metrics. https://duecode.io/blog/code-quality-metrics/

[10] Sealigths (2018) Code Quality Metrics: Is your code any good? https://www.sealights.io/code-quality/code-quality-metrics-is-your-code-any-good/

[11] Ian Sommerville (2011). Ingeniería de Software, Novena Edición.

[12] Kent Beck (2000). EXTREME PROGRAMMING EXPLAINED: EMBRACE CHANGE.

[13] Beller, M; Bacchelli, A; Zaidman, A; Juergens, E (Mayo 2014). Modern code reviews in open-source projects: which problems do they fix?.

[14] Dan Radigan. “Why code reviews matter (and actually save time!)”. https://www.atlassian.com/agile/software-development/code-reviews

[15] Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. (Jan. 2014) Best Practices for Scientific Computing. Plos Biology

[16] Guru99. (c2021). Phases of Compiler with Example. Guru99. https://www.guru99.com/compiler-design-phases-of-compiler.html

[17] Wikiwikiweb, Wiki. (2014). Abstract Syntax Tree. WikiWikiWeb. https://wiki.c2.com/?AbstractSyntaxTree

[18] Iulian Neamtiuneamtiu, Jeffrey S. Fosterjfoster, Michael Hicks Understanding Source Code Evolution Using Abstract Syntax Tree Matching. https://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=A9FF8C13D19C6A979C9C1AE6984FFD7E?doi=10.1.1.114.3726&rep=rep1&type=pdf

[19] Microsoft. (2020). Review and merge code with pull requests. Azure Repos | Microsoft Docs. https://docs.microsoft.com/en-us/azure/devops/repos/git/pull-requests?view=azure-devops

[20] Github. (c2021) Features - Code Review. Github. https://github.com/features/code-review/

[21] Gitlab. (2021). Introduction to GitLab Flow. GitLab Docs. https://docs.gitlab.com/ee/topics/gitlab_flow.html

[22] Stéphane Ducasse, Pavel Krivanek y Johan Fabry. (1ero de Mayo, 2020). Building user interfaces with Spec 2.0.

Apéndice A

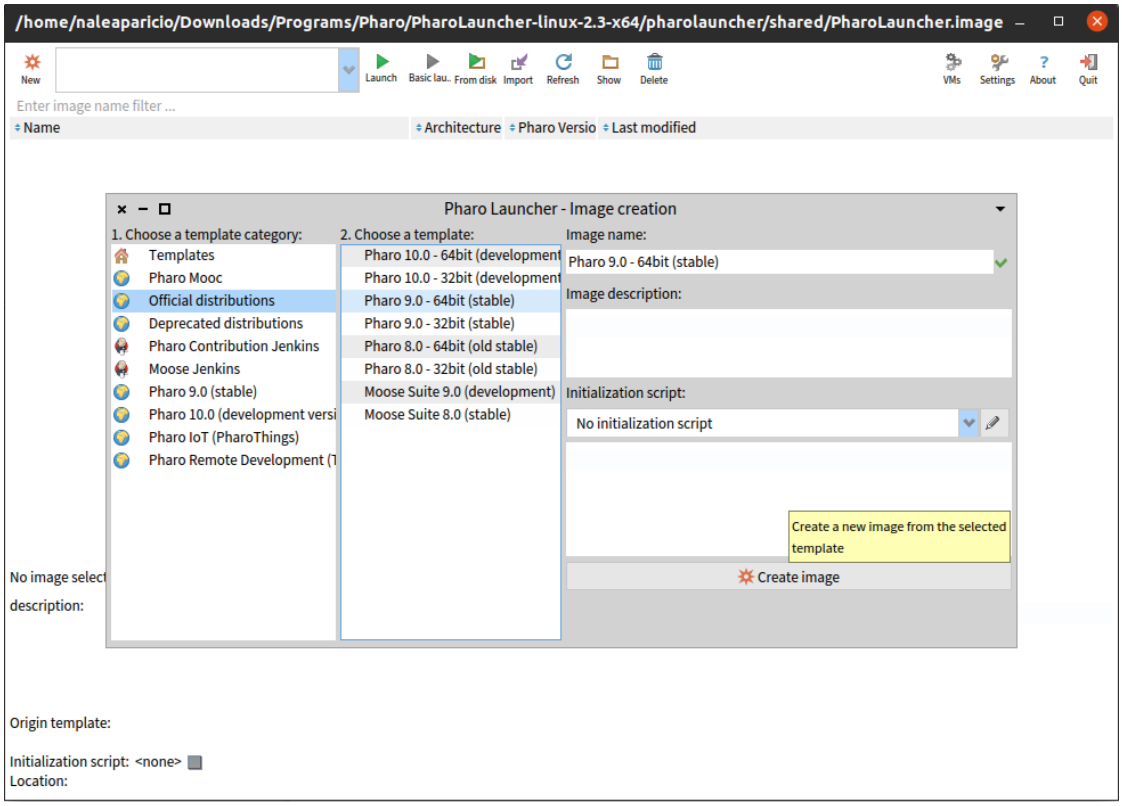
Pruebas de Usuario

Definición de la Prueba de Usuario

Este documento tiene como función facilitar la realización de la prueba de usuario para la tesina *CodeCaption – Una herramienta para realizar Code Review distribuido*. Se divide en unas pequeñas 3 partes con los pasos a seguir.

Parte 1

1. En caso de no tenerlo instalado, instalar Pharo Smalltalk https://pharo.org/download y correr el pharo launcher.
2. Crear una nueva imagen de Pharo 9.0 y abrirla

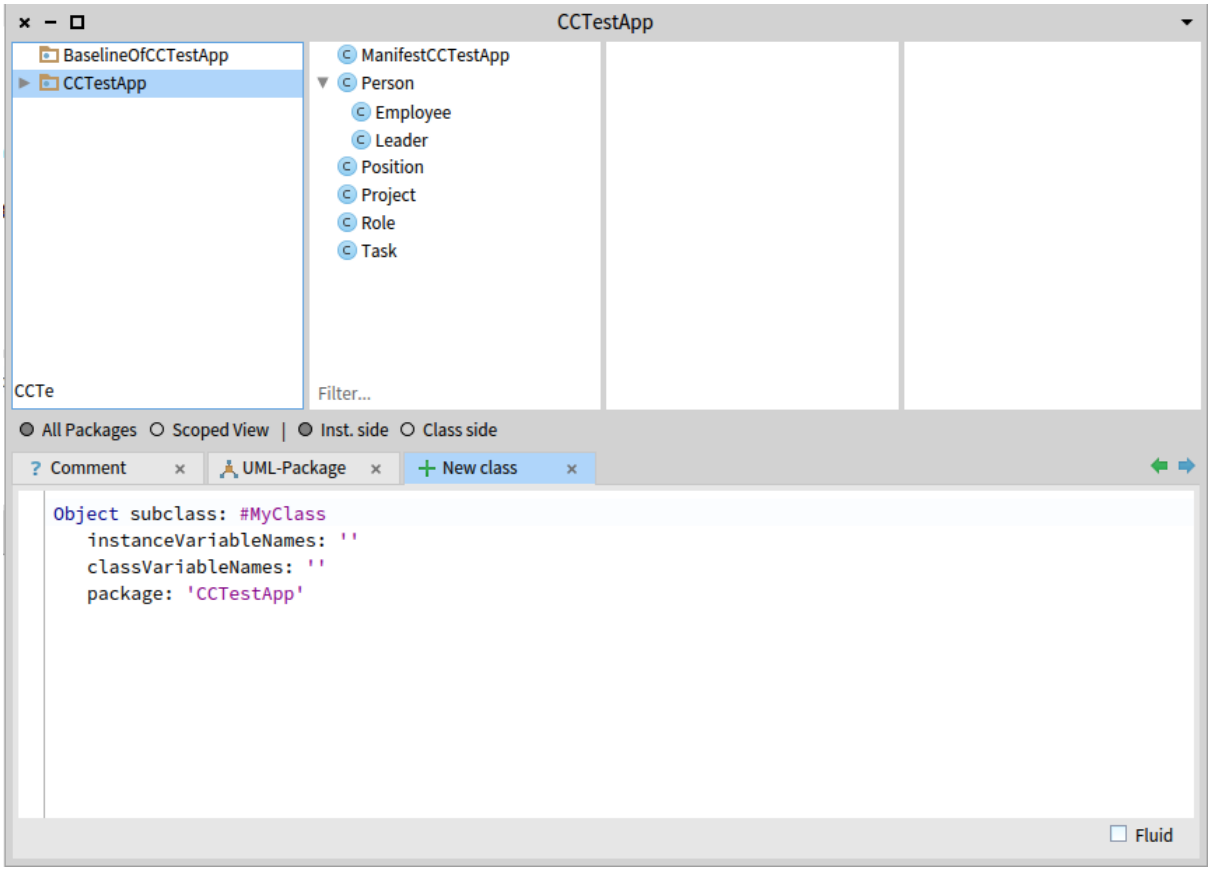


3. Ejecutar el siguiente script en una ventana de Playground del entorno:

```
Metacello new
   baseline: 'CCTestApp';
   repository: 'github://nahux/CCTestApp:demo';
   load.
```

Esto hará que se instale una aplicación de prueba CCTestApp desde github junto con CodeCaption y las dependencias necesarias para poder usar la herramienta.

4. Abrir una ventana del explorador de código del entorno (System Browser) y buscar el paquete CCTestApp de la aplicación de prueba.



5. En el código de la aplicación CCTestApp se encuentran distintos code smells a detectar. Buscar al menos 3 code smells y anótalos debajo en la siguiente sección indicando el lugar y la descripción del mismo.

Code Smells Encontrados:

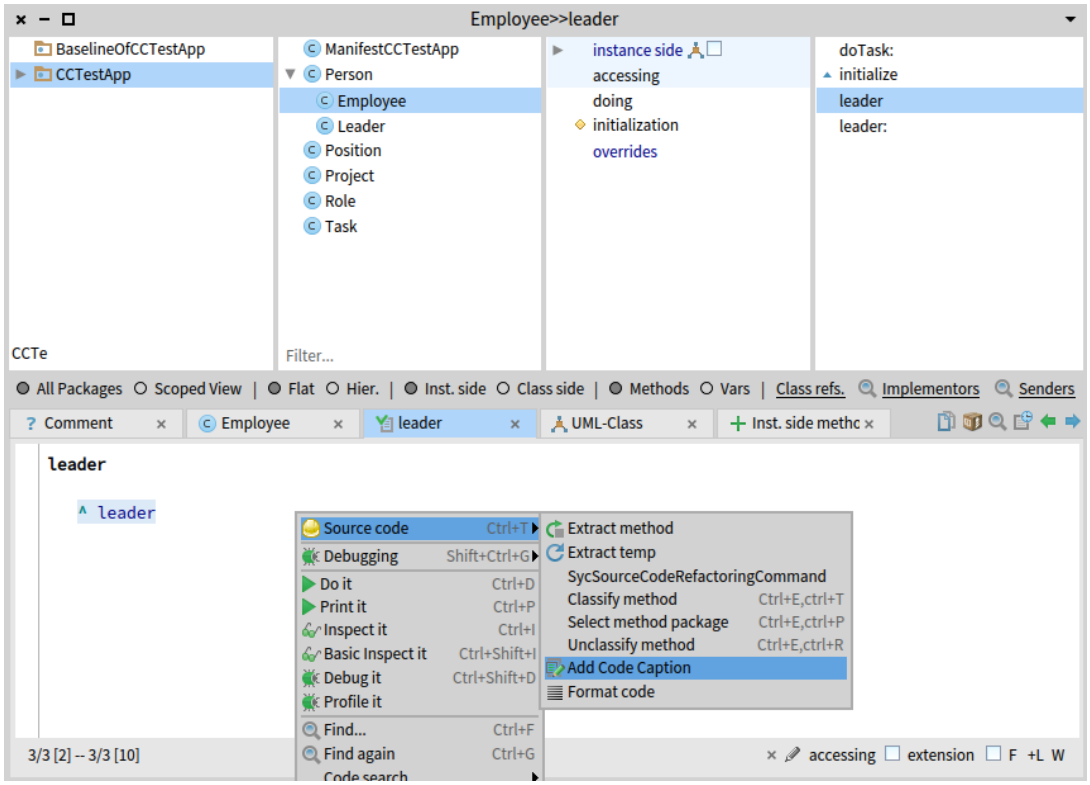
1.

Parte 2

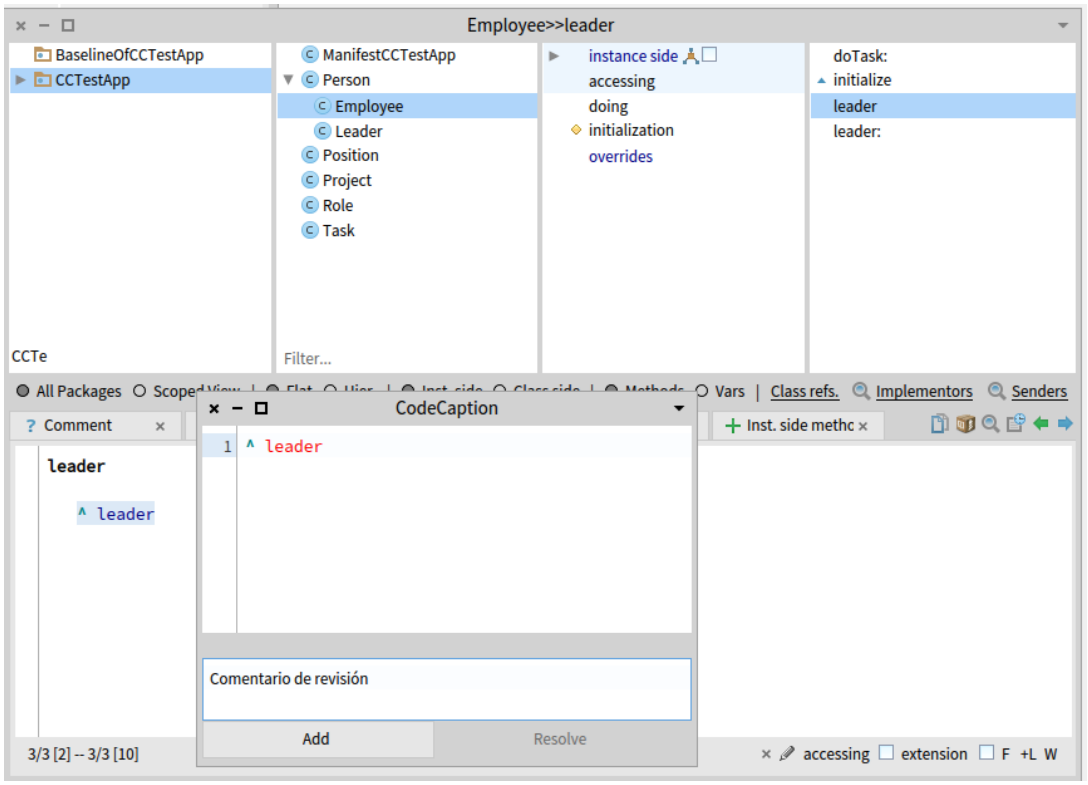
Para esta parte se requiere anotar los code smells encontrados en la parte anterior usando la herramienta CodeCaption dentro de Pharo.

Para agregar una revisión de código CodeCaption seguir los siguientes pasos:

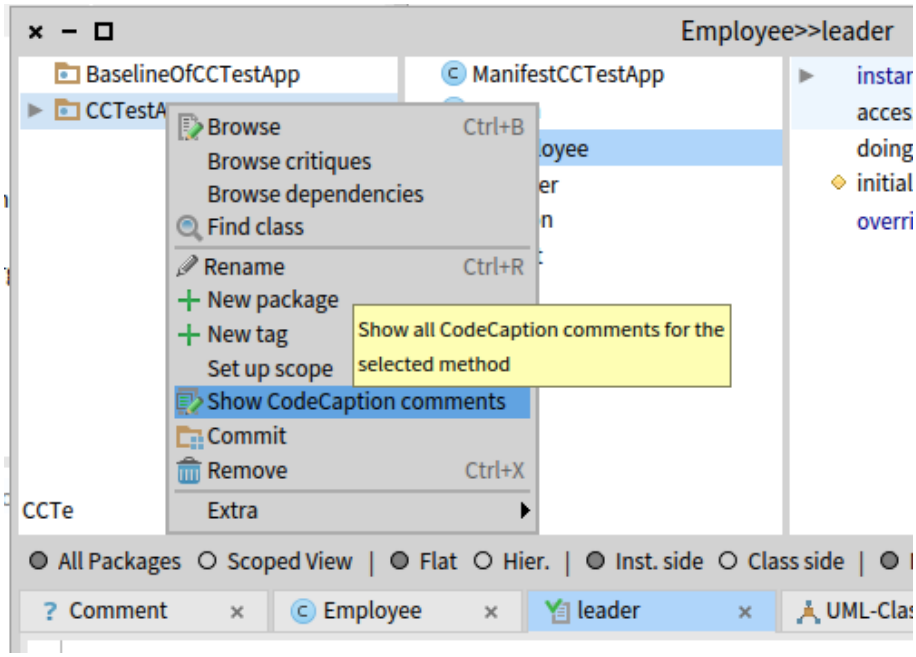
1. Ir a la porción de código con el code smell encontrado y seleccionalo. Luego hay que hacer click derecho e ir al menú “SourceCode”-->>”Add Code Caption”:



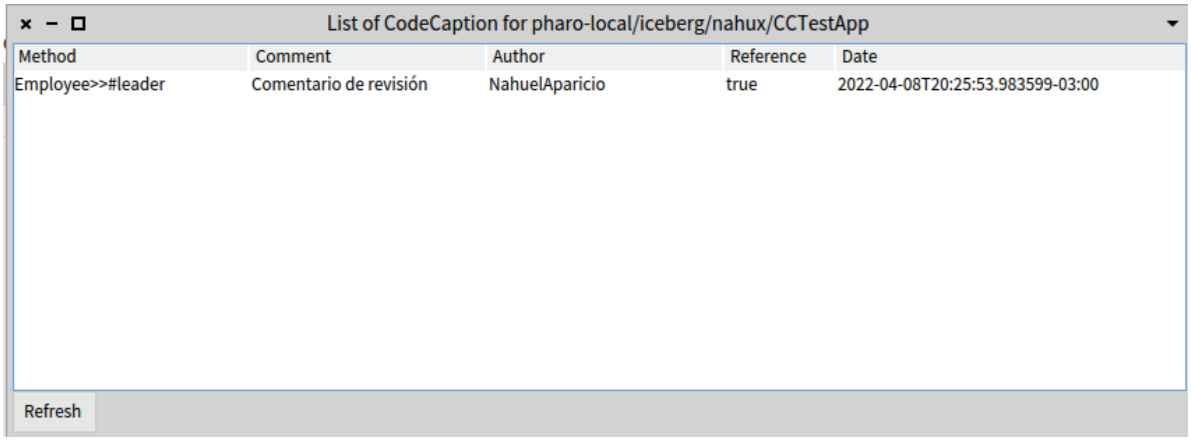
2. Esto abrirá una ventana donde se ve el código seleccionado y un recuadro para agregar un comentario. Aquí agregar un texto explicando el code smell encontrado y clickear en “Add” para guardar la revisión.



3. Una vez creadas las revisiones dentro de la herramienta para cada code smell encontrado, hacer click derecho en el paquete CCTestApp y seleccionar “Show CodeCaption comments” para ver las revisiones creadas:



4. Sacar una captura de pantalla a la tabla con todas las revisiones creadas en la herramienta y pegarla en la sección debajo.



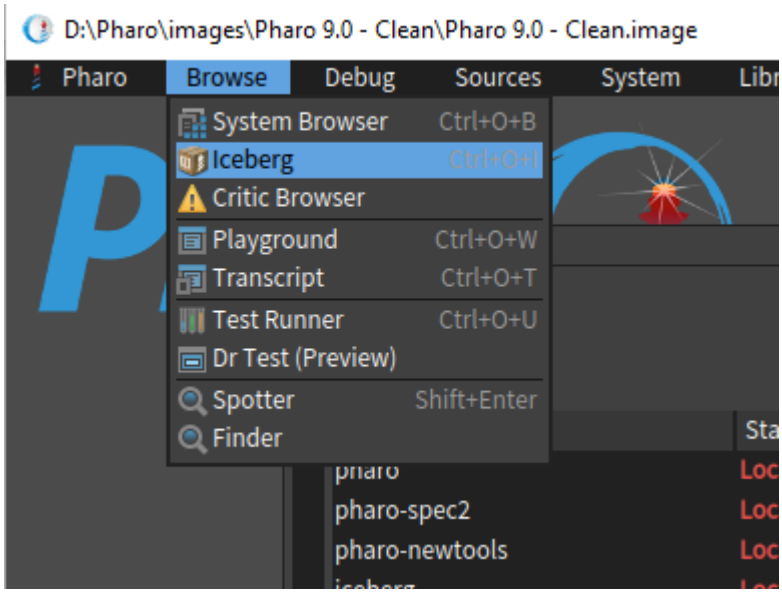
Captura de pantalla con las revisiones creadas:

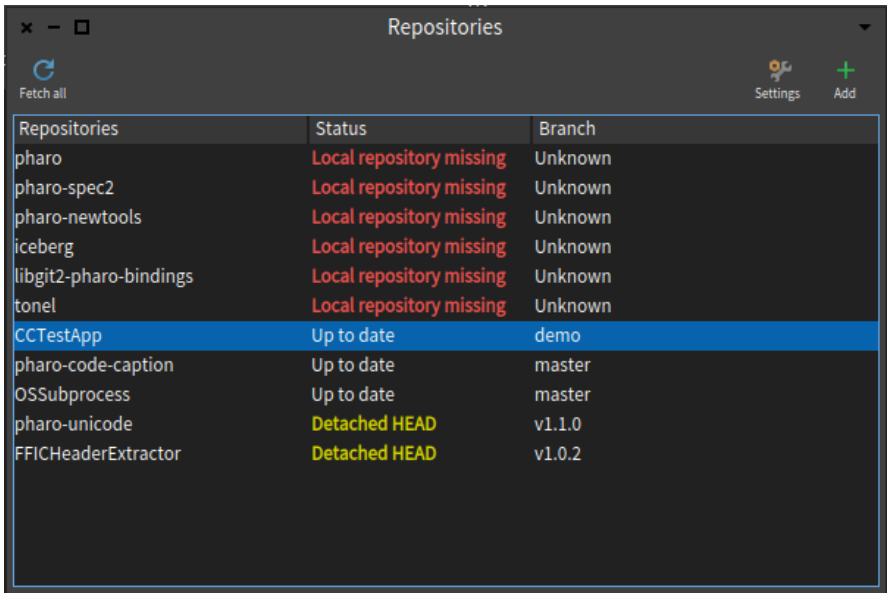
Pegar aquí la captura de pantalla con las revisiones de código creadas para cada code smell.

Parte 3

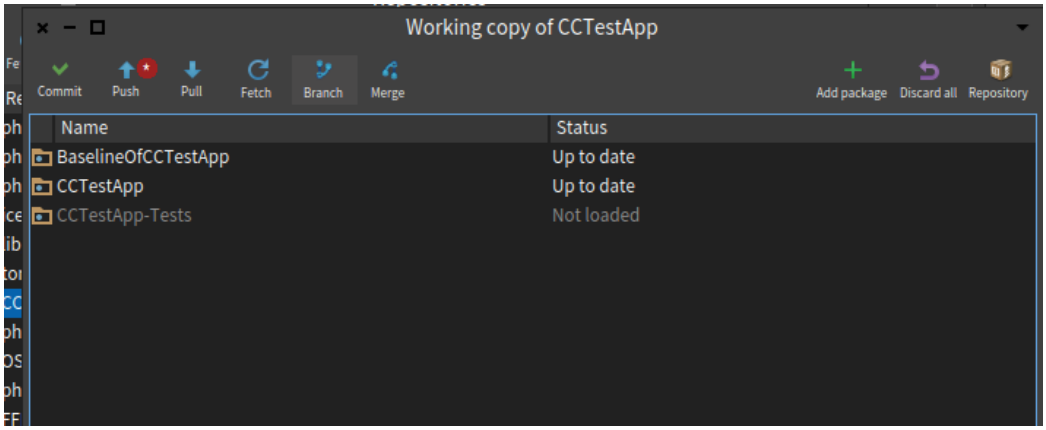
En esta parte se puede ver cómo visualizar las revisiones de código hechas por otras personas, facilitando el trabajo en equipo.
En este ejemplo se debe cambiar a una rama de git de la aplicación de prueba que tiene varias revisiones de código creadas en la herramienta en relación a los code smells presentes.

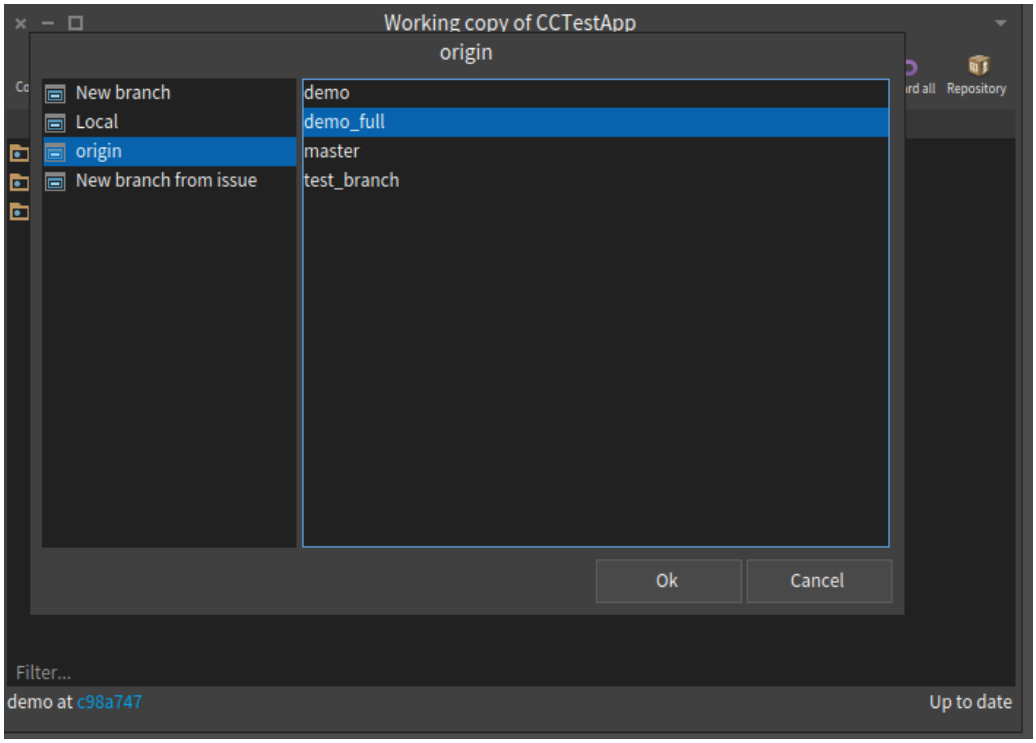
1. Para ver las revisiones de código, se debe cambiar de rama al repositorio del proyecto de la aplicación de prueba. Abrir iceberg y seleccionar el repositorio “CCTestApp”:

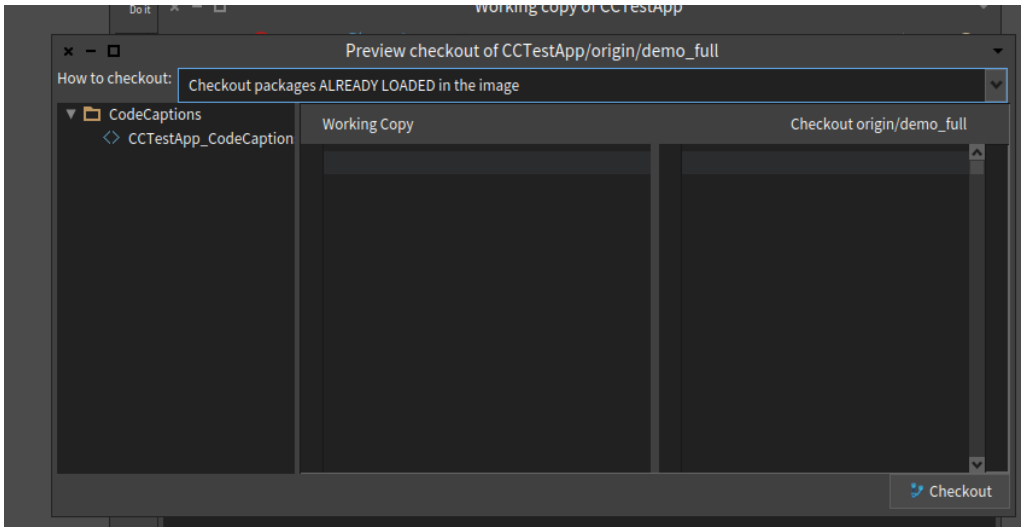




2. Ir a “Branch”, luego a “origin” y seleccionar la rama demo_full. Finalmente tocar en “Checkout”l:







3. De la misma forma que se hizo en la parte 2, se pueden visualizar las revisiones de código presentes en el System Browser al hacer clic derecho en el paquete CCTestApp y seleccionar “Show CodeCaption comments”.

Encuesta

Para terminar con la prueba por favor responder las siguiente pequeña encuesta sobre la experiencia con la herramienta. Por favor escribe una de las opciones de respuesta en caso de tener y agregá algún comentario de ser necesario:

1. ¿Qué tal te pareció la experiencia del uso de la herramienta? ¿Te resultó fácil de usar? *Opciones*: Muy Fácil de Usar - Fácil de Usar - Normal - Difícil de Usar - Muy Difícil de Usar
   Respuesta:

2. ¿Qué tan bien se compara en términos de rapidez y facilidad de uso la realización de code review usando la herramienta con respecto a hacerlo manualmente?
   Opciones: 1 al 10, (siendo 1 mucho peor que Manual, 5 iguales y 10 mucho mejor que Manual)
   Respuesta:

3. ¿Usas una herramienta de este tipo dentro de un IDE para realizar code reviews cotidianamente? ¿Si no usas actualmente, te gustaría usar una herramienta de este tipo? *Opciones*: Me encantaría, Me da Igual, No usaría.
   Respuesta:

4. ¿Qué te gustaría mejorar o añadir a la herramienta?
   Respuesta:

Prueba de Usuario Manuel D'argenio

Parte 1

Code Smells Encontrados:

1. Reinventar la rueda

```
doTask: aTask

   | isMyTask |
   isMyTask := false.
   aTask employees do: [ :e | e == self ifTrue: [ isMyTask := true ] ].
   isMyTask
      ifTrue: [
         aTask completed: true.
         ^ true ]
      ifFalse: [ ^ false ]
```

2. Envidia de atributos

```
assignTask: aTask toEmployee: anEmployee

   anEmployee class name = 'Leader' ifTrue: [ ^ false ].
   aTask employees add: anEmployee
```

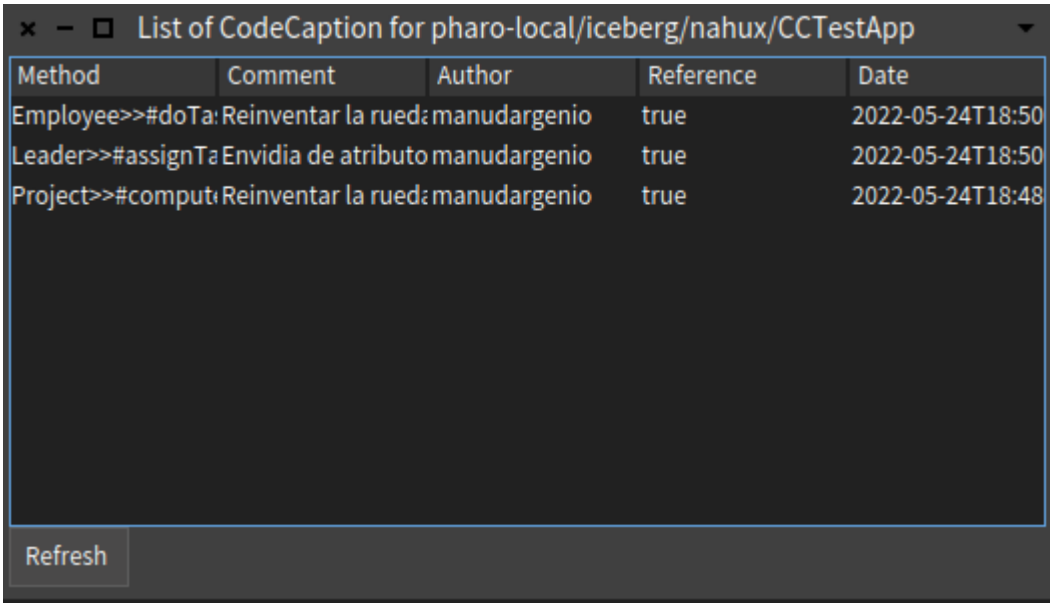
3. Reinventar la rueda

```
computeTheTotalCompletionPercentageOfTheProjectTasks
   | completedTasks |
   completedTasks:= OrderedCollection new.
   tasks do: [ :t | t completed ifTrue:[ completedTasks add: t ]].

    completionPercentage:= (((completedTasks size / tasks size) asFloat) * 100) round: 2
```

Parte 2

Captura de pantalla con las revisiones creadas:



Encuesta

1. **Respuesta:**

   Buena, me pareció fácil de utilizar.

2. **Respuesta:**

   6

3. **Respuesta:**

   No uso estas herramientas cotidianamente. Me da igual

4. **Respuesta:**

   Que te redirija al método para visualizar dónde está la línea de código.

Prueba de Usuario Sebastián Gallardo

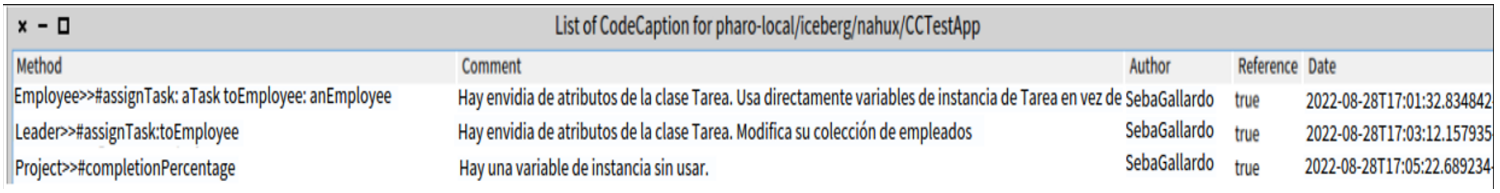
Parte 1

Code Smells Encontrados:

1. En la clase Employee, método assignTask: aTask toEmployee: anEmployee hay envidia de atributos de la clase Tarea. Usa directamente variables de instancia de Tarea en vez de delegar.
2. En la clase Leader, método metodo assignTask:toEmployee. Hay envidia de atributos de la clase Tarea. Modifica su colección de empleados
3. En la clase Project, método completionPercentage hay variable de instancia sin usar.

Parte 2

Captura de pantalla con las revisiones creadas:



List of CodeCaption for pharo-local/iceberg/nahux/CCTestApp

Method	Comment	Author	Reference	Date
Employee>>#assignTask: aTask toEmployee: anEmployee	Hay envidia de atributos de la clase Tarea. Usa directamente variables de instancia de Tarea en vez de	SebaGallardo	true	2022-08-28T17:01:32.834842
Leader>>#assignTask:toEmployee	Hay envidia de atributos de la clase Tarea. Modifica su colección de empleados	SebaGallardo	true	2022-08-28T17:03:12.157935
Project>>#completionPercentage	Hay una variable de instancia sin usar.	SebaGallardo	true	2022-08-28T17:05:22.689234

Encuesta

1. **Respuesta:**

   Fácil de usar en general

2. **Respuesta:**

   6. Esta bien pero la diferencia no es significativa

3. **Respuesta:**

   No uso pero me encantaría

4. **Respuesta:**

   Que se resalte el código que tiene un comentario

Prueba de Usuario Roberto Molina

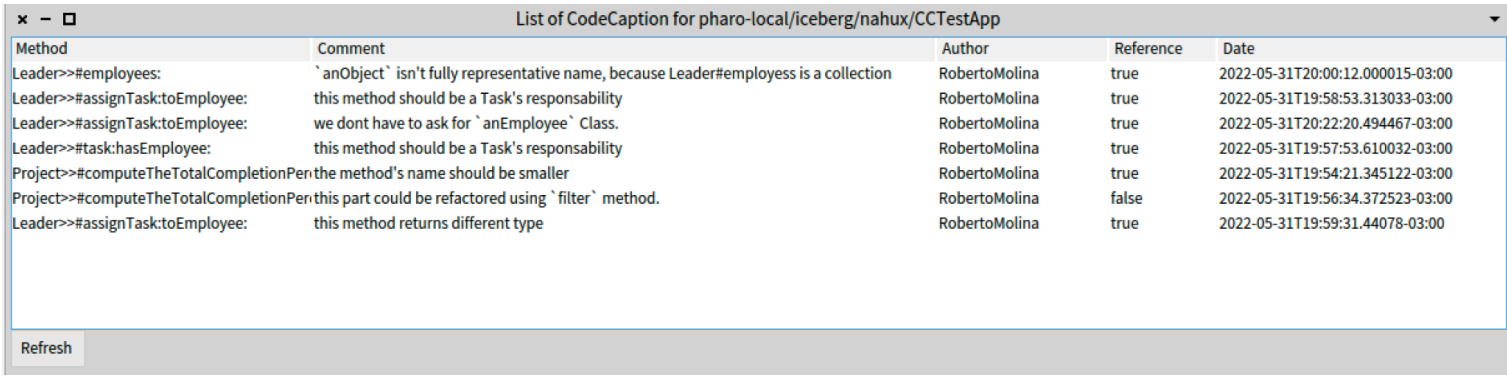
Parte 1

Code Smells Encontrados:

1. El metodo Leader#task:hasEmployee, deberia ser una responsabilidad de la clase Task.
2. el metodo Leader#assignTask:toEmployee, deberia ser una responsabilidad de la clase Task.
3. el metodo Leader#assignTask:toEmployee, no deberia preguntar por la clase de `anEmployee`.
4. el metodo Leader#assignTask:toEmployee, no siempre devuelve el mismo type.
5. el metodo Leader#employees, tiene como parametro `anObject` lo cual no es representativo, ya que la inicializacion de este atributo es una collection.
6. Project#computeTheTotalCompletionPercentageOfTheProjectTasks tiene un nombre muy largo.
7. Project#computeTheTotalCompletionPercentageOfTheProjectTasks puede usar el metodo filter de Collection para computar/filtrar las tareas terminadas. Tambien podria ser un reduce.

Parte 2

Captura de pantalla con las revisiones creadas:



Encuesta

1. **Respuesta:**

   Fácil de usar. La herramienta me resultó fácil de usar. Lo único que me generó duda fue en el momento de seleccionar el código y lo que se veía en la ventana que abría que no siempre era lo seleccionado, generalmente era más del código seleccionado.

2. **Respuesta:**

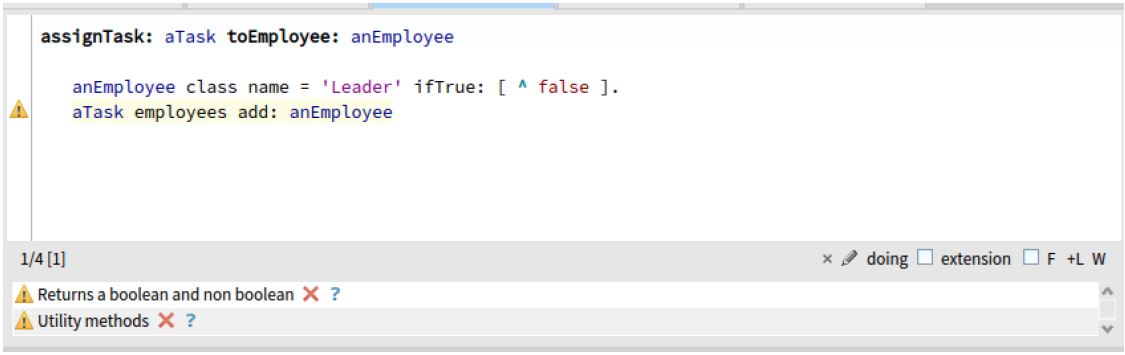
8. Está bueno la posibilidad de realizar los code review desde el IDE/Workspace debido a que está todo en el mismo lugar, no es necesario usar otra herramienta o abrir el código desde otra plataforma.

3. **Respuesta:**

Me encantaría. Si, cuando tengo la posibilidad trato de usar las herramienta de code review de VSCode.

4. **Respuesta:**

Al igual que la sección de warning, estaría bueno tener los code review listados en cada definición.



Prueba de Usuario Tomás Biasotti

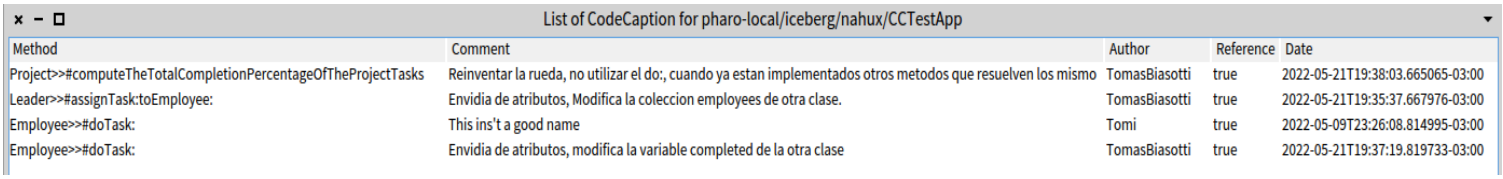
Parte 1

Code Smells Encontrados:

1. Clase Leader, envidia de atributos en método assignTask:toEmployee:, Modifica la colección employees de otra clase.
2. Clase Employee, envidia de atributos en método doTask:, Modifica la variable completed de otra clase.
3. Clase Project, reinventar la rueda en metodo computeTheTotalCompletionPercentageOfTheProjectTask, usa el do:

Parte 2

Captura de pantalla con las revisiones creadas:



List of CodeCaption for pharo-local/iceberg/nahux/CCTestApp

Method	Comment	Author	Reference	Date
Project>>#computeTheTotalCompletionPercentageOfTheProjectTasks	Reinventar la rueda, no utilizar el do:, cuando ya estan implementados otros metodos que resuelven los mismo	TomasBiasotti	true	2022-05-21T19:38:03.665065-03:00
Leader>>#assignTask:toEmployee:	Envidia de atributos, Modifica la coleccion employees de otra clase.	TomasBiasotti	true	2022-05-21T19:35:37.667976-03:00
Employee>>#doTask:	This ins't a good name	Tomi	true	2022-05-09T23:26:08.814995-03:00
Employee>>#doTask:	Envidia de atributos, modifica la variable completed de la otra clase	TomasBiasotti	true	2022-05-21T19:37:19.819733-03:00

Encuesta

1. **Respuesta:**

   Fue facil de utilizar y facil de ver los comentarios del resto de personas. Me gustaría probar con mucho mas comentarios si es facil buscar un comentario especifico.

2. **Respuesta:**

   8, creo que es mejor que manualmente. Aunque quizás necesita alguna que otra funcionalidad para ser perfecto

3. **Respuesta:**

   Totalmente usaria algo asi, hoy en dia dejamos comentarios como este en los Pull Request dentro de la plataforma de GitHub, pero estaria bueno poder verlos dentro del IDE que estemos usando

4. **Respuesta:**

Algo ya comente arriba, pero creo que estaría bueno que se vea más claramente donde ya hay un comentario en el código, y quizás un buscador entre todos los mensajes dentro del repositorio.

Prueba de Usuario Yanina Echevarria

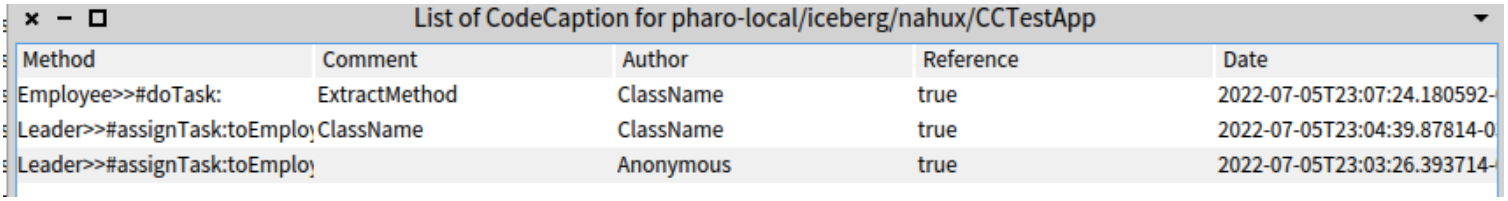
Parte 1

Code Smells Encontrados:

1. En la Leader en el método assignTask: aTask toEmployee: anEmployee pregunta por el nombre de la clase.
2. En el método doTask: aTask, debería haber otro método que chequee si la tarea era propia del empleado o no.
3. Podría haber un método que devuelva únicamente las tareas completadas y no como es el caso de un método que busca las completadas y calcula el promedio.

Parte 2

Captura de pantalla con las revisiones creadas:



Encuesta

1. **Respuesta:**

   Muy Fácil de Usar

2. **Respuesta:**

   10

3. **Respuesta:**

   Me encantaría

4. **Respuesta:**

   Que la visualización se realice en el recuadro donde se escribe el código y no en una ventana aparte.