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

SOPORTE PARA EL DESARROLLO INCREMENTAL EN ECLIPSE: UNA IMPLEMENTACIÓN ROBUSTA DE GHOSTS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

RICARDO MANUEL JACAS FRANZOY

PROFESOR GUÍA:
ÉRIC TANTER

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
MARCELA CALDERON CORAIL

SANTIAGO DE CHILE
2013

Resumen

Los ambientes de desarrollo, o IDE por su sigla en inglés, representan una ayuda fundamental para el programador. La mayoría de estos ambientes suelen integrar un sin fin de herramientas para facilitar el trabajo, como son el reporte estático de errores y los asistentes de autocompletado de código. Dentro de los ambientes de desarrollo disponibles para Java, Eclipse destaca por ser uno de los ambientes más completos. Eclipse, al igual que muchos entornos de desarrollo, cuenta con alertas de error en tiempo de compilación, alertas que permiten que el programador no incurra en errores por omisión de código.

En la actualidad es frecuente el uso de metodologías ágiles para el desarrollo de aplicaciones. Dentro de estas metodologías destacan las técnicas orientadas al buen diseño, como son el Desarrollo Guiado por Pruebas (o TDD) y la programación *Top-Down*, técnicas en las cuales se definen las interfaces mediante las que se utiliza el código de manera previa a la implementación real del funcionamiento. Estas técnicas permiten reducir la tasa de error del código generado al estar enfocado primero en el objetivo (¿Qué funcionalidad provee?,¿Cómo se usa?) y luego en la implementación (¿Cómo realizar dicha funcionalidad?).

Desgraciadamente, la programación *Top-Down* y TDD no están soportadas de manera adecuada en los IDE cotidianos, como Eclipse, puesto que el reporte de errores estático suele interferir fuertemente con la estructuración de código previa a su implementación, reportando al usuario muchos errores de los cuales está perfectamente consciente y que no aportan, errores del estilo “dicha entidad no existe”. Estos errores además oscurecen o deshabilitan por completo el reporte de otros errores que sí son de interés para el programador, como lo es el uso inconsistente de una interfaz.

Durante el transcurso de esta memoria se continuo el desarrollo de un *plug-in* para Eclipse, llamado Ghosts, que se encarga de atacar esta problemática, entregando un ambiente preparado para el uso de técnicas que requieren pensar primero en la interfaz y luego en la implementación. Este *plug-in* está basado en la noción de detectar aquellas entidades no definidas en el código y ordenar correctamente la información contextual que se puede recabar de ellas, utilizando únicamente el código provisto por el programador. Además de su desarrollo, el presente trabajo concluye con una versión que soluciona parcialmente el problema, dejando de lado exclusivamente las características más modernas del lenguaje Java, y se presentan las posibles alternativas de extensión y sus ventajas comparativas de acuerdo a los resultados obtenidos.

Agradecimientos

En primer lugar, quiero agradecer al docente Éric Tanter, por entregarme la oportunidad de trabajar en este proyecto y ayudarme a llegar a buen término.

De la misma forma, quiero agradecer a Oscar Callaú, por su apoyo en el desarrollo mismo del trabajo y su buena disposición a la hora de necesitar ayuda.

Asimismo, quiero agradecer a mis amigos Gustavo Soto, Leonardo Rojas, Jorge Bahamonde y Javiera Born, por su ánimo y su ayuda durante el transcurso de esta memoria.

Por último, quiero agradecer a mis padres, por acompañarme en este proceso y apoyarme cuando ha sido necesario. Sin su apoyo no estaría aquí.

Gracias a todos.

Ricardo Manuel Jacas Franzoy

Tabla de contenido

Capítulo 1

Introducción

En la actualidad, con el avance e impacto del software en todas las áreas de la industria, su desarrollo ha adquirido prácticas que aseguran que la fiabilidad y calidad del código generado se mantengan durante el ciclo de vida de las aplicaciones, sin que el costo de cambiar el código crezca de manera incontrolable a medida que crece la aplicación. Una práctica que va adquiriendo cada vez más notoriedad es el Desarrollo Guiado por Pruebas[1], en adelante TDD, la cual ha promovido la costumbre de desarrollar casos de prueba antes de implementar el sistema que va a satisfacerlos [1]. Bajo esta lógica, programar se reduce a completar el sistema de forma tal que todas las pruebas pasen correctamente. Similar es el caso de la programación *Top-Down* (Arriba-Abajo), en la cual cuando se programa un procedimiento para resolver un problema específico, se suele depender de procedimientos auxiliares, más pequeños, que podrían no estar implementados aún. Tanto en la programación *Top-Down*, como en TDD, programar es una actividad incremental, ya que los programadores escriben código que utiliza entidades que aún no han sido definidas, o cuya definición es aún parcial.

Pese a la popularidad de los métodos antes mencionados [2], los Ambientes de Desarrollo Integrado[2](IDE) modernos son, en su mayoría, incapaces de soportar estilos de programación puramente incrementales, dando como única retroalimentación errores sucesivos que interfieren en el desarrollo limpio del código, al solicitar, constantemente, estructuras aún no definidas. A pesar de algunos esfuerzos por realizar *plug-ins* que aborden esta problemática, dada la relevancia de la programación incremental, pocos resultados son capaces de ofrecer una experiencia no obstructiva y que ayude al programador.

Recientemente, Callaú y Tanter proponen una idea muy simple para atacar dicho problema: en vez de sólo reportar mensajes de error, se propone que los IDEs generen, de manera transparente y no intrusiva, una materialización, de acuerdo a su uso, de las entidades no definidas, refinada progresivamente a medida que se elabora el programa. De esta forma, el desarrollador puede

[1]TDD, por sus siglas en inglés (*Test Driven Development*).
[2]del inglés *Integrated Development Environment* o IDEs.

concentrarse en la tarea en cuestión, mientras razona acerca de entidades no definidas, pero con una retroalimentación útil basada en las restricciones y dependencias deducibles del uso. Una vez que el programador está listo para implementar dichas entidades, el IDE puede proveer un esqueleto de código apropiado que calza con todas las dependencias inferidas. A estas entidades no definidas se les llama Ghosts [3].

El trabajo presentado por Callaú y Tanter concluye en una versión rudimentaria de Ghosts, que tiene como objetivo presentar la idea y demostrar, a grandes rasgos, la forma en que soluciona el problema. A continuación, se presenta una versión acabada de Ghosts, que pretende ser un entorno integrado con Eclipse y orientado a solucionar los problemas con los que día a día deben lidiar los programadores, mejorando así su desempeño y facilitando su inmersión en el problema a tratar.

1.1 Motivación

Como se mencionó anteriormente, en la actualidad, si un programa utiliza una entidad que no ha sido definida (un procedimiento, un método, una clase, una interfaz, etc.) la única retroalimentación que el IDE provee es un error. Considerando que los acercamientos incrementales son parte de la práctica diaria del desarrollo de software – en una encuesta Web, con cerca de 300 participantes, más de la mitad reportó usar TDD [4] - y que los acercamientos al desarrollo de software iterativos y ágiles se han vuelto muy populares [2], se podría esperar que los IDEs soporten apropiadamente sus estilos de programación. Así, aunque normalmente se cuenta con la oportunidad de que el IDE cree automáticamente un esqueleto de código que se ajuste a las necesidades, en los casos analizados, esta cualidad resulta tediosa, obstructiva y limitada [3].

La introducción del concepto de Ghosts a los IDEs permite complementar la retroalimentación recibida al programar, lo que es muy beneficioso para el programador. Ello, pues no sólo permite mantener en espera la declaración de toda la estructura necesaria del código, como lo hacen algunos IDEs actuales [5], sino que efectivamente mejora la experiencia de programación, incrementando el valor del entorno de desarrollo, como asistente en la implementación.

Cabe mencionar que, si bien el concepto de Ghosts es muy noble, es necesario mostrar hasta qué punto puede afectar el desempeño al momento de programar. En un entorno libre de esta noción, es frecuente verse enfrentado a cambios de contexto: cuando se trabaja con una sola entidad no definida, Eclipse despliega el asistente de creación, no sólo para crear la clase, sino también una vez por cada método [3]. Por ello, es fundamental incorporar a Eclipse (el IDE más utilizado para desarrollo en Java [5]) la noción de Ghosts para Java, fomentando de esta forma el concepto de Ghosts y el uso de herramientas de desarrollo orientadas a la programación verdaderamente incremental.

1.2 Objetivos

El objetivo de esta memoria es proveer un ambiente de desarrollo incremental robusto, integrado con el entorno y las funcionalidades provistas por Eclipse, el cual permita razonar sobre entidades aún no definidas y produzca una retroalimentación adecuada y acorde al uso que se le dé a dichas entidades. Ello mediante la aplicación del concepto de Ghosts.

1.2.1 Objetivos Específicos

Reorganizar y mejorar la implementación.

La implementación original de Ghosts soporta varios tipos de mecanismos: Ghosts como clases, Ghosts como métodos y Ghosts como campos. Estos mecanismo son soportados, de manera genérica, en varios contextos, como son: dentro de la declaración de una clase, dentro de la declaración de un método, dentro de un ciclo *for*, dentro de una instrucción *if*, etc.

Lamentablemente, este soporte inicial ignora un gran número de contextos en que estas definiciones pueden estar presentes y, por lo tanto, una serie de casos en que el comportamiento y definición de los Ghosts debe variar. Por ello, el primer paso de esta memoria es corregir dicha versión inicial agregando soporte para la mayor cantidad de estructuras posibles, todas aquellas correspondientes a las instrucciones básicas de Java.

Completar el trabajo realizado por Callaú y Tanter.

La implementación original de Ghosts ignora una serie de mecanismos al recorrer el código en su búsqueda por entidades no definidas. Para poder utilizar el *plug-in* de Ghosts en desarrollos reales, es necesario completar los casos aún faltantes dentro de dicho *plug-in*. Al pasar por alto la inferencia de dichas entidades se limita la interacción del *plug-in*, lo que se traduce en una generalización no útil al inferir o derechamente en un error por entidad no definida. Para poder lograr esta generalización, es necesario, en primer lugar, realizar un rediseño del sistema de inferencia. Este rediseño se enfoca en la eliminación de comportamientos ad-hoc y la adición de las reglas necesarias para manejar todos los casos de interés.

Los mecanismos que se agregan al soporte del *plug-in* son:

- Excepciones de Java [6]

  En Java, las excepciones tienen un tipo de objeto, pertenecen a clases específicas y su sistema de jerarquía es, a menudo, utilizado para diferenciar los casos erróneos o inesperados en un programa mediante los distintos tipos de excepciones. Es esperable que dentro de cualquier desarrollo completo exista alguna excepción, la cual debería ser específica al código y, por lo tanto, nueva.

- Llamadas en cadena

  En un sistema basado en objetos, es normal que algunos objetos contengan a otros como campo (ej: `a.foo`) o como resultado de sus funciones (ej: `a.foo()`). En estos casos,

es natural que sea necesario solicitar algo de dichos objetos, los que, para este contexto, son miembros de otro objeto (ej: `a.foo.bar`, `a.foo().bar()`, `a.foo.bar()`, etc.).

- Asociación en expresiones

  Cuando se definen variables, ya sea locales o como miembros de un objeto, muchas veces se requiere de otras variables, cuyos tipos deben coincidir, o ser sub-clases, del tipo del resultado (ej: `int b = a.a + a.b`) y de las cuales se puede inmediatamente inferir información (`a.a` y `a.b`, en el ejemplo, son numéricos, no necesariamente `int`, pero numéricos).

- Acceso a miembros de la súper-clase

  En todo sistema basado en objetos, existen jerarquías de clases que cuya finalidad es reutilizar funcionalidades comunes a varios objetos. En este contexto, la reutilización, para cada objeto, se realiza llamando a los miembros de la superclase (ej: `super.a`, `super.foo()`), literal o anónimamente. En la implementación original, no está soportado ningún tipo de acceso a los campos de una súper-clase, con excepción de `Object`, la clase raíz de Java [7].

Integrar Ghosts a funcionalidades útiles presentes en Eclipse.

Eclipse cuenta con una serie de funcionalidades útiles que pueden tener gran impacto al integrarse con el concepto de Ghosts, como son:

- Autocompletado de código [8]

  Como la mayoría de los IDE, Eclipse cuenta con autocompletado basado en las bibliotecas que tiene en su sistema. Este autocompletado se nutre también de los proyectos que el usuario importe o cree. Dicha funcionalidad puede nutrirse con la información recabada por Ghosts, permitiendo autocompletar sobre entidades que aún no existen.

- Refactorización [9]

  Del mismo modo, Eclipse cuenta con una serie de herramientas de refactorización que, en algunos casos (renombres, mover variables o métodos arriba/abajo en una jerarquía, convertir variables locales en campos, etc.) son aplicables a Ghosts, por lo que es esperable que puedan ser aplicadas a ellos, desde la ventana de Ghosts [3].

Capítulo 2

Trabajando con entidades no definidas

Cuando se desarrolla un sistema, es común verse enfrentado a la necesidad de abstraer un concepto que se está utilizando como una nueva entidad, la cual en desarrollos en Java cotidianamente corresponde a una nueva clase o interfaz. Cuando esto sucede, es natural que el programador quiera probar una cierta abstracción antes de implementarla, ya sea mediante nuevas pruebas unitarias, o ejemplos simples que le permitan visualizar, de manera concreta, en qué se traduce la idea que éste tiene. Si bien muchos programadores, acostumbrados a trabajar en editores de texto simples o en IDEs, procederán a sencillamente escribir una versión simple de lo que pretende, y luego a completarla poco a poco, esta práctica los distrae del verdadero objetivo de crear la entidad en primer lugar, que es conseguir una abstracción para un problema específico al que se están viendo enfrentados. Otros programadores, como ya se había mencionado, prefieren optar por continuar escribiendo, asumiendo que su abstracción sí existe y que está disponible tal cual está siendo usada. Para los primeros, aquellos que usan un editor de texto simple, no habrá problema, siempre y cuando recuerden que nada de lo que están escribiendo está implementado realmente, pero en cambio para los segundos, aquellos que están utilizando un IDE, y que seguramente lo utilizan para ser más productivos que si sólo trabajaran con un editor de texto, tendrán que enfrentarse a una gran cantidad de alertas de error que el IDE reporta para ayudar al programador a recordar aquello que olvidó hacer. Estas alertas, en este contexto, son molestas y distraen al programador, perdiendo en cierta medida la ventaja de desarrollar en un entorno especializado.

2.1 Tropezando con Eclipse

En el caso particular de Eclipse, éste tiene herramientas mucho más poderosas que sólo la notificación de error. Eclipse cuenta con herramientas que permiten crear aquella entidad que se reconoció como no definida, para luego poder seguir trabajando con ella. El problema de estas herramientas es la forma en que operan. Cuando se está trabajando con una entidad no definida, ver Figura 2.1,

Eclipse reporta el mismo error en todos los sitios donde dicha entidad está siendo usada. Incluso, luego de crear la clase usando la herramienta de Eclipse, nos encontramos con casi la misma cantidad de errores, reportando ahora que la nueva clase no posee esos miembros que están siendo usados, ver Figura 2.2.

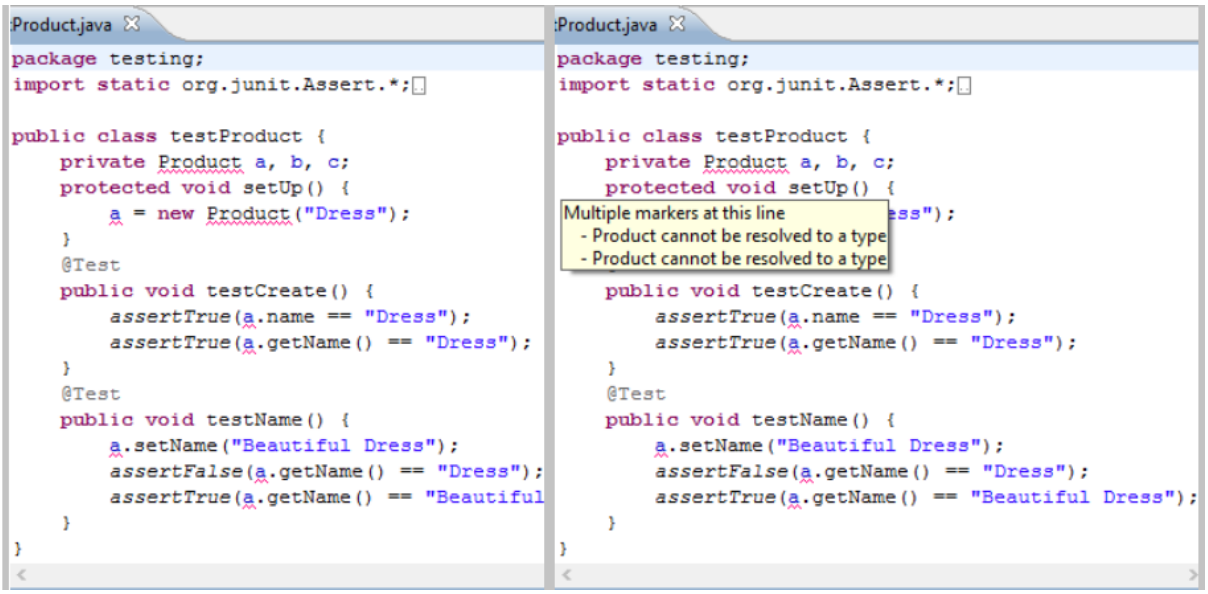

Figura 2.1: La figura muestra un archivo donde está presente una entidad no definida y el mensaje de error que se muestra en todas las líneas donde está esta entidad.

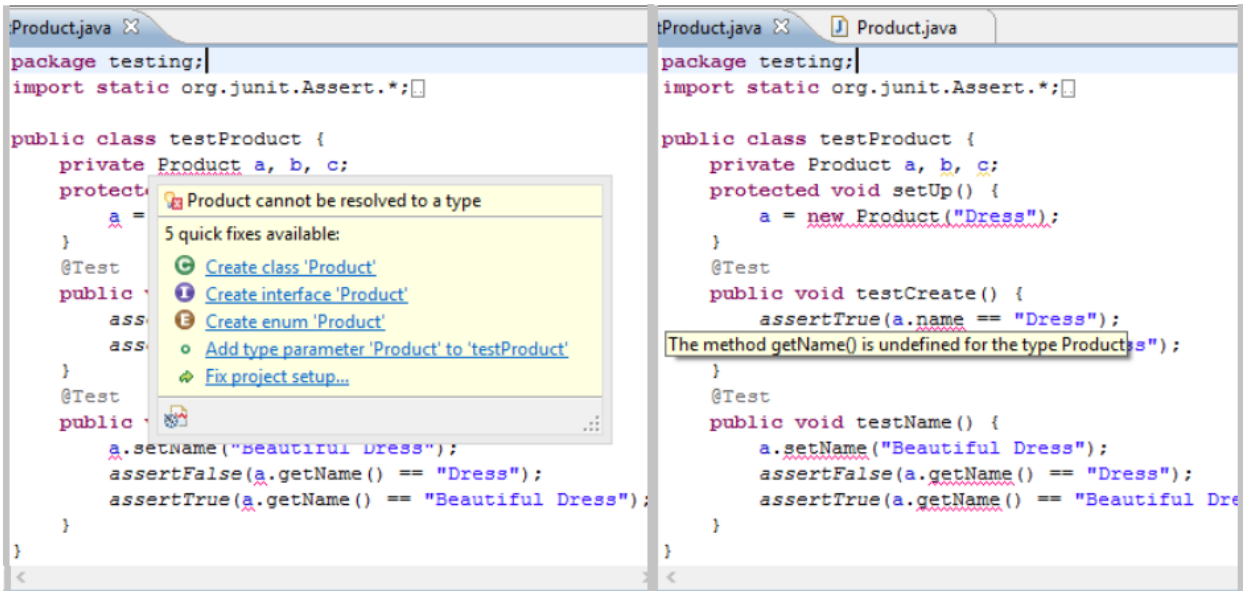

Figura 2.2: Una vez definida la entidad, el error es reemplazado por un conjunto de errores reportando los miembros aún no definidos.

Por si fuera poco, el hecho de que se reporte un conjunto de errores de los que el programador está totalmente consciente, Eclipse fuerza un cambio de contexto, a la nueva clase, cuando ésta se crea y cada vez que desea crear algún miembro mediante la herramienta de refactorización, ver Figura 2.3.

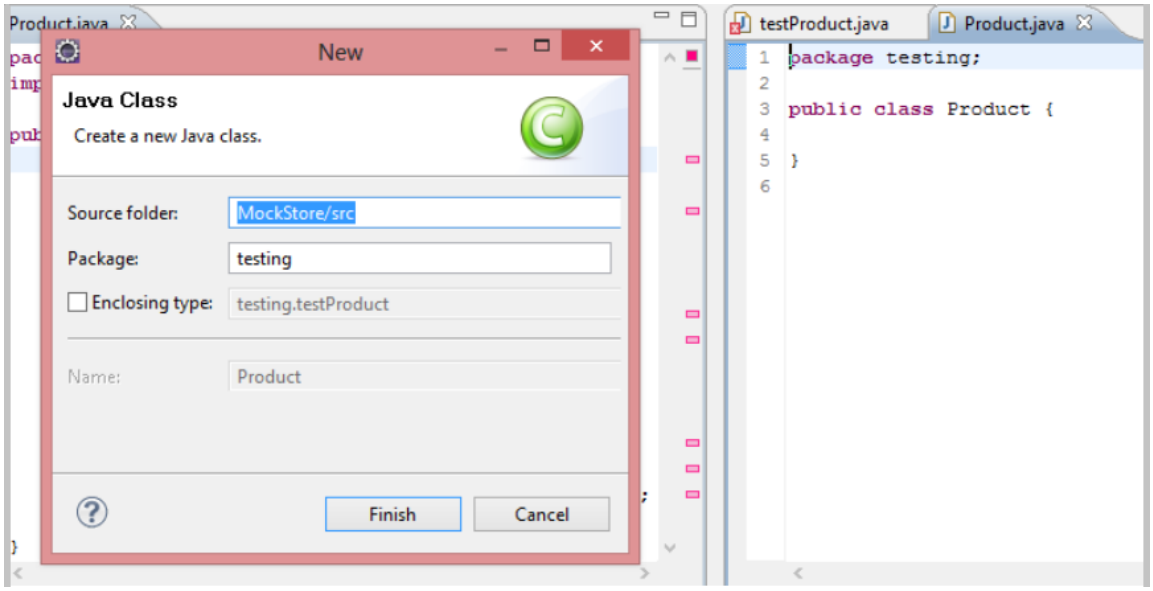

Figura 2.3: Cambio de contexto al crear la clase.

Estos cambios de contexto, sumados al hecho de que el programador debe preocuparse de crear cada miembro que ha agregado en su nuevo código, llevan a muchos a ignorar el uso de la herramienta e ir directamente a la nueva clase a crear los miembros, antes de usarlos. Esta forma de trabajar se hace rápidamente tediosa y redundante, tanto así que más de algún usuario de IDE tenderá a desactivar las alertas de error en esta etapa, o derechamente a preferir interfaces más básicas, como las que proveen editores de texto modernos. Incluso cuando ya se han creado todos los miembros que se pretende tenga la nueva clase, el desarrollo de las pruebas unitarias en esta etapa tiene una alta probabilidad de llevar al programador a modificar varias veces lo que acaba de hacer, puesto que al haber desarrollado primero el código a ser probado, es muy probable que haya olvidado temas importantes de interfaz, como por ejemplo que un mismo método necesite un retorno más general, para ser utilizado en distintos contextos.

El desarrollo previo a la prueba es una mala práctica, genera un código no orientado a la finalidad de la aplicación, sino un código pensado para lograr exclusivamente el calce entre las distintas piezas de código disponibles hasta el momento, perdiendo visibilidad sobre las consecuencias de implementar algo de una manera y no de otra. Sin mencionar que el desarrollar aplicaciones con pruebas malas (o ausentes) dificulta la mantenibilidad del código.

El mismo problema se presenta al intentar desarrollar siguiendo una lógica *top-down*. Al desarrollar cada método pensando en métodos más pequeños, aún no implementados, es innecesario que el IDE insista en marcar aquellos métodos como errores puesto que no lo son. Dichas alertas pueden causar confusión al programador. Sin mencionar que hacer caso a dichas alertas de error puede llevar al programador a saltar continuamente a nuevos contextos, implementando en el momento cada método, distrayéndolo de la funcionalidad que pretendía en el método original y cambiando su tarea original de implementar, en ese momento, un simple método a implementar inmediatamente una parte mucho mayor del proyecto.

Claramente, para un programador experimentado, sucesivos cambios de contexto no afectarán mayormente la calidad de su código, puesto que mientras implementa cada módulo está pensando en el panorama general. Pero mantener un panorama general todo el tiempo, en vez de desarrollar incrementalmente, es una práctica agotadora y poco mantenible en el tiempo.

2.2 Usando Ghosts

Como se mencionó anteriormente, el trabajo con entidades no definidas es necesario para asegurar un código de mayor calidad y orientado a la intención de lo que se está haciendo. Pero, en Eclipse, así como en la mayoría de los IDEs de uso masivo, el soporte para trabajar de esta manera no tiene en consideración el concepto de entidad no definida, sino que está orientado a la solución de problemas más sencillos, como errores de tipeo o el simple olvido por parte del programador de implementar la entidad en primer lugar.

Ghosts está pensando para atacar exactamente ese problema, y proveer al usuario de un entorno orientado a la identificación de errores teniendo en consideración que: las entidades no definidas en realidad sí lo están y que dicha definición es justamente lo que el programador pretende con el código escrito hasta el momento. Esta definición dada por el código actual es lo que se entiende como Ghost [3].

Como parte de su soporte a las entidades no definidas, la perspectiva entregada por Ghosts cuenta con una ventana que concentra las funcionalidades provistas por el *plug-in*. Esta ventana se denomina *GhostView*, ver Figura 2.4, y su principal función es mostrar al usuario los Ghosts que pueden deducirse del contexto actual.

El uso de Ghosts permite ignorar los errores por entidades no definidas y concentrarse en aquellos errores realmente importantes, como son, por ejemplo, las inconsistencias dentro del código que refiere a estas entidades. En la Figura 2.5, se puede ver un ejemplo de código que presenta una inconsistencia y cómo Ghosts entrega las herramientas para identificarla fácilmente.

Adicionalmente, Ghosts entrega al usuario la posibilidad de crear aquellos Ghost con los que está trabajando, mediante la opción *Bust it!*. A diferencia de Eclipse, al usar *Bust it!*, las entidades creadas mediante Ghosts contienen toda la información que el usuario provee mediante su código, no sólo una clase vacía con el nombre del Ghost, sino también nuevos tipos Ghost que se hayan utilizado como miembros del primero, ver Figura 2.6. También soporta Ghosts como miembros de clases que sí existen.

Cabe destacar que la creación de estas clases se hace sin llevar al programador a la clase que se acaba de crear, sino que se le permite seguir trabajando, como si nada hubiese pasado, pero sin tener en cuenta dicha clase como un Ghost.

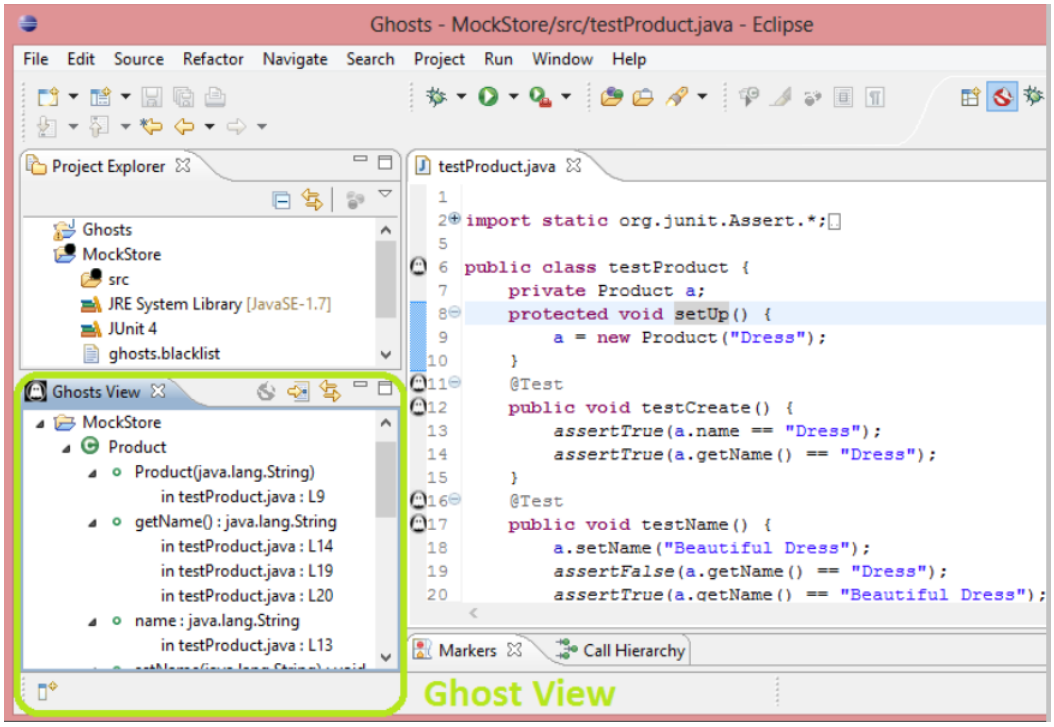

Figura 2.4: *Ghost View* dentro de la perspectiva de Ghosts, en Eclipse.

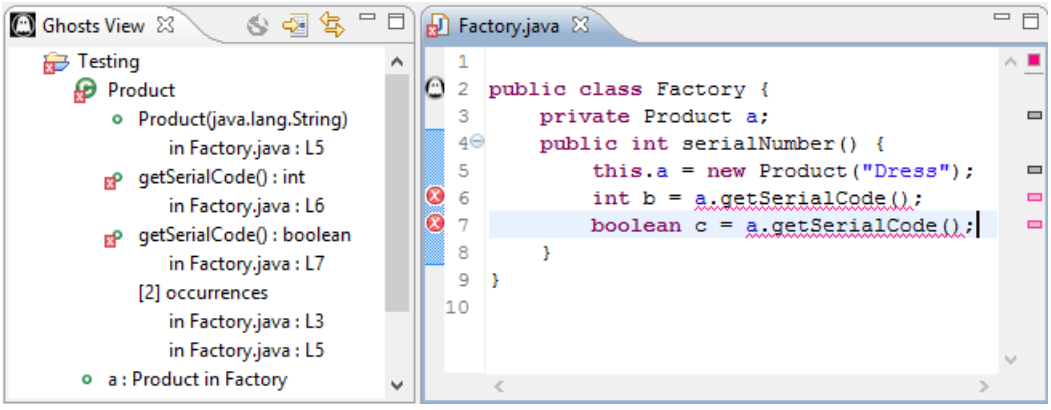

Figura 2.5: Cuando se produce una inconsistencia en el código, ésta se marca como un error y se muestran las dos opciones en la *Ghost View* también como error.

Es de esta forma que Ghosts permite un desarrollo enfocado en la finalidad del código, en vez de en su implementación exacta. Un desarrollo que integra completamente el concepto de entidad no definida y permite, de manera limpia, razonar sobre distintas opciones arquitectónicas sin tener que preocuparse de crear entidades de antemano. Esto posibilita la creación de entidades basadas enteramente en sus pruebas unitarias, rápidamente, sin que el programador tenga que preocuparse sobre todo lo que debe implementar, de una sola vez. Aún más, este enfoque le da la posibilidad, al programador de que las entidades resultantes le entreguen, de manera ordenada, lo que su código significa, y pueda corregirlo sin modificar nada más que el código donde se están utilizando sus entidades.

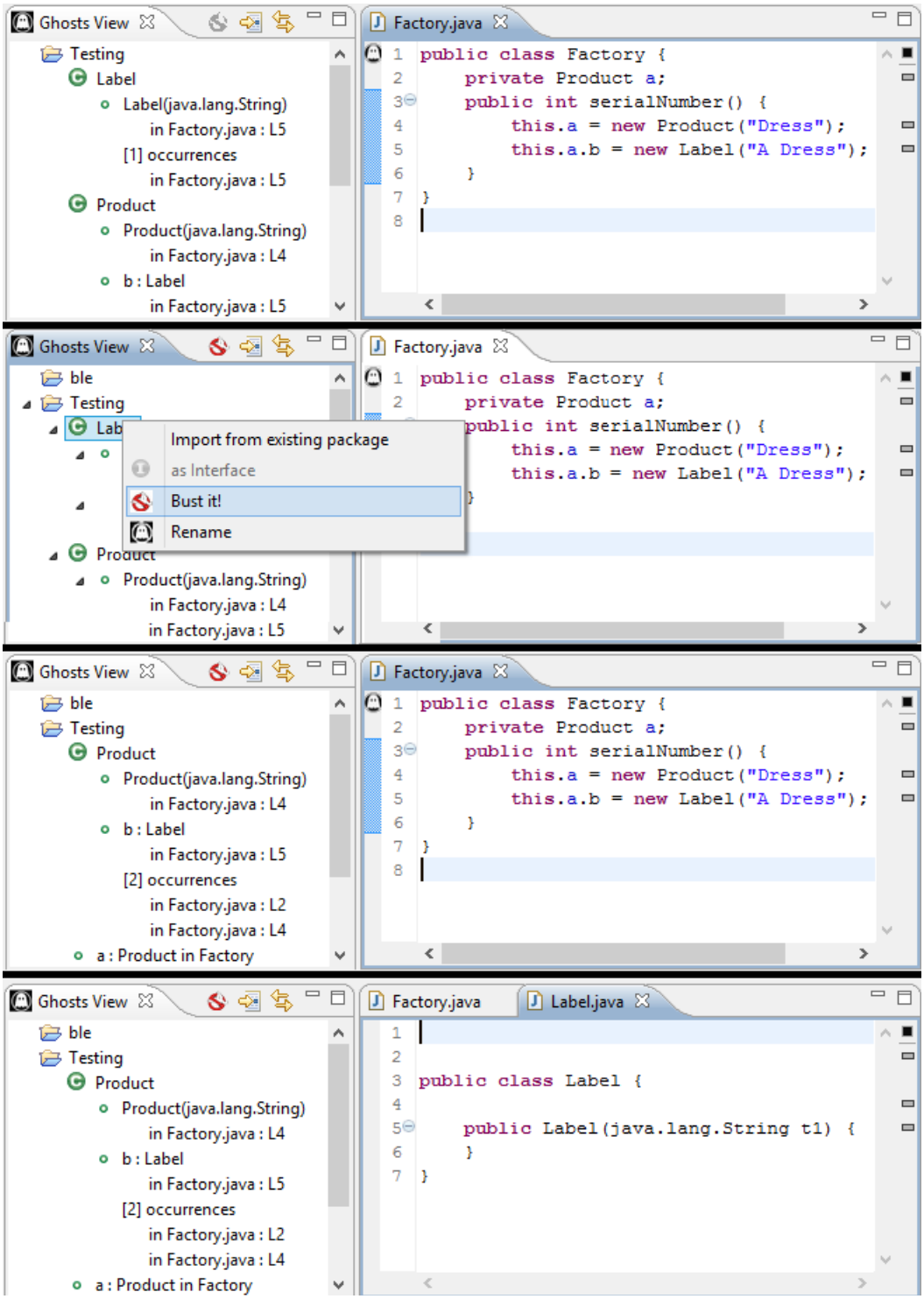

Figura 2.6: Al usar *Bust it!* inmediatamente desaparece Label de la *Ghost View*. Sin distraer al usuario, la clase Label se crea con todos sus miembros declarados en el código actual.

2.3 Estado inicial

La implementación inicial de Ghosts contiene todas las estructuras básicas mostradas en las secciones anteriores. Sin embargo, esta implementación soportaba exclusivamente los siguientes mecanismos:

Bust it!
La funcionalidad necesaria para hacer *Bust it!* a una clase, una interfáz, un método o un campo (en varios contextos) está presente en esta versión.

Ghost View
La vista, tal cual se muestra en la Perspectiva de Ghosts, también está disponible en esta versión. Y soporta correctamente el trabajo con clases, métodos y campos Ghosts.

Modelo de inferencia
El modelo de inferencia, para esta versión, ya es capaz de identificar correctamente ciertas estructuras en el código donde pueden estar presentes Ghosts, estas estructuras son:

- Invocación de métodos, para identificar métodos Ghost
- Creación de instancias de clase (instrucción *new*), para identificar nuevas clases Ghost y sus constructores.
- Acceso a Campos, para identificar nuevos campos Ghost.
- Nombres (tales como `a` o `Point)`, que pueden ser variables, nombres de clases o accesos a miembros de la clase actual. Casos entre los cuales esta versión solo soporta nombres de clases.

En las secciones siguientes se explica la estructura actual de la aplicación.

2.4 Trabajo relacionado

En ambientes Smalltalk, existe una ventana de navegación de código llamada *Class Browser*. Esta ventana, ver Figura 2.7, permite a los programadores organizar sus programas en paquetes que contienen clases, las cuales, a su vez, organizan sus métodos mediante categorías.

En su trabajo, Scharli y Black [10], nutren este sistema incorporando cuatro nuevas categorías automáticas que permiten una vista de alto nivel de las aplicaciones: la categoría *requires* que refiere a todos aquellos métodos que son enviados a *self* o a alguna de sus súper-clases, pero

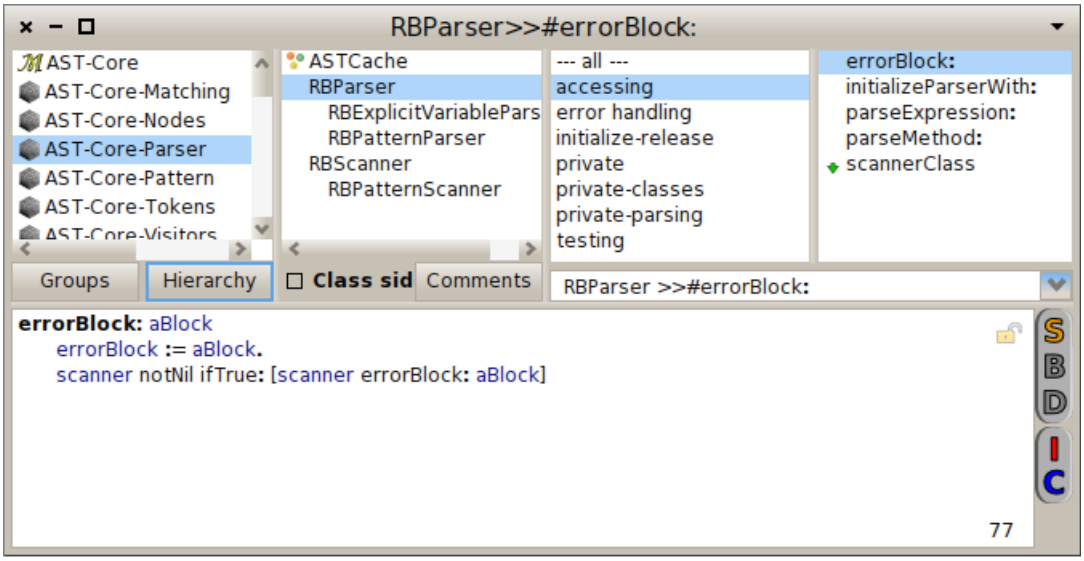

Figura 2.7: *Class Browser* en Smalltalk, la primera columna representa los paquetes, la segunda las clases, la tercera las categorías de métodos y la cuarta los métodos en la categoría seleccionada.

para los cuales no existe implementación; la categoría *supplies* que agrupa a aquellos métodos concretos que implementan métodos abstractos heredados; la categoría *overrides* que, como su nombre lo dice, engloba los métodos que re-implementan otros métodos concretos heredados; y por último, la categoría *sending super* que corresponde a todos aquellos métodos que envían mensajes a `super`. De entre estas categorías, resalta naturalmente *requires*, puesto que permite identificar aquella funcionalidad que aún está pendiente, pero que es utilizada por el código, y que representa la interacción entre una clase y su padre. Esta funcionalidad se asemeja a la forma en que Ghosts muestra las entidades no implementadas, pero de manera mucho más local, y únicamente resaltando la falta de concordancia entre miembros de una jerarquía.

Existen otros IDEs que incorporan, aunque en menor escala, soporte para entidades no definidas que, a pesar de mencionarlas como errores, son capaces de recolectar información relevante [3].

En su estudio, Tanter y Callaú analizaron los alcances de dichos soportes para Visual Studio 2010 (C#), IntelliJ's IDEA 11 (Java) y NetBeans 7.0.1 (Java), junto con dos *plug-ins* populares de Visual Studio, ReShaper [11] y CodeRush [12]. En la Figura 2.8 aparecen las preguntas relacionadas a los problemas que pretende resolver Ghosts, que se ha comprobado que perduran en versiones posteriores, donde, para cada pregunta se entrega la respuesta por defecto y se resalta aquellas excepciones notables. Si bien estas herramientas se encargan de manejar entidades no definidas, es el concepto de considerarlas como errores lo que conduce a soluciones que apuntan, no a la generación de una entidad basada en su uso, sino a la corrección local de un error mediante la generación de una entidad no definida en ese contexto.

Por otra parte, tanto Visual Studio (mediante *Generate From Usage* [13]), como CodeRush (mediante *Consumer First Development* [14]), apuntan explícitamente a la programación incremental y, aun así, pierden generalidad en el enfoque. Dicha pérdida es justamente el problema que pre-

Questions	Default answer	Notable features
Definition		
How are undefined types reported in the IDE?	as errors	
Can undefined types be distinguished easily?	no	IDEA, ReSharper, CodeRush report all undefined entities in a specific manner
Are members of undefined types also reported?	no	IDEA, ReSharper, CodeRush do report them
Is it possible to easily identify the current set of ghost members associated to a single (undefined) type?	no, all undefined entities are signaled similarly	
How are undefined members of external libraries reported?	like any undefined members	CodeRush reports them as actual errors, similar to type errors
Verification		
Is the use of undefined entities subject to type checking?	no	IDEA, ReSharper only check some argument types and some local variable assignments
Generation		
Can an undefined entity be fully generated in a single click?	no	NetBeans, IDEA, VisualStudio, ReSharper can only generate a class with a single constructor at once. Visual Studio can generate fields for constructor arguments. CodeRush can generate a type with several members at once, but only based on the usages in the current file
Does generation force a switch to the buffer of the new entity?	yes	VisualStudio can generate in the background

Figura 2.8: Comparación del soporte para programación incremental en distintas herramientas (tomado de [3]).

tende solucionar el uso de Ghosts como abstracción ante la presencia de código incompleto, que incluye, en sí mismo, decisiones arquitectónicas relevantes para el desarrollador. Finalmente, al interpretar de manera no intrusiva, se permite al usuario caer en errores menores de tipeo, de ahí la relevancia de una comprensión acabada de la intención del programador al momento de utilizar esta abstracción.

Capítulo 3

Reestructurando Ghosts

Ghosts es un *plug-in*, para desarrollo en Java, en Eclipse Indigo. Dentro de Eclipse, Ghosts tiene su propia perspectiva (la abstracción que hace Eclipse para referirse a los distintos ambientes de trabajo en los que se puede trabajar el mismo código). Esta perspectiva, como se muestra en la Figura 2.4, incluye, como características importantes, la *Ghost View*, el editor y el *Project Explorer*. Tanto en el editor como en el *Project Explorer*, el *plug-in* agrega indicadores para mostrar la presencia de Ghosts y mensajes de error personalizados para indicar la presencia de errores en las definiciones de los Ghosts.

Todo el código presente a continuación corresponde a trabajo realizado durante el transcurso de esta memoria. Ya sea por efecto de refactorización o derechamente por implementación.

3.1 Estructura de la aplicación actual

Ghosts está escrito en Java y sigue la lógica de integración de PDE [15]. Todos los paquetes que conforman la aplicación están nombrados siguiendo la práctica común de las bibliotecas de Eclipse, esto es, de la forma `cl.pleiad.ghosts*`. Cabe destacar que esta estructura de paquetes (salvo *Completion* y *Tests*) ya estaba en la primera versión de la aplicación. A continuación, se presenta una pequeña descripción de cada paquete y sus clases relevantes:

- Paquete principal (`cl.pleiad.ghosts`)

  Contiene una única clase, *GhostsPlugin*, que cumple con el modelo de integración con Eclipse mediante las bibliotecas de *plug-in*.

- Blacklist

  Contiene la clase *GBlackList* que se encarga de manejar y cargar la lista de bibliotecas no nativas de Java que se desea usar en la aplicación, para que Ghosts no las detecte como clases de tipo Ghost. Esto se hace mediante un archivo de configuración en el que se especifican aquellas clases.

- Completion

  En este paquete, se encuentran las clases encargadas de manejar el sistema autocompletado de Ghosts. Estas clases cumplen con la estructura de *CompletionProposal* de Eclipse [16]. Dicha estructura exige la implementación de un *CompletionProposalComputer* encargado de interpretar el código y producir, para cada autocompletado, una lista de *CompletionProposals* que representan las posibles formas de autocompletar, permitiendo así que Ghosts participe del proceso de autocompletado nativo.

- Core

  Este paquete contiene la jerarquía de clases que refiere a las entidades que se manejan en la aplicación, los Ghosts, representando los distintos tipos de Ghosts que pueden estar presentes en el código. Esta jerarquía contiene desde la clase principal (*Ghost.java*) hasta los casos específicos: Campos (*GField.java*), Variables (*GVariable.java*), clases (*GClass.java*), Métodos (*GMethod.java*), entre otros.

- Decorators

  Aquí se encuentra la clase encargada de agregar los decoradores, puntos negros sobre los iconos de los proyectos que indican la presencia de Ghosts al explorador de paquetes de Eclipse.

- Dependencies

  Todas las clases auxiliares, necesarias para completar la abstracción de un Ghost presente en el código se encuentran en este paquete. Estas clases complementan la información de la jerarquía presente en el *core* con datos sobre los tipos (*TypeRef.java*), sobre los trozos de código que están representando (*ISourceRef.java*) y sobre los marcadores, en el código, de los que son responsables (*MarkerSourceRef.java*).

  Adicionalmente, en este paquete se encuentra una de las clases más importantes de la aplicación, *TypeInferencer.java*. Esta clase es la responsable de inferir, en base al código actual y la información contenida en los Ghosts, el tipo de una cierta expresión para poder tipificar correctamente los nuevos Ghosts que se van reconociendo, así como las expresiones en las que estos Ghosts están contenidos. Esta clase será explicada en detalle en la siguiente sección.

- Engine

En este paquete están las clases encargadas del funcionamiento de la aplicación en sí. Aquí se encuentra la clase principal de la aplicación, *SGhostEngine.java*, que mantiene todas las instancias importantes. Junto con dicha clase se encuentran las clases encargadas de detectar el cambio en un archivo para refrescar la información de los Ghosts, los *listeners*, y el *visitor* encargado de detectar dónde se producen exactamente esos cambios.

Este paquete contiene otra de las clases más importantes de la aplicación, aquella encargada de identificar los contextos en los que se puede presentar un Ghost y como ese contexto determina qué tipo de Ghost es, *ASTGhostVisitor.java*. Esta clase también será explicada en detalle en la siguiente sección.

- Markers

  Contiene exclusivamente el marcador específico que indica la presencia de un Ghost en el código, en la ventana de edición de texto, *GhostMarker.java*.

- Perspective

  En este paquete se encuentra la clase que describe la forma que tiene la *Ghost Perspective* (*GhostPerspective.java*), como perspectiva específica del entorno de Eclipse.

- Tests

  Este paquete contiene una batería de pruebas unitarias para *core* junto con algunas clases utilitarias y de experimentación de funcionalidad.

- View

  Aquí se encuentran las clases que describen el elemento más importante de la *Ghost Perspective*, la *Ghost View* (*GhostView.java*). En esta clase se encuentran definidas todas las funcionalidades presentes en la *Ghost View*, incluyendo *bust it!*. Adicionalmente, en este paquete, se encuentra las clases que describen la forma en que se despliegan los elementos que encapsulan a los Ghosts (*GTreeNode.java*) en la vista misma: *GViewTreeContentProvider.java* y *GViewLabelProvider.java*.

- Writer

  Este paquete contiene la jerarquía de clases que se encargan realmente de la funcionalidad *bust it!*. Cuando *bust it!* se aplica a una clase Ghost (*GhostBusterType.java*) y cuando se aplica un miembro de una clase concreta (*GhostBusterMember.java*). Además, usando la misma abstracción, se encarga del caso en el que el Ghost sea una clase que existe y se ha importado (*NonGhostImporter.java*).

3.2 Interpretando Ghosts

Ghosts cuenta con un motor de inferencia que le permite identificar, cada vez que se produce un cambio en el código, los nuevos Ghosts que se producen, así como también las modificaciones a los Ghosts ya existentes en un proyecto. Este funcionamiento está dictado por dos entidades fundamentales. La primera, *ASTGhostVisitor.java*, que se encarga de recorrer el código, poniendo atención en aquellos casos de interés, recolectando ocurrencias de Ghosts. Y la segunda, *TypeInferencer.java*, que se encarga de responder a consultas enviadas por el *visitor* sobre los tipos de los nodos del código que se están analizando.

3.2.1 Modelo de inferencia

El objetivo del modelo de inferencia de Ghosts es el tipificar una expresión recibida desde el código para poder definir correctamente un Ghost en el momento de su creación. A continuación se presenta, en pseudo-código, una versión simplificada del funcionamiento general de *TypeInferencer.java*.

Algoritmo 1 Método inferType

```
function INFERTYPE(Exp e)
    if ISSIMPLENAME(e) OR ISMETHODINVOCATION(e) OR ISFIELDACCESS(e) then
        return GETCONTEXT(e)
    else
        return Object
    end if
end function
```

Como se muestra en el Algoritmo 1, el modelo de inferencia se encarga de retornar un tipo específico, de acuerdo al contexto en que se encuentre el código que se está analizando. Mediante el método *inferType* que recibe una expresión[1] y retorna un tipo[2]. Cuando la expresión es un nombre (*SimpleName*, que engloba tanto un nombre simple de variable como un nombre de tipo), una invocación de método (*MethodInvocation*) o un acceso a un campo (*FieldAccess*)), el método delega la responsabilidad de responder por el tipo de la expresión al contexto en que esta expresión está contenida. En cualquier otro caso, retorna el tipo básico para cualquier clase, *Object*. El soporte a estas estructuras en específico tiene que ver con la forma en que este método es invocado en *ASTGhostVisitor*.

Luego, según el Algoritmo 2, se utiliza la expresión definida para obtener su contexto, su padre en esta abstracción, que es a su vez otra expresión.

[1] de la clase `org.eclipse.jdt.core.dom.Expression`

[2] representado por un objeto de la clase `cl.pleiad.ghosts.dependencies.TypeRef`

Algoritmo 2 Método getContext

```
function GETCONTEXT(Exp e)
    Exp p ← GETPARENT(e)
    if ISSIMPLENAME(e) then
        if ISASSIGNMENT(p) then
            Exp o ← OTHERSIDE(p,e)
            return INFERTYPE(o)
        else
            return INFERTYPEOFVAR(e)
        end if
    end if
    if ISVARIABLE(p) then
        return INFERTYPEOFVAR(e)
    else if ISMETHODINVOCATION(p) then
        Type t ← INFERTYPEOFARGUMENT(p,e)
        return t
    else if ISRETURNSTATEMENT(p) then
        return INFERTYPEOFRETURN(p)
    else if ISEXPRESSIONSTATEMENT(p) then
        return Void
    else if ISDOWHILE(p) OR ISWHILE(p) OR ISIF(p) then
        return Boolean
    end if
    return Object
end function
```

En el caso que la expresión sea un *SimpleName* y que esté dentro de una asignación (como, por ejemplo, `a = 2`) se consulta al tipo del otro lado (en el ejemplo, tomando *e* como *a*, el tipo deducible de *a* es inmediatamente *Int*, dado que 2 es un entero). En caso contrario, se asume que ya se ha definido dicha variable en algún punto anterior del código y que, por lo tanto, se puede obtener dicho tipo de su declaración. Para obtener el tipo declarado de una variable se usa el método auxiliar *inferTypeOfVar*. Si la expresión está dentro de una variable se sigue la misma lógica que en el caso anterior.

Si el contexto es una invocación de método, se verifica el tipo de la expresión como parámetro de dicho método, mediante *inferTypeOfArgument*, asumiendo que el tipo se puede deducir de la definición del método y de la posición de la expresión en la invocación. De manera similar, en el caso de un *return*, el tipo se obtiene mediante una llamada al método *inferTypeOfReturn*.

Cuando la expresión está contenida en una declaración, como por ejemplo la invocación del método `a.print()` dentro de la declaración `a.print();`, se presume que es una expresión sin tipo de retorno, dado que no está siendo usada para nada, y se retorna *Void*, puesto que en Java no se puede utilizar de esa forma ningún método que no tenga tipo de retorno *Void*. Siguiendo la misma lógica, cuando la expresión es parte de la condición de un *do-while*, un *while* o un *if*, dicha

condición debe ser un *Boolean*. En caso de que el contexto no aplique en ninguno de los casos anteriores, se retorna *Object*.

Por simplicidad, se asume que el método *inferTypeOfVar* realiza exactamente lo que se espera, y si no encuentra una definición retorna *Object* o *Undefined* dependiendo si la variable es en realidad un campo sin tipo declarado o una variable que nunca ha sido declarada. Del mismo modo, *inferTypeOfArgument* y *inferTypeOfReturn* operan revisando localmente los tipos en el contexto dado, del método al cual refiere o en el cual está contenido, respectivamente.

3.2.2 Nodos en el código

Para poder identificar la aparición de nuevos Ghosts en el código, así como también los cambios ya recolectados, el primer paso es recorrer el código en busca de los puntos donde un Ghost puede estar definido. Para esto, Ghosts cuenta con *ASTGhostVisitor.java*, que se encarga de navegar los nodos (*ASTNode*) del código y extraer información de alto nivel sobre el contexto en el que se encuentra cada ocurrencia de un cierto nodo.

A continuación, se presentan, en pseudo-código, los casos de nodos que maneja *ASTGhostVisitor.java*, de manera simplificada. Estos casos, siguiendo el patrón *Visitor* [17], son casos del método *endVisit*.

Algoritmo 3 Método endVisit(SimpleType)

```
function ENDVISIT(SimpleType node)
    Type t ← RESOLVETYPE(node)
    if NOT (ISCLASS(t) OR ISINTERFACE(t)) then
        return
    else if ISINBLACKLIST(t) then
        return
    end if
    Ghost g ← GETGHOSTCONSIDERING(node)
    ADDDEPENDENCY(g,node)
    . . .
end function
```

En el caso de un *SimpleType*, ver Algoritmo 3, que representa a una declaración de tipo (como por ejemplo `Type t` o `Type` en ciertos contextos), al ser el caso más general, verifica si el tipo asociado a la expresión cumple con las condiciones base, ser una clase válida y no estar presente en la *BlackList*, y crea el Ghost, agregando inmediatamente el nodo como la primera referencia de éste. Este caso actúa como el caso base del sistema, todas las clases e interfaces Ghosts son creados aquí, todos los otros casos de encargan de dar propiedades específicas a estas clases o de recolectar

los miembros de ellas. Esto porque *SimpleType* es un sub-nodo de la gran mayoría de los nodos posibles. El código faltante será explicado más adelante.

El método *resolveType* es un pseudo-código para una serie de procedimientos de la biblioteca nativa de Eclipse que permite obtener y validar una clase mencionada en un nodo.

Algoritmo 4 Método endVisit(MethodInvocation)

```
function ENDVISIT(MethodInvocation node)
    if RESOLVEMETHOD(node) ≠ null then
        return
    end if
    GMethod m ← new GMETHOD(node)
    if GETEXPRESSION(node) ≠ null then
        SETOWNERTYPE(m,INFERTYPE(GETEXPRESSION(node)))
    else
        SETOWNERTYPE(m,GETCURRENTTYPE())
    end if
    SETRETURNTYPE(m,INFERTYPE(node))
    for all arg : GETARGUMENTS(node) do
        ADDPARAMETER(m,INFERTYPE(arg))
    end for
    ADDDEPENDENCY(m,node)
    CHECKANDUNIFY(m)
    CHECKANDMUTATE(m)
end function
```

La resolución de una invocación de método (*MethodInvocation*), ver Algoritmo 4, se realiza verificando que el método, en primer lugar, no sea un método ya definido (mediante el método auxiliar *resolveMethod*). Luego, se crea el Ghost que corresponde al método y se busca la clase a la que pertenece y cuál es su tipo de retorno. Para determinar su dueño, este método verifica si la llamada tiene referencia a su dueño o no (`a.b()` o `b()` respectivamente). En el primer caso, se llama a *inferType* para determinar el tipo de su dueño, usando la referencia presente en el nodo (`a` en el ejemplo). En caso que no se tenga referencia al dueño, el único posible dueño, siguiendo la sintaxis de Java, es la clase actual, en cuyo caso se llama al método *getCurrentType* para recibir directamente una referencia al tipo de la clase actual. El tipo de retorno se infiere del tipo disponible para el nodo de la invocación, nuevamente usando *inferType*. Una vez que se tiene esta información, se infieren los tipos de los argumentos y se agrega el nodo como dependencia del nuevo método Ghost.

Algoritmo 5 Método endVisit(ClassInstanceCreation)

```
function ENDVISIT(ClassInstanceCreation node)
    if RESOLVECONSTRUCTOR(node) ≠ null then
        return
    end if
    GConstructor c ← new GCONSTRUCTOR()
    SETOWNERTYPE(c,INFERTYPE(node))
    for all arg : GETARGUMENTS(node) do
        ADDPARAMETER(c,INFERTYPE(arg))
    end for
    ADDDEPENDENCY(c,node)
    CHECKANDUNIFY(c)
    CHECKANDMUTATE(c)
end function
```

Muy similar al caso anterior es la resolución de un *ClassInstanceCreation*, ver Algoritmo 5, que representa la creación de una instancia de clase (`Ghost g = new Ghost(1)`, por ejemplo). En este caso, para verificar la existencia del constructor, se utiliza el método *resolveConstructor*. Además, las únicas diferencias con el anterior son la información de tipo de retorno, que se obtiene gratis dado que un constructor retorna su tipo, y de donde se obtiene el dueño. En este caso, al estar en el contexto de la creación de instancia como nodo, la información de tipo está declarada directamente en la expresión, la única verificación que hace *inferType* es determinar si la igualdad es válida en base al tipo declarado a la izquierda de la expresión.

Algoritmo 6 Método endVisit(FieldAccess)

```
function ENDVISIT(FieldAccess node)
    if RESOLVEFIELD(node) ≠ null then
        return
    end if
    GField f ← new GFIELD(node)
    SETOWNERTYPE(f,INFERTYPE(GETEXPRESSION(node)))
    SETRETURNTYPE(f,INFERTYPE(node))
    ADDDEPENDENCY(f,node)
    CHECKANDUNIFY(f)
    CHECKANDMUTATE(f)
end function
```

Siguiendo la misma dinámica que en los casos anteriores, en el caso de un acceso a un campo, *FieldAccess*, ver Algoritmo 6, se realiza una verificación, mediante el método *resolveField*, para comprobar si el campo ya está definido. Luego de verificar, se crea el campo Ghost y se asignan su tipo de retorno, su dueño y su nueva referencia. La diferencia sustancial entre este caso y el de una invocación de método es que el sistema no reconoce un identificador solo como un acceso a campo, por lo que no es necesario verificar en qué contexto se encuentra el nodo.

Algoritmo 7 Método endVisit(SimpleName)

```
function ENDVISIT(SimpleName node)
    if ISASSIGNMENT(GETPARENT()) then
        if RESOLVETYPE(node) ≠ null then
            if NOT ISGHOST(node) then
                return
            end if
        end if
        GField f ← new GFIELD(node)
        SETOWNERTYPE(f,GETCURRENTTYPE())
        SETRETURNTYPE(f,INFERTYPE(node))
        ADDDEPENDENCY(f,node)
        CHECKANDUNIFY(f)
        CHECKANDMUTATE(f)
    end if
end function
```

Para el caso de una llamada a una variable que se encuentra en una asignación, ver Algoritmo 7, si el nodo no es un Ghost, entonces la única alternativa que queda es que éste sea un campo de la clase actual que está siendo llamado sin usar la pseudo-variable `this`. Este caso se trata de la misma manera que el de un acceso a campo, pero asignando la clase actual como dueño.

Los métodos *checkAndUnify* y *checkAndMutate*, presentes en todos los casos, tienen como objetivo evitar la duplicación y mantener la información más correcta posible, respectivamente. *checkAndUnify* es un método que busca en la lista actual de Ghosts si ese Ghost que se está agregando ya existe, y si existe le agrega la nueva información entregada por el nodo actual. *checkAndMutate* verifica, luego de agregar un miembro a una clase Ghost, si la clase ya tenía otro miembro y si no lo tenía, transforma dicha clase de interfaz a clase propiamente tal, dado que en Java, una clase sin métodos ni campos definidos es una interfaz.

Capítulo 4

Extendiendo Ghosts

El trabajo de esta memoria está enfocado en completar Ghosts como herramienta. Para ello es necesario extender el modelo presentado en el capítulo anterior a un modelo que soporte la mayor cantidad posible de código y de la mejor manera, considerando la información disponible en el código de un proyecto en que se está trabajando. Además, se necesita que la herramienta esté integrada de mejor manera con Eclipse y que no sea un simple *plug-in* con su propio entorno, sin soporte para acciones que el usuario espera poder realizar.

Todo el código presente a continuación corresponde a desarrollo realizado durante el transcurso de esta memoria.

4.1 Casos de interés

A continuación, se presentan todas las extensiones al modelo de inferencia, al *visitor* y sus respectivos desafíos particulares.

4.1.1 Excepciones

La primera extensión sencilla al modelo de Ghosts es el soporte para excepciones. En Java, las excepciones se presentan en tres contextos distintos, ver Figura 4.1.

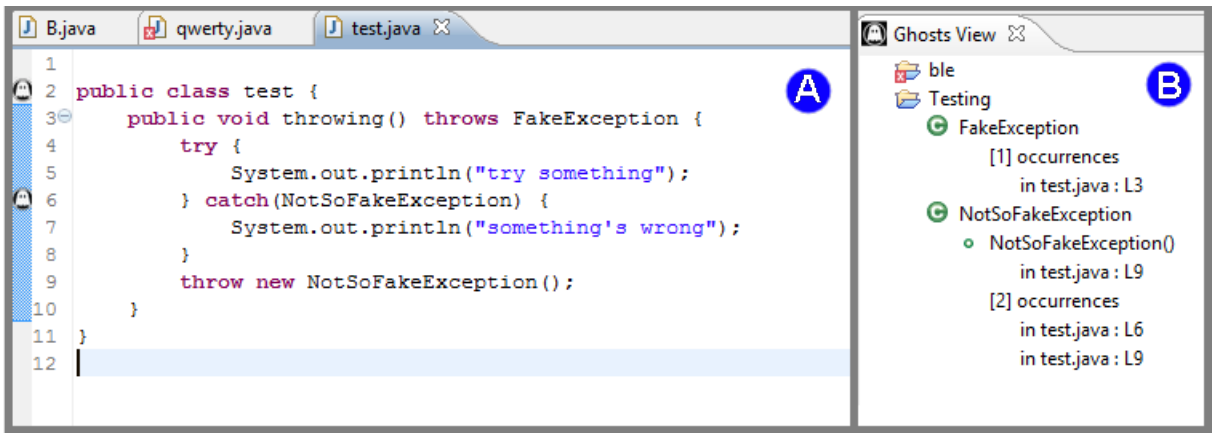

Figura 4.1: En *A* se puede ver, en el mismo método, los 3 contextos en que se puede presentar una excepción: dentro de un *catch*, en un *throw* o en un *throws* en la firma del método. En *B* se puede ver la información recabada en la *GhostView*.

La única característica particular de una excepción Ghost, con respecto a las demás clases, es que deben heredar de *java.lang.Exception*, cosa que debe ser asegurada cuando el usuario utiliza la herramienta *bust it!*, como se muestra en la Figura 4.2.

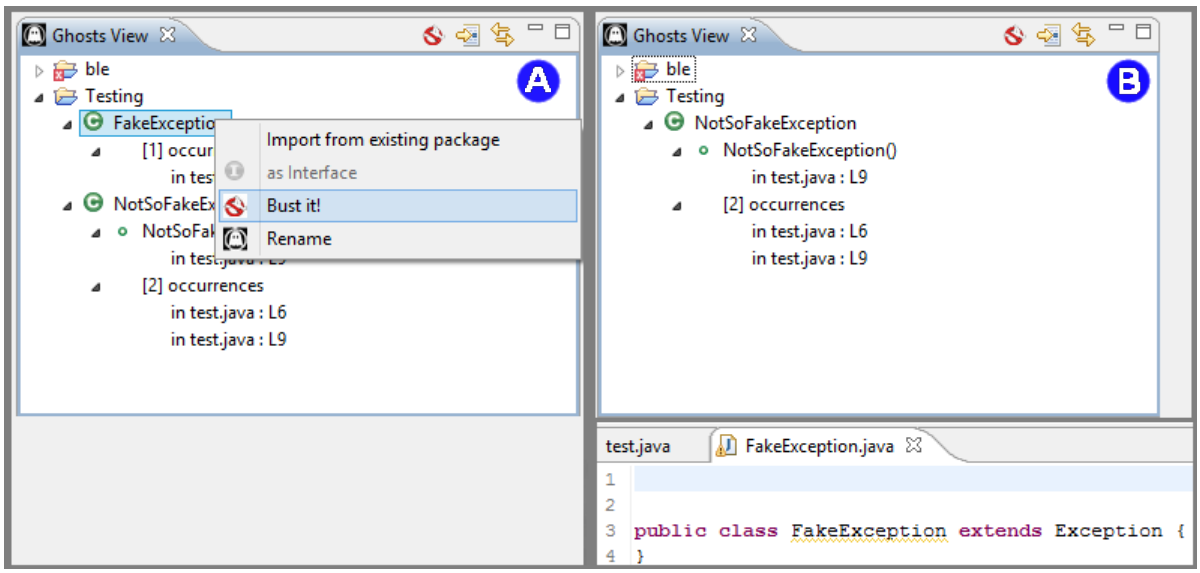

Figura 4.2: Cuando se presiona *Bust It!* (*A*) se concretiza la clase y, en el caso de una Excepción, ésta inmediatamente extiende de *java.lang.Exception* (*B*).

Para poder agregar soporte para excepciones, es necesario capturar estos tres contextos, entre los nodos del código, y modificar dichas clases Ghost para que extiendan de *java.lang.Exception*. Esto se logra agregando tres nuevos casos al *visitor*.

Algoritmo 8 Método endVisit(CatchClause)

```
function ENDVISIT(CatchClause node)
    Exception e ← GETEXCEPTION(node)
    if RESOLVETYPE(e) ≠ null then
        Ghost g ← GETGHOSTTYPE(e)
        if g ≠ null then
            if GETSUPERCLASS(g) = null then
                SETSUPERCLASS(g, Exception)
            else
                return
            end if
        end if
    end if
end function
```

En el primer caso, ver Algoritmo 8, se obtiene la excepción definida en el nodo, mediante *getException*, y, a partir de ella, la clase Ghost que ya debe estar en la lista, mediante *getGhostType*. Una vez obtenida la clase se verifica que el Ghost no tenga súper-tipo y se asigna *Exception*. En caso que la clase ya tenga tipo, por construcción, éste ya debe ser *Exception*, por lo que no es necesario volver a asignarlo.

Algoritmo 9 Método endVisit(ThrowStatement)

```
function ENDVISIT(ThrowStatement node)
    Exception e ← GETEXPRESSION(node)
    if RESOLVETYPE(e) ≠ null then
        Ghost g ← GETGHOSTTYPE(e)
        if g ≠ null then
            if GETSUPERCLASS(g) = null then
                SETSUPERCLASS(g, Exception)
            else
                return
            end if
        end if
    end if
end function
```

El caso de *throw*, ver Algoritmo 9, el algoritmo es prácticamente igual, con la salvedad que para obtener la excepción se solicita a la declaración su expresión interna. En el código real, esto involucra un par de chequeos adicionales (de tipo en variables internas de la biblioteca).

Algoritmo 10 Método endVisit(MethodDeclaration)

```
function ENDVISIT(MethodDeclaration node)
    for all excep : THROWNEXCEPTIONS(node) do
        Ghost g ← GETGHOSTTYPE(excep)
        if g ≠ null then
            if GETSUPERCLASS(g) = null then
                SETSUPERCLASS(g, Exception)
                ADDDEPENDENCY(g,node)
            else
                return
            end if
        end if
    end for
end function
```

Por último, ver Algoritmo 10, cuando se buscan excepciones en la declaración de un método, se deben recorrer las excepciones lanzadas por el método y, para cada una, se debe realizar el algoritmo visto en la parte anterior. Cabe destacar que, en este caso, es necesario agregar la llamada en el método como una nueva dependencia, puesto que, en este contexto, las excepciones declaradas dentro del método no son *SimpleType* y, por lo tanto, no serán agregadas por el caso base, dejando al Ghost sin referencias.

Esta extensión no requiere ningún cambio en las reglas de inferencia, puesto que el cambio realizado a los tipos refiere al súper-tipo del nodo, información que, salvo en una declaración de clase, no es deducible del código, en Java.

4.1.2 Miembros de la súper-clase

Una extensión que es fundamental para permitir la construcción de jerarquías de clases, usando Ghosts, es la posibilidad de hacer llamados a miembros de la súper-clase, con súper-clase Ghost.

Para lograr esto, en primer lugar, es necesario agregar a las clases Ghost información sobre sus sub-tipos, para poder centralizar la validación de llamadas a la pseudo-variable *super*. Esto se consigue extendiendo la clase *GClass* agregando dicha información. Dicha extensión se llama *GExtendedClass*. Con esta extensión disponible, resta agregar a Ghosts soporte para los tres contextos de campos de súper-clase, ver Figura 4.3.

Lo anterior se consigue agregando una extensión al *visitor* para cada caso y extendiendo el modelo de inferencia para soportar la presencia de miembros de la súper-clase.

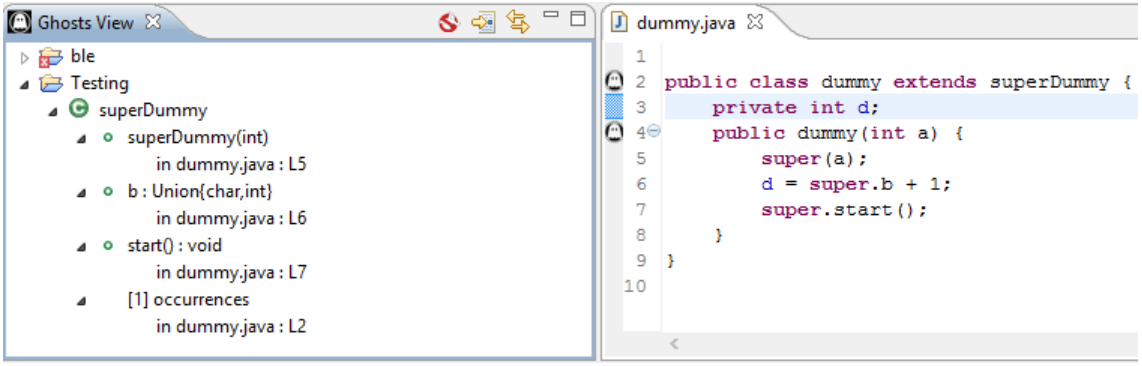

Figura 4.3: En el constructor de *dummy* se pueden ver los 3 contextos no ambiguos de miembros de la súper-clase: el súper constructor, un campo y un método.

Algoritmo 11 Método endVisit(SimpleType) extendido con súper-clase Ghost

function ENDVISIT(*SimpleType node*)
. . .
ASTNode parent ← GETPARENT(*node*)
. . .
if ISTYPEDECLARATION(*parent*) **then**
MUTATE(*g*)
ADDEXTENDER(*g*,GETCURRENTTYPE())
end if
end function

La primera extensión, ver Algoritmo 11, es agregar el soporte para una clase Ghost que es extendida, o sea que está en la declaración de la clase (`class a extends b {...}`, donde `b` es el Ghost). Para esto, se agrega al código de *SimpleType* la posibilidad de que la clase esté en dicho contexto. Si el padre del nodo que generó una nueva clase Ghost está inserto en una declaración de clase, entonces éste debe ser un Ghost que está siendo extendido, por construcción. Una vez identificado esto, se muta la clase, mediante el método *mutate*, para soportar la adición de clases que lo extiendan y se agrega la clase actual como sub-clase.

Algoritmo 12 Método endVisit(SuperMethodInvocation)

```
function ENDVISIT(SuperMethodInvocation node)
    Ghost superGhost ← GETSUPERGHOST()
    if superGhost ≠ null then
        GMethod m ← new GMETHOD(node)
        SETOWNERTYPE(m,superGhost)
        SETRETURNTYPE(m,INFERTYPE(node))
        for all arg : GETARGUMENTS(node) do
            ADDPARAMETER(m,INFERTYPE(arg))
        end for
        ADDDEPENDENCY(m,node)
        CHECKANDUNIFY(m)
    end if
end function
```

Una vez que ya se tiene en la lista de Ghosts el concepto de Ghost extendido, se agrega el soporte para miembros de la súper-clase. El primero de éstos, ver Algoritmo 12, es la invocación a método de la súper-clase (*SuperMethodInvocation*). Este método es prácticamente igual al caso de una invocación de método, con la salvedad que en este caso es necesario verificar la presencia de una súper-clase Ghost (dado que en Java sólo puede haber una súper-clase por clase) y no existe ambigüedad en quien es el dueño del método, siempre es la súper-clase.

Algoritmo 13 Método endVisit(SuperFieldAccess)

```
function ENDVISIT(SuperFieldAccess node)
    Ghost superGhost ← GETSUPERGHOST()
    if superGhost ≠ null then
        GField f ← new GFIELD(node)
        SETOWNERTYPE(f,superGhost)
        SETRETURNTYPE(f,INFERTYPE(node))
        ADDDEPENDENCY(f,node)
        CHECKANDUNIFY(f)
        CHECKANDMUTATE(f)
    end if
end function
```

Siguiendo la misma lógica, se agrega soporte para campos de la súper-clase, ver Algoritmo 13. Al igual que en el caso anterior, el algoritmo es idéntico al caso de campos de clase salvo por el chequeo de existencia de una súper-clase y que esta clase debe ser la dueña del campo.

Algoritmo 14 Método endVisit(SuperConstructorInvocation)

```
function ENDVISIT(SuperConstructorInvocation node)
    Ghost superGhost ← GETSUPERGHOST()
    if superGhost ≠ null then
        GConstructor c ← new GCONSTRUCTOR()
        SETOWNERTYPE(c,superGhost)
        SETRETURNTYPE(c,INFERTYPE(node))
        for all arg : GETARGUMENTS(node) do
            ADDPARAMETER(c,INFERTYPE(arg))
        end for
        ADDDEPENDENCY(c,node)
        CHECKANDUNIFY(c)
        CHECKANDMUTATE(c)
    end if
end function
```

El último caso de esta extensión, tratado especialmente por la biblioteca de *ASTNode*, el caso de una invocación al súper constructor (*SuperConstructorInvocation*), representada por las llamadas a método de la forma `super(<arguments>)`. Según se muestra en el Algoritmo 14, este caso es igual al caso de una invocación a método de la súper-clase, excepto por la creación de un constructor Ghost en vez de un método.

Finalmente, ver Algoritmo 15, la última extensión necesaria para soportar miembros de súper-clase es agregar el soporte correspondiente en el modelo de inferencia, o sea agregar los casos al método *inferType*.

Algoritmo 15 Método inferType extendido con miembros de súper-clase

```
function INFERTYPE(Exp e)
    if ISSIMPLENAME(e) OR ISMETHODINVOCATION(e)
    OR ISFIELDACCESS(e) OR ISSUPERMETHODINVOCATION(e)
    OR ISSUPERFIELDACCESS(e) then
        return GETCONTEXT(e)
    else
        return Object
    end if
end function
```

El método *getSuperGhost* presentado en los casos anteriores implica, a simple vista, una búsqueda en la lista de Ghosts para encontrar la súper-clase declarada en la clase en que se está trabajando. Esta búsqueda se realiza en paralelo a las búsquedas realizadas por *inferType*, por lo que, para poder visualizar la información más actualizada posible, sin caer en errores de edición concurrente, es necesario usar un tipo especial de lista. Para la lista de Ghosts, se utiliza la clase *java.util.concurrent.CopyOnWriteArrayList* [18]. Esta clase permite que ambas búsquedas, sumadas

a una tercera búsqueda que se menciona más adelante, puedan ocurrir de manera concurrente y sin pérdida de información.

4.1.3 Asociación de expresiones

Para que la expresividad del código que puede escribir el usuario tenga algún uso práctico, es necesario que los Ghosts que utilice en su código puedan estar incluidos en expresiones. Hasta ahora, si un miembro Ghost es parte de una expresión (`int c = a.b + 1`, con `b` Ghost) la única información que se puede deducir sobre el tipo de un miembro es la que se obtiene de su declaración, por lo tanto, a falta de dicha información, la inferencia de un miembro cae en el caso base (`b` es considerado de tipo *Object*).

Extender el modelo de inferencia para el soporte de expresiones requiere de incorporar al modelo reglas de inferencia específicas para tipificar una expresión en cada contexto. Para esto, ver Algoritmo 16, es necesario extender el método *getContext*.

Algoritmo 16 Método getContext, extensiones para asociación de expresiones

```
function GETCONTEXT(Exp e)
    Exp p ← GETPARENT(e)
    ...
    if ISVARIABLE(p) then
        ...
    else if ISASSIGNMENT(p) then
        return INFERTYPE(LEFTSIDE(p))
    else if ISPREFIXEXPRESSION(p) then
        if ISNOT(GETOPERATOR(p)) then
            return Boolean
        else
            return Union(Char, Int)
        end if
    else if ISPOSTFIXEXPRESSION(p) then
        return Union(Char, Int)
    else if ISINFIXEXPRESSION(p) then
        return INFERINFIX(e,p)
    end if
    return Object
end function
```

Agregando el caso que el contexto sea una asignación, considerando los parámetros de entrada de *getContext*, se manejan todas la expresiones básicas en las que el lado izquierdo de la asignación

no es la declaración (`a = super.b` en contraste con `int a = super.b`, donde el tipo de `a` está explícito en la misma expresión).

Para las expresiones propiamente tales, es necesario diferenciar entre tres tipos: Operaciones prefijas (`!a` o `++b`), operaciones sufijas (`b++`) y operaciones infijas (`a + b`). En operaciones prefijas, la única que es aplicable a un *Boolean* es negación, todas las demás corresponden a operaciones numéricas. En cuanto a operaciones sufijas, Java cuenta solamente con dos de ellas, la adición y la sustracción de 1 (`a++` y `a--`), que son sólo aplicables a números.

Dado que en Java un caracter (*Char*) es internamente un número, los caracteres soportan todas las operaciones que soporta un entero, por lo que Ghosts, al entregar la respuesta más genérica, retorna un tipo unión para todos los casos en que no se pueda evitar la ambigüedad, por lo que retorna *Union(Char,Int)* para todas las operaciones aplicables a enteros y *Union(Char,Int,String)* para el caso particular de la suma, que puede ser aplicada para concatenar *String*. Para desambiguar, Ghosts busca la información más exacta al momento de aplicar el método *checkAndUnify*, puesto que es en este método en donde recibe la información sobre las otras instancias de un cierto Ghost en el código.

El caso de las operaciones infijas es un poco más complejo, puesto que no se puede generalizar de la misma forma que con las anteriores. Además de que estas operaciones pueden estar anidadas de manera no trivial, produciendo errores de tipo que Ghost debe saber identificar para evitar tipificar incorrectamente una expresión.

El método *inferInfix*, ver Algoritmo 17, retorna para cada caso el tipo esperado en cada expresión, como por ejemplo, para un mayor o igual, retorna que la expresión debe ser numérica. En cada caso, cuando se está en presencia de una expresión anidada, el método llama recursivamente a *getContext*, que eventualmente puede volver a caer en *inferInfix* pero con el padre como nodo a analizar. Los dos casos interesantes en este algoritmo son: el caso de la suma y el caso de la igualdad (o desigualdad). En el caso de la suma, dado que Java soporta su sobrecarga como concatenación de *String* es necesario verificar cuál es el tipo resultante que se espera obtener para la expresión, preguntando recursivamente al padre, luego, con este resultado, se asigna un tipo acorde a la intención del programador. El caso de la igualdad (la rama `else` del algoritmo), al no tener mayor información sobre el tipo esperado que la que se encuentra en las expresiones izquierda y derecha, se consulta a la expresión contraria. En caso de recursión infinita en cualquiera de las ramas, el algoritmo real maneja la cantidad de llamadas y retorna `Object`, dado que ese es el único tipo que se puede entregar sin mayor información.

Algoritmo 17 Método inferInfix

```
function INFERINFIX(Exp e, Exp p)
    Operator o ← GETOPERATOR(p)
    if ISCONDITIONALAND(o) OR ISCONDITIONALOR(o)
    OR ISAND(o) OR ISOR(o) OR ISXOR(o) then
        if ISINFIXEXPRESSION(e) then
            return GETCONTEXT(p)
        else
            return Boolean
        end if
    else if ISLESS(o) OR ISGREATER(o) OR ISLESSEQUALS(o)
    OR ISGREATEREQUALS(o) OR ISTIMES(o) OR ISDIVIDE(o)
    OR ISREMAINDER(o) OR ISMINUS(o) OR ISLEFTSHIFT(o)
    OR ISRIGHTSHIFTSIGNED(o) OR ISRIGHTSHIFTUNSIGNED(o) then
        if ISINFIXEXPRESSION(e) then
            return GETCONTEXT(p)
        else
            return Union(Char, Int)
        end if
    else if ISPLUS(o) then
        if INFERTYPE(p) = String then
            return String
        else if INFERTYPE(p) = Char then
            return Union(Char, Int)
        else if INFERTYPE(p) = Int then
            return Union(Char, Int)
        else
            return Union(Char, Int, String)
        end if
    else
        if LEFTSIDE(p) = e then
            return INFERTYPE(RIGHTSIDE(p))
        else
            return INFERTYPE(LEFTSIDE(p))
        end if
    end if
    return Object
end function
```

4.1.4 Llamadas en cadena

Hasta ahora, Ghosts es capaz de tipificar y recolectar correctamente un gran espectro de expresiones, pero no es capaz de identificar correctamente a miembros de objetos que sean, a su vez,

miembros de un Ghost. A esto se le llama una llamada en cadena, una expresión de la forma `a.b.c` donde `a` y `b` son clases, que pueden ser, potencialmente, Ghosts.

Este soporte requiere de tomar en cuenta una serie de casos aún no considerados. En primer lugar, parte del código faltante en *endVisit(SimpleType)*, ver Algoritmo 18.

Algoritmo 18 Método endVisit(SimpleType)

```
function endVisit(SimpleType node)
    Type t ← resolveType(node)
    if NOT (isClass(t) OR isInterface(t)) then
        return
    else if isInBlackList(t) then
        return
    end if
    Ghost g ← getGhostConsidering(node)
    addDependency(g,node)
    ...
    ASTNode parent ← getParent(node)
    if isFieldDeclaration(parent) OR isVariableDeclaration(parent) then
        for all variable : getFragments(parent) do
            if isTypeDeclaration(getParent(parent)) then
                GField f ← new GField(variable)
                setOwnerType(f,getCurrentType())
                setReturnType(f,g)
                addDependency(f,node)
                checkAndUnify(f)
                checkAndMutate(f)
            else
                GVariable v ← new GVariable(variable)
                setContext(v,getParent(parent))
                setReturnType(v,g)
                addDependency(v,node)
                addGhostToList(v)
            end if
        end for
    else if isTypeDeclaration(parent) then
        mutate(g)
        addExtender(g,getCurrentType())
    end if
end function
```

En el Algoritmo 18, se presenta la versión completa de *endVisit(SimpleType)*. Además de mostrar el código agregado para soportar súper-clases Ghost se muestra también el código que se encarga de guardar las referencias más tempranas de los campos de la clase actual y de las variables

presentes en dicha clase. Por fragmento (*fragment*) este código hace referencia a cada una de las variables presentes en una declaración, dado que en Java una declaración puede ser múltiple (como por ejemplo `int a,b,c;`). Para cada uno de estos fragmentos, si es que el padre del padre es la declaración misma de la clase, se crean campos de la clase actual, de la misma forma que en *endVisit(FieldAccess)*. En caso contrario, los fragmentos corresponden a una declaración de variable. Una variable Ghost, a diferencia de un campo, no tiene un dueño asociado, sino un contexto, y no es chequeada en búsqueda de repeticiones puesto que cada declaración indica una variable distinta. Estas variables son agregadas directamente a la lista de Ghosts, trabajo que en el resto de los casos es parte del trabajo de *checkAndUnify*.

Con lo anterior en mente, se pueden analizar los distintos casos de llamadas en cadena. Los casos más fáciles de identificar, llamadas a miembros de la clase actual (mediante *this*) y de la súperclase (mediante *super*) ya están cubiertos. Las llamadas a miembros de una variable específica (`a.b`) ahora están totalmente cubiertos, ya sea un miembro de una variable o de un campo de la clase actual al cual se accede sin usar *this*.

Los casos más complejos, como llamadas múltiples (como por ejemplo `a.b.c.d`), tienen tres casos base fundamentales: Donde la llamada comienza en *this*, donde la llamada comienza en *super* y donde la llamada comienza en un nombre de miembro cualquiera. Cada uno de estos casos contiene un sub-caso que es importante recalcar, cuando alguno de los miembros intermedios no está definido en ningún otro lado, ver Figura 4.4.

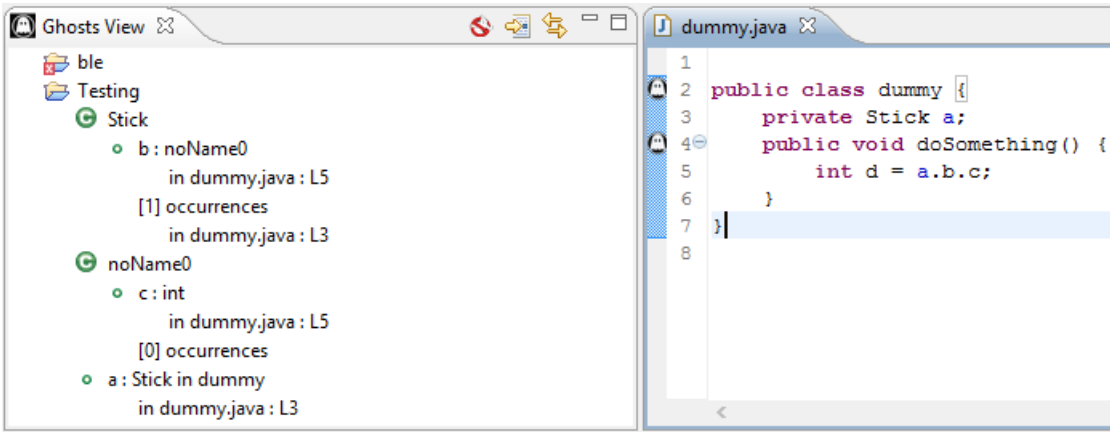

Figura 4.4: En la figura, *a* es un Ghost y *c* es un miembro de "algo" que contiene a un *c* entero.

En este caso, Ghosts genera un tipo ficticio que recolecte la información que sí puede ser recabada sobre él, sin necesidad de conocer su nombre.

Para poder soportar llamadas en cadena se deben realizar un par de extensiones a *getContext* y *inferType*.

En *inferType*, ver Algoritmo 19, el único cambio para tener la versión final es agregar *QualifiedName*, o sea, llamado en cadena, a la lista de posibles entradas del método.

Algoritmo 19 Método inferType, con extensión para llamdas en cadena

```
function INFERTYPE(Exp e)
    if ISSIMPLENAME(e) OR ISMETHODINVOCATION(e)
    OR ISFIELDACCESS(e) OR ISSUPERMETHODINVOCATION(e)
    OR ISSUPERFIELDACCESS(e) OR ISQUALIFIEDNAME(e) then
        return GETCONTEXT(e)
    else
        return Object
    end if
end function
```

En *getContext*, ver Algoritmo 20, es necesario incluir todos los nodos que son raíz de una llamada en cadena, incluyendo los casos recursivos. La idea es inferir los tipos de todos aquellos nodos, que se envían desde *inferType*, que son parte de una expresión más grande y que, por lo tanto, en algún paso de la recursión llegan a los casos base. Dicha recursión es manejada en el método *inferChainedContext*.

Algoritmo 20 Método getContext, extensiones para llamadas en cadena

```
function GETCONTEXT(Exp e)
    Exp p ← PARENT(e)
    . . .
    if ISVARIABLE(p) then
        . . .
    else if ISMETHODINVOCATION(p) OR ISFIELDACCESS(p)
    OR ISSUPERFIELDACCESS(p) OR ISSUPERMETHODINVOCATION(p)
    OR ISQUALIFIEDNAME(p) then
        return INFERCHAINEDCONTEXT(p)
    end if
    return Object
end function
```

Para manejar la recursión en la asignación de un tipo, ver Algoritmo 21, el método *inferChainedContext* maneja todos los posibles contextos. En caso de que la expresión padre sea *this*, se está en presencía de una llamada de la forma `this.a`, por lo que se busca la expresión entre los miembros de la clase actual.

Cuando el contexto es un *SimpleName* el caso es ligeramente diferente, puesto que significa que la llamada tiene la forma `a.b` donde `a` puede ser un miembro de la clase actual o una variable definida en el alcance del nodo. En el primer caso, es necesario identificar el tipo del padre (`a`) para luego buscar en ese Ghost (o clase) el tipo del nodo. En el segundo, se busca directamente en la lista por el padre, puesto que todas las variables se encuentran en la lista, y se le pregunta el tipo del miembro al cual representa el nodo.

Los casos de acceso a miembros (métodos o campos) de una clase y de acceso a miembros de la súper clase son equivalentes, salvo el hecho de que en el primer caso se realiza una llamada recursiva para buscar el tipo del resto de la expresión, dado que es en este caso en el que caen todas las llamadas con más de dos identificadores.

Una vez que ya se ha deducido el tipo más correcto queda hacer una última verificación. Si éste tipo llega a ser *Object*, Ghosts asume que esto se debe a que el programador olvidó definir dicho miembro y se crea un tipo ficticio para poder de todas formas recolectar la información adicional, presente en el código, sobre dicho tipo. El caso en que el programador realmente pretenda extender `Object` con nuevos miembros está descartado, puesto que dicha extensión no está permitida en Java.

Algoritmo 21 Método inferChainedContext

```
function INFERCHAINEDCONTEXT(Exp e)
    Exp p ← GETEXPRESSION(e)
    Type r
    if ISTHIS(p) then
        r ← GETMEMBERTYPE(e,GETCURRENTTYPE())
    else if ISSIMPLENAME(p) then
        Type aux ← GETMEMBERTYPE(p,GETCURRENTTYPE())
        if aux ≠ null then
            r ← GETMEMBERTYPE(e,aux)
        end if
        if r = null then
            r ← GETVARIABLETYPE(e,p)
        end if
    else if ISFIELDACCESS(p) OR ISMETHODINVOCATION(p) then
        Type parent ← INFERCHAINEDCONTEXT(p)
        r ← GETMEMBERTYPE(e,parent)
    else if ISSUPERFIELDACCESS(p) OR ISSUPERMETHODINVOCATION(p) then
        r ← GETMEMBERTYPE(p,GETSUPERGHOST())
    end if
    if r = Object then
        Type noName ← GETNONAMETYPE()
        Ghost noNameGhost ← new GCLASS(noName)
        ADDGHOSTTOLIST(noNameGhost)
        r ← noName
    end if
    return r
end function
```

Para que lo anterior funcione, falta agregar un caso al *visitor*, el caso de *QualifiedName*. Este caso, ver Algoritmo 22, genera un nuevo campo, de la misma forma que en el caso de un acceso a campo de clase, para los nodos que corresponden a llamados en cadena.

Algoritmo 22 Método endVisit(QualifiedName)

```
function ENDVISIT(QualifiedName node)
    if RESOLVEBINDING(node) ≠ null then
        return
    end if
    GField f ← new GFIELD(node)
    SETOWNERTYPE(f,INFERTYPE(GETEXPRESSION(node)))
    SETRETURNTYPE(f,INFERTYPE(node))
    ADDDEPENDENCY(f,node)
    CHECKANDUNIFY(f)
    CHECKANDMUTATE(f)
end function
```

4.1.5 Modelo de inferencia extendido

Finalmente, se presentan las versiones completas de *getContext* y *endVisit(SimpleType)*.

Algoritmo 23 Método getContext, versión final

```
function GETCONTEXT(Exp e)
    Exp p ← GETPARENT(e)
    if ISSIMPLENAME(p) then
        if ISASSIGNMENT(e) then
            Exp o ← OTHERSIDE(p,e)
            return INFERTYPE(o)
        else
            return INFERTYPEOFVAR(e)
        end if
    end if
    if ISVARIABLE(p) then
        return INFERTYPEOFVAR(e)
    else if ISMETHODINVOCATION(p) then
        Type t ← INFERTYPEOFARGUMENT(p,e)
        return t
    else if ISRETURNSTATEMENT(p) then
        return INFERTYPEOFRETURN(p)
    else if ISEXPRESSIONSTATEMENT(p) then
        return Void
    else if ISDOWHILE(p) OR ISWHILE(p) OR ISIF(p) then
        return Boolean
    else if ISMETHODINVOCATION(p) OR ISFIELDACCESS(p)
    OR ISSUPERFIELDACCESS(p) OR ISSUPERMETHODINVOCATION(p)
    OR ISQUALIFIEDNAME(p) then
        return INFERCHAINEDCONTEXT(p)
    else if ISASSIGNMENT(p) then
        return INFERTYPE(LEFTSIDE(p))
    else if ISPREFIXEXPRESSION(p) then
        if ISNOT(GETOPERATOR(p)) then
            return Boolean
        else
            return Union(Char, Int)
        end if
    else if ISPOSTFIXEXPRESSION(p) then
        return Union(Char, Int)
    else if ISINFIXEXPRESSION(p) then
        return INFERINFIX(e,p)
    end if
    return Object
end function
```

Algoritmo 24 Método endVisit(SimpleType) versión final

```
function ENDVISIT(SimpleType node)
    Type t ← RESOLVETYPE(node)
    if NOT (ISCLASS(t) OR ISINTERFACE(t)) then
        return
    else if ISINBLACKLIST(t) then
        return
    end if
    Ghost g ← GETGHOSTCONSIDERING(node)
    ADDDEPENDENCY(g,node)
    ASTNode parent ← GETPARENT(node)
    if ISFIELDDECLARATION(parent) OR ISVARIABLEDECLARATION(parent) then
        for all variable : GETFRAGMENTS(parent) do
            if ISTYPEDECLARATION(GETPARENT(parent)) then
                GField f ← new GFIELD(variable)
                SETOWNERTYPE(f,GETCURRENTTYPE())
                SETRETURNTYPE(f,g)
                ADDDEPENDENCY(f,node)
                CHECKANDUNIFY(f)
                CHECKANDMUTATE(f)
            else
                GVariable v ← new GVARIABLE(variable)
                SETCONTEXT(v,GETPARENT(parent))
                SETRETURNTYPE(v,g)
                ADDDEPENDENCY(v,node)
                ADDGHOSTTOLIST(v)
            end if
        end for
    else if ISTYPEDECLARATION(parent) then
        MUTATE(g)
        ADDEXTENDER(g,GETCURRENTTYPE())
    end if
end function
```

4.2 Integrándose a Eclipse

Lo visto en las secciones anteriores presenta a Ghosts como un *plug-in* que agrega la posibilidad de trabajar con entidades no definidas, pero que trabaja en su propia perspectiva y cuenta con sus propias herramientas, como *bust it!*. Pero Ghosts está integrado de manera más completa que eso. Ghosts cuenta con herramientas adicionales que le permiten al programador realizar operaciones que espera poder realizar al trabajar con clases cotidianas.

4.2.1 Autocompletado

Una de las herramientas que provoca que los programadores opten por utilizar IDEs es el autocompletado de código. En Eclipse, la herramienta de autocompletado permite al usuario manejar todos los nombres de clases, variables, métodos o campos, definidos en el alcance actual, sin tener que escribir sus nombres completos. Además, le permite conocer los miembros disponibles para una cierta clase que no conoce.

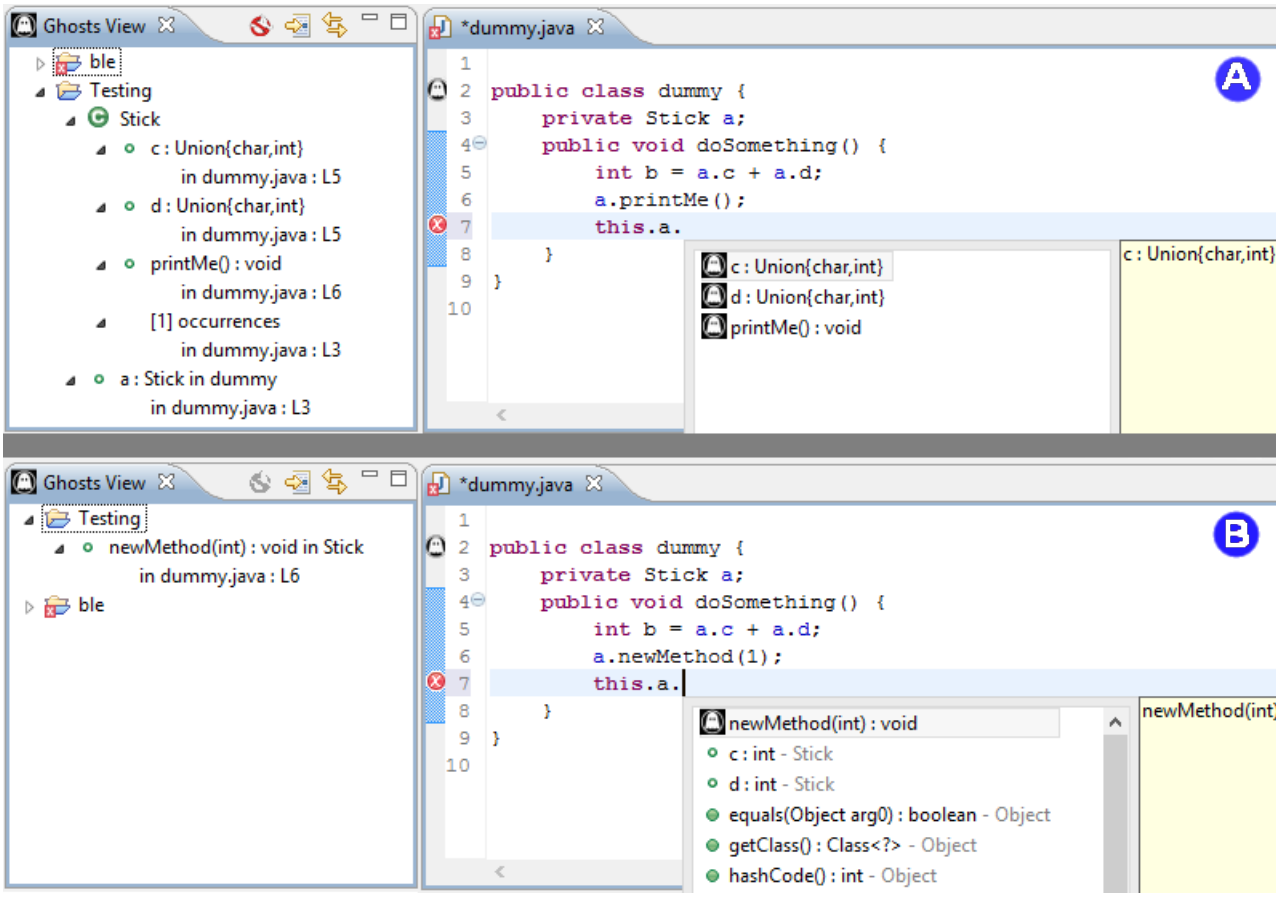

Figura 4.5: Autocompletado de Ghosts.

Ghosts permite autocompletado sobre los miembros de sus tipos. Este autocompletado se nutre de la información provista por los Ghosts y combina dicha información con la que el autocompletado nativo sí es capaz de identificar. Como se puede ver en Figura 4.5, cuando la clase de la que se está hablando es un Ghost, el sistema presenta todos sus campos como sugerencias que son Ghosts. En cambio cuando la clase existe, y tiene miembros Ghosts, sólo estos miembros se despliegan como Ghosts.

Para lograr esta integración se utiliza el punto de extensión de JDT llamado *Java Completion Proposal Computer* [16]. Esta biblioteca permite que Ghosts participe del proceso de generación de sugerencias de autocompletado. Para lograr esta participación Ghost tiene su propia sub-clase de *JavaCompletionProposalComputer* y su propia sub-clase de *JavaCompletionProposal* que le permiten computar la presencia y crear, respectivamente, las sugerencias de autocompletado referentes a un Ghosts. Si bien las implementaciones de estas clases no son triviales, por falta de soporte para identificar entidades no definidas en la biblioteca, su principal funcionalidad es el uso de *inferType* y la lista de Ghosts para identificar, en cada contexto, desde qué clase Ghost es necesario solicitar los miembros para generar las sugerencias de autocompletado.

4.2.2 Refactoring

Una de las herramientas más poderosas de Eclipse, y de la mayoría de los IDEs, es la posibilidad de realizar refactorizaciones triviales sobre el código, y de esa forma ahorrar al programador trabajo tedioso, repetitivo y sensible a errores. En Eclipse, todas las herramientas de refactorización permiten la participación de *plug-ins* mediante el punto de extensión de JDT llamado *Refactoring Contribution*.

Ghosts define sus herramientas de refactorización, por ahora *bust it!* y *rename*, de manera nativa al *plug-in*, sin conectarse con la biblioteca de extensión. Esto debido a que dicha biblioteca no es capaz de manejar una refactorización sobre un tipo que no está definido todavía, caso que trata como error y cae en un caso básico, para todas las refactorizaciones.

Tanto en *bust it!* como *rename*, toda la información necesaria para manejar la refactorización está disponible en los Ghosts, por lo cual su implementación es tan compleja como es el uso de la biblioteca de escritura a archivo. En el caso de *rename*, el único caso que es necesario controlar es el renombre de Ghosts *noName*, puesto que al ser Ghosts que no tienen ninguna referencia en el código. El renombrarlos no tiene ningún sentido, esto pues Ghosts deduce dichos tipos del código e indica, en su nombre, que no tienen un nombre declarado.

En la Figura 4.6, se pueden apreciar los pasos de un *rename*.

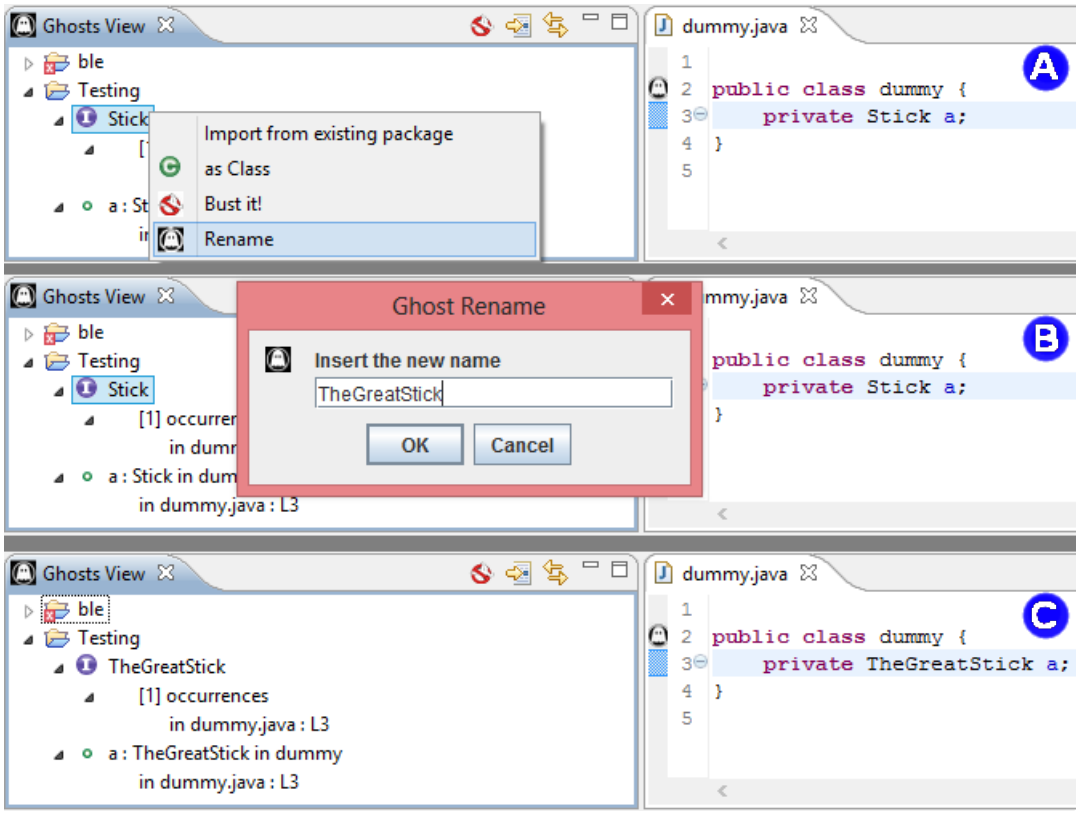

Figura 4.6: Pasos del *Rename*.

4.3 Limitaciones de la aplicación

Como se comentó en la sección anterior, una de las grandes limitaciones de Ghosts es que, por efecto de la biblioteca, cada módulo de refactorización no está acoplado a las herramientas de refactorización nativas de Eclipse. Por lo que, además de generar posibles confusiones al usuario, hace difícil el agregado de nuevas refactorizaciones, para tener disponibles las mismas funcionalidades para Ghosts que para el resto de las clases.

Otra limitación, visible al utilizar el *plug-in*, es la imposibilidad de probar el código que se está trabajando con Ghosts. Esto pues las funcionalidades provistas por Ghost no generan automáticamente las clases, eso debe hacerlo el usuario. Ghosts utiliza información que está disponible en tiempo de compilación, por lo que no necesita verificar nada relacionado con la ejecución de la aplicación.

Otro tema que puede ser problemático es que Ghosts no puede razonar sobre jerarquías de clases. La información que Ghosts maneja sobre las clases Ghost es, a lo sumo, la súper-clase obvia de un cierto Ghost. Por lo que, para realizar una jerarquía usando Ghosts, es necesario tener en cuenta que el soporte sólo considera la presencia de una súper-clase Ghost para una clase real y la presencia de Ghosts con súper-clases obvias, como son las excepciones.

Por último, la limitación más evidente al trabajar en desarrollos serios, presente en Ghosts, es la ausencia de soporte para *Generics* [19]. Ghosts no soporta el uso de clases Ghost dentro de clases genéricas típicas, como listas, o el uso de Ghosts que sean en si clases genéricas. Esto implica que, en su versión actual, Ghosts no permite el desarrollo usando *Generics* dentro del código, lo cual es especialmente problemático a la hora de programar un sistema que aprovecha las cualidades de un chequeo estático de tipos en miembros de un conjunto, en una asociación exclusiva entre dos clases, o en cualquier contexto donde sea necesario tener certeza de que ciertos elementos, cuyo tipo es entregado como parámetro, efectivamente son objetos de las clases que presumen ser.

Capítulo 5

Conclusiones

5.1 Contribuciones

Ghosts es una herramienta completa y funcional, útil a la hora de pensar en el desarrollo de un *software*, desde cero, sin claridad de la forma final que deben tener las clases que lo componen. Ghosts es un *plug-in*, para Eclipse, con su propio *workspace*, que mediante el concepto de Ghost presenta una interfaz simple y limpia, pues mantiene la disposición personalizada del usuario en su interfaz para Java, que permite programar incrementalmente sin interferir en el proceso mental del programador, dándole visibilidad de lo que su código significa en términos de entidades no existentes aún.

Ghosts cuenta con soporte para la presencia de entidades Ghost en elementos relevantes de la sintaxis de Java: clases, interfaces, métodos, campos, variables, ciclos, etc. Además, incluye soporte para estructuras importantes que entregan información interesante sobre los tipos de lo que se está programando: excepciones, expresiones (algebraicas y lógicas), llamadas a la pseudo-variable *this* y sus campos, llamadas a la pseudo-variable *super* y sus campos, y llamadas en cadena.

Ghosts cuenta también con una serie de herramientas para ayudar al programador. Dentro de la *GhostView* (Figura 5.1), para cada Ghost se pueden aplicar varias operaciones: Transformar clases en interfaces, y viceversa (Exclusivo para clases/interfaces Ghosts); importar la definición de algún paquete externo (Exclusivo para clases/interfaces Ghosts); aplicar *bust it!* y concretizar el Ghost; y renombrar la clase o el miembro del que se está hablando.

Entre las mencionadas, el renombrar, es la única de las herramientas que tiene como objetivo suplir la imposibilidad de aplicar la herramienta de refactorización, de Eclipse, a las clases Ghosts. Si bien está dentro de las pretensiones de la aplicación el suplir todas las funcionalidades

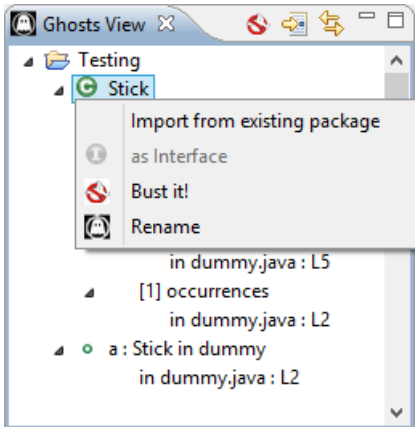

Figura 5.1: Herramientas de apoyo aplicables a un Ghost.

de refactorización que tengan sentido aplicar a un Ghost, por el momento, la única presente es el renombrar.

Dentro del transcurso de esta memoria, se refinaron, y completaron, todos los algoritmos explicados en la Sección 3. Adicionalmente, se realizaron todos los cambios explicados en la Sección 4.

Además de las implementaciones concretas, se estudiaron diversas alternativas para resolver el problema de desarrollo dedicado para los módulos de refactorización, entre las que destacan: Adaptaciones a la biblioteca nativa, mediante el uso de sub-clases de estructuras de uso interno, idea descartada por su gran complejidad y bajísima mantenibilidad; y el uso de respaldos ocultos del código recolectado, o *backends*, para guardar la información de los Ghosts en clases que si existieran, permitiendo utilizar las herramientas de Eclipse sin adaptación alguna, idea que se descartó por la necesidad de esconder del programador todos aquellos Ghosts que fueran miembros de una clase real, puesto que dicho código estaría presente en esas clases, introduciendo de forma invasiva código que el usuario no escribió.

De la misma forma, se estudió la posibilidad de agregar soporte para *Generics* siguiendo el modelo descrito en las secciones anteriores. Lamentablemente no se pudo completar el soporte, dejando fuera el caso de clases genéricas concretas instanciadas con un Ghost (un ejemplo de esto sería `java.util.ArrayList<A>` con `A` Ghost). Dado este resultado se resolvió quitar el soporte incompleto, puesto que el uso de *backends*, como se explicaba anteriormente, permite fácilmente este soporte.

5.2 Reflexiones

5.2.1 Herramientas no intrusivas

La programación de estilo incremental y la programación guiada por pruebas (TDD) son estrategias que aseguran que el foco del programador esté en la lógica de la aplicación en desarrollo, y no en la forma final del código. Estas técnicas, que son altamente valoradas por la comunidad, no tienen un buen soporte en Eclipse. El soporte disponible, carece de las características mínimas que permitan un trabajo fluido y enfocado en mantener al programador en línea con la intención de lo que se está programando. En este contexto, herramientas como Ghosts son necesarias para permitirle al programador utilizar aquellas técnicas que han surgido como necesidad para mejorar la mantenibilidad y la comprensión del código.

Es importante que las herramientas que se generen siguiendo esta lógica conserven el espíritu de ayuda al programador, de guía, y no de reemplazo del mismo. Las interfaces intrusivas y orientadas a reducir al mínimo el tiempo de desarrollo muchas veces generan problemas no fáciles de detectar, por la enorme cantidad de código que el programador no razona, sino que deja a la herramienta generar de manera automática. Si bien la mayoría de esos problemas son fáciles de solucionar, es una mejor idea entregar al programador, desde un principio, las decisiones de cómo debe ser el código y, sin distraerlo de su cometido, producir por él solamente lo que él ha declarado que desea. En este contexto, una posible mejora a Ghosts es reemplazar los tipos básicos de retorno, insertados automáticamente al utilizar *bust it!*, por excepciones que indiquen que dichos métodos no han sido implementados aún, pero dicha idea requiere de una segunda mirada.

5.2.2 Probando Ghosts

En estos momentos se está diseñando un experimento que tiene como objetivos comprobar la usabilidad de Ghosts y determinar, en comparación con Eclipse sin Ghosts, que tanto afecta el diseño de software utilizando Ghosts, en contraste con las herramientas nativas de Eclipse para la generación automática de código en presencia de entidades no definidas.

El experimento consiste en completar la batería de pruebas de dos problemas sencillos, uno usando Eclipse y el otro utilizando Eclipse con Ghosts. Cada uno de los sujetos debe realizar, al azar uno u otro experimento utilizando Ghosts. En cada uno de los problemas los sujetos son enfrentados a un conjunto de pruebas explicadas, en cuanto a lo que deben probar, y se les pide completar dichas pruebas sin implementar el código en sí. La idea detrás del experimento es enfrentar a los sujetos al desarrollo de un sistema, utilizando *TDD*, en un escenario con Ghosts y otro sin Ghosts.

Luego del realizado el experimento, se espera medir la cantidad y naturaleza de los fallos, incurridos por los sujetos. Con dicho experimento se espera medir el aporte real de Ghosts como apoyo al desarrollo y, de esta forma, manejar adecuadamente las expectativas de cómo seguir. Además, con estos resultados, en caso de ser positivos, se pretende realizar una publicación y fomentar el desarrollo de herramientas no intrusivas, orientadas al desarrollo *Top-Down*.

La versión actual del enunciado del experimento se encuentra en el Anexo 6.1.

5.2.3 Trabajo Futuro

El concepto de Ghost es una abstracción adecuada para enfrentar el problema de entidades no definidas. Si bien es un concepto sencillo, permite entender fácilmente el estado actual del código. Además es un concepto que no tiene ninguna ligazón particular con Eclipse o Java. Este concepto es perfectamente aplicable en cualquier contexto de desarrollo donde haya entidades no definidas. Un ejemplo de esto es la implementación de Ghosts existente para Pharo Smalltalk.

Existen dos caminos que pueden seguirse desde la implementación actual, para mejorarla.

El primer camino, siguiendo la lógica actual, consiste en agregar los casos de refactorización que tenga sentido soportar por un Ghost. Esta idea permite completar el soporte de refactorización para Ghosts replicando mecánicamente las funcionalidades provistas por Eclipse, utilizando el conocimiento adquirido al desarrollar *rename*. Esta solución tiene como ventaja el no acoplamiento con Eclipse, que asegura una mayor portabilidad a la hora de actualizar Ghosts para versiones posteriores de Eclipse. La desventaja de esta solución está en que, al no depender de la biblioteca de Eclipse, el *plug-in* no se ve completamente integrado y tiende a sobrecargar el trabajo realizado por el IDE en vez de aprovechar los procedimientos que ya están en marcha.

El segundo, es seguir explorando el uso de *backends* en búsqueda de una manera de esconder por completo la presencia del código. Una manera de abordar éste camino, es incorporar a Ghosts la capacidad de interferir la ejecución de la instancia misma de Eclipse y esconder el código asociado a los Ghosts, para permitir una absoluta integración a todas las herramientas nativas de Eclipse. Lo anterior, en teoría, puede ser logrado mediante el uso de Aspectos [21], en particular, de *AspectJ* [22], que está presente en Eclipse como *plug-in*. Esta idea tiene como desventaja la necesidad de acoplar Ghosts a un segundo *plug-in* (o a alguna otra biblioteca de orientación a Aspectos) haciendo difícil la mantenibilidad general de la aplicación.

Cualquiera sea el camino que se tome, es importante que el concepto detrás de esta memoria, el concepto de Ghosts, se extienda a otros ambientes y que la idea de desarrollar herramientas de este estilo continúe.

Bibliografía

[1] K. Beck. Test-Driven Development: By Example. Addison-Wesley, 2002.

[2] C. Larman. Agile and Iterative Development:A Manager's Guide. Addison-Wesley Professional, 2003.

[3] O. Callaú and É. Tanter. Programming with ghosts. Software, IEEE, 30(1):74–80, 2013.

[4] Scott W Ambler and Assoc. http://www.ambysoft.com/surveys/howAgileAreYou2010.html, 2010. S.W. Ambler, How Agile Are You? 2010 Survey Results.

[5] G. Goth. Beware the march of this ide:eclipse is overshadowing other tool technologies. Software, IEEE, 22(4):108–111, 2005.

[6] Java exception. http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Exception.html.

[7] Java Object. http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html.

[8] Intelligent Code Completion. http://eclipse.org/recommenders/features/completion/.

[9] Indigo Refactoring Support. http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm.

[10] Nathanael Schärli and Andrew P. Black. A browser for incremental programming. Comput. Lang. Syst. Struct., 30(1-2):79–95, 2004.

[11] Resharper, Developer Productivity Tool for Microsoft Visual Studio. http://www.jetbrains.com/resharper/.

[12] Devexpress, CodeRush. http://www.devexpress.com/Products/CodeRush/.

[13] Microsoft Development Network, "Generate from Usage". http://msdn.microsoft.com/en-us/library/dd409796.aspx.

[14] Devexpress, CodeRush, "Consume-first Development". http://www.devexpress.com/Products/CodeRush/consume_first_development.xml.

[15] Eclipse Plugin Development. http://www.eclipse.org/pde/.

[16] Eclipse Extension Point para participar en el proceso de sugerencias de contenido (Java Completion Proposal Computer). http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/extension-points/org_eclipse_jdt_ui_javaCompletionProposalComputer.html.

[17] Robert C. Martin. The Principles, Patterns, and Practices of Agile Software Development. Prentice Hall, 2002.

[18] API de CopyOnWriteArraylist.html. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html.

[19] Java Generics. http://docs.oracle.com/javase/1.5.0/docs/guide/language/generics.html.

[20] Stackoverflow, sitio comunitario de preguntas y respuestas orientadas a la programación.

[21] Anurag Mendhekar Chris Maeda Cristina Videira Lopes Jean-Marc Loingtier John Irwin Gregor Kiczales, John Lamping. Aspect-oriented programming. ECOOP, pages 220–242, 1997.

[22] Sitio del proyecto de Eclipse AspectJ. http://eclipse.org/aspectj/.

Capítulo 6

Anexo

6.1 Experimento

Problema 1, Cuentas Internacionales

Se desea hacer un programa que represente la interacción de un usuario con su cuenta bancaria, en cajeros automáticos que manejan distintos tipos de moneda.

Para ello debe crear las clases `User` y `Account` que representan al usuario y a su cuenta, respectivamente. Además se cree la clase `ATM` que representa a los cajeros automáticos. Por simplicidad, el sistema sólo debe manejar Dólares y Pesos, mediante las clases `Dollar` y `Peso`, ambas clases deberán proveer el método `getQuantity` que retorna la cantidad, en `double`, de dinero que representan.

Cada `User` podrá retirar dinero, en algún tipo de moneda, mediante su método `getMoney`, que extraerá dicho dinero de un `ATM`, indicando también de que cuenta (`Account`) y cuanto quiere retirar, expresado como `double`. Si la cantidad solicitada supera la cantidad presente en la cuenta, esta debe retornar el máximo presente en la cuenta.

Las cuentas se crearán para un tipo de moneda específica y un usuario en particular. La cantidad de dinero que almacenan debe ser representada por un tipo de moneda, no por un número. Estas deberán proveer el método `getAmount` que retorna la cantidad de dinero, en pesos/dólares que queda en la cuenta.

Los ATM también deben estar asociados a un tipo de moneda en específico, pero deben permitir la extracción de dinero de cualquier cuenta. Lo anterior mediante el método getMoney, que dado un usuario y una cuenta específicos, junto con una cantidad de dinero representada por un número (en double), retornará el dinero solicitado, en la moneda que maneja. Además los ATM deberán proveer las operaciones:

- depositMoney que dado un usuario, una cuenta y una cantidad de dinero, representada por un número (en double), aumenta la cantidad de dinero en la cuenta, guardando cuidado con la moneda que maneja el ATM.
- transferMoney que dado un usuario, una cuenta propia, una cuenta ajena y una cantidad de dinero representada por un número (en double), debe traspasar el dinero de una cuenta a otra, teniendo en consideración los tipos de monedas que manejan ambas cuentas, y por el ATM. Además deberá retornar true si la operación es exitosa y false si no lo es.

No debe crear jerarquías de ATM o de Account, sólo preocúpese que ambas clases manejen los dos tipos de moneda.

Para sus pruebas, cree la clase TestAccounts que extienda de TestCase y escriba los siguientes métodos de prueba:

- testGetMoney: Para probar el retiro de dinero por parte de un usuario, de su cuenta, en un cajero que utiliza la misma moneda que su cuenta.
- testGetTonsOfMoney: Para probar el retiro de dinero por parte de un usuario, de su cuenta, en un cajero que utiliza la misma moneda que su cuenta, pero pidiendo más dinero del que su cuenta contiene.
- testGetDollarFromPeso: Para probar el retiro de dinero por parte de un usuario, de su cuenta en pesos, en un cajero que utiliza dólares.
- testGetPesoFromDollar: Para probar el retiro de dinero por parte de un usuario, de su cuenta en dólares, en un cajero que utiliza pesos.
- testDepositDollarFromPeso: Para probar el depósito de dinero por parte de un usuario, de su cuenta en pesos, en un cajero que utiliza dólares.
- testDepositPesoFromDollar: Para probar el depósito de dinero por parte de un usuario, de su cuenta en dólares, en un cajero que utiliza pesos.
- testTransferFromPesoToDollarInPeso: Para probar el movimiento de dinero por parte de un usuario, de su cuenta en pesos a una cuenta en dólares, en un cajero que utiliza pesos.

Pregunta 2, Sistema de archivos

Un sistema de archivos cotidiano está compuesto por archivos y capetas, que contienen archivos y/o carpetas.

Se desea implementar un sistema de archivos donde cada carpeta tenga ordenados, por nombre, sus elementos. Para ello se implementarán las clases `NFolder` y `NFile`. Por simplicidad ambas clases se representarán por un nombre y tendrán un método `getPath` que retornará un objeto de tipo `NPath` para representar su ruta en el sistema de archivos. Además, las carpetas tendrán los siguientes métodos:

- `getSize`: para obtener la cantidad de archivos (`NFile`) que posee.
- `getElem`: que dada una posición, retorna el elemento en dicha posición.
- `addElem`: para agregar un elemento.
- `removeElem`: para quitar un elemento.
- `findFile`: que busca un archivo, recursivamente, por su nombre, y lo retorna. Si no existe, retorna `null`.
- `moveElem`: para mover un elemento a otra carpeta.

Los `NPath` deberán implementar 2 métodos: `getFullPath` y `getRelativePath`, que retornan la ruta completa, como `String`, del elemento a la carpeta más arriba en el sistema de archivos y la ruta relativa del elemento a una carpeta específica, respectivamente. Para la ruta relativa, utilice la notación clásica, esto es, describa la ruta de la forma `./carpeta1/carpeta2/archivo` donde el punto representa la carpeta de la cual se pide la ruta relativa.

Para reforzar que nuestras carpetas sólo puedan contener archivos y carpetas, y que estos estén ordenados por nombre, cada carpeta contendrá internamente una lista, representada por la clase `NList` que se encargará de contener los archivos dentro de la carpeta. **Tenga presente que** `NList` **no es una clase generica, solamente maneja** `NFolder` **y** `NFile`.

Para implementar `NList` cree una lista doblemente enlazada utilizando una clase `NNode` que se encargue de manejar los conceptos de sucesor y antecesor, y contener un elemento. Usando lo anterior implemente `NList` con los siguientes métodos:

- `size`: para obtener su cantidad de elementos.

- `get`: que dada una posición, retorna el nodo en dicha posición.
- `add`: para agregar un nodo.
- `remove`: para quitar un nodo.

Para sus pruebas, cree las clases `TestFileSystem` y `TestList` que extienda de `TestCase` y escriba los siguientes métodos de prueba:

Para `TestList`

- `testAdd`: Para probar que se generan los cambios esperados al agregar un nodo.
- `testAddRemove`: Para probar que la lista se mantiene coherente al agregar y quitar un elemento.
- `testAddOrdered`: Para probar si efectivamente los elementos están ordenados dentro de la lista.

Para `TestFileSystem`

- `testAddFile`: Para probar que se generan los cambios esperados al agregar un archivo.
- `testAddFileOrdered`: Para probar si efectivamente los archivos están ordenados dentro de la lista.
- `testAddFolder`: Para probar si al agregar una carpeta a otra carpeta todo sigue siendo coherente, esto es, que los métodos `size` y `getPath` funcionan correctamente.
- `testFind`: Para ver si la búsqueda de un archivo se realiza correctamente.
- `testPath`: Para probar si la operación `getPath` funciona correctamente al introducir archivos a carpetas y al anidar carpetas, probando las operaciones de la clase `NPath`.
- `testMove`: Para ver si al mover un elemento este desaparece del lugar de origen y aparece en el de destino. Recuerde probar si los `path` se actualizan.