UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de fin de Grado en Ingeniería informática

POTENCIAL DE UN EDITOR DE TEXTO LIGERO COMO BASE DE UN IDE DE USO PERSONAL

Carlos Alberto Piñero Olanda

Dirigido por: Pedro Javier Herrera Caro

Curso 2024/2025, Convocatoria de Septiembre

POTENCIAL DE UN EDITOR DE TEXTO LIGERO COMO BASE DE UN IDE DE USO PERSONAL

Proyecto de fin de Grado en Ingeniería Informática de modalidad genérica

Realizado por: Carlos Alberto Piñero Olanda

Dirigido por: Pedro Javier herrera Caro

Fecha de lectura y defensa: 25 de Septiembre de 2025

RESUMEN

La presenta memoria documenta un proyecto fin de grado en que primero se investiga el problema de la creación de un IDE en esta época, considerando las posibilidades que ofrece el mercado, tanto de pago como gratuitas. Se investiga asimismo cuáles son las opciones disponibles para construir uno.

Una vez explicada la opción que se escoge (a partir de un editor de texto), se realiza un somero análisis del sistema para identificar sus funcionalidades, después de lo cual se hace un diseño y se comenta cuál será la metodología que se seguirá.

Las siguientes secciones tratan de la implementación de las funcionalidades y las pruebas necesarias para testear su buen funcionamiento, seguida de un apartado de conclusiones y de posibles mejoras del sistema.

Listado de palabras clave: Entorno de desarrollo integrado, editor de texto ligero, modularidad, gestión de características secundarias, *TextAdept*, *Lua*, metadatos de la apariencia visual de un fichero.

ABSTRACT

This memory documents a final degree project, where the first step is investigating the problem of the creation of an IDE in this time, considering the possibilities that market offers, both payment and free. Likewise, it’s studied which are the available options to make one.

Once the chosen option is justified (from a text editor), a rough analysis of the system is done to identify its functionalities, after which a design is made, and it’s discussed which methodology will be employed.

The following sections discuss the implementation of the functionalities and the necessary tests to check whether it works correctly, followed by a part of conclusions and possible improvements of the system.

Keyword list: Integrated Development Environment, light text editor, modularity, secondary features management, *TextAdept*, *Lua*, file visual appearance metadata.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Capítulo 1

INTRODUCCIÓN

1.1 Motivación

La siguiente memoria recoge el trabajo realizado para el PFG genérico del departamento de Ingeniería de Software y Sistemas Informáticos de la UNED, *Entorno de Desarrollo Integrado de un lenguaje de programación*.

Los Entornos de Desarrollo Integrado (IDE en sus siglas en inglés, *Integrated Development Environment*) son un tipo de aplicación de software que ofrece herramientas inteligentes para el desarrollo de software (Integrated Development Environment, 2025). Normalmente consisten en un editor de código fuente, herramientas de automatización y compiladores. Este tipo de herramientas son tremendamente populares, existiendo un gran número de IDEs, tanto comerciales como de libre distribución.

1.2 Justificación

Un IDE simplifica los rigores de la creación de software. Proporciona un entorno en el que se puede trabajar mucho más rápido para programar, automatizando tanto la edición como la invocación a referencias. También agiliza la compilación del código final. En particular, estos entornos presentan estas características (Timbó, 2025):

1. La unificación de múltiples herramientas, muchas de uso común, en una interfaz gráfica (GUI, *Graphical User Interface*).
2. Mayor rapidez a la hora de crear aplicaciones (*apps*) principalmente por la anterior razón.
3. Acabado inteligente del código (*intelligent completion*).
4. Proporcionan diagramas para manejar los flujos de trabajo (*workflows*).
5. Ayudan a la identificación de los diversos errores: código, mensajes, etcétera.

Crear un IDE es una buena manera no sólo de lograr una posición dentro del mercado si su desarrollador es una empresa, es también un modo de conocer el estado de la tecnología en varios aspectos, como se verá en el apartado correspondiente, ya que un IDE reúne muchas funcionalidades que exigen un enfoque multidisciplinar.

1.3 Objetivos

El objetivo principal de este trabajo de fin de grado es la creación de un IDE. Para ello, es necesario cumplir la siguiente lista de subobjetivos:

Obj 1. Estudiar el estado del arte del problema.

Obj 2. Gestionar ficheros, sus agrupaciones y organizar los pertenecientes a un mismo proyecto (Gestión de Proyectos).

Obj 3. Gestionar plantillas de los ficheros o proyectos a manejar.

Obj 4. Disponer de funciones de edición habituales (cortar, copiar, borrar, etc.).

Obj 5. Manejar simultáneamente ventanas múltiples (de edición, de resultados, de ficheros).

Obj 6. Compilar, separada o completamente, depurar y ejecutar, completamente o paso a paso.

Obj 7. Integrar herramientas externas (desensamblado, control de código, realización de pruebas, otros editores, etc.).

Obj 8. Funcionar en múltiples plataformas.

Obj 9. Configurar el propio entorno (directorios, aspectos del interfaz, ayuda, etc.).

1.4 Estado del arte

1.4.1 Tipos de IDEs

Fundamentalmente, hay tres criterios por los que se pueden distinguir varios tipos de IDEs (Timbó, 2025):

1. La arquitectura para la que se va a crear el programa permite hablar de IDEs multiplataforma y aquellos que no lo son. Un caso particular son los IDEs de desarrollo para móviles, ya que tienen herramientas específicas para construir aplicaciones. Como existen diversas arquitecturas de móviles, es necesario que el IDE de trabajo sea multiplataforma. Ejemplos son Xamarin, Xcode y Android Studio.
2. Los lenguajes de programación con los que se puede trabajar permiten distinguir entre IDEs monolenguaje (si se han creado específicamente para uno en concreto) e IDEs multilenguaje. La variedad de lenguajes de programación ha hecho necesario que los IDEs soporten una gran variedad de ellos. Por ejemplo, Flycheck puede trabajar con más de 40 lenguajes. Uno de los más populares es Visual Studio, continuamente actualizado. Otros son Komodo, con opciones de mejor calidad (*premium*), o NetBeans, que asimismo incluye opciones para desarrollo móvil.
3. El desarrollo de las tecnologías de comunicación y en particular de la red permite por último distinguir entre IDEs de desarrollo en escritorio, que se desarrollan en un dispositivo (ya sea porque el propio IDE no ofrezca desarrollo web o basado en la nube, o porque el equipo de desarrolladores lo prefiera así), e IDEs de desarrollo web y basados en la nube, que se caracterizan por otorgar acceso desde cualquier dispositivo. Los más conocidos de este último tipo son CodeTasty (capaz de soportar más de 40 lenguajes), Codeanywhere (que es además multiplataforma) y Cloud9 (integrado en el ecosistema de Amazon Web Services). Por último, es digno de señalar que algunos IDEs permiten tanto el desarrollo en escritorio como en línea (*on-line*).

Como se observarán en algunos de los ejemplos del siguiente apartado, estos criterios afectan a la tipología del IDE.

1.4.2 Comparativa de IDEs más utilizados

Se someten a examen los siguientes IDEs por ser los más ampliamente utilizados y por dar soporte a los lenguajes más usados en programación (Vailshery, 2024). La información descrita a continuación procede mayoritariamente de la misma fuente (Timbó, 2025), en caso contrario se proporciona la cita correspondiente:

1. **Visual Studio**: Es el más usado, es de libre distribución y funciona en Windows y Linux, pero dejó de ofrecer soporte para macOS en agosto de 2024 (¿Que ha pasado con Visual Studio

para Mac?, 2024). Ofrece soporte para los lenguajes JavaScript, Python, Visual Basic, F#, C++, HTML y CSS. Es el IDE de Microsoft, por lo que está diseñado para crear aplicaciones para la plataforma Microsoft (Vera Code, 2025). Entre sus características, se incluye:

- Un editor de código (es decir, de texto).
- Un depurador (*debugger*, en terminología inglesa).
- Herramientas visuales de diseño (*visual designer*) para crear aplicaciones, para diseñar clases, Windows Forms y datos.
- Un navegador de objetos (*object browser*) para cada proyecto.
- Herramientas de testeo.
- Comandos para facilitar la navegación por el código.

2. **Visual Studio Code**. Es un IDE desarrollado por Microsoft para Windows, Linux, macOS y navegadores web. Ofrece soporte para la depuración, resaltado de sintaxis (*syntax highlighting*), acabado inteligente de código, fragmentos reutilizables de código, refactorización de código y un control de versión embebida con Git (Lardinois, 2015).
   - Es un software de propiedad basado en una licencia del MIT: Visual Studio Code – Open Source (Dias, 2015).
   - Puede usarse con variedad de lenguajes, como C, C#, C++, Fortran, Go, Java, JavaScript, Node.js, Python, Rust y Julia (Wikipedia, Visual Studio Code, 2025).
   - Está basado en web, por lo que su versión para escritorio es ligera (What is an IDE, 2025).
3. **Eclipse**: IDE de programación en ordenadores de libre distribución. Dispone de un espacio de trabajo básico extensible mediante un sistema de complementos (*plug-ins*) para personalizar a gusto de usuario. Si bien suele usarse para programar aplicaciones en Java, puede trabajar con Erlang, C, C++, C#, Julia, Perl, PHP y Ada. Sus otras características son:
   - Una enorme librería de complementos.
   - Automatiza la creación de la documentación para las clases empleadas en las aplicaciones con una herramienta de JavaDoc.
   - Una interfaz simple, consistente en mover los objetos con ratón.
   - Herramientas visuales para la depuración de código.
   - Un sistema de apoyo al usuario, tal como ayuda.
4. **Code::Blocks**. Un IDE multiplataforma de libre distribución con soporte para múltiples lenguajes, como Java, C, C++ y Fortran. Posiblemente uno de los mejores IDEs para principiantes por el diseño de su interfaz de usuario.
   - Un editor de código con plegado de código, resaltado de sintaxis, pestañas de clases, acabado automático (*autocompletion*) de código C++ y muchas más herramientas.
   - Un depurador que permite a los desarrolladores eliminar los errores de los programas.
   - Una herramienta de diseño mediante interfaz gráfica de usuario (*graphical user interface*, GUI).
   - Facilita la migración desde otros IDEs.
5. **Qt Creator**. Es un IDE multiplataforma (Linux, Windows y macOS) que busca maximizar la experiencia de desarrollo y al mismo tiempo es una buena opción para principiantes por el

apoyo de su comunidad y su documentación. Soporta tres lenguajes principales, Python, C++ y Qt QML y otros dos gracias a la comunidad, Rust y Go. Asimismo, incluye:

- Un sofisticado editor de código.
- Capacidad de integración de sistemas de control de versión de amplio uso, como Git, Perforce y Subversion.
- Capacidad para importar proyectos o crearlos desde cero.
- Opciones de ensamblado (*build*) que permiten a los desarrolladores seguir trabajando después de cambiar de sistema operativo.
- Qt Quick Compiler, una herramienta capaz de compilar código fuente a código máquina nativo, que además incrementa tanto el rendimiento de la interfaz de usuario, como el tiempo de comienzo y además protege la propiedad intelectual y el código fuente.

6. **IDLE**. Es un entorno de desarrollo integrado para Python, que se incluye con la implementación predeterminada del lenguaje desde la versión 1.5.2b1 (Wikipedia, IDLE, 2025) . Está disponible en Windows, Linux y macOS (Python Software Foundation, 2025).
7. **Spyder**. Es otro IDE para Python, asimismo multiplataforma y de código abierto (Overview, 2018). Está específicamente orientado a la programación para fines científicos. Entre otras, sus características son (Wikipedia, Spyder (software), 2025):
   - Resaltado de sintaxis, examen dinámico del tipo de las variables (ya que Python no es un lenguaje tipado) y acabado automático del código.
   - Puede soportar múltiples consolas iPython.
   - Proporciona una GUI para explorar y editar variables.
   - Incluye un panel de ayuda con información de clases y métodos.
   - Dispone de un depurador enlazado a una herramienta llamada Ipdb, para la ejecución paso a paso.
   - Realiza análisis estático de código, potenciado por Pylint.
   - Proporciona soporte de proyectos, permitiendo el trabajo en varios simultáneamente.
   - Está dotado de un explorador inserto de archivos y proyectos (*built-in file explorer*).
   - Incluye un historial para grabar todo comando de usuario en cada consola.
   - **BlueJ**. Un IDE gratuito de Java (y con soporte de Strider). Multiplataforma, pensado para principiantes (al tener una interfaz más simple y pequeña), interactivo y que permite inspeccionar los objetos y sus métodos, y tiene 15 años de historia y una gran comunidad.
8. **Theia**. Otro IDE de la Fundación Eclipse (creadores del IDE del mismo nombre) y por ello comparte sus características de flexibilidad, extensibilidad y ser multiplataforma e incluso disponible en la nube. Además, lo más importante de señalar es que la Fundación Eclipse protege a los desarrolladores y contribuyentes de Theia contra decisiones de ventas unilaterales. Entre sus características, destacan:
   - Una terminal integrada que se reconecta cuando recarga el navegador, manteniendo el historial completo.
   - Un esquema flexible compuesto de artilugios (*widgets*) ligeros y puertos que se pueden mover con el ratón (*draggable docks*).
   - La capacidad de alojar extensiones de VS Code.

- Acceso completo a la terminal.

9. **RStudio**. Es un IDE para el lenguaje de programación R, dedicado a la computación estadística y representaciones gráficas (Wikipedia, RStudio, 2025). Tiene entre sus características:
   - Está presente en dos formatos, Rstudio Desktop y RStudio Server. Respectivamente, un IDE de escritorio y otro basado en navegador web.
   - Soporta análisis reproducibles con documentos explicativos con R Markdown para mezclar texto con código en R, Python, Julia y otros.

La Tabla 1-1 resume las principales características de las herramientas descritas anteriormente.

Tabla 1-1 IDEs más populares y sus características

Nombre	Lenguajes	Plataformas	Libre distribución	Características
Visual Studio	JavaScript Python Visual Basic F# C++ HTML CSS	Windows Linux	Sí	Editor de código Depurador Guías visuales Pestaña de objetos Herramientas de pruebas Comandos de navegación
Eclipse	Java C C++ C# Julia Perl PHP	Windows Mac Linux	Sí	Librería de complementos de gran tamaño Documentación automática de clases Una interfaz de mover objetos con el ratón Depuración visual Ayuda para el usuario
Visual Studio Code	C C# C++ Fortran Java JavaScript Node.js Python Rust Julia	Windows Mac Linux Basado en web	No	Basado en VS Code – Open Source Versión de escritorio ligera
Code::Blocks	Java C C++ Fortran	Windows Mac Linux	Sí	Editor de código con: Plegado de código Resaltado de sintaxis Pestañas de clases Aacabado automático de código C++ Depurador GUI para diseño Migración desde otros IDEs
Qt Creator	C++ Qt QML Python Rust Go	Windows Mac Linux	Sí	Sofisticado editor de código Integración con sistemas de control de versión Capacidad de importar proyectos Ensamblado con opciones de SO Qt Quick Compiler con protección de propiedad intelectual

IDLE	Python	Windows Mac Linux	Sí	Se incluye predeterminadamente con el lenguaje Python
Spyder	Python C C++	Windows Mac Linux	Sí	Resaltado de sintaxis Examina el tipo Acabado automático del código Soporta múltiples consolas GUI para examinar variables Panel de ayuda Depurador con ejecución paso a paso Análisis estático de código Soporte de proyectos Historial de comandos de usuario
BlueJ	Java Stride	Windows Mac Linux	Sí	Adecuado para principiantes Interactivo 15 años de historia
Theia	JavaScript Python Java +60 lenguajes	Windows Mac Linux Basado en nube	Sí	Terminal integrada que reconecta con la recarga de la pestaña Esquema flexible Artilugios ligeros Puertos movibles con el ratón Alojamiento de extensiones de VS Code Acceso completo a la terminal
RStudio	R	Windows Mac Linux Web	Sí	Dos formatos, escritorio y navegador web Soporta documentos explicativos que mezclan texto y fragmentos de código

Del anterior listado se destacan los siguientes hechos:

1. Todos los IDEs ofrecen editores de texto.
2. Todos los editores ofrecen compiladores / intérpretes.
3. Todos los editores ejecutan el código o el programa compilado y muestran los resultados.
4. Todos los IDEs cuentan con herramientas visuales para sus funcionalidades. Al menos todos cuentan con botones para ejecutar y compilar, mientras que algunos cuentan con mapas de proyectos.
5. La mayoría de IDEs resaltan la gramática con indicadores visuales: coloreado de las palabras reservadas, uso de la indentación para indicar bloques funcionales, en algunos casos incluso se permite plegar bloques funcionales.
6. La mayoría cuenta con un depurador que permite detectar y localizar errores.
7. La mayoría de ellos compila al menos dos lenguajes. Los IDEs monolenguajes son escasos, los únicos de la lista lo son sólo para Python y R, lenguajes de uso científico. Algunos de estos lenguajes aparecen frecuentemente: C, C#, C++, Java y Python.
8. La mayoría de IDEs permite configurar el entorno.

Por lo tanto, teniendo en cuenta además los objetivos, el IDE dispondrá de las características que se detallan en la siguiente sección.

1.4.3 Características deseables en el IDE del proyecto

El anterior listado confirma una afirmación del apartado de justificación: los IDEs presentan una serie de servicios frecuentes. Por ello, se usarán herramientas que los proporcionen (Fowler, 2023):

1. Editar texto.
2. Compilar el código.
3. Ejecutar el código.

Aparte, las siguientes características forman parte de los requisitos del proyecto:

1. Gestión de proyectos y ficheros.
2. Gestión de plantillas.
3. Unión visual en un interfaz.
4. Integración de desemblasado, control de código (tal como resaltar las partes de la gramática).
5. Configuración del propio entorno.

Además, es deseable que el IDE tenga potencial tanto multiplataforma como multilenguaje, ya que, además de ser la tendencia actual entre los IDEs más señalados, esto aumenta la vida potencial del IDE al evitar que sea dependiente del uso de una tecnología concreta.

En este caso, el IDE trabajará con los lenguajes de programación C y Java, ya que son bien conocidos, y debería poder funcionar con Windows y con Ubuntu, al menos.

En el siguiente apartado se discuten las opciones valoradas para implementar las características anteriores y se elegirá una de las dos.

1.4.4 Opciones para construir un IDE

Se han valorado dos opciones para construir el IDE: mediante el uso únicamente de complementos y a partir de un editor de texto. Toda la información descrita a continuación procede de la misma fuente (Fowler, 2023) mientras no se indique lo contrario.

1.4.4.1 Mediante el uso únicamente de complementos (plug-ins)

Las características descritas en el anterior apartado son factibles de implementarse con complementos, por lo que una solución a este problema es hacer de todo un complemento. Esto implica:

1. Empezar con un entorno de trabajo como si fuera el complemento de otro programa.
2. Desarrollar entonces un complemento de edición.
3. Añadir un complemento de compilación.
4. Desarrollar complemento de visualización de salida de programa para estudiar los resultados de compilación.
5. Escribir complemento para el análisis sintáctico del código fuente.
6. Un complemento que resalte la sintaxis con colores (*syntax colour highlighting*).
7. Escribir un complemento de gestión de archivos de proyecto.
8. El siguiente paso es desarrollar un complemento de gestión de ensamblado de proyectos del lenguaje de elección.

Esta alternativa es trabajosa, pues supone tener que empezar desde el principio. Entre los IDEs que hemos listado anteriormente, usan esta aproximación:

1. **Eclipse**: «Eclipse usa complementos para proveer de toda la funcionalidad dentro y encima del sistema en tiempo de ejecución» (Eclipse (software), 2025).

2. **Visual Studio**: «Visual Studio no soporta ningún lenguaje de programación, solución o herramienta intrínsecamente; en su lugar, permite la complementación (*plugging*) de la funcionalidad codificada como VSPackage», (Visual Studio, 2025).
3. **Code::Blocks**: «Mediante una arquitectura de complementos, sus capacidades y características están definidas por los que sean proporcionados» (Code::Blocks, 2025).

1.4.4.2 A partir de un editor de texto

Otra alternativa es partir de un editor de texto como Notepad++, que puede recibir complementos. Como Notepad++ sólo funciona en entornos de Windows (Ho, 2025), esto podría ser un impedimento para que nuestro IDE funcione en Ubuntu, por lo que es más aconsejable usar otro.

La Tabla 1-2 resume (List of text editors, 2025) las características de varios editores de texto. Hemos limitado la selección a editores no exclusivos de un sistema operativo de uso restringido y que al menos tengan interfaz de soporte gráfico. La información mostrada procede mayoritariamente de la anterior fuente, incluyendo las demás referencias en la propia tabla.

Tabla 1-2 Lista de editores de texto

Nombre	Tipo de interfaz	Descripción	Licencia
Elvis	Gráfica y de texto	Un clon de vi/ex con nuevos comandos y características como un compilador que puede analizar errores y moverse a la fuente del error[1] Multiplataforma, incluyendo Unix, Linux y Microsoft Windows[1]	CIArtistic
GNU Emacs	Gráfica y de texto	Una bifurcación existente desde hace mucho del popular editor de Emacs Funciona en sistemas operativos semejantes a Unix (GNU, Linux, macOS, Windows)[2] Compilación nativa de Lisp[2]	GPL-3.0-or-later
XEmacs	Gráfica y de texto	Otra bifurcación existente desde hace mucho del popular editor de Emacs Multiplataforma para GNU, Linux, Windows, maOS, BSDs y más[3]	GPL-2.0-or-later
Textadept	Gráfica y de texto	Editor modular y multiplataforma, escrito en C y Lua, que usa Scintilla Puede compilar y ejecutar programas[4]	MIT
vile (vi like Emacs)	Gráfica y de texto	Un editor tipo vi que mantiene el plantel de comandos de vi y añade nuevas características: ventanas y buffers múltiples, opción de deshacer ilimitado, coloreado, capacidades de expansión de código. Multiplataforma[5]	GPL-2.0-only
vim	Gráfica y de texto	Un clon basado en ideas del editor vi Existente para Unix, Linux, Windows NT, MS_DOS, macOS, iOS, Android[6]	Vim

Acme	Gráfica	Una Interfaz de Usuario para Programadores por Rob Pike[7] Sistemas operativos similares a Unix, Windows[7]	MIT
Alphatk	Gráfica	Windows, Unix y Mac OS X[8] Provee de un anfitrión de infraestructuras para comunicarse con compiladores[8]	Propietario
Apache OpenOffice Writer	Gráfica	Procesador de documentos y editor de textos de la Apache Openffice Suite, basado en la suite de StarOffice Linux, macOS, Windows[9]	Apache-2.0
Arachnophilia	Gráfica	Editor de código fuente que es el sucesor de otro editor de HTML, WebThing Está escrito en Java, por lo que es compatible con cualquier SO[10]	Libre software
Atom	Gráfica	Un editor modular de propósito general construido usando HTML, CSS y JavaScript, sobre Chromium y Node.js macOS (a parit de 10.9), Windows 7 (y posterior), y Linux Está descontinuado[11]	MIT
Bluefish	Gráfica	Editor de texto con características para el desarrollo web Linux[12], macOS[13], Windows[14]	GPL-2.0-or-later
Crimson Editor	Gráfica	Editor de código fuente Microsoft Windows[15] Se caracteriza por la integración con compiladores[15]	Libre software
gedit	Gráfica	Antiguo editor por defecto para GNOME hasta GNOME 42 Linux, macOS, Windows[16] De pago para Windows[17]	GPL-2.0-or-later
GNOME Text Editor	Gráfica	Editor por defecto para GNOME desde GNOME 42 en adelante Linux[18]	GPL-3.0-or-later
Notepad (Bloc de notas)	Gráfica	Editor por defecto de Windows	Propietario
Notepad++	Gráfica	Editor de texto con pestañas	GPL-3.0-or-later
UltraEdit	Gráfica	Editor de texto y código fuente con resaltado de sintaxis Linux, macOS, Windows[19] Soporte nativo de compiladores[19]	Propietario

1. (Elvis (text editor), 2025)
2. (GNU Emacs, 2025)

3. (XEmacs, 2025)
4. (Eid, Textadept, 2007)
5. (vile (text editor), 2025)
6. (Vim (text editor), 2025)
7. (Acme (text editor), 2025)
8. (Alphatk, 2025)
9. (Apache OpenOffice, 2025)
10. (The Most Common Question about Arachnophilia, 2020)
11. (GitHub Staff, 2022)
12. (Hill, Helmke, & Burger, 2009)
13. (Bluefish for Mac, 2017)
14. (Brockmeier, 2010)
15. (Crimson Editor, 2025)
16. (gedit, a text editor, 2025)
17. (gedit text editor, 2025)
18. (Cooper, 2022)
19. (UltraEdit, 2025)

Por tanto, algunos editores de texto no sólo incluyen la característica fundamental de ser capaces de editar código, sino que ya añaden de serie el resaltado de la sintaxis con colores, la estructuración de textos o incluso la capacidad de compilar o interpretar código. Aun así, no son considerados formalmente IDEs, ni siquiera ligeros, por la simple razón de no ser tan extensibles o carecer de un explorador de proyectos muy detallado. Esta alternativa se alinea bien con los propósitos del presente proyecto, pues ahorra parte del trabajo y coincide con los objetivos, ya que varios de estos editores pueden usarse en Windows y Ubuntu y con dos lenguajes tan extendidos como C y Java.

De la lista de IDEs mostrada en la Tabla 1-1, siguen este enfoque los siguientes:

1. **IDLE**: tiene entre sus más llamativas características un editor de texto de ventanas múltiples con resaltado de sintaxis, acabado automático e indentación inteligente entre otras características (Python Software Foundation, 2025).
2. **Spyder**: El editor de texto es la primera característica de este IDE (Overview, 2018).

1.4.4.3 Opción elegida

De en entre estas opciones, la que supone menor carga de trabajo es la segunda, ya que existe un gran número de editores de texto cuyas licencias autorizan su uso y manipulación para dotarlos de nuevas funcionalidades.

En particular, de las opciones mostradas se va a usar *TextAdept* por las razones siguientes (List of text editors, 2025) (Eid, Textadept, 2007) (Eid, Textadept 12.7 Manual, 2025) (Eid, Textadept 12.7 API Documentation, 2025):

1. Tiene licencia del MIT, lo que permite su uso y manipulación.
2. Parte con varias de las características requeridas: compilación e interpretación de código, resaltado de colores y la estructuración de código.
3. Tiene hasta dos tutoriales en Internet.

4. Su estructura de módulos es muy sencilla de entender.
5. Usa *Lua*, un lenguaje de programación con similitudes a Python, en sus módulos para obtener gran extensibilidad.
6. Permite abrir los propios ficheros de su configuración y manipularlos fácilmente.

Por lo tanto, el IDE del proyecto será una extensión del editor de texto *TextAdept*, complementando los ficheros de este programa con aquellas características ausentes.

1.5 Organización del documento

Este documento se compone de las siguientes partes:

1. Introducción. Es la presente sección, en la que se ha presentado el problema, la motivación y la justificación del trabajo. A continuación, se ha descrito su estado del arte en cuatro apartados: estudio del tipo de IDEs, análisis de IDEs muy utilizados, qué funcionalidades debe presentar el creado y estudio de las posibilidades para construirlos, con la decisión de cuál es la más adecuada.

2. Análisis del sistema. Es un estudio de los componentes del sistema que debe resolver el problema. Primero se presenta un análisis general y, luego, se concreta para la elección realizada.

3. Diseño del sistema. Recoge la propuesta surgida del análisis para resolver el problema.

4. Implementación. Describe con qué tecnologías se ha creado el sistema y cómo se han usado.

5. Pruebas. Un listado de las pruebas que demuestran el éxito del sistema.

6. Conclusiones y trabajo futuro. Aparte de extraer las principales conclusiones fruto del trabajo desarrollado, se establecen las líneas en las que se podría expandir el sistema.

Bibliografía. Incluye las fuentes usadas tanto para elaborar el propio IDE (tecnologías y motivos por los que usarlas) como para documentar esta memoria.

Glosario con los términos más importantes del presente documento y otros cuyo significado facilita su comprensión.

Anexos. Son tres documentos:

1. El manual de usuario. Documento destinado al usuario del IDE.
2. El manual de instalación. Cómo instalarlo en cada sistema.
3. Código fuente. El propio instalador para usar el IDE.

Capítulo 2

ANÁLISIS DEL SISTEMA

2.1 Componentes del IDE y su interrelación

El siguiente esquema (What is an IDE, 2025) muestra los componentes del sistema que se pretende desarrollar en este proyecto:

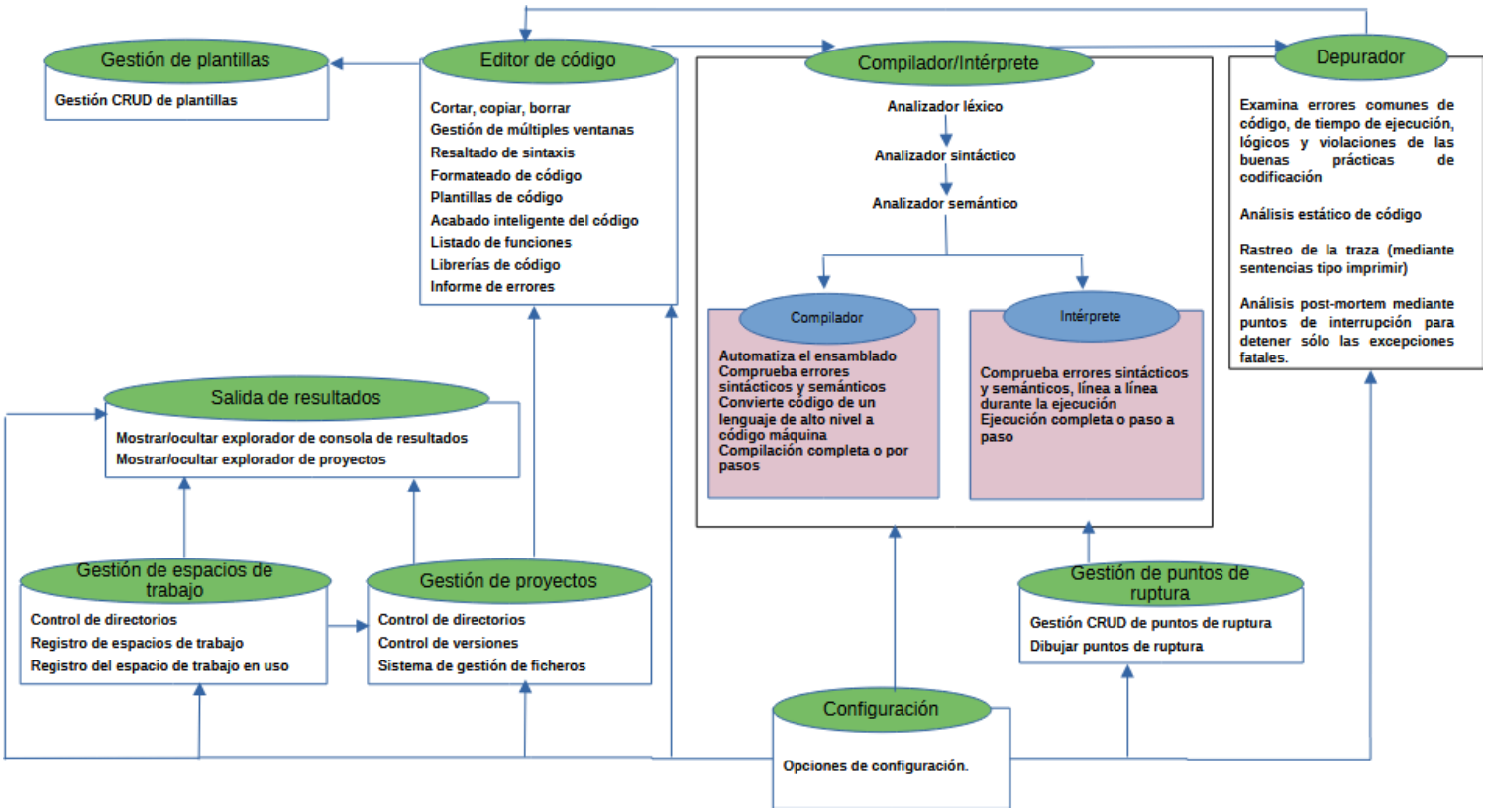


Figura 2-1. Esquema de los componentes de un IDE

El sistema se debe componer de los siguientes módulos:

1. El editor de texto.
2. El compilador y el intérprete, que realizan sucesivamente los análisis pertinentes según la tipología del lenguaje. En ambos casos, se ejecuta además el código.

3. El depurador que detecta los posibles errores habidos durante la compilación y ejecución.

El anterior esquema se debe complementar con los siguientes elementos:

1. Una pantalla de visualización de resultados, tanto para una correcta ejecución del código como para ofrecer información de los posibles errores.
2. Sistemas de gestión de espacios de trabajo, de proyectos y sus ficheros, de plantillas y de puntos de ruptura para la ejecución paso a paso.
3. Opciones de configuración, tanto de los proyectos como para ofrecer extensibilidad.
4. El marco de trabajo que dé unión visual a los anteriores componentes.

Como se ha expuesto en el capítulo anterior, un editor de texto es perfecto como punto de partida porque, aparte de sí mismo, ofrece un sistema de gestión de ficheros y hasta opciones de configuración, si bien ambos han de extenderse para considerar un proyecto de software. Así, el anterior esquema debe redefinirse tal y como se ilustra en la Figura 2-2.

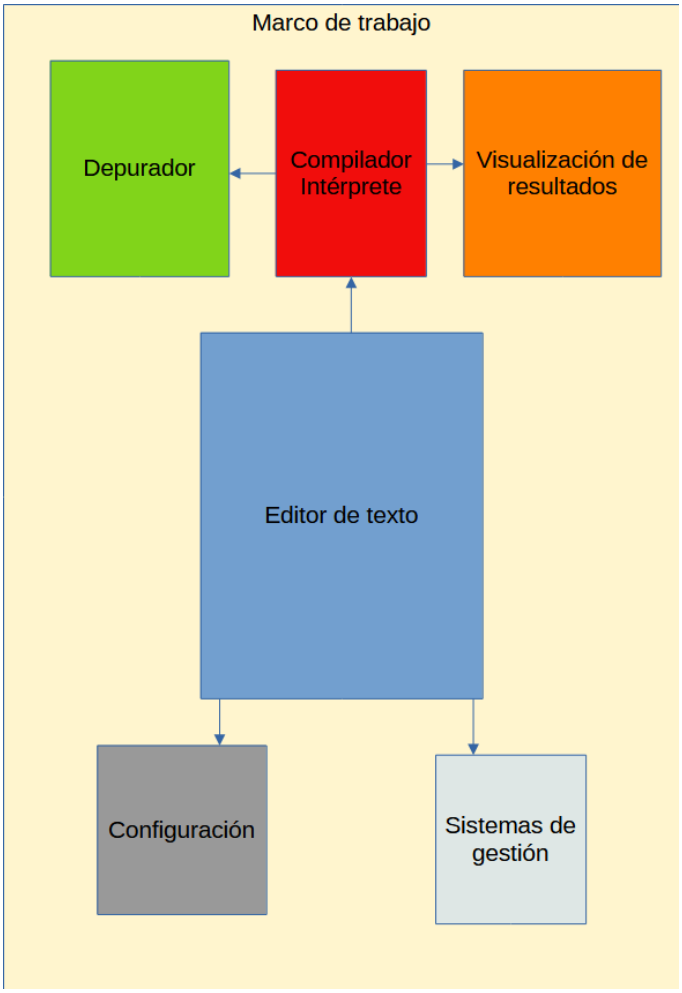


Figura 2-2. Esquema de los componentes del IDE del proyecto

El editor de texto es la parte central del IDE. Sobre este se montan mediante complementos el compilador/intérprete, la configuración y los sistemas de gestión. El compilador/intérprete se une a su vez al depurador y a la visualización de resultados, pero además en la práctica es parte del propio editor. El marco de trabajo les da unidad a todas estas funcionalidades y puede construirse con el propio editor gracias a la característica de presentar múltiples ventanas.

2.2 Planificación

Este proyecto se dividió en las siguientes fases:

Fase 1. Redacción de la motivación y justificación, estudio del estado del arte y decisión de la mejor alternativa.

Fase 2. Análisis del sistema y sus componentes teóricos, así como de sus casos de uso.

Fase 3. Diseño del sistema y especificación de componentes hardware y software mínimos.

Fase 4. Implementación.

Fase 5. Pruebas.

Fase 6. Redacción final de la memoria.

La figura 2-3 ilustra la planificación. Nótese que varias de las fases se simultanearon, ya que la metodología empleada lo favorecía, o incluso se llevaron a cabo en plazos discontinuos, ya que la fase 6, la elaboración de la memoria, era aconsejable empezarla cuanto antes.

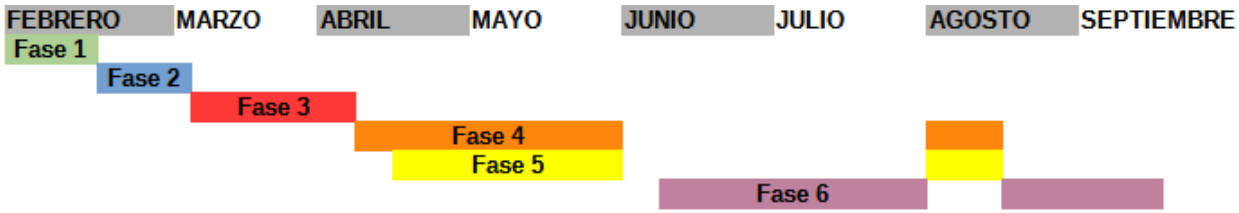


Figura 2-3. Planificación

2.3 Identificación de requisitos funcionales y no funcionales

Requisitos funcionales:

1. El sistema debe ser capaz de gestionar ficheros con las funciones de edición clásicas en un editor de texto, tales como copiar, cortar, eliminar, etc.

   Es decir, debe actuar como dos tipos de software simultáneamente: un editor de texto y un gestor de ficheros. Es parte de las características de un IDE.

2. El sistema debe ser capaz de reconocer qué ficheros son parte de un proyecto.

   A cada fichero le corresponde exclusivamente un proyecto, lo que se caracteriza mediante el directorio donde se aloja.

3. El sistema debe ser capaz de reconocer el lenguaje en que están escritos los códigos por detalles como la extensión o la propia gramática.

   La compilación o interpretación es un proceso automático. Los ficheros escritos en un lenguaje de programación suelen acabar en una extensión específica (.c, por ejemplo) que identifica su lenguaje. En otros casos, la gramática de cada lenguaje es específica del mismo y no hay dos lenguajes con la misma gramática, pues entonces uno sería superfluo, aunque el anterior sistema es más práctico. Así, el sistema puede aplicar automáticamente los analizadores correspondientes.

4. El sistema debe presentar opciones de compilación y ejecución, tales como: hacerlo totalmente, por partes, paso a paso.

   La complejidad de los programas actuales exige tener varias opciones tanto para la compilación como la ejecución, como se explica mejor en la siguiente sección, Casos de uso.

5. Debe tener opciones de configuración y de integración de herramientas externas.

   Tener opciones de configuración se justifica por el hecho de que no todos los usuarios necesitan el mismo tipo de entorno, por lo que dar la posibilidad de modificar detalles es una buena manera de hacer el resultado final más atractivo a un mayor número de usuarios.

   Respecto a la integración, por completo que resulte el sistema final siempre pueden aparecer nuevas herramientas que ofrezcan funcionalidades interesantes para algunos usuarios. Para ellos, la posibilidad de extensiones es siempre una buena idea.

Requisitos no funcionales:

1. El sistema debe ser independiente del sistema operativo para que sea multiplataforma.

   Como se ha comentado en la Introducción, es deseable que una tecnología sea multiplataforma para así lograr que su vida potencial sea mayor, al reducir su dependencia de otras tecnologías, lo que empieza por el propio sistema operativo.

2. El sistema debe ser capaz de ofrecer opciones de compilación en ficheros separados y de hacer marcado de puntos de ruptura (*breakpoints*) para la ejecución paso a paso.

   Sin estas opciones, sería muy difícil o irrealizable poder seleccionar estas funcionalidades, ya que en el primer caso podría ser pesado escribir el nombre de todos los ficheros manualmente y en el segundo los puntos de ruptura marcan el límite entre los pasos de este tipo de ejecución.

3. El sistema debe tener un sistema de información que le permita registrar tanto los espacios de trabajo, como los proyectos contenidos en estos, y a su vez los ficheros de cada proyecto.

   Para organizar eficientemente los proyectos existentes y sus archivos, es imperativa la estructuración de estos tres elementos (espacios de trabajo, proyectos y ficheros) para facilitarle al usuario un listado de qué existe en el sistema, conocido formalmente como explorador de proyectos.

2.4 Casos de uso

El diagrama mostrado en la Figura 2-4 resume los casos de uso principales y cuáles de los actores considerados están implicados. Es importante reparar que, para lo que se refiere al caso de uso, no importa si existen varios subcasos en los que no intervengan todos los actores, ya que se va a desglosar cada uno de ellos. Los actores se describen a continuación:

1. Usuario: Identifica a la persona que trabajará con el sistema final, esto es el usuario del propio IDE.
2. Sistema: Denota la parte del software que engloba los demás componentes, interacciona directamente con el usuario y realiza algunas tareas en las que no se precisa algún componente más específico, como la gestión de espacios de trabajo.
3. Editor de texto: Denota la parte del software que realiza las tareas relativas a la propia edición de texto, lo que conlleva la gestión de plantillas pues son un tipo de ficheros.
4. Gestión de proyectos y ficheros: Denota la parte del software que se ocupa de la gestión de proyectos, así como de su registro dentro del espacio de trabajo en uso.
5. Salida de resultados: Denota la ventana donde se muestran los resultados de las operaciones realizadas en el sistema.
6. Configuración: Denota la parte del software encargada de la propia configuración y las posibles extensiones que Usuario quiera incluir.
7. Compilador/Intérprete: Denota el software externo al sistema que realiza la compilación e interpretación.

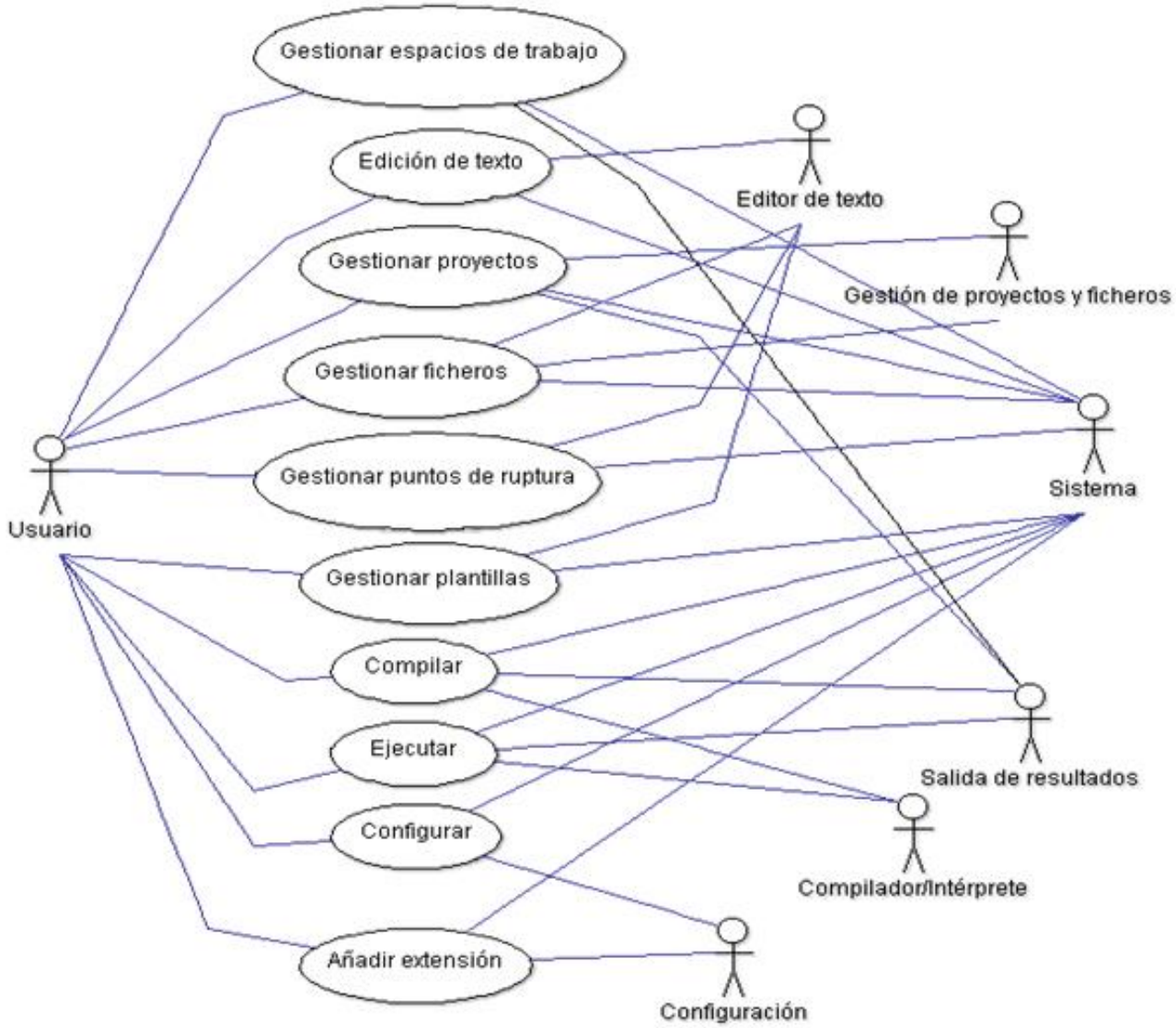


Figura 2-4. Casos del uso principales del sistema

Varios de estos casos de uso se descompondrán en otros secundarios, cuyas descripciones se listan a continuación. Se repite que es importante tener en cuenta que la participación de los actores expuestos en el anterior diagrama no tiene que cumplirse para todos los casos secundarios derivados.

2.4.1 Gestionar espacios de trabajo

Como se ha indicado en la sección de los requisitos funcionales, la gestión de proyectos supone la existencia de al menos un espacio de trabajo, un espacio virtual donde se registran los proyectos para facilitar su gestión e identificación por parte del usuario.

Por ello, es necesario que haya casos de uso para la gestión de estos espacios, en particular las cuatro de la gestión CRUD: crearlos, abrirlos, actualizarlos y eliminarlos, ya que su actualización tiene lugar como respuesta a la gestión de proyectos y de ficheros. La figura 2-5 desglosa estos casos de uso secundarios:

Figura 2-5. Casos de uso secundarios de Gestionar espacios de trabajo

2.4.1.1 Crear espacio de trabajo

El primer paso para tener múltiples espacios de trabajo es crearlos. Este caso de uso se limita a los actores del Usuario y el Sistema, ya que se limita a la creación de un fichero que almacena el nombre y la creación de un directorio del mismo nombre.

Tabla 2-1 Flujo normal de *Crear espacio de trabajo*

Usuario	Sistema
1. Selecciona crear un espacio de trabajo	2. Pide un nombre
3. Escribe un nombre	4. Comprueba que el nombre sea único
	5. Comunica la creación del espacio de trabajo
	6. Aparece el directorio del nuevo espacio de trabajo en el directorio

Flujos alternativos:

3. El usuario cancela la acción.

4. El nombre ya está en uso en espacio de trabajo.

2.4.1.2 Eliminar espacio de trabajo

La creación conlleva la posibilidad de la eliminación. En caso de que un espacio de trabajo no sea ya de utilidad, el Usuario puede solicitarle al Sistema su eliminación, es decir eliminar el registro de su nombre. Es importante reparar en que los propios proyectos no son alterados por esta funcionalidad, por lo que no intervienen otros actores.

Tabla 2-2 Flujo normal de *Eliminar espacio de trabajo*

Usuario	Sistema
1. Selecciona eliminar un espacio de trabajo	2. Muestra el listado de espacios de trabajo
3. Elige un espacio de trabajo	4. Elimina del fichero el nombre del espacio de trabajo
	5. Comunica la eliminación del espacio de trabajo

Flujos alternativos:

2. No hay espacios de trabajo excepto el asignado por defecto.

3. El usuario cancela la acción.

2.4.1.3 Abrir espacio de trabajo

Cuando existan varios espacios de trabajo, será necesario poder seleccionar en cuál se quiere trabajar para guardar los proyectos. Como esta operación no afecta a los proyectos ni a sus ficheros, sólo intervienen el Usuario y el Sistema.

Tabla 2-3 Flujo normal de *Abrir espacio de trabajo*

Usuario	Sistema
1. Selecciona abrir un espacio de trabajo	2. Muestra el listado de espacios de trabajo
3. Elige un espacio de trabajo	4. Cambia el registro del espacio de trabajo en uso
	5. Indica el nombre del nuevo espacio de trabajo

Flujos alternativos:

3. El usuario cancela la acción.

4. El espacio elegido era el mismo que el que ya existía. A efectos prácticos, no cambia nada.

2.4.1.4 Actualizar espacio de trabajo

Esta operación es una de aquellas en que no interviene el Usuario. Cada vez que otros casos de uso alteren el número de proyectos, se hace necesario actualizar el fichero correspondiente al explorador de proyectos. Sólo interviene, pues, Sistema.

Tabla 2-4 Flujo normal de *Actualizar espacio de rabajo*

Sistema
1. Detecta un cambio en el número de proyectos o en sus ficheros
2. Actualiza el fichero correspondiente del espacio de trabajo en uso

2.4.2 Edición de texto

Este caso de uso es sin duda elemental, pero aun así es necesario tenerlo en cuenta ya que es una de las funcionalidades básicas de un IDE. La más elemental de las funcionalidades implícitas es la propia manipulación de textos, ya que las operaciones de creación, recuperación y almacenamiento se podrían ejecutar con una gestión de ficheros genérica, pero el hecho de usar *TextAdept* ya otorga esas operaciones como parte de este caso de uso secundario.

La figura 2-6 desglosa los casos de uso secundarios.

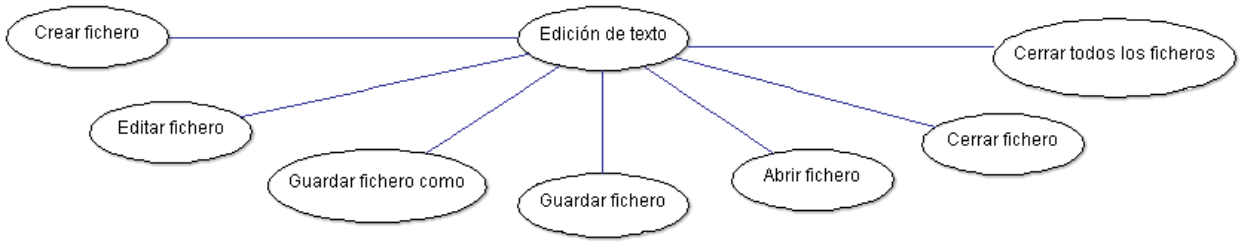


Figura 2-6. Casos de uso secundarios de Edición de texto

2.4.2.1 Crear fichero

Crear un fichero de texto es la primera funcionalidad de cualquier editor de texto, por lo que este sistema debe ser capaz de ejecutarla. Los únicos actores son el Usuario, el Sistema y el Editor de texto.

Tabla 2-5 Flujo normal de *Crear fichero*

Usuario	Sistema	Editor de texto
1. Selecciona crear un fichero	2. Solicita al editor de texto un nuevo fichero	3. Abre una nueva ventana

2.4.2.2 Editar fichero

Como un IDE debe ser entre otras cosas un editor de texto, supone que puede manipular la información contenida en un fichero de texto. Por ello, los únicos dos actores son Usuario y Editor de texto.

Tabla 2-6 Flujo normal de *Editar fichero*

Usuario	Editor de texto
1. Selecciona un fichero abierto	
2. Empieza a hacer cambios	3. Registra los cambios

Flujos alternativos:

1. No hay ningún fichero abierto.

2.4.2.3 Guardar fichero como

El siguiente paso lógico de crear y editar un fichero es guardarlo en el sistema. Intervienen los tres mismos actores de *Crear fichero*.

Tabla 2-7 Flujo normal de *Guardar fichero como*

Usuario	Sistema	Editor de texto
1. Selecciona guardar fichero como	2. Abre una ventana del gestor de ficheros del sistema operativo, pidiendo una ruta y un nombre	
3. Elige un directorio y escribe un nombre	4. Crea el fichero del nombre escrito dentro del directorio elegido	5. Actualiza el nombre del fichero

Flujos alternativos:

1. No hay ningún fichero abierto.

3. El usuario cancela la operación.

4. Ya existe un fichero con el nombre indicado en el directorio, por lo que pregunta si quiere sobreescribirlo.

2.4.2.4 Guardar fichero

Este caso de uso se diferencia del anterior en que consiste en guardar los cambios en un fichero que ya debería tener nombre y ruta, por lo que se limita a actualizarlo. Siguen interviniendo los mismos actores.

Tabla 2-8 Flujo normal de *Guardar fichero*

Usuario	Sistema	Editor de texto
1. Selecciona guardar fichero	2. Sobreescribe el fichero con los cambios	3. Borra la marca de pendiente de guardar en el nombre del fichero

Flujos alternativos:

1. No hay ningún fichero abierto.

3. El fichero aún no estaba guardado, por lo que el caso de uso continúa por el punto 2 de *Guardar fichero como*, siguiendo sus flujos.

2.4.2.5 Abrir fichero

Una vez guardado un fichero, el usuario tendrá la necesidad de abrirlo cuando no esté abierto. Así que este caso de uso es fundamental para un editor de texto, con exactamente los tres mismos actores.

Tabla 2-9 Flujo normal de *Abrir fichero*

Usuario	Sistema	Editor de texto
1. Selecciona abrir fichero	2. Muestra el navegador de ficheros del sistema operativo	
3. Selecciona un fichero	4. Transmite al Editor de texto la orden de abrirlo	5. Abre una ventana

Flujos alternativos:

3. Cancela la acción, ya sea por voluntad propia o porque no haya ficheros de texto guardados.

2.4.2.6 Cerrar fichero

Este caso de uso no es estrictamente necesario, pero lo incluía de base *TextAdept*. Desde el punto de vista funcional, sin embargo, es el paso lógico entre los casos de uso de guardar ficheros y el de abrirlo. Como antes, los tres mismos actores.

Tabla 2-10 Flujo normal de *Cerrar fichero*

Usuario	Sistema	Editor de texto
1. Selecciona cerrar fichero	2. Transmite al Editor de texto la orden de cerrar el fichero cuya ventana esté activa.	3. Cierra la ventana del fichero si ya estaba guardado y acaba. Si no, pasa a 4
5. Acepta guardarlo y continúa por el punto 2 de *Guardar fichero como*	4. Pregunta si quiere guardar primero el fichero	

Flujos alternativos:

1. No hay ningún fichero abierto.

5. Elige no guardarlo.

2.4.2.7 Cerrar todos los ficheros

El caso generalizado del anterior. Lógicamente, intervienen los mismos actores.

Tabla 2-11 Flujo normal de *Cerrar todos los ficheros*

Usuario	Sistema	Editor de texto
1. Selecciona cerrar todos los ficheros	2. Transmite al Editor de texto la orden de cerrar todos los ficheros	3. Si quedan ventanas abiertas, cierra la última activa si el correspondiente fichero ya estaba guardado y continúa. Si no hay más ventanas, acaba. Si no estaba guardado, pasa a 4.
5. Acepta guardarlo y continúa por el punto 2 de *Guardar fichero como*, volviendo a 3 cuando acaba.	4. Pregunta si quiere guardar primero el fichero	

Flujos alternativos:

1. No hay ningún fichero abierto.

5. Elige no guardarlo.

2.4.3 Gestionar proyectos

Esta es una de las funcionalidades definidas como más características y deseadas de un IDE: la capacidad de crear proyectos, es decir directorios que albergarán ficheros sin tener que crear al menos uno de estos primero fuera del propio sistema, y asimismo de eliminarlos de una vez. Otra funcionalidad es importarlos cuando el proyecto ya existe en el sistema. Por último, el explorador de texto es el beneficiario de las operaciones de actualización.

La figura 2.7 desglosa los casos de uso secundarios de esta funcionalidad.

Figura 2-7. Casos de uso secundarios de Gestionar proyectos

2.4.3.1 Iniciar proyecto

Esta operación es una de las dos maneras en que se implementa la operación crear de la gestión CRUD. Aparte del Usuario y el Sistema, se considera que interviene el actor Gestión de proyectos y ficheros.

Tabla 2-12 Flujo normal de *Iniciar proyecto*

Usuario	Sistema	Gestión de proyectos y ficheros
1. Selecciona iniciar un proyecto	2. Pide un nombre	
3. Escribe un nombre		
		4. Comprueba si el nombre está en uso
	6. Comunica la creación del proyecto	5. Registra el nombre y crea la ruta dentro del espacio de trabajo, creando un directorio con un repositorio
	7. Aparece el nuevo proyecto en el explorador de proyectos	

Flujos alternativos:

3. El usuario cancela la acción.

5. El nombre o la ruta más nombre está en uso.

5. El nombre contiene caracteres no permitidos.

2.4.3.2 Eliminar proyecto

La operación de eliminar es el siguiente paso lógico de la creación, pero en este caso existe la posibilidad de incluso eliminar el directorio principal del proyecto con todo lo que contiene si el usuario así lo elige. Obviamente, es muy arriesgada. Los actores son Usuario, Sistema, Gestión de proyectos y ficheros y, en el caso de que se elimine el directorio, también Editor de texto.

Tabla 2-13 Flujo normal de *Eliminar proyecto*

Usuario	Sistema	Gestión de proyectos y ficheros	Editor de texto
1. Selecciona eliminar un proyecto	2. Solicita a Gestión de proyectos y ficheros la lista	3. Envía la lista	
5. Selecciona un proyecto	4. Muestra el listado de proyectos		
	7. Comunica la eliminación del proyecto	6. Elimina el proyecto de la lista	
	8. Desaparece el proyecto del explorador de proyectos		
10. Confirma la eliminación	9. Pregunta a Usuario si quiere eliminar el directorio del proyecto	11. Elimina el directorio del proyecto	12. Cierra las ventanas de los ficheros pertenecientes al proyecto
	13. Comunica la eliminación del directorio		
	14. Se actualiza el explorador de proyectos		

Flujos alternativos:

5. No hay todavía ningún proyecto, por lo que cancela la acción.

5. Cancela la acción.

10. Cancela la acción y pasa al punto 13.

12. Ninguno de los ficheros del proyecto estaba abierto, en cuyo caso no pasa nada.

2.4.3.3 Importar proyecto

Esta operación es la otra manera en que se implementa la operación crear de la gestión CRUD, por lo que intervienen los tres mismos actores que entonces.

Tabla 2-14 Flujo normal de *Importar proyecto*

Usuario	Sistema	Gestión de proyectos y ficheros
1. Selecciona importar un proyecto	2. Pide el directorio del proyecto	
3. Selecciona un directorio		4. Comprueba si el nombre del directorio está registrado
	6. Comunica la importación del proyecto	5. Registra el nombre y copia el directorio dentro del espacio de trabajo
	7. Aparece el nuevo proyecto en el explorador de proyectos	

Flujos alternativos:

3. Cancela la acción.

4. El nombre o la ruta está en uso.

2.4.4 Gestionar ficheros

Este caso de uso primario es un caso particular del anterior: los proyectos se componen por definición de ficheros, sus componentes, sólo que ahora hay un caso de uso referido a la recuperación de los propios ficheros.

La figura 2-8 desglosa los casos de uso secundarios:

Figura 2-8. Casos de uso secundarios de Gestionar ficheros

2.4.4.1 Abrir ficheros de un proyecto

Representa la operación de recuperar. Cuando en un proyecto se tienen varios ficheros, surgirá antes o después la necesidad de querer abrir algunos de sus ficheros específicamente a partir de la información del espacio de trabajo, que recoge los directorios principales de los proyectos. Los actores serán cuatro: Usuario, Sistema, Gestión de proyectos y ficheros y Editor de texto.

Tabla 2-15 Flujo normal de *Abrir ficheros de un proyecto*

Usuario	Sistema	Gestión de proyectos y ficheros	Editor de texto
1. Selecciona abrir ficheros de un proyecto	2. Solicita a Gestión de proyectos y ficheros la lista de proyectos	3. Envía la lista de proyectos	
5. Selecciona un proyecto	4. Muestra el listado de proyectos		
	6. Solicita a Gestión de proyectos y ficheros la lista de ficheros del proyecto	7. Envía la lista de ficheros del proyecto	
	8. Muestra el listado de ficheros del proyecto		
9. Selecciona ficheros	10. Solicita al editor de texto que abra los ficheros		11. Abre una nueva ventana por cada fichero solicitado

Flujos alternativos:

5. No hay todavía ningún proyecto, por lo que cancela la acción.

5. Cancela la acción.

9. El proyecto no tiene ningún fichero, por lo que cancela la acción.

9. Cancela la acción.

11. Los datos están corruptos, por lo que se cancela automáticamente la acción.

2.4.4.2 Eliminar ficheros de un proyecto

La continuación lógica de la anterior operación. Es una operación expuesta a ciertos riesgos, ya que, a diferencia de *Eliminar proyecto*, siempre supone la eliminación de los propios ficheros de la estructura del dispositivo de trabajo. Los actores son Usuario, Sistema y Gestión de proyectos y ficheros, y en el caso de que algún fichero estuviera abierto, Editor de texto deberá cerrarlo.

Tabla 2-16 Flujo normal de *Eliminar ficheros de un proyecto*

Usuario	Sistema	Gestión de proyectos y ficheros	Editor de texto
1. Selecciona eliminar ficheros de un proyecto	2. Solicita a Gestión de proyectos y ficheros la lista de proyectos	3. Envía la lista de proyectos	
5. Selecciona un proyecto	4. Muestra el listado de proyectos		
	6. Solicita a Gestión de proyectos y ficheros la lista de ficheros del proyecto	7. Envía la lista de ficheros del proyecto	

9. Selecciona ficheros	8. Muestra el listado de ficheros del proyecto		
11. Confirma	10. Pregunta a Usuario si está seguro	12. Elimina el fichero de los datos del proyecto	13. Cierra la ventana de los ficheros abiertos
	14. Se actualiza el explorador de proyectos		

Flujos alternativos:

5. No hay todavía ningún proyecto, por lo que cancela la acción.

5. Cancela la acción.

9. El proyecto no tiene ningún fichero, por lo que cancela la acción.

9. Cancela la acción.

11. Cancela la acción.

13. El fichero no estaba abierto, por lo que no cambia.

2.4.4.3 Guardar fichero en un proyecto

Cuando un proyecto está en continuo uso, es lógico que genere un gran número ficheros. Por ello, lo más conveniente es que exista la opción de guardar nuevos ficheros en el mismo, independientemente de que sean ficheros ya guardados en otra parte de la estructura de ficheros del dispositivo. Los actores de uso son los mismos que en *Abrir ficheros de un proyecto*.

Tabla 2-17 Flujo normal de *Guardar fichero en un proyecto*

Usuario	Sistema	Gestión de proyectos y ficheros	Editor de texto
1. Selecciona guardar el fichero en un proyecto	2. Solicita a Gestión de proyectos y ficheros la lista de proyectos	3. Envía la lista de proyectos	
5. Selecciona un proyecto	4. Muestra el listado de proyectos		
7. Escribe el nombre	6. Pide un nombre para el fichero	8. Comprueba si el nombre está en uso dentro del proyecto	
		9. Registra el nombre dentro del proyecto	10. Muestra el nombre del fichero
	12. Confirma que se ha guardado		11. Guarda el estado actual del fichero
	13. Aparece el nuevo fichero dentro del explorador de proyectos		

Flujos alternativos:

1. No hay ningún fichero abierto.

5. No hay todavía ningún proyecto, por lo que cancela la acción.

5. Cancela la acción.

7. El usuario cancela la operación.

8. El nombre está en uso dentro del proyecto.

8. El nombre contiene caracteres no permitidos.

2.4.5 Gestionar puntos de ruptura

La ejecución paso a paso (que pertenece a otro caso de uso) presupone en primer lugar la posibilidad de establecer en una secuencia de comandos los llamados puntos de ruptura que indican donde se introduce la pausa en la ejecución del código.

En la sección de Implementación se discuten con mayor detalle su aspecto y cómo se han introducido. De momento, aquí se examinan sus casos de uso secundarios, explicados en la figura 2-9:

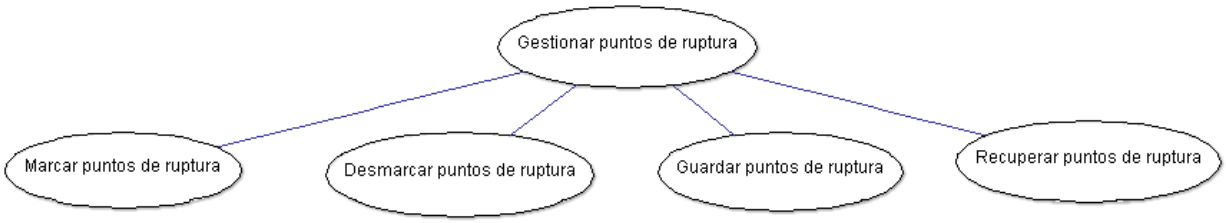


Figura 2-9 Casos de uso secundarios de Gestionar puntos de ruptura

2.4.5.1 Marcar puntos de ruptura

Este caso de uso corresponde a la operación Crear de CRUD: el usuario tendrá la necesidad de indicar aquella línea de una secuencia de comandos que debe experimentar una parada para poder examinar la ejecución de un fragmento de su código. Los actores intervinientes son sólo el Usuario y el Editor de texto.

Tabla 2-18. Flujo normal de *Marcar puntos de ruptura*

Usuario	Sistema	Editor de texto
1. Selecciona marcar un punto de ruptura		2. Señala en la posición elegida del fichero la marca visual correspondiente
	3. Guarda el nuevo punto de ruptura	

Flujos alternativos:

1. No ha seleccionado el margen correcto si lo hace con ratón.

1. La línea ya tenía un punto de ruptura, luego entra en *Desmarcar puntos de ruptura*.

2.4.5.2 Desmarcar puntos de ruptura

El contrario al anterior y la operación Eliminar de CRUD: el usuario puede verse en la necesidad de borrar un punto de ruptura por varios motivos, como que este se haya movido por editar la secuencia de comandos. Los actores intervinientes son otra vez el Usuario y el Editor de texto.

Tabla 2-19. Flujo normal de *Desmarcar puntos de ruptura*

Usuario	Sistema	Editor de texto
1. Selecciona desmarcar un punto de ruptura		2. Borra de la posición elegida del fichero la marca visual correspondiente
	3. Borra el punto de ruptura de la lista de puntos de ruptura del fichero	

Flujos alternativos:

1. No ha seleccionado el margen correcto si lo hace con ratón.

1. La línea no tenía un punto de ruptura, luego entra en *Marcar puntos de ruptura*.

2.4.5.3 Guardar puntos de ruptura

Una característica del marcado de líneas en *TextAdept* es que las marcas puestas sobre las líneas de un fichero abierto no se guardan una vez dicho fichero se cierre (o se cierre el sistema con el fichero abierto). Por ello, será necesario guardar los puntos de ruptura en el sistema de ficheros y ahora intervendrá el Sistema en adición de los dos actores de uso anteriores. En la gestión CRUD, esta es la operación Actualizar.

Tabla 2-20. Flujo normal de *Guardar puntos de ruptura*

Sistema	Editor de texto
	1. Manda a Sistema el listado de líneas con la marca visual del punto de ruptura
2. Almacena la información recibida	

Flujos alternativos:

1. El Usuario no ha marcado puntos de ruptura en el fichero abierto, así que no se ejecuta esta operación.

2.4.5.4 Recuperar puntos de ruptura

El caso contrario al anterior, pues una vez un fichero es abierto o el sistema se inicializa, es preciso recuperar sus puntos de ruptura y mostrárselos al usuario. Intervienen los mismos tres actores que antes. Es la operación Recuperar de la gestión CRUD.

Tabla 2-21. Flujo normal de *Recuperar puntos de ruptura*

Sistema	Editor de texto
1. Manda a Editor de Texto el listado de líneas que deben tener la marca visual del punto de ruptura	
	2. Marca los puntos de ruptura en sus líneas

Flujos alternativos:

1. No hay información de los puntos de ruptura porque el Usuario, así que acaba la operación.

2.4.6 Gestionar plantillas

Este caso es muy similar a *Gestionar ficheros* y su desglose en casos secundarios es análogo: crear, eliminar y abrir plantillas. Por ello, la figura 2-10 es también análoga a la 2-7.

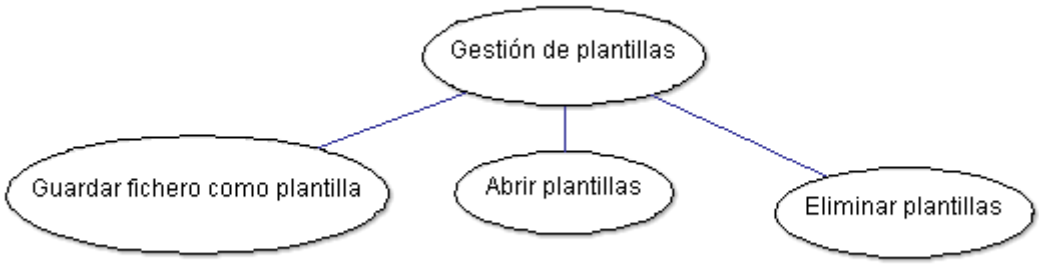


Figura 2-10. Casos de uso secundarios de Gestionar plantillas

2.4.6.1 Guardar fichero como plantilla

Una plantilla es un fichero normal que es guardada *exprofeso* como modelo de otros ficheros, a veces recibiendo una extensión específica. Los actores intervinientes son tres: Usuario, Sistema y Editor de texto. Gestión de proyectos y ficheros no interviene porque las plantillas son independientes de este sistema.

Tabla 2-22 Flujo normal de *Guardar fichero como plantilla*

Usuario	Sistema	Editor de texto
1. Selecciona guardar el fichero como plantilla	2. Pide un nombre	
3. Escribe un nombre	4. Comprueba si el nombre es único	
	5. Pide a Editor de texto que guarde el fichero	6. Guarda el fichero con el nombre indicado en el directorio

Flujos alternativos:

3. Cancela la acción.

4. El nombre ya lo tiene otra plantilla.

2.4.6.2 Abrir plantillas

Una vez se tengan las plantillas, una acción que cualquier Usuario deseará hacer es abrirlas. Como es posible necesitar varias plantillas, se da la opción de abrir varias a la vez. Los actores son los mismos: Usuario, Sistema y Editor de texto.

Tabla 2-23 Flujo normal de *Abrir plantillas*

Usuario	Sistema	Editor de texto
1. Selecciona abrir plantillas	2. Muestra el listado de plantillas	
3. Selecciona plantillas	4. Solicita al editor de texto que abra las plantillas	5. Abre una nueva ventana por cada plantilla solicitada

Flujos alternativos:

3. No hay plantillas aún, por lo que cancela la operación.

3. Cancela la acción.

2.4.6.3 Eliminar plantillas

La otra alternativa cuando existen plantillas es eliminarlas porque ya no sean necesarias. Otra vez intervienen los mismos actores: Usuario, Sistema y Editor de texto.

Tabla 2-24 Flujo normal de *Eliminar plantillas*

Usuario	Sistema	Editor de texto
1. Selecciona Eliminar plantillas	2. Muestra el listado de plantillas	
3. Selecciona plantillas	4. Elimina las plantillas	
	4. Solicita al Editor de texto que cierre las plantillas si están abiertas	5. Cierra las ventanas de las plantillas abiertas

Flujos alternativos:

3. No hay plantillas aún, por lo que cancela la operación.

3. Cancela la acción.

5. La plantilla no estaba abierta, por lo que no cambia.

2.4.7 Compilar

Este caso de uso primario es bastante importante, ya que una de las funcionalidades más esperadas de un IDE es que compile un proyecto. Por lo tanto, su justificación es inmediata.

El desglose de la figura 2-10 se limita a si la compilación es o no separada, que afecta a, por ejemplo, proyectos escritos en el lenguaje C, que dan esa opción.

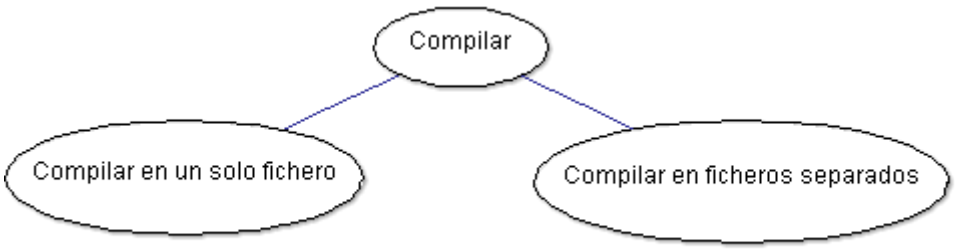


Figura 2-11. Casos de uso secundarios de Compilar

2.4.7.1 Compilar en un solo fichero

La modalidad más sencilla de compilación en C consiste en crear un solo fichero objeto a partir de un fichero fuente que contenga todo el código. Si bien el Editor de texto trae la funcionalidad de compilación, es necesaria la existencia del compilador correspondiente al lenguaje en que esté escrito el fichero. Así pues, los actores serán Usuario, Sistema, Editor de texto, Compilador/Intérprete y Salida de resultados.

Tabla 2-25 Flujo normal de *Compilar en un solo fichero*

Usuario	Sistema	Editor de texto	Compilador/Intérprete	Salida de resultados
1. Selecciona Compilar en un solo fichero	2. Abre el recuadro de introducción de órdenes			
3. Escribe una orden		4. Proporciona el directorio y el nombre del fichero	5. Compila	
			6. Si encuentra un error, informa del mismo	
			7. Genera un fichero compilado	8. Muestra mensaje de resultados de la compilación

Flujos alternativos:

2. No está activada la opción de escribir la orden, por lo que pasa al punto 4.

3. No es una orden válida.

5. El fichero no tiene una extensión reconocida o no se tiene instalado el software de compilación necesario.

5. El fichero tiene una extensión de Java u otro lenguaje en que lo normal es compilar en ficheros separados.

6. Encuentra errores, por lo que se detiene la compilación y se pasa al punto 8.

2.4.7.2 Compilar en ficheros separados

Esta modalidad de compilación es una generalización del caso anterior. La complejidad creciente de los programas llevó a dividir su código en diversos ficheros fuente (que fueron el antecedente de conceptos como las librerías o los módulos actuales), lo cual también influyó en el modo de compilar, ya que con un control de versiones es más práctico sólo compilar aquellos archivos que hayan sufrido alteraciones respecto a un registro. Los actores son los mismos que en el caso de uso anterior.

Tabla 2-26 Flujo normal de *Compilar en ficheros separados*

Usuario	Sistema	Editor de texto	Compilador/Intérprete	Salida de resultados
1. Selecciona Compilar en ficheros separados				
		2. Proporciona el directorio del fichero	3. Compila separadamente, examinando la carpeta	
			4. Si encuentra un error, informa del mismo	
			5. Genera ficheros compilados	6. Muestra mensaje de resultados de la compilación

Flujos alternativos:

3. Alguno de los ficheros no tiene una extensión reconocida o no se tiene instalado el software de compilación necesario.

3. El fichero tiene una extensión de java u otro lenguaje en que lo normal es compilar en ficheros separados.

4. Encuentra errores, por lo que se detiene la compilación y se pasa al punto 8.

2.4.8 Ejecutar

Este caso de uso primario es igual de importante que el de compilar, pues es su funcionalidad inmediata: ejecutar el programa que se ha compilado en el IDE y poder observar sus resultados sin tener que recurrir a medios externos como la ventana de comandos.

El desglose de la figura 2-12 nos muestra los dos casos secundarios más habituales:

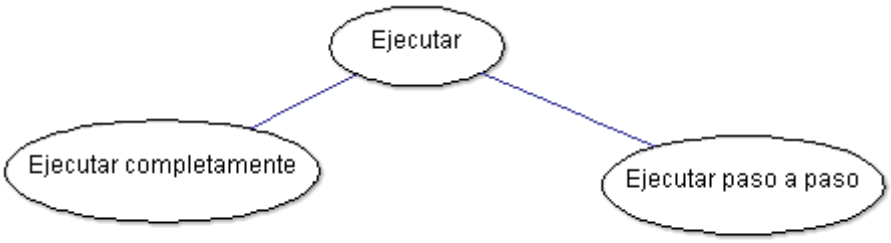


Figura 2-12. Casos de uso secundarios de Ejecutar

2.4.8.1 Ejecutar complemente

Ejecutar un programa de principio a fin es la opción más sencilla y directa. Es la opción estándar para programas muy sencillos o que carecen de caminos alternativos, aparte de que es razonable suponer que un programa se ejecutará sin otras pausas excepto las de sus propios usuarios. Los actores intervinientes son Usuario, Sistema, Editor de texto, Compilador/Intérprete y Salida de resultados.

Tabla 2-27 Flujo normal de *Ejecutar completamente*

Usuario	Sistema	Editor de texto	Compilador/Intérprete	Salida de resultados
1. Selecciona ejecutar completamente	2. Abre el recuadro de introducción de órdenes			
3. Escribe una orden		4. Proporciona el directorio y el nombre del fichero	5. Ejecuta el programa	
			6. Busca errores durante la ejecución	7. Muestra mensaje de resultados de la ejecución

Flujos alternativos:

2. No está activada la opción de escribir la orden, por lo que pasa al punto 4.

3. No es una orden válida.

5. No se ha compilado previamente el programa.

6. Encuentra errores, por lo que se detiene la ejecución.

2.4.8.2 Ejecutar paso a paso

La creciente complejidad de los programas informáticos ocurrió tanto en extensión como en flujos alternativos: un programa actual no sólo es una numerosa cadena de operaciones encadenadas, sino que también ofrecen múltiples opciones que no todos los usuarios usan en su totalidad. Ello supuso la necesidad de poder ejecutar un programa paso a paso mediante puntos de ruptura para poder

depurar mejor los posibles errores y establecer su causa última. Los actores son los mismos que en *Ejecutar completamente*.

Tabla 2-28 ***Flujo normal de Ejecutar paso a paso***

Usuario	Sistema	Editor de texto	Compilador/Intérprete	Salida de resultados
1. Selecciona ejecutar paso a paso	2. Abre el recuadro de introducción de órdenes			
3. Escribe una orden		4. Proporciona el directorio y el nombre del fichero	5. Ejecuta el programa	
8. Continúa (vuelve al paso 6)		7. Pide a Usuario continuar porque ha llegado a un punto de ruptura	6. Busca puntos de ruptura y errores durante la ejecución. Si encuentra una ruptura, pasa a 7	9. Muestra mensaje de resultados de la ejecución

Flujos alternativos:

2. No está activada la opción de escribir la orden, por lo que pasa al punto 4.

3. No es una orden válida.

5. No se ha compilado previamente el programa.

6. Encuentra errores, por lo que se detiene la ejecución.

6. No hay puntos de ruptura, por lo que actúa como una ejecución completa y pasa al punto 9.

2.4.9 Configurar

Este caso de uso no se desglosará en otros secundarios, ya que son muy numerosos.

Tabla 2-29 Flujo normal de ***Configurar***

Usuario	Sistema	Configuración
1. Selecciona la configuración	2. Avisa a Configuración	3. Envía el listado de opciones
5. Cambia el valor de una opción	4. Muestra las opciones disponibles	6. Registra el cambio efectuado

Flujos alternativos:

3. No hay opciones disponibles o no se pueden cambiar para el proyecto abierto. Se cancela.

5. El usuario decide no cambiar nada y cancela la operación.

2.4.10 Añadir extensión

Este caso de uso primario es igual de importante que el de compilar, pues depende del tipo de extensión que se quiera añadir, que será elegida por el propio Usuario.

Tabla 2-30 Flujo normal de *Añadir extensión*

Usuario	Sistema	Configuración
1. Elige añadir extensión	2. Avisa a Configuración	3. Despliega el menú de adición
4. Elige la extensión y le da un nombre		5. Añade la extensión a la lista de opciones
7. Confirma	6. Pregunta si confirma la adición	

Flujos alternativos:

4. La extensión no puede cargarse en el IDE por razones técnicas.

4. Ya existe ese nombre dentro de las opciones.

4. El nombre contiene caracteres no permitidos.

4. Cancela la operación.

7. Cancela la operación.

2.5 Modelo del dominio

La figura 2-13 muestra el modelo del dominio del sistema. De sus relaciones, se deduce fácilmente que el sistema alberga seis elementos: Espacios de trabajo, Gestión de proyectos y ficheros, Editor de textos, Configuración, Salida de resultados y Explorador de proyectos. Gestión de proyectos y ficheros es el que presenta mayor acoplamiento ya que es una pieza fundamental de la arquitectura, mientras que todos los demás bien interaccionan con sólo Sistema y otro elemento principal, o con alguna de sus creaciones (Editor de texto con los Ficheros, por ejemplo).

El único elemento externo a Sistema en última instancia son los compiladores e intérpretes necesarios para las tareas de compilación y ejecución.

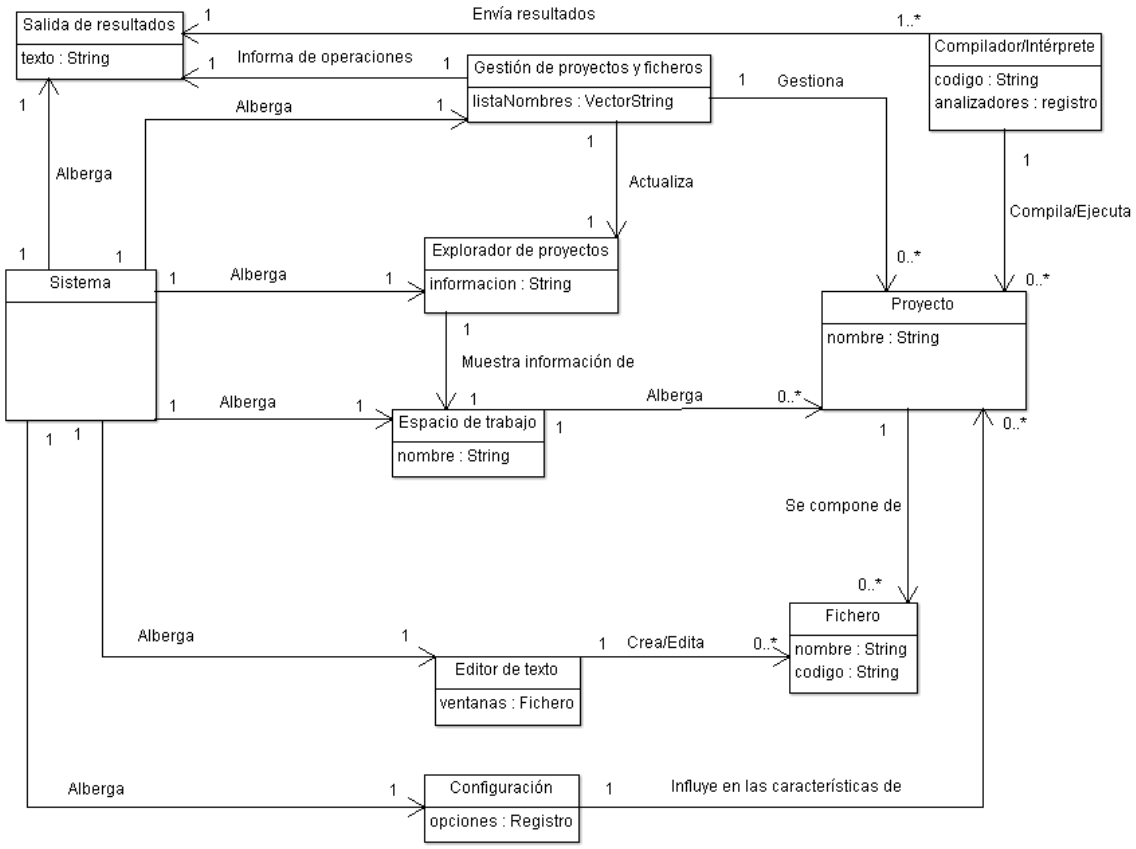


Figura 2-13. Modelo del dominio del sistema

Capítulo 3

DISEÑO DEL SISTEMA

Tal como se ha expuesto en las secciones anteriores, el diseño hará uso de la cualidad de ciertos editores de texto, que ya incluyen operaciones que son objetivos del presente proyecto fin de grado, tales como la compilación y ejecución, así como su extensibilidad. A estas funcionalidades básicas es necesario añadir tanto la gestión de proyectos, como la compilación separada; y alterar ligeramente la interfaz visual de la herramienta final.

Los apartados que se exponen a continuación siguen las guías recomendadas en el diseño software (Turturro, 2023).

3.1 Diseño arquitectónico

La Figura 3-1 ilustra el diseño. Básicamente, el editor de texto es el componente central del sistema. Los demás componentes pueden presentar una variación, a gusto del usuario, en su localización.

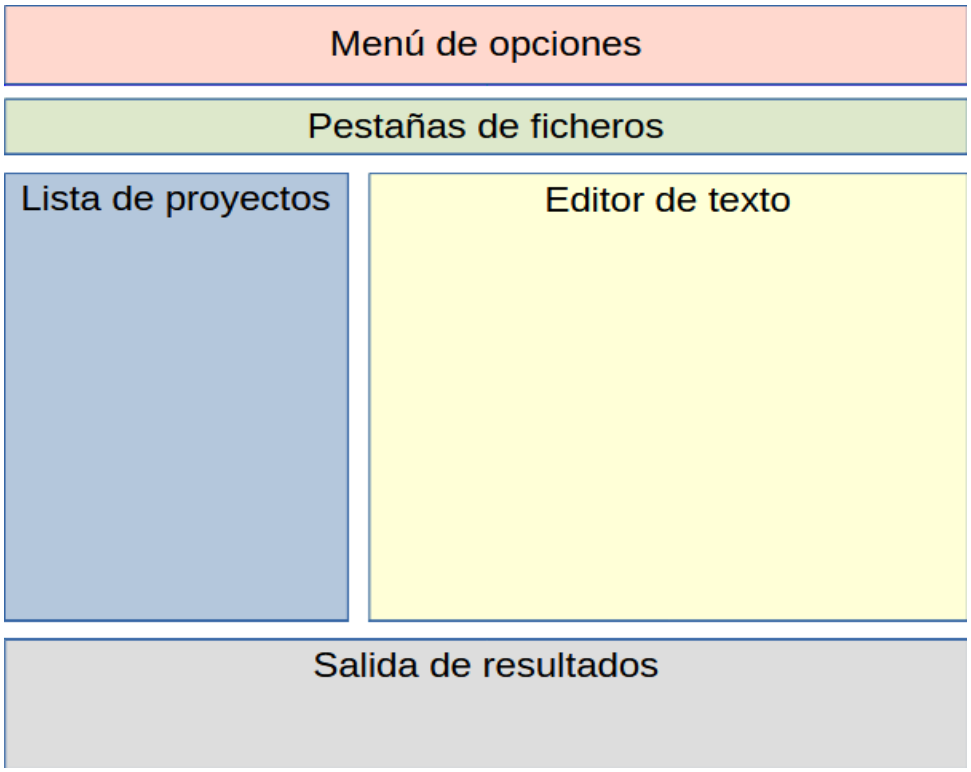


Figura 3-1. Diseño arquitectónico del sistema

3.2 Modelado de datos

Muchos editores de texto ya incluyen de base su propio sistema de modelado de datos para compilar el código fuente. También soportan su propio sistema de extensiones, donde es posible definir una restricción de las opciones de configuración que proporcione el IDE, habilitando una ventana exclusivamente para los resultados y otra para la gestión de proyectos.

En el segundo caso, *TextAdept* y otros editores de texto son capaces de reconocer proyectos si en una carpeta hay control de versiones, aparte de que es necesario definir cuatro tipos de ficheros de datos:

1. El primero debe guardar los nombres existentes de todos los espacios de trabajo. La opción más sencilla es un directorio que contenga:
   - Un directorio para cada espacio de trabajo.
   - Un fichero con el nombre del último espacio de trabajo cambiado por el usuario.
2. El segundo debe guardar el nombre de los proyectos del espacio de trabajo actual, junto a sus ficheros de cada proyecto. Este fichero se puede construir a partir de aquel del espacio de trabajo, leyendo los ficheros contenidos en cada ruta indicada, y puede estar en el mismo directorio del punto 1.
3. El tercero debe guardar el nombre de las plantillas, análogo al punto 1, lo más sencillo es crear un directorio exclusivo.
4. El cuarto debe referirse a los puntos de ruptura (*breakpoints*) necesarios para la ejecución paso a paso. Debe existir una colección de los registros de estos para cada posible fichero, por lo que la mejor manera de registrarlos es relacionarlos con sus rutas. Dentro de un directorio hay que crear:
   - Un directorio con directorios numerados para el directorio de cada fichero en el que se hayan creado puntos de ruptura, que guarden el registro de los puntos de ruptura.
   - Un directorio con directorios numerados para el directorio de cada fichero en el que se hayan creado puntos de ruptura, que guarden las secuencias de comandos con los puntos de ruptura traducidos.
   - Un fichero que relacione las rutas con los puntos de ruptura .
   - Un fichero con el contador de registros.

El formato de estos archivos puede ser *txt* u otro, incluso sin extensión, según se estime más conveniente.

3.3 Especificación de componentes de hardware y software

Estos componentes deberían ser suficientes:

- Ordenador personal de al menos 6 GB de RAM y procesador de 500 MHz.
- Monitor, teclado y ratón.
- Sistema operativo Windows 10 en una arquitectura de 64 bits o Linux con Qt 5 o GTK 3 (Eid, Textadept, 2007).
- Editor de texto *TextAdept*.
- GCC para compilar C.

- Máquina virtual de Java para interpretar este lenguaje.

3.4 Patrón de la arquitectura

Como *TextAdept* es la base del IDE, su arquitectura es la misma: una arquitectura modular, en la que se ha dividido el programa en varios directorios que agrupan funcionalidades. La figura 3-2 muestra una captura de su directorio principal, tal como originalmente está constituido a partir de la descarga de la página web que lo incluye.

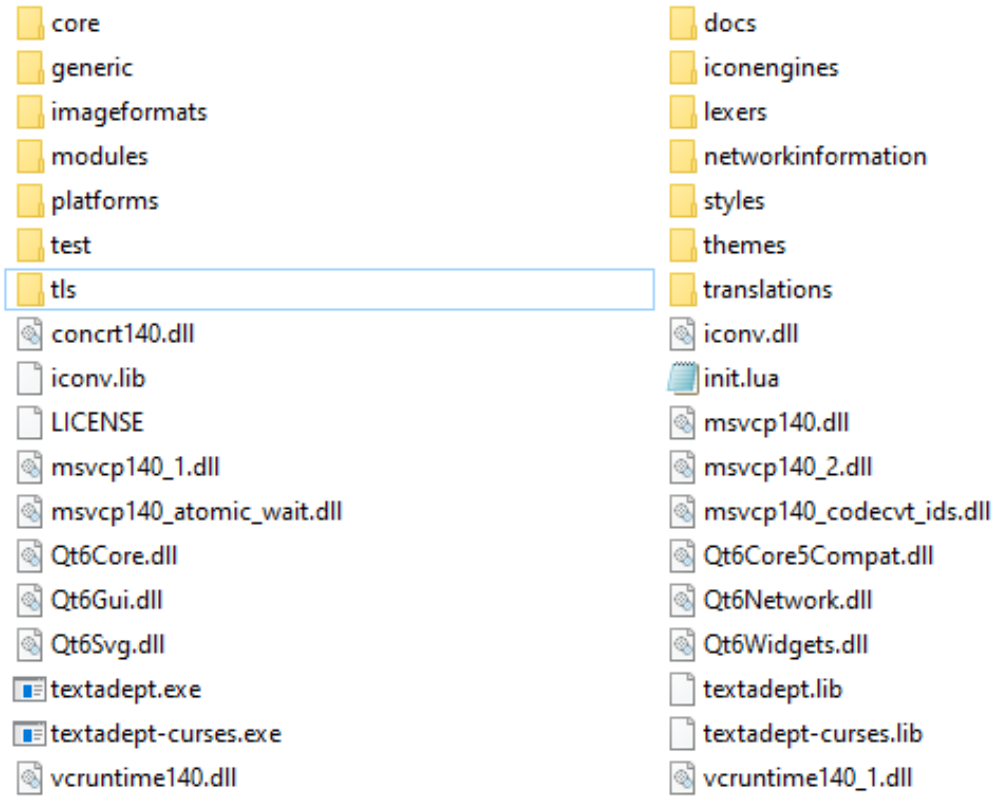


Figura 3-2. Directorio principal de TextAdept original

Esta arquitectura se compone de:

1. Un directorio principal, con el ejecutable de *TextAdept* y el fichero de inicialización ***init.lua***, encargado de invocar los archivos de los módulos.
2. Un directorio con las librerías de las funcionalidades principales del sistema, ***core***, como el control de eventos, el reconocimiento de la pulsación del teclado, la interfaz de usuario, el sistema operativo o el tratamiento de ficheros. En el Capítulo 4: Implementación, se explicará el uso que se le ha dado a algunas de ellas.
3. Un directorio, ***lexers***, que contiene los analizadores léxicos de varios lenguajes.
4. Un directorio de módulos, ***modules***, en el que cada módulo tiene su propio directorio. El módulo estándar se llama simplemente *TextAdept*.
5. Un directorio, ***platforms***, con los archivos encargados de que se ejecuten las pruebas.
6. Un directorio, ***tests***, con los archivos encargados de la correcta ejecución.

7. Otros directorios de interés, aunque no formen parte de los objetivos de esta memoria, como la documentación, las traducciones, los estilos y los iconos de escritorio.

Esta arquitectura no variará en lo sustancial, ya que la implementación modificará sólo aquellos ficheros necesarios para implementar las funcionalidades precisas aún no incorporadas.

3.5 Diseño de interfaces

El sistema extiende las capacidades nativas de *TextAdept*, para lo que la anterior sección sobre el patrón de la arquitectura es necesaria para comprender mejor cómo se va a ver el resultado final. Para el sistema se va a crear un nuevo módulo llamado *IDE*, en el que, partiendo de las secuencias de comandos originales, se van a editar algunas de las originales y a añadir otras nuevas. En esta sección interesa ver cómo se relacionan estos componentes para los casos de uso definidos en la sección 2, para lo que se pueden representar como diagramas de clases (aunque estos componentes no sean en rigor clases, funcionan de un modo similar) (sólo se representan aquellos casos de uso en que intervienen algunos de los objetos software creados para este proyecto):

3.5.1 Gestionar espacios de trabajo

3.5.1.1 Crear espacio de trabajo

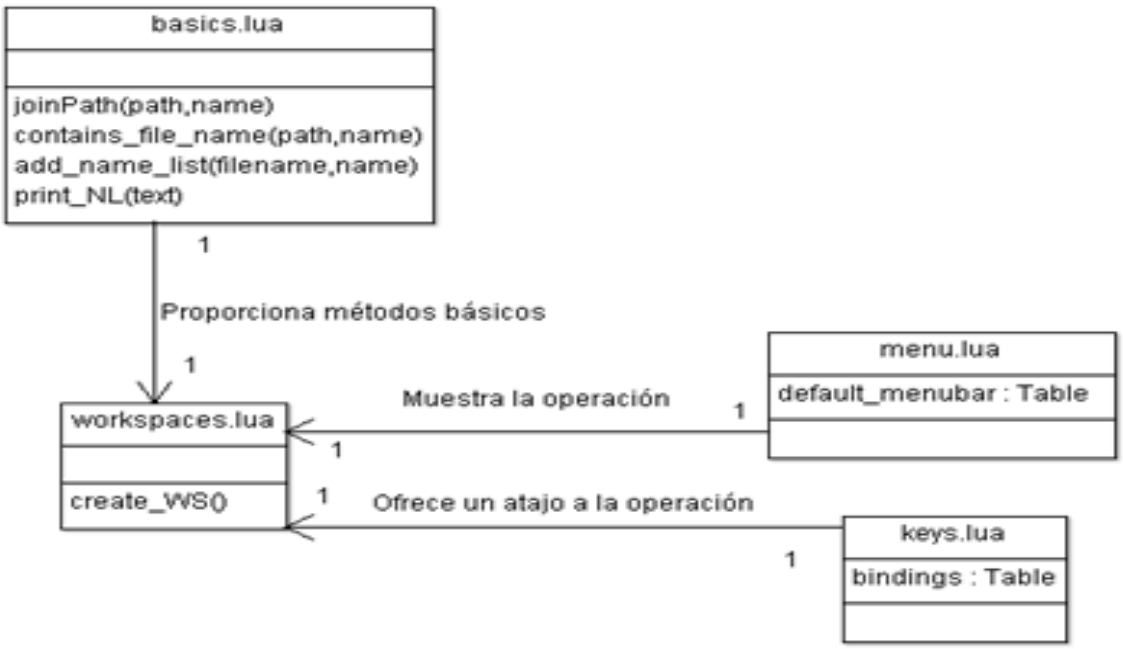


Figura 3-3. Diagrama de interfaces del caso de uso Crear espacio de trabajo

3.5.1.2 Eliminar espacio de trabajo

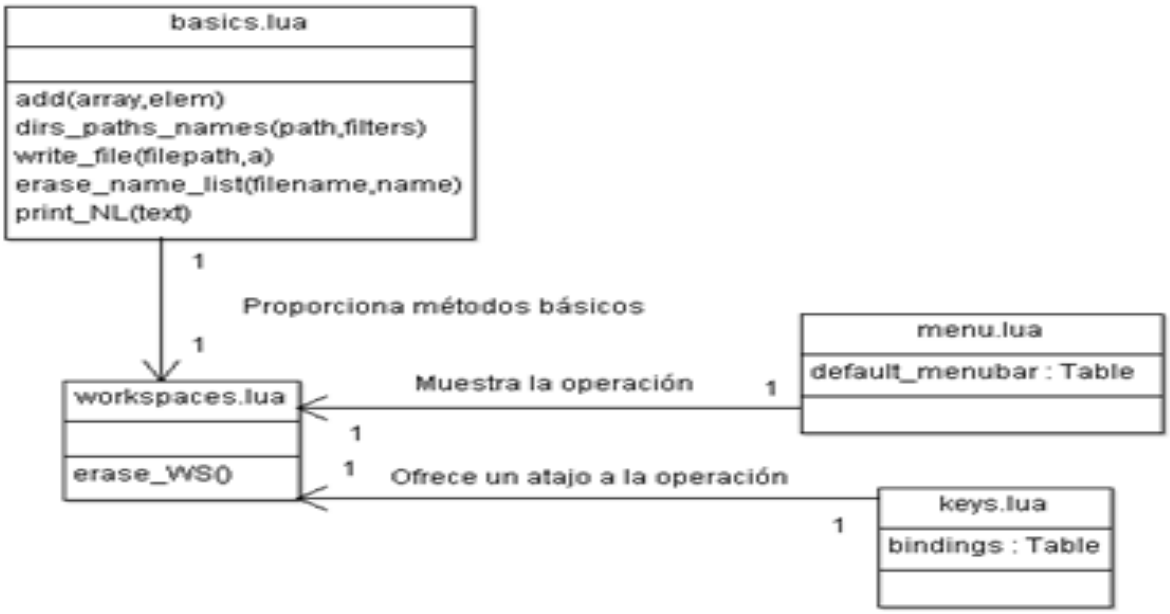


Figura 3-4. Diagrama de interfaces del caso de uso Eliminar espacio de trabajo

3.5.1.3 Abrir espacio de trabajo

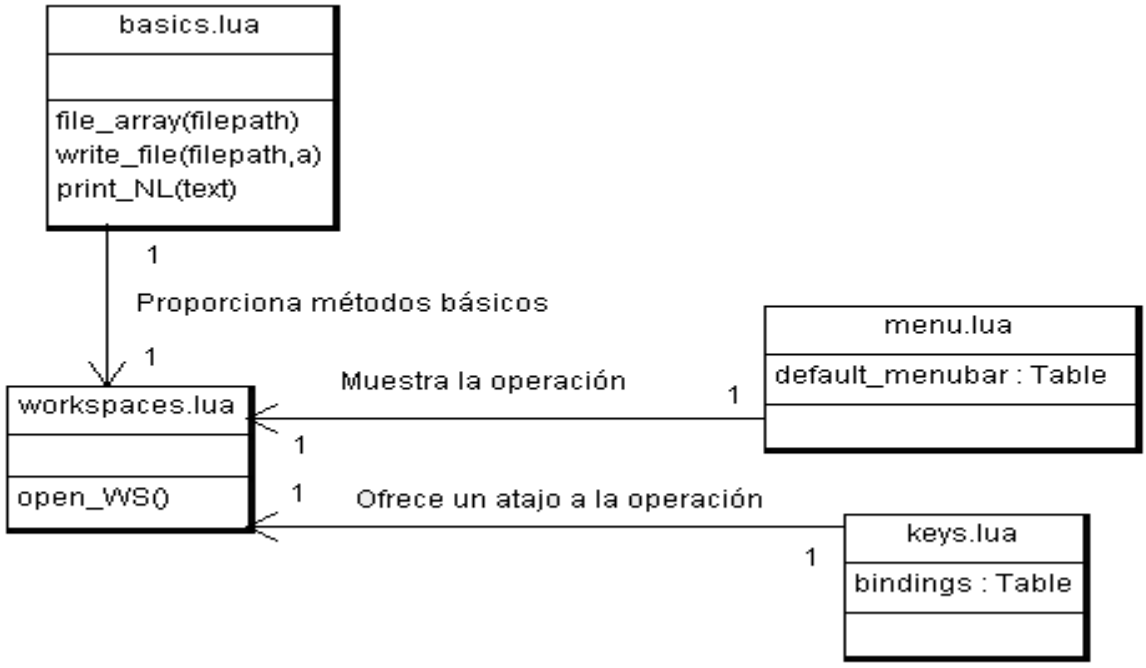


Figura 3-5. Diagrama de interfaces del caso de uso Abrir espacio de trabajo

3.5.1.4 Actualizar espacio de trabajo

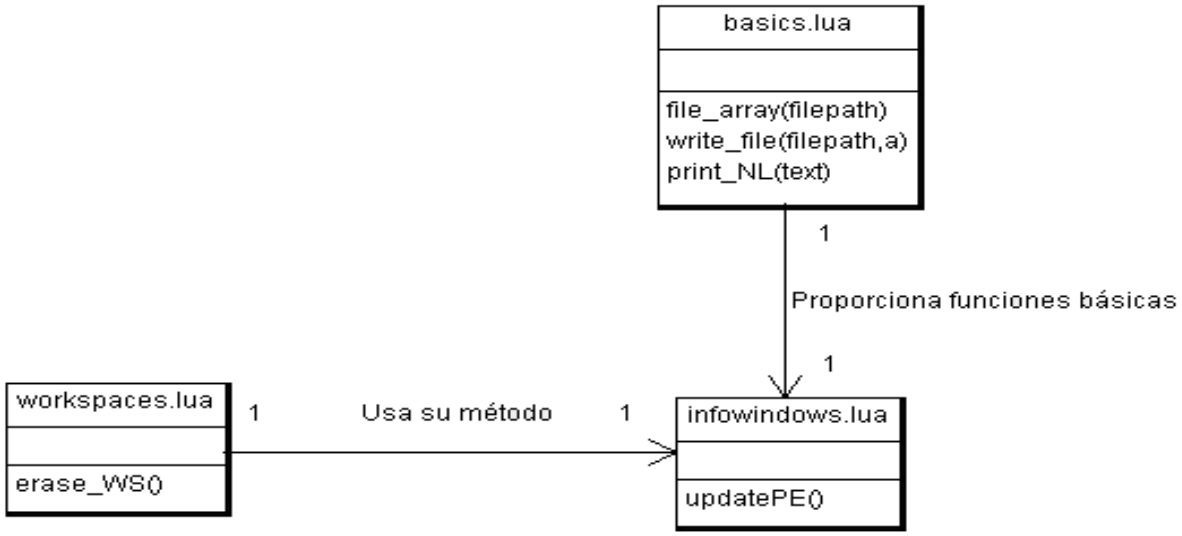


Figura 3-6. Diagrama de interfaces del caso de uso Actualizar espacio de trabajo

3.5.2 Gestionar proyectos

3.5.2.1 Iniciar proyecto

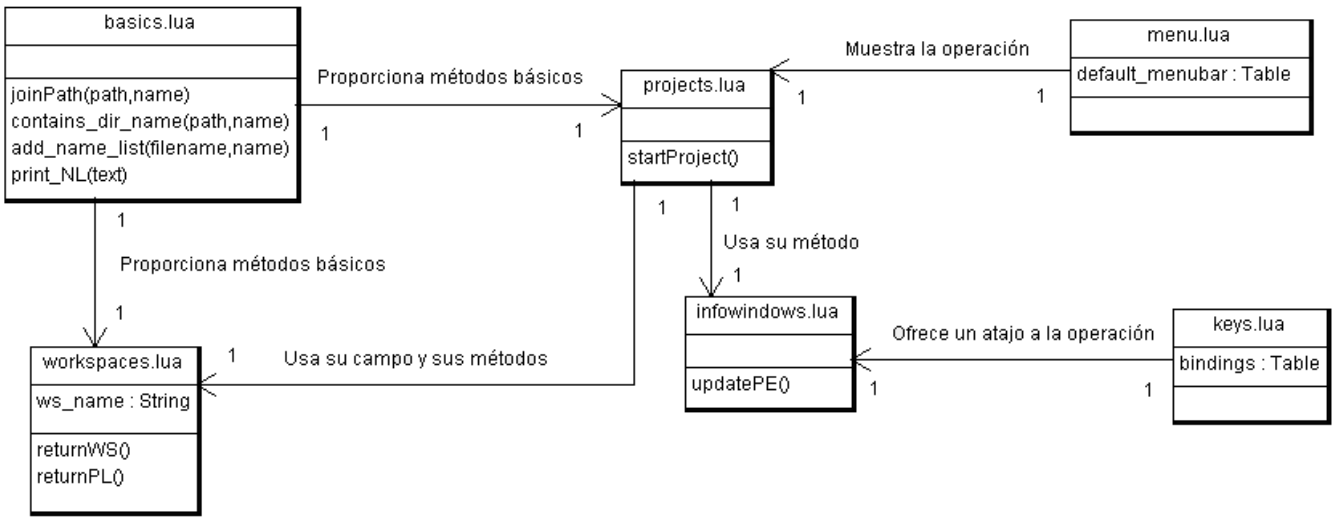


Figura 3-7. Diagrama de interfaces del caso de uso Iniciar proyecto

3.5.2.2 Eliminar proyecto

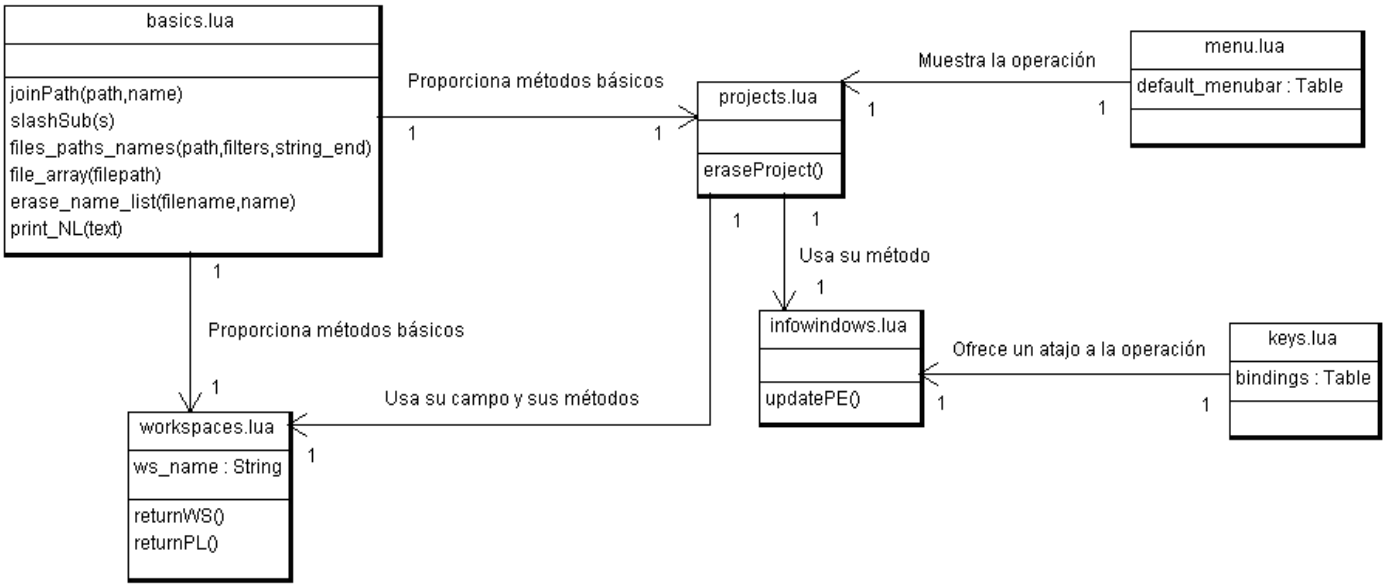


Figura 3-8. Diagrama de interfaces del caso de uso Eliminar proyecto

3.5.2.3 Importar proyecto

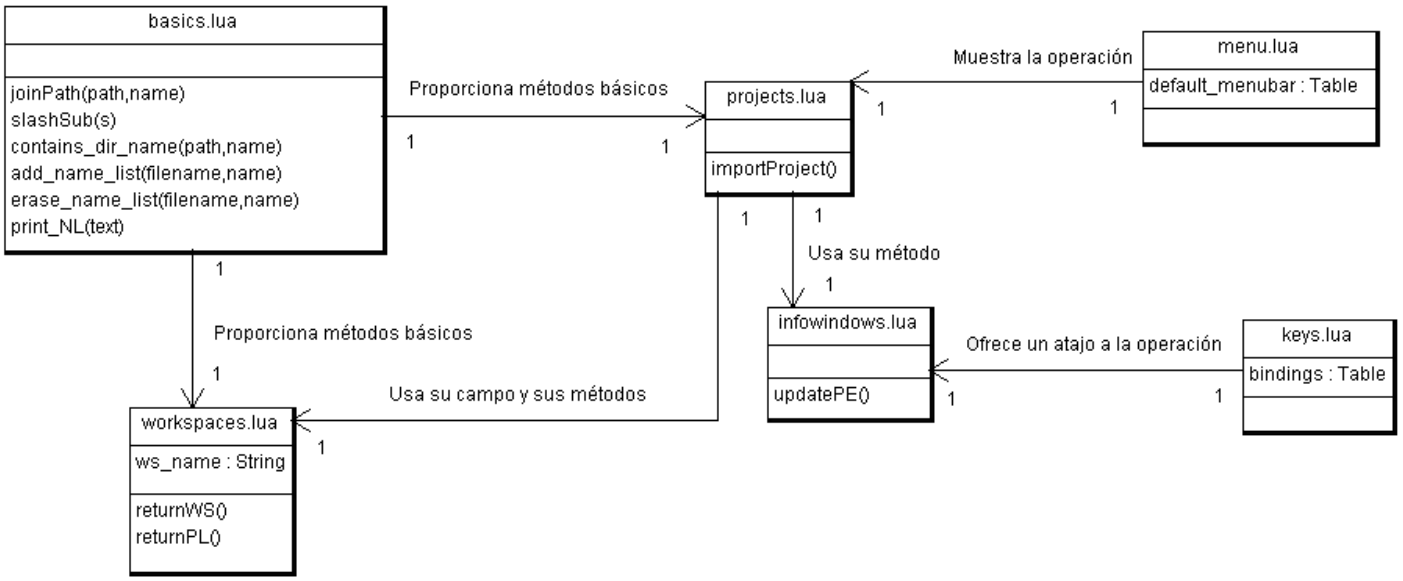


Figura 3-9. Diagrama de interfaces del caso de uso Importar proyecto

3.5.3 Gestionar ficheros

3.5.3.1 Abrir ficheros de un proyecto

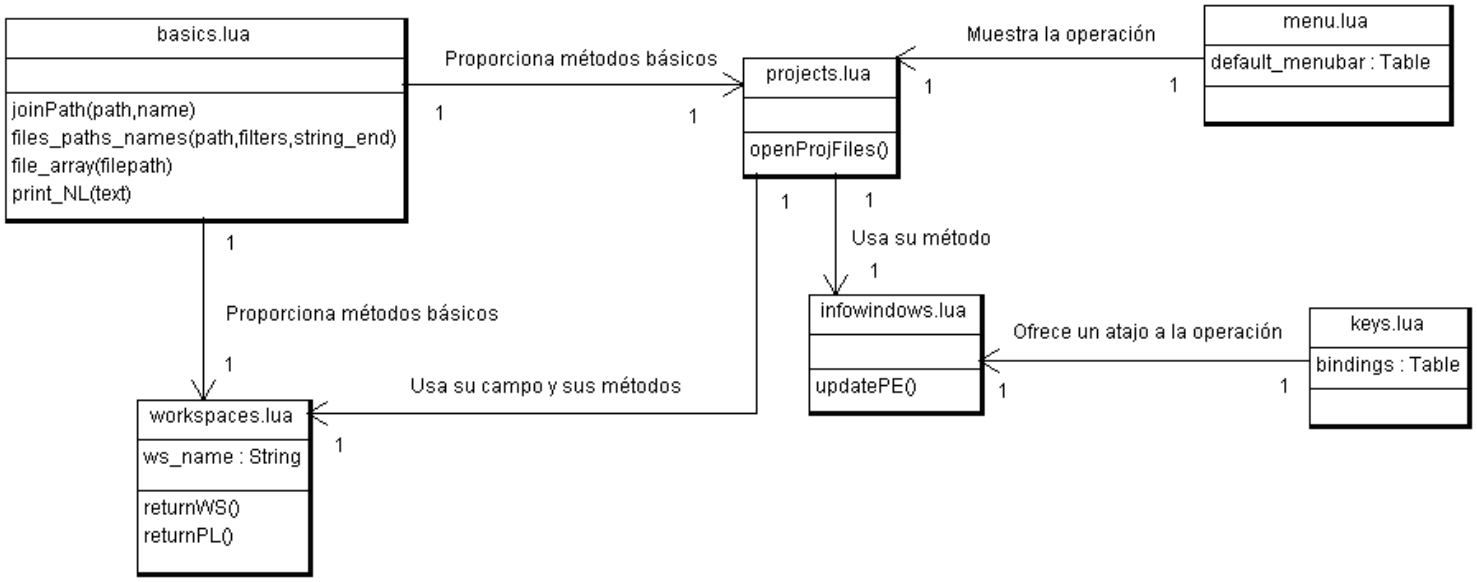


Figura 3-10. Diagrama de interfaces del caso de uso Abrir ficheros de un proyecto

3.5.3.2 Eliminar ficheros de un proyecto

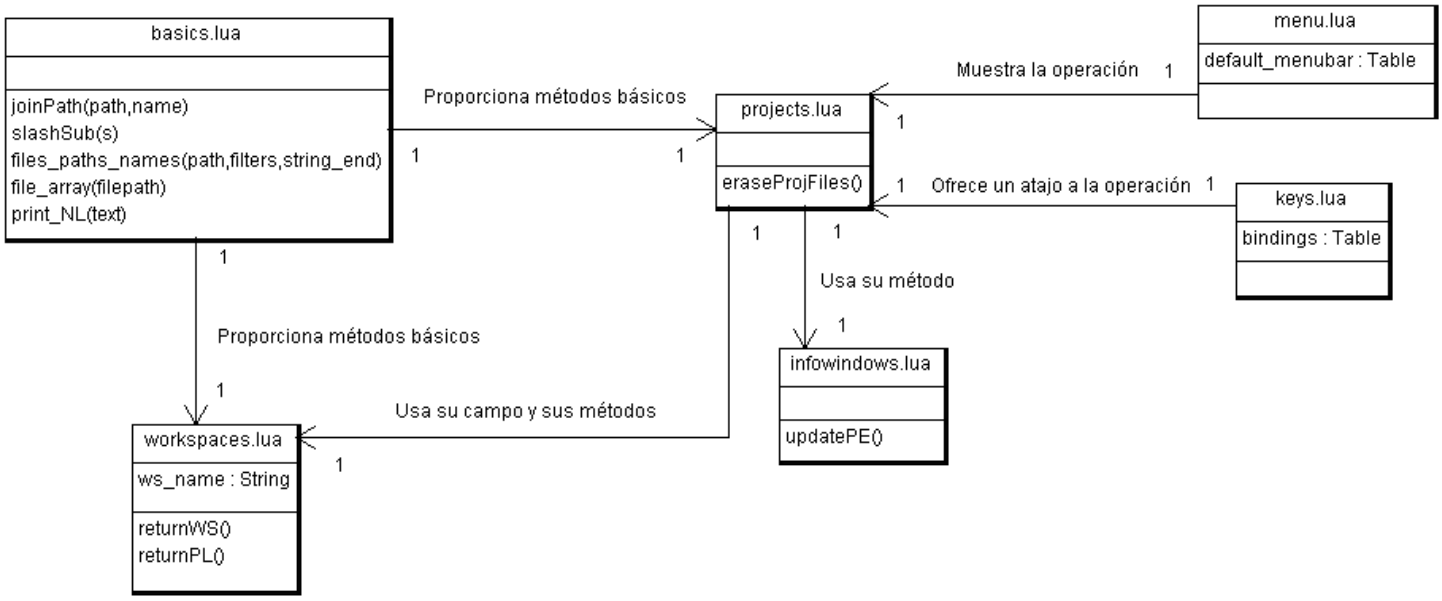


Figura 3-11. Diagrama de interfaces del caso de uso Eliminar ficheros de un proyecto

3.5.3.3 Guardar fichero en un proyecto

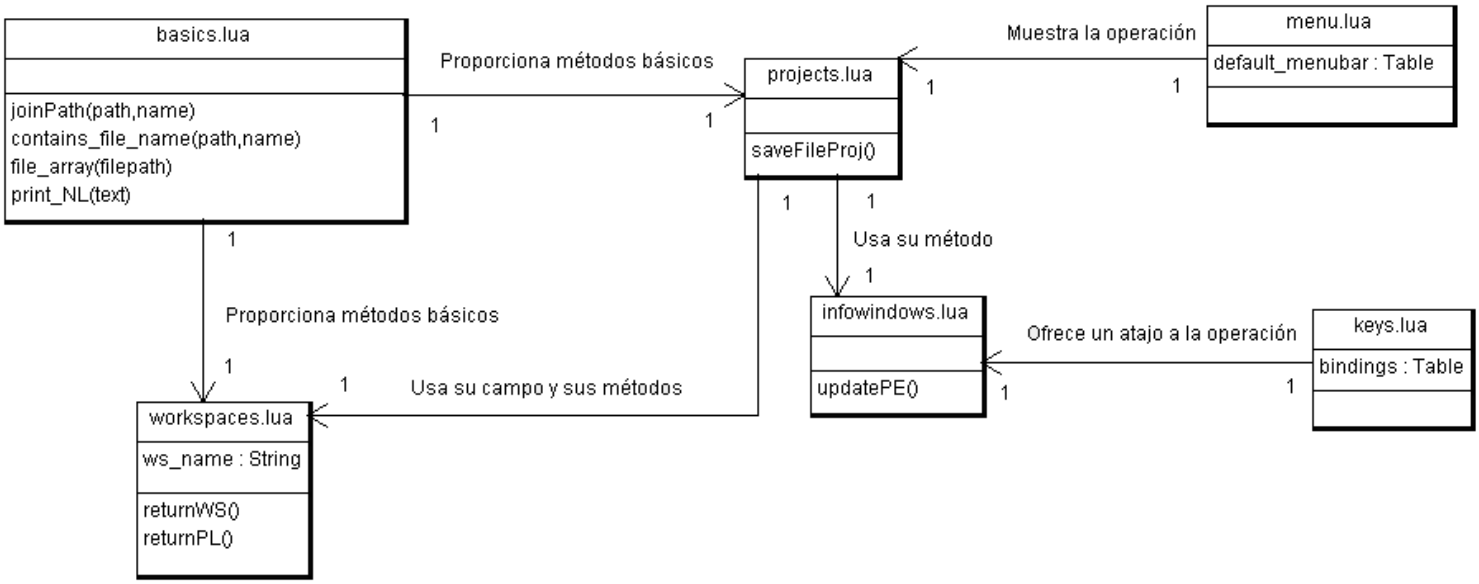


Figura 3-12. Diagrama de interfaces del caso de uso Guardar fichero en un proyecto

3.5.4 Gestionar puntos de ruptura

3.5.4.1 Marcar puntos de ruptura y desmarcar puntos de ruptura

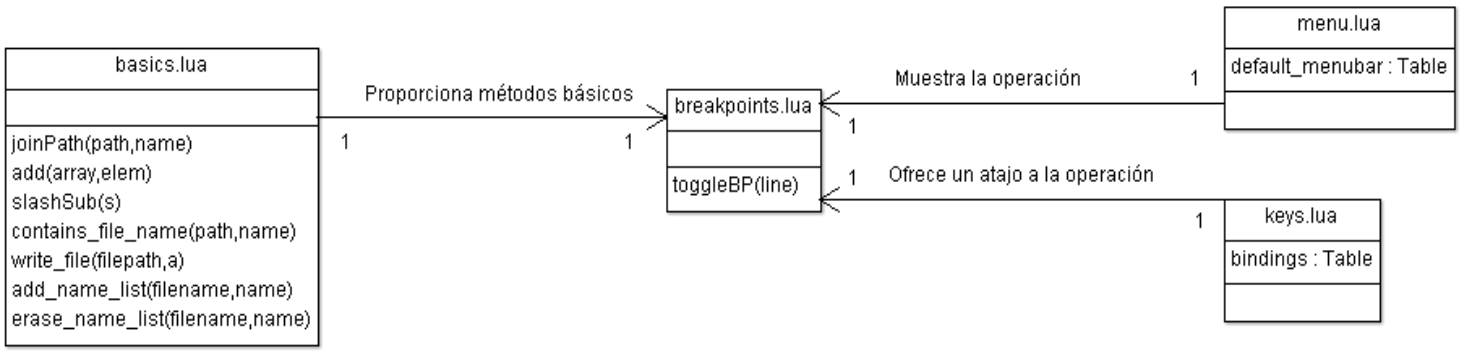


Figura 3-13. Diagrama de interfaces del caso de uso Marcar puntos de ruptura y desmarcar puntos de ruptura

3.5.4.2 Guardar puntos de ruptura

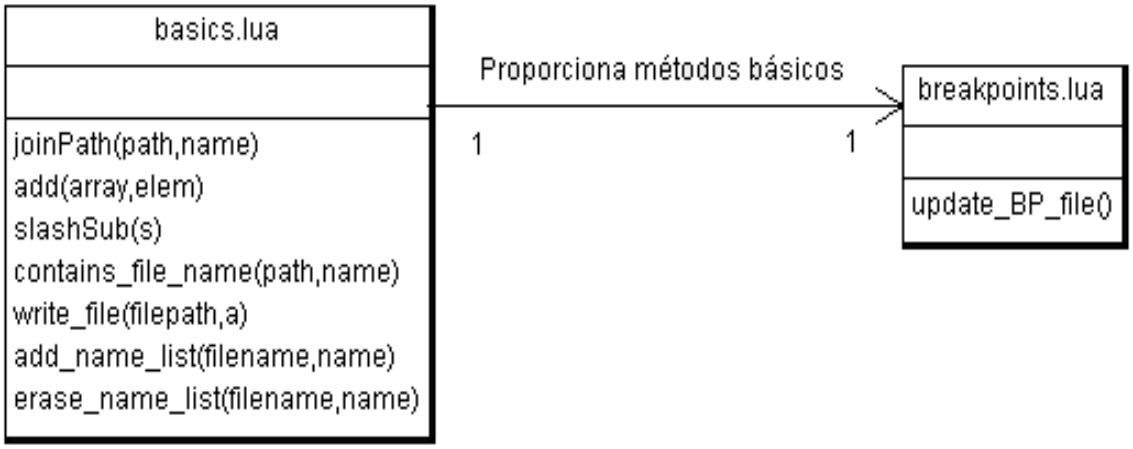


Figura 3-14. Diagrama de interfaces del caso de uso Guardar puntos de ruptura

3.5.4.3 Recuperar puntos de ruptura

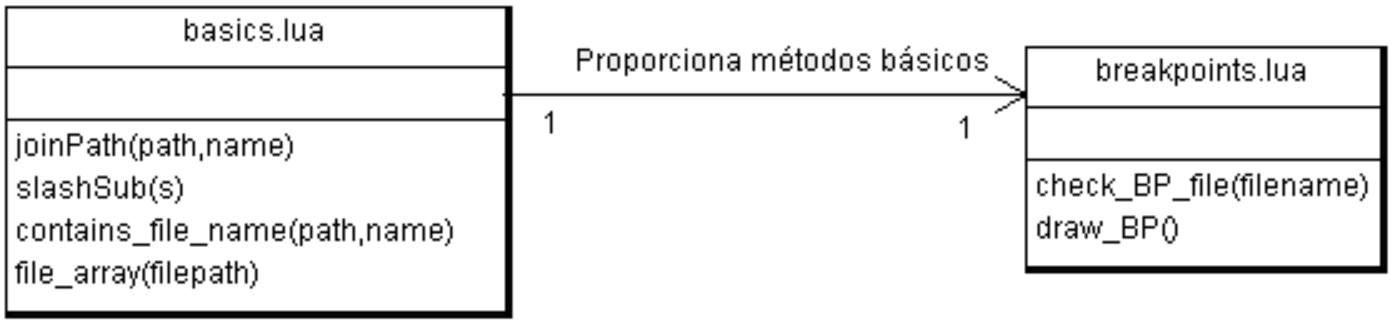


Figura 3-15. Diagrama de interfaces del caso de uso Recuperar puntos de ruptura

3.5.5 Gestionar plantillas

3.5.5.1 Guardar fichero como plantilla

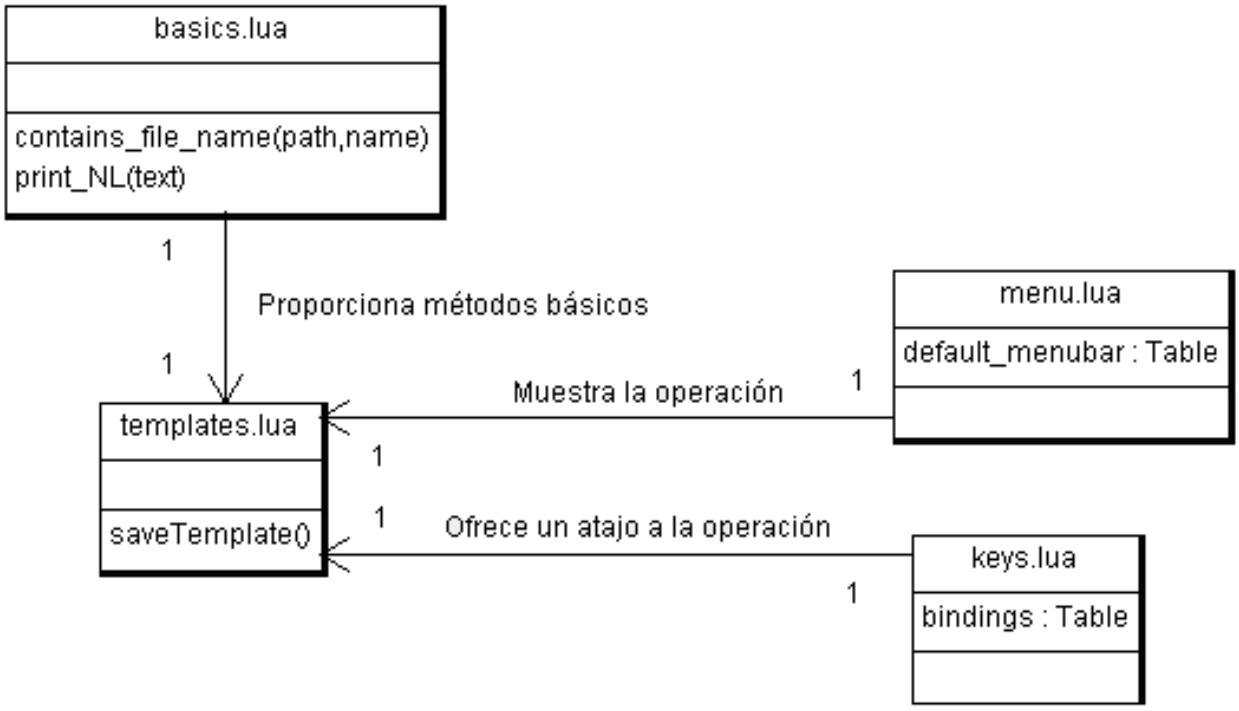


Figura 3-16. Diagrama de interfaces del caso de uso Guardar fichero como plantilla

3.5.5.2 Abrir plantillas

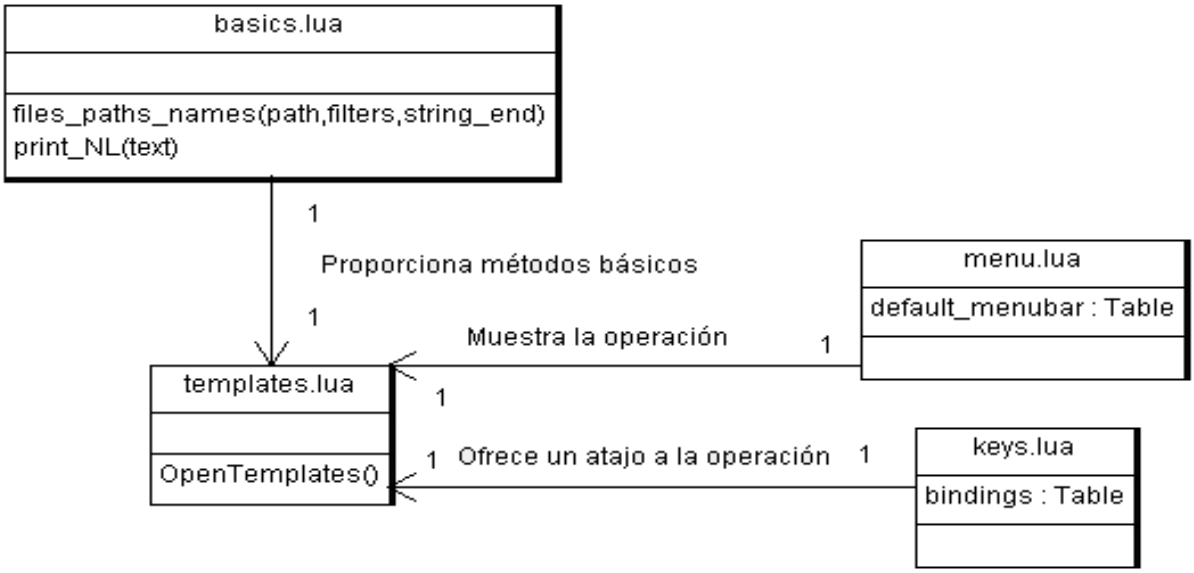


Figura 3-17. Diagrama de interfaces del caso de uso Abrir plantillas

3.5.5.3 Eliminar plantillas

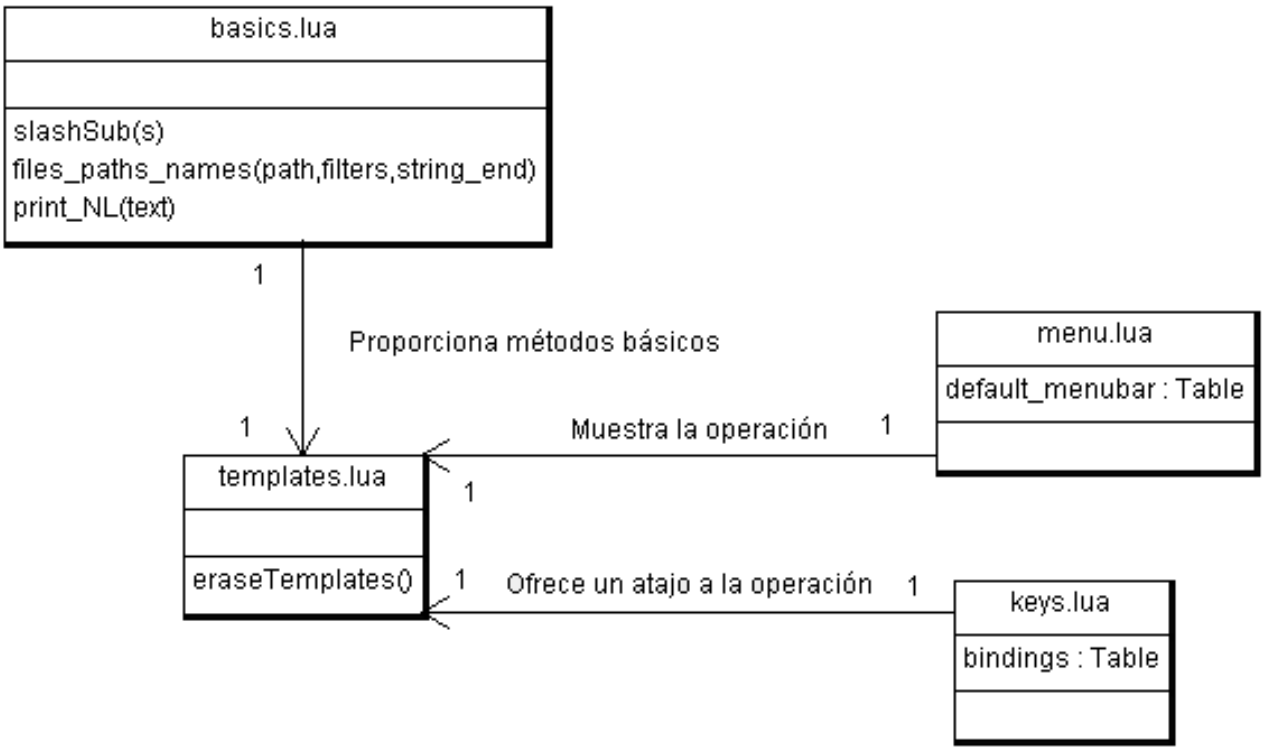


Figura 3-18. Diagrama de interfaces del caso de uso Eliminar plantillas

3.5.6 Compilar

3.5.6.1 Compilar en un solo fichero

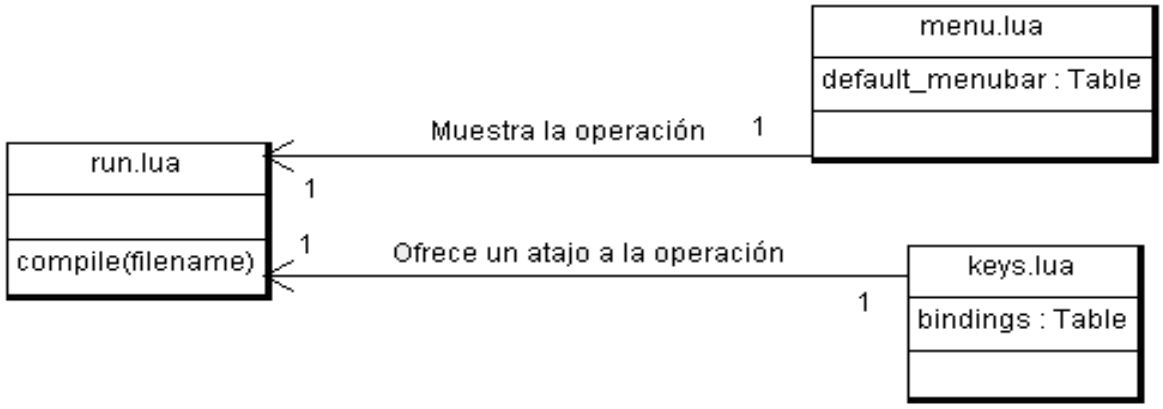


Figura 3-19. Diagrama de interfaces del caso de uso Compilar en un solo fichero

3.5.6.2 Compilar en ficheros separados

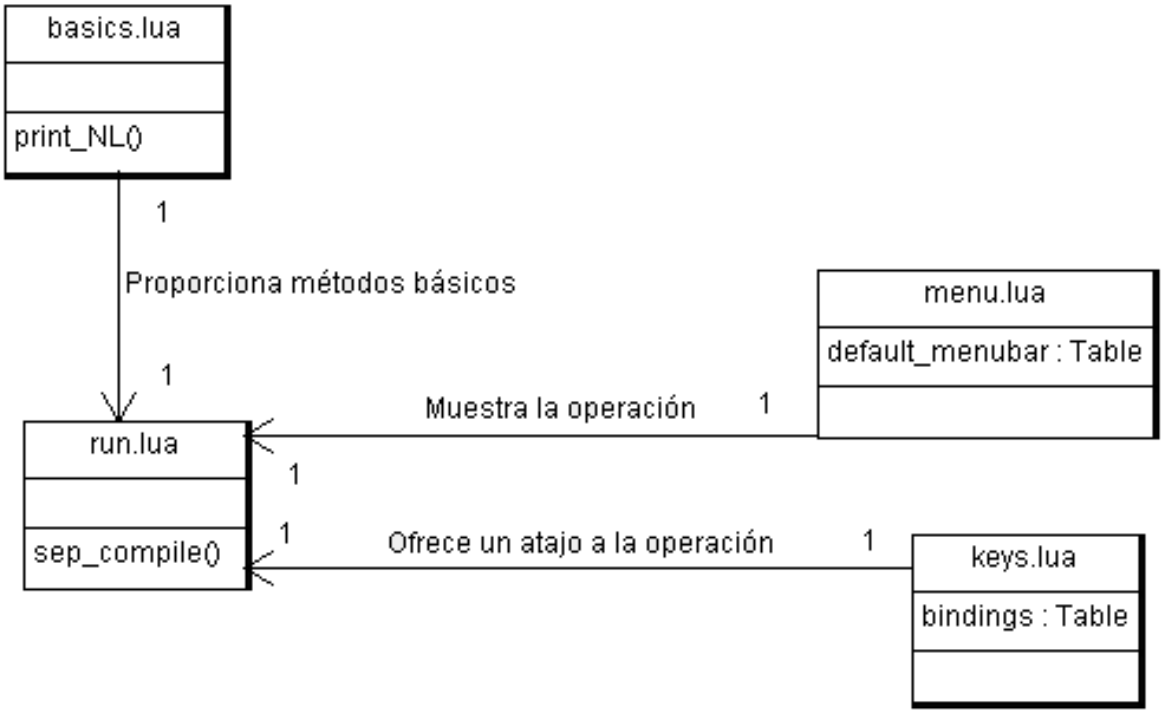


Figura 3-20. Diagrama de interfaces del caso de uso Compilar en ficheros separados

3.5.7 Ejecutar

3.5.7.1 Ejecutar completamente

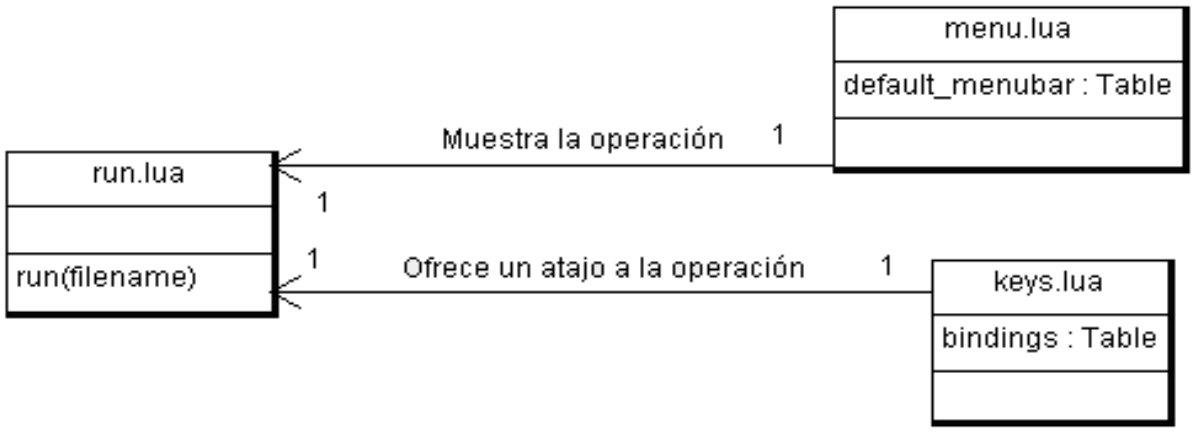


Figura 3-21. Diagrama de interfaces del caso de uso Ejecutar completamente

3.5.7.2 Ejecutar paso a paso

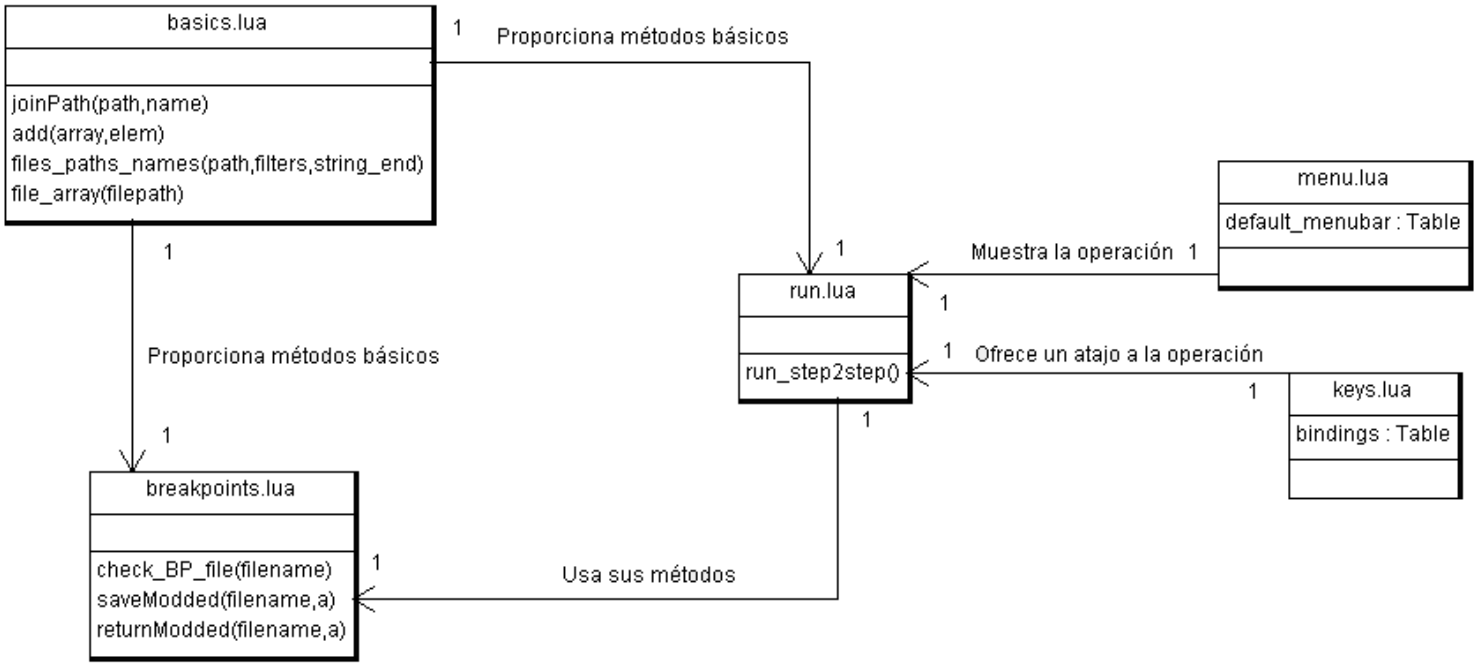


Figura 3-22. Diagrama de interfaces del caso de uso Ejecutar paso a paso

3.6 Metodología

La metodología usada es una variación del desarrollo ágil. Como sólo hay un programador, este implementa, prueba y, si no hay problemas, pasa al siguiente requisito. No se caracteriza por tomar un tiempo de plazo para implementar, sino que es variable según la dificultad de la tarea en concreto.

De las diversas metodologías ágiles que existen (¿Qué es la metodología ágil y cuáles son las más utilizadas?, 2023), la aplicada tiene muchas similitudes con el método ágil Kanban, particularmente porque:

1. Define las tareas pendientes en un documento con una lista de tareas y qué se ha hecho en cada una de ellas.
2. El progreso logrado en cada tarea se indica dentro de la sección dedicada a cada una, aparte de que se va a llevar un historial de las implementaciones y los resultados que se están logrando, que servirán como base para reflexionar si se está siguiendo un rumbo adecuado.

Capítulo 4

IMPLEMENTACIÓN

4.1 Características del lenguaje *Lua* y de *TextAdept*

El editor de texto *TextAdept* está creado sobre *Lua*, un lenguaje de programación con las siguientes características (Lua Programming in Lua (first edition), 2020):

1. Es un lenguaje imperativo.
2. Se precompila previamente y es interpretado en una máquina virtual de *Lua* (Lua, 2025).
3. Es de tipado dinámico, por lo que es necesario comprobar en tiempo de ejecución el tipo de las variables.
4. Tiene una alta capacidad de abstracción, permitiendo incluso almacenar declaraciones de funciones como elementos de un array.
5. Permite almacenar elementos en un array en cualquier posición, incluyendo negativas (esto es, denotado como -1, -2…), aunque por defecto se considera que empieza por 1.
6. *Lua* generaliza el comportamiento anterior para definir tablas, como tipos equivalentes a los *arrays* cuya clave es de tipo no numérico.
7. La invocación a módulos se hace mediante el comando *require nombre*, donde el nombre puede identificar a un archivo o a un directorio. Presenta una característica notable: no es necesario dar el camino relativo (o absoluto) exacto, sino que *Lua* buscará dentro de los directorios al alcance de la secuencia de comandos (*script*) invocando aquel archivo o directorio cuyo nombre coincida con el patrón pasado.
8. Cuenta con numerosas librerías.
    a. Por ejemplo, ***file*** provee de funciones para leer y escribir archivos en el sistema de directorios, el segundo en particular también guarda el fichero de no existir (Reading and Writing Files, 2025).
    b. Una librería para inspeccionar el sistema de ficheros del equipo, que incluye crear y eliminar directorios, ***lfs*** (Ierusalimschy, Carregal, & Guisasola, Manual, 2020).
    c. Una librería para tratar ficheros, ***io***, (Ierusalimschy, 21 – The I/O Library, 2004).

Por otro lado, *TextAdept* expande algunas de estas librerías e incluye otras específicas (Eid, Textadept 12.7 API Documentation, 2025) y en particular usa su propio sistema para cargar los módulos (Eid, Textadept 12.7 Manual, 2025), consistente en:

1. Una estructura de directorios, en la que un directorio (*modules*) contiene todas los módulos en subdirectorios exclusivos, incluyendo el módulo estándar.
2. Las secuencias de comandos de cada módulo distinguen aquellos objetos y subrutinas que deben ser accesibles desde otras mediante una tabla definida como una variable local M inicialmente vacía. Las figuras a continuación muestran un ejemplo sacado de las secuencias de comandos *run.lua* y *menu.lua*.
    a. Las líneas de *run.lua* (figura 4-1) definen una serie de objetos definidos con la sintaxis *M.objeto*. Esto es, se definen como miembros de la tabla M inicializada en la primera línea de código funcional. Como se ha indicado anteriormente, pueden ser de cualquier tipo: variables, funciones.

```
260   --- Prompts the user with the command entry to run file *filename* or the current file using an
261   -- appropriate shell command from the `textadept.run.run_commands` table.
262   -- The shell command is determined from the file's filename, extension, or language, in that order.
263   -- Emits `events.RUN_OUTPUT`.
264   -- @param[opt=buffer.filename] filename Optional path to the file to run.
265   function M.run(filename)
266           if not assert_type(filename, 'string/nil', 1) and not buffer.filename then return end
267           compile_or_run(filename or buffer.filename, M.run_commands)
268   end
```

Figura 4-1. Líneas de run.lua definiendo una función de la tabla M llamado run

    b. La última línea (figura 4-3) recoge la orden *return M*, que devuelve la tabla con todos sus objetos, para así ponerlos a disposición de otras secuencias de comandos (no todas las secuencias de comandos del módulo contienen esta orden).

```
517           return true
518   end)
519
520   -- Jump to the error or warning when double-clicking a line.
521   events.connect(events.DOUBLE_CLICK,
522           function(_, line) if is_out_buf(buffer) then M.goto_error(line) end end)
523
524   return M
525
```

Figura 4-2. La línea return M pone a disposición de otras secuencias de comandos los objetos definidos en run.lua

c. Dichos objetos estarán disponibles desde otras secuencias de comandos mediante la orden *textadept.run.NomObj*, ya que en este caso se quiere usar el objeto de nombre *NomObj* en la secuencia de comandos *run* del módulo *textadept* (figura 4-3).

```
194             {_L['Run'], textadept.run.run}, --
195             {_L['Run Setp by Step'], textadept.run.run_step2step}, --
196             {_L['Compile'], textadept.run.compile}, --
197             {_L['Compile separately'], textadept.run.sep_compile}, --
198             {_L['Build'], textadept.run.build}, --
199             {_L['Run tests'], textadept.run.test}, --
200             {_L['Run project'], textadept.run.run_project}, --
201             {_L['Stop'], textadept.run.stop},
202             {_L['Next Error'], function() textadept.run.goto_error(true) end},
203             {_L['Previous Error'], function() textadept.run.goto_error(false) end}, --
204             SEPARATOR, --
```

Figura 4-3. Invocación desde menu.lua de la misma función

3. Dentro de la librería para interaccionar con la interfaz de usuario (***ui***) hay una sublibrería de cajas de diálogo para dar información al usuario o pedírsela, ***dialogs***.
4. Extiende las librerías ***lfs*** e ***io*** con nuevas funciones.
5. Una librería para interaccionar directamente con el sistema operativo, ***os***, y así ejecutar acciones (por ejemplo, activar Git) directamente.
6. Una librería, ***view***, con funciones para la manipulación del aspecto de las ventanas.
7. Una librería, ***buffer***, con funciones para manipular los ficheros.
8. Una librería, ***events***, que permite añadir manejadores a eventos. En particular se destaca la función ***connect***, que activa una función, predefinida o definida dentro de la propia cláusula *connect*, cuando ocurre uno de los eventos reconocidos por *TextAdept*.

4.2 Implementación de las características deseadas

En las siguientes secciones se detallará cómo se implementaron las características definidas anteriormente. La explicación va a dividirse por funcionalidades, con una subsección para cada secuencia de comandos creada para implementar una familia de características, indicando al final de cada una cuáles de sus elementos de las secuencias de comandos están disponibles en su tabla M correspondiente. En algunos casos, se justifica brevemente su razón de ser.

4.2.1 Funcionalidades básicas

Hay tres secuencias de comandos relacionadas con las funcionalidades básicas: las dos secuencias de comandos *init.lua* y *basics.lua*.

4.2.1.1 Inicialización del sistema. Secuencia de comandos init.lua del directorio principal

El sistema *TextAdept* se inicializa desde la secuencia de comandos *init.lua* del directorio principal. En esta, el sistema pide cargar el módulo que se prefiera con el comando *require* ya visto en la sección 4.1. Los cambios realizados para el presente proyecto han consistido simplemente en cambiar la línea 16 (figura 4-4):

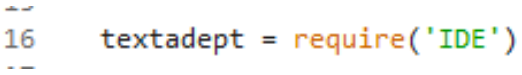


Figura 4-4. En la secuencia original la orden era textadept = require('textadept')

Y aumentar el tamaño del segundo margen en la línea 161 (figura 4-5):

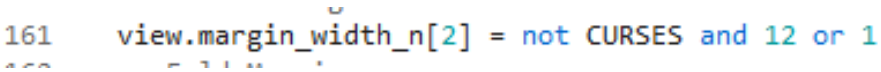
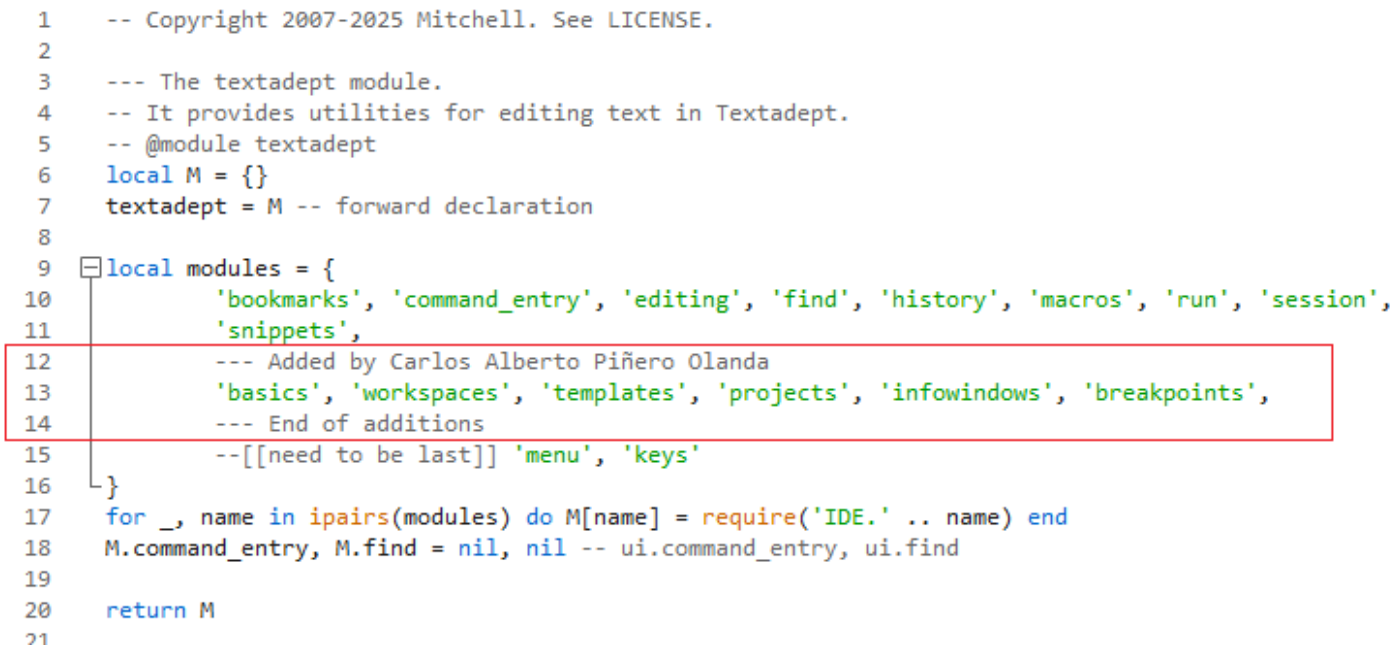


Figura 4-5. En la secuencia original la orden era view.margin_width_n[2] = not CURSES and 4 or 1

Esta secuencia de comandos carece de tabla M.

4.2.1.2 Inicialización del módulo. Secuencia de comandos init.lua del directorio del módulo

El propio módulo cargado desde la secuencia *init.lua* del anterior apartado (*IDE*) también cuenta con su otra secuencia de mismo nombre, pero mucho más simple, pues se limita a cargar las otras secuencias de comandos del mismo directorio, que son las que definen las funcionalidades del sistema. Como para este proyecto se han creado secuencias de comandos específicas para implementar las funcionalidades, la figura 4-6 resalta los cambios realizados:



```lua
-- Copyright 2007-2025 Mitchell. See LICENSE.

--- The textadept module.
-- It provides utilities for editing text in Textadept.
-- @module textadept
local M = {}
textadept = M -- forward declaration

local modules = {
	'bookmarks', 'command_entry', 'editing', 'find', 'history', 'macros', 'run', 'session',
	'snippets',
	--- Added by Carlos Alberto Piñero Olanda
	'basics', 'workspaces', 'templates', 'projects', 'infowindows', 'breakpoints',
	--- End of additions
	--[[need to be last]] 'menu', 'keys'
}
for _, name in ipairs(modules) do M[name] = require('IDE.' .. name) end
M.command_entry, M.find = nil, nil -- ui.command_entry, ui.find

return M
```

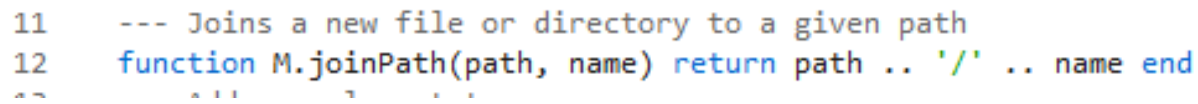
Figura 4-6. Código de la secuencia de comandos init.lua del módulo IDE. Los cambios realizados son los situados entre las líneas 12-14

En su tabla M entran dos campos que no guardan relación con el trabajo hecho.

4.2.1.3 Funciones de uso común. Secuencia de comandos basics.lua

Para facilitar el trabajo en las demás secuencias de trabajo, se ha creado una llamada *basics.lua* que reúne una serie de funciones de uso común que facilitan el trabajo, En particular, estas permiten:

1. Unir dos cadenas de texto en una ruta (figura 4-7):



```lua
--- Joins a new file or directory to a given path
function M.joinPath(path, name) return path .. '/' .. name end
```

Figura 4-7. Función que une dos cadenas de texto en una ruta con el carácter /

2. Añadir un elemento a un *array*, ya que *Lua* no parece ofrecer ninguna alternativa predefinida como consecuencia de la libertad para definir la posición, explicada en la sección 4.1 (figura 4-8).

```
13    --- Adds an element to an array
14    function M.add(array, elem)
15            array[#array + 1] = elem
16            return array
17    end
```

Figura 4-8. Función que añade un elemento a un array

3. Sustituir el carácter «\» por «/» en el nombre de una ruta para evitar errores de comparación por usar el primero, ya que no todos los sistemas operativos usan el mismo carácter para separar los nombres de los directorios en una ruta (figura 4-9).

```
18    --- Replaces backslashes by normal slashes on a file path name, to make comparisons
19    function M.slashSub(s) return s:gsub('\\', '/') end
```

Figura 4-9. Función que sustituye caracteres en el nombre de las rutas para facilitar comparaciones

4. Comprobar si existe una ruta, indicando si se busca un fichero o un directorio (figura 4-10).

```
20    --- Checks whether a path of a file or directory exists
21    function existsPath(filepath, isFile)
22            mode = 'file'
23            if not isFile then mode = 'directory' end
24            return (lfs.attributes(filepath, "mode") == mode)
25    end
```

Figura 4-10. Función que comprueba la existencia de una ruta con la indicación de si es un fichero o un directorio

5. Comprobar si existe un fichero determinado en un directorio (figura 4-11).

```
26    --- Checks whether a directory contains a file with a determinate name
27    function M.contains_file_name(path, name)
28            existing_files = lfs.walk(path)
29            name2 = path .. '/' .. name
30            name3 = name2:gsub('/', '\\')
31            name3 = string.lower(name3)
32            for file in existing_files do
33                    file = file:gsub('/', '\\')
34                    if string.lower(file) == name3 and existsPath(file, true) then return true end
35            end
36            return false
37    end
```

Figura 4-11. Función que comprueba si existe un fichero determinado

6. Comprobar si existe un directorio determinado en un directorio (figura 4-12).

```
38    --- Checks whether a directory contains a subdirectory with a determinate name
39    function M.contains_dir_name(path, name) return existsPath(M.joinPath(path, name), false) end
```

Figura 4-12. Función que comprueba si existe un directorio determinado

7. Devolver en dos arrays los nombres y rutas de los ficheros de un directorio con los filtros que se prefieran (figura 4-13).

```
40    --- Reads every file on a directory and returns two arrays, the first containing paths, the second, only names
41    function M.files_paths_names(path, filters, string_end)
42            path_len = #path + 2
43            existing_files = lfs.walk(path, filters, 0)
44            paths, names = {}, {}
45            for file in existing_files do
46                    if existsPath(file, true) then
47                            M.add(paths, file)
48                            M.add(names, string.sub(file, path_len, string_end))
49                    end
50            end
51            return paths, names
52    end
```

Figura 4-13. Función que devuelve en dos arrays los nombres y rutas de los ficheros de una directorio

8. Devolver en dos arrays los nombres y rutas de los subdirectorios de un directorio con los filtros que se prefieran (figura 4-14).

```
53   --- Reads every subdirectory on a directory and returns two arrays, the first containing paths, the second, only names
54  function M.dirs_paths_names(path, filters)
55          path_len = #path + 2
56          existing_files = lfs.walk(path, filters, 0, true)
57          paths, names = {}, {}
58          for file in existing_files do
59                  if existsPath(file, false) then
60                          M.add(paths, file)
61                          M.add(names, string.sub(file, path_len))
62                  end
63          end
64          return paths, names
65  end
```

Figura 4-14. Función que devuelve en dos arrays los nombres y rutas de los directorios de una directorio

9. Obtener un *array* con el contenido de un fichero (figura 4-15).

```
66    --Reads a file and returns an array with its lines
67  function M.file_array(filepath)
68          file = io.open(filepath,"r")
69          local lines = file:lines()
70          local a = {}
71          for line in lines do M.add(a, line) end
72          file:close()
73          return a
74  end
```

Figura 4-15. Función que devuelve un array con el contenido de un fichero

10. Sobreescribir un fichero con el contenido de un *array* (si no existe, lo crea) (figura 4-16).

```lua
--- Overwrites a file whose path is given or creates it if it does not exist
function M.write_file(filepath, a)
	file = io.open(filepath,"w")
	if #a == 0 then
		file:close()
		return
	end
	file:write(a[1])
	file:close()
	file = io.open(filepath,"a")
	for i = 2, #(a) do
		file:write('\n' .. a[i])
	end
	file:close()
end
```

Figura 4-16. Función que sobreescribe un fichero con el contenido de un array

11. Añadir un elemento en un fichero con datos ordenados procurando no repetirlos. Devuelve un valor booleano de cierto si se ha incluido y de falso si el elemento ya existía (figura 4-17).

```lua
--- Reads a data file and checks whether a given name exists. If not, it adds it and returns true, and false otherwise
function M.add_name_list(filename, name)
	local nameLow = string.lower(name)
	local file = io.open(filename,"r")
	local lines = file:lines()
	local a = {}
	local pos = 1;
	for line in lines do
		local lineLow = string.lower(line)
		if lineLow == nameLow then
			file:close()
			return false
		elseif lineLow > nameLow then
			a[#a + 2] = line
		else
			M.add(a, line)
			pos = pos + 1
		end
	end
	a[pos] = name
	file:close()
	textadept.basics.write_file(filename, a)
	return true
end
```

Figura 4-17, Función que añade un elemento a una lista ordenada si no está incluido

12. Eliminar un elemento de un fichero con datos ordenados y no repetidos (figura 4-18).

```
114    --- Reads a data file and erases a given name
115    function M.erase_name_list(filename, name)
116            local nameLow = string.lower(name)
117            local file = io.open(filename,"r")
118            local lines = file:lines()
119            local a = {}
120            for line in lines do
121                    local lineLow = string.lower(line)
122                    if lineLow ~= nameLow then M.add(a, line) end
123            end
124            file:close()
125            textadept.basics.write_file(filename, a)
126    end
```

Figura 4-18. Función que eliminar un elemento de un fichero con datos ordenados y no repetidos

13. Imprimir un mensaje en la consola de resultados con punto y aparte (figura 4-19).

```
127    --- Adds a new line after printing silently
128    function M.print_NL(text) ui.output_silent(text .. '\n') end
```

Figura 4-19. Función que imprime un mensaje en la consola de resultados con punto y aparte

Pertenecen a la tabla M todas las funciones excepto 4, que es auxiliar.

4.2.2 Espacios de trabajo. Secuencia de comandos *worspaces.lua*

Los IDEs definen espacios virtuales para gestionar el almacenamiento de proyectos. Estos espacios suelen cumplir las siguientes características:

1. Cada proyecto tiene un nombre único dentro del espacio de trabajo.
2. Cada proyecto tiene una dirección asignada dentro del equipo. Estas direcciones no tienen que estar contenidas dentro del mismo directorio, aunque es posible definir uno por defecto.
3. Se pueden importar directorios existentes como proyectos al espacio de trabajo.
4. Asimismo, es posible eliminar proyectos del espacio de trabajo y, opcionalmente, eliminar sus directorios y ficheros del sistema operativo.

Si bien no estaba incluido en los requisitos del proyecto, es aconsejable incluir la gestión de espacios de trabajo, ya que facilita la administración de proyectos. Controlar correctamente esta

característica ha precisado de la creación de la secuencia de comandos *workspace* dentro del módulo *IDE*, encargada de:

1. Crear en el directorio principal del IDE un subdirectorio específico para los espacios de trabajo con la siguiente estructura de datos (figura 4-20):

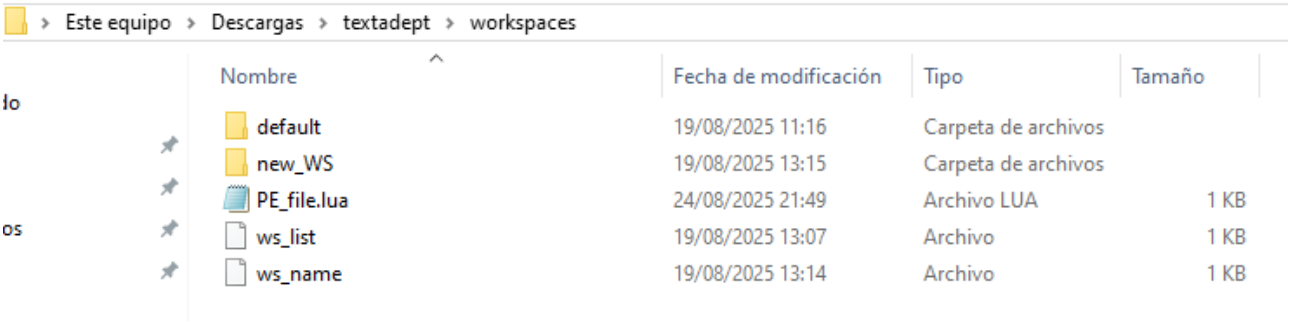


Figura 4-20. Detalle del contenido del subdirectorio específico para los espacios de trabajo

   a. Un fichero de texto que registre el espacio de trabajo usado por última vez (figura 4-21).

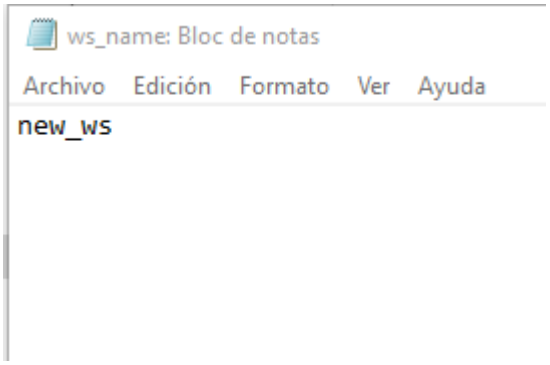


Figura 4-21. Contenido del fichero que registra el nombre del último espacio de trabajo en uso

b. Una lista de los espacios de trabajo registrados vez (figura 4-22).

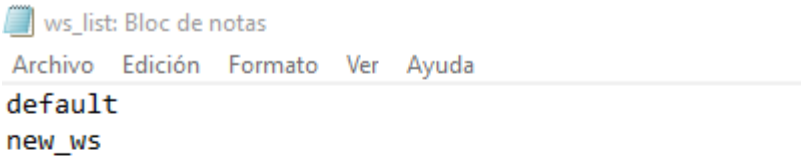


Figura 4-22. Contenido de la lista de espacios de trabajo registrados

c. Un directorio para cada espacio de trabajo existente con un fichero para registrar el nombre de sus proyectos. Si un espacio de trabajo se elimina, el directorio permanece. En la figura 4-20, presentada en el punto 1, se pueden observar dos subdirectorios llamados *default* y *new_WS*.

2. Para realizar lo anterior, en *workspaces.lua* se han implementado las siguientes operaciones:

   a. Definir variables locales para el directorio principal del sistema, para el nombre del espacio de trabajo actual (asignándole inicialmente un nombre por defecto), para la ruta del directorio *workspace* y la de los ficheros del anterior punto (figura 4-23).

```
10    --- Defines the directory for workspaces
11    M.root_path = textadept.basics.slashSub(lfs.currentdir())
12    ws_dir_name = 'workspaces'
13    ws_dir = textadept.basics.joinPath(M.root_path, ws_dir_name)
```

Figura 4-23. Instrucciones iniciales de workspaces.lua

b. Comprobar si existe la estructura de datos del primer punto y crearla en caso negativo. En caso afirmativo, leer el nombre del último espacio de trabajo registrado (figura 4-24).

```
14  --- Creates the directory for workspaces if it doesn't exist
15  lfs.mkdir(ws_dir)
16  --- Defines the name of the current workspace and the files of the workspace data architecture
17  local default_ws = 'default'
18  M.ws_name = default_ws
19  ws_file_n = 'ws_name'
20  ws_file = textadept.basics.joinPath(ws_dir, ws_file_n)
21  ws_list_n = 'ws_list'
22  ws_list = textadept.basics.joinPath(ws_dir, ws_list_n)
23  project_list = 'projects'
24  --- Reads the name of the last used wiorkspace if the name file exists, creates the file otherwise
25  if textadept.basics.contains_file_name(ws_dir, ws_file_n) then
26          file = io.open(ws_file,"r")
27          lines = file:lines()
28          for line in lines do M.ws_name = line end
29          file:close()
30  else textadept.basics.write_file(ws_file, {M.ws_name}) end
31  --- Creates the file of the workspace list if it doesn't exist
32  if not textadept.basics.contains_file_name(ws_dir, ws_list_n) then
33          textadept.basics.write_file(ws_list, {default_ws})
34  end
35  --- Creates the directory containing the projects of the current workspace if it doesn't exist
36  lfs.mkdir(textadept.basics.joinPath(ws_dir, M.ws_name))
37  --- Creates the file listing the projects of the current workspace if it doesn't exist
38  if not textadept.basics.contains_file_name(textadept.basics.joinPath(ws_dir, M.ws_name), project_list) then
39          textadept.basics.write_file(textadept.basics.joinPath(
40          textadept.basics.joinPath(ws_dir, M.ws_name), project_list), {})
41  end
```

Figura 4-24. Instrucciones de workspaces.lua para crear la estructura de datos

c. Tres funciones para gestionar el espacio de trabajo, encargadas de:

   i. Registrar un nuevo espacio de trabajo, cuyo nombre debe ser único.

      1. Se pide al usuario un nombre para el espacio de trabajo (*dialogs*).
      2. Se comprueba si es único (con *basics.lua*).
      3. Se crea un subdirectorio para el nuevo espacio de trabajo (*lfs*) en el directorio de los espacios de trabajo en caso de que no hubiera ya uno con el mismo nombre.
      4. Se comprueba si existe el fichero de proyectos. En caso negativo, se crea (*basics.lua*).

La figura 4-25 muestra la implementación.

```lua
--- Creates a workspace. It asks the user a new name and adds if it's new'
function M.create_WS()
        local ws_name = ui.dialogs.input{title = 'Write a name for the workspace.\nIt must be a new different name.'}
        if ws_name == nil then
                textadept.basics.print_NL("Canceled operation.")
                return
        end
        if not textadept.basics.add_name_list(ws_list, ws_name) then
                textadept.basics.print_NL('There is already a workspace with that name.')
                return
        end
        new_dir = textadept.basics.joinPath(ws_dir, ws_name)
        local dir = lfs.mkdir(new_dir)
        textadept.basics.print_NL('Workspace ' .. ws_name .. ' has been created.')
        if not dir then textadept.basics.print_NL('There was already a directory with the same name.') end
        if not textadept.basics.contains_file_name(new_dir, project_list) then
                textadept.basics.write_file(textadept.basics.joinPath(new_dir, project_list), {})
        end
end
```

Figura 4-25. Implementación de la función que crea espacios de trabajo

ii. Eliminar un espacio de trabajo. Esta opción jamás supone la eliminación de su directorio específico y los proyectos contenidos. Nunca se podrá eliminar el espacio de trabajo definido por defecto.

    1. Se pide que se seleccione un nombre (*dialogs*) de la lista de espacios de trabajo (*basics.lua*).
    2. Se elimina el fichero correspondiente (*basics.lua*).
    3. Si el espacio eliminado fuera el actual, se cambia la variable del nombre al espacio por defecto, se guarda (*basics.lua*) y se invoca la función que actualiza el explorador de proyectos.

La figura 4-26 muestra la implementación.

```lua
--- Erases a workspace except the default workspace. Does not erase the projects
function M.erase_WS()
	local ws_names = textadept.basics.file_array(ws_list)
	local a = {}
	for i = 1, #(ws_names) do if ws_names[i] ~= default_ws then textadept.basics.add(a, ws_names[i]) end end
	if #(a) == 0 then
		ui.dialogs.message{title = 'Warning', text = 'No workspaces to erase, except ' .. default_ws .. '.', button1 = 'Accept'}
		return
	end
	option = ui.dialogs.list{title = 'List of workspaces',
	columns = {'Select the workspace you want to erase.\nNote: THIS OPERATION DOESN\'T ERASE THE PROJECTS CONTAINED IN THE WORKSPACES.'},
	items = a}
	if option == nil then
		textadept.basics.print_NL('Canceled operation.')
		return
	end
	erased_ws = a[option]
	textadept.basics.erase_name_list(ws_list, erased_ws)
	textadept.basics.print_NL('Workspace ' .. erased_ws .. ' has been erased.')
	if M.ws_name == erased_ws then
		M.ws_name = default_ws
		textadept.basics.print_NL('Current workspace has been erased. The new workspace is ' .. default_ws .. '.')
		textadept.basics.write_file(ws_file, {M.ws_name})
		--- Updates PE
		textadept.infowindows.updatePE()
	end
end
```

Figura 4-26. Implementación de la función que elimina espacios de trabajo

iii. Cambiar el espacio de trabajo en uso.

1. Se pide que se seleccione un nombre (*dialogs*) de la lista de espacios de trabajo (*basics.lua*).
2. Se cambia el valor de la variable del nombre del espacio de trabajo actual y se guarda (*basics.lua*).
3. Se comunica el cambio de espacio de trabajo (*ui*).
4. Se invoca la función que actualiza el explorador de proyectos.

La figura 4-27 muestra la implementación.

```
89   --- Opens a workspace
90   function M.open_WS()
91           local ws_names = textadept.basics.file_array(ws_list)
92           local option = ui.dialogs.list{title = 'List of workspaces', columns = {'Select the workspace you want to open.'},
93           items = ws_names}
94           if option == nil then
95                   textadept.basics.print_NL('Canceled operation.')
96                   return
97           end
98           M.ws_name = ws_names[option]
99           textadept.basics.write_file(ws_file, {M.ws_name})
100          textadept.basics.print_NL('Current workspace is ' .. M.ws_name .. '.')
101          --- Updates PE
102          textadept.infowindows.updatePE()
103  end
```

Figura 4-27. Implementación de la función que cambia el espacio de trabajo en uso

d. Tres funciones para facilitar que otras secuencias de comandos conozcan algunas de las variables creadas:

   i. La ruta del directorio de los espacios de trabajo (figura 4-28).

```
104    --- Returns the path of the directory of all workspaces
105    function M.returnWSdir() return ws_dir end
```

Figura 4-28. Función que devuelve la ruta del directorio de los espacios de trabajo

   ii. La ruta del directorio del espacio de trabajo en uso (figura 4-29).

```
106    --- Returns the path of the current workspace directory
107    function M.returnWS() return textadept.basics.joinPath(ws_dir, M.ws_name) end
```

Figura 4-29. Función que devuelve la ruta del directorio del espacio de trabajo en uso

   iii. La ruta del fichero de proyectos del espacio de trabajo en uso (figura 4-30).

```
108    --- Returns the path of the current workspace project list
109    function M.returnPL() return textadept.basics.joinPath(M.returnWS(), project_list) end
```

Figura 4-30. Función que devuelve la ruta del directorio de los espacios de trabajo

Pertenecen a la tabla M:

1. La ruta del directorio base
2. El nombre del espacio de trabajo.
3. Las funciones de gestión de espacios de trabajo.
4. Las tres funciones para recuperar variables internas.

4.2.3 Gestión de proyectos

TextAdept tiene capacidad de trabajar con proyectos, siempre que el directorio principal esté bajo una de las formas reconocidas de control de versiones (Eid, Textadept 12.7 Manual, 2025), como *Git*, la elegida por su libre licencia y facilidad de uso.

Aparte de este tecnicismo, lo único que necesitaba *TextAdept* era tener una gestión de proyectos avanzada, es decir:

1. La gestión de proyectos propiamente dicha: Creación, consulta, actualización y eliminación (gestión *CRUD*, en inglés *Create*, *Retrieve*, *Update*, *Delete*).
2. Una ventana que muestre la información de los proyectos existentes en el espacio de trabajo y sus ficheros: un explorador de proyectos. Para ello, se crea un fichero de extensión *lua* relacionado con la gestión de proyectos en el directorio de los espacios de trabajo, creándose si no existiera al inicializar el sistema.

A continuación, se explican las funcionalidades relacionadas con cada parte.

4.2.3.1 Gestión propiamente dicha. Secuencia de comandos projects.lua

Hay seis operaciones relacionadas, tres con los proyectos y tres con sus ficheros, que se han implementado en la secuencia de comandos *projects.lua*-. Las seis son funciones:

1. Crear un directorio con un repositorio de *Git* y su registro en el espacio de trabajo en uso de su nombre.
    a. Se pide al usuario un nombre de proyecto (*dialogs*).
    b. Se comprueba si es único y se registra en el fichero de proyectos del espacio de trabajo (*basics.lua* y *workspaces.lua*).
    c. Se crea un directorio con el nombre elegido dentro de la ruta (*lfs*) si no existía ya uno.
    d. Se inicia un repositorio de *Git* (*os*).
    e. Se invoca la función que actualiza el explorador de proyectos.

La figura 4-31 muestra la implementación.

```lua
--- Starts a project
function M.startProject()
	local project_name = ui.dialogs.input{title = 'Write a name for the project.\nIt must be unique within workspace.'}
	if project_name == nil then
		textadept.basics.print_NL("Canceled operation.")
		return
	end
	ws_path = textadept.workspaces.returnWS()
	full_path = textadept.basics.joinPath(ws_path, project_name)
	if not textadept.basics.add_name_list(textadept.workspaces.returnPL(), project_name) then
		ui.dialogs.message{title = 'Warning',
		text = 'The workspace contains a project of the same name.\nPlease write other name.',
		button1 = 'Accept'}
		return
	end
	textadept.basics.print_NL('Project "' .. project_name .. '" created in workspace "' .. textadept.workspaces.ws_name .. '".')
	local dir = lfs.mkdir(full_path)
	if not dir then textadept.basics.print_NL('There was already a directory with the same name in the workspace directory.') end
	if not textadept.basics.contains_dir_name(full_path, '/.git') then
		proc = os.spawn('git init', full_path)
		proc:wait()
	end
	--- Updates The Project Explorer and, if open, reloads it
	textadept.infowindows.updatePE()
end
```

Figura 4-31. Implementación de la función que crea un proyecto

2. Eliminar el nombre del proyecto del espacio de trabajo y, opcionalmente, el propio directorio con todos sus ficheros.
    a. Se presenta al usuario la lista de proyectos del espacio de trabajo para que elija uno (*basics.lua*, *workspaces.lua* y *dialogs*).
    b. Se elimina el nombre del fichero de proyectos del espacio de trabajo (*basics.lua*).
    c. Se le pregunta si quiere eliminar el directorio relacionado (*dialogs*), aplicando una función auxiliar recursiva para eliminar primero los ficheros y luego los directorios (*lfs*).
    d. Si se eligiera eliminar los archivos, se cierran sus archivos abiertos (*buffer*).
    e. Se invoca la función que actualiza el explorador de proyectos.

La figura 4-32 muestra la implementación y la 4-33, la función auxiliar recursiva.

```lua
  --- Erases a project
function M.eraseProject()
	local projects = textadept.workspaces.returnPL()
	local names = textadept.basics.file_array(projects)
	if #(names) == 0 then
		ui.dialogs.message{title = 'Warning',
		text = 'No projects exist in workspace' .. textadept.workspaces.ws_name .. '.',
		button1 = 'Accept'}
		return
	end
	option = ui.dialogs.list{title = 'List of projects in workspace ' .. textadept.workspaces.ws_name .. '.',
	columns = {'Select the project you want to erase.'}, items = names}
	if option == nil then
		textadept.basics.print_NL('Canceled operation.')
		return
	end
	local proj_name = names[option]
	textadept.basics.erase_name_list(projects, proj_name)
	textadept.basics.print_NL('Project "' .. proj_name .. '" erased from workspace ' .. textadept.workspaces.ws_name .. '.')
	--- Updates The Project Explorer and, if open, reloads it
	textadept.infowindows.updatePE()
	delete = ui.dialogs.message{title = 'Decide what to do with project directory.',
			text = 'Do you wish to erase the directory of the project?\nThis operation CANNOT BE UNDONE.',
			icon = 'dialog-question', button1 = 'Erase.', button2 = 'Reject.'}
	if delete == 2 then
		textadept.basics.print_NL('The directory of the project ' .. proj_name .. ' has not been erased.')
		return
	end
	--- Closing open files from the erased project
	local proj_dir = textadept.basics.joinPath(textadept.workspaces.returnWS(), proj_name)
	local files_paths, _ = textadept.basics.files_paths_names(proj_dir)
	for f = 1, #files_paths do
		for i = 1, #_BUFFERS do
			local buffer = _BUFFERS[i]
			local name = buffer.filename or buffer._type or _L['Untitled']
			if textadept.basics.slashSub(name) == textadept.basics.slashSub(files_paths[f]) then
				buffer:close(true)
				break
			end
		end
	end
	erase_dir(proj_dir)
	textadept.basics.print_NL('The directory of the project ' .. proj_name .. ' has been erased.')
end
```

Figura 4-32. Implementación de la función que elimina un proyecto

```lua
  --- Erases a directory with all its contents
local function erase_dir(path)
	existing_files = lfs.walk(path, lfs.default_filter, 0, true)
	for file in existing_files do
		if lfs.attributes(file, "mode") == "file" then os.remove(file)
		elseif lfs.attributes(file, "mode") == "directory" then erase_dir(file) end
	end
	lfs.rmdir(path)
end
```

Figura 4-33. Función auxiliar recursiva que elimina el contenido de un directorio

3. Seleccionar un directorio del sistema y añadirlo como proyecto al espacio de trabajo, creándole un repositorio en *Git* si no tuviera ninguno.

    a. Se pide al usuario que elija la ruta del proyecto a importar (*dialogs*).

    b. Se comprueba que su nombre y su ruta sean únicos, guardándose en ese caso (*basics.lua* y *workspaces.lua*). Si acaso el nombre no estuviera registrado pero la ruta corresponda a una ya incluida en el espacio de trabajo, se pregunta si se quiere importar de todos modos (*dialogs*).

    c. Se usa una función auxiliar recursiva para crear el nuevo directorio y copiar en este el contenido del original.

    d. Si no tuviera ya un repositorio de Git (*lfs*), se crea (*os*).

    e. Se invoca la función que actualiza el explorador de proyectos.

    La figura 4-34 muestra la implementación y la 4-35, la función auxiliar recursiva.

```
114   --- Imports a project onto the current workspace
115   function M.importProject()
116           local project_path = ui.dialogs.open{title = 'Select directory from system.', dir = _HOME, only_dirs = true}
117           if project_path == nil then
118                   textadept.basics.print_NL("Canceled operation.")
119                   return
120           end
121           local path_string = textadept.basics.slashSub(project_path)--:gsub('\\', '/')
122           local pp_len = #project_path
123           local project_name = ''
124           for c = 1, pp_len do
125                   if string.sub(path_string, c, c) == '/' then project_name = ''
126                   else project_name = project_name .. string.sub(project_path, c, c) end
127           end
128           local projects = textadept.workspaces.returnPL()
129           if not textadept.basics.add_name_list(projects, project_name) then
130                   ui.dialogs.message{title = 'Warning',
131                   text = 'The path contains another project with the same name.\nPlease write other name or select other path.',
132                   button1 = 'Accept.'}
133                   return
134           end
135           ws_path = textadept.workspaces.returnWS()
136           full_path = textadept.basics.joinPath(ws_path, project_name)
137           if textadept.basics.contains_dir_name(ws_path, project_name) then
138                   --If it in the same workspace, ask whether it is OK
139                   if textadept.basics.slashSub(project_path) == textadept.basics.slashSub(full_path) then
140                           option = ui.dialogs.message{title = 'Path imported from same workspace.',
141                                   text = 'The project already existed in the same workspace.\nDo you want to still import it?',
142                                   icon = 'dialog-question', button1 = 'Accept.', button2 = 'Reject.'}
143                           if option == 2 then
144                                   textadept.basics.erase_name_list(projects, project_name)
145                                   textadept.basics.print_NL("Canceled operation.")
146                                   return
147                           else textadept.basics.print_NL('Project "' .. project_name .. '" imported into the workspace ' .. textadept.workspaces.ws_name)
148                           end
149                   else
150                           textadept.basics.erase_name_list(projects, project_name)
151                           local this_text = 'There is a directory with the same name inside the workspace, but not registered.\n'
152                           this_text = this_text .. 'Please change the name of the project before importing it.'
153                           ui.dialogs.message{title = 'Path with the same route.', text = this_text, icon = 'dialog-question', button1 = 'Accept.'}
154                           return
155                   end
156           else
157                   local this_text = 'Project "' .. project_name .. '" imported from directory "' .. project_path .. '" into workspace '
158                   this_text = this_text .. 'Project "' .. textadept.workspaces.ws_name .. '.'
159                   textadept.basics.print_NL(this_text)
160                   copy_dir(project_path, full_path)
161           end
162           if not textadept.basics.contains_dir_name(full_path, '/.git') then
163                   proc = os.spawn('git init', full_path)
164                   proc:wait()
165           end
166           --- Updates The Project Explorer and, if open, reloads it
167           textadept.infowindows.updatePE()
168   end
```

Figura 4-34. Implementación de la función que importa un proyecto

```lua
--- Copy a directory with all its contents
local function copy_dir(path, new_path)
	lfs.mkdir(new_path)
	local path_len = #path + 2
	local existing_files = lfs.walk(path, lfs.default_filter, 0, true)
	local directories, files = {}, {}
	for file in existing_files do
		if lfs.attributes(file, "mode") == "file" then textadept.basics.add(files, file)
		elseif lfs.attributes(file, "mode") == "directory" then textadept.basics.add(directories, file) end
	end
	for i = 1, #(directories) do
		local dir_name = string.sub(directories[i], path_len, -2)
		local new_dir = textadept.basics.joinPath(new_path, dir_name)
		copy_dir(directories[i], new_dir)
	end
	for i = 1, #(files) do
		local file = files[i]
		local file_name = string.sub(file, path_len)
		local file_info = io.open(file,"r")
		local lines = file_info:lines()
		local a = {}
		for line in lines do textadept.basics.add(a, line) end
		file_info:close()
		textadept.basics.write_file(textadept.basics.joinPath(new_path, file_name), a)
	end
end
```

Figura 4-35. Función auxiliar recursiva que copia el contenido de un directorio

4. Respecto a los ficheros de un proyecto, las tres funciones se encargan de:
    a. Abrir ficheros de un proyecto. Es análogo a la función para abrir un espacio de trabajo, sólo que con algunos cambios:
        i. Se selecciona el proyecto (*basics.lua*, *workspaces.lua* y *dialogs*).
        ii. Se recuperan los ficheros que contenga (*basics.lua* y *workspaces.lua*).
        iii. Se seleccionan ficheros dentro del proyecto (*basics.lua*, *workspaces.lua* y *dialogs*), es posible elegir varios.
        iv. Se abren los ficheros seleccionados (*io*).

La figura 4-36 muestra la implementación.

```lua
--- Opens files from a chosen project
function M.openProjFiles()
        local projects = textadept.workspaces.returnPL()
        local project_names = textadept.basics.file_array(projects)
        if #(project_names) == 0 then
                ui.dialogs.message{title = 'Warning', text = 'No projects exist in workspace ' .. textadept.workspaces.ws_name .. '.',
                button1 = 'Accept.'}
                return
        end
        option = ui.dialogs.list{title = 'List of projects in workspace ' .. textadept.workspaces.ws_name .. '.',
        columns = {'Select the project you want to open files from.'}, items = project_names}
        if option == nil then
                textadept.basics.print_NL('Canceled operation')
                return
        end
        local project_name = project_names[option]
        local project_path = textadept.basics.joinPath(textadept.workspaces.returnWS(), project_name)
        local files_paths, files_names = textadept.basics.files_paths_names(project_path)
        if #(files_names) == 0 then
                ui.dialogs.message{title = 'Warning', text = 'No files exist in project ' .. project_name .. '.',
                button1 = 'Accept'}
                return
        end
        local options = ui.dialogs.list{title = 'List of files in project ' .. project_names[option] .. '.',
        columns = {'Select the files you want to open.'}, items = files_names, multiple = true}
        if options == nil then
                textadept.basics.print_NL('Canceled operation')
                return
        end
        for i = 1, #options do io.open_file(files_paths[options[i]]) end
end
```

Figura 4-36. Implementación de la función que abre ficheros de un proyecto

b. Guardar ficheros en un proyecto. Es parecida a la función para crear un proyecto.

   i. Se selecciona proyecto (*basics.lua*, *workspaces.lua* y *dialogs*).

   ii. Se pide un nombre para el fichero (*dialogs*).

   iii. Se comprueba si el nombre está ya en uso en el proyecto (*basics.lua* y *workspaces.lua*).

   iv. Se guarda el fichero abierto en el directorio del proyecto con el nombre seleccionado (*buffer*).

   v. Se invoca la función que actualiza el explorador de proyectos.

La figura 4-37 muestra la implementación.

```lua
--- Saves files into a chosen project
function M.saveFileProj()
	local projects = textadept.workspaces.returnPL()
	local project_names = textadept.basics.file_array(projects)
	if #(project_names) == 0 then
		ui.dialogs.message{title = 'Warning', text = 'No projects exist in workspace ' .. textadept.workspaces.ws_name .. '.',
		button1 = 'Accept'}
		return
	end
	option = ui.dialogs.list{title = 'List of projects in workspace ' .. textadept.workspaces.ws_name .. '.',
	columns = {'Select the project you want to save the file into.'}, items = project_names}
	if option == nil then
		textadept.basics.print_NL('Canceled operation')
		return
	end
	local file_name = ui.dialogs.input{title = 'Write a name for the file.\nIt must be unique within project.'}
	if file_name == nil then
		textadept.basics.print_NL('Canceled operation')
		return
	end
	local project_name = project_names[option]
	local project_path = textadept.basics.joinPath(textadept.workspaces.returnWS(), project_name)
	if textadept.basics.contains_file_name(project_path, file_name) then
		ui.dialogs.message{title = 'Warning',
		text = 'File name already exists in project' .. project_names[option] .. '.',
		button1 = 'Accept'}
		return
	end
	buffer:save_as(textadept.basics.joinPath(project_path, file_name))
	textadept.basics.print_NL('File ' .. file_name .. ' saved in project ' .. project_name .. '.')
	--- Updates The Project Explorer and, if open, reloads it
	textadept.infowindows.updatePE()
end
```

Figura 4-37. Implementación de la función que guarda un fichero en un proyecto

c. Eliminar ficheros de un proyecto. Es similar a la función ya vista de abrir ficheros, pero con la diferencia de que al segundo paso le sigue una petición de confirmación, ya que los archivos se eliminarán.

   i. Se selecciona proyecto (*basics.lua*, *workspaces.lua* y *dialogs*).

   ii. Se recuperan los ficheros que contenga (*basics.lua* y *workspaces.lua*).

   iii. Se seleccionan ficheros dentro del proyecto (*basics.lua*, *workspaces.lua* y *dialogs*), es posible elegir varios.

   iv. Se pide confirmación de la eliminación (*dialogs*).

   v. Se eliminan los ficheros seleccionados (*os*).

   vi. Se cierran aquellos abiertos (*buffer*).

   vii. Se invoca la función que actualiza el explorador de proyectos.

La figura 4-38 muestra la implementación.

```
233   --- Erases files from a chosen project
234 function M.eraseProjFiles()
235         local projects = textadept.workspaces.returnPL()
236         local project_names = textadept.basics.file_array(projects)
237         if #(project_names) == 0 then
238                 ui.dialogs.message{title = 'Warning',
239                 text = 'No projects exist in workspace ' .. textadept.workspaces.ws_name .. '.',
240                 button1 = 'Accept'}
241                 return
242         end
243         local option = ui.dialogs.list{title = 'List of projects in workspace ' .. textadept.workspaces.ws_name .. '.',
244         columns = {'Select the project you want to erase files from.'}, items = project_names}
245         if option == nil then
246                 textadept.basics.print_NL('Canceled operation')
247                 return
248         end
249         local project_name = project_names[option]
250         local project_path = textadept.basics.joinPath(textadept.workspaces.returnWS(), project_name)
251         local files_paths, files_names = textadept.basics.files_paths_names(project_path)
252         if #(files_names) == 0 then
253                 ui.dialogs.message{title = 'Warning', text = 'No files exist in project ' .. project_name .. '.',
254                 button1 = 'Accept'}
255                 return
256         end
257         options = ui.dialogs.list{title = 'List of files in project ' .. project_name .. '.',
258         columns = {'Select the files you want to erase.'},
259         items = files_names, multiple = true}
260         if options == nil then
261                 textadept.basics.print_NL('Canceled operation.')
262                 return
263         end
264         delete = ui.dialogs.message{title = 'Confirm deletion of files',
265                         text = 'Do you really want to erase the selected files?\nThis operation CANNOT BE UNDONE.',
266                         icon = 'dialog-question', button1 = 'Erase.', button2 = 'Reject.'}
267         if delete == 2 then
268                 textadept.basics.print_NL('Canceled operation.')
269                 return
270         end
271         filenames = ''
272         for i = 1, #options do
273                 filenames = filenames .. files_names[options[i]]
274                 for b = 1, #_BUFFERS do
275                         local buffer = _BUFFERS[b]
276                         local name = buffer.filename or buffer._type or _L['Untitled']
277                         if textadept.basics.slashSub(name) == textadept.basics.slashSub(files_paths[options[i]]) then
278                                 buffer:close(true)
279                                 break
280                         end
281                 end
282                 os.remove(files_paths[options[i]])
283         end
284         textadept.basics.print_NL('Files ' .. filenames .. 'erased from project ' .. project_names[option] .. '.')
285         --- Updates The Project Explorer and, if open, reloads it
286         textadept.infowindows.updatePE()
287 end
```

Figura 4-38. Implementación de la función que elimina ficheros de un proyecto

Pertenecen a la tabla M las seis funciones.

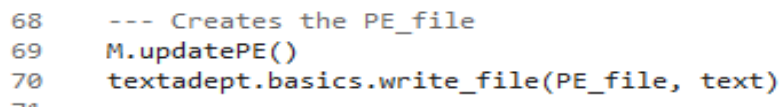
4.2.3.2 Explorador de proyectos. Secuencia de comandos *infowindows.lua*

Un explorador de proyectos es una ventana que muestra información sobre los proyectos registrados en el espacio de trabajo. Para crearlo, se hace uso de dos capacidades de *TextAdept*, abrir varias ventanas y el plegado de código cuando un fichero tiene extensión de código, por ejemplo, cuando contiene un *array*.

Para ello, se ha definido una secuencia de comandos para todas las operaciones con ventanas (como se verá en la posterior sección sobre la implementación de la consola de resultados), ***infowindows.lua***. Se encarga de:

1. Cuando el sistema se inicia, crear un fichero de extensión *lua* en el directorio de los espacios de trabajo, que almacena la información del directorio del último espacio de trabajo abierto y de los directorios de los proyectos registrados, guardándolos en un fichero de extensión (*basics.lua* y *workspaces.lua*). Así, se muestran los proyectos como *arrays* y sus ficheros, como sus elementos, lo que aprovecha el plegado de código de los *arrays* en *Lua*.

   La figura 4-39 muestra la implementación.

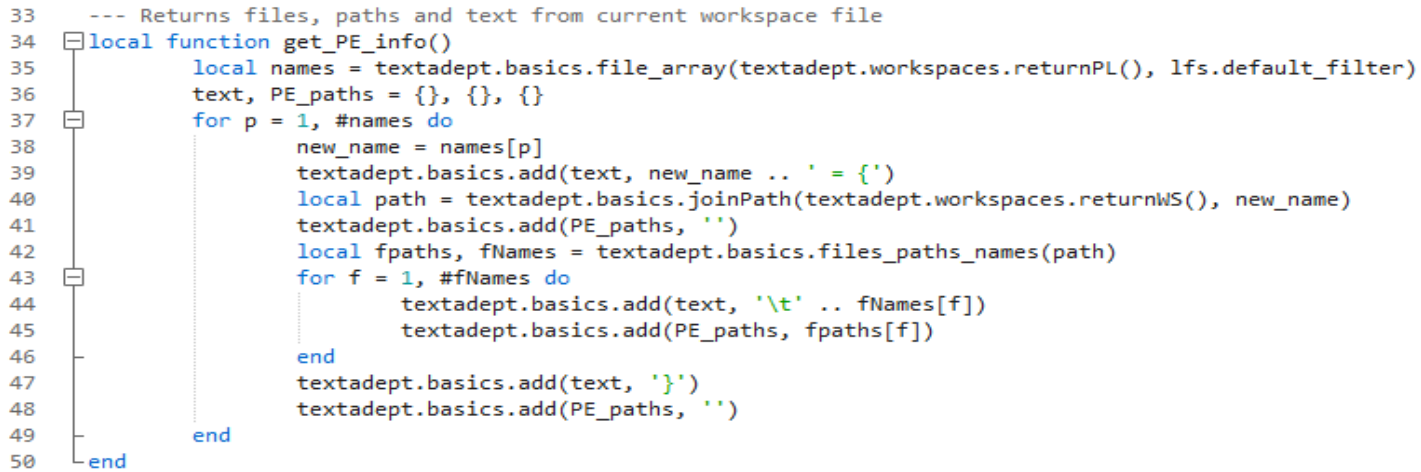


```
68    --- Creates the PE_file
69    M.updatePE()
70    textadept.basics.write_file(PE_file, text)
```

Figura 4-39. Orden que toma información del espacio de trabajo y la guarda en el fichero correspondiente

2. Definir tres funciones relacionadas con el explorador de proyectos::

   a. Una función auxiliar para obtener la información sobre el espacio de trabajo en uso (figura 4-40).



```
33  --- Returns files, paths and text from current workspace file
34  local function get_PE_info()
35          local names = textadept.basics.file_array(textadept.workspaces.returnPL(), lfs.default_filter)
36          text, PE_paths = {}, {}, {}
37          for p = 1, #names do
38                  new_name = names[p]
39                  textadept.basics.add(text, new_name .. ' = {')
40                  local path = textadept.basics.joinPath(textadept.workspaces.returnWS(), new_name)
41                  textadept.basics.add(PE_paths, '')
42                  local fpaths, fNames = textadept.basics.files_paths_names(path)
43                  for f = 1, #fNames do
44                          textadept.basics.add(text, '\t' .. fNames[f])
45                          textadept.basics.add(PE_paths, fpaths[f])
46                  end
47                  textadept.basics.add(text, '}')
48                  textadept.basics.add(PE_paths, '')
49          end
50  end
```

Figura 4-40. Función que recoge la información del espacio de trabajo en uso

b. Actualizar el explorador cuando se realicen acciones de la gestión de espacios de trabajo o de proyectos, comentadas en sus secciones (*basics.lua*, *buffer*, *view* e *io*) (figura 4-41).

```
51    --- Updates the Project Explorer when it's changed as a consequence of Project Management
52    function M.updatePE()
53            get_PE_info()
54            textadept.basics.write_file(PE_file, text)
55            if PE_buffer ~= nil then --PE_buffer:reload() end
56                    PE_buffer:close(true)
57                    view:unsplit()
58                    view:split(true)
59                    ui.goto_view(-1)
60                    io.open_file(PE_file)
61                    local buffer = _G.buffer
62                    buffer.read_only = true
63                    buffer.tab_label = 'Project Explorer of ' .. textadept.workspaces.ws_name .. '.'
64                    PE_buffer = buffer
65                    ui.goto_view(-1)
66            end
67    end
```

Figura 4-41. Función que actualiza el explorador de proyectos

c. Abrir o cerrar el propio explorador (*buffer*, *view* e *io*). Funciona así:

   i. Cuando el Explorador de Proyectos no está abierto, lo abre y divide verticalmente las ventanas, situando este a la izquierda.

   ii. Cuando el Explorador de Proyectos sí está abierto, lo cierra y ejecuta la función para deshabilitar la última división de ventanas (si se han hecho más divisiones, el aspecto final depende del orden en que se hayan hecho las divisiones).

La figura 4-42 muestra la implementación.

```
72    --- Toggles Project Explorer
73    function M.togglePE()
74            --M.updatePE()
75            if PE_buffer ~= nil then
76                    PE_buffer:close(true)
77                    view:unsplit()
78                    PE_buffer = nil
79                    return
80            end
81            view:split(true)
82            ui.goto_view(-1)
83            io.open_file(PE_file)
84            local buffer = _G.buffer
85            buffer.read_only = true
86            buffer.tab_label = 'Project Explorer of ' .. textadept.workspaces.ws_name .. '.'
87            PE_buffer = buffer
88            ui.goto_view(-1)
89    end
```

Figura 4-42. Implementación de la función que abre o cierra el explorador de proyectos

3. Conectar tres eventos:

    a. Si se pulsa con el botón izquierdo del ratón dos veces sobre el nombre de un fichero, este se abre (figura 4-43).

```
90    --- Connects a double click on the Project Explorer to opening files
91    events.connect(events.DOUBLE_CLICK, function(position, line)
92                    local buffer = _G.buffer
93                    local bufName = buffer.filename ~= nil and textadept.basics.slashSub(buffer.filename)
94                    if bufName ~= PE_file then return end
95                    local path = PE_paths[line]
96                    if path ~= '' then io.open_file(path) end
97            end
98    )
```

Figura 4-43. Conexión entre el doble click y abrir un fichero del explorador

b. La inicialización del sistema con la detección de si está abierto el explorador de proyectos (figura 4-44).

```
10  --- Variables for info windows management
11  local PE_file = textadept.basics.joinPath(textadept.workspaces.returnWSdir(), 'PE_file.lua')
12  PE_file = textadept.basics.slashSub(PE_file)
13  local text, PE_paths = {}, {}
14  local PE_buffer = nil
15  local output_buffer = nil
16  events.connect(events.INITIALIZED, function()
17              for i = 1, #_BUFFERS do
18                  local buffer = _BUFFERS[i]
19                  local name = buffer.filename or buffer._type or _L['Untitled']
20                  if textadept.basics.slashSub(name) == PE_file then
21                      buffer.read_only = true
22                      buffer.tab_label = 'Project Explorer of ' .. textadept.workspaces.ws_name .. '.'
23                      PE_buffer = buffer
24                  end
25                  if name == '[Output Buffer]' then
26                      output_buffer = buffer
27                  end
28                  if PE_buffer ~= nil and output_buffer ~= nil then break end
29              end
30      end
31  )
```

Figura 4-44. Conexión entre la inicialización del sistema y la detección del explorador de proyectos

c. Cerrar un archivo con la posibilidad de que se haya cerrado el explorador de proyectos (figura 4-45).

```
116   --- Connects closing a buffer to check whether the PE and the Output View are still open
117 events.connect(events.BUFFER_DELETED, function()
118             is_PE_buffer = nil
119             is_output_buffer = nil
120             for i = 1, #_BUFFERS do
121                 local buffer = _BUFFERS[i]
122                 local name = buffer.filename or buffer._type or _L['Untitled']
123                 if textadept.basics.slashSub(name) == PE_file then is_PE_buffer = buffer end
124                 if name == '[Output Buffer]' then is_output_buffer = buffer end
125                 if is_PE_buffer ~= nil and is_output_buffer ~= nil then break end
126             end
127             if is_PE_buffer == nil then PE_buffer = nil end
128             if is_output_buffer == nil then output_buffer = nil end
129     end
130 )
```

Figura 4-45. Conexión entre cerra un fichero y comprobar si era el explorador de proyectos

Pertenecen a la tabla M sólo la función de actualización y la de abrir o cerrar el explorador.

4.2.4 Consola de resultados. Secuencia de comandos *infowindows.lua*

TextAdept tiene una función de la librería *ui* para la aparición de una ventana de edición de resultados (técnicamente es para imprimir en esta ventana un mensaje pasado como parámetro, pero se puede ejecutar sin ninguno para hacer que aparezca).

La mayoría de IDEs se caracterizan por tener una ventana de resultados, normalmente diferente a aquellas de edición de resultados, con la posibilidad de cerrarla. En este caso se ha optado por añadir a la secuencia de comandos *infowindows.lua* (ya presentada en la anterior sección) estas funcionalidades:

1. Una función de abrir y cerrar la consola de resultados, análoga a la de abrir y cerrar el explorador de proyectos (figura 4-46).

```
100   --- Toggles output window
101   function M.toggleOutput()
102           if output_buffer ~= nil then
103                   output_buffer:close(true)
104                   view:unsplit()
105                   output_buffer = nil
106                   return
107           end
108           view:split()
109           ui.goto_view(-1)
110           ui.output("Output view. Results will be shown here. Beware that it can change its position.\n")
111           local buffer = _G.buffer
112           output_buffer = buffer
113           ui.goto_view(-1)
114   end
```

Figura 4-46. Implementación de la función que abre o cierra la consola de resultados

2. Dos de los eventos conectados comentados en la sección anterior del explorador de proyectos, los de detectar si este está abierto o cerrado, cuando se inicializa el sistema o se cierra un fichero también lo están para la consola de resultados. Se muestran en las figuras 4-44 y 4-45.

Sólo la función que abre o cierra la consola de resultados pertenece a la tabla M.

4.2.5 Compilación separada. Secuencia de comandos *run.lua*

TextAdept incluye, como se ha comentado antes, compilación de *C* y Java entre otros. No obstante, no es el caso de la compilación separada para lenguajes como *C* (en el caso de *Java*, su manera de compilar es separada por definición).

Por ello, se ha modificado la secuencia de comandos de *TextAdept* encargada de la compilación y ejecución de programas, *run.lua*. El cambio consiste en una sola función, basada en la función de compilación nativa y añadida a la tabla M, que realiza lo siguiente:

1. Usando los comandos de reconocimiento de ficheros, se realiza un barrido en el fichero de compilación con el patrón de la extensión de *C* (la única considerada) y se crean los archivos objeto de extensión *o*.
2. Con los archivos objeto ya creados, se pueden compilar todos de un modo análogo a la compilación estándar.
3. Se comunica al usuario si la compilación ha sido o no exitosa.

La figura 4-47 muestra la implementación:

```
210   -- Compiles separately a group of files
211   -- Based on the above function, from line 182
212   M.sep_compile_commands = {c={'gcc -Wall -g -c', 'gcc -o ', ' -lm', '.o'},java={'javac "%f"'},lua={'luac -o "%e.luac" "%f"'}}
213   function M.sep_compile()
214           filename = buffer.filename
215           local ext = filename:match('[^/\\.]+$')
216           local lang = filename == buffer.filename and buffer.lexer_language or lexer.detect(ext)
217           local command = M.sep_compile_commands[filename] or M.sep_compile_commands[ext] or M.sep_compile_commands[lang]
218           if #command == 1 then
219                   M.compile(filename)
220                   return
221           end
222           local dirname, basename = '', filename
223           if filename:find('[/\\]') then dirname, basename = filename:match('^(.+)[/\\]([^/\\]+)$') end
224           -- This loop searches every implementarton file in the directory and generates the object program
225           dirL = #dirname + 2
226           files = ''
227           for file in lfs.walk(dirname, {'.' .. ext}, 0) do
228                   object_file = ' ' .. string.sub(file, dirL)
229                   proc = os.spawn(command[1] .. object_file, dirname)
230                   proc:wait()
231                   files = files .. string.sub(object_file, 1, -2 - #ext) .. command[4]
232           end
233           -- Finally, the program generates the executable from the implementation lines
234           exec_name = string.sub(basename, 1, -2 - #ext)
235           proc = os.spawn(command[2] .. exec_name .. files .. command[3], dirname)
236           proc:wait()
237           exec_path = dirname .. '/' .. exec_name
238           if lfs.attributes(exec_path, "mode") == "file" or lfs.attributes(exec_path .. ".exe", "mode") == "file" then
239                   textadept.basics.print_NL('Successful separate compilation.')
240           else textadept.basics.print_NL('Separate compilation has failed.') end
241   end
```

Figura 4-47. Implementación de la función que realiza la compilación separada

Dicha función se ha añadido a la tabla M de la secuencia de comandos, que incluye de base otras funciones y variables.

4.2.6 Gestión de plantillas. Secuencia de comandos *templates.lua*

El trabajo con proyectos hace deseable tener disponibles diversos tipos de ficheros que puedan modificarse fácilmente para nuevos propósitos, es decir, plantillas. Su creación ha sido bastante sencilla, comparada con los pasos anteriores, necesitando sólo una secuencia de comandos, *templates.lua*.

Ha consistido en:

1. Crear un directorio específico para las plantillas, ***templates***. Se crea inicialmente si no existe (figura 4-48).

```
10    --- Variables for template management
11    local template_dir = textadept.workspaces.root_path .. '/templates'
12    --- Creates the directory for templates if it doesn't exist
13    lfs.mkdir(template_dir)
```

Figura 4-48. Instrucciones iniciales de templates.lua

2. Tres funciones para gestionar las plantillas, encargas de:
    a. Guardar el documento abierto como plantilla (análogo a ***Save As File in Project***).
        i. Se pide un nombre para la plantilla (*dialogs*).
        ii. Se comprueba si hay una plantilla con ese nombre (*basics.lua*).
        iii. Se guarda el fichero abierto en el directorio del proyecto con el nombre seleccionado (*buffer*).

    La figura 4-49 muestra la implementación.

```
15    --- Saves a files as a template
16  function M.saveTemplate()
17          local templ_name = ui.dialogs.input{title = 'Write a name for the template.\nIt must be a new different name.'}
18          if templ_name == nil then
19                  textadept.basics.print_NL("Canceled operation.")
20                  return
21          end
22          if textadept.basics.contains_file_name(template_dir, templ_name) then
23                  textadept.basics.print_NL("There is already a template with that name.")
24                  return
25          end
26          buffer:save_as(template_dir .. '/' .. templ_name)
27          textadept.basics.print_NL('Template ' .. templ_name .. ' saved on template directory.')
28  end
```

Figura 4-49. Implementación de la función que guarda un documento abierto como plantilla

    b. Abrir un número de plantillas seleccionadas. Es análoga a la función que abre ficheros de un proyecto.
        i. Se recopila la lista de plantillas existentes (*basics.lua*).
        ii. Se seleccionan plantillas dentro del proyecto (*dialogs*), es posible elegir varias.

iii. Se abren los ficheros seleccionados (*io*).

La figura 4-50 muestra la implementación.

```
29    --- Opens templates
30    function M.openTemplates()
31            templ_paths, templ_names = textadept.basics.files_paths_names(template_dir)
32            if #(templ_paths) == 0 then
33                    ui.dialogs.message{title = 'Warning', text = 'No templates to open.', button1 = 'Accept'}
34                    return
35            end
36            options = ui.dialogs.list{title = 'List of templates',
37            columns = {'Select the templates you want to open.'}, items = templ_names, multiple = true}
38            if options == nil then
39                    textadept.basics.print_NL('Canceled operation.')
40                    return
41            end
42            for i = 1, #options do io.open_file(templ_paths[options[i]]) end
43    end
```

Figura 4-50. Implementación de la función que abre plantillas previamente guardadas

c. Eliminar un número de plantillas seleccionadas. Análogo a la función para eliminar ficheros de un proyecto.

   i. Se recopila la lista de plantillas existentes (*basics.lua*).

   ii. Se seleccionan plantillas dentro del proyecto (*dialogs*), es posible elegir varias.

   iii. Se abren los ficheros seleccionados (*os*).

   iv. Si alguna de las plantillas estuviera abierta, se cierra (*buffer*).

La figura 4-51 muestra la implementación.

```lua
--- Erases templates
function M.eraseTemplates()
	templ_paths, templ_names = textadept.basics.files_paths_names(template_dir)
	if #(templ_paths) == 0 then
		ui.dialogs.message{title = 'Warning', text = 'No templates to erase.', button1 = 'Accept'}
		return
	end
	options = ui.dialogs.list{title = 'List of templates',
	columns = {'Select the templates you want to erase.\nWARNING: This operation CANNOT BE UNDONE.'},
	items = templ_names, multiple = true}
	if options == nil then
		textadept.basics.print_NL('Canceled operation.')
		return
	end
	filenames = ''
	for i = 1, #options do
		filenames = filenames .. ' ' .. templ_names[options[i]]
		for b = 1, #_BUFFERS do
			local buffer = _BUFFERS[b]
			local name = buffer.filename or buffer._type or _L['Untitled']
			if textadept.basics.slashSub(name) == textadept.basics.slashSub(templ_paths[options[i]]) then
				buffer:close(true)
				break
			end
		end
		os.remove(templ_paths[options[i]])
	end
	textadept.basics.print_NL('The templates' .. filenames .. ' have been erased.')
end
```

Figura 4-51. Implementación de la función que elimina plantillas previamente guardadas

Las tres funciones pertenecen a la tabla M.

4.2.7 Ejecución paso a paso

Esta es la funcionalidad que ha precisado mayor trabajo de implementación, pues tiene varios subcomponentes:

1. Poder marcar líneas de secuencias de comando con puntos de ruptura (*breakpoint*), ya que esta información ha de mostrarse visualmente al usuario.
2. Guardar información de dónde se marcan los puntos de ruptura para que no se pierdan cuando se cierre el archivo o el IDE, ya que son externas a la secuencia de comandos.
3. Traducir los puntos de ruptura a pausas cuando se elija la opción de ejecutar paso a paso.

Se explican las tres.

4.2.7.1 Marcar líneas del código. Secuencias de comandos init.lua en el directorio principal y breakpoints.lua

Se ha creado la secuencia de comandos *breakpoints.lua* para introducir esta funcionalidad y la de la siguiente sección. Esta hace uso de las librerías *view* y *buffer* de *TextAdept* para la implementación.

No obstante, también hay un cambio menor de la secuencia de comandos *init.lua* del directorio principal ya comentado en funcionalidades básicas: Las ventanas de *TextAdept* incluyen hasta cinco márgenes, cuyo ancho y aspecto visual se pueden cambiar. En particular, por defecto el primer margen incluye el número de línea de código y el tercero, los límites de las zonas de plegado del código. En este caso la edición se ha limitado a editar la línea de código para aumentar el ancho del margen número 2, destinado a mostrar el marcador visual de los puntos de ruptura ya que no tenía uso en *TextAdept* estándar.

El resto se ha realizado en *breakpoints.lua*:

1. Definir una variable que identifique la marca de punto de ruptura con un valor numérico entero (*view*) y relacionarla con un icono visual determinado en los que ofrece *TextAdept* (*view*) (figura 4-52).

```
10    --- Variables for breakpoints
11    --- First, a marker number is assigned for them and they are given a form
12    MARK_BREAKPOINT = view.new_marker_number()
13    view:marker_define(MARK_BREAKPOINT, view.MARK_CIRCLE)
```

Figura 4-52, Definición de la marca de los puntos de ruptura

2. Definir una función que, dados unos datos de entrada, pone marcas en aquellas líneas donde el usuario haya marcado un punto de ruptura (*buffer* y *basics.lua*) (figura 4-53).

```
134   --- Draws breakpoints on buffer second margin
135   function draw_BP()
136           local buffer = _G.buffer
137           local filename = buffer.filename
138           if filename == nil then return end
139           local path = M.check_BP_file(filename)
140           if path == nil then return end
141           local dirname, basename = '', path
142           if path:find('[/\\]') then dirname, basename = path:match('^(.+)[/\\]([^/\\]+)$') end
143           if not textadept.basics.contains_file_name(dirname, basename) then return end
144           local breakpoints = textadept.basics.file_array(path)
145           for bp = 1, #breakpoints do
146                   buffer:marker_add(tonumber(breakpoints[bp]), MARK_BREAKPOINT)
147           end
148   end
```

Figura 4-53. Función que dibuja la marca de los puntos de ruptura en el fichero activo

3. Conectar la anterior función al hecho de que un fichero se abra (incluye la propia inicialización del sistema) (*events*) (figura 4-54).

```
149    ---Links the function to opening a file
150    events.connect(events.FILE_OPENED, draw_BP)
```

Figura 4-54. Conexión entre el evento de abrir un fichero y la función anterior

4. Definir una función que, en función de la línea en que se esté trabajando, marque un punto de ruptura o lo desmarque si ya lo había (figura 4-55).

```
152    --- Checks whether a line contains a breakpoint
153    --- If not, it marks it
154    --- Otherwise, it deletes it
155    function M.toggleBP(line)
156            local buffer = _G.buffer
157            local filename = buffer.filename
158            if filename == nil then return end
159            local path = update_BP_file(filename)
160            local new_BP = textadept.basics.add_name_list(path, line)
161            if not new_BP then textadept.basics.erase_name_list(path, line) end
162            buffer:marker_delete_all(MARK_BREAKPOINT)
163            draw_BP()
164    end
```

Figura 4-55. Función que marca o desmarca puntos de ruptura

5. Usar la conexión de eventos para conectar una pulsación de botón izquierdo del ratón en el segundo margen de un fichero abierto con la anterior función (figura 4-56).

```
165    --- Connect a double click to the former function
166    events.connect(events.MARGIN_CLICK, function(margin, position)
167                    if margin ~= 2 then return end
168                    local line = buffer:line_from_position(position)
169                    M.toggleBP(line)
170            end
171    )
```

Figura 4-56. Conexión entre el evento de clic y la función anterior

Pertenece a la tabla M sólo la segunda función, la que marca/desmarca puntos de ruptura.

4.2.7.2 Guardar posiciones de los puntos de ruptura. Secuencia de comandos breakpoints.lua

La anterior sección ha avanzado un hecho: los datos de entrada de la función que dibuja los puntos de ruptura suponen que existe un tipo de almacenamiento. Es importante darse cuenta de que los puntos de ruptura son información externa a los propios ficheros: no pertenecen al fichero propiamente, que al fin y al cabo registra sólo texto distribuido en líneas. La consecuencia de esto es que los puntos de ruptura se perderían cuando se cerrara el fichero o el sistema entero. Por ello, cuando se (des)marca un punto de ruptura, se guarda en un fichero dónde están situados sus puntos de ruptura. La estructura de datos y funciones usadas son:

1. Se ha crea un directorio dentro del principal de *TextAdept* si no existe cuando el sistema se inicializa. En este se guardan estos elementos (también se crean si no existen) (figura 4-57):

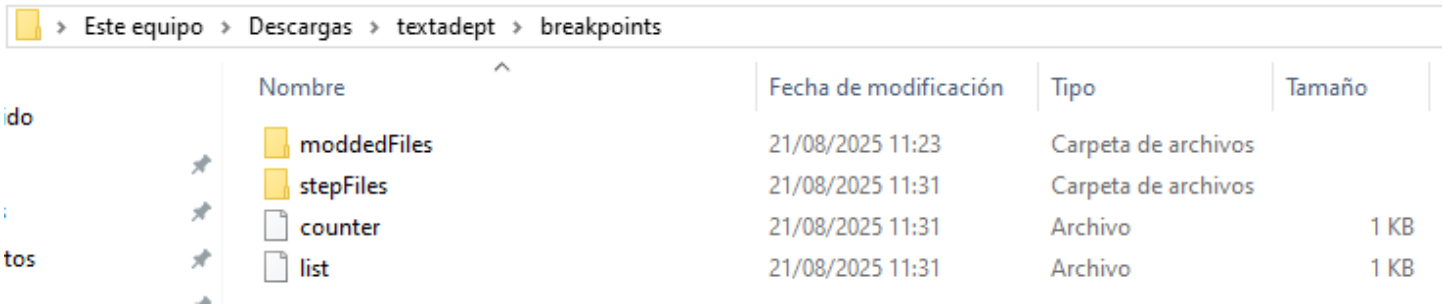


Figura 4-57. Detalle del contenido del subdirectorio específico para los puntos de ruptura

   a. Un subdirectorio que almacenará ficheros, del mismo nombre que el original, con las líneas con puntos de ruptura, ordenados en subdirectorios numerados de modo que a cada directorio le corresponde uno, La razón de que se haga así tiene que ver con la compilación separada, evitando así tener que realizar demasiadas búsquedas (figura 4-58).

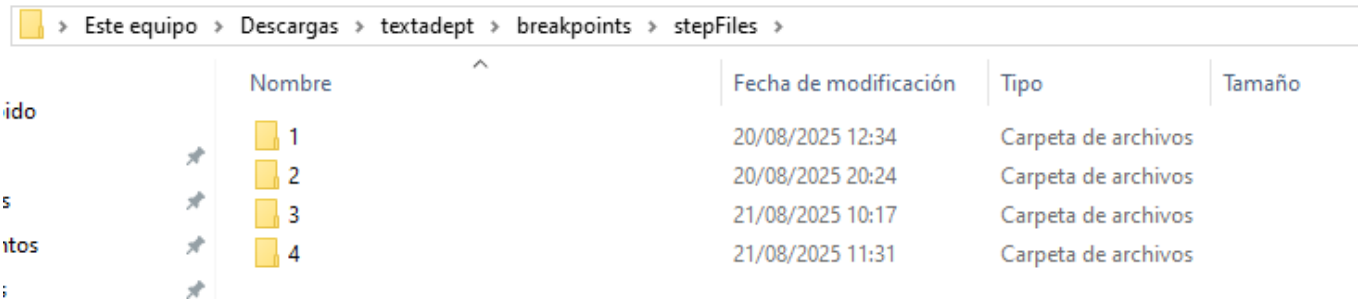


Figura 4-58. Detalle del subdirectorio que almacena los ficheros con las líneas con punto de ruptura. Nótese que están numerados

b. Otro subdirectorio almacenará los ficheros manipulados para que se detengan en los puntos pedidos. Se explica con mayor detalle en la próxima sección.

c. Un contador de los ficheros de puntos de ruptura creados (figura 4-59).

Figura 4-59. Contenido del fichero que guarda el contador de los ficheros de puntos de ruptura creados

d. Un registro que relaciona las direcciones de los ficheros con puntos de ruptura con el identificador numérico que los guarda. Si hubiera uno nuevo, se le asigna un identificador nuevo, aumentando el valor del anterior contador (figura 4-60).

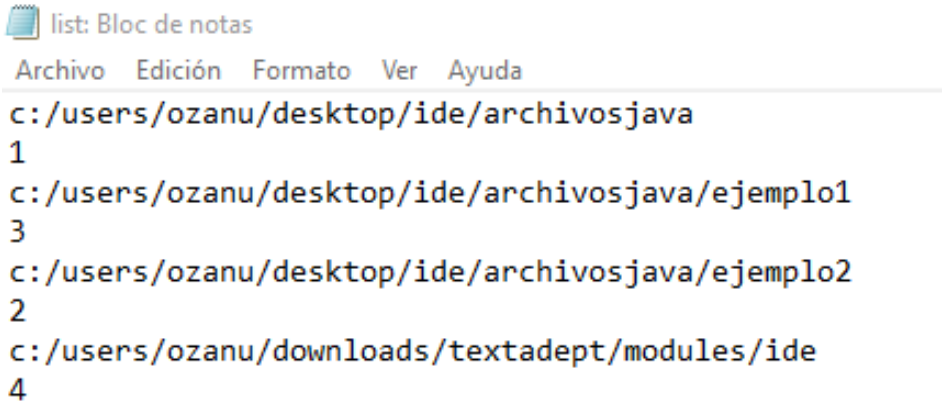


Figura 4-60. Contenido del fichero que guarda el registro de las direcciones de los ficheros con puntos de ruptura con un identificador numérico

2. Una función que busca, si existe, el fichero de los puntos de ruptura correspondiente al fichero de trabajo (figura 4-61).

```
41    --- Checks whether a file was already contained in the
42  function M.check_BP_file(filename)
43          filename = string.lower(textadept.basics.slashSub(filename))
44          local dirname, basename = '', filename
45          if filename:find('[/\\]') then dirname, basename = filename:match('^(.+)[/\\]([^/\\]+)$') end
46          local pos = 1
47          local found = false
48          local list_file = io.open(BP_list,"r")
49          local lines = list_file:lines()
50          for line in lines do
51                  local fn = string.lower(textadept.basics.slashSub(line))
52                  -- Found dirname. The next line is the breakpoint list
53                  if pos % 2 == 1 and fn == dirname then found = true
54                  -- Dirname does not exist, so the breakpoint file will be updated
55                  elseif pos % 2 == 1 and fn > dirname then break
56                  -- Returns filename breakpoints
57                  elseif found then return textadept.basics.joinPath(textadept.basics.joinPath(BP_step_dir, line), basename)
58                  end
59                  pos = pos + 1
60          end
61          list_file:close()
62          return nil
63  end
```

Figura 4-61. Función que busca ficheros de puntos de ruptura, si existe

3. Una función auxiliar es la encargada de actualizar estos ficheros cuando se añade o elimina un punto de ruptura, creándolos si es necesario primero (figura 4-62).

```lua
--- Reads the breakpoints directory, creating it if necessary
local function update_BP_file(filename)
        filename = string.lower(textadept.basics.slashSub(filename))
        local dirname, basename = '', filename
        if filename:find('[/\\]') then dirname, basename = filename:match('^(.+)[/\\]([^/\\]+)$') end
        local a = {}
        local list_file = io.open(BP_list,"r")
        local lines = list_file:lines()
        for line in lines do textadept.basics.add(a, line) end
        list_file:close()
        local number = #a + 1
        local found = false
        for pos = 1, #a do
                line = a[pos]
                local fn = string.lower(textadept.basics.slashSub(line))
                -- Found dirname. The next line is the breakpoint list
                if pos % 2 == 1 and fn == dirname then found = true
                -- Dirname does not exist, so the search stops
                elseif pos % 2 == 1 and fn > dirname then
                        number = pos
                        break
                -- Returns filename breakpoints
                elseif found then
                        -- First, check whether the filename exists
                        local new_bp_dir = textadept.basics.joinPath(BP_step_dir, line)
                        local BP_file = textadept.basics.joinPath(new_bp_dir, basename)
                        if not textadept.basics.contains_file_name(new_bp_dir, basename) then
                                textadept.basics.write_file(BP_file, {})
                        end
                        return BP_file
                end
        end
        local a2 = {}
        for pos = 1, number - 1 do
                textadept.basics.add(a2, a[pos])
        end
        textadept.basics.add(a2, dirname)
        total_BP_files = total_BP_files + 1
        textadept.basics.add(a2, total_BP_files)
        for pos = number, #a do
                textadept.basics.add(a2, a[pos])
        end
        -- As the directory was not included, it is necessary to update the two files
        textadept.basics.write_file(BP_list, a2)
        textadept.basics.write_file(BP_counter, {total_BP_files})
        --- Create a new directory
        local new_bp_dir = textadept.basics.joinPath(BP_step_dir, total_BP_files)
        lfs.mkdir(new_bp_dir)
        --- Create a new file inside the created directory
        local BP_file = textadept.basics.joinPath(new_bp_dir, basename)
        textadept.basics.write_file(BP_file, {})
        -- Now return the full path
        return BP_file
end
```

Figura 4-62. Función auxiliar que actualiza los ficheros con los puntos de ruptura. Si es necesario, crea el fichero

Pertenece a la tabla M sólo la primera función, la que comprueba si existe el fichero.

4.2.7.3 Traducir los puntos de ruptura a pausas. Secuencias de comandos breakpoints.lua y run.lua

La idea se basa en manipular el código con las funciones de pedirle al usuario información. Dichas funciones paran el código de modo natural, por lo que se ajustan a este propósito.

Las funciones usadas se han comprobado que funcionan para *Java*, pero *C* tiene problemas (detallados en la sección de autocrítica). Las secuencias de comandos *breakpoints.lua* y *run.lua* son las encargadas de ofrecer estas funcionalidades.

1. *breakpoints.lua*:
    a. Cuando el sistema es inicializado, comprueba que exista un subdirectorio, destinado a los ficheros generados para la ejecución paso a paso, dentro del directorio creado para englobar todo lo relacionado con los puntos de ruptura. Como el subdirectorio de los ficheros con los puntos de ruptura de la sección anterior, guarda los ficheros en directorios numerados, asignándole a un fichero marcado con puntos de ruptura la misma numeración, Lo crea si es necesario (figura 4-63).

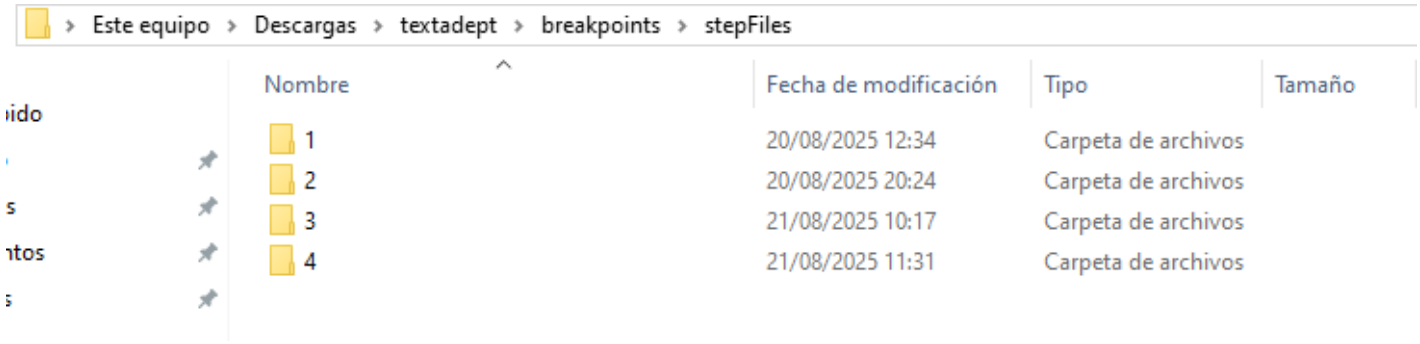


Figura 4-63. Detalle del subdirectorio que almacena los ficheros con generados para la ejecución paso a paso

b. Define una función para guardar los ficheros con puntos de ruptura funcionales. Esta hace uso de *basics.lua* y los almacena en un directorio numerado como aquel donde se guardan los ficheros de los puntos de ruptura (figura 4-64).

```
118   --- Saves files manipulated so there is a stop on each breakpoint
119   function M.saveModded(filename, a)
120           local path = update_BP_file(filename)
121           path = path:gsub(BP_step_dir, BP_modded_dir)
122           local dirname, basename = '', filename
123           if filename:find('[/\\]') then dirname, basename = filename:match('^(.+)[/\\]([^/\\]+)$') end
124           local dirname2, basename2 = '', path
125           if path:find('[/\\]') then dirname2, basename2 = path:match('^(.+)[/\\]([^/\\]+)$') end
126           --- If necessary, the directory is created
127           lfs.mkdir(dirname)
128           textadept.basics.write_file(textadept.basics.joinPath(dirname2, basename), a)
129           return path
130   end
```

Figura 4-64. Función que guarda los ficheros con puntos de para funcionales

c. Define otra función para recuperar los anteriores ficheros (figura 4-65).

```
131   --- Returns the modified executable
132   function M.returnModded(filename, a)
133           local path = update_BP_file(filename)
134           return path:gsub(BP_step_dir, BP_modded_dir)
135   end
```

Figura 4-65. Función que recupera la ruta de los ficheros con puntos de para funcionales

2. *run.lua*:

   a. Define comandos específicos para los lenguajes (sólo *Java* en este caso) de un modo análogo a como se ha visto en compilación separada. La orden que se ha escrito es la siguiente, haciendo uso de la librería *Scanner*:

M.step_commands = {java={'import java.util.Scanner;', 'System.out.print("This is a breakpoint. Press enter to continue.");\nString text = "";\nwhile (text.compareTo("") == 0) {\ntry {\nScanner s = new Scanner(System.in);\ntext = s.nextLine();\n} catch (Exception e){\nSystem.out.println(" Error.");\n}\n}'}}

b. Define una función auxiliar encargada de compilar el programa manipulado, haciendo uso tanto de *breakpoints.lua* como de *basics.lua* (figura 4-66).

```
273   --- This auxiliar function compiles modified files
274 function compile_bp(filename)
275         local dirname, basename = '', filename
276         if filename:find('[/\\]') then dirname, basename = filename:match('^(.+)[/\\]([^/\\]+)$') end
277         local ext = filename:match('[^/\\.]+$')
278         local commands = M.step_commands[ext]
279         if commands == nil then return end
280         local filepaths, filenames = textadept.basics.files_paths_names(dirname)
281         --local modded_filename = textadept.breakpoints.check_BP_file(filenames[0])
282         for f = 1, #filepaths do
283                 local fp = filepaths[f]
284                 local code_array = textadept.basics.file_array(fp)
285                 local bp_array = textadept.basics.file_array(textadept.breakpoints.check_BP_file(fp))
286                 if bp_array == nil then bp_array = {} end
287                 local final_array = {}
288                 textadept.basics.add(final_array, commands[1])
289                 for line = 1, #code_array do
290                         if #bp_array > 0 then
291                                 if line == tonumber(bp_array[1]) then
292                                         local  bp_array_prov = {}
293                                         for i = 2, #bp_array_prov do textadept.basics.add(bp_array_prov, bp_array[i]) end
294                                         bp_array = bp_array_prov
295                                         textadept.basics.add(final_array, commands[2])
296                                 end
297                         end
298                         textadept.basics.add(final_array, code_array[line])
299                 end
300                 ---Saves a new file in the created directory
301                 textadept.breakpoints.saveModded(fp, final_array)
302         end
303         M.compile(textadept.breakpoints.returnModded(filename))
304 end
```

Figura 4-66. Función auxiliar que compila el programa manipulado con puntos de ruptura funcionales

c. Define una función para realizar este compilado y luego hacer una ejecución normal de *TextAdept* (figura 4-67).

```
305   --- This executes the program with breakpoints
306   function M.run_step2step(filename)
307           if filename == nil then filename = buffer.filename end
308           ui.print('Starting step by step compilation of ' .. filename .. '.')
309           compile_bp(filename)
310           modded_path = textadept.breakpoints.returnModded(filename)
311           local dirname, basename = '', filename
312           if filename:find('[/\\]') then dirname, basename = filename:match('^(.+)[/\\]([^/\\]+)$') end
313           local dirname2, basename2 = '', modded_path
314           if modded_path:find('[/\\]') then dirname2, basename2 = modded_path:match('^(.+)[/\\]([^/\\]+)$') end
315           ui.print('Starting run step by step of ' .. filename .. '.')
316           M.run(textadept.basics.joinPath(dirname2, basename))
317   end
```

Figura 4-67. Función que ejecuta la auxiliar y luego ejecuta normalmente

Pertenecen a las tablas M las dos funciones de *breakpoints.lua* y la función de ejecución paso a paso de *run.lua*.

4.2.8 Interfaz de usuario

Hasta ahora se ha explicado cómo se implementaron las diversas funcionalidades. Queda una sola explicación: cómo se ha implementado la interfaz de usuario. La respuesta es elemental: como *TextAdept* ya traía su propia interfaz, se ha editado para que el usuario pueda hacer uso de las funcionalidades implementadas.

Esto es, básicamente se han añadido nuevas opciones de menú y nuevos atajos de teclado.

4.2.8.1 Menú visual. Secuencia de comandos menu.lua

Esta secuencia de comandos introduce el sistema de menús de *TextAdept*. Su utilización es sencilla, se limita a definir una etiqueta con un nombre y acompañarla de la función (previamente definida o creada para la ocasión) que se quiere que ejecute. Como ya existen varias opciones de menú en *TextAdept*, es importante encontrar un lugar adecuado.

En concreto, las figuras 4-68 a 4-70 muestran dónde se ha implementado cada funcionalidad:

```
title = _L['File'], --
{_L['Create Workspace'], textadept.workspaces.create_WS}, --
{_L['Erase Workspace'], textadept.workspaces.erase_WS}, --
{_L['Open Workspace'], textadept.workspaces.open_WS}, --
SEPARATOR, --
{_L['Start Project'], textadept.projects.startProject}, --
{_L['Erase Project'], textadept.projects.eraseProject}, --
{_L['Import Project'], textadept.projects.importProject}, --
SEPARATOR, --
{_L['Open Files from Project'], textadept.projects.openProjFiles}, --
{_L['Save File in Project'], textadept.projects.saveFileProj}, --
{_L['Erase Files from Project'], textadept.projects.eraseProjFiles}, --
SEPARATOR, --
{_L['New'], buffer.new}, --
{_L['Open'], io.open_file}, --
{_L['Open Recent...'], io.open_recent_file}, --
{_L['Reload'], buffer.reload}, --
{_L['Save'], buffer.save}, --
{_L['Save As'], buffer.save_as}, --
{_L['Save All'], io.save_all_files}, --
SEPARATOR, --
{_L['Close'], buffer.close}, --
{_L['Close All'], io.close_all_buffers}, --
SEPARATOR, --
{_L['Save As Template'], textadept.templates.saveTemplate}, --
{_L['Open Templates'], textadept.templates.openTemplates}, --
{_L['Erase Templates'], textadept.templates.eraseTemplates}, --
SEPARATOR, --
{_L['Load Session...'], textadept.session.load}, --
{_L['Save Session...'], textadept.session.save}, --
SEPARATOR, --
{_L['Quit'], quit}
```

Figura 4-68. Implementación en menu.lua de las funcionalidades de gestión de espacios de trabajo, proyectos y plantillas

```
title = _L['Tools'], --
{_L['Command Entry'], ui.command_entry.run},
{_L['Select Command'], function() M.select_command() end}, --
SEPARATOR, --
{_L['Run'], textadept.run.run}, --
{_L['Run Step by Step'], textadept.run.run_step2step}, --
{_L['Compile'], textadept.run.compile}, --
{_L['Compile separately'], textadept.run.sep_compile}, --
{_L['Build'], textadept.run.build}, --
{_L['Run tests'], textadept.run.test}, --
{_L['Run project'], textadept.run.run_project}, --
{_L['Stop'], textadept.run.stop},
{_L['Next Error'], function() textadept.run.goto_error(true) end},
{_L['Previous Error'], function() textadept.run.goto_error(false) end}, --
SEPARATOR, --
{_L['Toggle Breakpoint'], function()
        textadept.breakpoints.toggleBP(buffer:line_from_position(buffer.current_pos))
        end}, --
SEPARATOR, --
```

Figura 4-69. Implementación en menu.lua de las funcionalidades de compilación separada y ejecución paso a paso

```
294             title = _L['View'], --
295             {_L['Next View'], function() ui.goto_view(1) end},
296             {_L['Previous View'], function() ui.goto_view(-1) end}, --
297             SEPARATOR, --
298             {_L['Toggle Project Explorer'], textadept.infowindows.togglePE}, --
299             {_L['Toggle Output View'], textadept.infowindows.toggleOutput}, --
300             SEPARATOR, --
```

Figura 4-70. Implementación en menu.lua de las funcionalidades de mostrar o no el explorador de proyectos y la consola de resultados

Esta secuencia de comandos no devuelve su tabla M.

4.2.8.2 Atajos de teclado. Secuencia de comandos keys.lua

Esta secuencia de comandos introduce el sistema de menús de *TextAdept*. Su utilización se sincroniza con *menu.lua*, donde se visualizará qué atajos corresponden a cada opción. En el manual de usuario se detallan los atajos. Las figuras 4-71 a 4-73 muestran dónde se ha implementado cada funcionalidad.

```
327         -- File.
328         [textadept.workspaces.create_WS] = {'ctrl+alt+c', 'ctrl+cmd+c', 'ctrl+alt+c'}, --
329         [textadept.workspaces.erase_WS] = {'ctrl+alt+s', 'ctrl+cmd+s', 'ctrl+alt+s'}, --
330         [textadept.workspaces.open_WS] = {'ctrl+alt+o', 'ctrl+cmd+o', 'ctrl+alt+o'}, --
331         [textadept.projects.startProject] = {'ctrl+N', 'cmd+N', 'ctrl+meta+n'}, --
332         [textadept.projects.eraseProject] = {'ctrl+I', 'cmd+I', 'ctrl+meta+i'}, --
333         [textadept.projects.importProject] = {'ctrl+E', 'cmd+E', 'ctrl+meta+e'}, --
334         [textadept.projects.openProjFiles] = {'ctrl+alt+i', 'ctrl+cmd+i', 'ctrl+alt+i'}, --
335         [textadept.projects.saveFileProj] = {'ctrl+alt+n', 'ctrl+cmd+n', 'ctrl+alt+n'}, --
336         [textadept.projects.eraseProjFiles] = {'ctrl+alt+q', 'ctrl+cmd+q', 'ctrl+alt+q'},
337         [buffer.new] = {'ctrl+n', 'cmd+n', 'ctrl+n'}, --
338         [io.open_file] = {'ctrl+o', 'cmd+o', 'ctrl+o'},
339         -- TODO: io.open_recent_file
340         -- TODO: buffer.reload
341         [buffer.save] = {'ctrl+s', 'cmd+s', {'ctrl+s', 'meta+s', 'meta+S'}}, --
342         [buffer.save_as] = {'ctrl+S', 'cmd+S', 'ctrl+meta+s'},
343         -- TODO: io.save_all_files
344         [buffer.close] = {'ctrl+w', 'cmd+w', 'ctrl+w'}, --
345         [io.close_all_buffers] = {'ctrl+W', 'cmd+W', 'ctrl+meta+w'},
346         [textadept.templates.saveTemplate] = {'ctrl+alt+S', 'cmd+cmd+S', 'ctrl+alt+S'},
347         [textadept.templates.openTemplates] = {'ctrl+alt+O', 'cmd+cmd+O', 'ctrl+alt+O'},
348         [textadept.templates.eraseTemplates] = {'ctrl+alt+D', 'cmd+cmd+D', 'ctrl+alt+D'},
349         -- TODO: textadept.sessions.load
350         -- TODO: textadept.sessions.save
351         [quit] = {'ctrl+q', 'cmd+q', {'ctrl+q', 'meta+q'}},
```

Figura 4-71. Implementación en keys.lua de las funcionalidades de gestión de espacios de trabajo, proyectos y plantillas

```
414         -- Tools.
415         [ui.command_entry.run] = {'ctrl+e', 'cmd+e', 'ctrl+e'},
416         [m('Tools/Select Command')] = {'ctrl+p', 'cmd+p', 'ctrl+p'},
417         [textadept.run.run] = {'ctrl+r', 'cmd+r', 'ctrl+r'},
418         [textadept.run.run_step2step] = {'ctrl+alt+B', 'cmd+cmd+B', 'ctrl+alt+B'},
419         [textadept.run.compile] = {'ctrl+C', 'cmd+C', 'ctrl+meta+c'},
420         [textadept.run.sep_compile] = {'ctrl+alt+C', 'cmd+cmd+C', 'ctrl+alt+C'},
421         [textadept.run.build] = {'ctrl+B', 'cmd+B', 'ctrl+meta+b'},
422         [textadept.run.test] = {'ctrl+T', 'cmd+T', 'ctrl+meta+t'},
423         [textadept.run.run_project] = {'ctrl+R', 'cmd+R', 'ctrl+meta+r'},
424         [textadept.run.stop] = {'ctrl+X', 'cmd+X', 'ctrl+meta+x'},
425         [m('Tools/Next Error')] = {'ctrl+alt+e', 'ctrl+cmd+e', 'meta+e'},
426         [m('Tools/Previous Error')] = {'ctrl+alt+E', 'ctrl+cmd+E', 'meta+E'},
427         [m('Tools/Toggle Breakpoint')] = {'ctrl+alt+b', 'ctrl+cmd+b', 'ctrl+alt+b'}, --
```

Figura 4-72. Implementación en keys.lua de las funcionalidades de compilación separada y ejecución paso a paso

```
477         -- View.
478         [m('View/Next View')] = {
479                 'ctrl+alt+pgdn', 'ctrl+cmd+pgdn', WIN32 and 'meta+pgdn' or 'ctrl+meta+pgdn'
480         }, [m('View/Previous View')] = {
481                 'ctrl+alt+pgup', 'ctrl+cmd+pgup', WIN32 and 'meta+pgup' or 'ctrl+meta+pgup'
482         },
483         [textadept.infowindows.togglePE] = {'ctrl+t', 'cmd+t', 'ctrl+t'},
484         [textadept.infowindows.toggleOutput] = {'ctrl+alt+T', 'cmd+cmd+T', 'ctrl+alt+T'},
```

Figura 4-73. Implementación en keys.lua de las funcionalidades de mostrar o no el explorador de proyectos y la consola de resultados

4.3 Resumen de implementación

Se van a resumir en una tabla cuáles de las características del sistema han sido implementadas como parte del trabajo pertinente a la realización del proyecto de un modo completo (es decir, se ha creado una secuencia de comandos específicamente para ese fin), cuáles son producto de haber manipulado la estructura de datos y secuencias de comandos ya presente en *TextAdept* y cuáles estaban ya presentes y no son, pues, trabajo original del proyecto.

Tabla 4-1. Resumen de las características del sistema según si se han introducido en este proyecto o no

Característica	**Es trabajo original**	**Es una modificación**	**Está presente en *TA***
Edición de texto habitual (cortar, copiar, pegar…)			X
Gestión simple de ficheros del sistema operativo			X
Manejo simultáneo de ventanas múltiples			X
Funcionalidades básicas (gestión de arrays, guardar ficheros…)	X		
Gestión de espacios de trabajo	X		

Gestión de proyectos y ficheros de proyectos	X		
Control del explorador de proyectos	X		
Control de la consola de resultados	X		
Compilación estándar			X
Compilación separada		X	
Gestión de plantillas	X		
Gestión de puntos de ruptura	X		
Ejecución estándar			X
Ejecución paso a paso		X	
Depurador			X
Interfaz visual (menú)		X	
Interfaz del teclado		X	
Multiplataforma			X
Configuración y herramientas externas			X

Capítulo 5

PRUEBAS

5.1 Tipología de las pruebas realizadas

Como el sistema ha de ser un IDE, las pruebas necesarias para asegurar el funcionamiento correcto de las características implementadas han de dividirse por cada una de estas y, dentro de algunas categorías, han de subdividirse por la existencia de varias operaciones. Es decir, este apartado sigue la misma estructura que el anterior, Implementación.

1. El conjunto de pruebas de los espacios de trabajo se divide en estas subcategorías:
    a. Comprobar la existencia de las estructuras de datos descritas en Implementación, lo que incluye cerciorarse de que se crean cuando se inicia el sistema.
    b. Comprobar que existe al menos un espacio de trabajo, no disponible para ser eliminado, para que así exista en todo momento al menos uno.
    c. Comprobar que la operación de eliminar no está disponible cuando sólo existe el espacio de trabajo por defecto.
    d. Comprobar que otros espacios de trabajo se pueden crear, abrir y eliminar, impidiendo que dos espacios de trabajo compartan el mismo nombre,
2. El conjunto de pruebas de la Gestión de proyectos y ficheros es la más compleja, ya que afecta también al Explorador de proyectos.
    a. Comprobar que al inicio del sistema el explorador de proyectos está vacío y que no es posible eliminar proyectos ni ejecutar cualquiera de las operaciones relacionadas con ficheros.
    b. Comprobar que la creación de proyectos funciona y que no pueden existir, dentro del mismo espacio de trabajo, dos proyectos con el mismo nombre, aunque sus rutas sean distintas, así como que actualiza el Explorador de proyectos.
    c. Comprobar que la importación de proyectos funciona correctamente, evitando importar directorios ya registrados en un espacio de trabajo o con un nombre ya existen, y que actualiza el Explorador de proyectos correctamente.
    d. Eliminar uno de los proyectos eliminados y volver a crearlo, actualizando en ambos casos el Explorador de proyectos.

    e. Comprobar que las operaciones de gestión de ficheros de un proyecto funcionan correctamente.
    f. Comprobar que se puede trabajar en más de un espacio de trabajo con los proyectos.
    g. Comprobar que el explorador de proyectos en sí mismo funciona correctamente.
3. La consola de resultados tiene un conjunto de pruebas bastante menor, pues en las pruebas anteriores ya se ha podido comprobar si se abre para comunicar el resultado de alguna operación. Simplemente se prueba si se abre y cierra correctamente.
4. La compilación en ficheros separados tampoco exige muchas pruebas, pues la compilación normal funcionaba en *TextAdept*, la base del sistema. Básicamente:
    a. Comprobar que funciona para C, detectando errores.
    b. Comprobar que es indiferente para Java.
    c. Comprobar que no funciona para otras extensiones.
5. Las pruebas de gestión de plantillas son análogas a las de la gestión de ficheros en un proyecto.
    a. Se comprueba que el sistema crea un directorio exclusivo para las plantillas en su directorio principal si no existe.
    b. Se comprueba que el sistema no puede abrir ni eliminar plantillas antes de que existan.
    c. Se crean plantillas y se abren posteriormente.
    d. Se eliminan algunas de las plantillas y se comprueba
6. La ejecución paso a paso precisa de tres conjuntos de pruebas.
    a. Comprobar que se marcan (y desmarcan) visualmente puntos de ruptura en el código.
    b. Comprobar que dichas posiciones se guardan y se recuperan cuando el sistema se inicializa o se abre el fichero.
    c. Comprobar que se traducen correctamente a paradas en el código.
7. Por último, se va a probar la compilación y ejecución con programas más elaborados que los usados en anteriores ejemplos:
    a. Pruebas en *C* y *Java* con recursividad.
    b. Una prueba en Java con *arrays*.

Se listan a continuación las pruebas.

5.2 Espacios de trabajo

WS 1 Creación de la estructura de datos para los espacios de trabajo (1)

No existe el directorio para los espacios de trabajo, se crea cuando arranca el sistema.

a. Inicialmente el directorio principal del sistema tiene la misma estructura de datos que tiene *TextAdept* (figura 5-1).

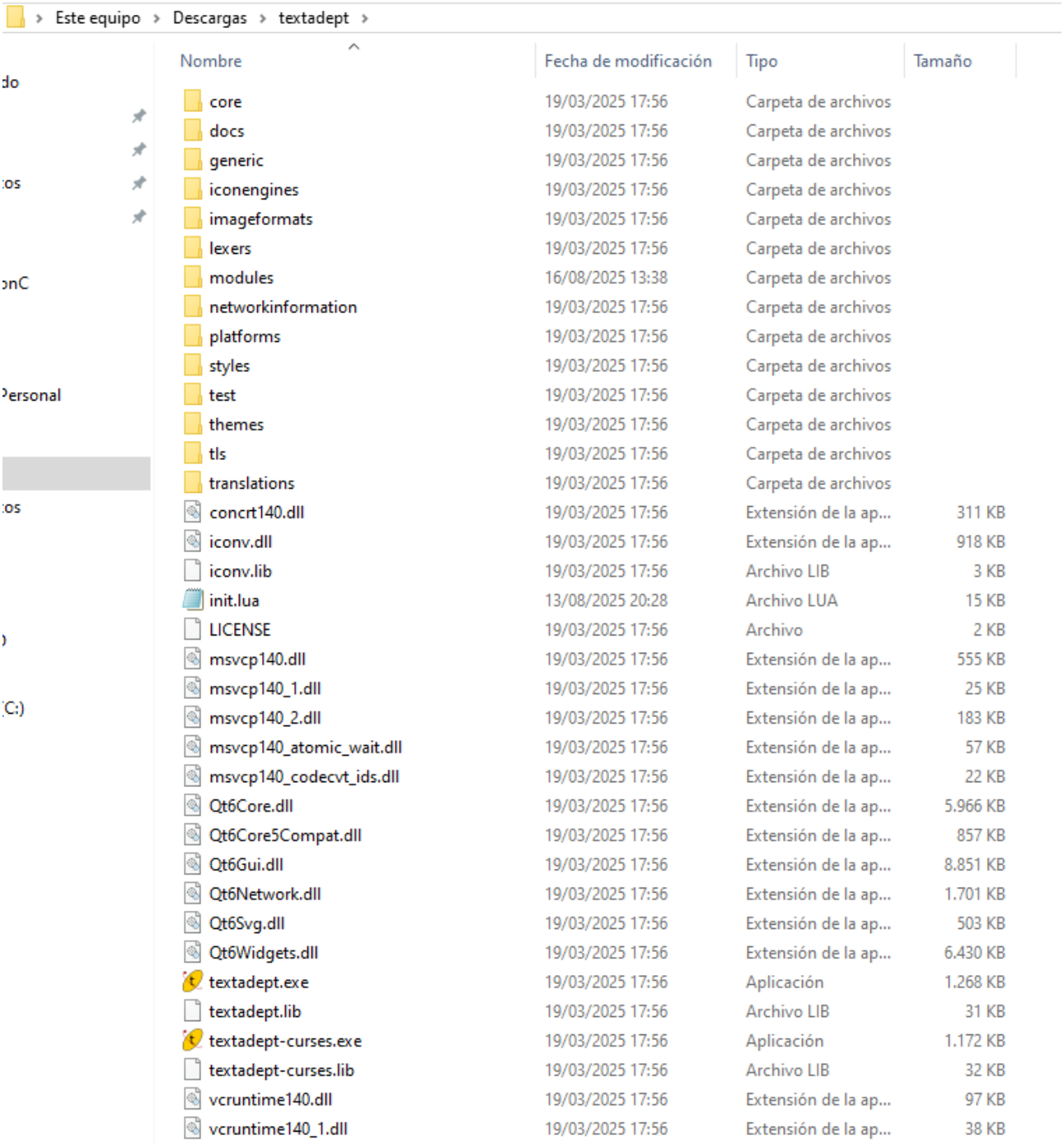


Figura 5-1. Arquitectura de ficheros inicial del sistema

b. Iniciar el sistema conlleva la creación en el mismo de un subdirectorio llamado *workspaces* (figura 5-2).

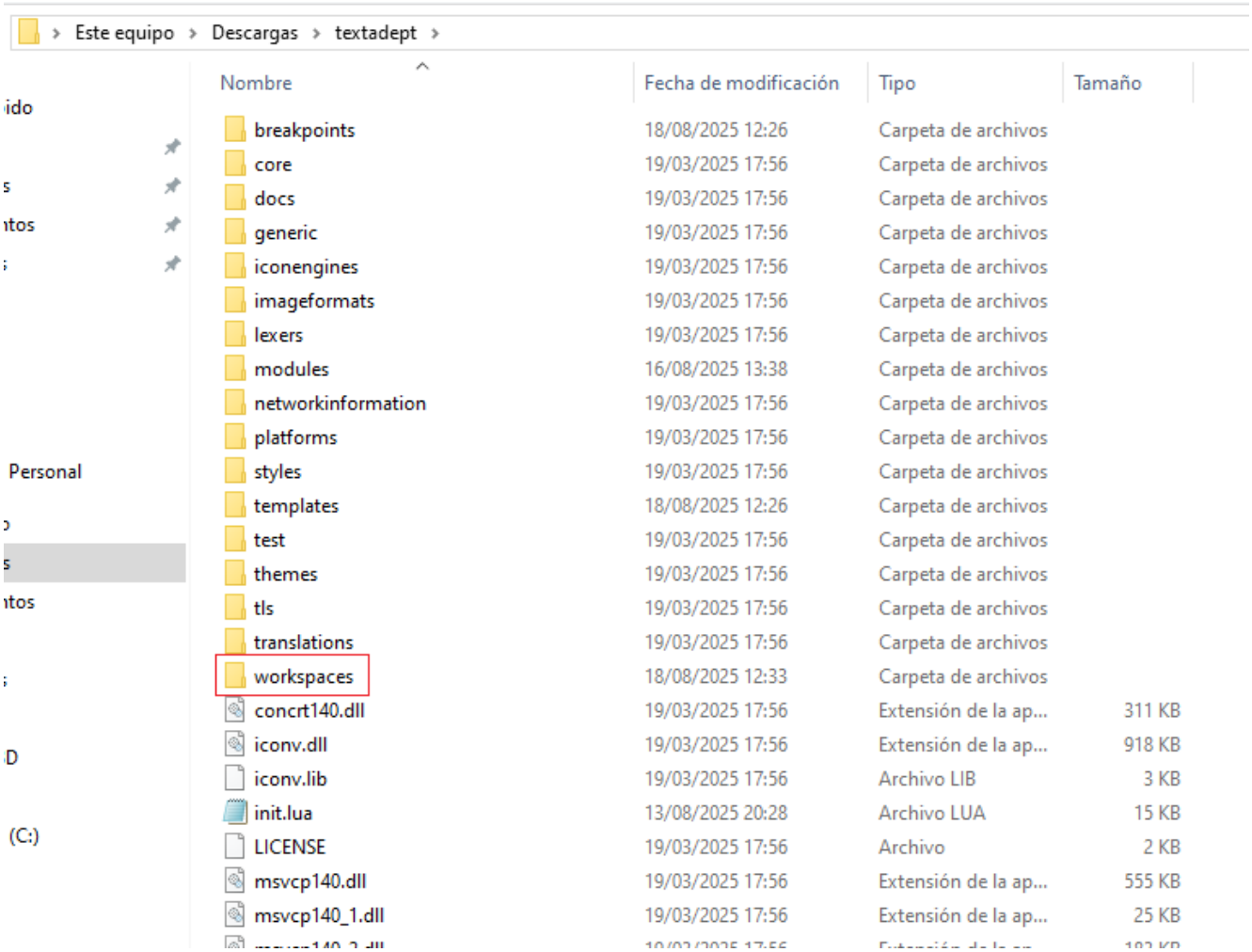


Figura 5-2. Aparición del directorio workspaces después de inicializar el sistema

WS 2 Creación de la estructura de datos para los espacios de trabajo (2)

No existen el fichero para guardar el nombre del espacio de trabajo, el de la lista de espacios de trabajo ni el directorio correspondiente al espacio de trabajo por defecto, por lo que todos se deben crear cuando se arranca el sistema.

a. Inicialmente el subdirectorio *workspaces* está vacío (figura 5-3).

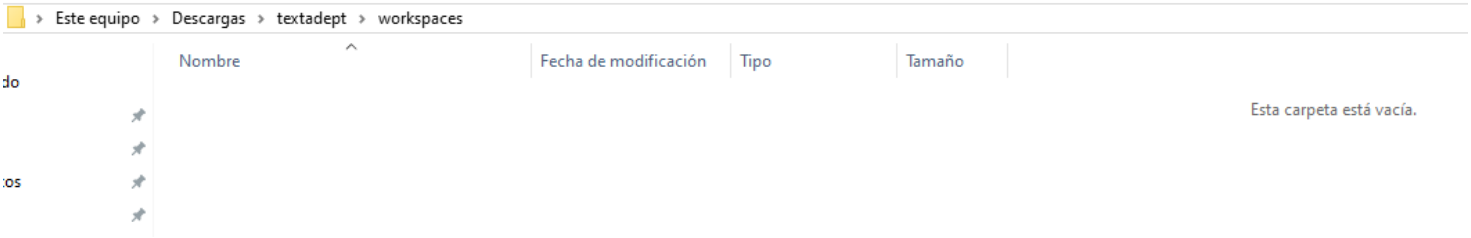


Figura 5-3. Directorio workspaces inicialmente vacío

b. Iniciar el sistema conlleva que aparezcan dos ficheros, *ws_list* y *ws_name*, y un directorio de nombre *default* (figura 5-4).

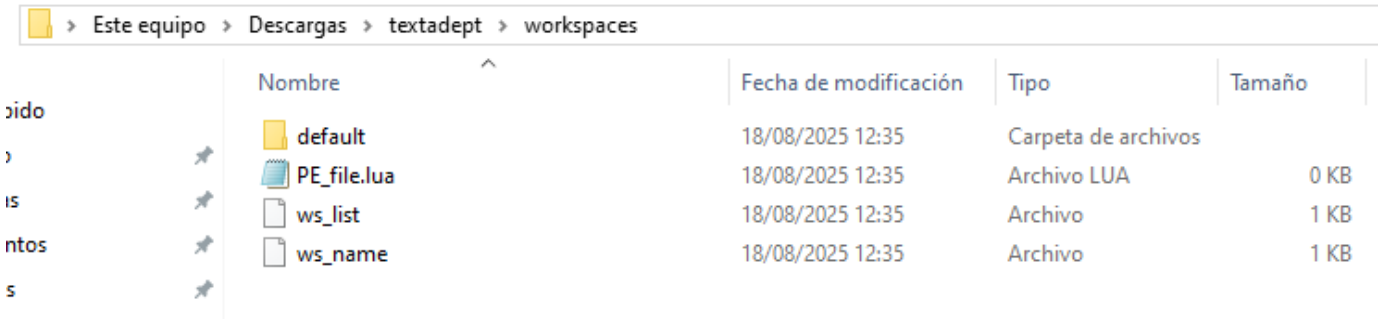


Figura 5-4. Directorio workspaces con la estructura de datos necesaria para la gestión de los espacios de trabajo

WS 3 Existencia de un espacio de trabajo por defecto (1)

Aunque no se ha creado ningún espacio de trabajo, se asigna uno por defecto (figura 5-5).

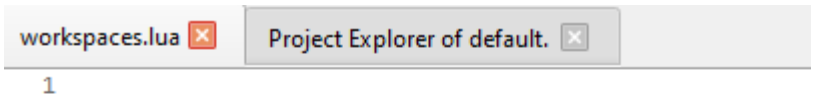


Figura 5-5. El explorador de proyectos se llama default, que es el nombre al espacio de trabajo existente por defecto

WS 4 Existencia de un espacio de trabajo por defecto (2)

El espacio de trabajo por defecto aparece disponible para abrirlo.

a. Se selecciona en el menú la opción *Open Workspace*, dotada de un atajo de teclado (figura 5-6).

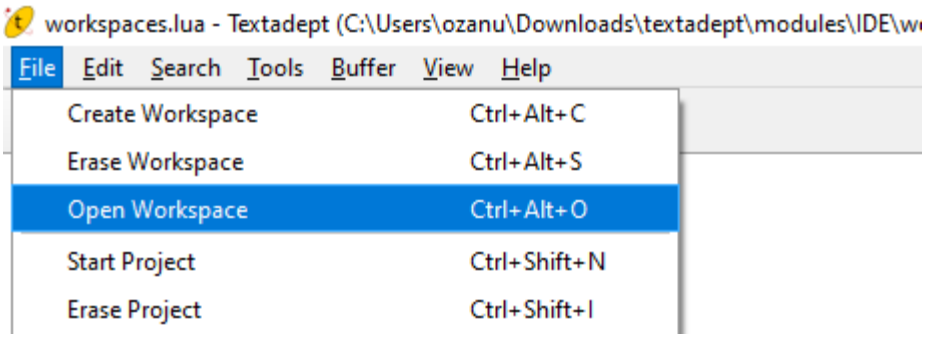


Figura 5-6. Opción Open Workspace

b. Aparece un menú en que se puede seleccionar el espacio de trabajo *default* (figura 5-7).

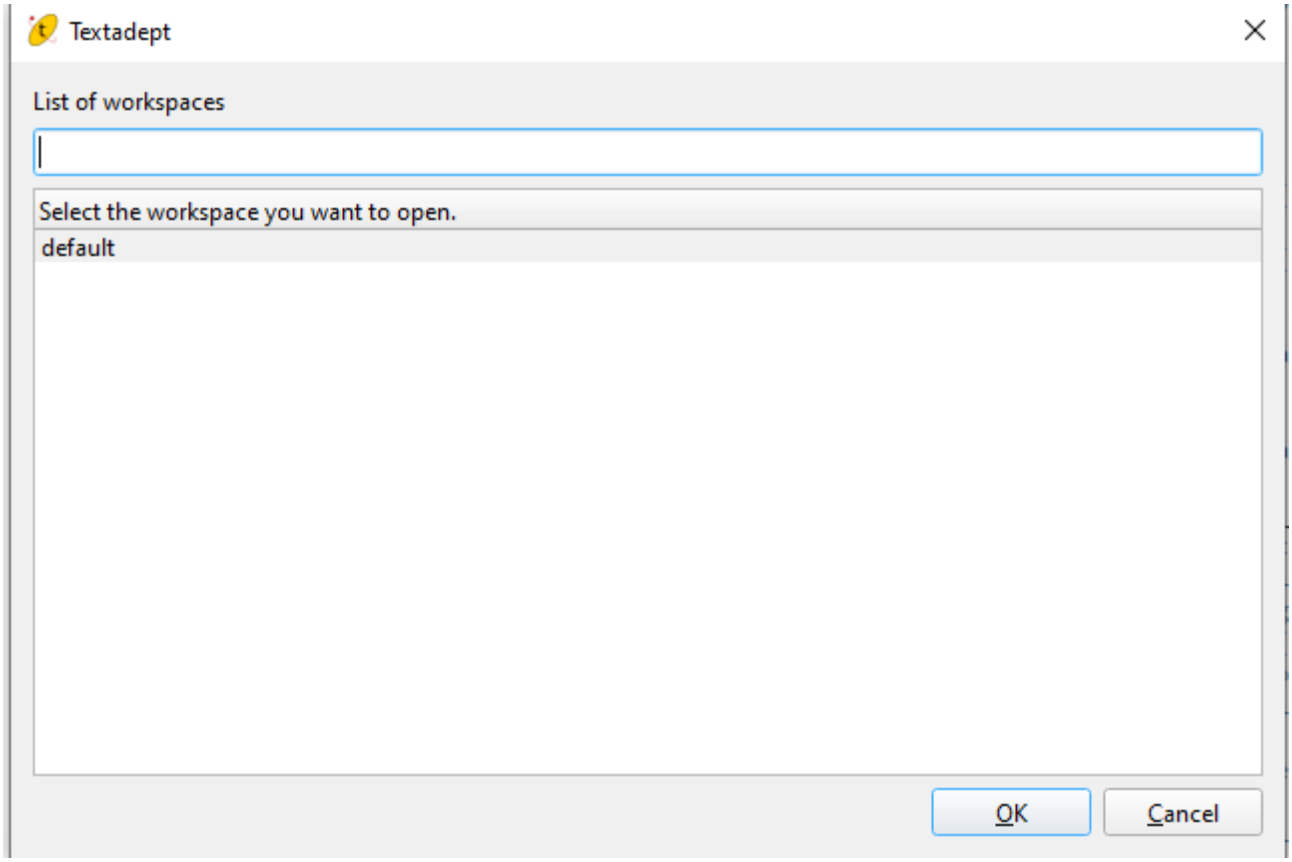


Figura 5-7. Menú para seleccionar el espacio de trabajo definido por defecto default

WS 5 Comprobación de que no se puede eliminar el espacio de trabajo por defecto

Seleccionar la eliminación de un espacio de trabajo; cuando sólo existe el espacio por defecto arroja un aviso.

a. Se selecciona en el menú la opción *Erase Workspace*, dotada de un atajo de teclado (figura 5-8).

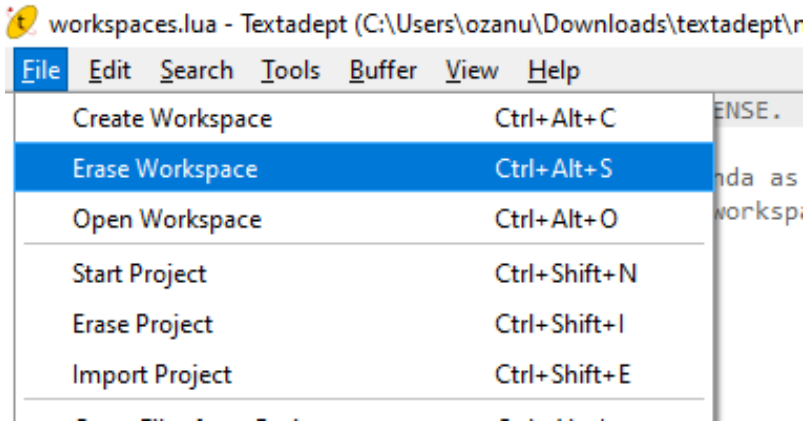


Figura 5-8. Opción Erase Workspace

b. Aparece un mensaje de aviso (figura 5-9).

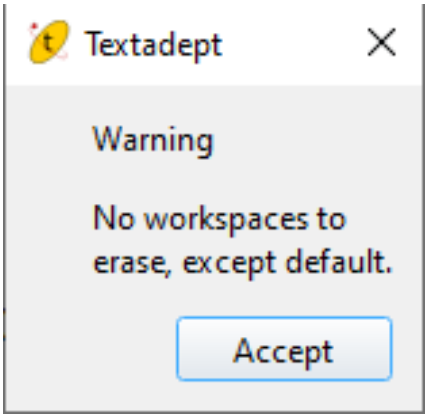


Figura 5-9. Mensaje de aviso de que no hay otros espacios de trabajo excepto el definido por defecto

WS 6 Creación de un espacio de trabajo

Crear un espacio de trabajo.

a. Se selecciona en el menú la opción Create Workspace, dotada de un atajo de teclado (figura 5-10).

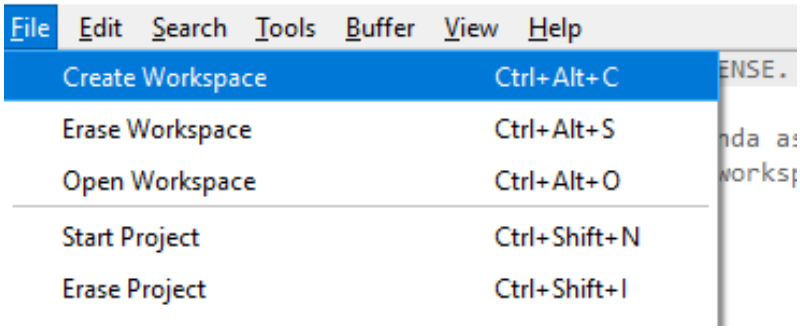


Figura 5-10. Opción CreateWorkspace

b. Se abre un menú en que el usuario puede escribir el nombre de un nuevo espacio de trabajo, new_WS en el ejemplo (figura 5-11).

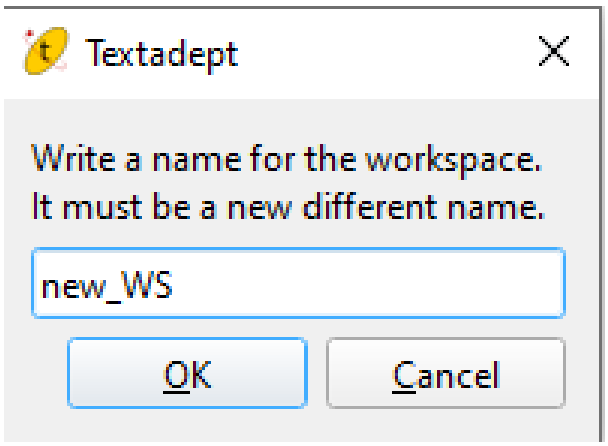


Figura 5-11. Menú que permite al usuario escribir el nombre del espacio de trabajo

c. Aceptar el nombre escrito conlleva la aparición de un mensaje (figura 5-12).

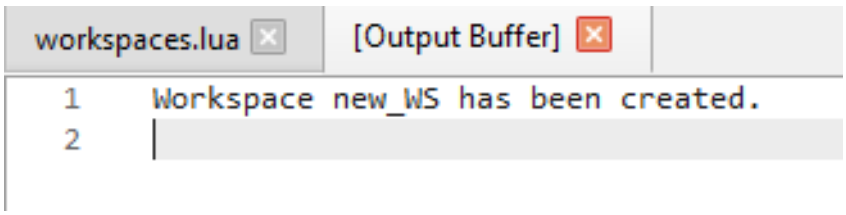


Figura 5-12. Mensaje de confirmación del nuevo espacio de trabajo

d. Se crea un nuevo directorio en el subdirectorio workspaces (figura 5-13).

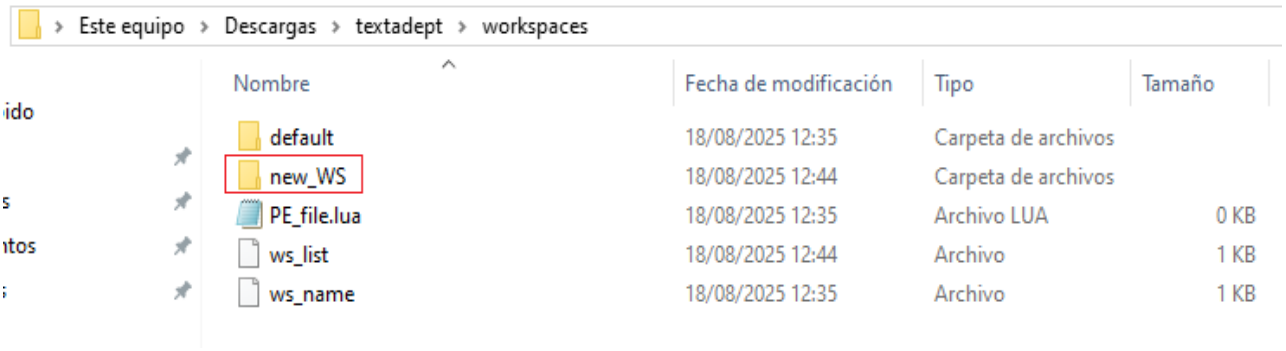


Figura 5-13. Nuevo directorio new_WS

WS 7 Comprobación de que no se puede eliminar el espacio de trabajo por defecto (2)

El espacio de trabajo por defecto no aparece disponible para ser eliminado (figura 5-14).

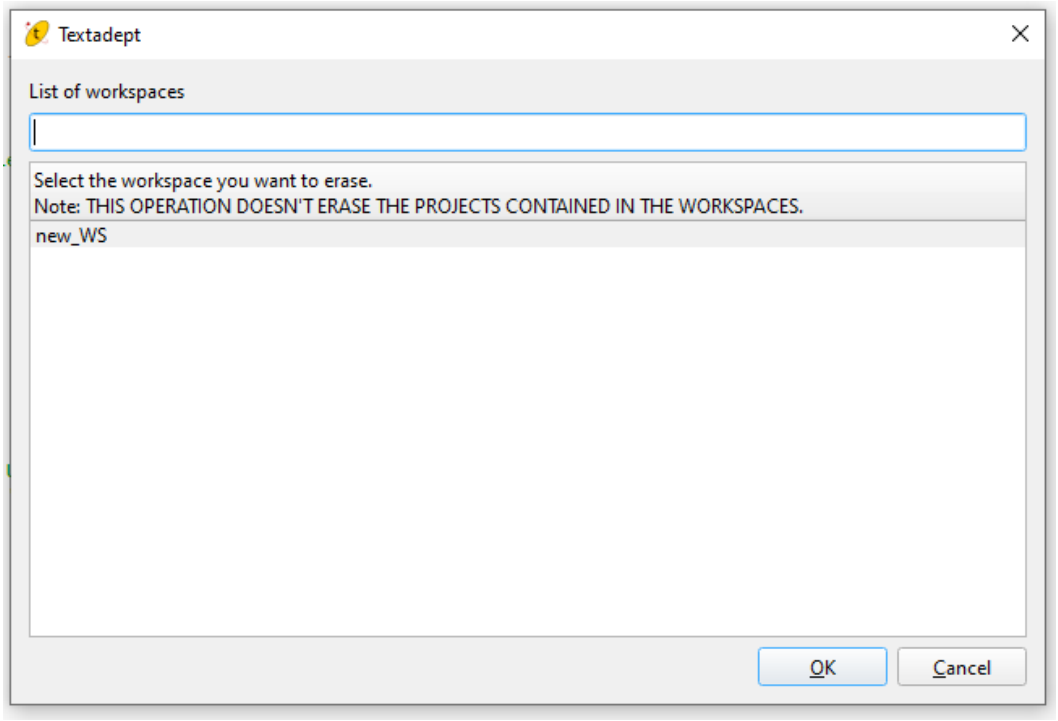


Figura 5-14. Sólo el espacio de trabajo new_WS aparece disponible para ser eliminado

WS 8 Selección del nuevo espacio de trabajo

Cambiar el espacio de trabajo al recién creado.

a. Se selecciona de nuevo *Open Workspace* y se elige *new_WS* (figura 5-15).

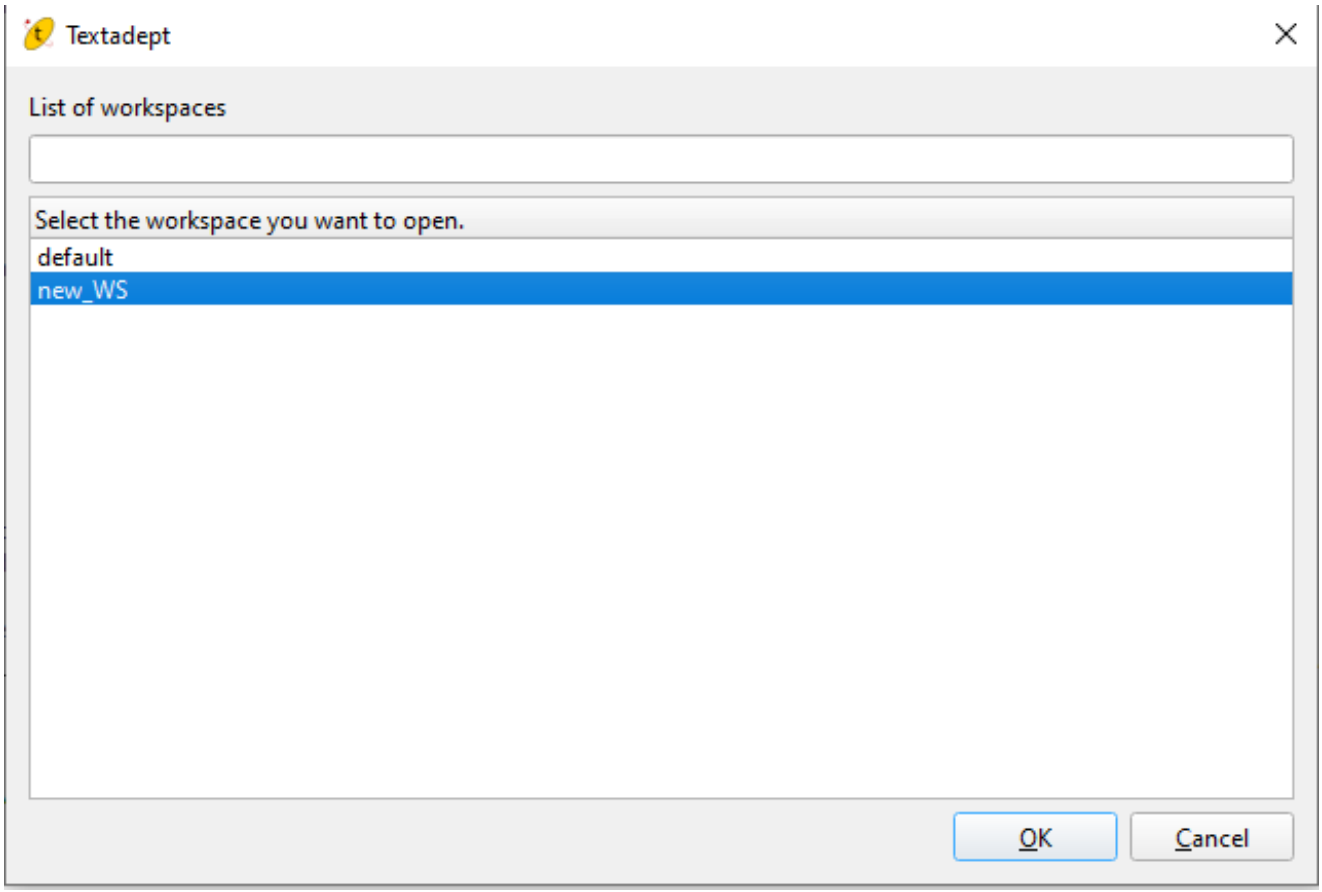


Figura 5-15. En el menú de elección de espacio de trabajo, se elige new_WS

b. Aparece un mensaje de confirmación (figura 5-16).

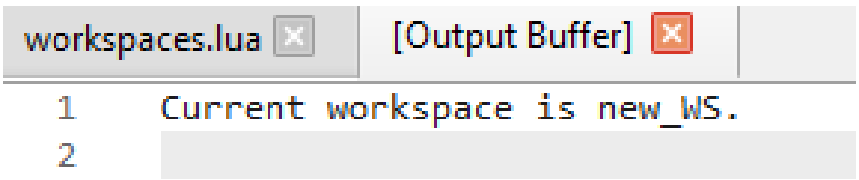


Figura 5-16. Mensaje confirmando el cambio de espacio de trabajo

c. Se actualiza el explorador de proyectos (figura 5-17).

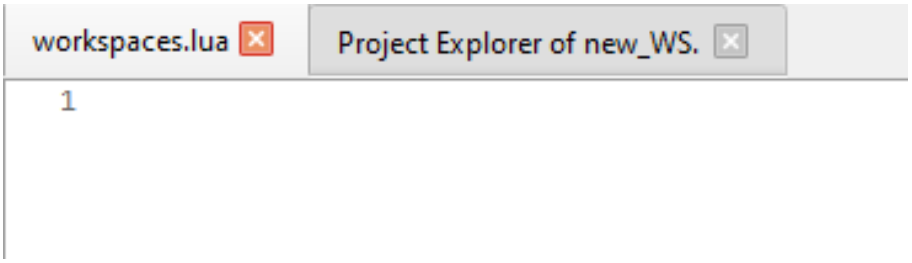


Figura 5-17. Actualización del explorador de proyectos, ahora del espacio de trabajo new_WS

WS 9 Unicidad de los nombres de los espacios de trabajo

Intentar crear un nuevo espacio de trabajo con un nombre ya existente arroja un aviso.

a. Se selecciona de nuevo Create Workspace para volver a escribir el nombre new_WS (figura 5-18).

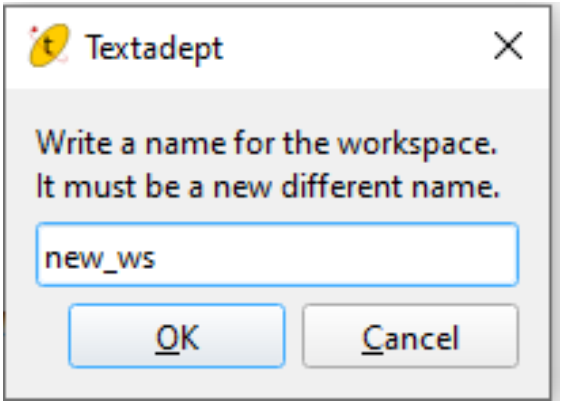


Figura 5-18. De nuevo se escribe new_WS en el menú

b. Aparece un mensaje de aviso (figura 5-19).

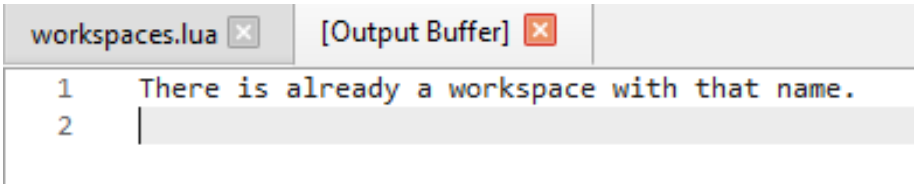


Figura 5-19. Mensaje de aviso de nombre de espacio de trabajo ya registrado

WS 10 Eliminación de un espacio de trabajo que no sea el definido por defecto

Eliminar un espacio de trabajo creado. Si fuera el espacio abierto, debe cambiar al espacio de trabajo por defecto.

a. Se selecciona de nuevo *Erase Workspace* y se elige el espacio de trabajo *new_WS* (figura 5-20).

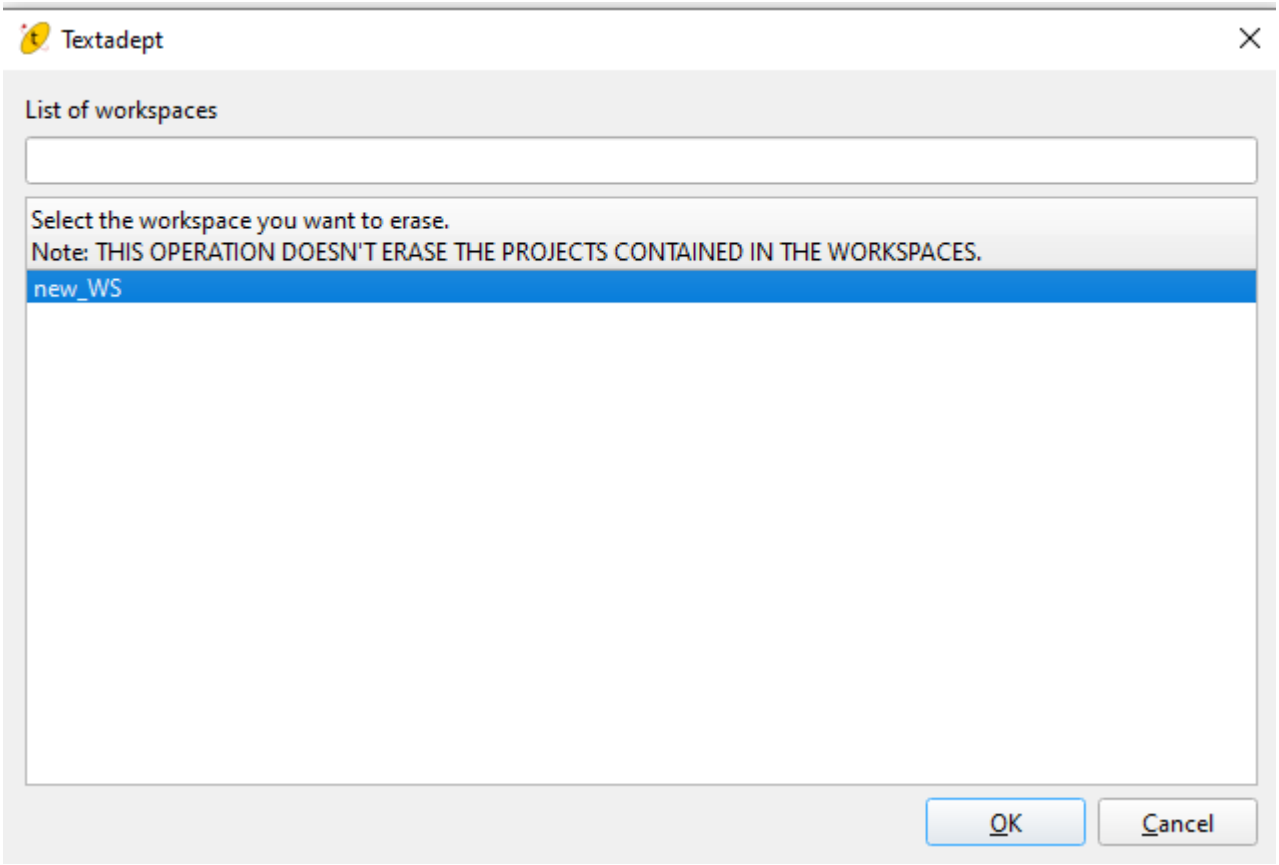


Figura 5-20. Se elige en el menú de eliminar un espacio de trabajo new_WS

b. Aparece información de esta operación como la advertencia de que, como era el espacio de trabajo abierto, se ha vuelto al espacio por defecto (figura 5-21).

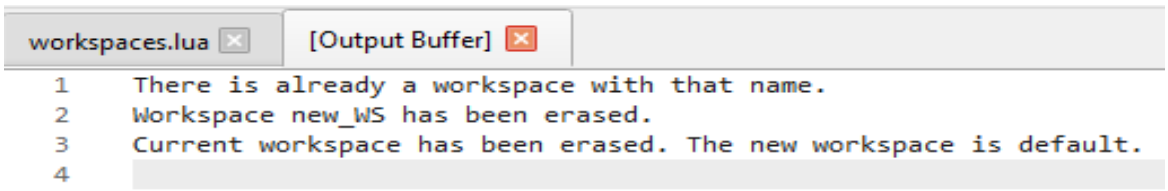


Figura 5-21. Información sobre la eliminación del espacio de trabajo y de que se ha debido cambiar al espacio de trabajo por defecto

c. Se actualiza el explorador de proyectos (figura 5-22).

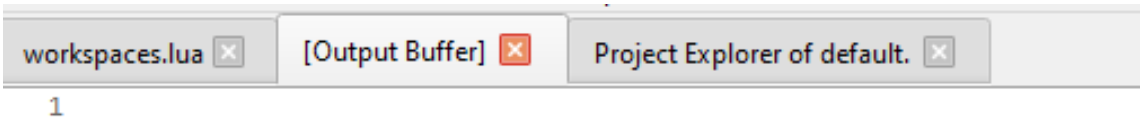


Figura 5-22. Actualización del explorador de proyectos

5.3 Gestión de proyectos

PM 1 Explorador de proyectos inicialmente vacío

Cuando se arranca el IDE por primera vez, el explorador de proyectos está vacío (figura 5-5 de la sección anterior).

PM 2 Imposibilidad de eliminar proyectos si no hay

Intentar eliminar un proyecto cuando aún no hay ninguno arroja un aviso.

a. Se selecciona en el menú la opción *Erase Project*, dotada de un atajo de teclado (figura 5-23),

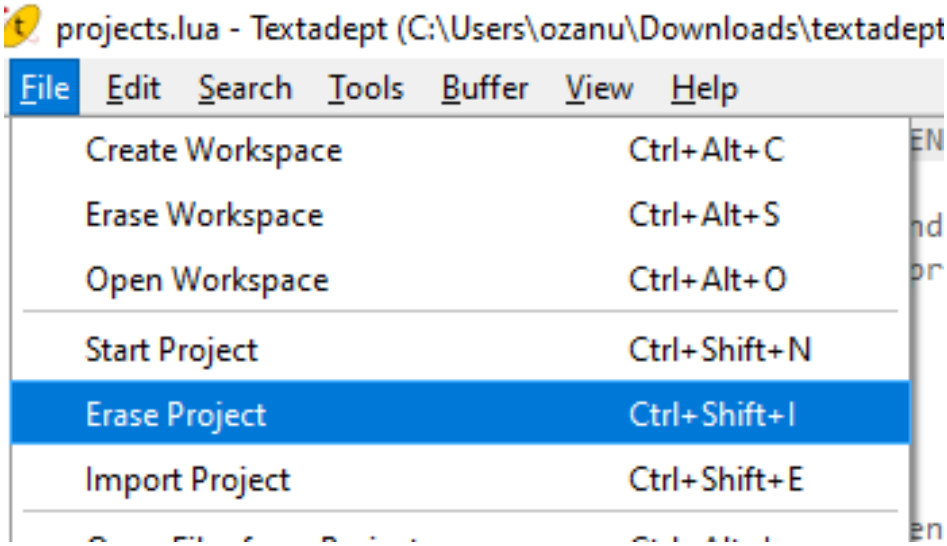


Figura 5-23. Opción Erase Project

b. Aparece un mensaje de aviso (figura 5-24).

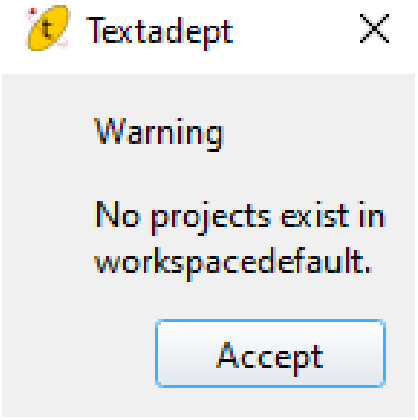


Figura 5-24. Mensaje de aviso, ya que no hay aún proyectos en el espacio de trabajo

PM 3 Imposibilidad de abrir ficheros sin proyectos

Intentar abrir ficheros de un proyecto cuando aún no hay ninguno arroja un aviso.

a. Se selecciona en el menú la opción *Open Files from Project*, dotada de un atajo de teclado (figura 5-25).

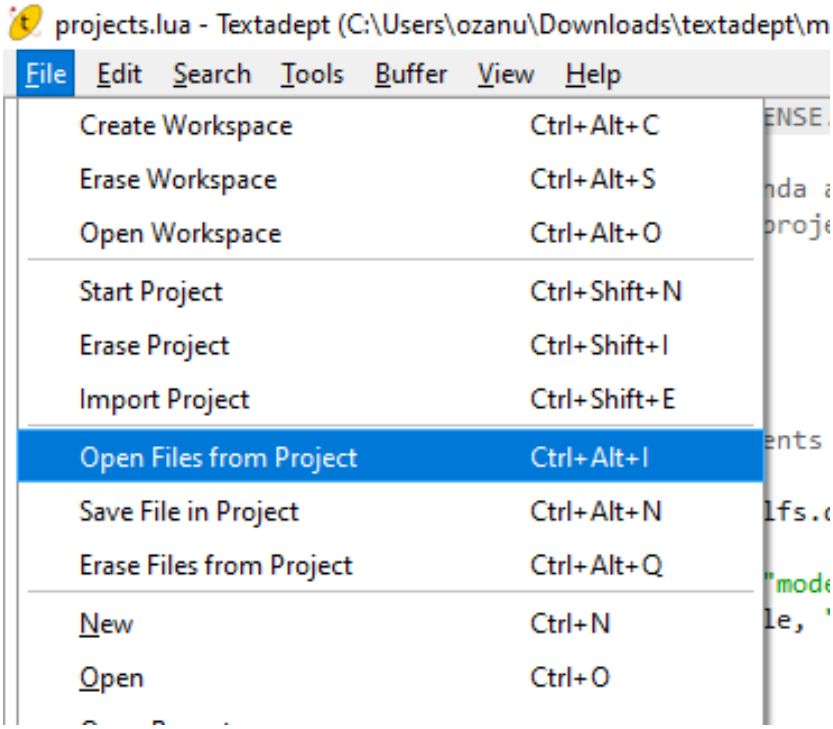


Figura 5-25. Opción Open Files from Project

b. Aparece un mensaje de aviso (figura 5-26).

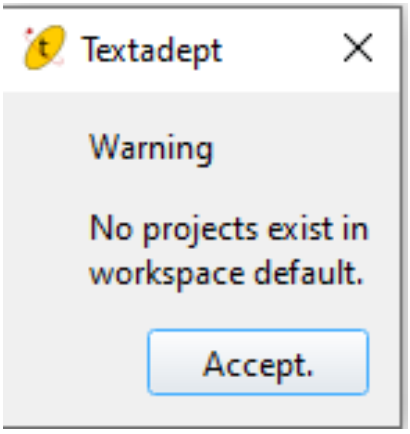


Figura 5-26. Mensaje de aviso, ya que no hay aún proyectos en el espacio de trabajo

PM 4 Imposibilidad de guardar ficheros sin proyectos

Intentar guardar ficheros de un proyecto cuando aún no hay ninguno arroja un aviso.

a. Hay un fichero abierto (figura 5-27).

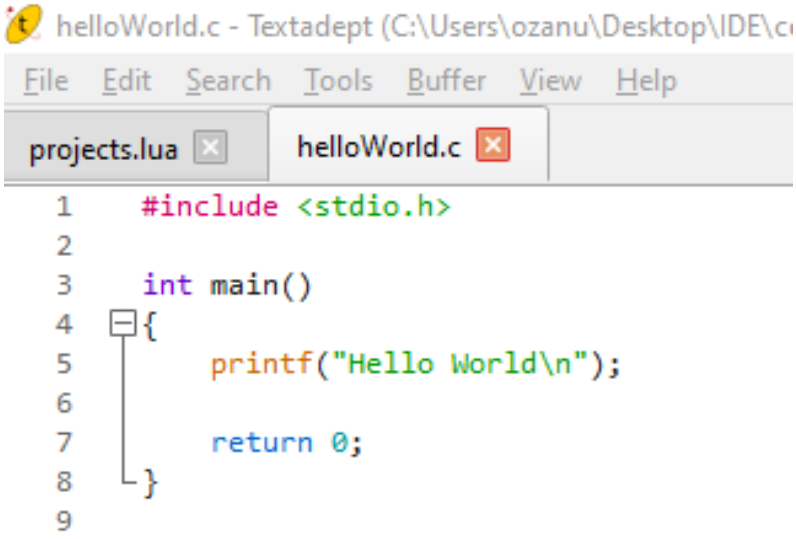


Figura 5-27. Fichero abierto en el sistema

b. Se selecciona en el menú la opción *Save File in Project*, dotada de un atajo de teclado (figura 5-28).

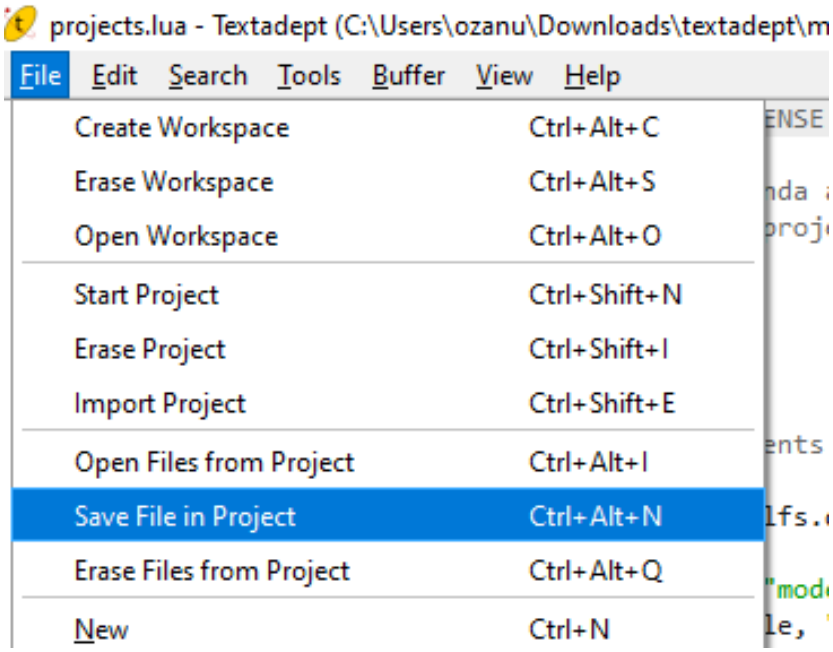


Figura 5-28. Opción Save File in Project

c. Aparece un mensaje de aviso (figura 5-29).

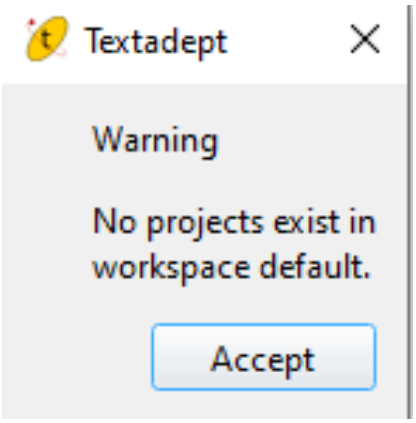


Figura 5-29. Mensaje de aviso, ya que no hay aún proyectos en el espacio de trabajo

PM 5 Imposibilidad de eliminar ficheros sin proyectos

Intentar eliminar ficheros de un proyecto cuando aún no hay ninguno arroja un aviso.

a. Se selecciona en el menú la opción *Erase Files from Project*, dotada de un atajo de teclado (figura 5-30).

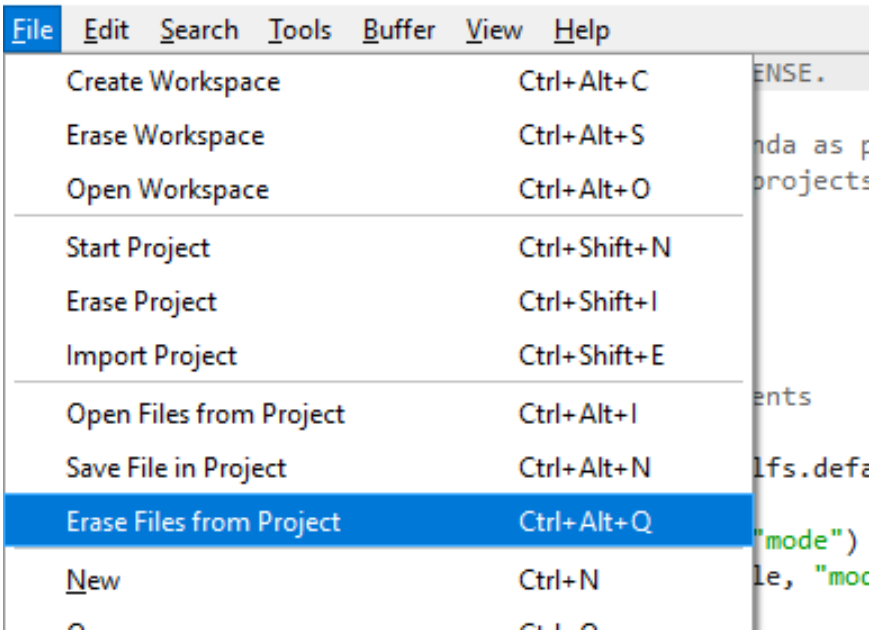


Figura 5-30. Opción Erase Files from Project

b. Aparece un mensaje de aviso (figura 5-31).

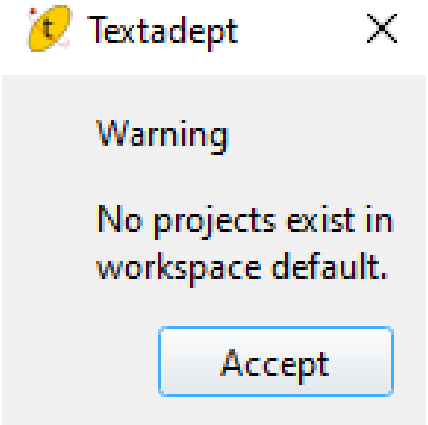


Figura 5-31. Mensaje de aviso, ya que no hay aún proyectos en el espacio de trabajo

PM 6 Crear un proyecto

Empezar un proyecto y comprobar que existe un nuevo directorio en el espacio de trabajo con el nombre indicado, así como en el explorador de proyectos.

a. El espacio de trabajo al principio no cuenta con directorios específicos de proyectos (figura 5-32).

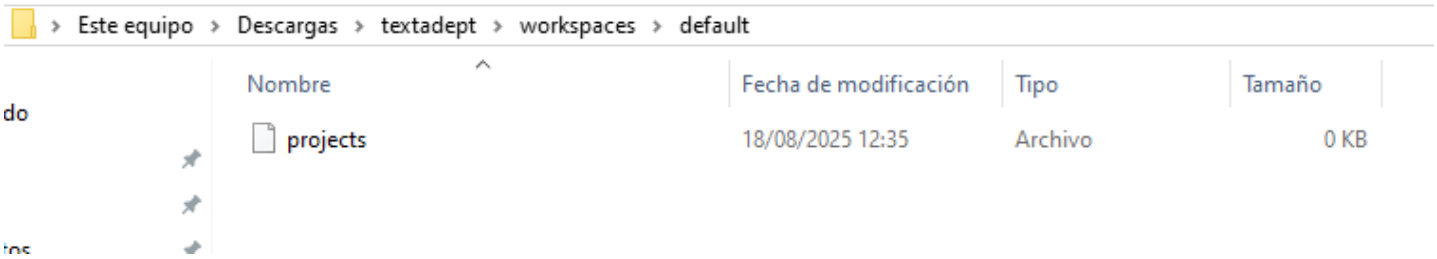


Figura 5-32. Contenido inicial del directorio específico del espacio de trabajo

b. Se selecciona en el menú la opción *Start Project*, dotada de un atajo de teclado (figura 5-33).

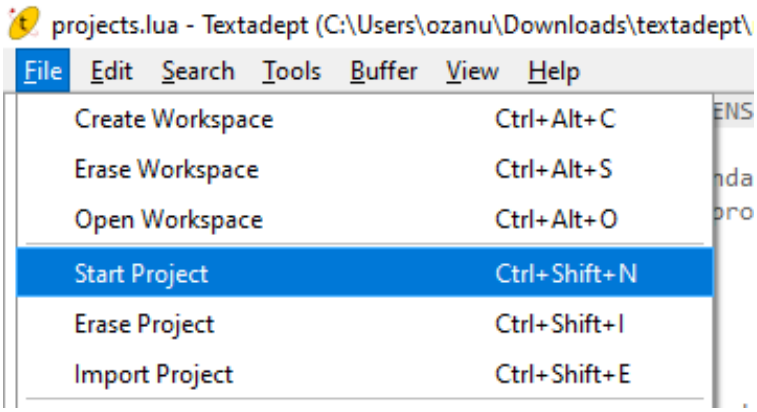


Figura 5-33. Opción Start Project

c. Se abre un menú en que el usuario puede escribir el nombre de un nuevo proyecto, *new_project* (figura 5-34).

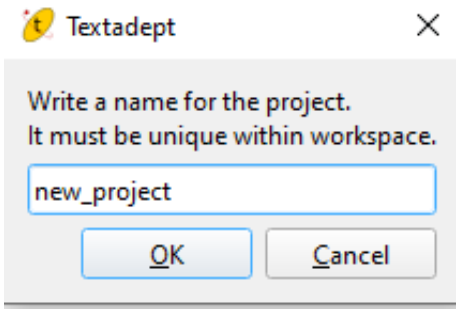


Figura 5-34. Menú que permite al usuario escribir el nombre del proyecto

d. Aceptar conlleva la aparición de un mensaje (figura 5-35).

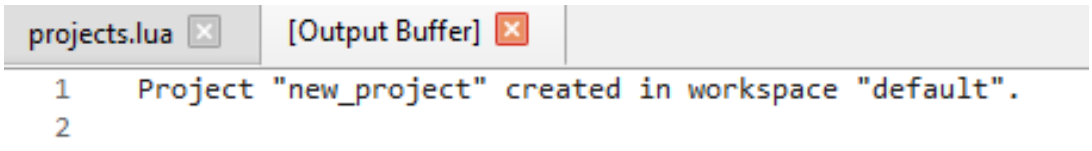


Figura 5-35. Información respecto a la operación de creación de un proyecto

e. Se crea de un nuevo subdirectorio en el directorio del espacio de trabajo abierto (figura 5-36).

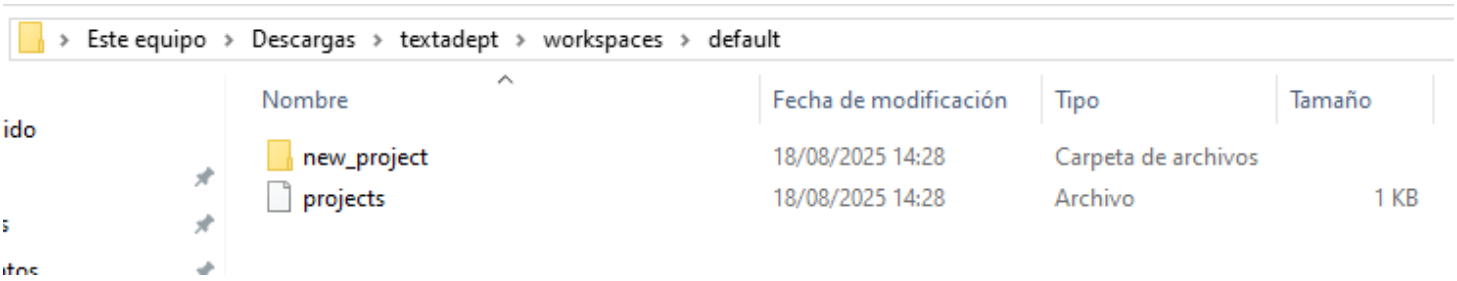


Figura 5-36. Ha aparecido un directorio del mismo nombre que el proyecto creado

f. Se actualiza explorador de proyectos (figura 5-37).

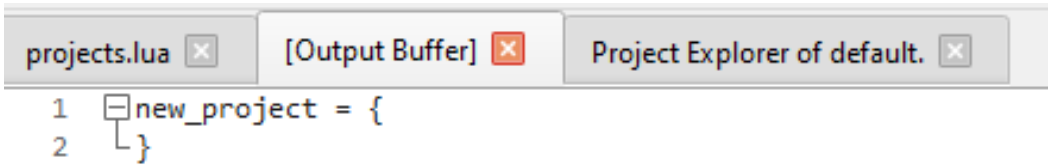


Figura 5-37. El explorador de proyectos se ha actualizado

PM 7 Unicidad de los nombres de los proyectos dentro de un espacio de trabajo

Intentar empezar un proyecto cuyo nombre ya esté siendo usado en el espacio de trabajo arroja un aviso.

a. Se selecciona de nuevo Start Project para volver a escribir el nombre new_project (figura 5-38).

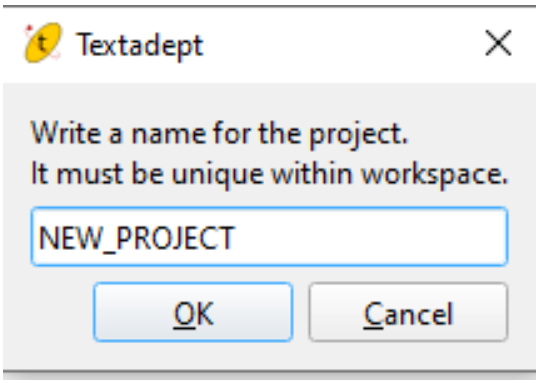


Figura 5-38. De nuevo se escribe new_project en el menú

b. Aparece un mensaje de aviso (figura 5-39).

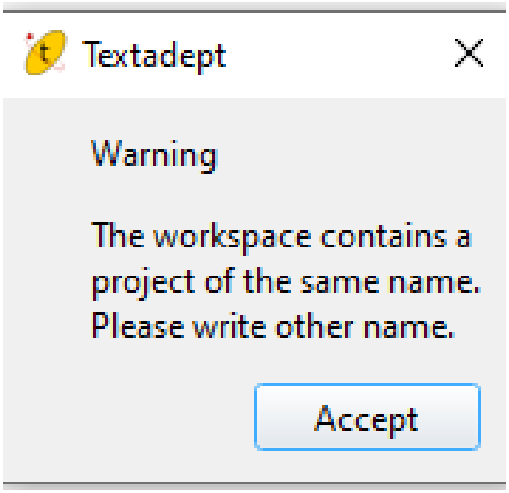


Figura 5-39. Mensaje que avisa de que ya hay otro proyecto con el mismo nombre

PM 8 Importar un proyecto

Importar un proyecto ya existente y comprobar que actualiza el explorador de proyectos.

a. Inicialmente, existe un proyecto situado fuera del espacio de trabajo default (figura 5-40).

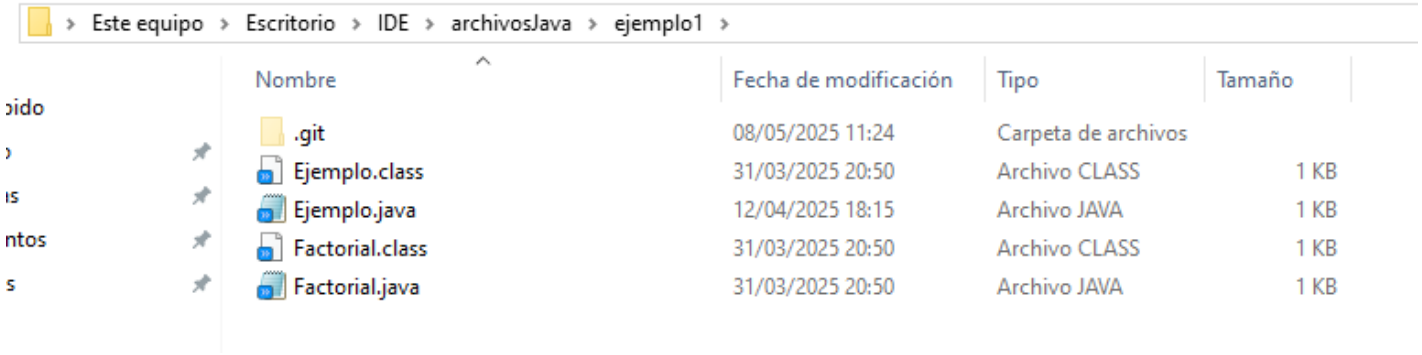


Figura 5-40. Contenido de un proyecto, llamado ejemplo1

b. Se selecciona en el menú la opción *Import Project*, dotada de un atajo de teclado (figura 5-41).

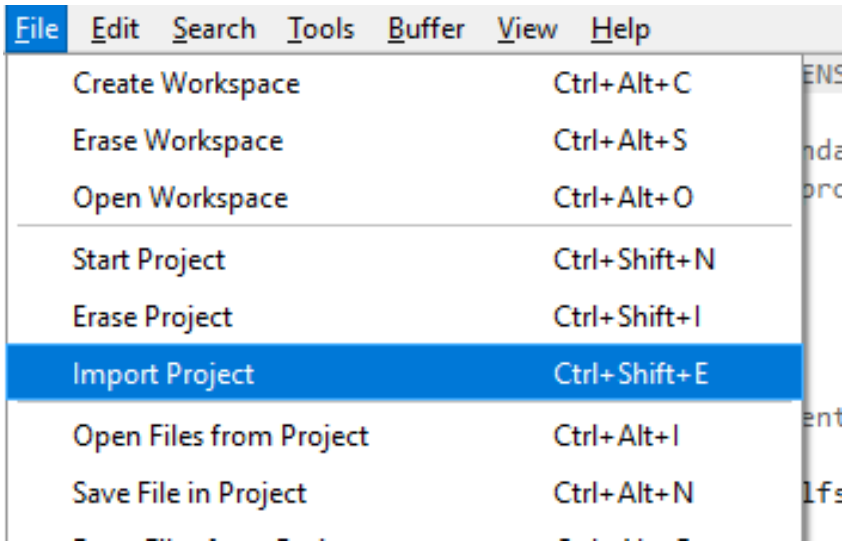


Figura 5-41. Opción Import Project

c. Se abre un menú en que el usuario puede buscar la ruta del proyecto (figura 5-42).

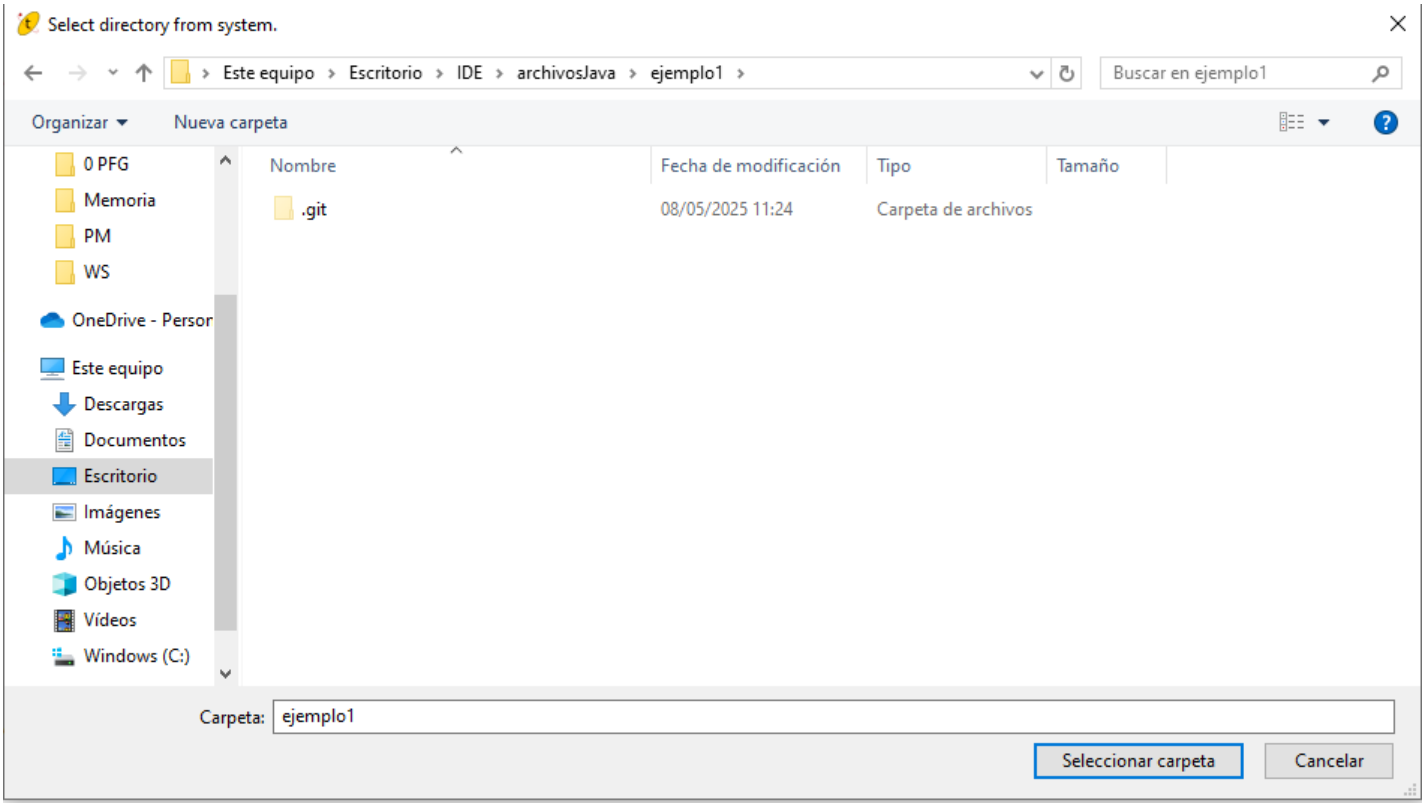


Figura 5-42. Ventana del sistema de ficheros abierta en el directorio que contiene el proyecto

d. Aceptar conlleva la aparición de un mensaje (figura 5-43).

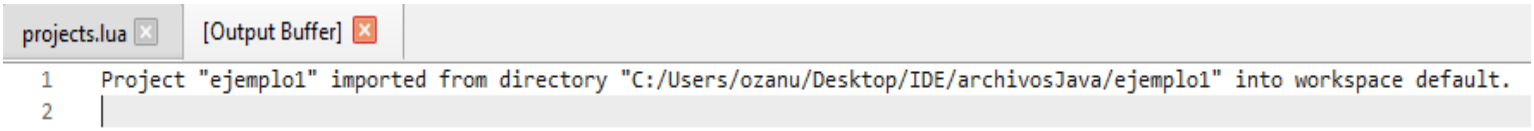


Figura 5-43. Mensaje de confirmación del nuevo proyecto importado

e. Se crea un nuevo subdirectorio en el directorio del espacio de trabajo abierto, con el mismo contenido que el original (figura 5-44).

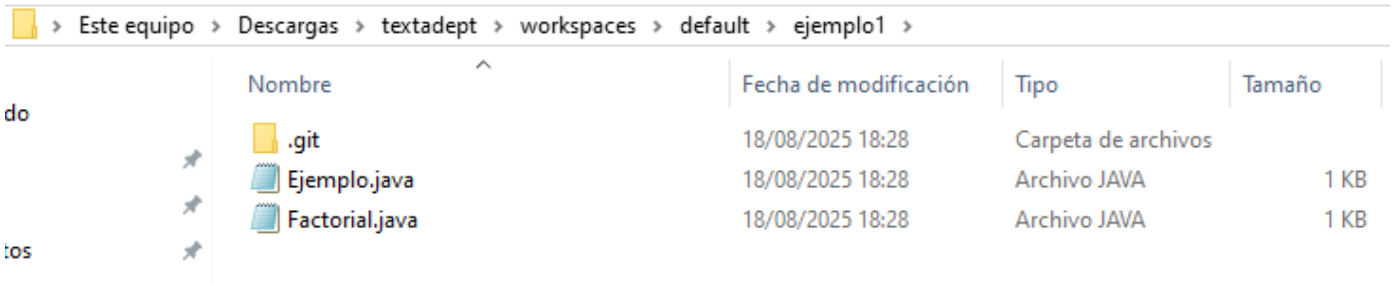


Figura 5-44. El contenido del nuevo proyecto, dentro del espacio de trabajo en uso, es como el del proyecto originalmente importado

f. Se actualiza el explorador de proyectos (figura 5-45).

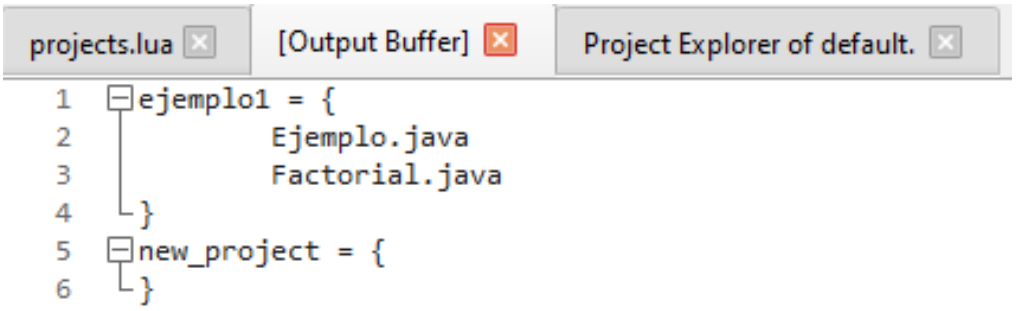


Figura 5-45. El explorador de proyectos se ha actualizado

PM 9 Importar un proyecto de nombre ya existente en el espacio de trabajo

Intentar importar un proyecto con el mismo nombre que otro ya existente en el espacio de trabajo arroja un aviso.

a. Se selecciona de nuevo *Import Project* para elegir un proyecto en una ruta externa también llamado *new_project* (figura 5-46).

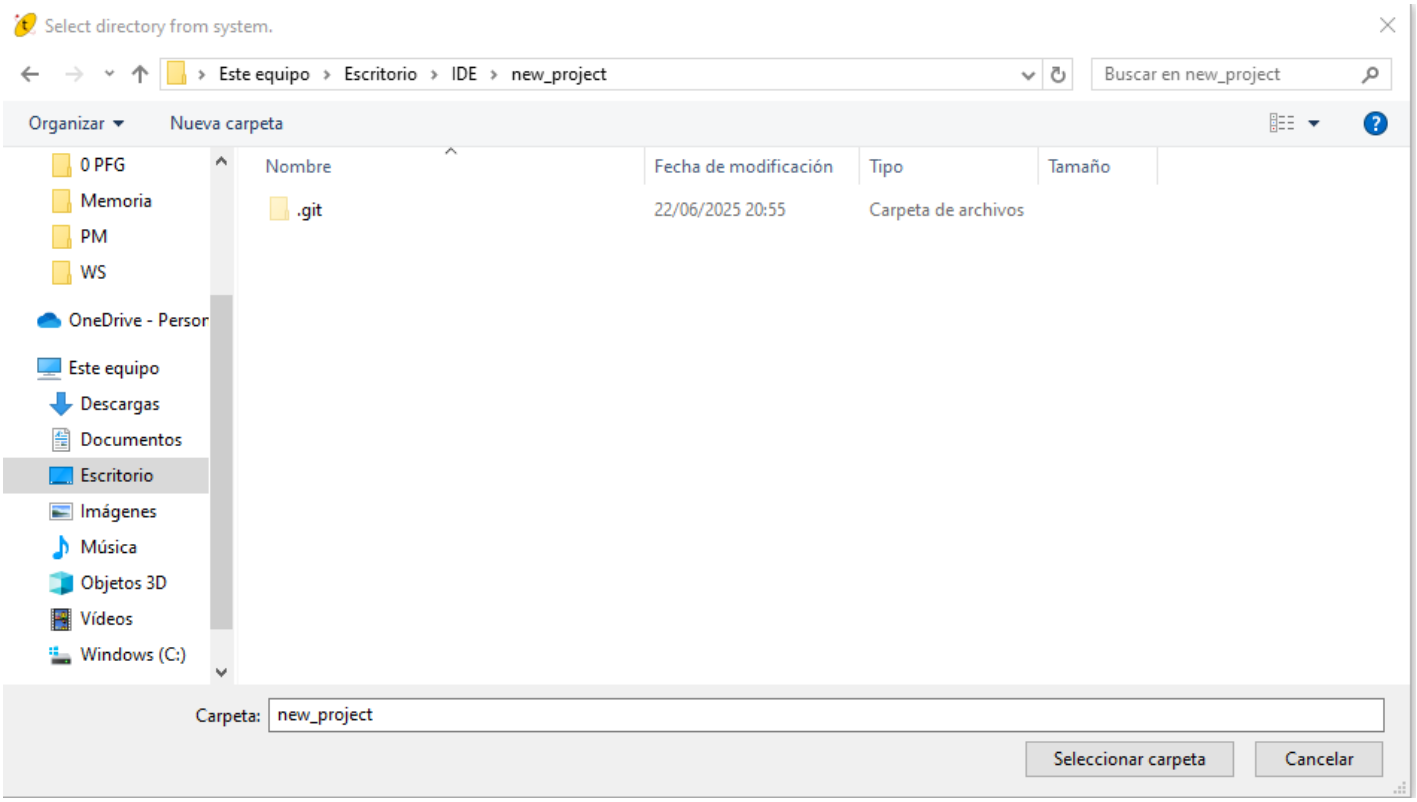


Figura 5-46. Se selecciona un proyecto fuera del espacio de trabajo llamado new_project, como el creado en una prueba anterior

b. Aparece un mensaje de aviso (figura 5-47).

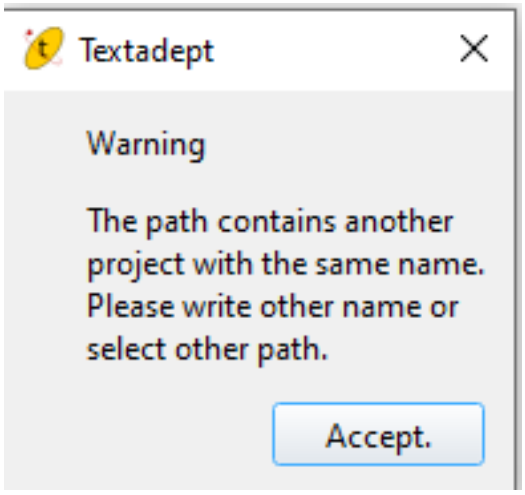


Figura 5-47. Mensaje de aviso porque hay otro proyecto con el mismo nombre en el espacio de trabajo

PM 10 Eliminar un proyecto

Eliminar un proyecto creado con el IDE y comprobar su efecto en el explorador de proyectos.

a. Se selecciona de nuevo Erase Project y se elige eliminar new_project (figura 5-48).

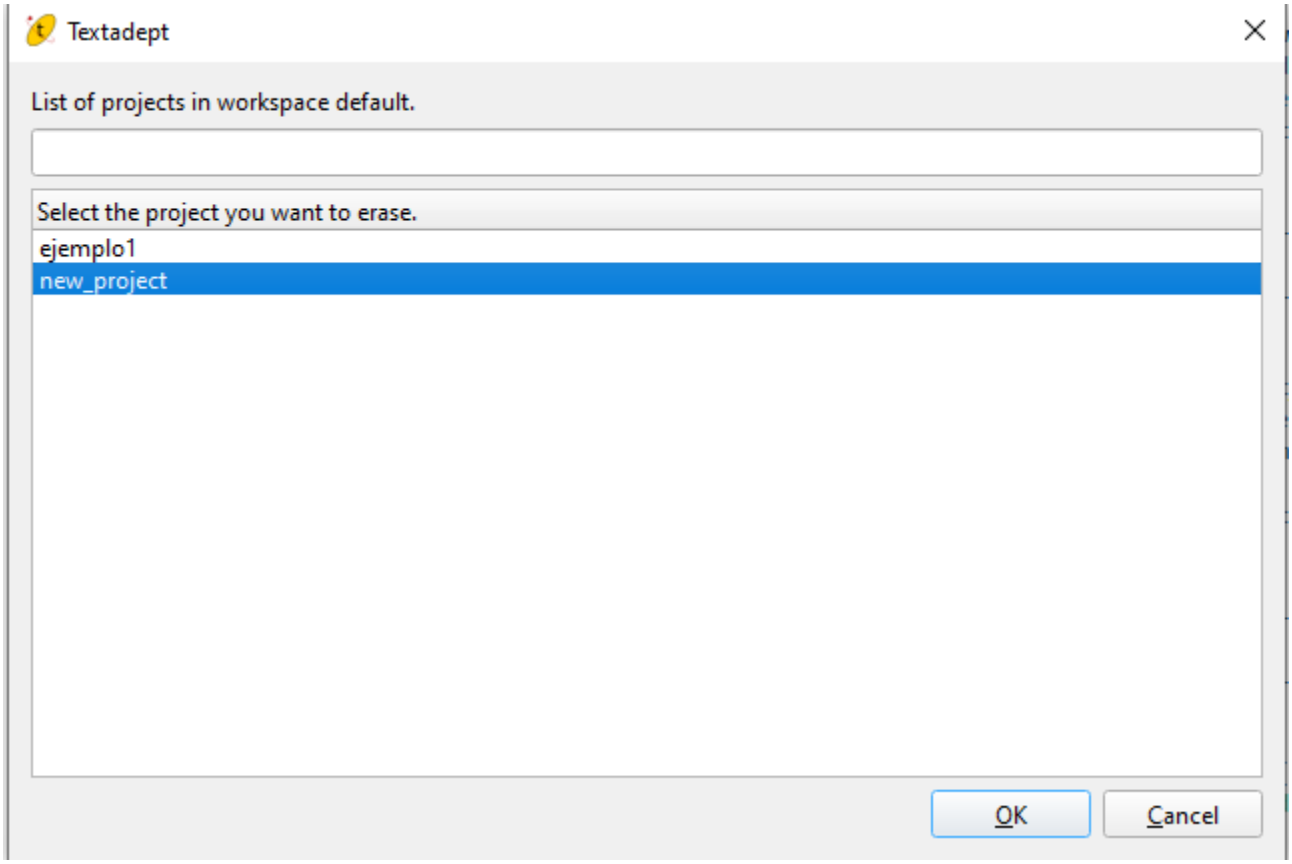


Figura 5-48. En el menú de eliminación se elige el proyecto new_project

b. Se abre una pregunta de si eliminar el directorio (figura 5-49).

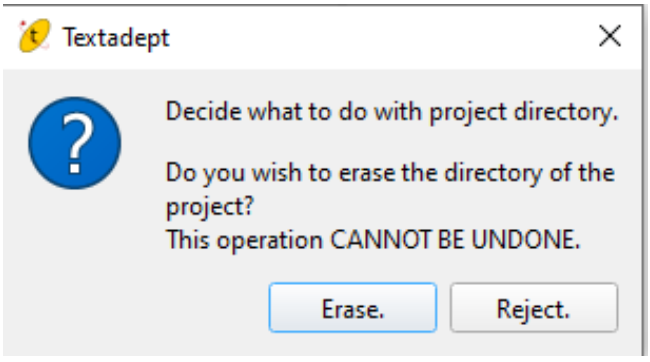


Figura 5-49. Pregunta de confirmación acerca de la eliminación de los directorios del proyecto

c. Aparece un mensaje de confirmación (figura 5-50).

```
1   Project "new_project" erased from workspace default.
2   The directory of the project new_project has not been erased.
3
```

Figura 5-50. Mensaje de confirmación de la eliminación del proyecto

d. Se actualiza el explorador de proyectos (figura 5-51).

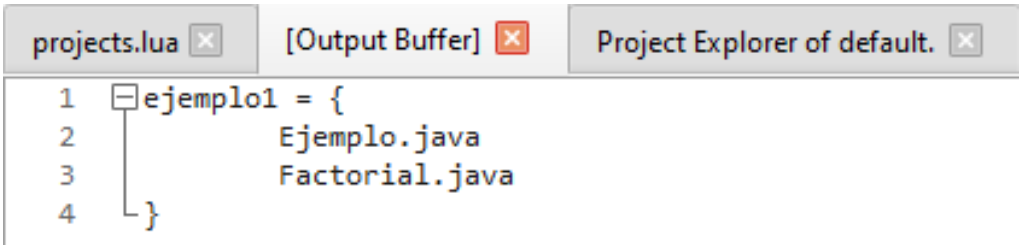


Figura 5-51. El explorador de proyectos se ha actualizado

e. No obstante, el directorio no ha sido eliminado del espacio de trabajo *default* (figura 5-52).

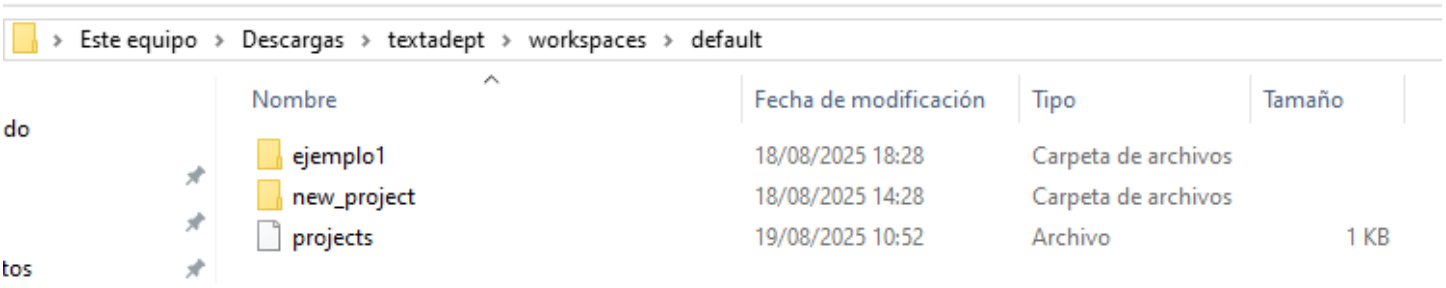


Figura 5-52. Dentro del espacio de trabajo en uso, aún está el directorio del proyecto eliminado

PM 11 Eliminar un proyecto y su directorio

Eliminar un proyecto creado con el IDE, comprobar su efecto en el explorador de proyectos y además eliminar su carpeta.

a. Hay dos ficheros abiertos de *ejemplo1* (figura 5-53).

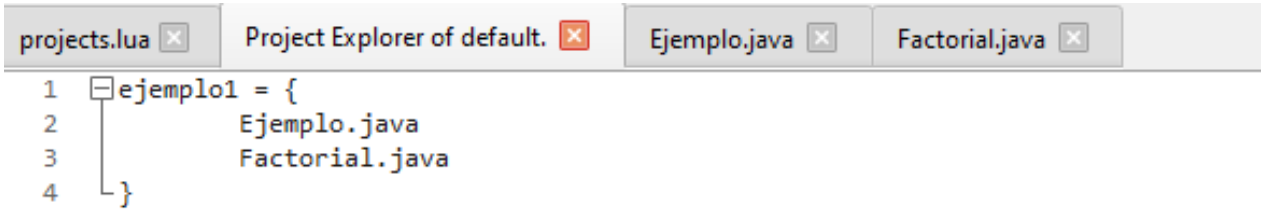


Figura 5-53. Los ficheros Ejemplo.java y Factorial.java están abiertos. Se puede apreciar el código del primero

b. Se selecciona de nuevo *Erase Project* y se elige la eliminación de este proyecto (figura 5-54).

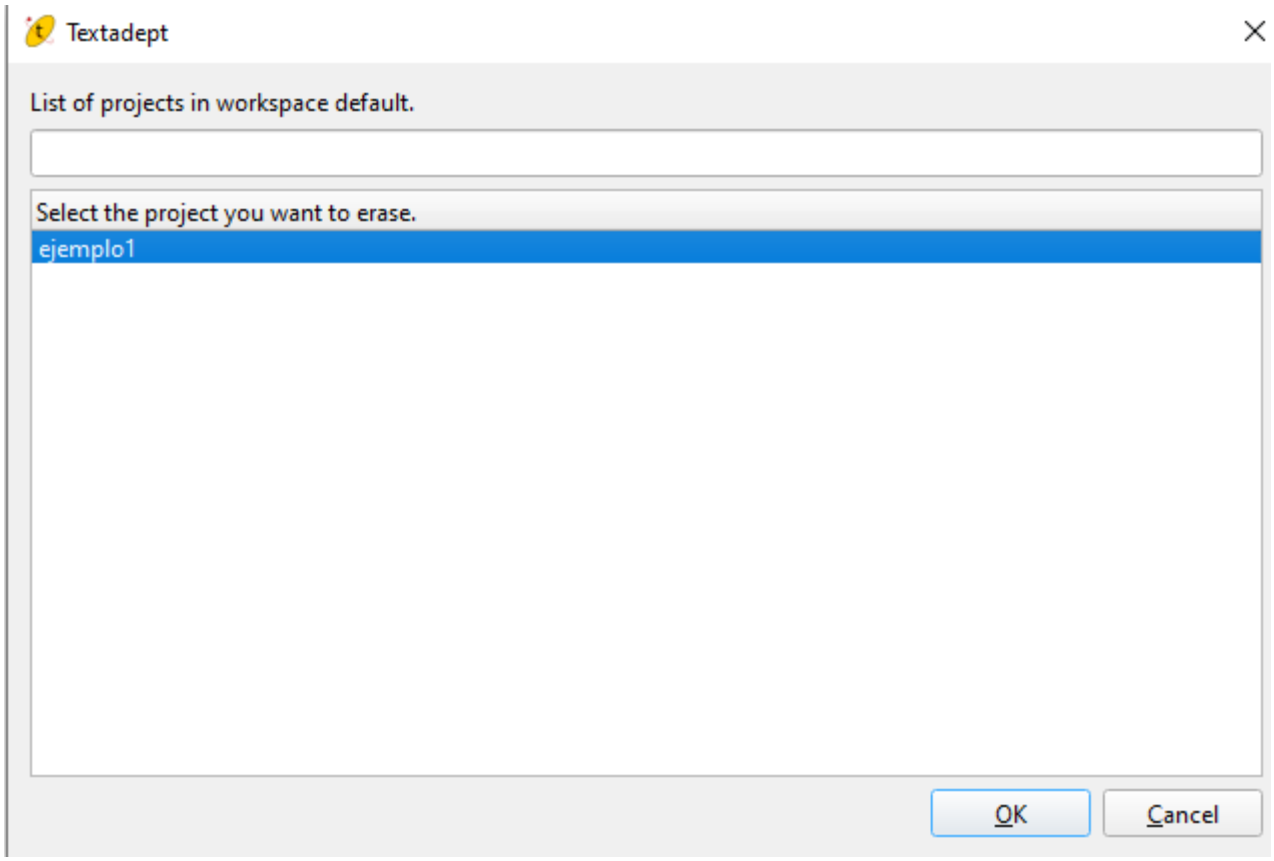


Figura 5-54. Se vuelve a eliminar un proyecto en este menú

c. Lleva a las mismas preguntas de la anterior sección. Elegir que sí conlleva que se cierren los ficheros (figura 5-55).

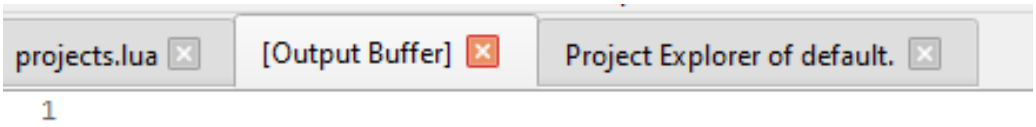


Figura 5-55. Los ficheros han sido cerrados

d. Ahora sí ha desaparecido el directorio en el espacio de trabajo (figura 5-56).

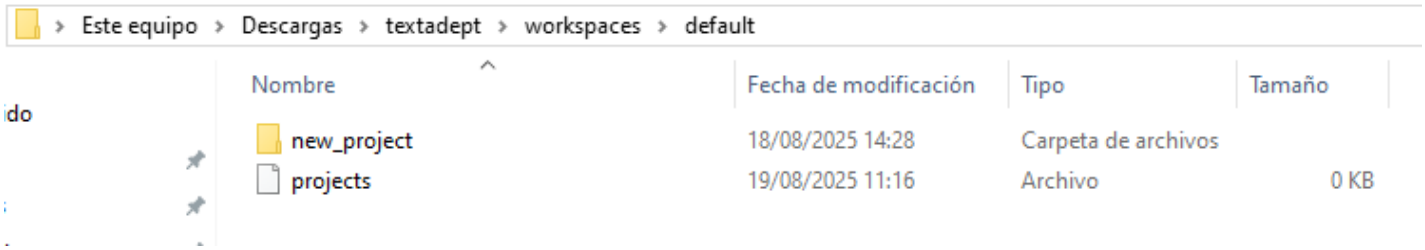


Figura 5-56. En el espacio de trabajo en uso, ya no aparece el directorio del proyecto eliminado

PM 12 Empezar otro proyecto con el mismo nombre que el previamente eliminado

Volver a empezar el proyecto eliminado anteriormente sin impedimentos, con el mismo nombre y ruta.

a. Se selecciona de nuevo *Erase Project* y se escribe de nuevo *new_project* (figura 5-57).

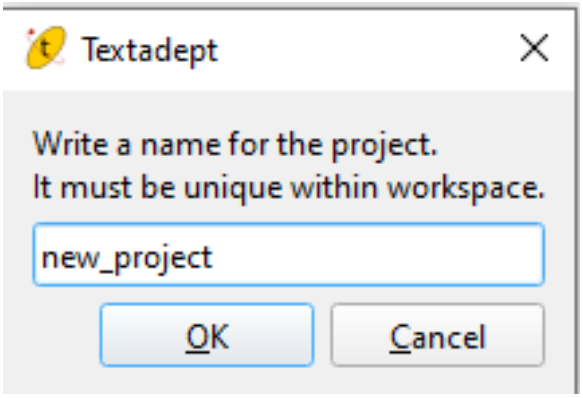


Figura 5-57. Se vuelve a crear un proyecto llamado new_project

b. La situación final es idéntica a cuando se creó por primera vez (figura 5-58).

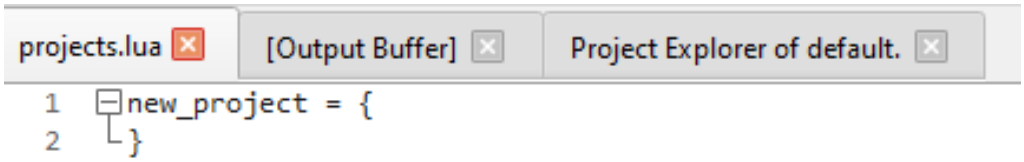


Figura 5-58. El explorador de proyectos se ha actualizado como cuando se creó en una prueba anterior

PM 13 Imposibilidad de abrir ficheros de un proyecto vacío

Intentar abrir ficheros del nuevo proyecto arroja un mensaje de aviso al estar vacío.

a. Se selecciona de nuevo *Open Files from Project* abre un menú para elegir el proyecto (figura 5-59).

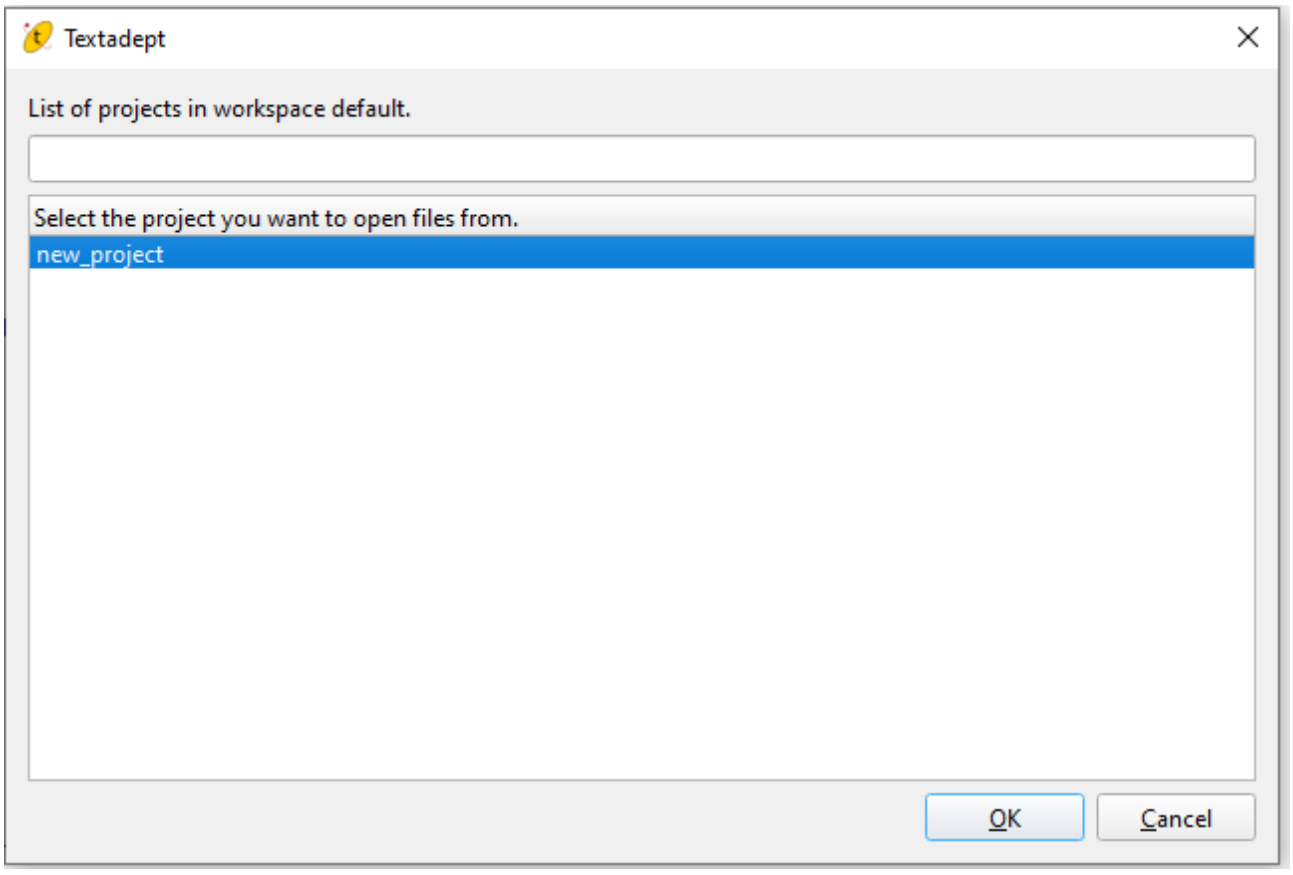


Figura 5-59. Menú para elegir proyecto

b. Aparece un aviso porque el proyecto no contiene fichero (figura 5-60).

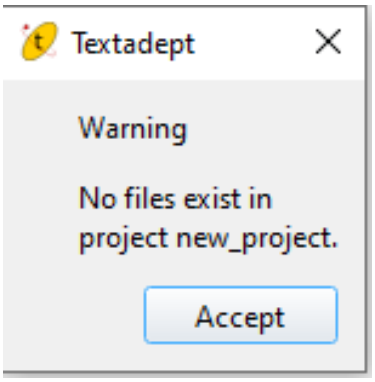


Figura 5-60. Advertencia sobre el hecho de que el proyecto elegido no contiene ficheros

PM 14 Imposibilidad de eliminar ficheros de un proyecto vacío

Intentar eliminar ficheros del nuevo proyecto arroja un mensaje de aviso al estar vacío.

a. La imagen muestra que seleccionar de nuevo *Erase Files from Project* abre un menú para elegir el proyecto (figura 5-61).

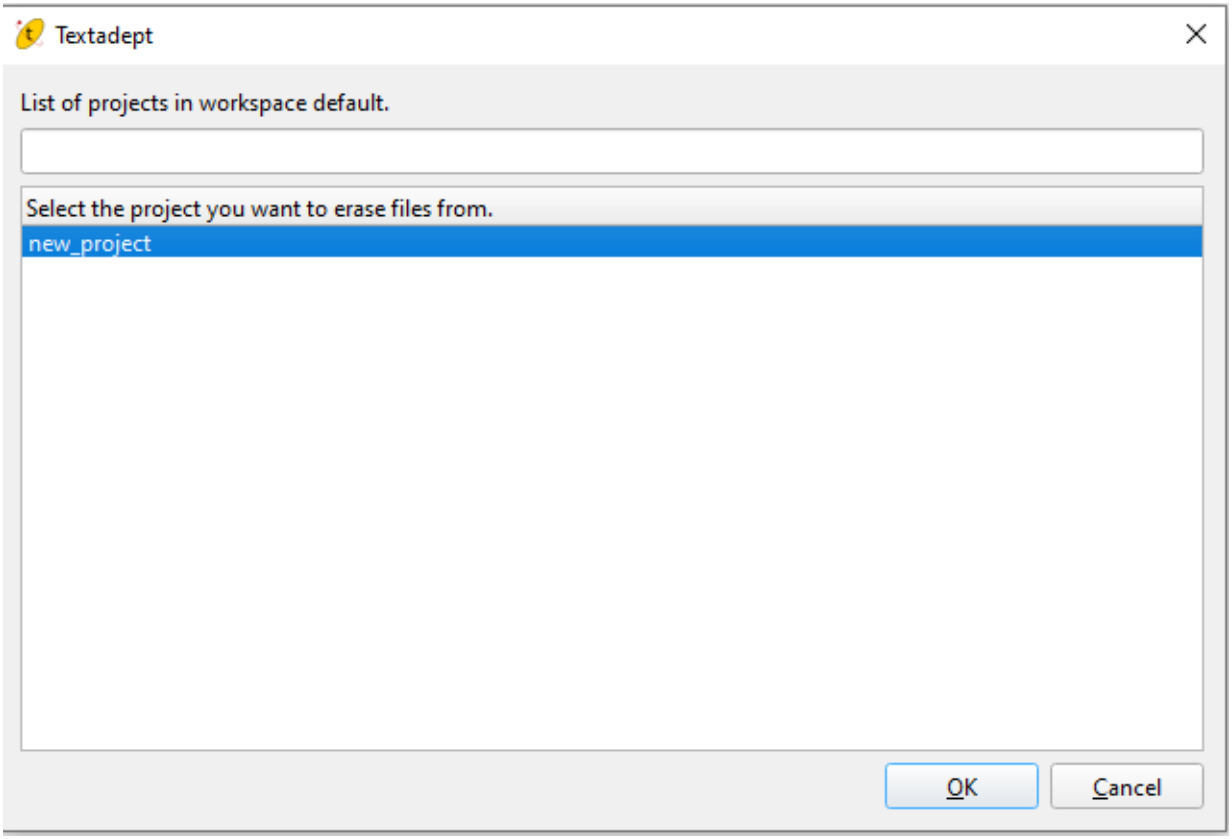


Figura 5-61. Menú para elegir el proyecto del que se quieren eliminar ficheros

b. No obstante, al estar vacío el único disponible, el sistema arroja un aviso (figura 5-62).

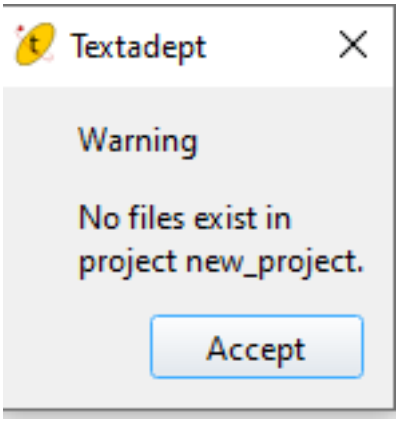


Figura 5-62. Advertencia sobre el hecho de que el proyecto elegido no contiene ficheros

PM 15 Guardar un fichero en un proyecto

Guardar un archivo en un proyecto y comprobar su efecto en el explorador de proyectos.

a. Inicialmente hay un fichero abierto (figura 5-63).

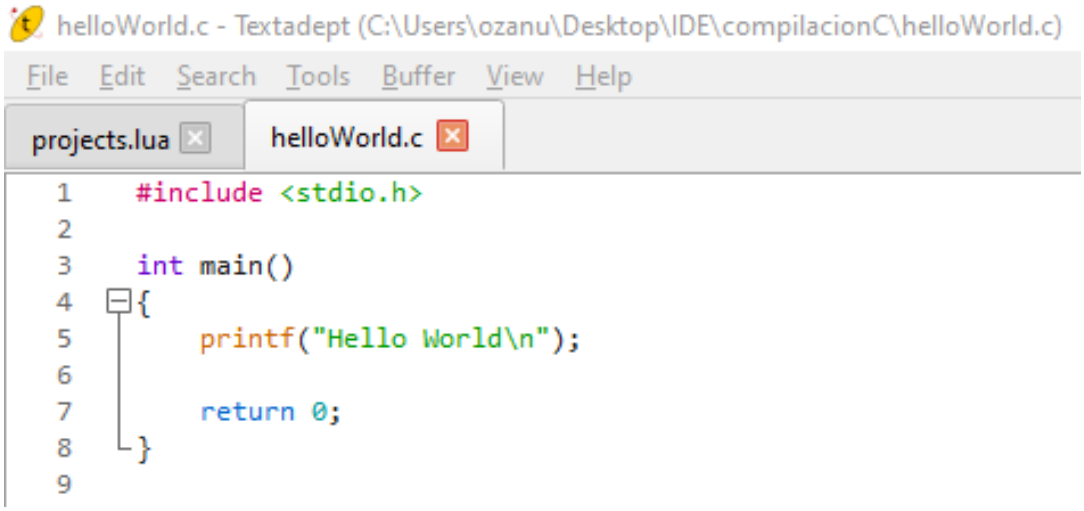


Figura 5-63. El fichero helloWorld.c está abierto

b. Se selecciona de nuevo *Save File in Project*, lo que abre un menú para seleccionar un proyecto (figura 5-64).

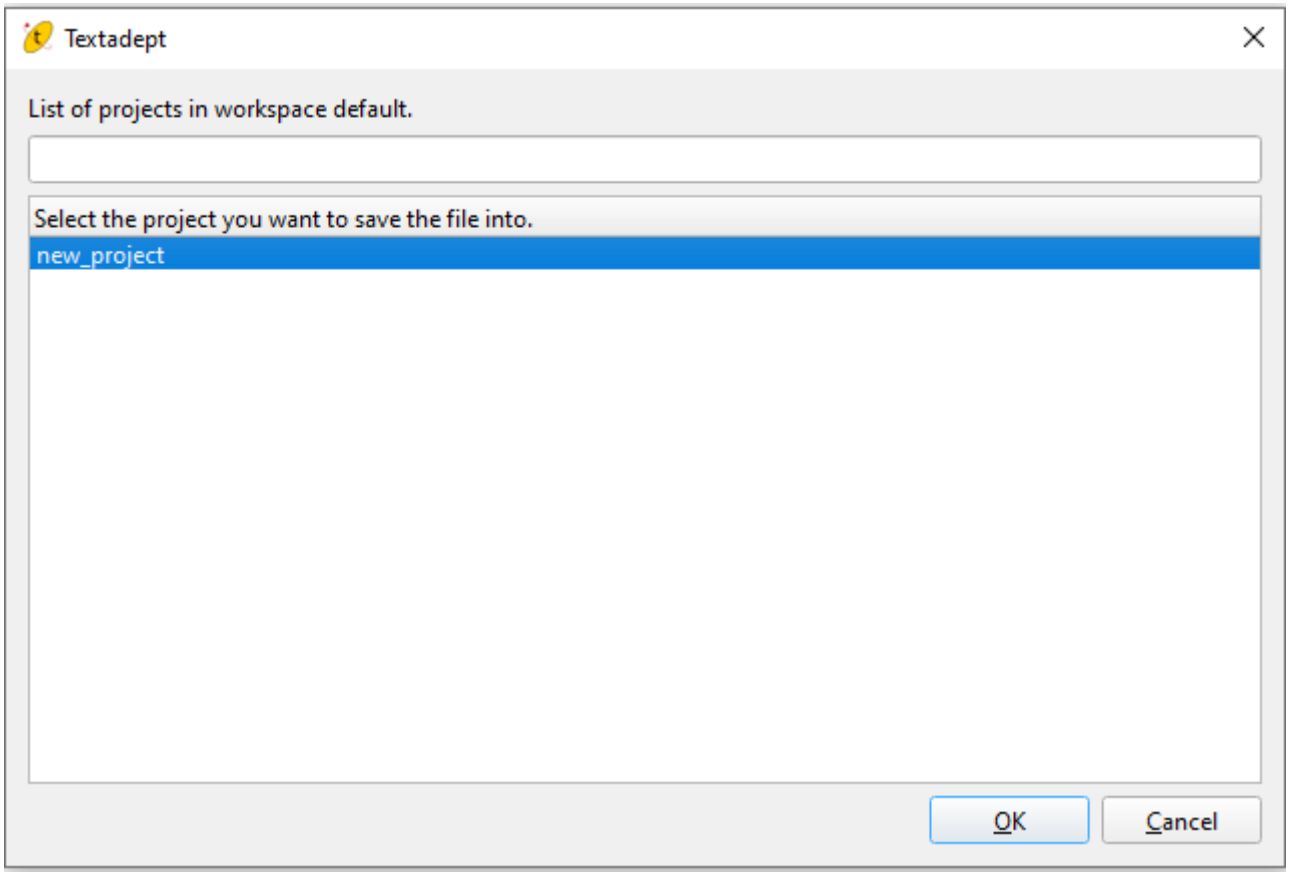


Figura 5-64. Menú para elegir el proyecto donde guardar el fichero

c. Una vez elegido, aparece un nuevo menú en que se puede escribir un nombre, *new_file.c* en el ejemplo (figura 5-65).

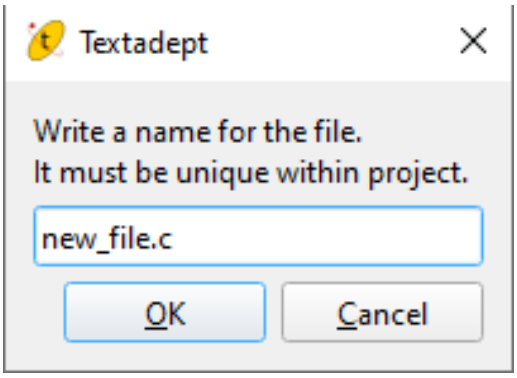


Figura 5-65. Diálogo que recoge el nombre con el cual se quiere guardar el fichero

d. Esto arroja un mensaje de confirmación (figura 5-66).

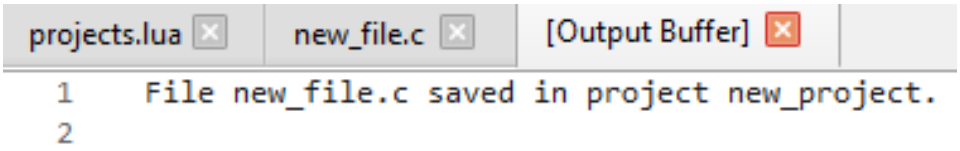


Figura 5-66. Mensaje de confirmación del fichero guardado en el proyecto

e. Cambia el nombre del fichero (figura 5-67).

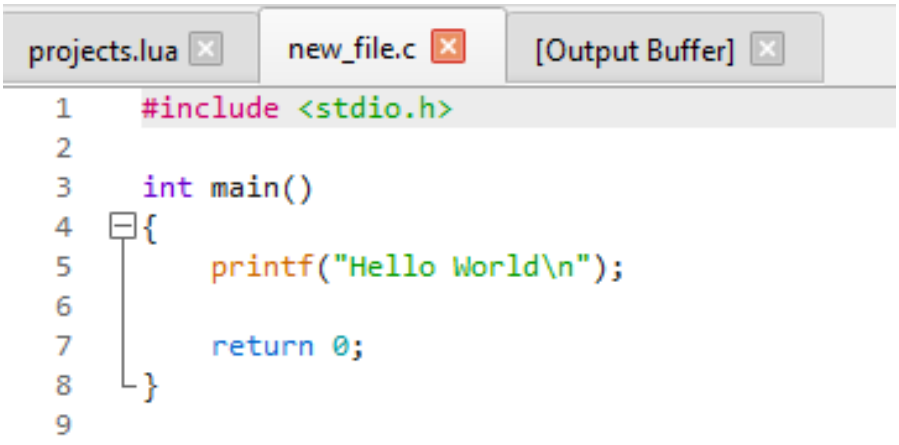


Figura 5-67. El fichero muestra en su cabecera el nuevo nombre

PM 16 Unicidad de los nombres de los ficheros en un proyecto

Intentar guardar un nuevo fichero con el mismo nombre arroja un mensaje de error.

a. Hay un nuevo fichero abierto (figura 5-68).

Figura 5-68. Un nuevo fichero abierto

b. Se vuelve a elegir *Save File in Project*, escogiendo el mismo proyecto con el mismo nombre (figura 5-69).

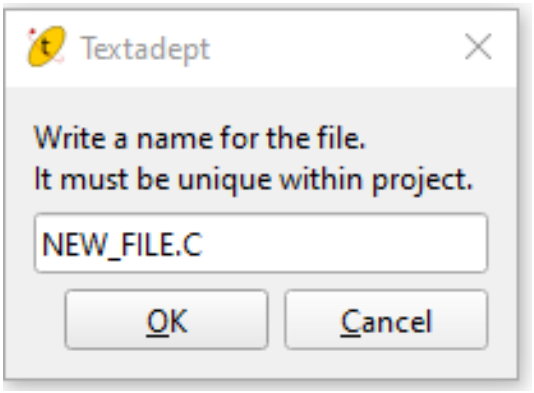


Figura 5-69. Se vuelve a escribir el mismo nombre

c. Aparece un mensaje de aviso (figura 5-70).

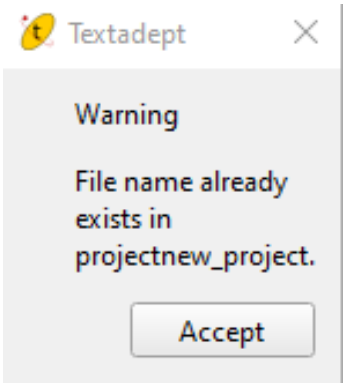


Figura 5-70. Mensaje de advertencia, ya que hay un fichero de mismo nombre en el proyecto

PM 17 Abrir ficheros en un proyecto

Abrir ficheros de un proyecto y comprobar su efecto en el explorador de proyectos.

a. La imagen muestra que inicialmente no hay ficheros abiertos de ningún proyecto (figura 5-71).

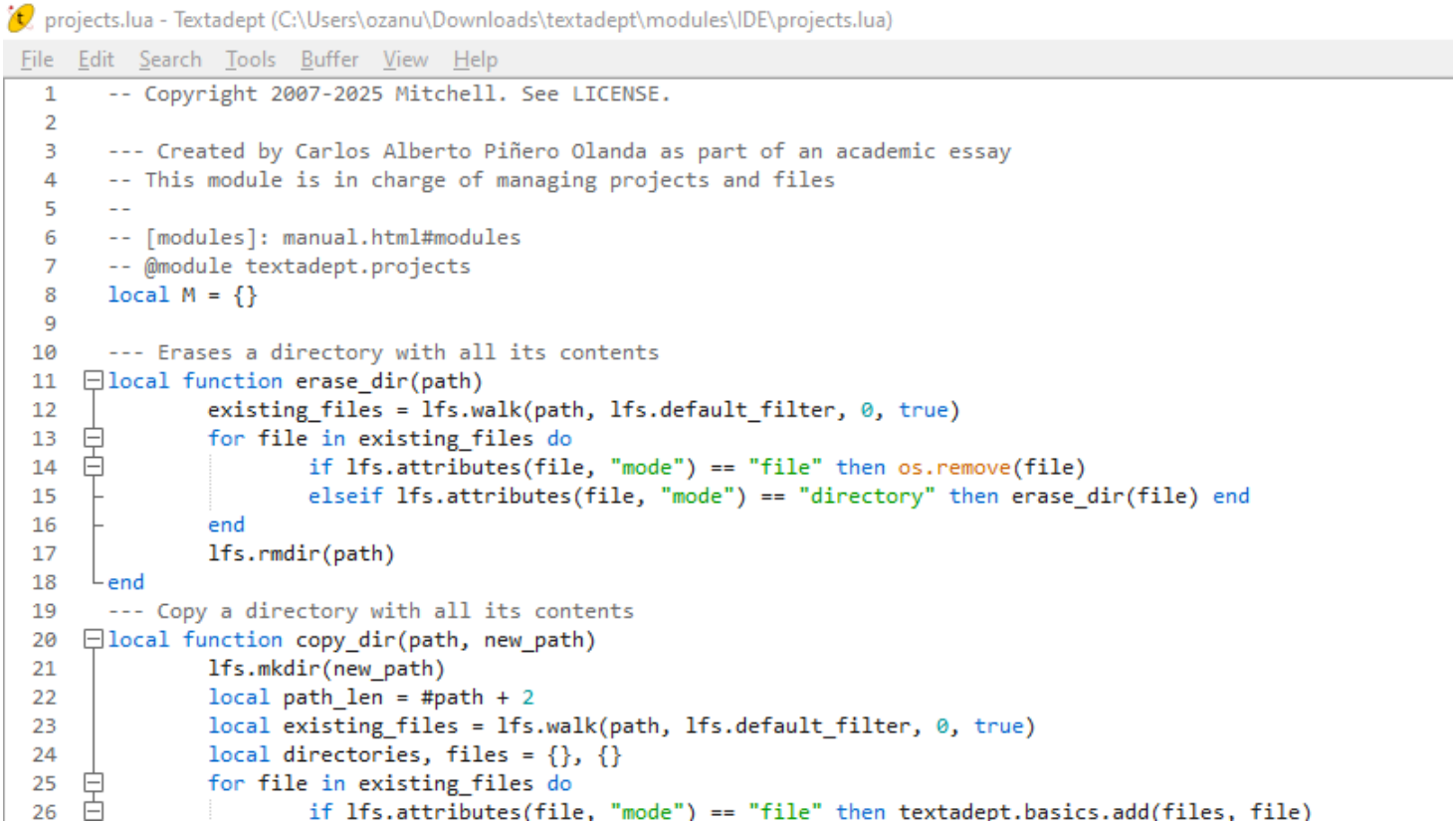


Figura 5-71. No hay ficheros abiertos, excepto projects.lua

b. Se selecciona de nuevo *Open Files from Project*, se elige un proyecto y se abre un menú en que se pueden seleccionar varios ficheros (figura 5-72).

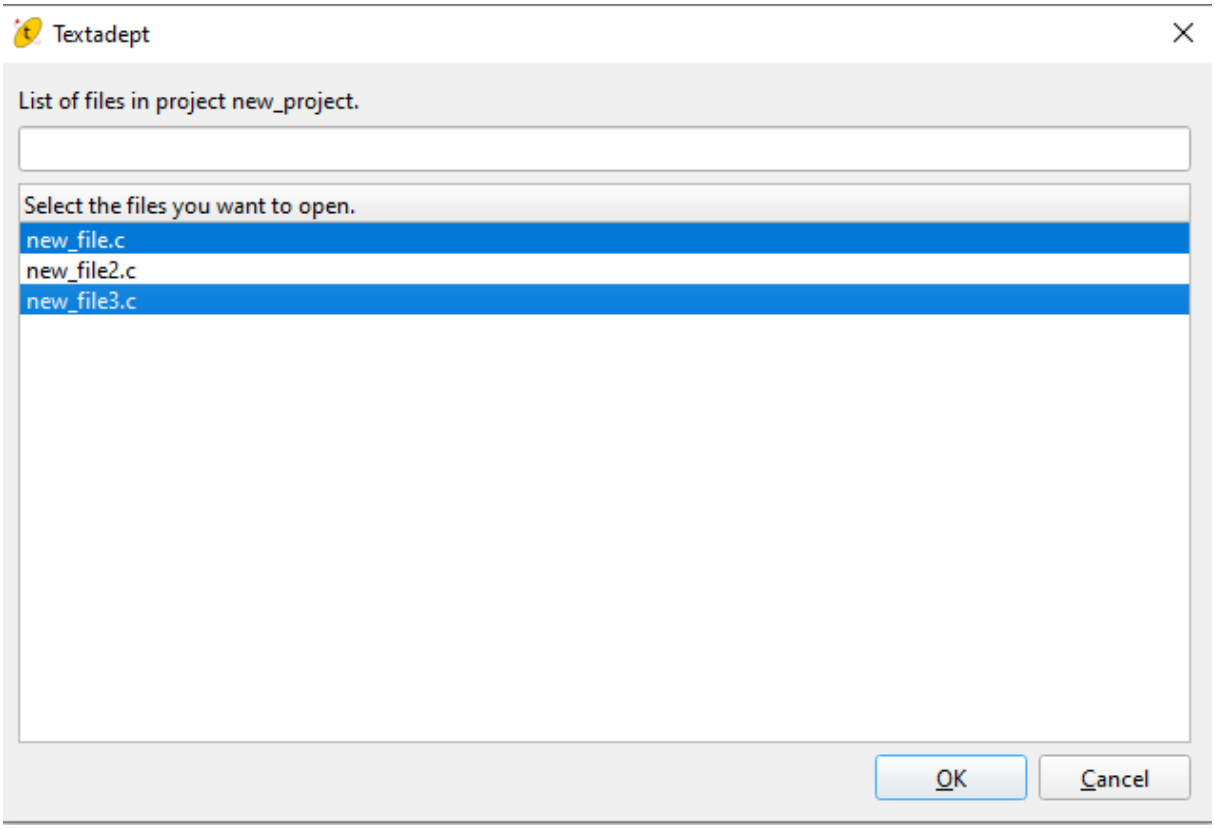


Figura 5-72. Se han elegido dos ficheros para abrir

c. Los ficheros están abiertos (figura 5-73).

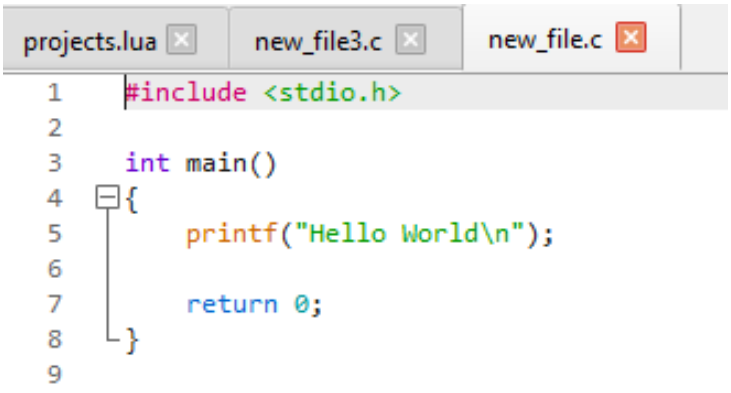


Figura 5-73. Los dos ficheros elegidos están abiertos. Nótese que se puede ver el código de uno de ellos

PM 18 Eliminar ficheros en un proyecto

Eliminar ficheros de un proyecto y comprobar su efecto en el explorador de proyectos.

a. Partiendo del resultado de la prueba anterior, se selecciona de nuevo *Erase Files from Project* y elige el mismo proyecto, abriéndose el menú de selección de ficheros (figura 5-74).

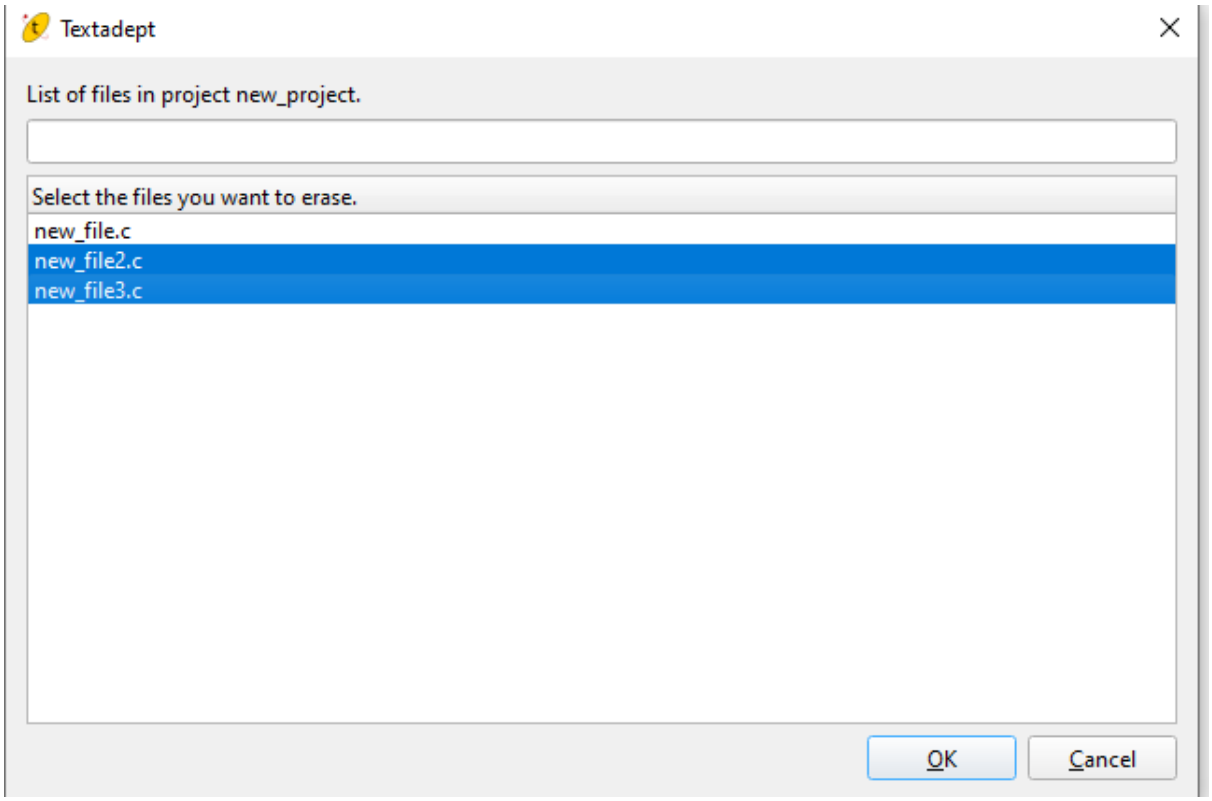


Figura 5-74. Se seleccionan dos ficheros para eliminar

b. Aparece un menú pidiendo confirmación del usuario (figura 5-75).

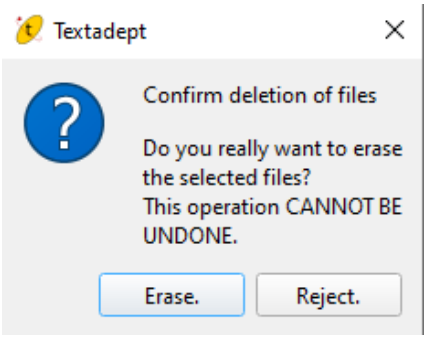


Figura 5-75. Mensaje de confirmación, pues es un paso que no se puede deshacer

c. Su aceptación lleva a un mensaje y a cerrar aquellos ficheros eliminados que estuvieran abiertos (figura 5-76).

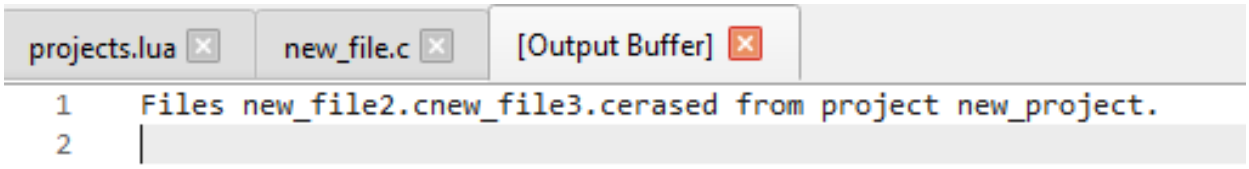


Figura 5-76. Mensaje de confirmación de la eliminación de los ficheros. Nótese que ya no está abierto new_file3.c

PM 19 Crear un proyecto en un espacio de trabajo con un nombre usado en otro

Cambiar de espacio de trabajo y empezar un proyecto cuyo nombre esté usado en el anterior y comprobar que no haya problemas.

a. La imagen muestra el explorador de proyectos de *default* (figura 5-77).

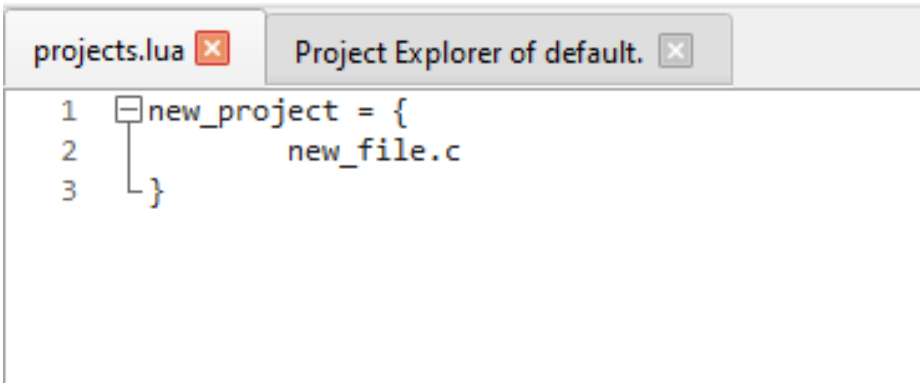


Figura 5-77. Explorador de proyectos del espacio de trabajo default

b. Se escoge el espacio de trabajo *new_WS* (figura 5-78).

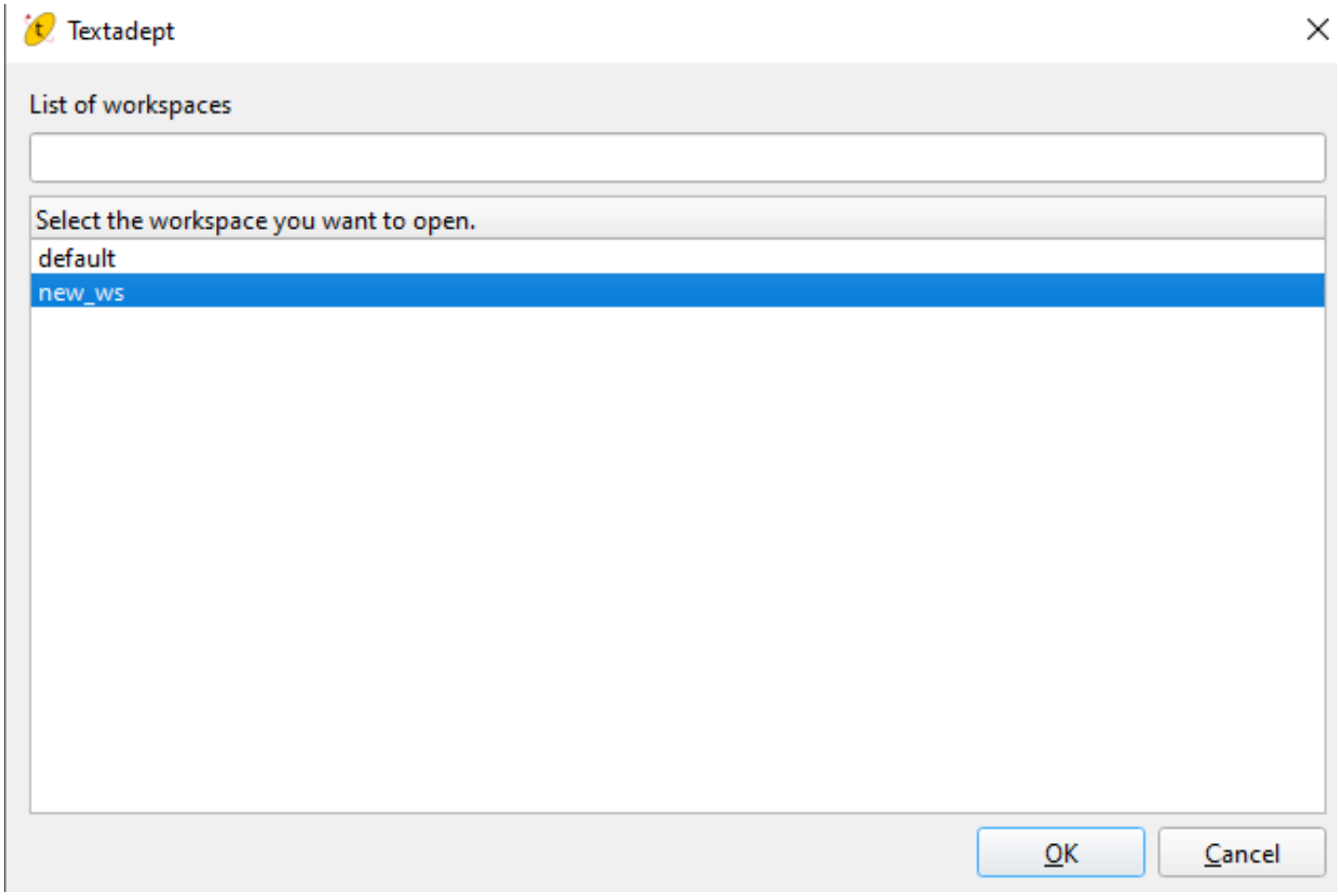


Figura 5-78. Se escoge el espacio de trabajo new_WS

c. Se abre el menú de *Start Project* para crear un proyecto también llamado *new_project* (figura 5-79).

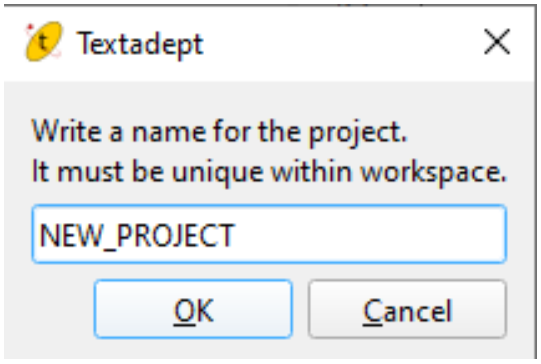


Figura 5-79. Se crea un proyecto llamado NEW_PROJECT

d. No hay ningún impedimento como se ve en el explorador de proyectos de *new_WS* (figura 5-80).

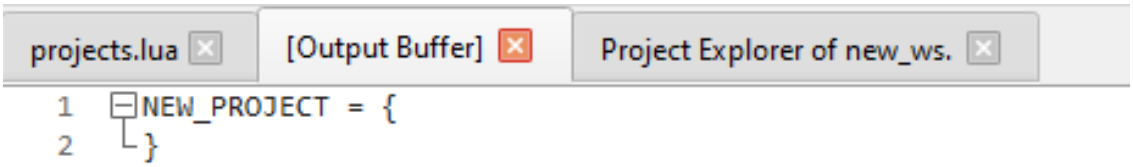


Figura 5-80. Explorador de proyectos del espacio de trabajo new_WS. Nótese que tiene un proyecto de igual nombre que en default

PM 20 Abrir el explorador de proyectos

El explorador de proyectos está cerrado. Se abre con la opción del menú.

a. El explorador de proyectos está inicialmente cerrado (figura 5-81).

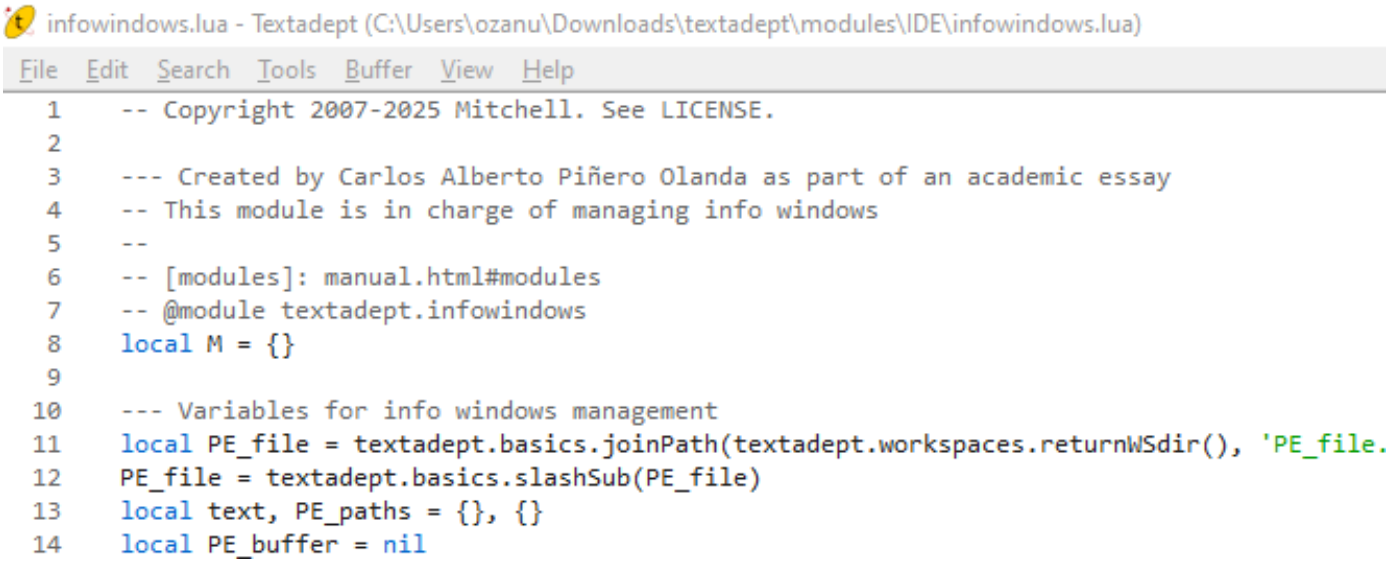


Figura 5-81. El explorador de proyectos está cerrado

b. Se selecciona *Toggle Project Explorer*, dotada de un atajo de teclado (figura 5-82).

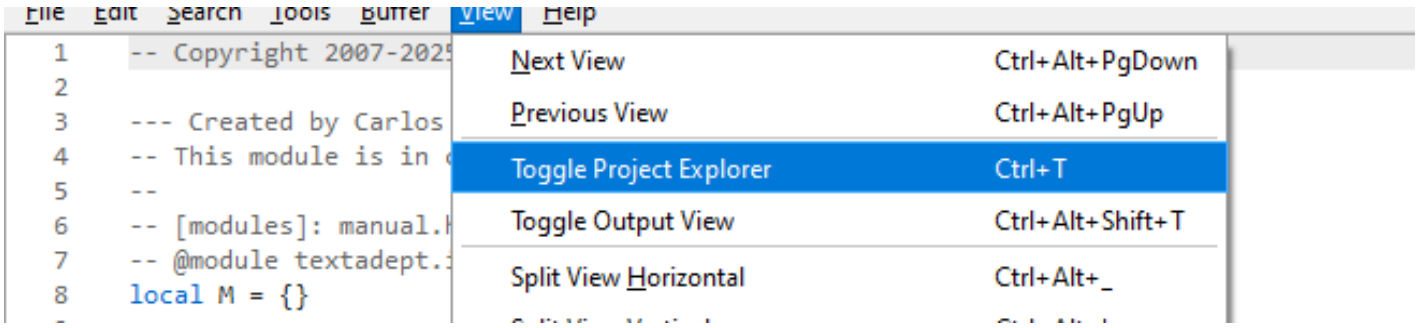


Figura 5-82. Opción Toggle Project Explorer

c. El explorador de proyectos está abierto (figura 5-83).

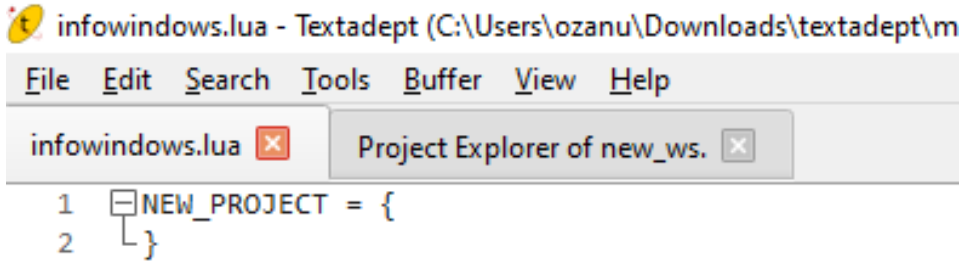


Figura 5-83. El explorador de proyectos está abierto

PM 21 Cerrar el explorador de proyectos

El explorador de proyectos está abierto. Se cierra con la opción del menú. Las figuras correspondientes son las de la prueba anterior, intercambiadas.

PM 22 El sistema recuerda que el sistema de explorador estaba abierto

Se cierra el IDE con el explorador de proyectos abierto. Al volver a inicializarlo, el explorador de proyectos sigue abierto con su título.

a. El explorador de proyectos está abierto (figura 5-84).

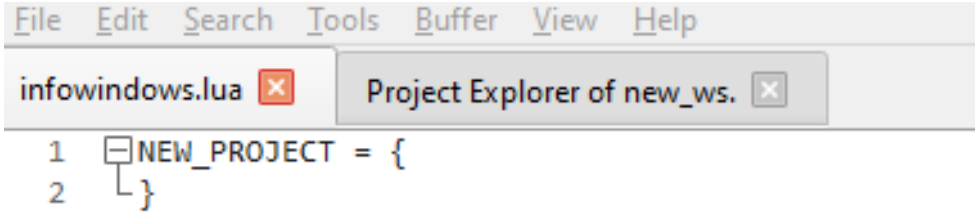


Figura 5-84. El explorador de proyectos está abierto

b. Se cierra el sistema (figura 5-85).

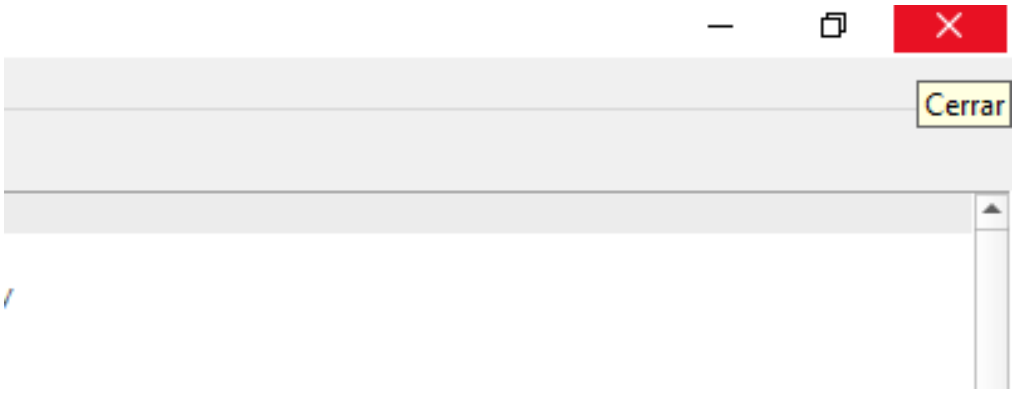


Figura 5-85. Se cierra el sistema

c. Al volver a abrirlo lo vuelve a mostrar como en la figura 5-84.

5.4 Consola de resultados

OC 1 Abrir la consola de resultados

La consola de resultados está cerrada. Se abre con la opción del menú.

a. La consola de resultados está inicialmente cerrada (figura 5-86).

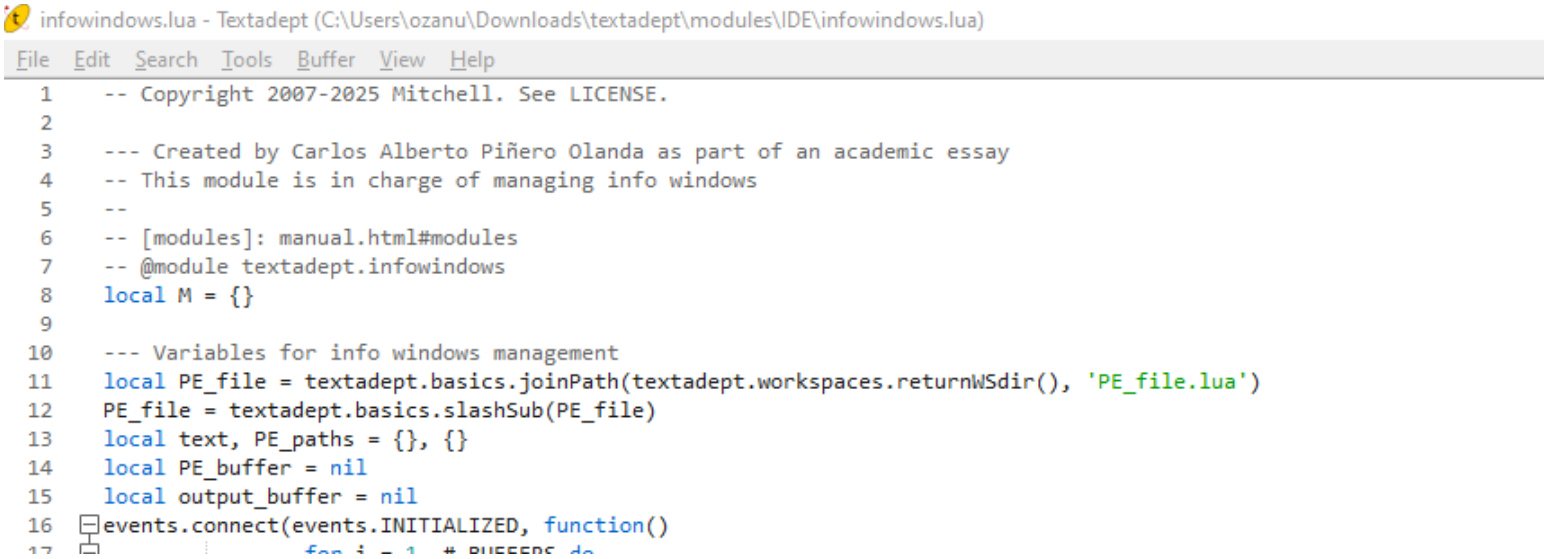


Figura 5-86. La consola de resultados está cerrada

b. Se selecciona *Toggle Output View*, dotada de un atajo de teclado (figura 5-87).

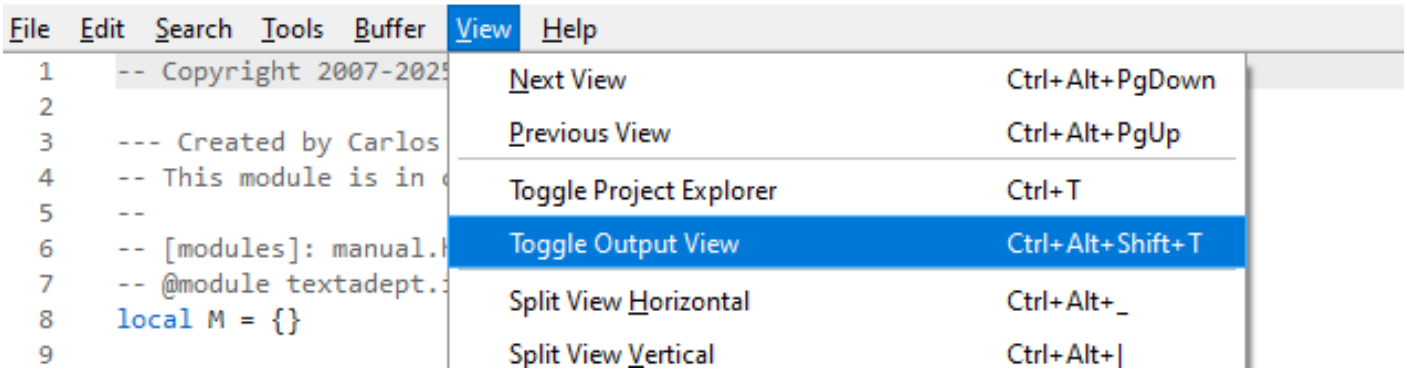


Figura 5-87. Opción Toggle Output View

c. La consola de resultados está abierta (figura 5-88).

infowindows.lua [Output Buffer]

```
1   -- Copyright 2007-2025 Mitchell. See LICENSE.
2
3   --- Created by Carlos Alberto Piñero Olanda as part of an academic essay
4   -- This module is in charge of managing info windows
5   --
6   -- [modules]: manual.html#modules
7   -- @module textadept.infowindows
8   local M = {}
9
10  --- Variables for info windows management
11  local PE_file = textadept.basics.joinPath(textadept.workspaces.returnWSdir(), 'PE_file
12  PE_file = textadept.basics.slashSub(PE_file)
13  local text, PE_paths = {}, {}
14  local PE_buffer = nil
15  local output_buffer = nil
16  events.connect(events.INITIALIZED, function()
17              for i = 1, #_BUFFERS do
18                  local buffer = _BUFFERS[i]
19                  local name = buffer.filename or buffer._type or _L['Untitled']
20                  if textadept.basics.slashSub(name) == PE_file then
21                      buffer.read_only = true
22                      buffer.tab_label = 'Project Explorer of ' .. textadept
23                      PE_buffer = buffer
24                  end
25                  if name == '[Output Buffer]' then
26                      output_buffer = buffer
27                  end
28                  if PE_buffer ~= nil and output_buffer ~= nil then break end
29              end
```

```
1   Output view. Results will be shown here. Beware that it can change its position.
2
```

Figura 5-88. La consola de resultados está abierta

OC 2 Cerrar la consola de resultados

La consola de resultados está abierta. Se cierra con la opción del menú. Las figuras correspondientes son las de la prueba anterior, intercambiadas.

OC 3 El sistema recuerda que la consola de resultados estaba abierta

La consola de resultados está abierta y el sistema se cierra. Al volver a inicializarlo, la consola de resultados sigue abierta con su título.

a. La consola de resultados está abierta (figura 5-89).

infowindows.lua | [Output Buffer]

```
1   -- Copyright 2007-2025 Mitchell. See LICENSE.
2
3   --- Created by Carlos Alberto Piñero Olanda as part of an academic essay
4   -- This module is in charge of managing info windows
5   --
6   -- [modules]: manual.html#modules
7   -- @module textadept.infowindows
8   local M = {}
9
10  --- Variables for info windows management
11  local PE_file = textadept.basics.joinPath(textadept.workspaces.returnWSdir(), 'PE_file.lua'
12  PE_file = textadept.basics.slashSub(PE_file)
13  local text, PE_paths = {}, {}
14  local PE_buffer = nil
15  local output_buffer = nil
16  events.connect(events.INITIALIZED, function()
17          for i = 1, #_BUFFERS do
18                  local buffer = _BUFFERS[i]
19                  local name = buffer.filename or buffer._type or _L['Untitled']
20                  if textadept.basics.slashSub(name) == PE_file then
21                          buffer.read_only = true
22                          buffer.tab_label = 'Project Explorer of ' .. textadept.work
23                          PE_buffer = buffer
24                  end
25                  if name == '[Output Buffer]' then
26                          output_buffer = buffer
27                  end
28                  if PE_buffer ~= nil and output_buffer ~= nil then break end
29          end
```

```
1   Output view. Results will be shown here. Beware that it can change its position.
2
```

Figura 5-89. La consola de resultados está abierta

b. Se cierra el sistema (figura 5-90).

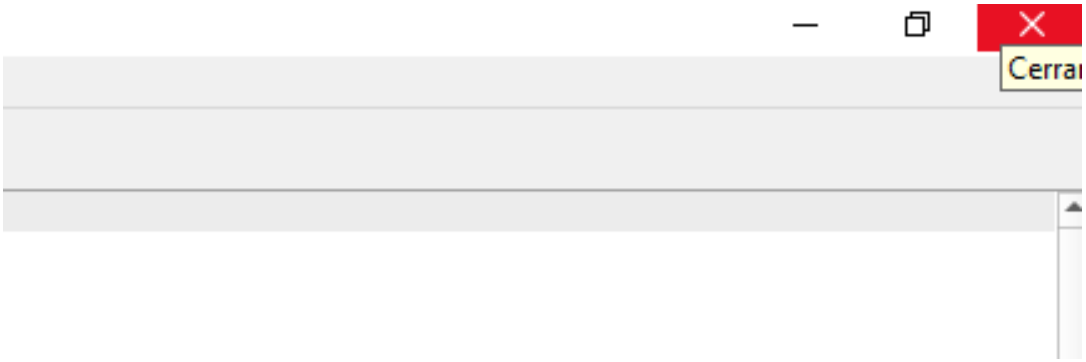


Figura 5-90. Se cierra el sistema

c. Volver a abrirlo lo vuelve a mostrar como en la primera imagen, la figura 5-89.

5.5 Compilación separada

SC 1 Compilación separada

Se ejecuta la compilación separada de un directorio de ficheros correctamente codificados en C.

a. Hay un fichero abierto (figura 5-91).

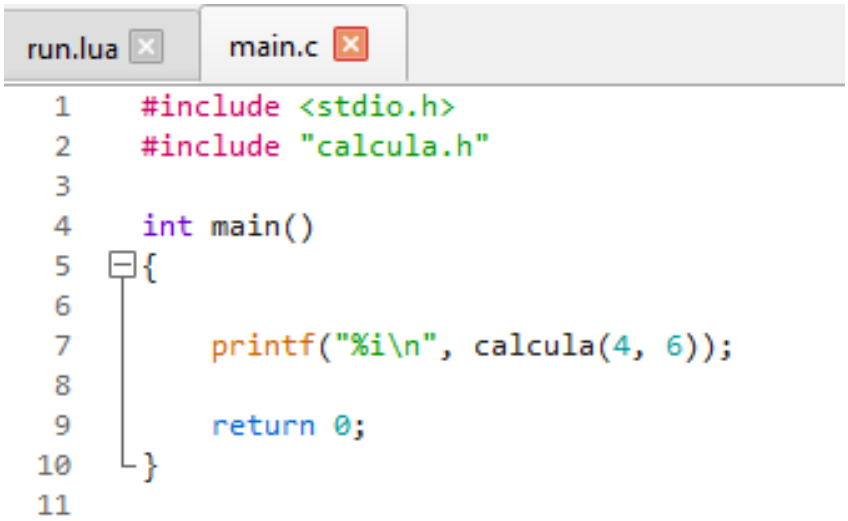


Figura 5-91. El fichero main.c está abierto. Nótese que invoca una cabecera en la segunda línea

b. Este fichero pertenece a un directorio con varios ficheros, todos codificados en C (figura 5-92).

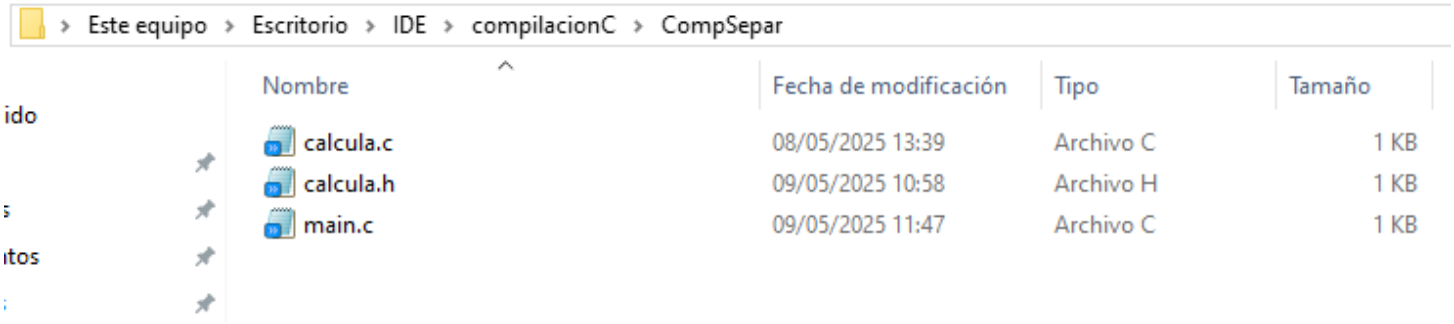


Figura 5-92. Contenido del directorio que aloja el anterior fichero

c. Se selecciona *Compile separately*, dotada de un atajo de teclado (figura 5-93).

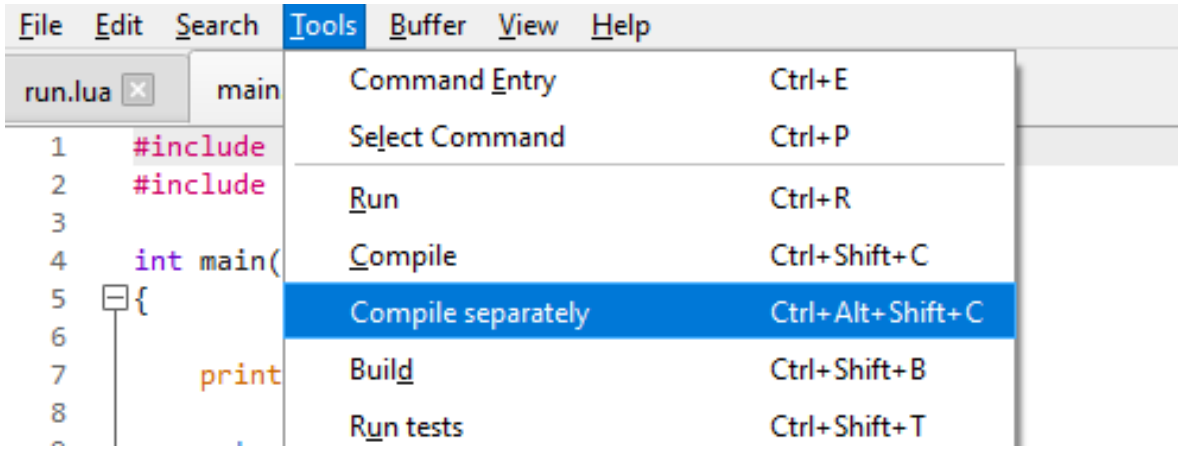


Figura 5-93. Opción Compile separately

d. Aparece un mensaje de confirmación (figura 5-94).

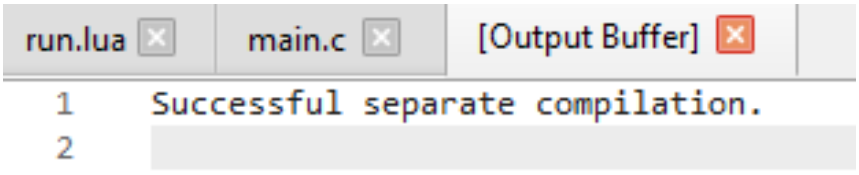


Figura 5-94. Mensaje de confirmación

e. El anterior directorio tiene ahora los ficheros objeto y el ejecutable (figura 5-95).

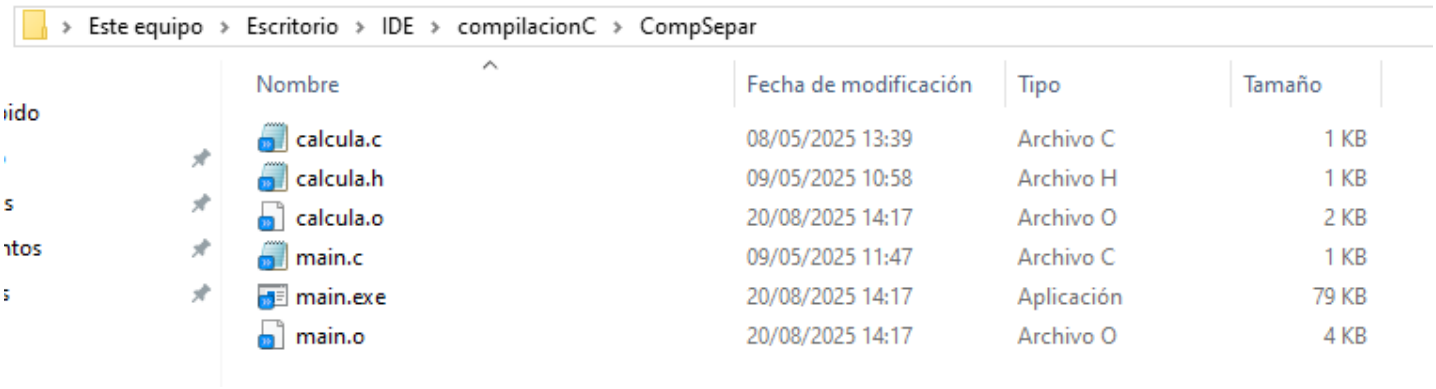


Figura 5-95. El directorio tiene ahora dos ficheros de extensión .o y un ejecutable

SC 2 Compilación separada de un sistema de ficheros con errores

Se intenta la compilación separada de un directorio de ficheros incorrectamente codificados en C, lo que arroja un mensaje de error.

a. Hay abierto un fichero en C incorrectamente codificado (figura 5-96).

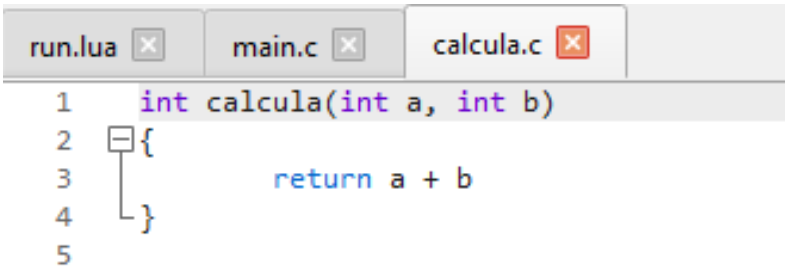


Figura 5-96. El fichero carece del punto y coma necesario en las declaraciones en C

b. Se vuelve a seleccionar *Compile separately*, pero como el fichero tiene un error, aparece un mensaje (figura 5-97).

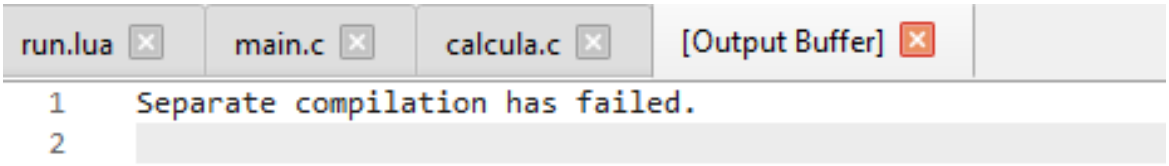


Figura 5-97. La compilación separada ha fallado

SC 3 Compilación separada de un sistema de ficheros en *Java*

Se ejecuta la compilación separada de un directorio de ficheros correctamente codificados en *Java*, cuyo resultado es idéntico a la compilación estándar en *Java*.

a. Hay un fichero abierto en *Java* (figura 5-98).

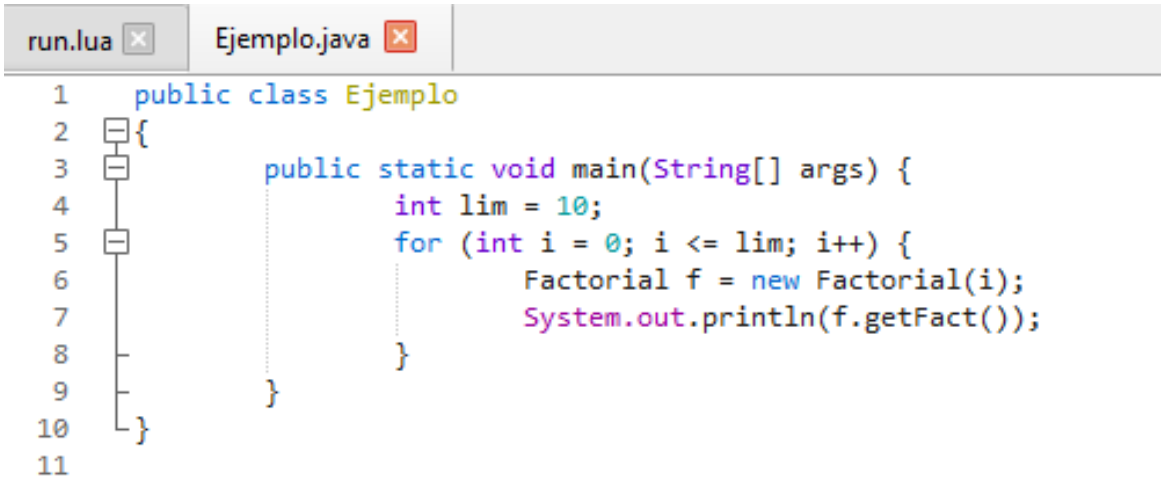


Figura 5-98. El fichero Ejemplo.java está abierto

b. Este fichero pertenece a un directorio con varios ficheros, todos codificados en *Java* (figura 5-99).

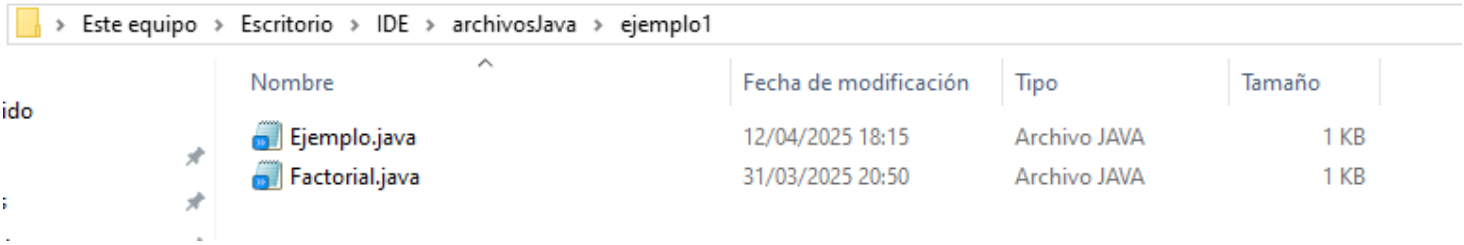


Figura 5-99. Contenido del directorio que aloja el anterior fichero

c. Seleccionar de nuevo *Compile separately* arroja un mensaje de confirmación (figura 5-100).

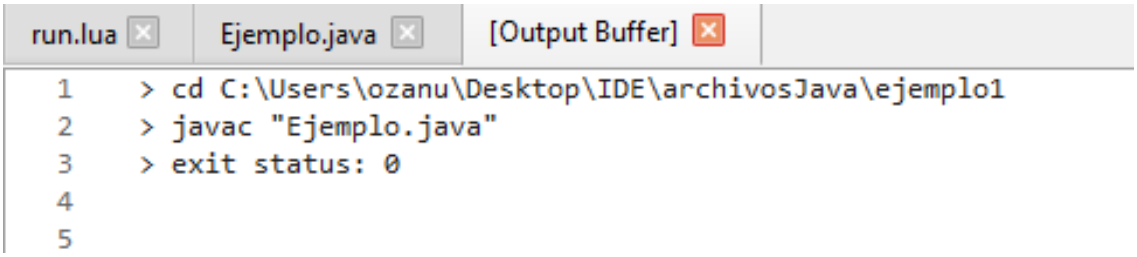


Figura 5-100. Mensaje de confirmación, semejante al de la compuilación estándar

d. El directorio tiene ahora los ficheros que se obtienen compilando en *Java* (figura 5-101).

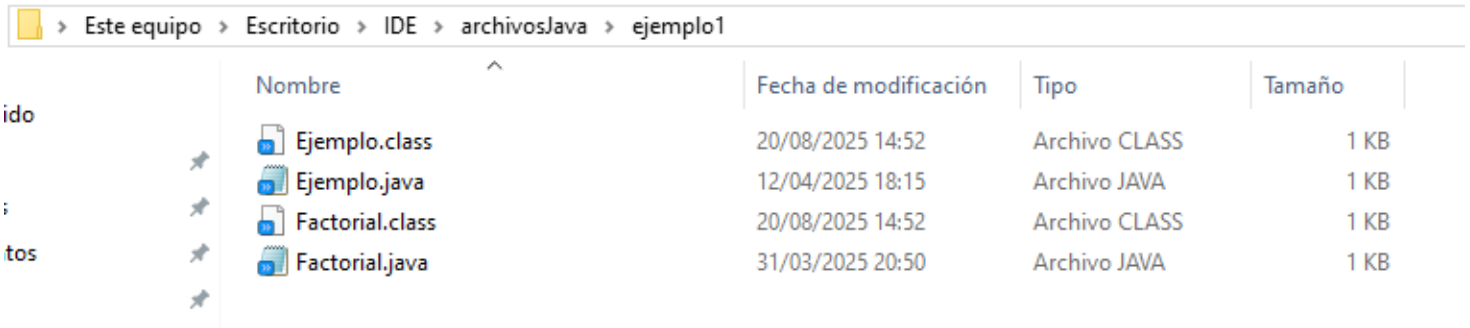


Figura 5-101. El directorio tiene ahora dos ficheros de extensión .class

5.6 Gestión de plantillas

TM 1 Creación de la estructura de datos para las plantillas

El directorio de plantillas no existe, pero se crea al inicializar el IDE.

a. Inicialmente el directorio principal del sistema tiene la misma estructura de datos que tiene *TextAdept* (figura 5-102).

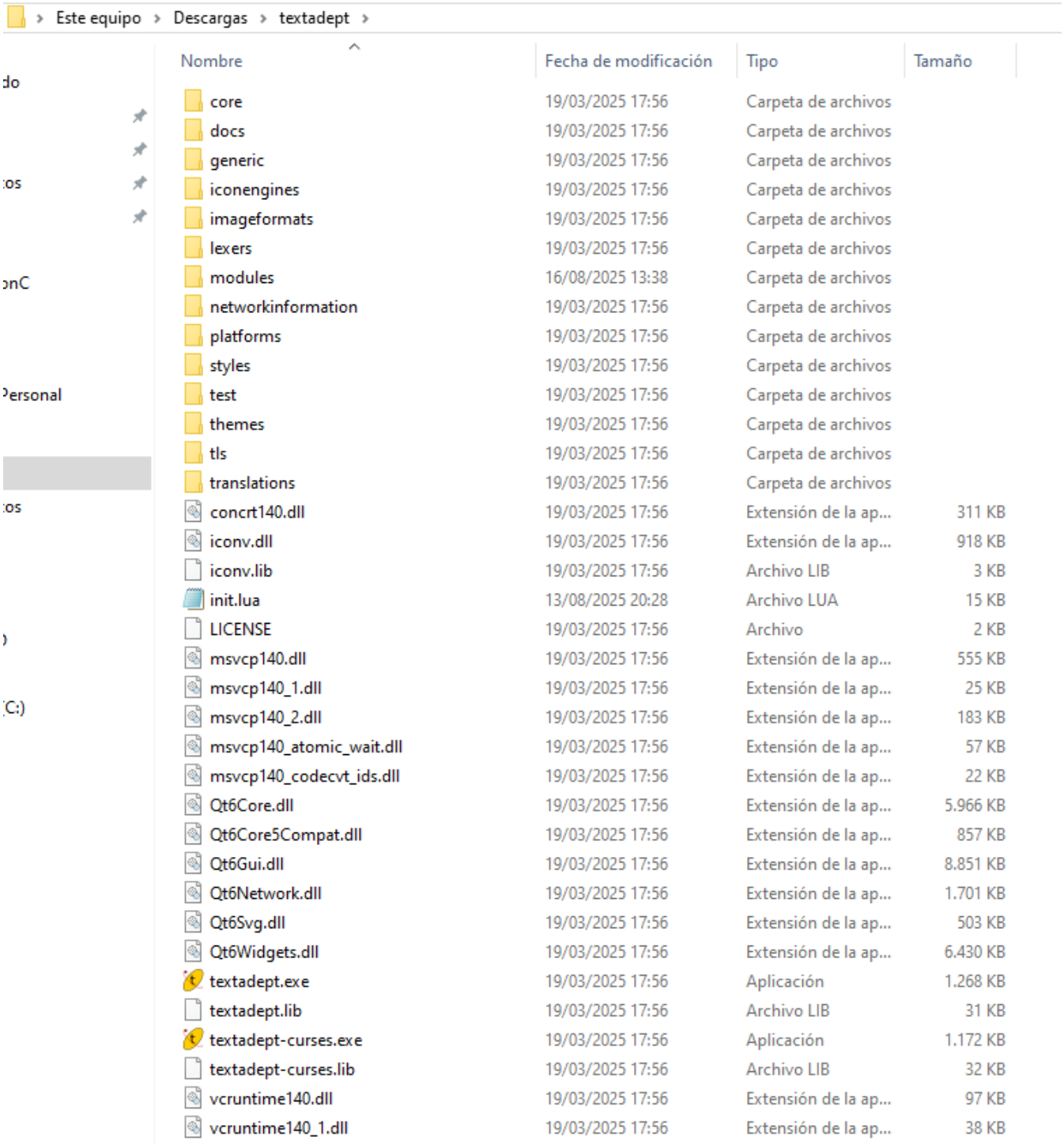


Figura 5-102. Arquitectura de ficheros inicial del sistema

b. Iniciar el sistema conlleva la creación en el mismo de un subdirectorio llamado *templates* (figura 5-103).

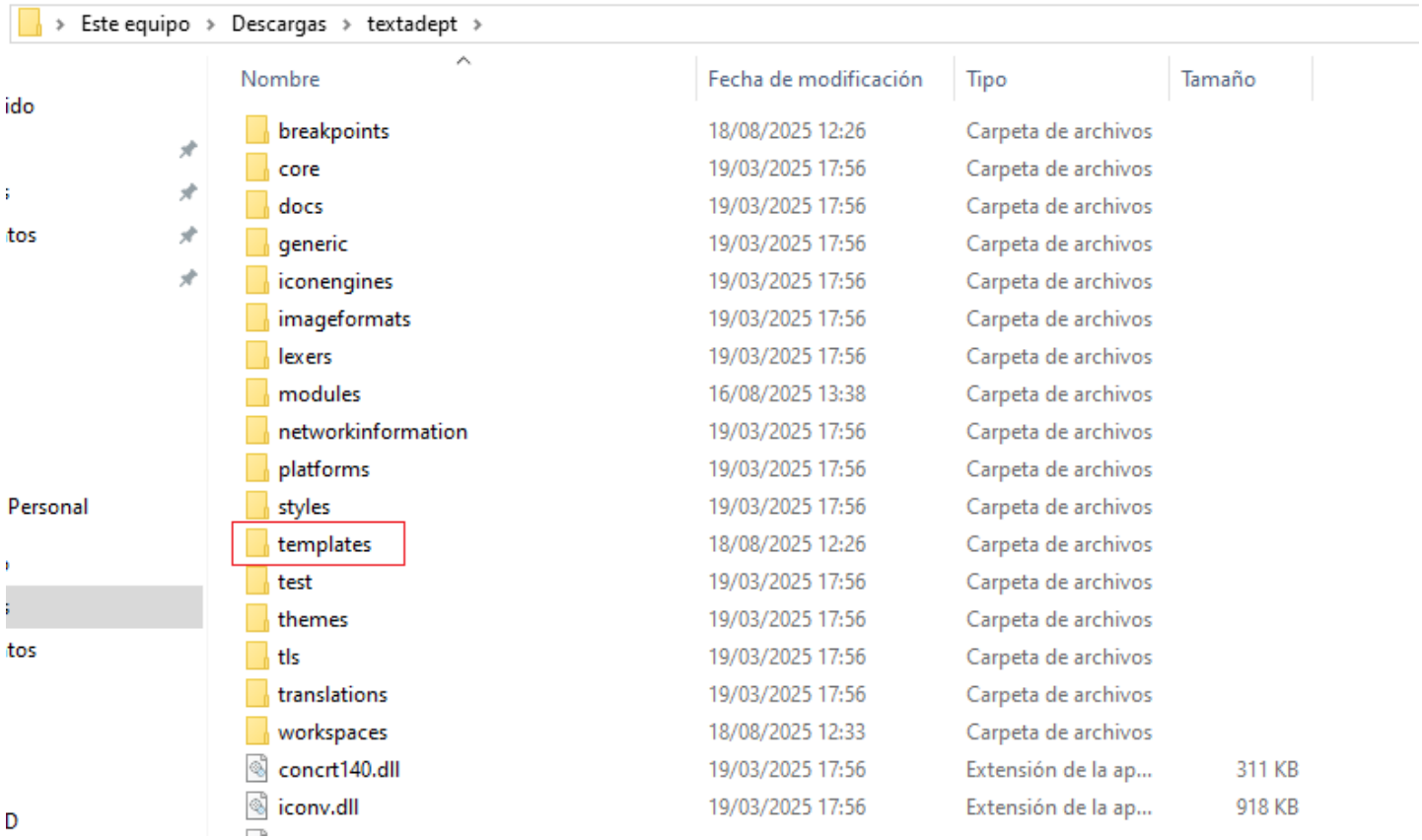


Figura 5-103. Aparición del directorio templates después de inicializar el sistema

TM 2 Imposibilidad de iniciar plantillas si no hay

Intentar abrir plantillas cuando no hay ninguna, lo que arroja un mensaje de aviso.

a. Se selecciona en el menú la opción Open Templates, dotada de un atajo de teclado (figura 5-104).

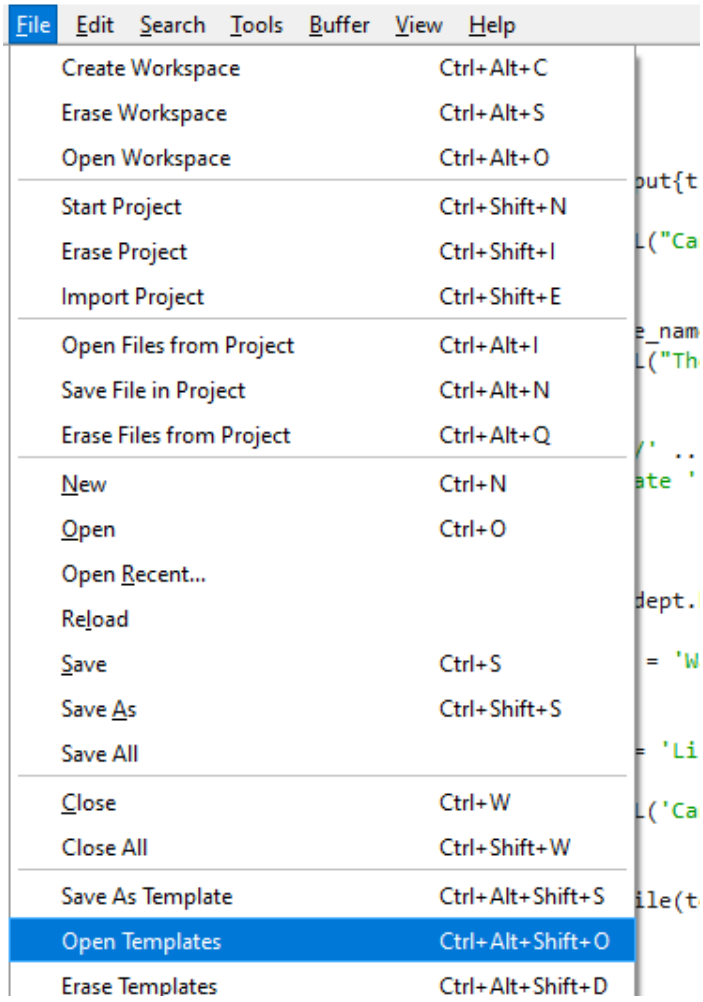


Figura 5-104. Opción Open Templates

b. Aparece un mensaje de aviso (figura 5-105).

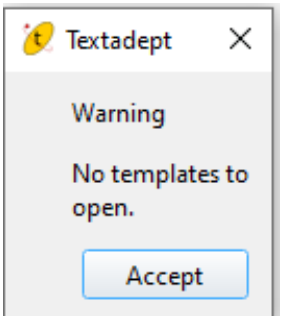


Figura 5-105. Mensaje de aviso porque no hay plantillas

TM 3 Imposibilidad de eliminar plantillas si no hay

Intentar eliminar plantillas cuando no hay ninguna, lo que arroja un mensaje de aviso.

a. Se selecciona en el menú la opción *Erase Templates*, dotada de un atajo de teclado (figura 5-106).

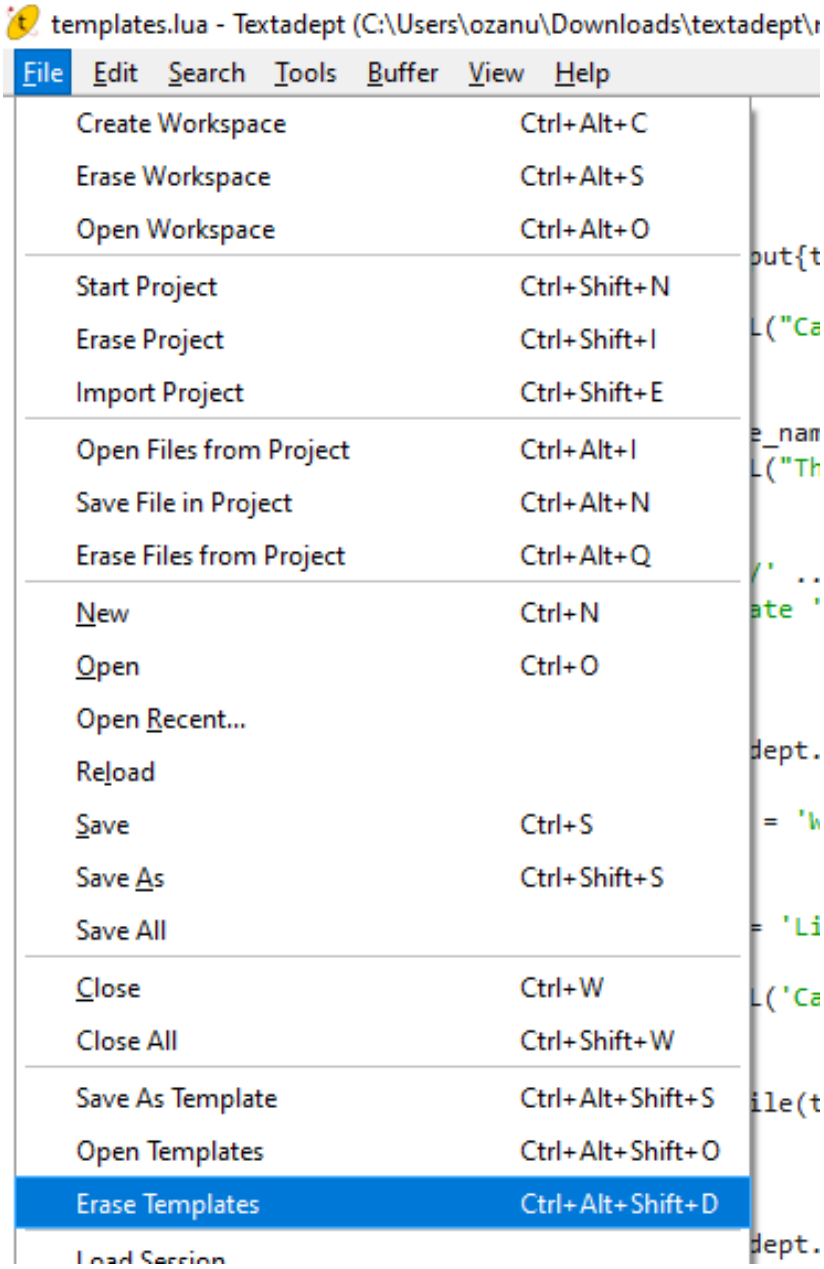


Figura 5-106. Opción Erase Templates

b. Aparece un mensaje de aviso (figura 5-107).

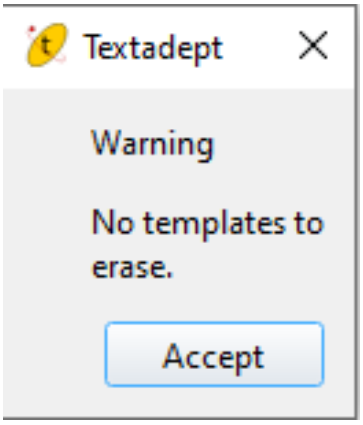


Figura 5-107. Mensaje de aviso porque no hay plantillas

TM 4 Guardar una plantilla

Se guarda un fichero como plantilla.

a. Hay un fichero abierto (figura 5-108).

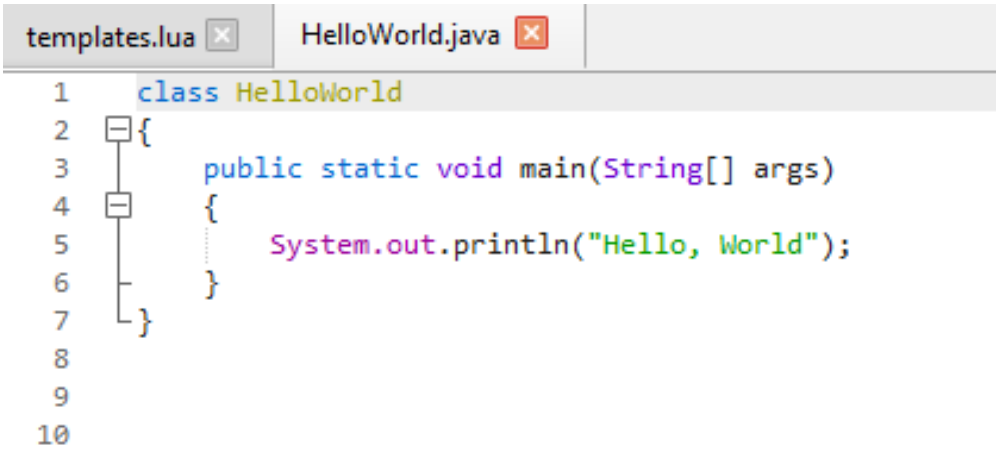


Figura 5-108. Fichero HelloWorld.java abierto

b. Seleccionar *Save As Template*, dotada de un atajo de teclado (figura 5-109).

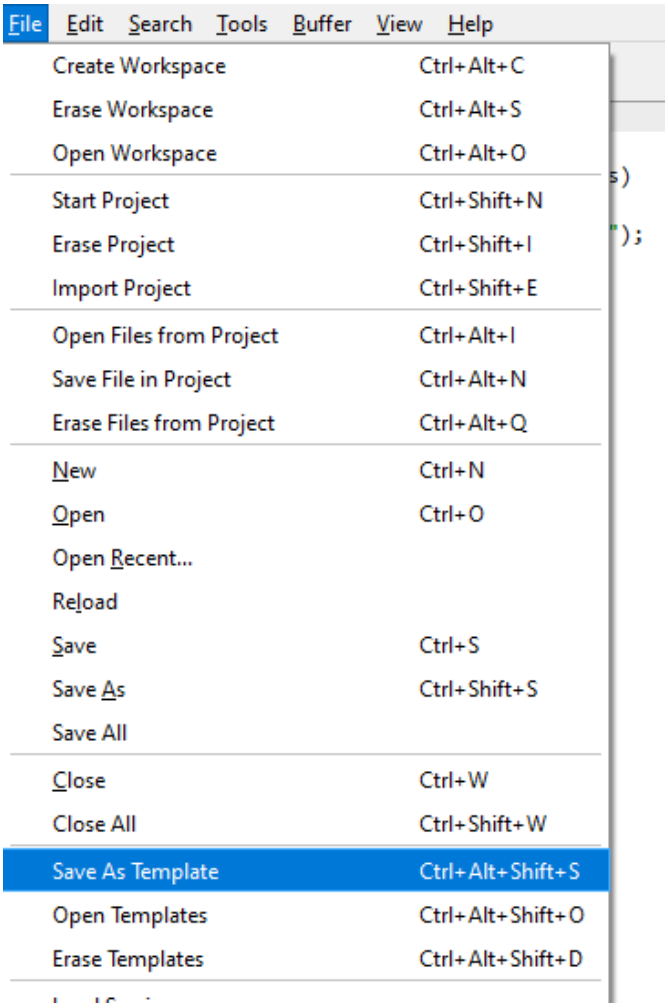


Figura 5-109. Opción Save As Template

c. Abre un menú pidiendo un nuevo nombre (figura 5-110).

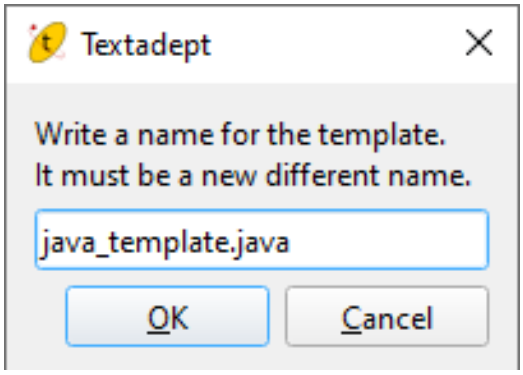


Figura 5-110. Diálogo para escribir el nombre de la plantilla

d. Escribir un nombre abre un mensaje de confirmación (figura 5-111).

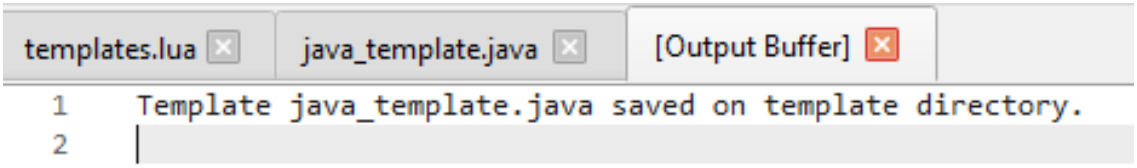


Figura 5-111. Mensaje de confirmación de que se ha guardado la plantilla

e. Ha cambiado el nombre del fichero (figura 5-112).

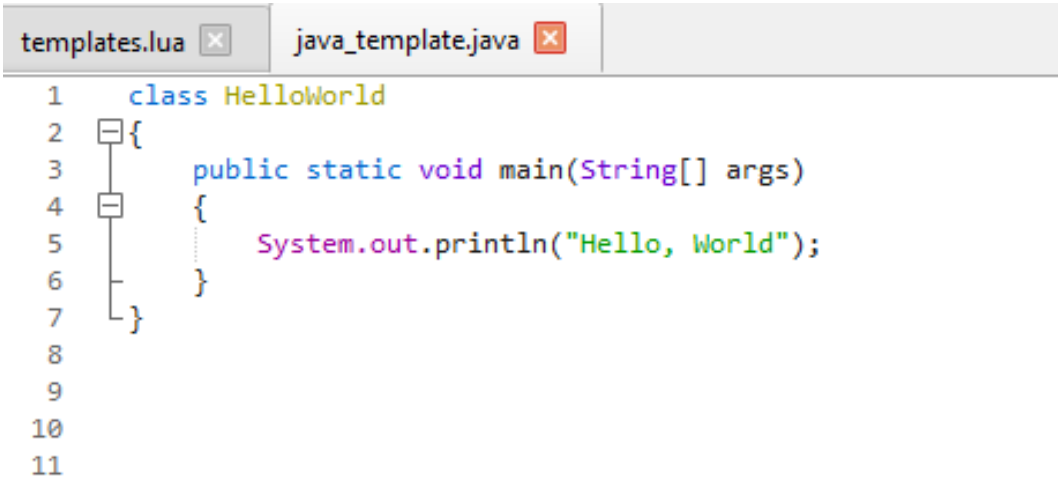


Figura 5-112. La plantilla muestra su nombre

TM 5 Unicidad de los nombres de las plantillas

Se intenta guardar un fichero como plantilla con un nombre ya existente, lo que arroja un mensaje de aviso.

a. La imagen muestra otro fichero abierto (figura 5-113).

templates.lua | Factorial.java

```
1   public class Factorial
2   {
3           private int fact;
4
5           public Factorial(int n) {
6                   this.fact = 1;
7                   for (int i = 1; i <= n; i++) {this.fact *= i;}
8           }
9
10          public int getFact() {return this.fact;}
11  }
```

Figura 5-113. Fichero Factorial.java abierto

b. Seleccionar de nuevo *Save As Template* y escribir de nuevo el mismo nombre (figura 5-114).

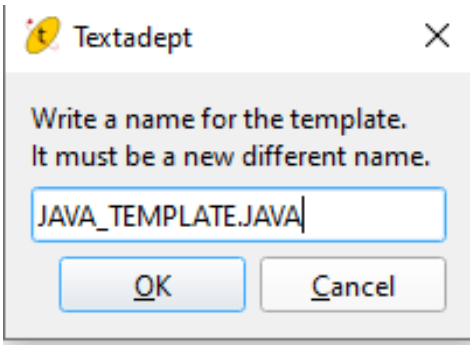


Figura 5-114. Se vuelve a escribir el mismo nombre

c. Aparece un mensaje de aviso (figura 5-115).

templates.lua | Factorial.java | [Output Buffer]

```
1   There is already a template with that name.
2
```

Figura 5-115. Mensaje de aviso por escribir un nombre ya existente

TM 6 Abrir plantillas

Se abren algunas de las anteriores plantillas.

a. Inicialmente no hay plantillas abiertas (figura 5-116).

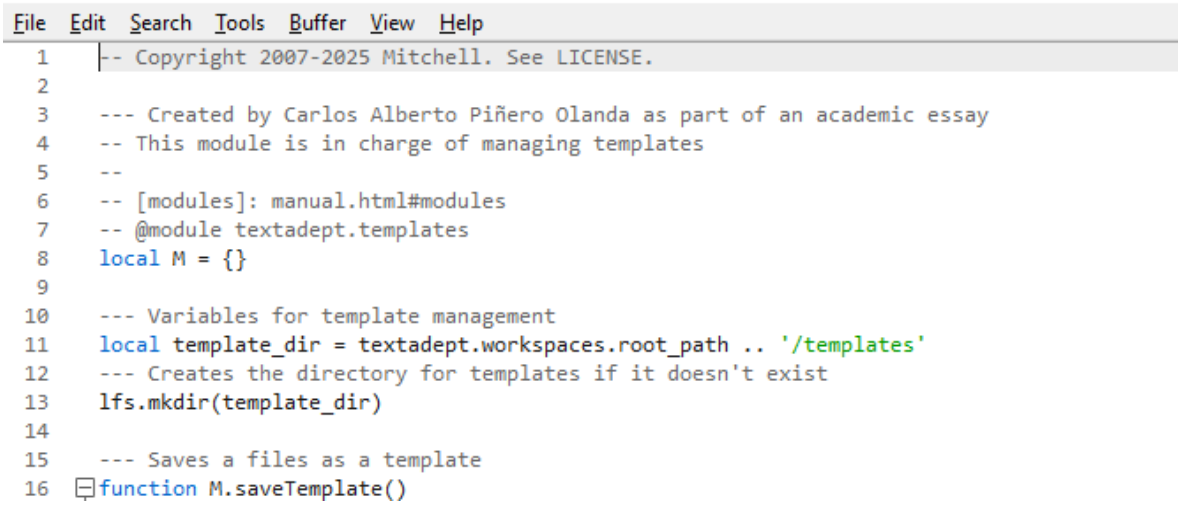


Figura 5-116. No hay plantillas abiertas inicialmente

b. Seleccionar de nuevo *Open Templates* abre un menú en que se pueden seleccionar varias plantillas (figura 5-117).

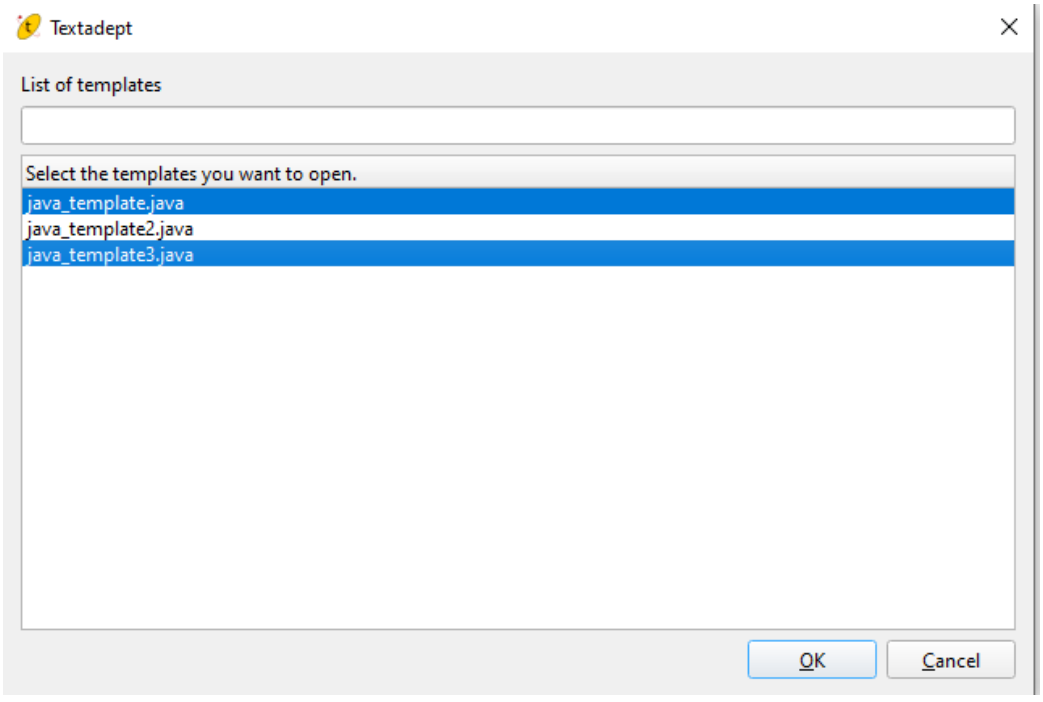


Figura 5-117. Menú para seleccionar plantillas. Nótese que se han seleccionado dos

c. Las platillas están abiertas (figura 5-118).

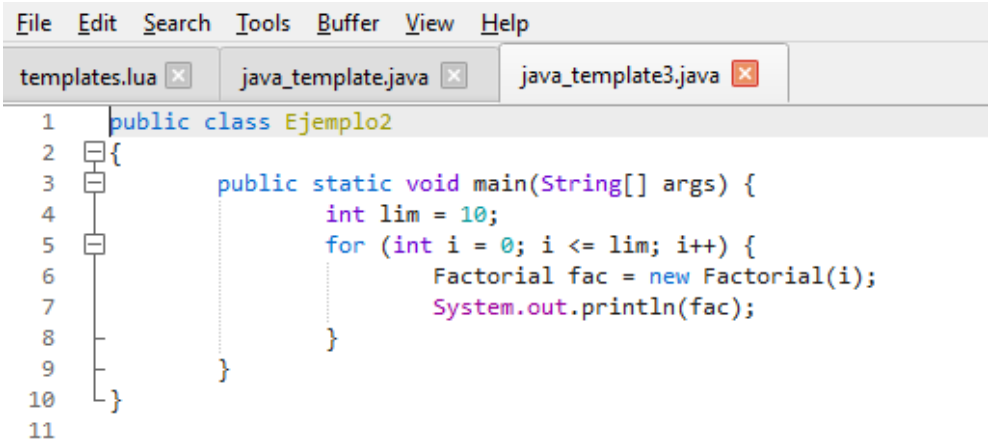


Figura 5-118. Las dos plantillas elegidas están abiertas

TM 7 Eliminar plantillas

Se eliminan algunas de las plantillas, por lo que al abrir el menú de selección ya no existen.

a. Partiendo del resultado de la prueba anterior, se selecciona de nuevo *Erase Templates*, lo que abre un menú en que se pueden seleccionar varias (figura 5-119).

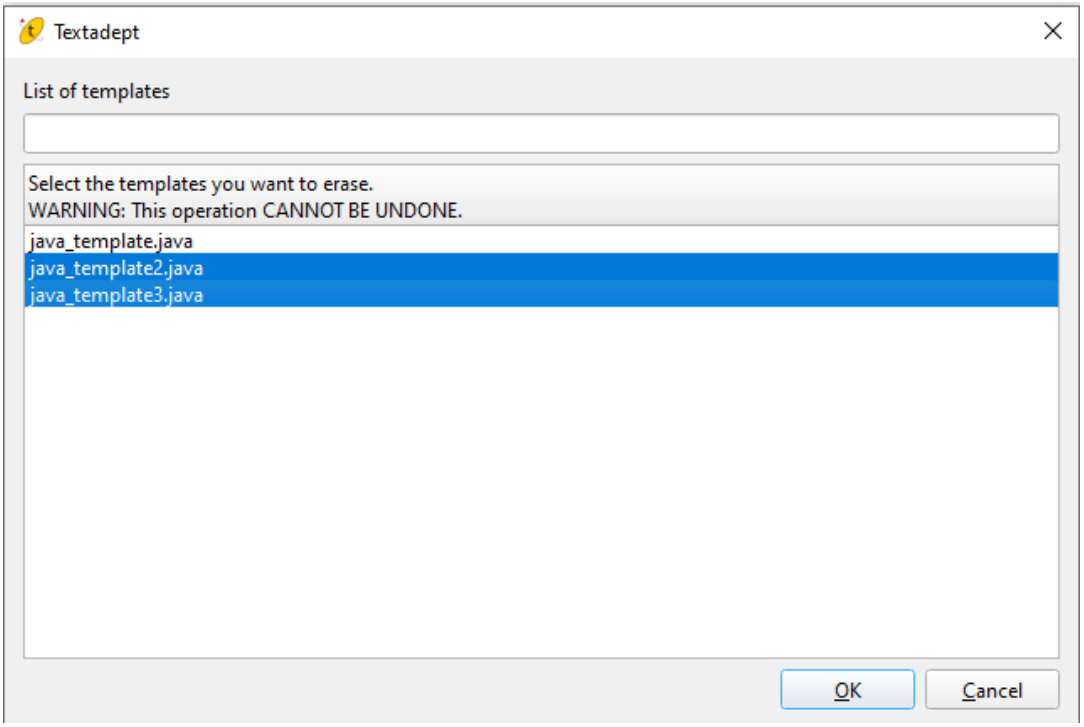


Figura 5-119. Se seleccionan dos plantillas

b. Su aceptación lleva a un mensaje y a cerrar aquellas plantillas eliminadas que estuvieran abiertas (figura 5-120).

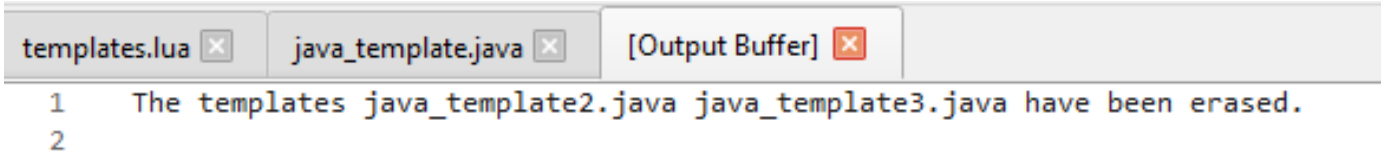


Figura 5-120. Mensaje de confirmación y, además, ya no está abierta java_template3.java

5.7 Ejecución paso a paso

RSBS 1 Creación de la estructura de datos para la ejecución paso a paso (1)

Cuando el sistema se inicia, se crea un directorio específico para los puntos de ruptura si no lo había.

a. Inicialmente el directorio principal del sistema tiene la misma estructura de datos que tiene *TextAdept* (figura 5-121).

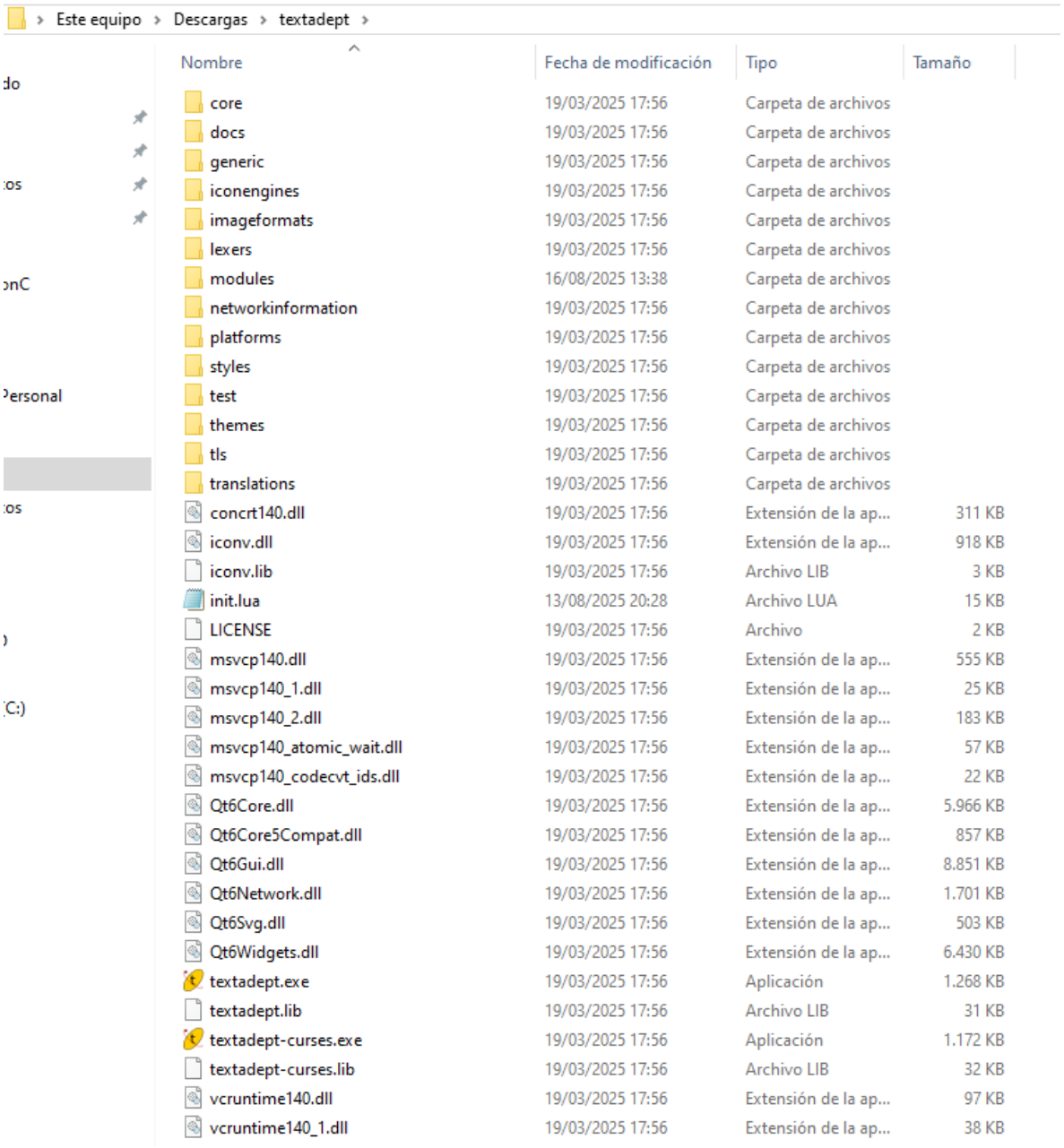


Figura 5-121. Arquitectura de ficheros inicial del sistema

b. Iniciar el sistema conlleva la creación en el mismo de un subdirectorio llamado *breakpoints* (figura 5-122).

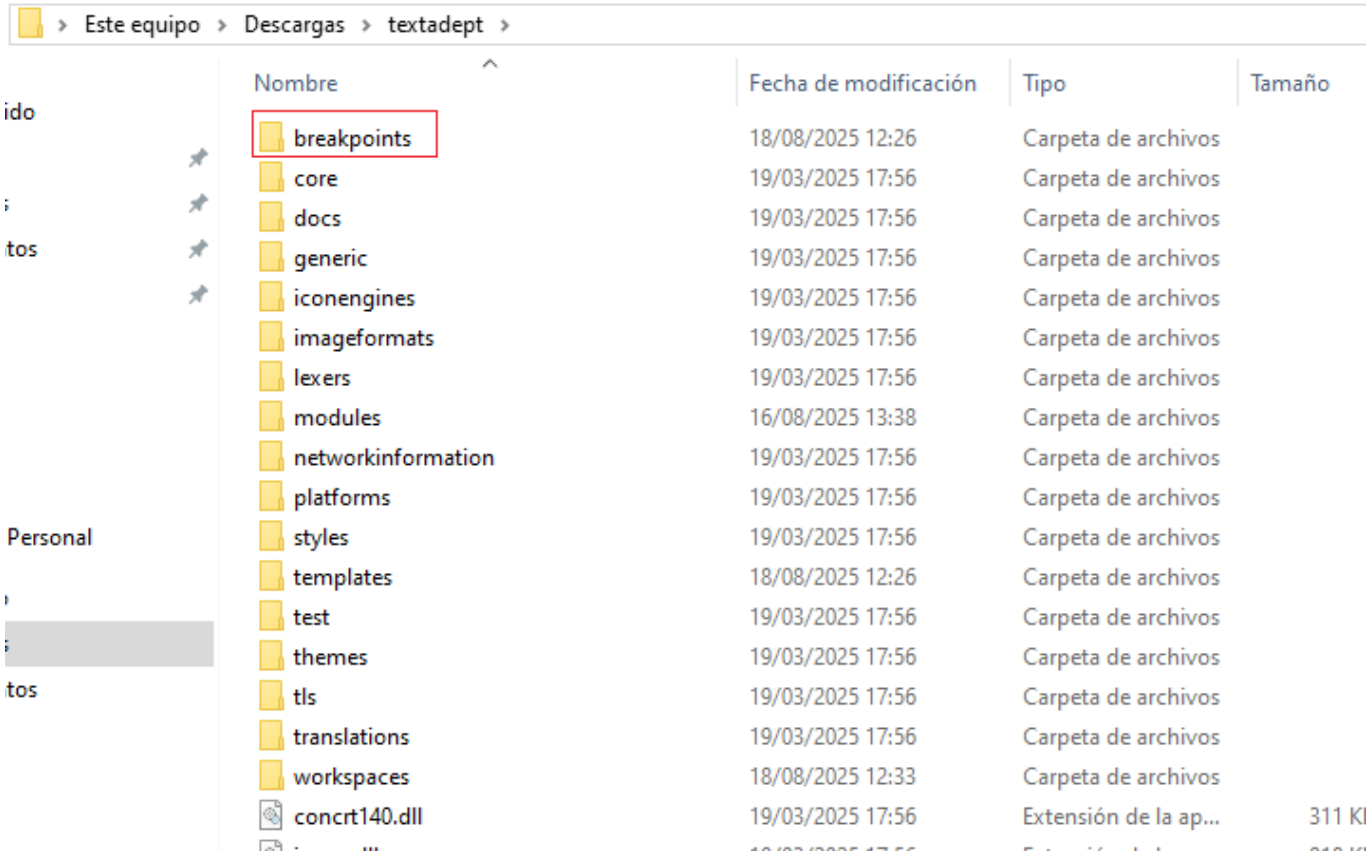


Figura 5-122. Aparición del directorio breakpoints después de inicializar el sistema

RSBS 2 Creación de la estructura de datos para la ejecución paso a paso (2)

El contenido del directorio específico para los puntos de ruptura crea sus elementos cuando el sistema se inicializa si no existían.

a. Inicialmente el subdirectorio *breakpoints* está vacío (figura 5-123).

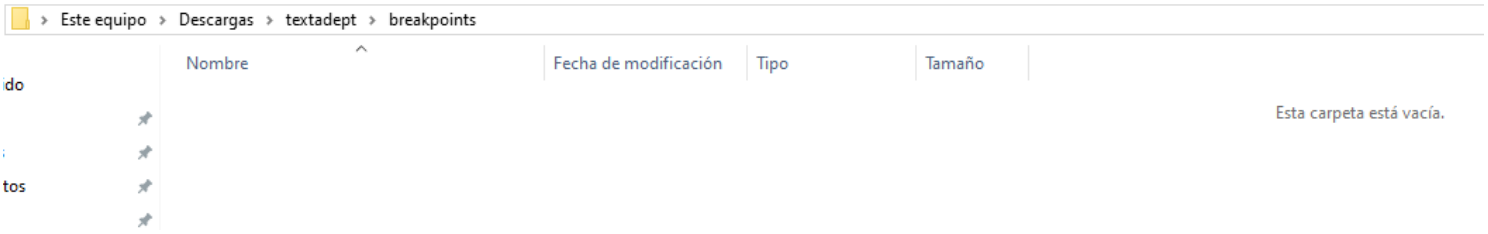


Figura 5-123. Directorio breakpoints vacío

b. Iniciar el sistema conlleva que aparezcan dos ficheros, *list* y *counter*, y dos directorios, *stepFiles* y *moddedFiles* (figura 5-124).

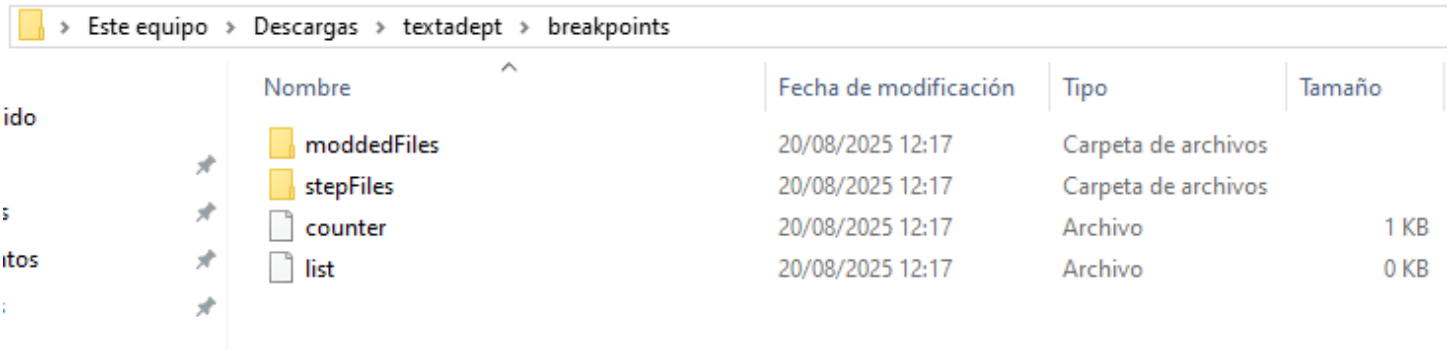


Figura 5-124. Directorio breakpoints con la estructura de datos necesaria para la gestión de los puntos de ruptura

RSBS 3 Marcar un punto de ruptura

Se pulsa con el ratón en el margen inmediatamente a la derecha de los números en la línea de un fichero sin punto de ruptura. La correspondiente línea muestra entonces un marcador.

a. Hay un fichero abierto sin marcas (figura 5-125).

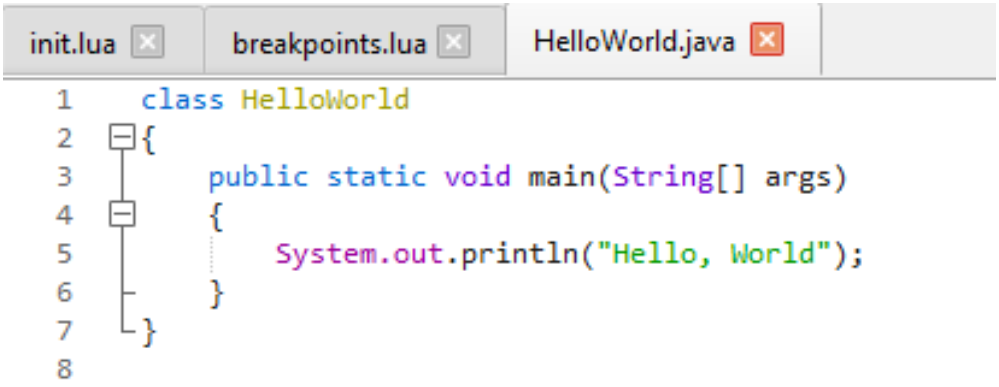


Figura 5-125. Fichero HelloWorld.java abierto

b. Pulsar con el botón izquierdo en la línea 5 hace que aparezca un punto de ruptura en esa línea (figura 5-126).

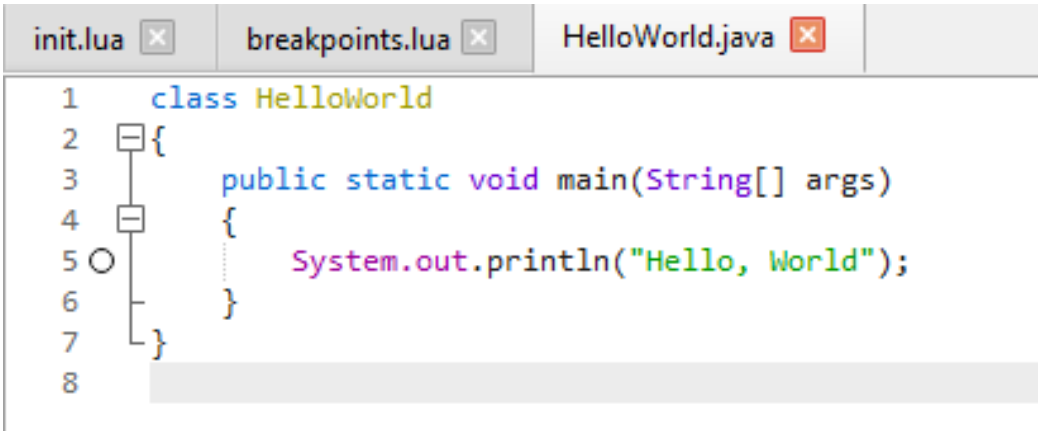


Figura 5-126. El mismo fichero HelloWorld.java presenta una marca en forma de círculo blanco en la línea 5

RSBS 4 Desmarcar un punto de ruptura

Se pulsa con el ratón en el margen inmediatamente a la derecha de los números en la línea de un fichero con punto de ruptura. La correspondiente línea pierde el marcador que tenía. Las figuras son las mismas de la prueba anterior, pero intercambiadas.

RSBS 5 Gestionar punto de ruptura con el menú

Se pueden marcar y desmarcar puntos de ruptura con el menú. En concreto, se selecciona *Toggle breakpoint*, dotada de un atajo de teclado (figura 5-127).

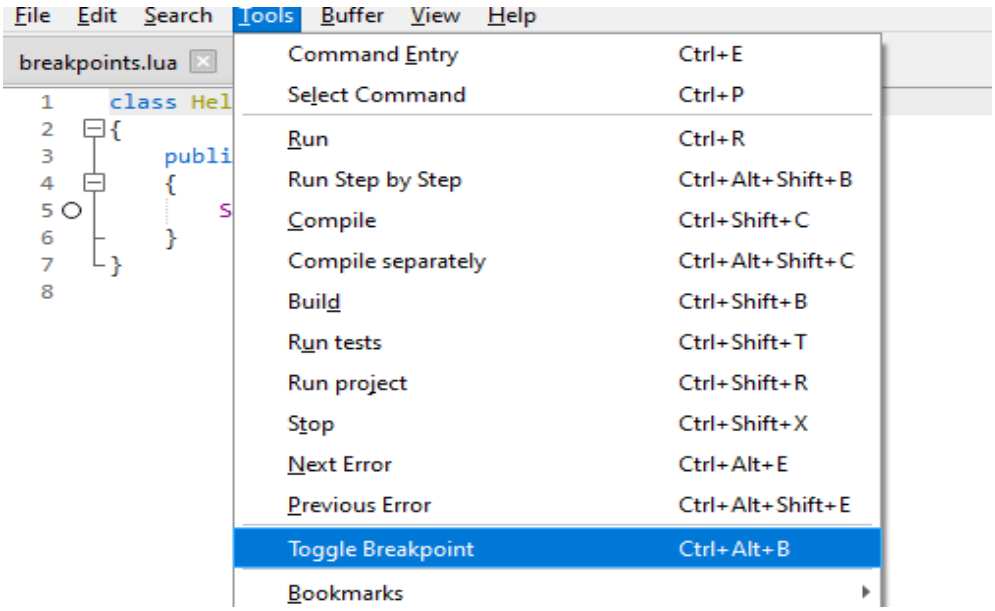


Figura 5-127. Opción Toggle breakpoint

RSBS 6 Gestionar punto de ruptura con el menú

Se guarda un archivo con marcadores. Se cierra el sistema y se vuelve a abrir. El fichero tiene exactamente los mismos marcadores. Las figuras son las mismas de pruebas anteriores y se cierra el sistema como en otros grupos de pruebas, clicando en el icono de cerrar en la esquina superior derecha.

RSBS 7 Los puntos de ruptura se almacenan, aunque se cierre el fichero

Se cierra un fichero que fue previamente guardado. Al abrirlo, tiene los mismos marcadores. La opción de cerrar se realiza con la opción *Close*, dotada de un atajo de teclado (figura 5-128).

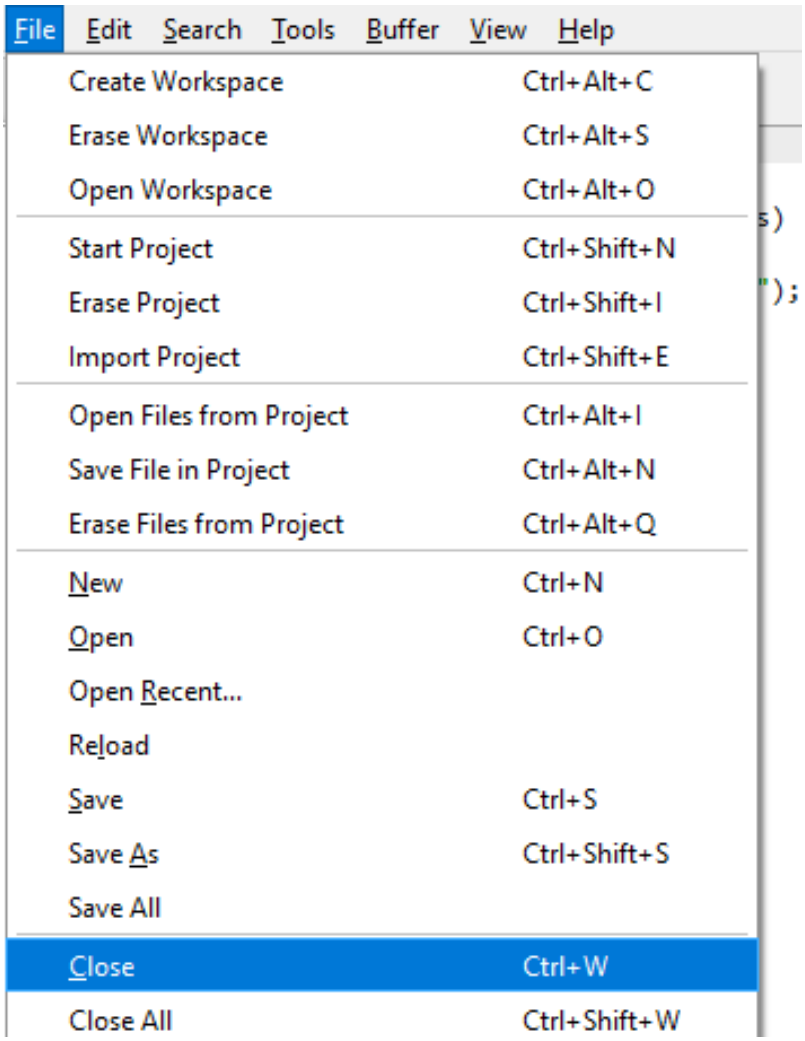


Figura 5-128. Opción Close

RSBS 8 Ejecución paso a paso de un fichero con marcadores

Se realiza la ejecución paso a paso de un fichero.

a. Hay un fichero en *Java* con un punto de ruptura (figura 5-129).

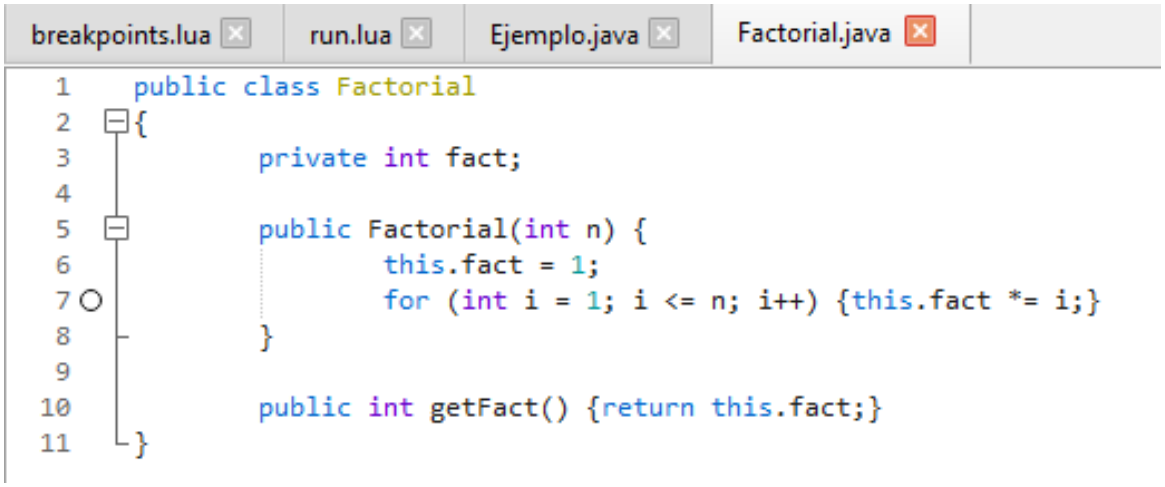


Figura 5-129. Fichero Factorial.java, con un punto de ruptura

b. Se sitúa el cursor activo sobre el fichero con el método *main()* (figura 5-130).

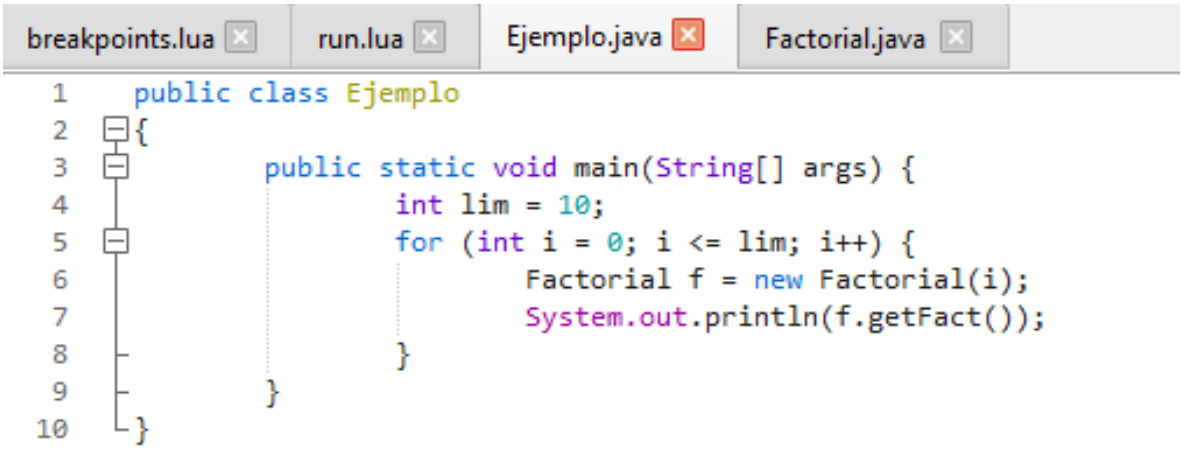


Figura 5-130. Cursor activo sobre el fichero con el método main(), Ejemplo.java

c. Se selecciona *Run Step by Step*, dotada de un atajo de teclado (figura 5-131).

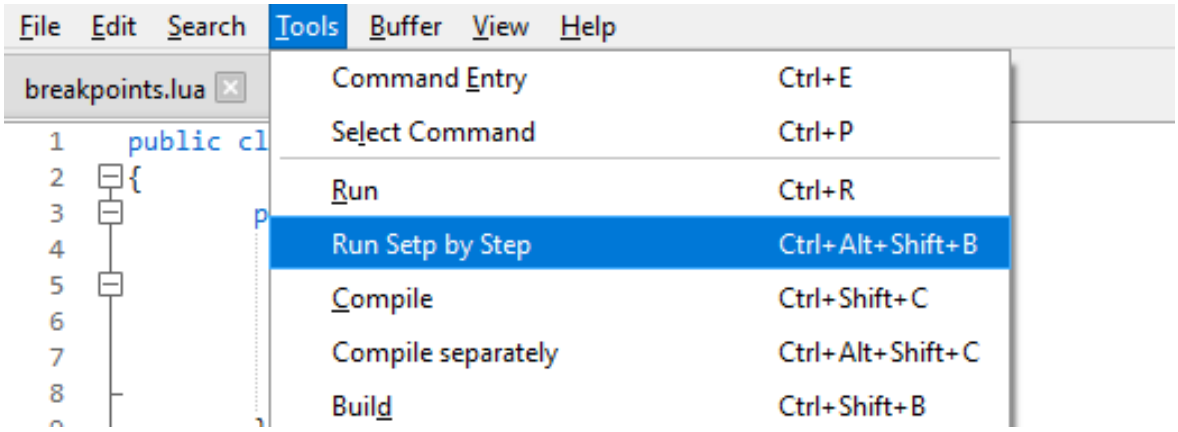


Figura 5-131. Opción Run Step by Step

d. La ejecución del programa se realiza con puntos de ruptura (figura 5-132).

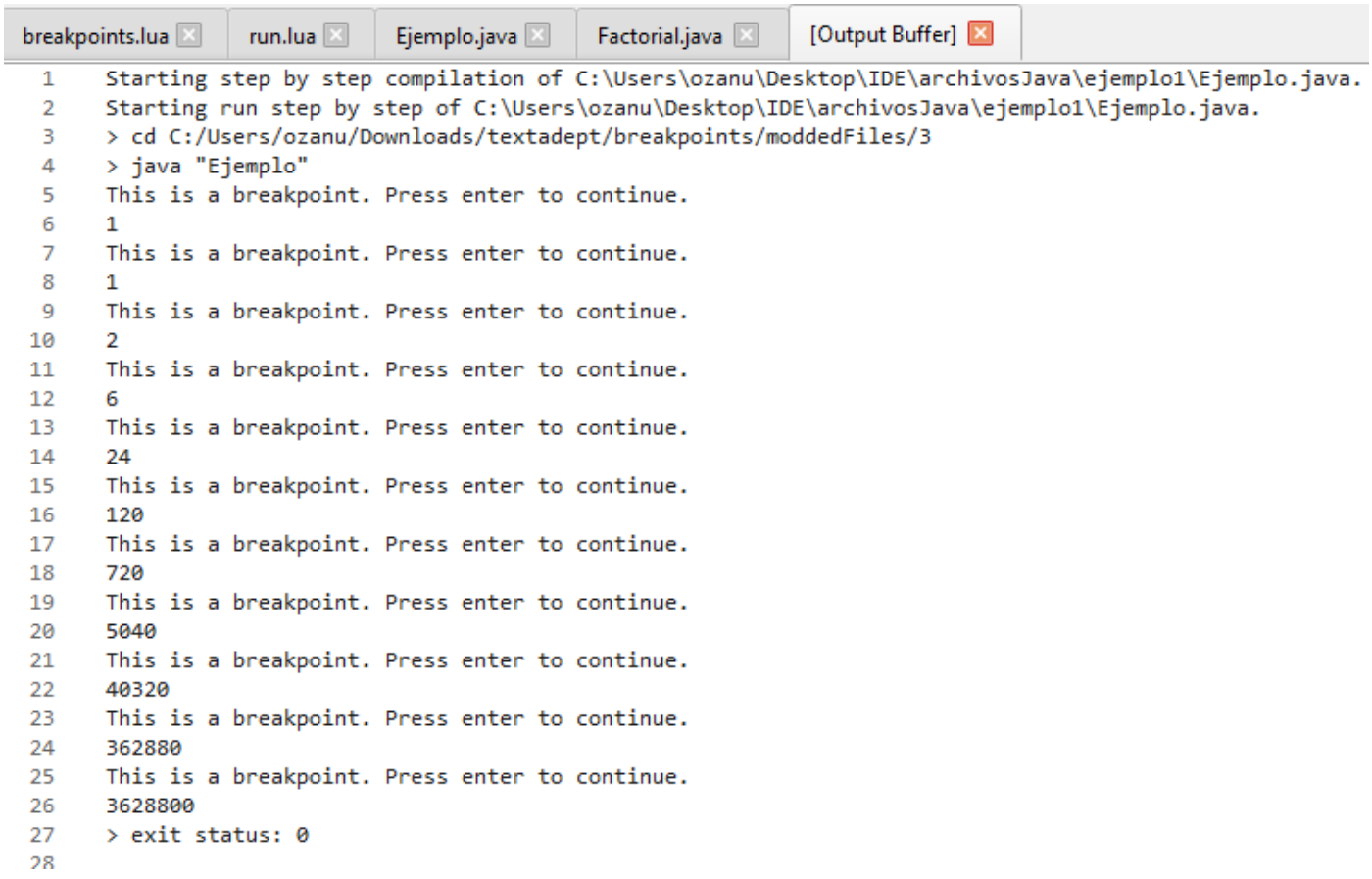


Figura 5-132. Ejecución paso a paso, en la que se indica cuándo se alcanza un punto de ruptura

5.8 Programas con recursividad o *arrays*

CP 1 Recursividad en *C*

Se realiza la compilación y ejecución de un programa en *C* con una función recursiva.

a. Hay un fichero en *C* con una función recursiva (figura 5-133).

```
File  Edit  Search  Tools  Buffer  View  Help
 1  #include <stdio.h>
 2
 3  int suma_enteros(int a)
 4  {
 5          if (a == 1) return a;
 6          return a + suma_enteros(a - 1);
 7  }
 8
 9  int main()
10  {
11
12      int a = 1;
13      int b = 9;
14
15      printf("The sum of %d is %d\n", a, suma_enteros(a));
16      printf("Their sum of %d is %d\n", b, suma_enteros(b));
17
18      return 0;
19  }
20
```

Figura 5-133. Fichero en C con una función recursiva, suma_enteros(int a)

b. Se compila y ejecuta como en otras pruebas anteriores, lo que da el resultado visible en la figura 5-134.

```
recur.c   [Output Buffer]
1   > cd C:\Users\ozanu\Desktop\IDE\compilacionC\Recursividad
2   > "recur"
3   The sum of 1 is 1
4   Their sum of 9 is 45
5   > exit status: 0
6
7
```

Figura 5-134. Ejecución correcta del programa

CP 2 Recursividad en *Java*

Se realiza la compilación y ejecución de un programa en *Java* con una función recursiva.

a. Hay un fichero en *Java* con una función recursiva (figura 5-135).

EjemploRecur.java | FactRecur.java

```
1   public class FactRecur
2   {
3           private int fact;
4
5           public FactRecur(int n) {
6                   this.fact = 1;
7                   if (n > 1) {
8                           FactRecur f = new FactRecur(n - 1);
9                           this.fact = n * f.getFact();
10                  }
11          }
12
13          public int getFact() {return this.fact;}
14  }
15
```

Figura 5-135. Fichero en Java con una clase recursiva, FactRecur(int n)

b. Se compila y ejecuta como en otras pruebas anteriores, lo que da el resultado de la figura 5-136.

EjemploRecur.java | FactRecur.java | [Output Buffer]

```
1   > cd C:\Users\ozanu\Desktop\IDE\archivosJava\Recur
2   > java "EjemploRecur"
3   1
4   1
5   2
6   6
7   24
8   120
9   720
10  5040
11  40320
12  362880
13  3628800
14  > exit status: 0
```

Figura 5-136. Ejecución correcta del programa

CP 3 Arrays

Se realiza la compilación y ejecución de un programa en *Java* que usa la clase *ArrayList*.

a. Hay un fichero en *Java* con una función recursiva (figura 5-137).

Listado.java | Lector.java | Member.java

```java
import java.util.ArrayList;
import java.util.Scanner;

public class Listado
{
	public static void main(String[] args) {
		ArrayList<Member> members = new ArrayList<Member>();
		while (true) {
			System.out.println("Do you wish to add a new team member?\nPush N if you don't want.");
			String conf = Lector.leeTexto();
			if (conf.equals("N") || conf.equals("n")) break;
			System.out.println("Write a name. ");
			String name = Lector.leeTexto();
			System.out.println("Write an age. ");
			int age = Lector.leeDatosNumericos();
			System.out.println("Write M if it is a man, otherwise if it is a woman. ");
			String sex = Lector.leeTexto();
			if (sex.equals("M") || sex.equals("m")) sex = "M";
			else sex = "F";
			System.out.println("Write the nationality. ");
			String nationality = Lector.leeTexto();
			System.out.println("Write the occupation. ");
			String occupation = Lector.leeTexto();
			members.add(new Member(name, age, sex, nationality, occupation));
		}
		System.out.println("The data of the team members are the following:");
		for (Member m : members) System.out.println(m);
		System.out.println("Finished.");
	}
}

```

Figura 5-137. Fichero en Java con arrays

b. Se compila y ejecuta como se observa en las figuras 5-138 y 139.

Listado.java | Lector.java | Member.java | [Output Buffer]

```
> cd C:\Users\ozanu\Desktop\IDE\archivosJava\Arrays
> java "Listado"
Do you wish to add a new team member?
Push N if you don't want.
Sviatlana
Write a name.
Sviatlana
Write an age.
24
Write M if it is a man, otherwise if it is a woman.
F
Write the nationality.
Belarus
Write the occupation.
Scouter
Do you wish to add a new team member?
Push N if you don't want.
Y
Write a name.
Julio
Write an age.
25
Write M if it is a man, otherwise if it is a woman.
M
Write the nationality.
Spain
Write the occupation.
Mathematician
Do you wish to add a new team member?
Push N if you don't want.
S
Write a name.
Yekaterina
Write an age.
25
Write M if it is a man, otherwise if it is a woman.
F
Write the nationality.
Ukrain
Write the occupation.
Anthropologist
Do you wish to add a new team member?
Push N if you don't want.
n
```

Figura 5-138. Primera parte de la ejecución del programa

```
45  The data of the team members are the following:
46
47  Data of member Sviatlana
48  Age = 24
49  Nationality: Belarus
50  Occupation:Scouter
51
52  Data of member Julio
53  Age = 25
54  Nationality: Spain
55  Occupation:Mathematician
56
57  Data of member Yekaterina
58  Age = 25
59  Nationality: Ukrain
60  Occupation:Anthropologist
61  Finished.
62  > exit status: 0
```

Figura 5-139. Segunda parte de la ejecución del programa

Capítulo 6

CONCLUSIONES Y TRABAJO FUTURO

6.1 Conclusiones principales

Varias conclusiones se extraen del trabajo realizado, por lo que es conveniente listarlas:

1. Un IDE es una herramienta de software de tipo múltiple o, dicho de otra manera, es un grupo de numerosas herramientas unificado dentro de un marco de trabajo. Tan sólo la edición de texto ha recomendado que se parta de una herramienta, *TextAdept*, existente bajo una licencia de software libre y todo lo demás se ha tenido que crear herramienta por herramienta.
2. La mayor dificultad que impone un IDE está relacionada con el hecho de que su funcionalidad principal, la compilación de ficheros codificados en un lenguaje de programación, implica la instalación de una máquina virtual o de un software con las instrucciones por un lado y la capacidad por parte del IDE de transmitir al sistema operativo las órdenes que la ejecutan.
3. En la actualidad existe una gran oferta de IDEs, tanto comerciales como de libre distribución. Lo más aconsejable si se quiere crear un IDE propio es partir de un editor de texto de licencia libre, ya que automatiza varias de las funcionalidades de un IDE.
4. Cuando el trabajo lo lleva a cabo una sola persona, lo más difícil es elegir qué hacer y cómo, ya que la simultaneidad que dan varios trabajadores implica la simultaneidad del avance en el trabajo.

6.2 Objetivos logrados

Se examina a continuación para cada objetivo el grado en que se ha logrado.

1. El estudio del arte del problema ha consistido en definir una tipología de IDEs, en realizar una comparativa entre los más usados y en comparar las alternativas presentes para construirlos. Grado satisfactorio.
2. La gestión de proyectos está implementada. Grado satisfactorio.
3. Se ha implementado la posibilidad de sacar el explorador de ficheros y la consola de resultados. Grado satisfactorio.
4. La gestión de plantillas alcanza a los ficheros. Grado parcialmente satisfactorio.

5. *TextAdept* ya incluía las funciones de edición habituales y otras más. Grado satisfactorio.
6. *TextAdept* ya incluía las funciones habituales de manejo de ventanas. Grado satisfactorio.
7. Se pueden marcar puntos de ruptura. Grado satisfactorio.
8. Si bien ya incluía compilación y ejecución estándar, se han añadido opciones para tanto compilación separada en C, como de ejecución paso a paso. No obstante, la última sólo existe para *Java*. Grado parcialmente satisfactorio.
9. *TextAdept* ya incluía la capacidad de integrar herramientas externas. Grado satisfactorio.
10. El funcionamiento multiplataforma está garantizado por el hecho de haber usado *TextAdept* como base, ya que su implementación, basada en el lenguaje de programación *Lua*, la garantiza. Grado satisfactorio.
11. *TextAdept* ya incluía opciones de configuración. Grado satisfactorio.

Como resumen, en esta tabla se exponen cuáles de los objetivos se han cumplido (en su caso) como consecuencia del trabajo presente o eran nativos de *TextAdept*, totalmente o modificando una secuencia de comandos nativa.

Tabla 6-1. Resumen sobre cómo se han cumplido los objetivos del sistema

Objetivo	**Es trabajo original**	**Es una modificación**	**Está presente en *TA***
2	X		
3	X		
4	X		
5			X
6			X
7	X		
8		X	
9			X
10			X
11			X

6.3 Cambios realizados durante el desarrollo del proyecto

Los imprevistos son parte del desarrollo de un proyecto. Por otro lado, la primera vez que se realiza una tarea es imposible saber cuáles de las actividades que supone será la más compleja, por ignorancia o por otras razones (por ejemplo, que las tecnologías usadas tengan fallos).

Se listan los cambios realizados y, en su caso, se añade una autocrítica:

1. La gestión de proyectos y ficheros sufrió varias alteraciones durante su implementación.

    a. La primera aproximación era crear un fichero que contuviera el nombre de todos los ficheros del proyecto, pero se abandonó por innecesario y farragoso. En su lugar, se usan métodos de la librería *lfs* para investigar cuáles ficheros existen en un proyecto.

    Este problema no habría ocurrido con un mejor conocimiento de esta librería.

    b. No se creó sino con posterioridad el fichero que registra el espacio de trabajo al que se ha asignado un proyecto, pues no se cayó en la cuenta sino después de la posibilidad de que, primero, es confuso que un proyecto pertenezca a dos espacios de trabajo y, segundo, que sería problemático que en uno de los espacios se eliminara un proyecto y además se eliminara su directorio, quedando en el otro el registro de su existencia.

    Este problema apareció como consecuencia de decidir que los espacios de trabajo fueran virtuales, lo que habría facilitado la tarea, aunque la importación desde otros directorios no hubiera sido posible.

    c. El registro de proyectos y ficheros distinguía entre mayúsculas y minúsculas porque es el modo por defecto en que la comparación de texto se hace en

2. La gestión de espacios de trabajo, de plantillas y de puntos de ruptura incorporaron después la comprobación de que existieran sus directorios y ficheros específicos para asegurar el correcto registro de estos elementos.

    El problema no fue reparar en que era aconsejable crear la estructura de datos in situ en su caso.

3. Uno de los mayores cambios fue que, si bien se empezó introduciendo las funciones en los módulos existentes de *TextAdept*, con el tiempo se crearon módulos específicos para cada grupo de funcionalidades relacionadas, ya que así se facilitaba la legibilidad del trabajo realizado y la propia edición para invocarlo con el teclado, por ejemplo.

    Sin duda, se debería haber considerado mejor cómo implementarlo antes.

4. Ha habido que familiarizarse con los detalles del lenguaje de programación *Lua*, que tiene muchas peculiaridades.

5. Este problema es específico del lenguaje *C*: la función usada para detener la ejecución del código y esperar a la confirmación del usuario se tuvo que cambiar porque la tradicional *scanf* no ha sido continuada en las últimas versiones de este lenguaje y da problemas en el programa objeto compilado.

6.4 Funcionalidades del IDE

Se resumen a continuación las funcionalidades finales del IDE creado en el presente proyecto.

1. Tiene las mismas funcionalidades de un editor de texto porque *TextAdept* es su base.

2. Tiene resaltado y plegado de código, pudiendo activarse el acabado inteligente de código.
3. Presenta diversos tipos de gestión: de espacios de trabajo, de proyectos y ficheros y de plantillas.
4. Tiene un explorador de ficheros y una consola de resultados, pudiendo activarse o desactivarse como se quiera.
5. Compila tanto en un solo fichero como separadamente.
6. Permite indicar puntos de ruptura.
7. Ejecuta programas, en *Java* existe la opción de hacerlo paso a paso.
8. Reconoce por la extensión del fichero en qué lenguaje está escrita una secuencia de comandos.
9. Tiene opciones de configuración y extensibilidad.

6.5 Líneas de trabajo futuro

Respecto al trabajo futuro, hay también varias posibilidades.

1. Incluir más lenguajes de programación, como *Python*, el propio *Lua* o *Go*.
2. Incluir una visualización gráfica para distinguir mejor las ventanas del explorador de proyectos y la consola de resultados.
3. Mejorar la detección de puntos de ruptura mediante la indicación de cuáles puntos del código son aptos para detener la ejecución del código.
4. Mejorar la compilación separada, permitiendo un mejor seguimiento de la traza e introducir la orden que se quiera.
5. Relacionado con el anterior: la ejecución paso a paso es más informativa para el usuario con información visual de las variables existentes en el punto en que se ha parado el código, junto a información del ámbito y otros aspectos de la programación.
6. Relacionado con lo anterior: procurar que exista la ejecución paso a paso también para *C*.
7. Relacionado con lo anterior: añadir precompilación que permita descartar aquellos puntos de ruptura inviables.
8. Incluir más idiomas, como el español.

Bibliografía

¿Qué es la metodología ágil y cuáles son las más utilizadas? (14 de Febrero de 2023). Obtenido de zendesk: https://www.zendesk.com.mx/blog/metodologia-agil-que-es/

¿Que ha pasado con Visual Studio para Mac? (16 de Octubre de 2024). Obtenido de Microsoft Learn: https://learn.microsoft.com/es-es/visualstudio/releases/2022/what-happened-to-vs-for-mac

Acme (text editor). (2024 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Acme_(text_editor)&oldid=1227907098

Alphatk. (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Alphatk&oldid=1191299170

Apache OpenOffice. (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Apache_OpenOffice&oldid=1284381433

Bluefish for Mac. (23 de Enero de 2017). Obtenido de MacUpdate: https://www.macupdate.com/app/mac/58574/bluefish

Brockmeier, J. (10 de Marzo de 2010). *Bluefish 2.0: Slim but powerful*. Obtenido de LWN.net: https://lwn.net/Articles/377999/

Chacon, S., & Straub, B. (16 de Junio de 2025). *Downloads*. Obtenido de git --everything-is-local: https://git-scm.com/downloads

Code::Blocks. (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Code::Blocks&oldid=1283378666

Cooper, M. (10 de Mayo de 2022). *What's new in Fedora Workstation 36*. Obtenido de Fedora Magazine: https://fedoramagazine.org/whats-new-fedora-36-workstation/

Crimson Editor. (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Crimson_Editor&oldid=1162050123

Dias, C. (3 de Diciembre de 2015). *Issue: Menu license links to non Open Source license*. Obtenido de Microsoft/vscode repo: https://github.com/Microsoft/vscode/issues/60#issuecomment-161792005

Eclipse (software). (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Eclipse_(software)&oldid=1281324054

Eid, A. (2007). *Textadept*. Obtenido de Textadept: https://orbitalquark.github.io/textadept/

Eid, A. (1 de Junio de 2025). *Textadept 12.7 API Documentation*. Obtenido de Textadept: https://orbitalquark.github.io/textadept/api.html

Eid, A. (1 de Junio de 2025). *Textadept 12.7 Manual*. Obtenido de TextAdept: https://orbitalquark.github.io/textadept/manual.html

Elvis (text editor). (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Elvis_(text_editor)&oldid=1246314999

Fowler, D. S. (6 de Octubre de 2023). *How to Create an IDE - Why Build Your Own IDE?* Obtenido de Tek Eye: https://tekeye.uk/programming/how-to-create-an-ide

gedit text editor. (2025). Obtenido de Microsoft: https://apps.microsoft.com/detail/9ncq8b0nvcw3?hl=es-ES&gl=ES

gedit, a text editor. (2025). Obtenido de gedit, a text editor: https://gedit-text-editor.org/

GitHub Staff. (8 de Junio de 2022). *Sunsetting Atom*. Obtenido de GitHub: https://github.blog/news-insights/product-news/sunsetting-atom/

GNU Emacs. (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=GNU_Emacs&oldid=1282763562

Hill, B. M., Helmke, M., & Burger, C. (2009). *The Official Ubuntu Book.* Prentice Hall .

Ho, D. (2025). *What is Notepad++*. Obtenido de Notepad++ Org: https://notepad-plus-plus.org/

Ierusalimschy, R. (2004). *21 – The I/O Library*. Obtenido de Programming in Lua: https://www.lua.org/pil/21.html

Ierusalimschy, R., Carregal, A., & Guisasola, T. (22 de Abril de 2020). *Manual*. Obtenido de LuaFileSystem: https://lunarmodules.github.io/luafilesystem/manual.html

Integrated Development Environment. (29 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Integrated_development_environment&oldid=1283543835

Lardinois, F. (29 de Abril de 2015). *Microsoft Launches Visual Studio Code, A Free Cross-Platform Code Editor For OS X, Linux And Windows*. Obtenido de TechCrunch: https://web.archive.org/web/20171028161004/https://techcrunch.com/2015/04/29/microsoft-shocks-the-world-with-visual-studio-code-a-free-code-editor-for-os-x-linux-and-windows/

List of text editors. (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=List_of_text_editors&oldid=1281169808

Lua. (23 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Lua&oldid=1295877480

Lua Programming in Lua (first edition). (3 de Julio de 2020). Obtenido de Contents: https://www.lua.org/pil/contents.html

Overview. (2018). Obtenido de Spyder IDE: https://web.archive.org/web/20190123054617/http://docs.spyder-ide.org/overview.html

Python Software Foundation, 2. (24 de junio de 2025). *IDLE - Python Editor and Shell*. Obtenido de Python.org: https://docs.python.org/es/3.13/library/idle.html

Reading and Writing Files. (23 de Junio de 2025). Obtenido de Faily Historian 7: https://www.family-historian.co.uk/help/fh7-plugins/lua/readingandwritingfiles.html

The Most Common Question about Arachnophilia. (29 de Octubre de 2020). Obtenido de Arachnophilia: https://arachnoid.com/arachnophilia/#The_Most_Common_Question_about_Arachnophilia

The require function, 2. (2004). *The require function*. Obtenido de The Programming Language Lua: https://www.lua.org/pil/8.1.html

Timbó, R. (26 de Mayo de 2025). *Integrated Development Environments: Definition, What Is, and Purpose*. Obtenido de Revelo: https://www.revelo.com/blog/integrated-development-environments

Turturro, N. (14 de Noviembre de 2023). *Análisis y diseño de sistemas: Explorando los Sistemas Modernos*. Obtenido de DOOR3: https://www.door3.com/es/blog/system-analysis-and-design#:~:text=Diseño%20del%20sistema,-El%20diseño%20del&text=Esta%20fase%20incluye%20el%20diseño,los%20objetivos%20de%20la%20organización.

UltraEdit. (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=UltraEdit&oldid=1272798569

Vailshery, L. S. (Febrero de 2024). *Most used programming languages among developers worldwide as of 2024*. Obtenido de statista: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

Vera Code, 2. (2025). *What Is An Integrated Development Environment (IDE)?* Obtenido de Vera Code: https://www.veracode.com/security/integrated-development-environment/

vile (text editor). (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Vile_(text_editor)&oldid=1268122539

Vim (text editor). (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Vim_(text_editor)&oldid=1283239266

Visual Studio. (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Visual_Studio&oldid=1282322257

What is an IDE. (2025). Obtenido de MongoDB: https://www.mongodb.com/resources/basics/what-is-an-ide

Wikipedia, 2. (24 de Junio de 2025). *IDLE*. Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=IDLE&oldid=1274637768

Wikipedia, 2. (24 de Junio de 2025). *RStudio*. Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=RStudio&oldid=1282174211

Wikipedia, 2. (24 de Junio de 2025). *Spyder (software)*. Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Spyder_(software)&oldid=1287844172

Wikipedia, 2. (24 de Junio de 2025). *Visual Studio Code*. Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=Visual_Studio_Code&oldid=1294641412#cite_ref-17

XEmacs. (24 de Junio de 2025). Obtenido de Wikipedia: https://en.wikipedia.org/w/index.php?title=XEmacs&oldid=1280144774

GLOSARIO

Se listan los términos más importantes.

1. Aplicación: También llamada *app*(*lication*) por influencia del inglés, es un programa en un entorno cerrado.
2. Arquitectura: En un sistema informático, es la descripción de la estructura que forman los componentes de un computador, de un programa informático, de un conjunto de datos o cualquier otro elemento.
3. *Array*: En programación, se llama así a una lista de elementos ordenados. Por lo general, suelen ser del mismo tipo y ordenados a partir de 0 en orden numérico natural ascendente. En *Lua* no se cumplen ninguna de ambas de condiciones.
4. C: Lenguaje de programación imperativo, usado desde hace años por su versatilidad y capacidad modular.
5. Código: Texto escrito en un lenguaje de programación.
6. Compilación: Proceso consistente en traducir una secuencia de comandos sucesivamente a lenguaje intermedio, a lenguaje final y, finalmente, en lenguaje máquina.
7. Configuración: Conjunto de opciones en un programa o aplicación que se puede cambiar a voluntad del usuario.
8. Depurador: En un IDE, identifica aquellos elementos software encargados de trazar los errores que tengan lugar durante la compilación o ejecución de un programa.
9. Directorio: Ramificación existente dentro del sistema de ficheros del sistema operativo, dotada de la capacidad de contener ficheros.
10. Editor de texto: Software creado específicamente para la creación, edición y almacenamiento de ficheros de texto.
11. Ejecutable: Se llama así a un fichero que contiene un programa y puede, pues, ejecutarse desde la consola de comandos.
12. Entorno de desarrollo integrado: También conocido como IDE (de sus siglas en inglés, *Integrated Development Environment*), es un software creado específicamente para facilitar la edición, compilación y ejecución de programas informáticos.
13. Espacio de trabajo: Estructura de datos en la que se almacenan proyectos. Puede corresponder a un directorio o bien a otra estructura de datos. Suelen caracterizarse por la unicidad de los nombres de los proyectos y sus rutas.
14. Extensión de aplicación: Software externo a una aplicación que se otorga a esta para darle nuevas funcionalidades.

15. Extensión del fichero: Final del nombre de un fichero, seguida normalmente de un punto, que puede indicar el programa con el que se puede abrir el fichero o, si es una secuencia de comandos, el lenguaje en que se ha codificado.
16. Fichero: Elemento software que encapsula una secuencia de datos.
17. Función: En un programa denota un conjunto de líneas de código que pueden ser invocado desde otros puntos para que ejecuten las mismas operaciones sin tener que repetirlas cada vez. Muchas veces, tienen parámetros que corresponden a variables locales. En los inicios de los lenguajes de programación, la anterior definición correspondía a *subprograma* (también *subrutina*), que se distinguían en procedimientos y funciones, según si devolvían un valor (no y sí, respectivamente). Hoy en día se suelen llamar funciones de tipo vacío y con tipo.
18. Funcionalidad: Capacidad de un programa para realizar una tarea concreta.
19. Interpretación: Proceso análogo al de la compilación, con la diferencia de que en este caso primero se compila al llamado lenguaje intérprete y, posteriormente, la traducción del segundo a lenguaje máquina se hace a medida que el programa se ejecuta. Es típico de lenguajes de programación como Java.
20. Java: Lenguaje de programación orientado a objetos e interpretado.
21. Lenguaje de programación: Conjunto de palabras reservadas y reglas sintácticas que permite la elaboración de códigos que, pasando por analizadores léxicos y sintácticos, constituyen una secuencia de comandos.
22. Librería: Conjunto de secuencias de comandos que implementan una serie de herramientas software creadas para uso de otros programas. Es normal que tengan una relación temática, por ejemplo, un conjunto de funciones para operaciones matemáticas constituiría una librería de soporte matemático.
23. Lua: Lenguaje de programación imperativo, interpretado, en el que está codificado *TextAdept*.
24. Módulo: Conjunto de secuencias de comandos y librerías que modifican un programa concreto, otorgándole funcionalidades. En algunos casos también recibe el nombre módulo a los componentes del programa original.
25. Multiplataforma: Capacidad de un software de funcionar en más de un sistema operativo.
26. Plantilla: Modelo de fichero (como una secuencia de comandos) que se almacena para manipular sus copias según las necesidades del usuario.
27. Programa: Conjunto de secuencias de comandos y de otros tipos de ficheros cuya ejecución permite la realización de una o varias funciones. Una de las secuencias arranca el programa.
28. Programación: Acto de crear un programa en un lenguaje específico para tal fin.
29. Programación imperativa: Es aquella programación fundamentada en el uso de órdenes al sistema operativo, expresadas por lo general como asignaciones de memoria y llamadas a funciones.

30. Proyecto: Elaboración de un programa informático, consistente en los ficheros que llevará el programa final más otros que existan para realizar pruebas. Suele denotarse con versiones.
31. Punto de ruptura: Señal en una línea de una secuencia de comandos en la que, cuando se realiza una ejecución paso a paso, el programa se detiene y queda a decisión del usuario cuándo continuar.
32. Secuencia de comandos: Fichero de texto escrito en un lenguaje de programación, codificado según las reglas del segundo. La información se presenta en líneas de texto.
33. Sistema operativo: Arquitectura de información situada entre el núcleo del hardware y los programas de alto nivel, se encarga de que los últimos puedan transmitir las órdenes al primero.
34. Software: Componentes lógicos, esto es compuestos de información y no por elementos físicos.
35. Usuario: Persona que usará un programa informático, tanto para fines profesionales como personales.
36. Variable: En una secuencia de comandos, es una asignación de memoria que, para ser usada en otros puntos fácilmente, recibe un identificador, por lo general texto y, según el lenguaje de programación, números u otros símbolos.
37. Ventana: Interfaz visual que muestra la información de un fichero. En el caso de un fichero de texto, muestra su contenido.

Se emplean además estos dos acrónimos.

1. CRUD, gestión: *Create, Retrieve, Delete, and Update*. Representan las operaciones básicas de la gestión de los elementos de un archivo: Crear, recuperar, eliminar y actualizar.
2. IDE: *Integrated Development Environment*, (entorno de desarrollo integrado).

Apéndice A

Manual de instalación

1. Instalar *TextAdept* desde la página web de la referencia (Eid, Textadept, 2007).
2. Instalar Git, disponible en (Chacon & Straub, 2025). Es necesario encontrar la versión acorde al SO y la arquitectura del PC de trabajo, así como seguir las instrucciones del instalador.
3. Instalar gcc. Una manera sencilla de hacerlo es mediante esta página: https://sourceforge.net/projects/mingw/files/Installer/mingw-get-setup.exe/download Como antes, seleccionar el SO y la arquitectura es fundamental.
4. Instalar la máquina virtual de Java. https://www.java.com/en/download/manual.jsp
5. En el directorio principal de *TextAdept*, es necesario:
    a. Del código fuente, copiar la secuencia de comandos *init.lua* del directorio principal y sobreescribirla en la instalación.
    b. Añadir dentro del directorio *modules* el directorio *IDE* creado en este PFG.

Apéndice B

Manual de usuario

Este manual se basa en el manual de *TextAdept* (Eid, Textadept 12.7 Manual, 2025) (Eid, Textadept 12.7 API Documentation, 2025). Se explican a continuación someramente las funcionalidades comentadas en los casos de uso, detallando tanto la pestaña del menú en la que se aloja la operación como el atajo de teclado correspondiente.

- Funcionalidades introducidas como trabajo del proyecto.
    1. File:
        - *Create Workspace*: CTRL + ALT + C
        - *Erase Workspace*: CTRL + ALT + S
        - *Open Workspace*: CTRL + ALT + O
        - *Start Project*: CTRL + SHIFT + N
        - *Erase Project*: CTRL + SHIFT + E
        - *Import Project*: CTRL + SHIFT + I
        - *Open Files from Project*: CTRL + ALT + I
        - *Save File in Project*: CTRL + ALT + N
        - *Erase Files from Project*: CTRL + ALT + Q
        - *New:* CTRL + N
        - *Open:* CTRL + O
        - *Save:* CTRL + S
        - *Save As:* CTRL + SHIFT + S
        - *Close:* CTRL + W
        - *Close All:* CTRL + SHIFT + W
        - *Save As Template*: CTRL + SHIFT + ALT + S
        - *Open Templates*: CTRL + SHIFT + ALT + O
        - *Erase Templates*: CTRL + SHIFT + ALT + D
    2. Tools:
        - *Compile:* CTRL + SHIFT + C
        - *Compile separately*: CTRL + SHIFT + ALT + C
        - *Run:* CTRL + R
        - *Run Step by Step*: CTRL + SHIFT + ALT + B
    3. View:
        - *Toggle Project Explorer*: CTRL + T.
        - *Toggle Output View*: CTRL + SHIFT + ALT + T

Apéndice C

Código fuente

Este es el enlace que lleva al software en formato comprimido *.7z*:
Sistema IDE