Desarrollo de una Herramienta de Refactorización en Erlang Integrada en un IDE

César Tomás Franco

Universidad Politécnica de Valencia

Directores:

Germán Fco. Vidal Oriola

Salvador Tamarit Muñoz

Valencia, Septiembre 2010

Índice

1. INTRODUCCIÓN

El presente proyecto se centra en la implementación de una herramienta para **refactorizar programas codificados mediante el lenguaje Erlang**. Esta herramienta está integrada en un entorno de desarrollo dotándola así de una mayor usabilidad y funcionalidad.

A continuación se va a definir a grandes rasgos algunos de los conceptos clave para entender de forma clara la temática de este proyecto.

Erlang [5] es un lenguaje de programación **funcional** con soporte de programación concurrente y un sistema de ejecución que incluye una máquina virtual y bibliotecas. Originalmente, Erlang era un lenguaje propietario de la compañía Ericsson, pero fue cedido como software de código abierto en 1998.

La mayor fortaleza de Erlang es el soporte para **concurrencia**. Tiene un pequeño pero potente conjunto de primitivas para crear procesos y comunicar entre los mismos. El soporte para procesos distribuidos es también parte de Erlang. Los procesos se pueden crear en nodos remotos, y la comunicación con ellos es transparente.

El término **refactoring** o **refactorización** [16] se usa a menudo para describir la modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por *limpiar el código*. La refactorización se realiza a menudo como parte del proceso de desarrollo del software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Los test aseguran que la refactorización no cambia el comportamiento del código.

La refactorización es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad. El objetivo, por el contrario, es mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento en el futuro así como incrementar su eficiencia. En este proyecto se va a diseñar una herramienta que analizará el código proporcionado y devolverá el programa refactorizado (atendiendo a los distintos criterios de refactorización contemplados) con la misma funcionalidad.

En este proyecto se van a tener en cuenta diversos criterios a la hora de aplicar las refactorizaciones a un programa Erlang dado.

Por último, destacaremos el hecho que esta herramienta esté **integrada el entorno de programación** (IDE) **Eclipse** [21]. Se puede considerar que este es uno de los puntos fuertes del desarrollo del proyecto, ya que hoy en día la mayor parte de desarrollo de software (y especialmente el de Erlang) se realiza bajo este tipo entornos.

Los IDE proporcionan un marco sencillo y amigable a la hora de desarrollar software al tener múltiples opciones y herramientas enfocadas a facilitar al programador su tarea de codificación. Pese a que Erlang es un lenguaje relativamente nuevo, existen actualmente diversos entornos de programación que ya soportan dicho lenguaje, en el que se proporcionan

herramientas (como por ejemplo debuggers) y funcionalidades que simplifican enormemente el esfuerzo dedicado por los programadores a tal fin.

Así, la integración de esta herramienta en Eclipse le proporciona una usabilidad y funcionalidad que es esencial para que esta sea aceptada y usada por la comunidad de programadores que estén interesados en el desarrollo de software con el lenguaje Erlang.

Con este proyecto se pretende impulsar y fomentar varios conceptos que están experimentando un aumento en el interés de la comunidad de programadores, como son la programación funcional, la concurrencia además de la eficiencia y legibilidad del código desarrollado. Si a esto le añadimos el hecho de que esta herramienta esté integrada en un entorno de programación podemos concluir que nos encontramos delante de un proyecto que puede llegar a tener una aplicación práctica realmente útil.

1.1 CONCURRENCIA

Concurrencia en el ámbito de la computación se conoce por la habilidad de diferentes tareas para ejecutarse simultáneamente sin afectarse entre ellas (en términos de comunicación y sincronización) a menos que se desee explícitamente.

Según Joe Amstrong en [4], el mundo real en el que vivimos es concurrente, sin embargo la mayoría de los lenguajes de programación que usamos para escribir programas que interactúan con el mundo real son generalmente secuenciales.

El hecho que se usen lenguajes de programación esencialmente secuenciales para codificar programas concurrentes dificulta aún más esta tarea e induce a la noción de que la programación concurrente es complicada. Efectivamente si escribimos algún programa concurrente en lenguajes como Java o C++ que están diseñados para la programación secuencial nos encontraremos ante un problema realmente complicado.

Por el contrario, existe una generación de lenguajes orientados a la concurrencia, como son Erlang u Oz [23], en los que codificar programas concurrentes es la forma natural de expresarse. Joe Amstrong afirma que para poder clasificar un lenguaje como un Lenguaje Orientado a la Concurrencia se debería aplicar el siguiente criterio:

- Debe permitir crear un gran número de procesos.
- La creación y destrucción de procesos debe ser muy eficiente.
- El paso de mensajes entre procesos debe tener un coste (computacional, temporal) muy bajo.
- Los procesos no deben compartir información y deberían comportarse como si estuvieran ejecutándose en maquinas distintas.

El concepto de proceso es muy conocido entre los programadores y los científicos de computación, lamentablemente pocos lenguajes o sistemas operativos están enfocados para proporcionar al programador usabilidad respecto a estos. En la mayoría de los lenguajes el modelo de concurrencia es el mismo que el del sistema operativo subyacente. Esto implica,

por ejemplo, que un programa concurrente escrito en Java ejecutándose en un sistema operativo (con sus mecanismos de creación de procesos o política de planificación particular) tiene una semántica completamente diferente a la que tendría ejecutándose en otro sistema operativo. Esta dependencia del sistema operativo subyacente también ofrece desventajas en el sentido que los sistemas operativos más utilizados actualmente no soportan "procesos ligeros", además de no permitir la ejecución concurrente de un gran número de procesos.

Joe Amstrong afirma que la concurrencia debería ser responsabilidad del lenguaje de programación y no del sistema operativo sobre el que se está ejecutando, hecho que se da precisamente en el lenguaje Erlang. Para ilustrar esto Joe Amstrong nos ofrece multitud de pruebas y experimentos realizados sobre distintos sistemas operativos y lenguajes de programación que nos muestran el alto coste asociado a la creación y destrucción de procesos así como el paso de mensajes entre estos.

A continuación se va a detallar cual es la filosofía adoptada por Erlang respecto a su modelo de concurrencia.

- En Erlang los procesos son "ligeros", esto significa que se necesita muy poco coste para la creación o destrucción de estos, concretamente del orden de magnitud de una o dos veces más "ligeros" o rápidos que los hilos de los sistemas operativos.
- Además, se pueden crear cientos o miles de estos procesos sin acusar un descenso notable del rendimiento del sistema (a menos, claro está, que todos estuviera ejecutando "algo" al mismo tiempo).
- Los procesos en Erlang no comparten ningún tipo de información, hecho que incrementa enormemente la eficiencia del código. La única forma de intercambiar la información es mediante paso de mensajes explícito. Los mensajes en Erlang no contienen punteros, así que cada proceso debe trabajar con una copia de la información que requiere, hecho que propicia una programación simple de sistemas tolerantes a fallos. Así, la sincronización se realiza también mediante el intercambio de mensajes. Las principales ventajas que ofrece la filosofía de no compartir información son:
    - Los sistemas son fácilmente distribuibles, para convertir un programa en distribuible es tan sencillo como ubicar los distintos procesos paralelos en maquinas distintas.
    - Los sistemas se pueden hacer fácilmente tolerantes a fallos, distinguiendo entre procesos ejecutores (o "workers") y procesos observadores, que se encargan de realizar el tratamiento del error en el caso de que alguna cosa haya fallado en los procesos ejecutores.
    - Facilidad en la escalabilidad de los sistemas, añadiendo más procesadores y moviendo procesos entre estos.

Supongamos que quisiéramos programar un servidor de mensajería instantánea en Erlang, que soportara transmisión de mensajes entre miles de usuarios en un sistema como lo es Google Talk o el chat de Facebook. La filosofía de diseño de Erlang es asignar un proceso nuevo a cada evento para que la estructura del programa refleje directamente la concurrencia de múltiples usuarios intercambiando mensajes. En un sistema de mensajería instantánea, un evento podría considerarse a una actualización de presencia de usuario, un mensaje enviado o recibido, una solicitud de acceso… Cada proceso se encargaría del evento que maneja y terminaría cuando la petición hubiera sido completada. Se podría hacer lo mismo en lenguajes tales como C o Java, pero ofrecerían grandes dificultades a la hora de escalar el sistema a cientos de miles de eventos simultáneos. Erlang no presenta estas dificultades ya que no usa los hilos del sistema subyacente para representar los procesos, sino que tiene su propio planificador en la maquina virtual, que permite crear procesos de forma muy eficiente y

minimizando su uso de memoria. Esta eficiencia se mantiene sin tener en cuenta el número de procesos concurrentes presentes en el sistema.

Con todo esto, se puede afirmar que el uso de Erlang para programar sistemas concurrentes implica usar el modelo de concurrencia más potente que se conoce hoy en día.

1.2 ERLANG

Como hemos introducido anteriormente, Erlang es un lenguaje de programación concurrente y un sistema de ejecución que incluye una máquina virtual y bibliotecas. El subconjunto de programación secuencial de Erlang es un lenguaje funcional, con evaluación estricta, asignación única y tipado dinámico.

Fue diseñado para realizar aplicaciones distribuidas, tolerantes a fallos, soft-real-time y de funcionamiento ininterrumpido. Proporciona el cambio en caliente del código de forma que éste se puede cambiar sin parar el sistema.

La creación y gestión de procesos es trivial en Erlang, mientras que, en muchos lenguajes, los hilos de ejecución se consideran un apartado complejo y propenso a errores. En Erlang toda concurrencia es explícita, los distintos procesos se comunican usando el paso de mensajes mediante variables compartidas, sin tener la necesidad de usar bloqueos.

A continuación se describen las principales características del lenguaje Erlang:

- **Lenguaje Funcional:** Constituido únicamente por definiciones de funciones, entendiendo éstas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la transparencia referencial, y por tanto, la carencia total de efectos laterales.
- **Tiempo real:** Erlang está diseñado para programar sistemas complejos de tiempo real cuando se requiere un tiempo de respuesta de orden de milisegundos.
- **Operación continua:** Erlang tiene primitivas que permiten reemplazar código en un sistema en ejecución permitiendo a las distintas versiones del código ejecutarse al mismo tiempo. Esta característica tiene una gran aplicación en sistemas que no se pueden interrumpir para realizar cambios en el software, tales como sistemas de control de tráfico aéreo, intercambio telefónico…etc.
- **Concurrencia:** Erlang tiene un modelo basado en el proceso de concurrencia con paso de mensajes asíncrono. Los mecanismos de concurrencia en Erlang requieren muy poca memoria permitiendo así soportar aplicaciones con un número muy elevado de procesos concurrentes. Además, la creación y destrucción de procesos y paso de mensajes requiere muy poco coste computacional.
- **Robustez:** La seguridad es un requerimiento crucial en ciertos sistemas como los descritos anteriormente. Erlang provee tres construcciones para la detección de

errores en tiempo real, que pueden ser utilizadas para programar aplicaciones robustas en cuanto a seguridad se refiere.

- **Gestión de la memoria:** Erlang es un lenguaje de programación simbólico con un recolector de basura en tiempo real. La memoria se asigna automáticamente cuando es requerida y se desasigna cuando lleva un tiempo si ser utilizada. De esta forma nunca pueden aparecer los típicos errores de asignación de memoria.
- **Distribución:** Erlang no tiene memoria compartida. Todas las interacciones entre procesos se realizan mediante paso de mensajes asíncrono, de esta forma la construcción de sistemas distribuidos se realiza de forma muy sencilla. Cabe destacar también que aplicaciones diseñadas para ser ejecutadas en un solo procesador se pueden migrar para ser ejecutadas en redes de procesadores fácilmente.
- **Integración:** Erlang tiene un mecanismo de "puerto" que permite a los diferentes procesos comunicarse con el "exterior" (otros sistemas o procesos escritos en otros lenguajes de programación) de forma semánticamente equivalente al paso de mensajes entre los procesos en Erlang.

El objetivo inicial para el cual se concibió Erlang fue la producción de un lenguaje pequeño, simple y eficiente para programar aplicaciones industriales concurrentes robustas a gran escala. Así, por razones de eficiencia se evitaron incluir muchas características presentes en los lenguajes lógicos o funcionales modernos, tales como currificación, evaluación perezosa, variables lógicas…etc. No obstante su ausencia no representa un inconveniente a la hora de programar aplicaciones típicas de control industrial. El uso de la sintaxis de ajuste de patrones (*pattern matching*) junto con la propiedad de asignación única de variables de Erlang dan como resultado programas claros, cortos y fiables.

1.2.1 HISTORIA DE ERLANG

La historia de Erlang es importante para poder entender su filosofía. Erlang fue desarrollado para resolver requerimientos de sistemas distribuidos, tolerantes a fallos, masivamente concurrentes y de tiempo real. El hecho de que servicios web, computación de telefonía, sistemas de mensajes, integración empresarial, por mencionar solo algunos, comparten los mismos requerimientos en sistemas de telecomunicación explica porque Erlang ha ganando mucho terreno en todos estos sectores.

A mediados de los 80, el Laboratorio Científico de Computación de *Ericsson* empezó a investigar en lenguajes de programación para su nueva generación de productos de telecomunicaciones. Joe Armstrong, Robert Virding, y Mike Williams (bajo la supervisión de Bjarne Däcker) estuvieron dos años realizando prototipos de aplicaciones de telecomunicación con muchos de los lenguajes de programación disponibles en ese momento. Su conclusión fue que muchos de los lenguajes tenían características interesantes y no existía un único lenguaje que las abarcara casi todas, así que decidieron crear su propio lenguaje. *Erlang* estuvo influenciado por lenguajes funcionales tales como *ADA*, *Modula* y *Chill*, así como el lenguaje de programación lógica *Prolog*. Se tuvieron en cuenta las propiedades de actualización del software de *Smalltalk*, al igual que otras propiedades de los lenguajes de Ericsson *EriPascal* y *PLEX*.

Con una máquina virtual de Erlang basada en Prolog (VM) (programación lógica), se prototiparon aplicaciones de telecomunicaciones con una evolución de lenguaje que al final se fue convirtiendo en el lenguaje Erlang tal y como lo conocemos actualmente (lenguaje funcional). En 1991, Mike Williams creó la primera máquina virtual basada en C y un año después se puso en marcha el primer proyecto comercial con un pequeño grupo de desarrolladores. Este proyecto consistía en un servidor de movilidad que permitía comunicarse mediante teléfonos inalámbricos a través de redes de oficinas privadas. Este producto se puso en marcha finalmente en 1994 y proporcionó información valiosa sobre algunas mejoras que fueron integradas en Erlang en 1995.

Fue entonces cuando se consideró suficientemente maduro para su uso en grandes proyectos con cientos de desarrolladores tales como la banda ancha de Ericsson, GPRS o soluciones de conmutación ATM. Junto con estos proyectos se desarrolló el *framework Open Telecom Platform*, *OTP* que fue desarrollado por Ericsson para programar la siguiente generación de conmutadores. OTP incluía el sistema completo de desarrollo en Erlang junto con un conjunto de librerías escritas en Erlang y otros lenguajes. En 1998 Ericsson lanzó Erlang y las librerías OTP como código abierto. Actualmente OTP representa el uso comercial del lenguaje de programación funcional más importante fuera del ámbito académico.

El lanzamiento de Erlang como código abierto se realizó sin presupuesto ni comunicados de prensa ni ayuda del departamento de marketing de las empresas. En enero de 1999 el sitio web erlang.org tenía unas 36000 impresiones. Diez años después este número ha alcanzado los 2.8 millones. Este aumento es un reflejo de una comunidad cada vez mayor como resultado de una combinación de éxito comercial, investigación y proyectos de código abierto, blogs, y libros, todos impulsados por la necesidad de resolver problemas de software complejos en el dominio para el que Erlang originalmente fue concebido.

Actualmente nos podemos encontrar con numerosos e importantes proyectos desarrollados en Erlang tales como:

- Sistema de chat de Facebook
- Sistema de chat de la red social Tuenti
- Ejabberd: Servidor de mensajería instantánea
- Yahoo! Delicious
- Amazon SimpleDB: Base de datos distribuida que forma parte de los servicios web de Amazon.
- Twitterfal: Servicio para ver tendencias y diseños de Twitter.
- Wings 3D: Modelador 3D
- Servidor web Yaws

1.3 REFACTORIZACIÓN

Escribir código de forma clara y simple es una tarea recomendable, esto sin embargo puede resultar difícil incluso tedioso. Para poder adquirir esta habilidad se requiere mucha experiencia en el desarrollo de programas en dicho lenguaje y un buen conocimiento de todas las construcciones del lenguaje disponibles. De esta forma se pueden encontrar alternativas a la hora de expresar de distintas formas las intenciones de programación, incluso aplicar un poco de "disciplina" o buenas prácticas a la hora de escribir código.

Para ayudar a los programadores a escribir código "de calidad" la mayoría de los lenguajes actuales tienen asociados sitios web, libros o incluso blogs que proporcionan abundante información sobre buenas prácticas de codificación. No obstante, los programadores normalmente muestran cierta reticencia a la lectura y estudio de estos documentos, tendiendo a escribir y acumular grandes cantidades de código de baja calidad.

Se denomina refactoring al proceso de mejora del diseño de un programa sin cambiarle el comportamiento externo. Así, esta preservación del comportamiento garantiza que el refactoring no introduce o elimina ningún error. El objetivo principal que persigue la técnica del refactoring es eliminar las grandes cantidades de código de baja calidad existente, mejorando su facilidad de comprensión, cambiando su estructura y diseño, incluso eliminando código muerto, facilitando su mantenimiento en el futuro así como incrementando su eficiencia.

La refactorización no se basan únicamente en la sintaxis de los programas, sino que también requiere conocimiento semántico (tipos, estructura de módulos, …) adquirido mediante diversos análisis estáticos. Además, cada refactorización tiene normalmente asociadas una serie de precondiciones que han de tenerse en cuenta antes de poder aplicarse.

A continuación introducimos una serie de ejemplos que muestran distintos fragmentos de código Erlang con el objetivo de ilustrar este concepto de una forma más clara:

- **Ejemplo 1:**

En este ejemplo se muestra una de las posibles aplicaciones de refactoring más sencillas que se pueden encontrar, a la par que efectivas en algunos casos. La transformación "extracción de función" extrae una expresión o una secuencia de expresiones de una función a una nueva función diferente. Esta transformación crea una nueva definición de función y reemplaza el código en cuestión con una aplicación de dicha función. Las variables que se usan en el interior del código pero son declaradas o usadas en el exterior se convierten en parámetros formales de la nueva función. En el siguiente código se puede observar como quedaría el código después de aplicar esta transformación:

Extracción expresión X+2	Resultado después de la extracción

`func(X) ->`
`  Y = X + 2.`
	`func(X) ->`
	`  Y = newfun(X).`
	
	`newfun(X) ->`
	`  X + 2.`

Como se puede observar en el ejemplo, la aplicación de esta transformación preserva totalmente la funcionalidad y semántica del código original. En este sencillo ejemplo se realiza una mejora en cuanto a la estructuración del código se refiere. No obstante, si el código que se ha extraído de la función se hubiera repetido más veces dentro del código proporcionado se hubiese producido también un aumento de la eficiencia, reducción del tamaño del código de la función y dotándola de una mejor legibilidad. A continuación mostramos un ejemplo un poco más complejo del mismo tipo de transformación.

- **Ejemplo 2:**

Código original	Resultado después de la transformación

`quadratic(A,B,C) ->`
`  D = B * B - 4 * A * C,`
`  if`
`  D == 0 ->`
`  {-B / 2 / A};`
`  D > 0 ->`
`  S = math:sqrt(D),`
`  {-(B+S)/2/A,`
`  -(B-S)/2/A}.`
`  D < 0 ->`
`  no_solution`
`  end.`
	`quadratic(A,B,C) ->`
	`  D = B * B - 4 * A * C,`
	`  if`
	`  D == 0 ->`
	`  {-B / 2 / A};`
	`  D > 0 ->`
	`  two_sol(A, B, D);`
	`  D < 0 ->`
	`  no_solution`
	`  end.`
	
	`two_sol(A, B, D) ->`
	`  Sqrt = math:sqrt(D),`
	`  {-(B + Sqrt) / 2 / A,`
	`  -(B - Sqrt) / 2 / A}.`

En este ejemplo se observa que se produce una mejora de la estructuración y legibilidad del código al extraer el fragmento de código de la variable S del código original en la función `two_sol` del código resultante de la aplicación de la transformación. De esta forma, para preservar la semántica, la nueva función ha de tener tres parámetros que obtienen valores en el exterior del fragmento de código afectado por la transformación y se usan en su interior.

- **Ejemplo 3:**

En el último ejemplo que se va a mostrar se aplica una serie de transformaciones de simplificación de expresiones, en la que se puede observar una notable reducción del código original, aumentando su eficiencia y legibilidad y reduciendo considerablemente su tamaño. Este código se encarga de comprobar la validez de la fecha que se le esta pasando (en forma de tupla con tres valores) como argumento.

Código original

```
is_valid_time({H1, H2, H3}) ->
     Hour = if (H1 >= 0) and (H1 < 24) -> true;
               true -> false
            end,
     Minute = if (H2 >= 0) and (H2 < 60) -> true;
               true -> false
            end,
     Sec = if (H3 >= 0) and (H3 < 60) -> true;
               true -> false
            end,
     lists:all(fun(X) -> X == true end,
               [Hour, Minute, Sec]).
```

Aplicación de la simplificación 1

```
is_valid_time({H1, H2, H3}) ->
          Hour = (H1 >= 0) and (H1 < 24),
          Minute = (H2 >= 0) and (H2 < 60),
          Sec = (H3 >= 0) and (H3 < 60),
          lists:all(fun (X) -> X == true end,
                [Hour, Minute, Sec]).
```

Aplicación de la simplificación 2

```
is_valid_time({H1, H2, H3}) ->
          Hour = is_between(H1, 0, 23),
          Minute = is_between(H2, 0, 59),
          Sec = is_between(H3, 0, 59),
          Hour andalso Minute andalso Sec.
```

Llegados a este punto se puede afirmar que el proceso de refactorización es una tarea bastante costosa si se realiza a mano, pero actualmente con las herramientas de refactorización disponibles se convierte en una tarea muy sencilla y que proporciona grandes ventajas en cuanto a la calidad del código se refiere. Este preciso objetivo es el que se persigue en el presente proyecto. En el siguiente punto se analizan algunas de estas herramientas disponibles actualmente.

1.3.1 HERRAMIENTAS DE REFACTORIZACIÓN

Una de las opciones a la hora de aplicar la técnica del refactoring es hacerlo manualmente, para ello el usuario debe identificar en el código posibles aplicaciones de distintas refactorizaciones, pudiendo utilizar herramientas que facilitan la aplicación de dicha técnica. Así, aunque sea posible refactorizar un programa manualmente, esta tarea resulta tediosa, imprecisa y propensa a errores, por lo tanto, es casi necesario el uso de una de estas herramientas de refactorización de programas, ya que permiten aplicar (incluso deshacer) las distintas refactorizaciones de forma muy sencilla.

Las herramientas de refactorización necesitan acceder tanto a la información sintáctica como a la semántica del programa en cuestión. La mayoría de estas herramientas operan de la siguiente manera: primeramente transforman el código fuente del programa a una representación interna (como por ejemplo un Árbol Sintáctico Abstracto o *AST*). Seguidamente se extrae de manera estática la información semántica del programa, necesaria para a continuación poder llevar a cabo el análisis del programa basándose en dicha información y en la representación interna extraída, comprobando previamente las precondiciones de dicha refactorización. Si se satisfacen dichas precondiciones se llevan a cabo en la representación interna las distintas transformaciones contempladas por el criterio de refactorización en cuestión del programa. Por último, la representación interna transformada del programa se ha de revertir al código fuente antes de volver a mostrarse al usuario, preservado de la mejor forma posible la representación original que este tenía (comentarios, estructura…). Así el programador podrá reconocer su código después de la aplicación de las transformaciones.

Centrándonos ahora en el ámbito del presente proyecto, cabe destacar que actualmente existen distintas herramientas muy completas de refactorización de programas escritos en Erlang. Algunos ejemplos de ellas son:

- **Wrangler** [24]: Herramienta de refactorización interactiva de programas Erlang disponible bajo licencia de código abierta pública e integrada en los IDE Emacs y Eclipse (con la integración del plug-in ErlIde [22]). En la figura 1.1 y 1.2 podemos ver como se realiza esta integración. Se trata de la herramienta de refactorización de código Erlang más madura. Fue ideada y desarrollada principalmente por Huiqing Li y Simon Thompson de la Universidad de Kent (Reino Unido). Está implementada en el propio Erlang, soportando variedad de refactorizaciones, funcionalidades de detección y eliminación de “smell code” (síntomas en el código de la existencia de posibles problemas) o de código duplicado o clonado.

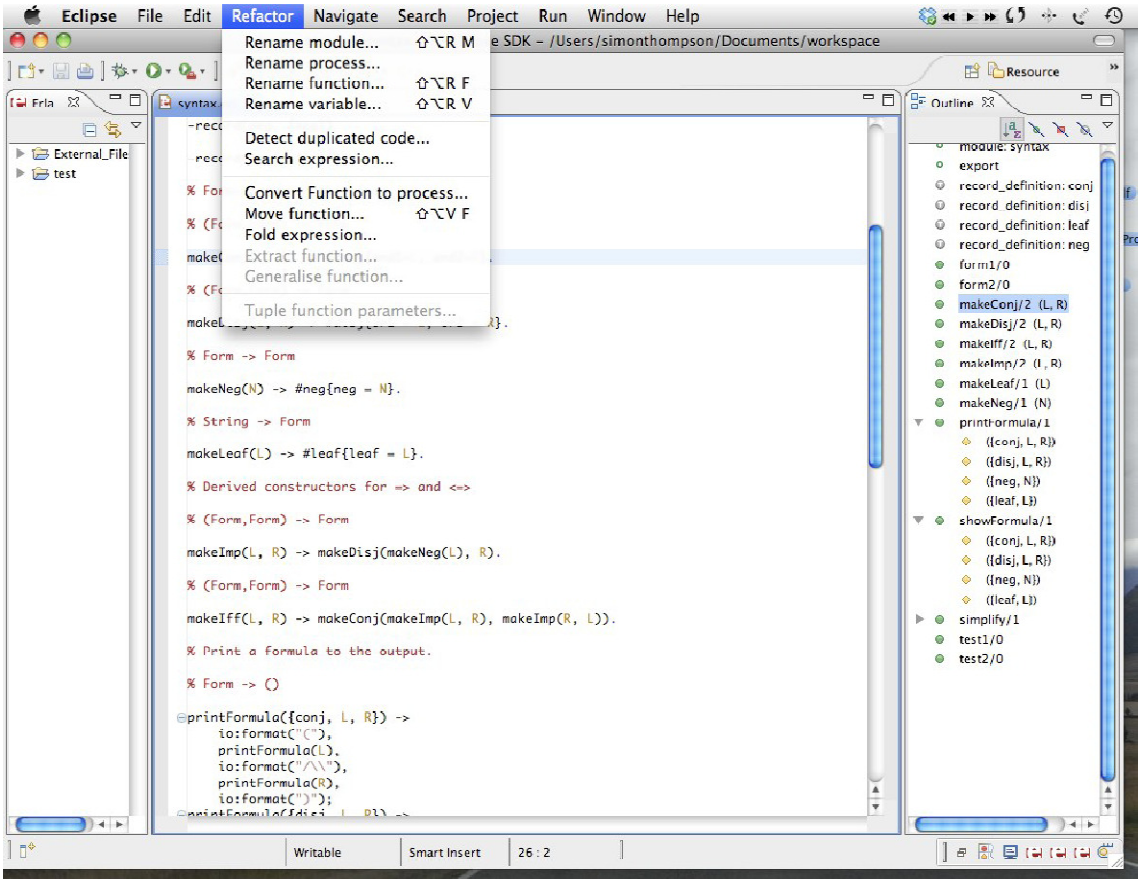

Figura 1.1: Integración de Wrangler en el IDE Eclipse

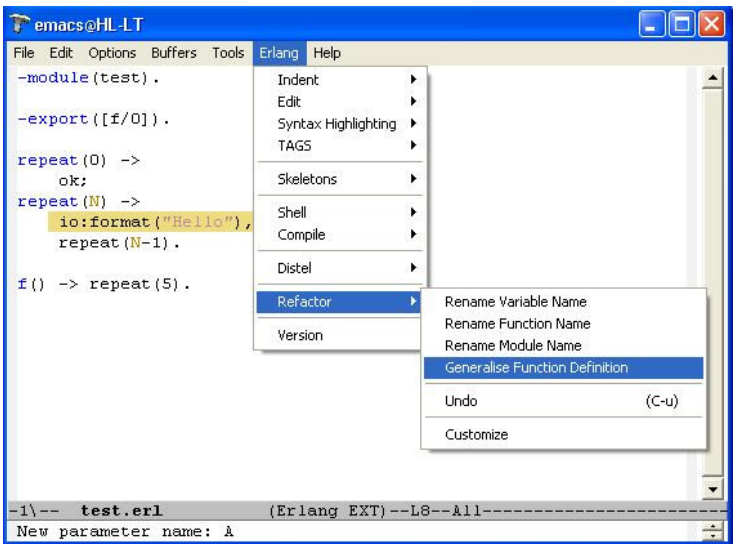

Figura 1.2: Integración de Wrangler en el IDE Emacs

- **RefactorErl** [25]: Desarrollada por investigadores de la Universidad de Eötvös Loránd en Budapest (Hungría). Asiste a los programadores de Erlang realizando refactorizaciones semiautomáticas de su código. Crea un grafo de la semántica formal del código y lo guarda en una base de datos relacional, pudiendo ser modificado en el

nivel del árbol sintáctico, en el cual se puede volver a reproducir el código. Proporciona una interfaz de usuario sencilla para aplicar las refactorizaciones. Se trata de una herramienta de fácil instalación y uso tanto en Windows como en sistemas Unix.

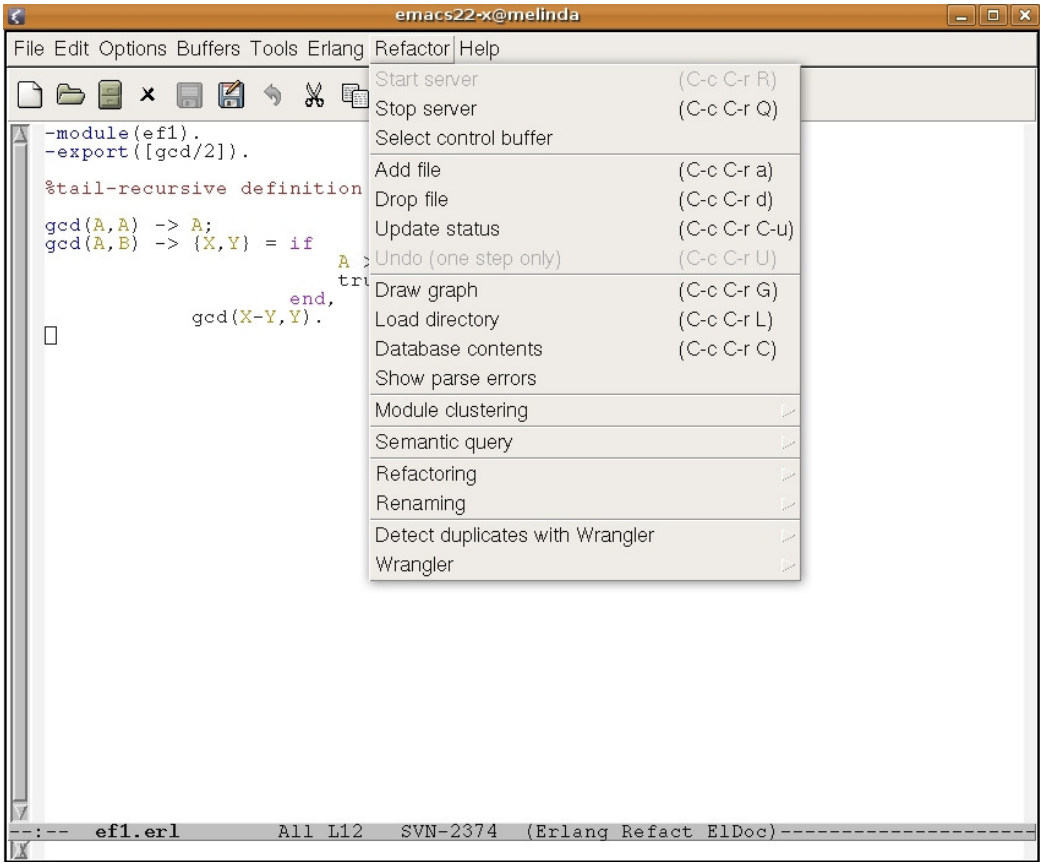

Figura 1.3: RefactorErl en Emacs

- **Tidier** [2]: Herramienta de refactorización que soporta el modo completamente automático (sin la supervisión del usuario). Ofrece mucha flexibilidad al usuario a la hora de configurar como se van a aplicar las distintas refactorizaciones, permitiendo incluso elegir entre la aplicación automática o supervisada. Ofrece multitud de acciones a realizar sobre el código (eliminación de "smell code", modernización de construcciones obsoletas del lenguaje…) además de las distintas aplicaciones de refactorizaciones disponibles. De fácil uso e instalación, sin estar ligada a ningún editor o IDE.

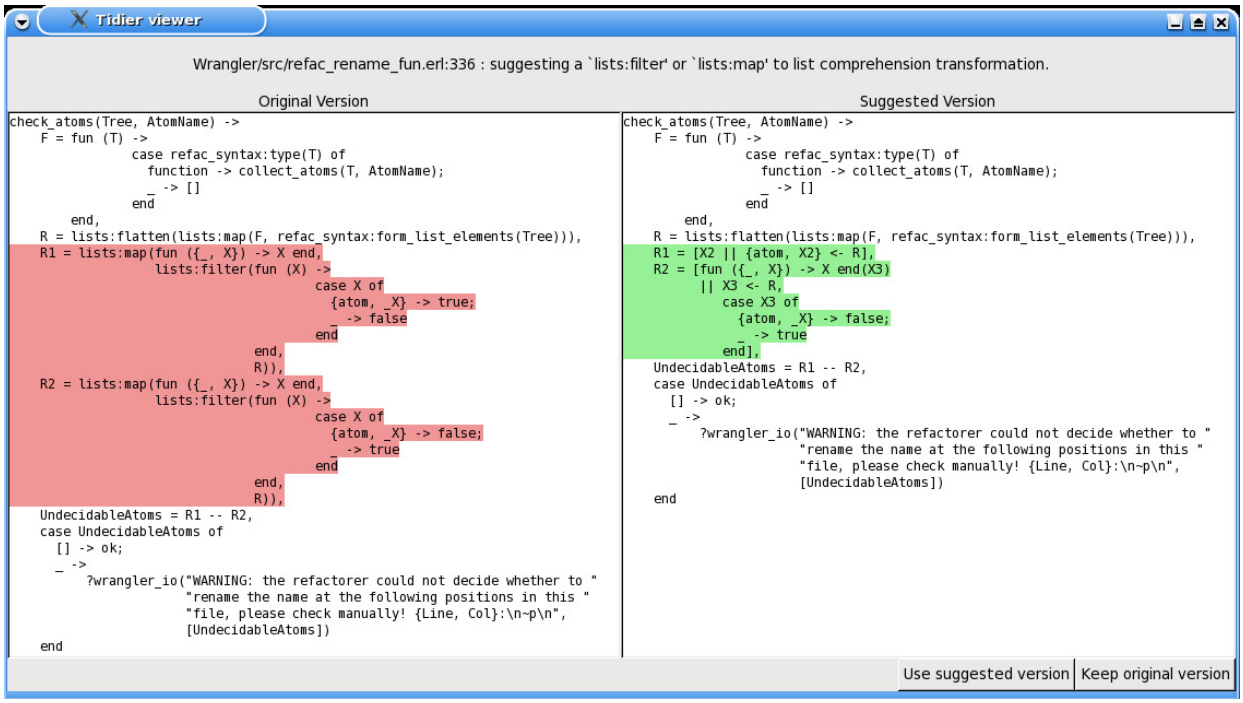

Figura 1.4: Ejecución de Tidier simplificando código del programa Wrangler

DISTRIBUCIÓN DEL DOCUMENTO

En el capítulo 2 de este documento encontramos los detalles referentes al diseño de esta herramienta. La implementación de esta herramienta se explica en el capítulo 3. En el capítulo 4 se aborda la integración de esta herramienta en un entorno de desarrollo. Se ofrece también un manual de usuario en el capítulo 5. Por último, en el capítulo 6 se realiza una conclusión del estudio realizado para el desarrollo de esta herramienta. En los apéndices de este documento podemos encontrar detallado el Abstract Format de Erlang así como la implementación de las funciones del módulo *SMERL* utilizado en el desarrollo de esta herramienta.

2. DISEÑO

En este punto se va a detallar la aproximación elegida en este proyecto para el desarrollo del mismo. Después de investigar un poco sobre las distintas opciones disponibles para el desarrollo de la herramienta de refactorización se optaron por los siguientes componentes y características:

- En cuanto al **front-end** de Erlang elegido, se ha optado por las *SyntaxTools* de la distribución Erlang/OTP. Esta librería contiene módulos para el manejo de los Árboles de sintaxis abstracta (AST) de Erlang, proporcionando numerosas funcionalidades para realizar análisis sintáctico (Parsing) de código Erlang.
- Para la **implementación** de los distintos refactorings se ha decidido trabajar con la representación estándar de los Árboles de Sintaxis Abstracta de Erlang o también conocido como ***Abstract Format*** [Apéndice A].
- Si nos fijamos en **la interfaz con el usuario** hay que destacar que este proyecto se ha basado en la herramienta de refactorización ***Wrangler,*** que junto con *Erlide* (plug-in para el desarrollo de Erlang en el IDE Eclipse) han permitido la integración de esta herramienta de refactorización en dicho entorno de desarrollo tan extendido y usado actualmente.

En la figura 2.1 se puede ver un diagrama de la implementación de la arquitectura de la herramienta de refactorización.

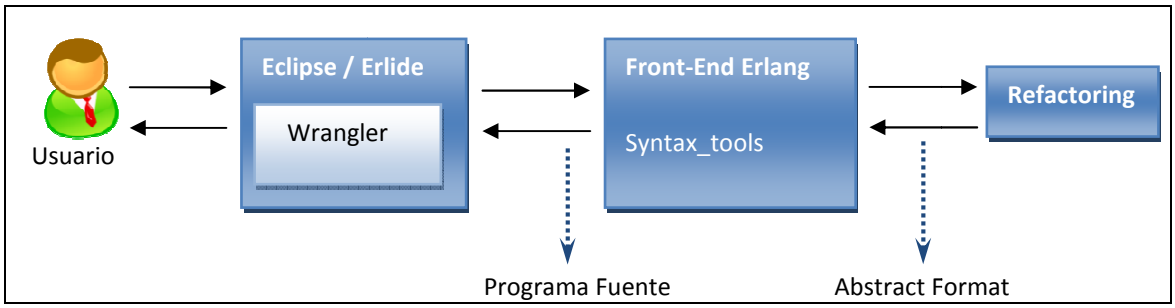

Figura 2.1: Arquitectura general de la herramienta de refactorización.

A continuación se va a describir con más detalle cada uno de los componentes presentes en la arquitectura elegida para el desarrollo de esta herramienta.

2.1. FRONT-END

Después de investigar en las distintas opciones disponibles en cuanto a Front-End (parte del software que interactúa con el o los usuarios) de Erlang, se decidió implementar la herramienta de refactorización con la infraestructura proporcionada por la librería ***SyntaxTools*** [26] de la distribución oficial de Erlang/OTP. La forma de trabajar sobre esta librería se ha realizado mediante un módulo llamado ***SMERL*** [20] que abstrae y simplifica enormemente el proceso de generación y manipulación de código Erlang en tiempo de ejecución. Para desarrollar las distintas refactorizaciones se decidió trabajar directamente sobre el ***Abstract Format*** de Erlang.

A continuación se explican con más detalle los distintos componentes mencionados:

- La librería **SyntaxTools** de Erlang contiene módulos para el manejo de Arboles de Análisis Sintáctico (AST) de forma que sean compatibles con los AST del módulo de la librería estándar *erl_parse* [27] junto con utilidades para la lectura de archivos de código fuente, impresión de dichos AST y reversión al código fuente.
  Así, la capa abstracta definida por las SyntaxTools (definida en el módulo *erl_syntax*) está muy bien estructurada con los tipos de los nodos independientes del contexto. Esta capa permite conectar de forma transparente los comentarios de código fuente y las anotaciones de usuario con los nodos del árbol. El uso de esta capa abstracta hace a las aplicaciones menos sensibles a los cambios en las estructuras de datos de *erl_parse*, y sólo requiere el módulo *erl_syntax* para estar actualizada.
    - En la librería ***erl_parse*** se definen las funciones básicas para realizar tareas de análisis sintáctico de Erlang, que convierte los distintos componentes léxicos en la forma abstracta, ya sean formularios (por ejemplo, construcciones de alto nivel), expresiones o términos. En el siguiente ejemplo se puede ver el uso de esta librería trabajando sobre la instrucción: `foo()-> "Hello!".`

```
1> erl_scan:string("foo()->\"Hello!\".")
   [{atom,1,foo},{'(',1},{')',1},{'->',1}{string,1,"Hello!"},
   {dot,1}]

2> erl_parse:parse_form([{atom,1,foo},{'(',1},{')',1},{'-
   >',1}, {string,1,"Hello!"}, {dot,1}]).
   {ok,{function,4,foo,0,[{clause,4,[],[],[{string,4,"Hello!"
   }]}]}}
```

      Se puede observar como el módulo *erl_scan* de la librería estándar se usa para la conversión de código en los distintos *tokens* (o componentes léxicos), y que el resultado de la llamada a la función `parse_form` del módulo *erl_parse* devuelve la forma abstracta de dichos *tokens*.

- Las capacidades que ofrece Erlang en cuanto a su flexibilidad a la hora de manipular código en tiempo de ejecución (o metaprogramación) queda mermada por la densidad

y complejidad de los módulos que las contienen. Es por este motivo por el que se decidió usar el módulo ***SMERL*** *(Simple Metaprogramming for Erlang)*, librería que facilita y abstrae el uso de las librerías relacionadas con el área de la metaprogramación en Erlang. El módulo SMERL, desarrollado por Yariv Sadan y publicado en su blog sobre Erlang [13], permite tanto la creación de nuevos módulos como la manipulación de los existentes en tiempo de ejecución. A continuación mostramos un pequeño ejemplo que ilustra la facilidad con la que se puede crear un módulo usando el módulo SMERL:

```
3>    test_smerl()-> % Creamos el modulo y le añadimos la función bar()
              C1 = smerl:new(foo),
              {ok, C2} = smerl:add_func(C1, "bar() -> 1 + 1."),
              smerl:compile(C2),
              foo:bar().                % devuelve 2
```

En este proyecto se ha hecho uso de este módulo sobretodo en la lectura/escritura de ficheros y su transformación en el Abstract Format, como se detallará en el apartado de implementación (Capítulo 3).

- El ***Abstract Format*** es la representación estándar de los árboles sintácticos de programas Erlang como términos Erlang. Para aplicar las refactorizaciones en este proyecto se ha decidido trabajar directamente sobre el Abstract Format del código (previa transformación del mismo al Abstract Format mediante el uso de las *syntax_tools* y el módulo *Smerl*). En el apéndice A del presente documento se puede encontrar la descripción detallada de esta representación.
  A continuación se muestra un pequeño ejemplo de la correspondencia entre un fragmento de código y su representación en la Abstract Format para ilustrar de forma más clara el formato de este tipo de representación.

Código

```
-module(example).

-export([foo/0]).

foo() -> "Hello!".
```

Abstract Format

```
[{attribute,1,module,example},
 {attribute,3,export,[{foo/0}]},
 {function,5,foo,0,[{clause,5,[],[],
           [{string,5,"Hello!"}]}]}]
```

Como se puede observar, el Abstract Format está compuesto por una lista de formularios (*forms*), que son tuplas que representan las construcciones de alto nivel como declaraciones de función o atributos. Así, por ejemplo `{attribute,1,module,example1}` seria un form compuesto por tupla que representa a la declaración de módulo `-module(example1)`. Estos forms se generan agrupando y interpretando los componentes léxicos escaneados del código fuente mediante el uso del módulo `erl_scan`.

En la figura 2.2 se muestra un pequeño esquema en el que se ilustran de forma más gráfica los pasos que se siguen en el proceso de refactorización de código en nuestra herramienta.

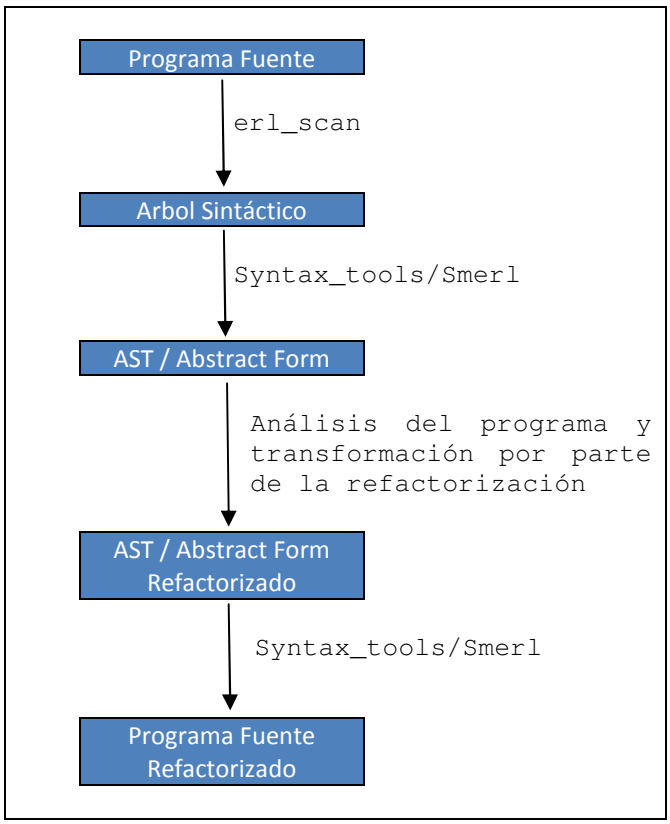

Figura 2.2: Esquema de los pasos seguidos en la refactorización

En el proceso de estudio de las diferentes herramientas de refactorización actuales de Erlang se ha observado que se pone especial hincapié en lo referente a las **anotaciones del código y la preservación del *layout* (o apariencia)** del mismo en la medida de lo posible para presentar al usuario un código lo más parecido posible al original (exceptuando las transformaciones). Actualmente entre las múltiples funciones que proporciona la librería `syntax_tools` de Erlang para la impresión de código mediante el módulo `prettypr` no se contempla la preservación del layout en cuanto a los comentarios se refiere, ya que por ejemplo, sólo se guarda la línea del mismo sin incluir información relativa a la columna que estos ocupan. Las

distintas aproximaciones estudiadas adoptan diversas soluciones interesantes como por ejemplo la adoptada por la herramienta Wrangler que exponemos a continuación.

Wrangler opta por extender cada nodo del AST producido por las `syntax_tools` con mas información (como posiciones en el código de variables, funciones, espacios en blanco, paréntesis y comentarios o información semántica que facilita la implementación de algunas transformaciones), obteniendo así un Árbol de Sintaxis Abstracta Anotado *(AAST, Annotated Abstract Syntax Tree)* y usando una base de datos relacional MySQL para su almacenamiento y manipulación. En la figura 2.3 se muestra la arquitectura de dicha aproximación:

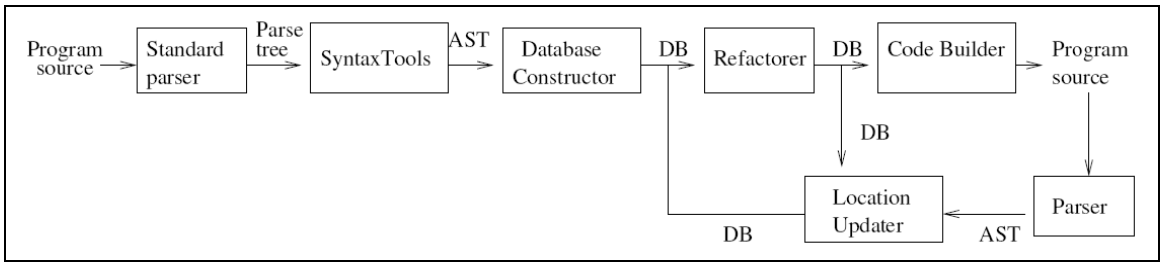

Figura 2.3: Arquitectura de la aproximación de la herramienta Wrangler

En el caso de este proyecto, por simplicidad (ya que extender el AST es una tarea bastante costosa) y al no considerarse como uno de los objetivos principales subyacentes al caso de estudio del mismo, ha sido omitida la inclusión de los comentarios y la preservación de la apariencia del código. El código resultante después de la aplicación de las distintas transformaciones se presenta como una traducción inmediata del Abstract Form que se obtiene después de la aplicación de las mismas. De esta forma se presenta al usuario un código que difiere del código original y que presenta cierta dificultad a la hora de la lectura.

2.2. INTERFAZ

Uno de los puntos más importantes que se presentó en la fase de diseño de esta herramienta fue el momento de la elección de la interfaz con el usuario. Después de contemplar distintas opciones se eligió integrar esta herramienta en el entorno de programación Eclipse basándose en la estructura proporcionada por la herramienta de refactorización Wrangler. Todos los detalles de la implementación de esta integración se pueden encontrar en el capítulo 4 de este documento, no obstante en este punto se va a especificar la arquitectura seguida para el desarrollo de dicha integración.

A continuación se describen brevemente los distintos componentes implicados en la capa de la interacción con el usuario.

- El entorno de programación (IDE) en el que se ha integrado esta herramienta es **Eclipse**, uno de los más utilizados actualmente con soporte multi-lenguaje, multitud de extensiones, utilidades y facilidades para el desarrollador.
- **Erlide** es el entorno de desarrollo para Eclipse, implementado como un plug-in. Proporciona numerosas características para facilitar al desarrollador la tarea de codificación en Erlang, como por ejemplo editor con remarcado sintáctico, herramienta de depuración, consola integrada, herramienta de generación automática o predicción de código.
- Esta herramienta se ha basado en la herramienta de refactorización **Wrangler**, sobre la que se han integrado las distintas refactorizaciones desarrolladas, eliminando las que ya tenía implementada la herramienta Wrangler. De esta forma se ha logrado simplificar enormemente el esfuerzo dedicado a la integración de esta herramienta en el entorno de desarrollo Eclipse.

En la figura 2.4 se ilustra cómo están distribuidos los componentes recién descritos en la comunicación con el usuario.

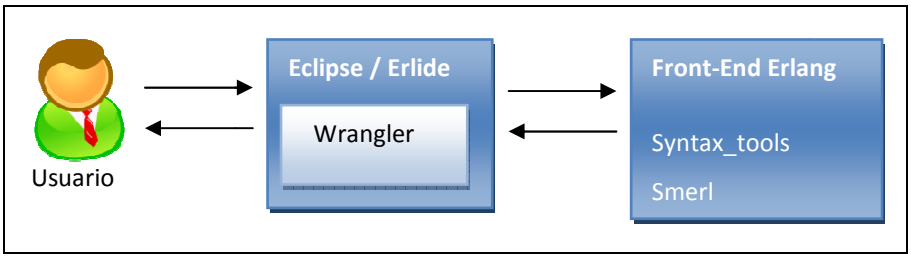

Figura 2.4: Distribución de los componentes de la interfaz

Cabe destacar que el desarrollo de la interfaz de este proyecto se ha centrado básicamente en adaptar el plug-in de Wrangler a nuestra herramienta de refactorización, como se podrá ver en el siguiente capítulo del documento (4. Integración en Eclipse).

3. IMPLEMENTACIÓN

Este punto del presente documento se va a centrar en el análisis de la implementación de las distintas refactorizaciones contempladas en la herramienta desarrollada. No se va a tener en cuenta la implementación de la integración en el entorno de programación Eclipse, sino que únicamente se va a detallar la estrategia adoptada para el desarrollo de dichas transformaciones, tanto a nivel general de la herramienta como particular de cada transformación.

3.1. ESTRUCTURA GENERAL DE LAS TRANSFORMACIONES

Todas las transformaciones soportadas por esta herramienta se han desarrollado partiendo de una misma estructura general. A continuación se detalla las principales componentes y características de esta estructura, para poder así entender con facilidad la implementación de cada uno de los refactorings que se explicaran posteriormente.

- Como se ha descrito en el apartado del diseño, el front-end de nuestra herramienta es la encargada de transformar el código fuente Erlang del fichero en cuestión a su *Abstract Format* subyacente del AST. Esta transformación se ha realizado mediante el uso de la librería estándar *syntax_tools* de la distribución oficial Erlang/OTP. La complejidad y densidad asociada a esta librería ha sido abstraída por el módulo *Smerl*. En todas nuestras transformaciones se ha hecho uso de este módulo para realizar la mencionada transformación al Abstract Format y su uso se ilustra a continuación:

```
{ok,Meta_model} = smerl:for_file(NombreFichero),
Forms=smerl:get_forms(Meta_model)
```

  En la primera instrucción se invoca a la función `for_file` del módulo *Smerl*, que se encarga de devolver (si todo ha ido bien) una tupla con un átomo `ok` indicando que la llamada devuelve un resultado correcto y el meta modelo asociado al módulo Erlang cuyo nombre se le ha pasado como argumento (`Meta_model`). Este meta modelo contiene el Abstract Format así como información (nombre, ruta…) del módulo en cuestión. La segunda instrucción se encarga de extraer la lista de *forms* que conforman el Abstract Format del módulo.

  En el caso contrario, una vez se ha aplicado la refactorización pertinente al Abstract Format se ha de volver a traducir al formato de código fuente, este proceso se realiza también de forma muy sencilla mediante el uso del módulo Smerl de la siguiente forma:

```
TransformedAbstractForm = applyRefac(Forms),
MetaModelNew =
  smerl:set_forms(Meta_model,lists:reverse(TransformedAbstractForm),
smerl:to_src(MetaModelNew,FileName),
```

La variable `TransformedAbstractForm` almacena el Abstract Format con la refactorización aplicada (`applyRefac(Forms)`). La función `set_forms` sustituye el nuevo Abstract Format en el meta modelo inicial (`Meta_model`). Por último, se realiza la traducción de dicho meta modelo al código fuente y almacenándolo en el archivo `FileName` mediante la instrucción `smerl:to_src(MetaModelNew,FileName)`.

A continuación se va a explicar la estructura general del Abstract Format de un programa, aunque este se puede ver con más detalle en el apéndice A de este documento.

- El Abstract Format de un **módulo** está compuesto por una secuencia (lista) de *forms.*
- Los ***forms*** pueden ser a su vez declaraciones de función o atributos
- Los **atributos** pueden ser por ejemplo las funciones a exportar/importar del módulo, la propia declaración del módulo…etc.
- Las **declaraciones de función** están compuestas por clausulas de función.
- Las **clausulas de función** tienen unos patrones, unas guardas y un cuerpo.
- El **cuerpo** (Body) de las clausulas de función están formadas por una lista de **expresiones**
- Los **patrones y guardas de una clausula** de función están compuestos por un subconjunto de las expresiones.

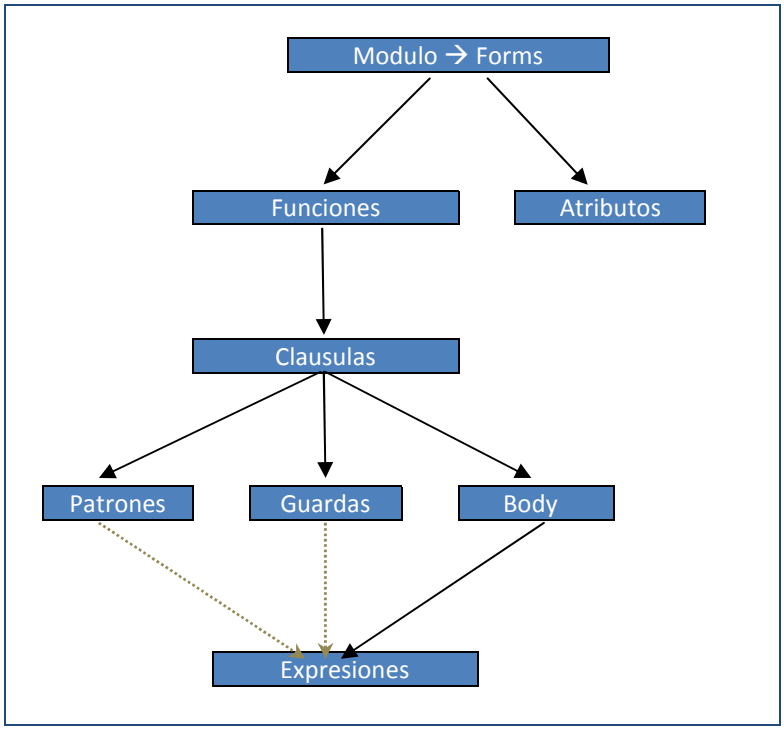

Figura 3.1: Esquema de los componentes del Abstract Format

En la figura 3.1 se puede observar como los patrones y las guardas están relacionadas con una flecha discontinua con las expresiones. Esto es por el hecho de que los patrones y las guardas estén compuestos por un subconjunto de las expresiones, es decir en este tipo de construcciones no se admiten ciertas expresiones que, por el contrario, sí que se aceptan en el cuerpo de una clausula de función. No obstante, en el apéndice A se puede ver como se consideran por separado los tres casos.

Una vez entendida la estructura seguida por el Abstract Format para representar el Árbol sintáctico de un programa Erlang se va a mostrar y analizar el código de la implementación de una transformación. Cabe destacar que esta transformación no se trata de ningún refactoring, ya que lo único que realiza es añadir el carácter 'x' al final del nombre de todas las variables existentes. La única finalidad de esta transformación es mostrar un ejemplo para entender la estructura del Abstract Format sobre la cual se ha trabajado en la implementación de los distintos refactorings de esta herramienta.

```
transform(Program)->
     {ok,Meta_Model} = smerl:for_file(Program),
     Forms=smerl:get_forms(Meta_Model),
     NXX = smerl:set_forms(
           Meta_Model,lists:reverse(change_varName_funsGeneric(Forms))),
     smerl:to_src(NXX,FileName),


%%************************************************************************
%%*************   change_varName_funsGeneric
%%************************************************************************
change_varName_funsGeneric([])->[];
change_varName_funsGeneric([{function,LINE,Name,Arity,Clauses}|Funs]) ->
     NewClauses = change_varName_clausesGeneric(Clauses),
     [{function,LINE,Name,Arity,NewClauses}|change_varName_funsGeneric(Funs)];
change_varName_funsGeneric(Other)->
     Other.


%%************************************************************************
%%*************   change_varName_clausesGeneric
%%************************************************************************
change_varName_clausesGeneric([]) -> [];
change_varName_clausesGeneric([{clause,LINE,Patterns,Guards,Body}|Clauses])->
     NBody = change_varName_expressionsGeneric(Body),
     NPatterns = change_varName_expressionsGeneric(Patterns),
     NGuards = change_varName_expressionsGeneric(Guards),
      [{clause,LINE,NPatterns,NGuards,NBody}|change_varName_clausesGeneric(
          Clauses)];
change_varName_clausesGeneric(Other)->
     Other.


%%************************************************************************
%%*************   change_varName_expressionsGeneric
%%************************************************************************
change_varName_expressionsGeneric([])->[];
change_varName_expressionsGeneric([Exp|Exps]) ->
     [change_varName_expressionGeneric(Exp)|change_varName_expressionsGeneric
(Exps)];
change_varName_expressionsGeneric(Other)->
     change_varName_expressionGeneric(Other).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%   change_varName_expressionGeneric
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
change_varName_expressionGeneric({var,LINE,V}) ->
     {var,LINE,list_to_atom(lists:append(atom_to_list(V),"x"))};
change_varName_expressionGeneric({match,LINE,E1,E2})->
     {match,LINE,change_varName_expressionsGeneric(E1),
      change_varName_expressionsGeneric(E2)};
change_varName_expressionGeneric({tuple,LINE,Exps})->
     {tuple,LINE,change_varName_expressionsGeneric(Exps)};
change_varName_expressionGeneric({cons,LINE,E1,E2})->
     {cons,LINE,change_varName_expressionsGeneric(E1),
      change_varName_expressionsGeneric(E2)};
change_varName_expressionGeneric({op,LINE,Op,E1,E2})->
     {op,LINE,Op,change_varName_expressionsGeneric(E1),
      change_varName_expressionsGeneric(E2)};
change_varName_expressionGeneric({op,LINE,Op,E})->
     {op,LINE,Op,change_varName_expressionsGeneric(E)};

                                        .
                                        .
                                        .
                                        .
                                        .

change_varName_expressionGeneric(Other)->
             Other.
```

Destacar que no se han puesto todas las expresiones que se deberían tratar en el ejemplo por simplicidad del mismo, sólo se han puesto las primeras, que incluyen las que realizan la transformación deseada. Con la ayuda de este ejemplo se puede observar el recorrido que se realiza sobre todo el Abstract Format de forma recursiva, desde los *forms* hasta todas las posibles expresiones que puedan aparecer. Si nos fijamos, solo se ajustan los patrones de aquellas expresiones, *forms* y clausulas que tienen a su vez expresiones o clausulas en su interior, por ejemplo en el caso de las tuplas sería:

```
change_varName_expressionGeneric({tuple,LINE,Exps})->
     {tuple,LINE,change_varName_expressionsGeneric(Exps)};
```

El resto se dejan tal cual (por ejemplo, en el caso de las expresiones, la guarda:

```
change_varName_expressionGeneric(Other)->Other).
```

También remarcar que en la estructura general adoptada no se ha hecho ningún tipo de diferenciación entre los conjuntos de expresiones pertenecientes a las guardas, patrones y cuerpo de las clausulas de función. Esto viene dado porque el proceso de *parseo* (o traducción) realizado por el Front-End mediante las *syntax_tools* nos asegura que el Abstract Format que se recibe a la hora de aplicar dicha transformación es sintácticamente correcto, con lo que nunca se va a producir una violación en cuanto al dominio de dichos conjuntos se refiere.

En el ámbito del ejemplo, se ha remarcado en color amarillo la instrucción concreta que realiza la transformación del programa analizado, es decir, modifica el nombre de las variables añadiéndoles el carácter 'x'.

A continuación mostramos un ejemplo de ejecución de esta transformación:

Programa Original

```
-module(prova2).

prova(A,B,C,D) ->
    E=A+B,
    F=lists:reverse(D),
    case C of
       true -> 0;
       false -> 1
    end.
```

Programa Transformado

```
-module(prova2).

prova(Ax, Bx, Cx, Dx) ->
    Ex = Ax + Bx,
    Fx = lists:reverse(Dx),
    case Cx of
       true -> 0;
       false -> 1
    end.
```

Observando que añade correctamente el carácter 'x' al final del nombre de todas las variables.

Una vez introducida la estructura básica que sigue la ejecución de las transformaciones de esta herramienta se va a pasar a explicar cada una de ellas con más detalle.

3.2. IMPLEMENTACIÓN DE LAS REFACTORIZACIONES

3.2.1 REFACTORING SOBRE LISTAS

En esta transformación se centra en la mejora de las estructura de datos de las listas. Así, se contemplan dos criterios:

a) En Erlang se puede añadir un elemento a una lista de diversas formas, dos de las cuales son:
   - Utilizando el constructor *[Head|Tail]* directamente, como `[1|[2,3,4]]`.
   - Utilizando el operador `++`, como por ejemplo `[1]++[2,3,4]`.

Las dos variantes producen el mismo resultado (`[1,2,3,4]`), pero (como argumentan Francesco Cesarini y Simon Thompson en [10] pag. 27) el operador `++` es menos eficiente y puede provocar que los programas se ejecuten de forma sustancialmente más lenta. De esta forma, cuando se quiere añadir un solo elemento al principio de una lista, se debería usar siempre el constructor *[Head|Tail]* ya que es más eficiente.

b) Como afirma Konstantinos Sagonas en [3], las llamadas a las funciones *lists:append* y *lists:substract* se pueden representar mediante los operadores ++ y -- respectivamente. Este refactoring es trivial y su finalidad es hacer el código fuente más breve y concreto.

Por lo que respecta a la implementación de este refactoring se han tenido que contemplar los distintos casos posibles en cada criterio, aplicándolos sobre el caso de las expresiones pertinente:

- **Implementación criterio a)**

```
1) applyRefac_expressionGeneric({op,LINE,'++',{cons,_,E1,{nil,_}},E2})->
            {cons,LINE,applyRefac_expressionGeneric(E1),
                        applyRefac_expressionsGeneric(E2)};

2) applyRefac_expressionGeneric({op,_,'++',{nil,_},E2})->
            applyRefac_expressionsGeneric(E2);

3) applyRefac_expressionGeneric({op,_,'++',E1,{nil,_}})->
            applyRefac_expressionsGeneric(E1);

4) applyRefac_expressionGeneric({op,LINE,'++',{string,_,[C]},{string,_,E2}})->
            {string,LINE,[C|E2]};

5) applyRefac_expressionGeneric({op,LINE,'++',{string,_,""},{string,_,E2}})->
            {string,LINE,E2};

6)  applyRefac_expressionGeneric({op,LINE,'++',{string,_,E1},{string,_,""}})->
            {string,LINE,E1};
```

Se han contemplado 4 casos:

- En la primera cláusula se trata el caso en que se utilice el operador ++ con dos listas, la primera de las cuales tiene únicamente un elemento (`{cons,_,E1,{nil,_}}`), sustituyéndolo por el constructor *[Head|Tail]* (`{cons,LINE,Head,Tail}`) aplicándole las llamadas recursivas pertinentes.
- En la segunda y tercera cláusula se contempla el caso en que se utilice el operador ++ y uno de los dos argumentos sea la lista vacía (`{nil,_}`), devolviendo únicamente el argumento que no es dicha lista vacía.

- En la cuarta cláusula se trata el caso de que se use el operador ++ en dos strings, la primera de las cuales tiene únicamente un carácter, sustituyéndolo por el constructor *[Head|Tail]* (`{cons,LINE,[C|E2]}`).
- Las últimas cláusulas sustituyen el operador ++ aplicado sobre strings con un argumento vacio (`{string,_,""}`) por el argumento que no está vacío.

- **Implementación criterio b)**

```
1) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},
   {atom,_,append}},[{cons,_,E1,{nil,_}},E2]})->
        {cons,LINE,applyRefac_expressionGeneric(E1),
                   applyRefac_expressionsGeneric(E2)};

2) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},
   {atom,_,append}},[{nil,_},E2]})->
        applyRefac_expressionsGeneric(E2);

3) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},{atom,_
   ,append}},[E1,{nil,_}]})->
        applyRefac_expressionsGeneric(E1);

4) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},
   {atom,_,append}},[{string,_,[C]},{string,_,E2}]})->
        {string,LINE,[C|E2]};

5) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},
   {atom,_,append}},[{string,_,""},{string,_,E2}]})->
        {string,LINE,E2};

6) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},{atom,_
   ,append}},[{string,_,E1},{string,_,""}]})->
        {string,LINE,E1};

7) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},
   {atom,_,substract}},[{nil,_},E2]})->
         {nil,LINE}.

8) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},
   {atom,_,substract}},[E1,{nil,_}]})->
         applyRefac_expressionsGeneric(E1);

9) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},
   {atom,_,substract}},[{string,_,""},{string,_,E2}]})->
         {string,LINE,””};

10) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},{atom,
    _,substract}},[{string,_,E1},{string,_,""}]})->
        {string,LINE,E2};

11) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},
    {atom,_,substract}},[{cons,_,E1,E2},{cons,_,E3,E4}]})->
        ({op,LINE,'--',{cons,LINE,E1,E2},{cons,LINE,E3,E4}});

12) applyRefac_expressionGeneric({call,LINE,{remote,LINE,{atom,_,lists},
    {atom,_,substract}},[{string,_,E1},{string,_,E2}]})->
        ({op,LINE,'--',{string,LINE,E1},{string,LINE,E2}});
```

Las 6 primeras cláusulas tratan sobre la llamada a la función *append* mientras que las 6 últimas tratan sobre la llamada a la función *substract*:

- En la primera cláusula se contempla el caso de que se llame a la función *append* con el primer argumento un solo elemento, sustituyéndolo por el constructor de listas *[Head|Tail]* (`{cons,LINE,Head,Tail}`) aplicándole las llamadas recursivas pertinentes.
- La segunda y tercera cláusula se utilizan cuando se llama a la función *append* con uno de sus argumentos como lista vacía, simplificándola al argumento que no es dicha lista vacía.
- En la cuarta cláusula se sustituye la llamada a la función *append* sobre dos strings, la primera de las cuales es un solo carácter, con el constructor de la lista correspondiente *[Head|Tail]*.
- La quinta y sexta cláusula tratan el caso de que se llame a la función *append* con uno de los argumentos la cadena vacía y sustituyéndolo únicamente con el argumento que no es dicha cadena vacía.
- A partir de la séptima cláusula se trata el caso de las llamadas a la función *substract* del módulo *lists.* En la séptima y la octava clausula se trata el caso en que uno de los argumentos de la llamada sea la lista vacía. El caso de las cláusula s9 y 10 es idéntico pero para el caso de que los argumentos sean strings.
- Las cláusulas 11 y 12 sustituyen las llamadas a la función *substract* por el operador -- para el caso de listas y strings respectivamente.

A continuación mostramos un ejemplo de ejecución del refactoring en el que se aplican todos los posibles casos recién vistos:

Código original

```
listas() ->
       A=[a]++[b,c,d],
       B=[]++[1,2,3],
       C=[1,2,3]++[],
       D="a"++"bcd",
       E=""++"abc",
       F="abc"++"",
       G= lists:append([1],[2,3,4]),
       H= lists:append([],[1,2,3]),
       I= lists:append([1,2,3],[]),
       J= lists:append("a","bcd"),
       K= lists:append("","abc"),
       L= lists:append("abc",""),
       M= lists:substract([],[1,2,3]),
       N= lists:substract([1,2,3],[]),
       O= lists:substract("","abc"),
       P= lists:substract("abc",""),
       Q= lists:substract([1,2,3,4],[2,3]),
       R= lists:substract("abcd","bc").
```

Código refactorizado

```
listas() ->
    A = [a, b, c, d],
    B = [1, 2, 3],
    C = [1, 2, 3],
    D = "abcd",
    E = "abc",
    F = "abc",
    G = [1, 2, 3, 4],
    H = [1, 2, 3],
    I = [1, 2, 3],
    J = "abcd",
    K = "abc",
    L = "abc",
    M = [],
    N = [1, 2, 3],
    O = "",
    P = "abc",
    Q = [1, 2, 3, 4] -- [2, 3],
    R = "abcd" -- "bc".
```

3.2.2 Refactoring Sobre Declaraciones de Import

Esta refactorización está basada en afirmaciones que realizan tanto K. Sagonas y T.Avgerinos en su artículo [3] como S.Thompson y F.Cesarini en su libro [10] (pag. 42).

La directiva `-import(Module,[Function/Arity,…])` nos permite importar funciones de otros módulos e invocarlos localmente como si se tratara de funciones del módulo sobre el que estamos trabajando. El uso de la directiva `import` puede provocar que el código sea más difícil de entender. Si alguien intenta entenderlo puede que a primera vista interprete que la función externa importada es una función local, ya que no existe ningún tipo de diferenciación que indique que dicha función esta importada de otro módulo si no se fija en la directiva `import`. Es por eso que en la comunidad de Erlang se hace hincapié en el hecho de usar cuidadosamente dicha directiva.

Este refactoring se encarga de eliminar las directivas `import` presentes y sustituir todas las llamadas a las funciones importadas por llamadas al módulo externo en las que están declaradas. Es cierto que este refactoring es cuestión de gustos. Esto es porque su objetivo principal no es el de mejorar la eficiencia del código ni reducir su tamaño, sino que la finalidad que persigue es la de hacerlo más entendible dando claridad a la hora de interpretar qué llamadas a funciones son locales al módulo actual y cuales son llamadas externas. De esta forma el usuario puede escribir el código utilizando la forma que seguramente le sea más cómoda, y cuando finalice se desarrollo podría utilizar este refactoring para hacerlo más utilizable.

Pasamos a continuación a describir como se ha implementado este refactoring a partir del código:

```
refac_Import(FileName,Editor) ->
       {ok,Meta_mod} = smerl:for_file(FileName),
       Forms=smerl:get_forms(Meta_mod),
       Imports=getImports(Forms),
       Forms_ImportDeleted=deleteImports(Forms),
       Meta_mod_new                =              smerl:set_forms(X,lists:reverse(
                   applyRefacImportsFun(Forms_ImportDeleted,Imports))),
       smerl:to_src(NXX, Meta_mod_new),


getImports([])->[];
getImports([{attribute,_,import,{Mod,Funcs}}|Attributes]) ->
       [{Mod,Funcs}|getImports(Attributes)];
getImports([_|Attributes]) ->
       getImports(Attributes).

deleteImports([])->[];
deleteImports([{attribute,_,import,_}|Attributes]) ->
       deleteImports(Attributes);
deleteImports([Head|Attributes])->
       [Head|deleteImports(Attributes)].


findImport(_,[]) -> false;
findImport(Func,[{Mod,Functions}|Modules])->
       case lists:member(Func,Functions) of
              true -> {ok,Mod};
              false -> findImport(Func,Modules)
       end.

                                          .
                                          .
                                          .
                                          .
%%************************************************************************
applyRefac_expressionGeneric({call,LINE,{remote,LINE,EM,EF},Exps},Imports)->
       {call,LINE,{remote,LINE,applyRefac_expressionsGeneric(EM,Imports),
                               applyRefac_expressionsGeneric(EF,Imports)},
                               applyRefac_expressionsGeneric(Exps,Imports)};

applyRefac_expressionGeneric({call,LINE,EF,Exps},Imports)->
       {Var,_,Func}=EF,
       case findImport({Func,length(Exps)},Imports) of
              {ok,Module} -> {call,LINE,{remote,LINE,{Var,LINE,Module},
                                   applyRefac_expressionGeneric(EF,Imports)},
                                   applyRefac_expressionsGeneric(Exps,Imports)};
              false -> {call,LINE,applyRefac_expressionsGeneric(EF,Imports),
                                   applyRefac_expressionsGeneric(Exps,Imports)}
       end;
%%************************************************************************
                                          …
                                          …
```

En esta implementación se ha tenido que guardar en una lista todas la funciones importadas de módulos externos mediante la instrucción `Imports=getImports(Forms)` pasándole el meta modelo inicial como argumento. Esta función recorre el Abstract Format en busca de la directiva `import` y cuando la encuentra devuelve una lista con todas las funciones importadas. Cabe destacar que esta lista se ha tenido que pasar como nuevo argumento a todas la funciones implicadas en el recorrido del Abstract Format del refactoring para en el caso de encontrarse con una llamada a una de las funciones presentes en dicha lista poder realizar la transformación pertinente.

La instrucción `Forms_ImportDeleted=deleteImports(Forms)` se encarga de devolver el Abstract Format que se le pasa como argumento pero eliminando la directiva `import` (previo guardado de la lista de funciones presente en esta).

La función `findImport` se encarga de comprobar que la función que se le pasa como primer argumento está o no presente en la lista de funciones importadas que se le pasa como segundo argumento, devolviendo así una tupla con un átomo `ok` y el nombre del módulo al que pertenece dicha función (`{ok,Mod}`) en caso de ser una de las que estaba en la lista de imports o un `false` en caso contrario. De esta forma cuando estamos recorriendo el Abstract Format (concretamente en la función `applyRefac_expressionGeneric`, tal y como se ve en la parte final del código anterior), si nos encontramos con una llamada a función local (`{call,LINE,EF,Exps}`) pasaremos a comprobar la presencia de dicha función en la lista de imports que se le pasa como nuevo argumento. En el caso de estar presente se sustituye dicha llamada local por una llamada remota a la función del módulo que nos ha devuelto la función `findImport` como parte del resultado, en caso contrario la llamada se deja igual que estaba ya que no es una de las funciones externas importadas.

A continuación mostramos un ejemplo de aplicación de esta refactorización:

Código original

```
-import(lists,
       [reverse/1, nth/2, sort/1]).

test() ->
    lists:append([1], [2, 3]),
    fun_local(),
    reverse("prova"),
    sort([1,2, 3]),
    X=nth(2,[1,2, 3]).

fun_local() -> true.
```

Código refactorizado

```
Test () ->
    lists:append([1], [2, 3]),
    fun_local(),
    lists:reverse("prova"),
    lists:sort([1, 2, 3]),
    X = lists:nth(2, [1, 2, 3]).

fun_local() -> true.
```

Como era de esperar, ha sido eliminada la directiva `import` del código y las llamadas a las funciones externas que estaban en la directiva han sido sustituidas por llamadas externas al módulo correspondiente.

3.2.3 REFACTORING SOBRE FUNCIONES CON UNA SOLA INSTRUCCIÓN CASE

S.Thompson y F.Cesarini en su libro [10] (pags. 47,48) afirman que las funciones que tienen una sola instrucción *case* se pueden reescribir como distintas guardas de la misma función, en las que se añadiría como argumento la variable o expresión sobre la que se codificaba la instrucción *case*. Esto es porque se considera que el ajuste de patrones o *pattern matching* (que se realizaría en las guardas de la función transformada) en las definiciones de funciones puede ser más compacto que el uso de una expresión case en su cuerpo.

De esta forma, la funcionalidad de este refactoring es la de sustituir aquellas funciones que tienen una sola instrucción *case* en su interior por una función con tantas guardas como casos tenia dicha instrucción, dejando en el cuerpo de cada guarda de la función las instrucciones que pertenecían a cada caso de la instrucción *case*.

Respecto a la implementación de este refactoring se va a detallar a partir del código:

```
applyRefacCaseFun([])->[];
applyRefacCaseFun([{function,LINE_FUN,Name,Arity,Clauses}|Funs]) ->
       [{function,LINE_FUN,Name,Arity,applyRefac_clausesGeneric(
              applyRefacCaseFunRecursive(Clauses))}|applyRefacCaseFun(Funs)];
applyRefacCaseFun(Other)->
       Other.



applyRefacCaseFunRecursive([])->[];
applyRefacCaseFunRecursive([{clause,LINE_CLAUSE,Patterns,Guards,[{'case',LINE,
                                  CASE,VarCase,ClausesCase}]}|ClausesTail])->
       case VarCase of
              [{_,_,_}] ->
                  NewClauses=fusionaClauses(Patterns,VarCase,ClausesCase),
                  applyRefac_clausesGeneric(NewClauses) ++
                       applyRefacCaseFunRecursive(ClausesTail);
              {_,_,_} ->
                  NewClauses=fusionaClauses(Patterns,VarCase,ClausesCase),
                  applyRefac_clausesGeneric(NewClauses)++
                       applyRefacCaseFunRecursive(ClausesTail);
              _ ->   [{clause,LINE_CLAUSE,Patterns,Guards,
                           [{'case',LINE_CASE,VarCase,ClausesCase}]}] ++
                               applyRefacCaseFunRecursive(ClausesTail)
       end;
applyRefacCaseFunRecursive([{clause,LINE_CLAUSE,Patterns,Guards,Body}|
                                 ClausesTail])->
       [{clause,LINE_CLAUSE,Patterns,Guards,Body}|
                                    applyRefacCaseFunRecursive(ClausesTail)].



fusionaClauses(ArgsFunc,VarCase,ClausesCase) ->
       NewPatterns=marcaPatterns(ArgsFunc,VarCase),
       addPatterns(NewPatterns,ClausesCase).



marcaPatterns([],_)->[];
marcaPatterns(Args,[])->Args;
marcaPatterns([{Tipo,_,Valor}|ArgTail],{Tipo,_,Valor})->
       [{Tipo,x,Valor}|ArgTail]; %Marcamos el argumento a sustituir
marcaPatterns([ArgHead|ArgTail],VarCase)->
       [ArgHead|marcaPatterns(ArgTail,VarCase)].
```

```
addPatterns(_,[]) -> [];
addPatterns(Args,[{clause,LINE_CLAUSE,Patterns,Guards,Body}|ClauseTail]) ->
       [{clause,LINE_CLAUSE,addPatternRecurs(Args,Patterns),Guards,Body}|
                                                  addPatterns(Args,ClauseTail)].


addPatternRecurs([],_)->[];
addPatternRecurs([{_,x,_}|ArgsTail],Patterns)->Patterns++ArgsTail;
addPatternRecurs([Other|ArgsTail],Patterns)->
         [Other|addPatternRecurs(ArgsTail,Patterns)].
```

El código de la implementación de este refactoring es un poco más complicado que en los anteriores. Las principales características de esta implementación son:

- La función `applyRefacCaseFunRecursive` recorre las clausulas de función que se le pasan en busca de alguna que tenga una única instrucción case en su cuerpo. En caso de encontrarla realiza una llamada a la función `fusionaClauses`.

- La función `fusionaClauses` es la encargada de devolver las nuevas clausulas de la función fusionando las clausulas de la función original y las de la instrucción *case*. Para realizar esto realiza llamadas a las funciones `marcaPatterns` y `addPatterns`.

- La función `marcaPatterns` se encarga de marcar el argumento de la función original cuyo uso es el de la variable de la instrucción case. Este marcado del argumento nos servirá a continuación para saber cuál es el argumento a sustituir por cada una de las expresiones que conforman los distintos casos de la instrucción *case*. Para marcar dicho argumento se pone una x en el lugar que ocuparía la línea en la construcción del Abstract Format de este argumento, ya que como se ha mencionado en puntos anteriores, en esta herramienta no se preserva la apariencia del código original y, por lo tanto, se hace caso omiso de este atributo pudiéndose utilizar para guardar esta información.

- Para añadir la expresión de la guarda correspondiente de la instrucción case en el lugar que ocupaba la variable de dicha instrucción en los argumentos de la función original se hace uso de las funciones `addPatterns` y `addPatternRecurs`. Estas funciones se encargan de recorrer todas la clausulas que se le pasan como argumento sustituyendo en sus patrones la variable marcada con la x con la expresión de la guarda de la instrucción *case* correspondiente.

A continuación podemos observar que este código se comporta de la forma esperada mediante un ejemplo:

Código original

```
prueba(A,B,C,D) ->
    E=A+B,
    F=lists:reverse(D),
    case C of
      true -> 0;
      false -> 1
    end.

add_to_hs(a,b,c,d,e,f) -> true;
add_to_hs(b,c,d, T,e,f) ->
    case T of
      {Min} -> true;
      _ -> false
    end.

add_to_hs(a, _) -> true;
add_to_hs(T, b) ->
    case T of
      a -> true;
      _ -> false
    end.
```

Código refactorizado

```
prueba(A, B, C, D) ->
    E = A + B,
    F = lists:reverse(D),
    case C of
      true -> 0;
      false -> 1
    end.

add_to_hs(a, b, c, d, e, f) -> true;
add_to_hs(b, c, d, {Min}, e, f) -> true;
add_to_hs(b, c, d, _, e, f) -> false.

add_to_hs(a, _) -> true;
add_to_hs(a, b) -> true;
add_to_hs(_, b) -> false.
```

Se puede observar que en el caso de que la instrucción *case* no sea la única dentro de la función (como es el caso de la función `prueba`) no se aplica la refactorización. En los dos casos de la función `add_to_hs` se puede comprobar cómo se comporta conforme se espera, extrayendo la variable de la instrucción case como argumento de la nueva función en el lugar que les corresponde y sustituyendo el cuerpo de la función por la clausula de la instrucción case correspondiente.

4. INTEGRACIÓN EN ECLIPSE

Sin duda uno de las tareas más complicadas y a la que se ha dedicado más tiempo en este proyecto ha sido la de la integración de la herramienta en un entorno de desarrollo. Paralelamente al gran crecimiento que se ha producido en los últimos años en el número de desarrolladores, ha aumentado la popularidad de los entornos de programación integrados o (*IDE Integrated Development Environment*). Un IDE es un entorno de programación que ha sido empaquetado como un programa de aplicación, es decir, consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI). Los IDEs pueden ser aplicaciones por sí solas o pueden ser parte de aplicaciones existentes. Los IDE proporcionan un marco sencillo y amigable a la hora de desarrollar software al tener múltiples opciones y herramientas enfocadas a facilitar al programador su tarea de codificación.

En el momento de plantearse la **plataforma** sobre la que se iba desarrollar esta herramienta surgieron **distintas alternativas**, de la cuales se estudió la viabilidad de las siguientes:

- La opción más sencilla que se planteó fue la de implementar esta herramienta como un **programa de consola** en el que se especificaría mediante un comando el tipo de refactorización a aplicar y la ruta del programa a transformar. No obstante, esta opción fue descartada por ofrecer un entorno demasiado “plano” y con una interacción más bien pobre con el usuario, hecho que le restaría bastante utilidad y usabilidad a la herramienta.

- Otra opción que se planteó y que no se descartó hasta última hora fue la de la integración de esta herramienta en el entorno de programación **Emacs**. Emacs es un editor de texto altamente configurable con una herramienta de resaltado de sintaxis, interfaz de depuración y otros, pero (como su nombre indica "Editor de MACroS") es sólo un editor con funcionalidades adicionales. Cabe destacar que la integración es este entorno hubiese sido bastante más sencilla que la opción elegida.

- Finalmente la opción elegida fue el entorno de programación **Eclipse**, probablemente el mejor IDE de código abierto multiplataforma. Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas. Eclipse fue desarrollado originalmente por IBM y es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios. Eclipse tiene un sistema de plug-ins (o extensiones) para proporcionar toda su funcionalidad en la parte superior (e incluyendo) del sistema de ejecución, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Está escrito principalmente en Java y se puede utilizar para desarrollar aplicaciones en Java y, por medio de diversos plug-ins, se puede utilizar para desarrollar en otros idiomas, incluyendo Erlang mediante el plug-in Erlide.

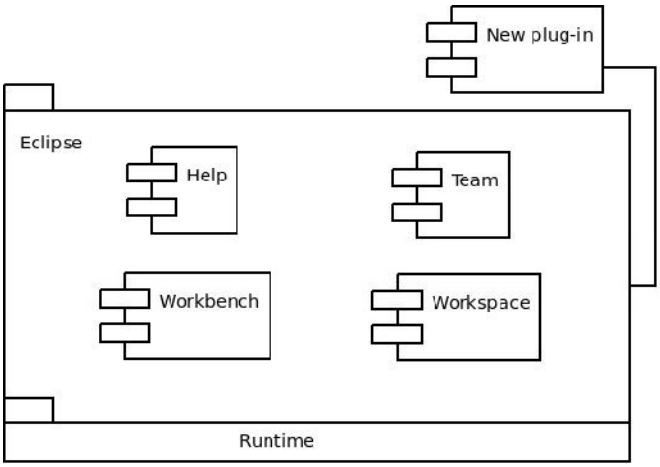

Figura 4.1: Arquitectura de Eclipse

El plug-in Erlide proporciona numerosas características para facilitar al desarrollador la tarea de codificación en Erlang, como por ejemplo editor con remarcado sintáctico, herramienta de depuración, consola integrada, herramienta de generación automática o predicción de código.

Ya en el marco de este proyecto, la decisión de integrar esta herramienta en Eclipse fue principalmente condicionada por el hecho de que se observó que se podría "personalizar" con relativa facilidad el plug-in de Wrangler para adaptarlo a las posibles refactorizaciones de esta herramienta. Wrangler es una conocida herramienta de refactorización de programas Erlang desarrollada como plug-in de Eclipse y sus principales características se han descrito en la sección 1.3 de este documento. Así, la tarea de integración de esta herramienta en Eclipse se ha basado básicamente en la adaptación del plug-in Wrangler a las distintas refactorizaciones desarrolladas.

A continuación se va a explicar el **proceso de adaptación del plug-in de Wrangler** a nuestras refactorizaciones.

- Después de descargar el código de este plug-in, ponerlo en ejecución y entender su funcionamiento se pasó a la depuración del mismo viendo instrucción a instrucción como ejecutaba los distintos refactorings. En este punto se analizó la distribución de los distintos archivos, paquetes y proyectos implicados, observando cuales eran los que se tendrían que modificar para poder añadir los refactorings desarrollados. De esta forma los dos prinicipales proyectos de eclipse implicados en el plug-in para nuestro interés son:

  - El proyecto básico del plug-in es *org.erlide.wrangler.refactoring* y es donde se encuentran la mayor parte de los archivos implicados en la ejecución del mismo. Este proyecto está escrito en Java en casi toda su totalidad , excepto algún archivo .xml de configuración de la herramienta. Está estructurado en 14 paquetes que agrupan cada uno archivos con funcionalidades distintas como por ejemplo las

clases abstractas (*org.erlide.wrangler.refactoring.core*), archivos que proporcionan dsitintas utilidades para el plug-in (*org.erlide.wrangler .refactoring.util*) o los archivos que implementan la interfaz de comunicación con los archivos Erlang (*org.erlide.wrangler.refactoring.backend*).

- El proyecto en el que se encuentran los archivos Erlang implicados en esta herramienta es *org.erlide.wrangler.core*. En este proyecto se encuentran las implementaciones de las distintas refactorizaciones además de otros archivos necesarios para poder llevar a cabo dicha ejecución.

- El siguiente paso fue modificar los archivos identificados en el paso anterior o crear las clases correspondientes para añadir los distintos refactorings. Dichos archivos son:

  - *org.erlide.wrangler.refactoring.ui.RefactoringHandler.java*: Clase que extiende a la clase Abstracta *org.eclipse.core.commands.AbstractHandler* del proyecto Eclipse. Su método *execute* recibe el evento de la pulsación de las distintas opciones del menú del plug-in. Dependiendo del identificador de dicho evento se crea una instancia de la clase que crea la infraestructura necesaria para lanzar el refactoring correspondiente ejecutándolo a continuación.

  - Se añaden las distintas clases que crean la infraestructura necesaria para poder lanzar el refactoring correspondiente dentro del paquete *org.erlide. wrangler.refactoring.core.internal*: Estas clases son *ImportRefactoring.java*, *ListaRefactoring.java* y *CaseFunRefactoring.java*. En esta clase se envía la llamada a procedimiento remoto (o *RPC, Remote Procedure Call*) correspondiente a la interfaz de Erlang invocando al módulo y función de Erlang correspondiente al refactoring en cuestión.

  - En el proyecto org.erlide.wrangler.core se han tenido que añadir los distintos ficheros Erlang que implementan las transformaciones, en este caso los archivos *refac_CaseFun.erl*, *refac_Import.erl* y *refac_Lista.erl* la implementación de los cuales se ha explicado con detalle en el punto 3 de este documento. Además se ha tenido que modificar el fichero *wrangler.erl* que es el que actúa como interfaz de comunicación entre los archivos Erlang de la implementación de las transformaciones y las RPC que se envían desde el proyecto *org.erlide.wrangler.refactoring*.

  - Finalmente se ha modificado el fichero plugin.xml del proyecto *org.erlide.wrangler.refactoring.* Este fichero es el que describe el plug-in y indica a Eclipse todo lo necesario para poder activarlo (menús, botones, posibles acciones…etc.). De esta forma se ha conseguido añadir correctamente los menús correspondientes a los nuevos refactorings implementados y enlazarlos correctamente al código de Wrangler.

- La última tarea que se realizó en esta integración fue la de quitar todos los refactorings de Wrangler para dejar solamente los implementados para nuestra herramienta. De esta forma se eliminaron numerosos archivos implicados en los refactorings de Wrangler y se eliminó gran parte de código de otros, como por ejemplo el archivo *RefactoringHandler.java* o *plugin.xml* consiguiendo reducir considerablemente el tamaño de estos proyectos y en consecuencia el tamaño del plug-in, dejando únicamente los necesarios para la correcta ejecución del mismo.

Tal y como se ha indicado al principio de este punto, la tarea de integración en este IDE ha requerido mucho esfuerzo. Esto ha sido provocado por el gran tamaño del proyecto de la herramienta de refactorización Wrangler, ya que ha sido necesario invertir bastante tiempo en entender la estructura del mismo, su modo de funcionamiento interno e identificar cuáles eran los ficheros que había que modificar para la correcta integración de nuestras transformaciones. De esta forma, en el desarrollo de esta tarea se han encontrado muchos obstáculos que ha habido que sortear, quedando algunos aspectos de la integración en el aire como por ejemplo el mensaje de confirmación al usuario indicando que el refactoring se ha realizado correctamente, ya que muestra de forma equivocada que el fichero sobre el que se ha aplicado la refactorización no se ha modificado. Cabe destacar que el principal objetivo que se pretendía alcanzar con el desarrollo de este proyecto era centrarse más en la implementación de los distintos refactorings para el lenguaje Erlang que el de una buena integración en el entorno de desarrollo. No obstante, se asume que con más esfuerzo y dedicación al estudio de la herramienta Wrangler se hubiera podido realizar una buena integración de esta herramienta.

5. MANUAL DE USUARIO

Este capítulo consiste en un manual de introducción a la herramienta de refactorización del presente documento, con la intención de ofrecer la información necesaria para la correcta instalación y uso de esta herramienta.

REQUISITOS

Para poder ejecutar esta herramienta es necesario:

- Tener instalado el Entorno de Desarrollo ***Eclipse***, que se puede descargar desde su página web http://www.eclipse.org, así como poseer conocimientos de su uso (se pueden consultar distintos tutoriales desde la misma página web de la descarga).
- Tener instalado el plug-in de Erlang para el entorno de desarrollo Eclipse, ***Erlide***. Desde su página web oficial http://erlide.sourceforge.net/ se puede descargar dicho plug-in así como guías de instalación y diversos tutoriales.

CARGA DEL PROYECTO

Una vez se han descargado y configurado los componentes de los requisitos, se ha de cargar el código fuente herramienta en el entorno de desarrollo Eclipse. Para realizar esto arrancamos el entorno de desarrollo Eclipse y seleccionamos la ruta donde tenemos ubicado el *workspace* que contiene los proyectos que conforman el código de esta herramienta tal y como se muestra en la figura 5.1

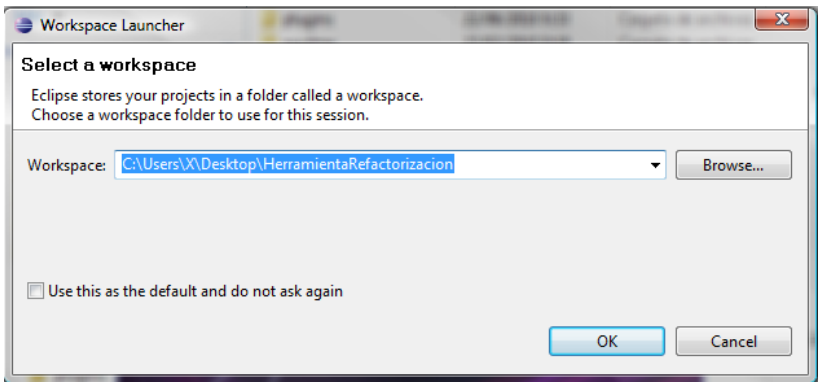

Figura 5.1 Selección del workspace a cargar

A continuación Eclipse se encargará de cargar dicho workspace con los distintos proyectos, quedando el entorno de desarrollo como se muestra en la figura 5.2

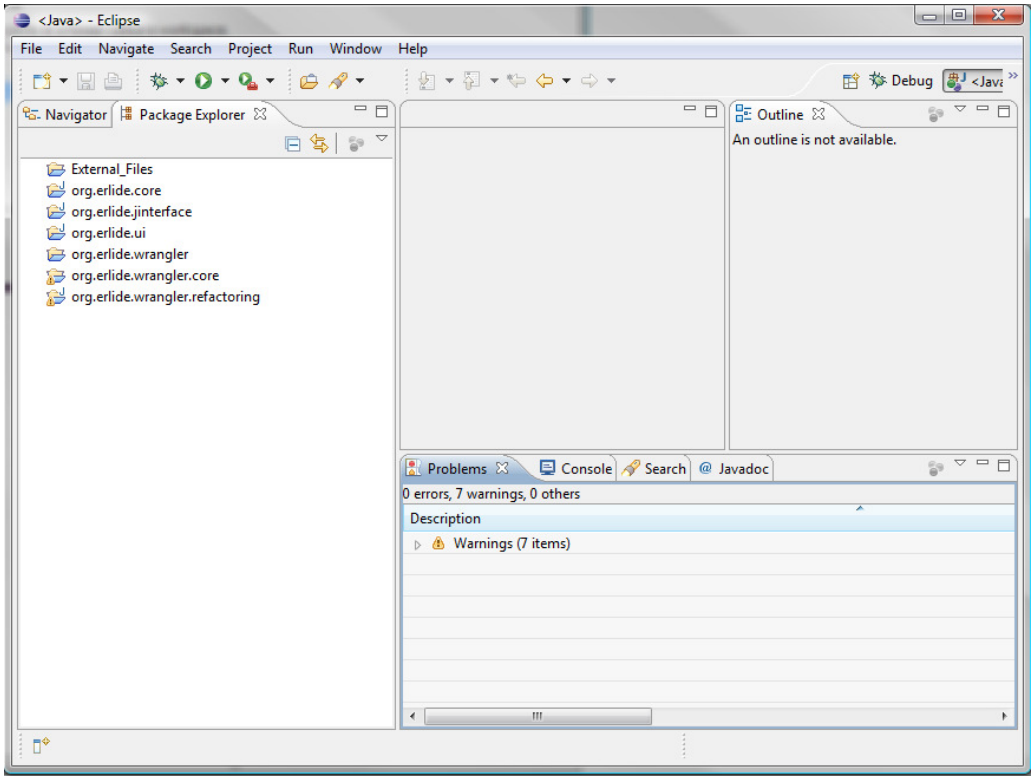

Figura 5.2: Worksape del código funte de la herramienta

En la ventana de la parte izquierda *Package Explorer* se puede observar cómo se han cargado los distintos proyectos que conforman el workspace del código fuente de esta herramienta. Y en la ventana *Problems* podemos comprobar que se han cargado correctamente, sin ningún error. En el caso que hayan aparecido errores es probable que se deba a una mala configuración del plug-in Erlide, consulte con las guias de instalación existentes en su página web http://erlide.sourceforge.net/.

EJECUCIÓN DEL PROYECTO

El siguiente paso a la carga del workspace, si todo ha ido correctamente, es el de la ejecución del mismo para poder empezar a usar esta herramienta. Para realizar esto nos hemos de dirigir al proyecto *org.erlide.wrangler.refactoring* en la ventana *Package Explorer* de la parte derecha del entorno de desarrollo pulsando sobre el mismo el botón derecho del ratón. En el menú contextual que aparece seleccionamos la opción *Run As – Eclipse Aplication.* Hay que destacar que esta acción solo es necesaria realizarla la primera vez que se va a poner en ejecución la herramienta, ya que a partir de ese momento se puede poner en ejecución pulsando simplemente sobre el botón de la barra superior.

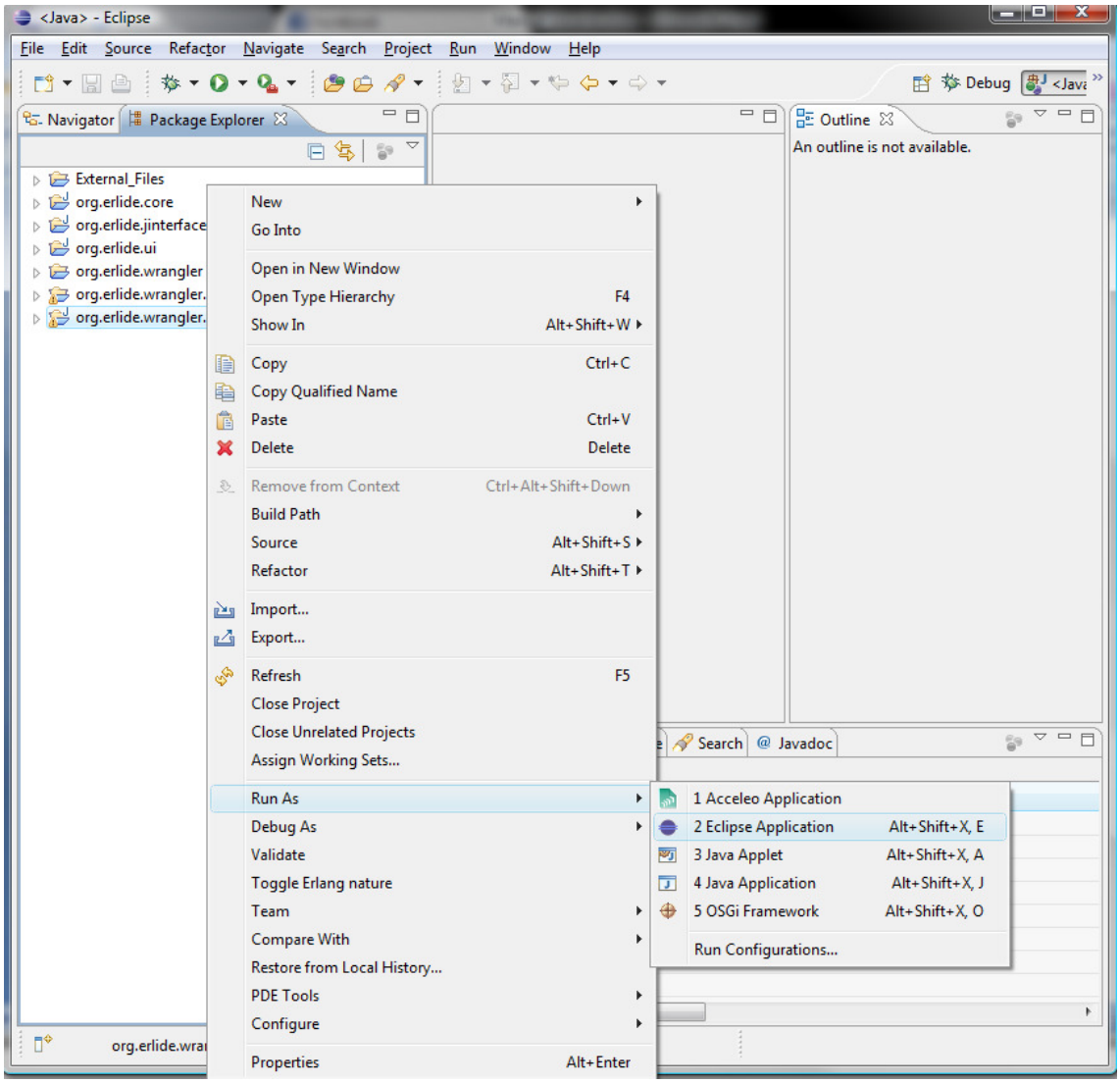

Figura 5.3: Ejecución de la herramienta

Podremos observar como en unos instantes se nos abre una nueva instancia del entorno de desarrollo Eclipse. Esta instancia se trata del entorno de programación que tiene el plug-in de esta herramienta ya cargado y listo para usarse.

USO DE LA HERRAMIENTA

Una vez que tenemos la instancia del Eclipse con el plug-in cargado aparecerá el entorno de desarrollo sin ningún proyecto cargado. Es necesario crear o importar un proyecto de Erlang para poder utilizar esta herramienta. A continuación se detallan las dos opciones para crear un nuevo proyecto:

- Nos dirigimos a la opción *New – Erlang Project* del menú *File* de la parte superior del entorno de desarrollo.

- A continuación introducimos el nombre del proyecto que queremos crear como en la figura 5.4

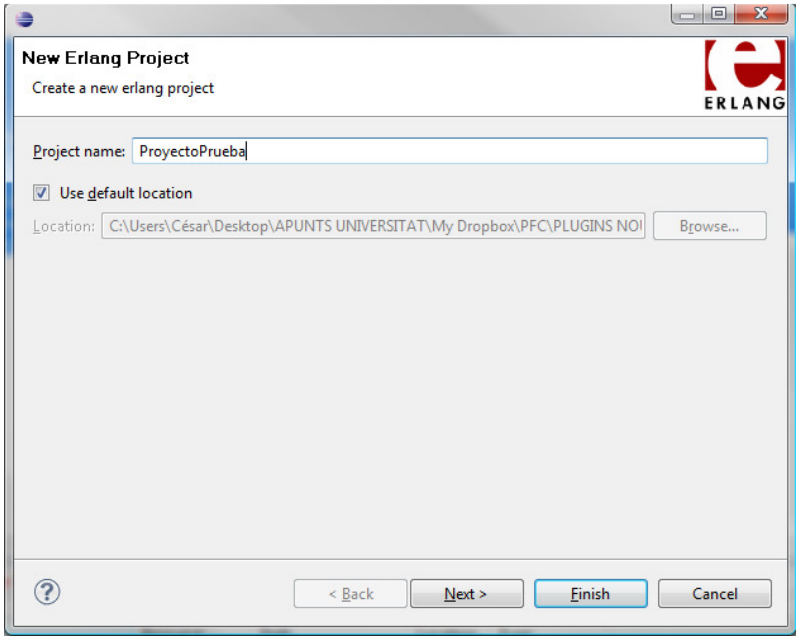

Figura 5.4: Creación de nuevo proyecto Erlang

- Podremos observar como se ha creado el nuevo proyecto con el nombre indicado mirando la ventana *Package Explorer.* El siguiente paso es añadir un nuevo módulo o importar uno existente. Para crear un nuevo módulo expandiremos el proyecto en la ventana *Package Explorer* pulsando botón derecho sobre la carpeta src. En el menú contextual que sale pulsamos sobre la opción *New Module*, rellenando a continuación el nombre del módulo en la ventana que saldrá tal y como se puede observar en la figura 5.5

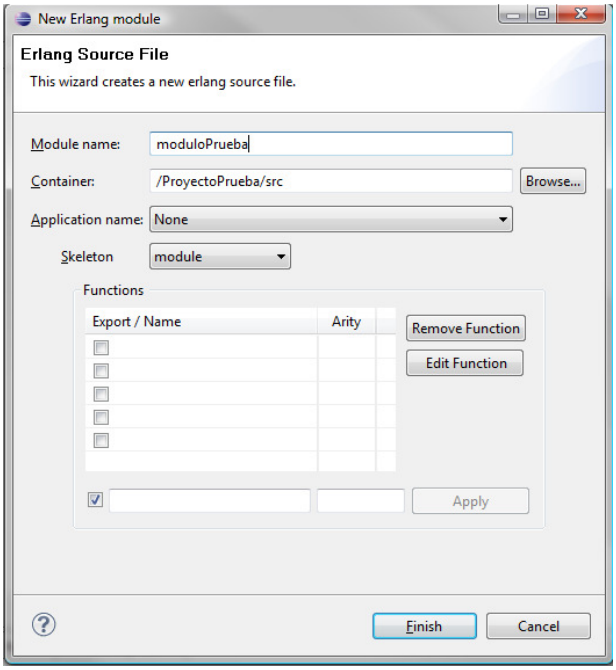

Figura 5.5: Creación de nuevo módulo Erlang

- El siguiente paso sería introducir código Erlang en el módulo recién creado en el caso de no haber importado ningún módulo existente.

- Para aplicar las distintas refactorizaciones de esta herramienta simplemente tendremos que abrir el módulo (para que la herramienta sepa sobre que módulo se va aplicar la refactorización) y nos dirigimos al menú *Refactor* de la barra superior del entorno de desarrollo, seleccionando a continuación el tipo de refactorización que se desea aplicar.

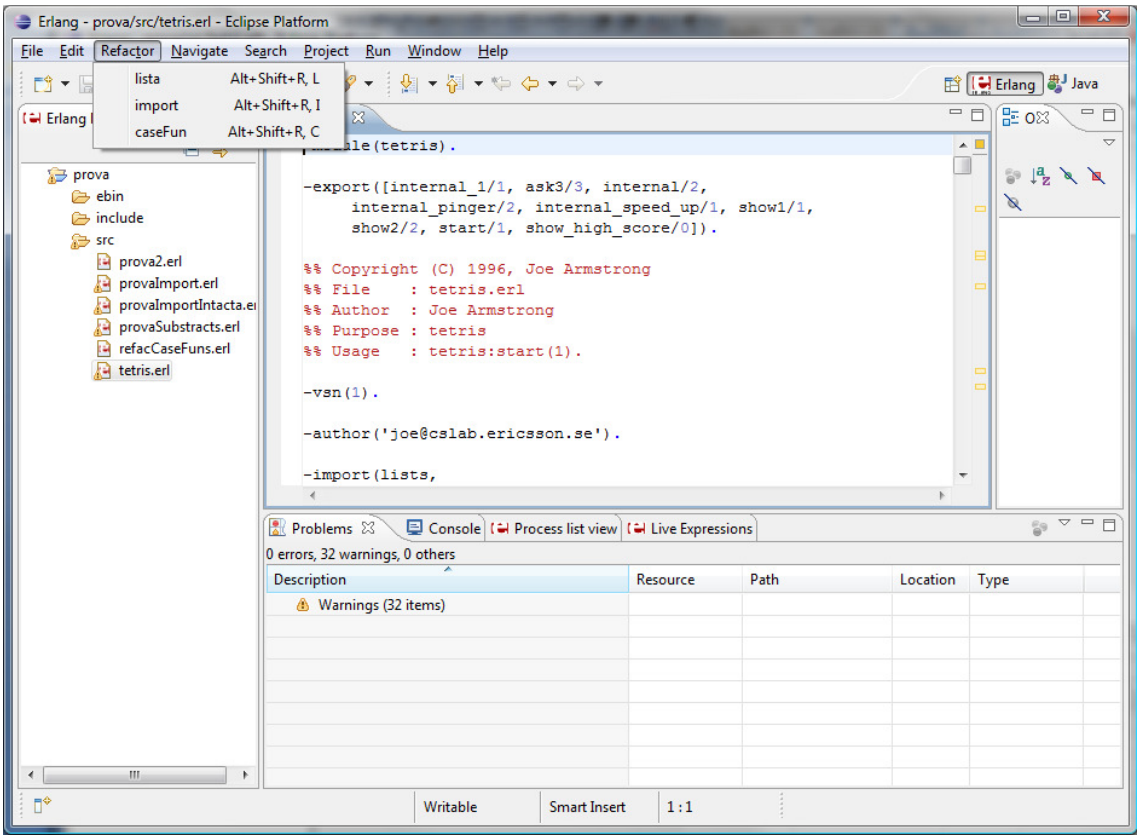

Figura 5.6: Menú de la herramienta integrado en el entorno de desarrollo

- Si la refactorización se ha podido llevar a cabo sin problemas se muestra a continuación un mensaje en el que se indica de forma errónea que la refactorización no ha modificado ningún fichero tal y como se muestra en la figura 5.7

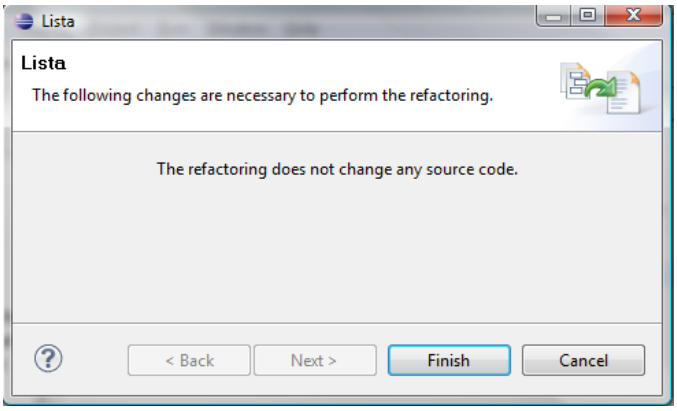

Figura 5.7: Mensaje de notificación erróneo

El usuario no debe tener en cuenta lo que explica este mensaje, ya que como se ha mencionado anteriormente el mensaje se muestra de forma errónea, con lo que al pulsar sobre el botón aceptar saldrá un nuevo mensaje indicando que si que se ha realizado el cambio:

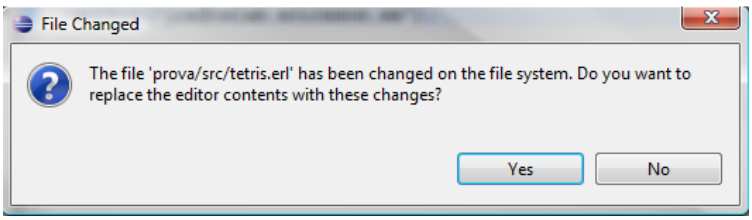

Figura 5.8: Mensaje indicando que el fichero si que ha cambiado

De esta forma, si se confirma este mensaje se podrá observar como el fichero sobre el que se ha aplicado la refactorización efectivamente ha cambiado, a no ser que la representación del mismo coincidiera con la que se muestra después de aplicar la refactorización. Este hecho es bastante improbable a no ser que se aplique sobre un fichero al que previamente se le ha aplicado ya la refactorización. Esto viene dado al no preservarse en esta herramienta la apariencia de los módulos sobre los que se aplican las distintas refactorizaciones.

CONCLUSIÓN

Los desarrolladores utilizan la técnica del refactoring para conseguir que su código sea más legible, estructurado e incluso eficiente. No obstante, la aplicación de esta técnica suele ser tediosa e incluso propensa a errores si se realiza a mano, es por este motivo porque se hace necesario el uso de herramientas de refactorización que reduzcan al desarrollador el esfuerzo dedicado a tal fin. De esta forma, en el ámbito del presente proyecto se ha desarrollado una de estas herramientas para la aplicación de la técnica en el lenguaje de programación Erlang.

Erlang fue desarrollado para resolver requerimientos de sistemas distribuidos, tolerantes a fallos, masivamente concurrentes y de tiempo real. Actualmente Erlang representa el uso comercial del lenguaje de programación funcional más importante fuera del ámbito académico.

Posiblemente, la característica principal del lenguaje Erlang es su orientación a la concurrencia, gracias a su pequeño pero potente conjunto de primitivas para crear y comunicar procesos, que consigue dotarlo de gran eficiencia en cuanto al manejo de los mismos se refiere. Este hecho ha propiciado que su uso y popularidad en la comunidad de programadores haya experimentado un aumento más que considerable en los últimos años, pudiendo encontrar sistemas de gran envergadura (por ejemplo el chat de Facebook) implementados en dicho lenguaje.

No obstante, el uso de las herramientas de refactorización va íntimamente ligado a la usabilidad que le proporciona su integración en un buen entorno de desarrollo. Por este motivo, este aspecto es uno de los que más hincapié se ha puesto en el desarrollo de este proyecto. El resultado ha sido la integración de esta herramienta en el entorno de programación Eclipse (probablemente el entorno multi-plataforma más usado actualmente). Esta integración se ha basado en la estructura proporcionada por la herramienta de refactorización Wrangler. Cabe destacar que esta labor no ha resultado para nada trivial y, aunque este objetivo ha sido alcanzado en este proyecto, ciertos problemas que han aparecido en el desarrollo del mismo (preservación de la apariencia, mensaje de confirmación de la correcta ejecución de los refactorings…) han tenido que ser ignorados por la complejidad de los mismos y al no ser considerados como procesos críticos respecto al objetivo principal que se pretendía alcanzar en el caso de estudio de este proyecto.

Como se ha mencionado anteriormente, el uso del lenguaje Erlang se está extendiendo a una gran velocidad entre la comunidad de programadores. Por este motivo, a partir de este momento, se pueden presentar oportunidades muy buenas respecto al hecho de abordar futuros trabajos de investigación basados en este lenguaje, representando este proyecto una base sólida y un punto de partida muy bueno en cuanto al desarrollo de estos proyectos se refiere.

Los objetivos principales que se marcaron al elegir este trabajo como proyecto final de carrera fueron la adquisición de conocimientos básicos en el lenguaje Erlang, el desarrollo de una herramienta que nos introdujera en el estudio del análisis estático (aplicable a la técnica del refactoring) y el hecho de dotar a dicha herramienta de la usabilidad que le proporcionaría la integración de la misma en un entorno de desarrollo. Llegados a este punto, podemos concluir que estos objetivos han sido alcanzados con éxito adquiriendo los conocimientos básicos de cara a futuras investigaciones. No obstante, por el camino se han encontrado ciertos obstáculos que ha habido que sortear y que con un poco más de tiempo y dedicación podrían haberse solventado de una forma más elegante. Pese a no tener una finalidad comercial, esperamos que la implementación de este proyecto así como la documentación que se presenta pueda servir de base para todo aquel que quiera iniciarse en la investigación de la temática subyacente expuesta.

BIBLIOGRAFÍA

[1] Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, and T. Nagy. Refactoring Erlang Programs. In EUC'06, Stockholm, Sweden, November 2006a.

[2] K. Sagonas and T. Avgerinos. Automatic refactoring of Erlang programs. In Proceedings of the Eleventh International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, New York, NY, USA, Sept. 2009. ACM.

[3] T. Avgerinos and K. Sagonas. Cleaning up Erlang code is a dirty job but somebody's gotta do it. In Proceedings of the Eighth ACM SIGPLAN Erlang Workshop, New York, NY, USA, Sept. 2009. ACM.

[4] T. Nagy and A. Nagyn´e-V´ıg. Erlang testing and tools survey. In Proceedings of the 7th ACM SIGPLAN Workshop on Erlang, New York, NY, USA, Sept. 2008. ACM.

[5] Wikipedia, Definición Erlang (Español), http://es.wikipedia.org/wiki/Erlang

[6] Li, H., Lindberg, A., Schumacher, A., Thompson. Improving your test code with Wrangler. Technical Report 4-09, School of Computing, Univ. of Kent, UK.

[7] J. Armstrong, R. Virding, C.Wikstrom, and M.Williams. Concurrent programming in Erlang. Prentice Hall, second edition, 1996.

[8] J. Armstrong. Concurrency oriented programming in Erlang. Invited talk, FFG 2003.

[9] J. Armstrong. Erlang - A survey of the language and its industrial applications. In INAP'96 - The 9th Exhibitions and Symposium on Industrial Applications of Prolog, Hino, Tokyo, Japan, October 1996.

[10] Cesarini, F., Thompson, Erlang Programming. O'Reilly Media, Inc. (2009)

[11] Erlang Programming Language, Official Website, http://www.erlang.org

[12] Wikipedia, Definición Erlang (Inglés), http://en.wikipedia.org/wiki/Erlang_(programming_language)

[13] Yariv's Blog, Adventures in Open Source Erlang, http://yarivsblog.com/

[14] Erlang, The Abstract Fotmat, http://www.erlang.org/doc/apps/erts/absform.html

[15] R Kitlei, L Lövei, M Tóth, Z Horváth, T Kozsik, R Király, I Bozó, C Hoch, D Horpácsi. Automated syntax manipulation in RefactorErl. In *14th International Erlang/OTP User Conference*, Stockholm, Sweden, Nov 2008.

[16] Wikipedia, Definición Refactorización, http://es.wikipedia.org/wiki/Refactorizaci%C3%B3n

[17] K. Sagonas, D. Luna. Gradual Typing of Erlang Programs: A Wrangler Experience. Seventh ACM SIGPLAN Erlang Workshop (Erlang'08), pp. 73-82, 2008.

[18] G. Orosz. The Eclipse integration of the Wrangler Erlang refactor tool. Report, Computing Lab, Univ. of Kent, 2008.

[19] Li, H., Thompson, S., Orosz, G., T oth, M.: Refactoring with Wrangler, updated. In: ACM SIGPLAN Erlang Workshop 2008, Victoria, British Columbia, Canada

[20] Smerl: Simple Metaprogramming for Erlang., Module smerl, http://erlyweb.org/doc/smerl.html

[21] Eclipse.org home, http://www.eclipse.org

[22] erlIDE, Turbo charged Erlang development, http://erlide.sourceforge.net/

[23] Oz (Programming language), http://en.wikipedia.org/wiki/Oz (programming_language)

[24] Wrangler, an Erlang Refactorer, http://www.cs.kent.ac.uk/projects/forse/wrangler/doc/ overview-summary.html

[25] RefactorErl, Erlang Refactorer, http://plc.inf.elte.hu/erlang/dl/

[26] Erlang Syntax Tools, http://www.erlang.org/doc/apps/syntax_tools/chapter.html

[27] Módulo erl_parse, http://www.erlang.org/doc/man/erl_parse.html

APÉNDICES

APÉNDICE A - THE ABSTRACT FORMAT

This document describes the standard representation of parse trees for Erlang programs as Erlang terms. This representation is known as the **Abstract Format**. Functions dealing with such parse trees are compile:forms/ [1,2] and functions in the modules epp, erl_eval, erl_lint, erl_pp, erl_parse, and io. They are also used as input and output for parse transforms (see the module compile).

We use the function Rep to denote the mapping from an Erlang source construct C to its abstract format representation R, and write R = Rep(C).

The word LINE below represents an integer, and denotes the number of the line in the source file where the construction occurred. Several instances of LINE in the same construction may denote different lines.

Since operators are not terms in their own right, when operators are mentioned below, the representation of an operator should be taken to be the atom with a printname consisting of the same characters as the operator.

MODULE DECLARATIONS AND FORMS

A module declaration consists of a sequence of forms that are either function declarations or attributes.

- If D is a module declaration consisting of the forms *F_1, ..., F_k*, then *Rep(D) = [Rep(F_1), ..., Rep(F_k)].*
- If F is an attribute *-module(Mod*), then *Rep(F) = {attribute,LINE,module,Mod}.*
- If F is an attribute *-export([Fun_1/A_1, ..., Fun_k/A_k]),* then *Rep(F) = {attribute,LINE,export,[{Fun_1,A_1}, ..., {Fun_k,A_k}]}.*
- If F is an attribute *-import(Mod,[Fun_1/A_1, ..., Fun_k/A_k]),* then *Rep(F) ={attribute,LINE,import,{Mod,[{Fun_1,A_1}, ..., {Fun_k,A_k}]}}.*
- If F is an attribute *-compile(Options),* then *Rep(F) = {attribute,LINE,compile,Options}.*
- If F is an attribute *-file(File,Line)*, then *Rep(F) = {attribute,LINE,file,{File,Line}}.*
- If F is a record declaration *-record(Name,{V_1, ..., V_k}),* then *Rep(F) ={attribute,LINE,record,{Name,[Rep(V_1), ..., Rep(V_k)]}}*. For Rep(V), see below.
- If F is a wild attribute *-A(T)*, then *Rep(F) = {attribute,LINE,A,T}.*
- If F is a function declaration *Name Fc_1 ; ... ; Name Fc_k*, where each *Fc_i* is a function clause with a pattern sequence of the same length *Arity*, then *Rep(F) = {function,LINE,Name,Arity,[Rep(Fc_1), ...,Rep(Fc_k)]}.*

RECORD FIELDS

Each field in a record declaration may have an optional explicit default initializer expression

- If V is A, then *Rep(V) = {record_field,LINE,Rep(A)}.*
- If V is A = E, then *Rep(V) = {record_field,LINE,Rep(A),Rep(E)}.*

REPRESENTATION OF PARSE ERRORS AND END OF FILE

In addition to the representations of forms, the list that represents a module declaration (as returned by functions inerl_parse and epp) may contain tuples *{error,E}* and *{warning,W},* denoting syntactically incorrect forms and warnings, and *{eof,LINE},* denoting an end of stream encountered before a complete form had been parsed.

ATOMIC LITERALS

There are five kinds of atomic literals, which are represented in the same way in patterns, expressions and guards:

- If L is an integer or character literal, then *Rep(L) = {integer,LINE,L}.*
- If L is a float literal, then *Rep(L) = {float,LINE,L}.*
- If L is a string literal consisting of the characters *C_1, ..., C_k,* then *Rep(L) = {string,LINE,[C_1, ..., C_k]}.*
- If L is an atom literal, then *Rep(L) = {atom,LINE,L}.*

Note that negative integer and float literals do not occur as such; they are parsed as an application of the unary negation operator.

PATTERNS

If Ps is a sequence of patterns *P_1, ..., P_k, then Rep(Ps) = [Rep(P_1), ..., Rep(P_k)].* Such sequences occur as the list of arguments to a function or fun.

Individual patterns are represented as follows:

- If P is an atomic literal L, then Rep(P) = Rep(L).
- If P is a compound pattern *P_1 = P_2*, then *Rep(P) = {match,LINE,Rep(P_1),Rep(P_2)}.*
- If P is a variable pattern V, then *Rep(P) = {var,LINE,A},* where A is an atom with a printname consisting of the same characters as V.
- If P is a universal pattern _, then *Rep(P) = {var,LINE,'_'}.*
- If P is a tuple pattern *{P_1, ..., P_k},* then *Rep(P) = {tuple,LINE,[Rep(P_1), ..., Rep(P_k)]}.*
- If P is a nil pattern *[]*, then *Rep(P) = {nil,LINE}.*
- If P is a cons pattern *[P_h | P_t],* then *Rep(P) = {cons,LINE,Rep(P_h),Rep(P_t)}.*

- If E is a binary pattern *<<P_1:Size_1/TSL_1, ..., P_k:Size_k/TSL_k>>*, then *Rep(E) = {bin,LINE, [{bin_element, LINE, Rep(P_1), Rep(Size_1), Rep(TSL_1)}, ..., {bin_element,LINE,Rep(P_k),Rep(Size_k),Rep(TSL_k)}]}.* For *Rep(TSL),* see below. An omitted *Size* is represented by default. An omitted TSL (type specifier list) is represented by default.
- If *P* is *P_1 Op P_2,* where *Op* is a binary operator (this is either an occurrence of ++ applied to a literal string or character list, or an occurrence of an expression that can be evaluated to a number at compile time), then *Rep(P) ={op,LINE,Op,Rep(P_1),Rep(P_2)}.*
- If *P* is *Op P_0*, where *Op* is a unary operator (this is an occurrence of an expression that can be evaluated to a number at compile time), then *Rep(P) = {op,LINE,Op,Rep(P_0)}.*
- If *P* is a record *pattern #Name{Field_1=P_1, ..., Field_k=P_k}*, then *Rep(P) = {record,LINE,Name, [{record_field,LINE,Rep(Field_1),Rep(P_1)}, ..., {record_field,LINE,Rep(Field_k),Rep(P_k)}]}.*
- If *P* is *#Name.Field,* then *Rep(P) = {record_index,LINE,Name,Rep(Field)}.*
- If *P* is *(P_0),* then *Rep(P) = Rep(P_0),* i.e., patterns cannot be distinguished from their bodies.

Note that every pattern has the same source form as some expression, and is represented the same way as the corresponding expression.

EXPRESSIONS

A body *B* is a sequence of expressions *E_1, ..., E_k,* and *Rep(B) = [Rep(E_1), ..., Rep(E_k)].*

An expression E is one of the following alternatives:

- If *P* is an atomic literal *L*, then *Rep(P) = Rep(L).*
- If *E* is *P = E_0*, then *Rep(E) = {match,LINE,Rep(P),Rep(E_0)}.*
- If *E* is a variable *V*, then *Rep(E) = {var,LINE,A},* where *A* is an atom with a printname consisting of the same characters as *V*.
- If *E* is a tuple skeleton *{E_1, ..., E_k}*, then *Rep(E) = {tuple,LINE,[Rep(E_1), ..., Rep(E_k)]}.*
- If *E* is *[]*, then *Rep(E) = {nil,LINE}.*
- If *E* is a cons skeleton *[E_h | E_t]*, then *Rep(E) = {cons,LINE,Rep(E_h),Rep(E_t)}.*
- If *E* is a binary constructor *<<V_1:Size_1/TSL_1, ..., V_k:Size_k/TSL_k>>*, then *Rep(E) ={bin,LINE,[{bin_element,LINE,Rep(V_1),Rep(Size_1),Rep(TSL_1)}, ..., {bin_element,LINE,Rep(V_k),Rep(Size_k),Rep(TSL_k)}]}*. For Rep(TSL), see below. An omitted Size is represented by default. An omitted TSL (type specifier list) is represented by default.
- If *E* is *E_1 Op E_2*, where *Op* is a binary operator, then *Rep(E) = {op,LINE,Op,Rep(E_1),Rep(E_2)}.*
- If *E* is *Op E_0*, where *Op* is a unary operator, then *Rep(E) = {op,LINE,Op,Rep(E_0)}.*
- If *E* is *#Name{Field_1=E_1, ..., Field_k=E_k}*, then *Rep(E) = {record,LINE,Name, [{record_field,LINE,Rep(Field_1),Rep(E_1)}, ..., {record_field,LINE,Rep(Field_k),Rep(E_k)}]}.*
- If *E* is *E_0#Name{Field_1=E_1, ..., Field_k=E_k},* then *Rep(E) = {record,LINE,Rep(E_0),Name, [{record_field,LINE,Rep(Field_1),Rep(E_1)}, ..., {record_field,LINE,Rep(Field_k),Rep(E_k)}]}.*
- If *E* is *#Name.Field*, then *Rep(E) = {record_index,LINE,Name,Rep(Field)}.*

- If E is *E_0#Name.Field*, then *Rep(E) = {record_field,LINE,Rep(E_0),Name,Rep(Field)}.*
- If *E* is catch *E_0*, then *Rep(E) = {'catch',LINE,Rep(E_0)}.*
- If *E* is *E_0(E_1, ..., E_k),* then *Rep(E) = {call,LINE,Rep(E_0),[Rep(E_1), ..., Rep(E_k)]}.*
- If E *is E_m:E_0(E_1, ..., E_k)*, then *Rep(E) = {call,LINE,{remote,LINE,Rep(E_m),Rep(E_0)},[Rep(E_1), ..., Rep(E_k)]}.*
- If *E* is a list comprehension *[E_0 || W_1, ..., W_k],* where each *W*_i is a generator or a filter, then *Rep(E) ={lc,LINE,Rep(E_0),[Rep(W_1), ..., Rep(W_k)]}*. For *Rep(W)*, see below.
- If *E* is a binary comprehension *<<E_0 || W_1, ..., W_k>>,* where each *W_i* is a generator or a filter, then *Rep(E) ={bc,LINE,Rep(E_0),[Rep(W_1), ..., Rep(W_k)]}.* For *Rep(W)*, see below.
- If *E* is begin *B* end, where *B* is a body, then *Rep(E) = {block,LINE,Rep(B)}.*
- If *E* is if *Ic_1 ; ... ; Ic_k* end, where each *Ic_i* is an if clause then *Rep(E) = {'if',LINE,[Rep(Ic_1), ..., Rep(Ic_k)]}.*
- If *E* is case *E_0 of Cc_1 ; ... ; Cc_k* end, where *E_0* is an expression and each *Cc_i* is a case clause then *Rep(E) ={'case',LINE,Rep(E_0),[Rep(Cc_1), ..., Rep(Cc_k)]}.*
- If *E* is *try B catch Tc_1 ; ... ; Tc_k end*, where *B* is a body and each *Tc_i* is a catch clause then *Rep(E) ={'try',LINE,Rep(B),[],[Rep(Tc_1), ..., Rep(Tc_k)],[]}.*
- If *E* is *try B of Cc_1 ; ... ; Cc_k catch Tc_1 ; ... ; Tc_n end*, where *B* is a body, each *Cc_i* is a case clause and each *Tc_j* is a catch clause then *Rep(E) = {'try',LINE,Rep(B),[Rep(Cc_1), ..., Rep(Cc_k)],[Rep(Tc_1), ..., Rep(Tc_n)],[]}.*
- If *E* is *try B after A end*, where *B* and *A* are bodies then *Rep(E) = {'try',LINE,Rep(B),[],[],Rep(A)}.*
- If *E is try B of Cc_1 ; ... ; Cc_k after A end*, where *B* and *A* are a bodies and each *Cc_i* is a case clause then *Rep(E) = {'try',LINE,Rep(B),[Rep(Cc_1), ..., Rep(Cc_k)],[],Rep(A)}.*
- If *E* is *try B catch Tc_1 ; ... ; Tc_k after A end*, where *B* and *A* are bodies and each *Tc_i* is a catch clause then *Rep(E) = {'try',LINE,Rep(B),[],[Rep(Tc_1), ..., Rep(Tc_k)],Rep(A)}.*
- If *E* is *try B of Cc_1 ; ... ; Cc_k catch Tc_1 ; ... ; Tc_n after A end*, where *B* and *A* are a bodies, each *Cc_i* is a case clause and each *Tc_j* is a catch clause then *Rep(E) = {'try',LINE,Rep(B),[Rep(Cc_1), ..., Rep(Cc_k)],[Rep(Tc_1), ..., Rep(Tc_n)],Rep(A)}.*
- If *E is receive Cc_1 ; ... ; Cc_k end*, where each *Cc_i* is a case clause then *Rep(E) ={'receive',LINE,[Rep(Cc_1), ..., Rep(Cc_k)]}.*
- If *E* is receive *Cc_1 ; ... ; Cc_k after E_0 -> B_t end*, where each *Cc_i* is a case clause, *E_0* is an expression and *B_t* is a body, then *Rep(E) = {'receive',LINE,[Rep(Cc_1), ..., Rep(Cc_k)],Rep(E_0),Rep(B_t)}.*
- If *E* is *fun Name / Arity*, then *Rep(E) = {'fun',LINE,{function,Name,Arity}}.*
- If *E* is fun *Module:Name/Arity, then Rep(E) = {'fun',LINE,{function,Module,Name,Arity}}.*
- If *E* is *fun Fc_1 ; ... ; Fc_k* end where each *Fc_i* is a function clause then *Rep(E) ={'fun',LINE,{clauses,[Rep(Fc_1), ..., Rep(Fc_k)]}}.*
- If *E* is query *[E_0 || W_1, ..., W_k]* end, where each *W_i* is a generator or a filter, then *Rep(E) ={'query',LINE,{lc,LINE,Rep(E_0),[Rep(W_1), ..., Rep(W_k)]}}*. For *Rep(W)*, see below.
- If *E* is *E_0.Field*, a Mnesia record access inside a query, then *Rep(E) = {record_field,LINE,Rep(E_0),Rep(Field)}.*
- If *E* is *(E_0)*, then *Rep(E) = Rep(E_0)*, i.e., parenthesized expressions cannot be distinguished from their bodies.

GENERATORS AND FILTERS

When *W* is a generator or a filter (in the body of a list or binary comprehension), then:

- If *W* is a generator *P <- E*, where *P* is a pattern and *E* is an expression, then *Rep(W) ={generate,LINE,Rep(P),Rep(E)}.*
- If *W* is a generator *P <= E*, where *P* is a pattern and *E* is an expression, then *Rep(W) ={b_generate,LINE,Rep(P),Rep(E)}.*
- If *W* is a filter *E*, which is an expression, then *Rep(W) = Rep(E).*

BINARY ELEMENT TYPE SPECIFIERS

A type specifier list TSL for a binary element is a sequence of type specifiers *TS_1 - ... - TS_k*. *Rep(TSL) = [Rep(TS_1), ..., Rep(TS_k)].*

When *TS* is a type specifier for a binary element, then:

- If *TS* is an atom *A*, *Rep(TS) = A.*
- If *TS* is a couple *A:Value* where *A* is an atom and Value is an integer, *Rep(TS) = {A, Value}.*

CLAUSES

There are function clauses, if clauses, case clauses and catch clauses.

A clause *C* is one of the following alternatives:

- If *C* is a function clause *(Ps) -> B* where *Ps* is a pattern sequence and *B* is a body, then *Rep(C) ={clause,LINE,Rep(Ps),[],Rep(B)}.*
- If *C* is a function clause *(Ps) when Gs -> B* where *Ps* is a pattern sequence, *Gs* is a guard sequence and *B* is a body, then *Rep(C) = {clause,LINE,Rep(Ps),Rep(Gs),Rep(B)}.*
- If *C* is an if clause *Gs -> B* where *Gs* is a guard sequence and *B* is a body, then *Rep(C) ={clause,LINE,[],Rep(Gs),Rep(B)}.*
- If *C* is a case clause *P -> B* where *P* is a pattern and *B* is a body, then *Rep(C) = {clause,LINE,[Rep(P)],[],Rep(B)}.*
- If *C* is a case clause *P when Gs -> B* where *P* is a pattern, *Gs* is a guard sequence and *B* is a body, then *Rep(C) ={clause,LINE,[Rep(P)],Rep(Gs),Rep(B)}.*
- If *C* is a catch clause *P -> B* where *P* is a pattern and *B* is a body, then *Rep(C) ={clause,LINE,[Rep({throw,P,_})],[],Rep(B)}.*
- If *C* is a catch clause *X : P -> B* where *X* is an atomic literal or a variable pattern, *P* is a pattern and *B* is a body, then *Rep(C) = {clause,LINE,[Rep({X,P,_})],[],Rep(B)}.*
- If *C* is a catch clause *P when Gs -> B* where *P* is a pattern, *Gs* is a guard sequence and *B* is a body, then *Rep(C) ={clause,LINE,[Rep({throw,P,_})],Rep(Gs),Rep(B)}.*
- If *C* is a catch clause *X : P when Gs -> B* where *X* is an atomic literal or a variable pattern, *P* is a pattern, *Gs* is a guard sequence and *B* is a body, then *Rep(C) = {clause,LINE,[Rep({X,P,_})],Rep(Gs),Rep(B)}.*

GUARDS

A guard sequence *Gs* is a sequence of guards G_*1; ...; G_k*, and *Rep(Gs) = [Rep(G_1), ..., Rep(G_k)].* If the guard sequence is empty, *Rep(Gs) = [].*

A guard G is a nonempty sequence of guard tests *Gt_1, ..., Gt_k*, and *Rep(G) = [Rep(Gt_1), ..., Rep(Gt_k)].*

A guard test *Gt* is one of the following alternatives:

- If *Gt* is an atomic literal *L*, then *Rep(Gt) = Rep(L).*
- If *Gt* is a variable pattern *V*, then *Rep(Gt) = {var,LINE,A}*, where *A* is an atom with a printname consisting of the same characters as V.
- If *Gt* is a tuple skeleton *{Gt_1, ..., Gt_k},* then *Rep(Gt) = {tuple,LINE,[Rep(Gt_1), ..., Rep(Gt_k)]}.*
- If *Gt* is *[]*, then *Rep(Gt) = {nil,LINE}.*
- If *Gt* is a cons skeleton *[Gt_h | Gt_t]*, then Rep(Gt) = {cons,LINE,Rep(Gt_h),Rep(Gt_t)}.
- If *Gt* is a binary constructor *<<Gt_1:Size_1/TSL_1, ..., Gt_k:Size_k/TSL_k>>*, then *Rep(Gt) ={bin,LINE,[{bin_element,LINE,Rep(Gt_1),Rep(Size_1),Rep(TSL_1)}, ..., {bin_element,LINE,Rep(Gt_k),Rep(Size_k),Rep(TSL_k)}]}.* For Rep(TSL), see above. An omitted *Size* is represented by default. An omitted TSL (type specifier list) is represented by default.
- If *Gt* is *Gt_1 Op Gt_2*, where *Op* is a binary operator, then *Rep(Gt) = {op,LINE,Op,Rep(Gt_1),Rep(Gt_2)}.*
- If *Gt* is *Op Gt_0*, where *Op* is a unary operator, then *Rep(Gt) = {op,LINE,Op,Rep(Gt_0)}.*
- If *Gt* is *#Name{Field_1=Gt_1, ..., Field_k=Gt_k}*, then *Rep(E) = {record,LINE,Name, [{record_field,LINE,Rep(Field_1),Rep(Gt_1)}, ..., {record_field,LINE,Rep(Field_k),Rep(Gt_k)}]}.*
- If *Gt* is *#Name.Field*, then *Rep(Gt) = {record_index,LINE,Name,Rep(Field)}.*
- If *Gt* is *Gt_0#Name.Field*, then *Rep(Gt) = {record_field,LINE,Rep(Gt_0),Name,Rep(Field)}.*
- If *Gt* is *A(Gt_1, ..., Gt_k),* where *A* is an atom, then *Rep(Gt) = {call,LINE,Rep(A),[Rep(Gt_1), ..., Rep(Gt_k)]}.*
- If *Gt* is *A_m:A(Gt_1, ..., Gt_k),* where *A_m* is the atom erlang and *A* is an atom or an operator, then *Rep(Gt) ={call,LINE,{remote,LINE,Rep(A_m),Rep(A)},[Rep(Gt_1), ..., Rep(Gt_k)]}.*
- If Gt is *{A_m,A}(Gt_1, ..., Gt_k)*, where *A_m* is the atom erlang and *A* is an atom or an operator, then *Rep(Gt) ={call,LINE,Rep({A_m,A}),[Rep(Gt_1), ..., Rep(Gt_k)]}.*
- If *Gt* is *(Gt_0)*, then *Rep(Gt) = Rep(Gt_0)*, i.e., parenthesized guard tests cannot be distinguished from their bodies.

Note that every guard test has the same source form as some expression, and is represented the same way as the corresponding expression.

THE ABSTRACT FORMAT AFTER PREPROCESSING

The compilation option *debug_info* can be given to the compiler to have the abstract code stored in the abstract_code chunk in the BEAM file (for debugging purposes).

In OTP R9C and later, the abstract_code chunk will contain

{raw_abstract_v1,AbstractCode}

where *AbstractCode* is the abstract code as described in this documen.

APÉNDICE B - SMERL

En este anexo se adjunta la implementación de las funciones del módulo SMERL utilizadas en este proyecto.

TIPO DE DATOS META_MOD()

```
-record(meta_mod, {module, file, exports = [], forms = [],
                   export_all = false}).
```

Se trata de la estructura de datos que contiene la representación abstracta del módulo en cuestión.

SMERL:FOR_FILE(FILENAME)

```
for_file(SrcFilePath) ->
    for_file(SrcFilePath, []).

for_file(SrcFilePath, IncludePaths) ->
    for_file(SrcFilePath, IncludePaths, []).

for_file(SrcFilePath, IncludePaths, Macros) ->
    case epp:parse_file(SrcFilePath, [filename:dirname(SrcFilePath) |
                                      IncludePaths], Macros) of
       {ok, Forms} ->
           mod_for_forms(Forms);
       _err ->
           {error, {invalid_module, SrcFilePath}}
    end.



mod_for_forms([{attribute,_,file,{FileName,_FileNum}},
               {attribute, _, module, ModuleName}|Forms]) ->
    {Exports, OtherForms, ExportAll} =
       lists:foldl(
         fun({attribute, _, export, ExportList},
             {ExportsAcc, FormsAcc, ExportAll}) ->
                {ExportList ++ ExportsAcc, FormsAcc, ExportAll};
            ({attribute, _, compile, export_all},
             {ExportsAcc, FormsAcc, _ExportAll}) ->
                   {ExportsAcc, FormsAcc, true};
            ({eof, _}, Acc) ->
               Acc;
            (Form, {ExportsAcc, FormsAcc, ExportAll}) ->
               {ExportsAcc, [Form | FormsAcc], ExportAll}
         end, {[], [], false}, Forms),
    {ok, #meta_mod{module = ModuleName,
                   file = FileName,
                   exports = Exports,
                   forms = OtherForms,
                   export_all = ExportAll
                  }};

mod_for_forms(Mod) ->
    {error, {invalid_module, Mod}}.
```

Devuelve el meta modelo (record `#meta_mod`) del fichero que se le pasa como argumento. La función encargada del *parseo* del fichero es `parse_file` de la librería estándar de erlang `epp` que preprocesa el código de Erlang. La función `mod_for_forms` se encarga de crear el meta_mod.

SMERL:GET_FORMS(META_MOD)

```
get_forms(MetaMod) ->
    MetaMod#meta_mod.forms.
```

Devuelve la lista de *forms* que conforman el Abstract Format del fichero en cuestión accediendo al campo correspondiente de la estructura de datos `meta_mod`.

SMERL:SET_FORMS(META_MOD,FORMS)

```
set_forms(MetaMod, Forms) ->
    MetaMod#meta_mod{forms = Forms}.
```

Asigna al campo correspondiente de la estructura de datos `meta_mod` la lista de *forms* (que conforman el Abstract Format) que se le pasa como segundo argumento.

SMERL:TO_SRC

```
to_src(MetaMod, FileName) ->
    Src = to_src(MetaMod),
    file:write_file(FileName, list_to_binary(Src)).


-spec(to_src(MetaMod::t_meta_mod()) -> string()).
to_src(MetaMod) ->
    ExportsForm =
      {attribute,1,export,get_exports(MetaMod)},
    AllForms = [{attribute,1,module,get_module(MetaMod)}, ExportsForm
                    |get_forms(MetaMod)],
    erl_prettypr:format(erl_syntax:form_list(AllForms)).
```

Este método escribe el código fuente Erlang correspondiente al meta modelo que se le pasa como primer argumento al fichero con el nombre dado. Utiliza la función format del módulo `erl_prettypr` y la función `form_list` del módulo `erl_syntax` de la librería estándar de Erlang `syntax_tools`.