JavaFMI una librería Java para el estándar Functional Mockup Interface

Grado de ingeniería informática
Proyecto fin de grado
Diciembre de 2014

Autor Johan Sebastian Cortes Montenegro
Tutor Jose Juan Hernandez Cabrera, Dpto. Informática y Sistemas

Agradecimientos

Con la finalización de este documento se cierra uno de los capítulos mas importantes de la historia mi vida. Mi educación formal termina aquí y me siento muy orgulloso de haber conseguido todo lo que conseguí. Tengo muchísimas personas a las que agradecer porque desde luego no he estado solo y sin el apoyo y compañía de todas estas personas no lo habría conseguido. La carrera ha sido larga y con muchos obstáculos pero siempre he aprovechado los problemas para aprender. Por supuesto que no todo ha sido malo y me llevo de esta etapa de mi vida una infinidad de buenos recuerdos, muchos buenos amigos, unos cuantos mentores y todo el conocimiento que con gusto aprendí.

Me gustaría agradecer primero a toda mi familia porque cada uno ha puesto ‘‘un ladrillo en esta casa’’, porque los granos de arena no son suficientes para describir su ayuda. Especialmente me gustaría agradecer a mi esposa Lisseth por todos los sacrificios y esfuerzos que ha tenido que hacer para brindarme todo el apoyo, el tiempo y el amor que he necesitado. A mi hijo Airam, porque con su corta edad supo comprender cuando *papi no puedía jugar porque tenía examen el lunes*. A mi padre Jorge que realizó un invaluable esfuerzo en brindarme la posibilidad de estudiar en un país lejano y porque siempre que pudo me extendió su mano en los momentos donde el dinero escaseaba. A mi madre porque siempre ha deseado lo mejor para mi y aunque no estuvimos juntos durante esta época, nunca he dejado de percibir su cariño y siempre se ha preocupado de mi bienestar. Se que hoy se sentirá orgullosa.

En la universidad conocí personas maravillosas y de las cuales aprendí demasiado. Entre estas personas se encuentra Jose Juan Hernández mi tutor, profesor y sobre todo mentor, al cual le tengo que dedicar también un agradecimiento especial porque su ejemplar forma de enseñar y su filosofía abrieron para muchos como yo las puertas de un mundo donde la gente programa por diversión y con gusto. A Fran Santana le agradezco porque me permitió estrechar mi relación con la universidad a la vez que, con su apoyo, incentivaba mis ganas de querer ayudar y compartir lo aprendido fuera de las clases con toda la comunidad universitaria. Al resto de profesores que con cariño, gusto y paciencia atendieron mis preguntas en sus clases les debo quizá el mejor de los agradecimientos porque: ‘‘yo vivo de preguntar, saber no puede ser lujo’’

A mis amigos también les tengo que agradecer por los momentos agradables y porque han ayudado a que me lleve nada más que buenos recuerdos. Especialmente tengo que mencionar a Cristopher Lopez Santana porque en cada debate que tuvimos aprendimos mucho uno del otro y porque juntos conseguimos fundar una comunidad de desarrolladores de la que estamos orgullosos y nos ha permitido conocer muchísimas personas muy inteligentes que trabajan de manera voluntaria para hacer de la escuela y de nuestra profesión un lugar mejor.

Me gustaría concluir este documento diciendo a todos ustedes,
con mucho cariño y mis más sincero aprecio,
Gracias Totales!!

Índice general

Capítulo 1

Motivación y objetivos iniciales

Este proyecto surge de la necesidad de importar modelos que satisfacen el estándar Functional Mockup Interface (FMI) en simulaciones escritas en Java viceversa. La interoperabilidad entre distintos modelos creados cada uno con una herramienta distinta es un problema que plantea un reto tecnológico muy interesante. Si estos modelos no tienen la capacidad de entenderse unos a otros son todos como piezas de un puzzle distinto que nunca llegaran a formar ninguna imagen. El estándar FMI permite encapsular modelos dinámicos de simulación en lo que ellos denominan una Functional Mockup Unit (FMU) y ofrece mecanismos para que estos modelos se comuniquen unos con otros.

Antes de desarrollar el proyecto ya existía una solución, llamada JFMI que permitía manipular FMUs desde aplicaciones Java. Esta librería tiene importantes carencias en aspectos como la legibilidad, el rendimiento y usabilidad. El código de esta librería no sigue las guías de estilo de Java, no respeta los principios de la orientación a objetos, tiene una estructura poco ordenada y además es lenta y muy complicada de utilizar. La expresividad en el código y la legibilidad son dos factores muy importantes en el desarrollo de todo producto software, tanto para su mantenimiento como para su adopción por otros usuarios, por eso serían una piedra angular en el desarrollo de la librería. Una vez alcanzada una API limpia y auto documentada, el siguiente objetivo sería superar el rendimiento.

Metodologicamente hablando, mi principal motivación era desarrollar un producto teniendo en mente la filosofía ágil. En clase había aprendido mucho sobre esto y me hacia preguntas cómo: ¿es posible medir rápidamente el impacto de las decisiones de diseño en los usuarios?, ¿aporta verdadero valor el gestionar los requisitos como historias de usuario?, ¿qué valor aporta para mi y para los usuarios el establecer ciclos cortos de comunicación?, ¿Es el desarrollo dirigido por pruebas tan bueno como suena?. La curiosidad y ganas de aprender sobre todo esto han sido una pieza clave en las ganas que he puesto a desarrollar este proyecto.

Capítulo 2

Antecedentes y contextualización

El estándar Functional Mockup Interface (FMI), es un estándar abierto e independiente de cualquier aplicación o herramienta que permite compartir modelos de sistemas dinámicos entre ellas. Provee una interfaz escrita en lenguaje C que ha de ser implementada por las distintas herramientas exportadoras y pone en común un conjunto de funciones para manipular los modelos. Al estar escrita en C, permite a los modelos ejecutarse en diferentes maquinas y arquitecturas e incluso sistemas empotrados. Esta interfaz se caracteriza por su bajo nivel de abstracción, aspecto que dificulta la manipulación directa de los modelos. Ademas, según el tipo de modelo, puede hacer falta contar con integradores numéricos y mecanismos de resolución de ecuaciones. La primera versión del estándar se publicó en 2010 y actualmente se encuentra en la versión 2.0.

FMI contempla dos tipos de simulación: model exchange y cosimulation. El objetivo del tipo model exchange es permitir resolver numéricamente sistemas diferenciales, algebraicos y ecuaciones discretas. Esto implica que gran parte del trabajo que ha de ser hecho por quien utiliza la FMU de este tipo es escribir código de métodos numéricos capaces de resolver dichas ecuaciones. El tipo Co-simulation está pensado principalmente para dos cosas: el poder acoplar modelos que son parte de un sistema mayor y se exportan juntos incluyendo el código necesario para resolver sus ecuaciones y para el acoplamiento de modelos hechos con distintas herramientas de simulación.

Un modelo de simulación que cumple con el estándar se denomina Functional Mockup Unit (FMU) y está compuesto principalmente de dos elementos: un conjunto de librerías dinámicas (Windows) ó compartidas (Linux) y un fichero XML denominado model description que describe aspectos del modelo como las variables, los tipos de datos y los parámetros por defecto entre otros. Opcionalmente puede incluir recursos necesarios en tiempo de ejecución y el código fuente. Esto se empaqueta en un fichero zip sin compresión y tiene la extensión fmu.

Aunque inicialmente el estándar fue pensado para la industria automovilística, existen hoy en día muchísimas herramientas de propósito general y de otros dominios que permiten importar y exportar FMUs, entre las mas importantes se encuentran: Dymola, Open Modelica, MatLab y Energy Plus. Ademas de la cantidad de herramientas que han adoptado el estándar, tenemos también un conjunto importante de compañías que participan en su desarrollo, algunas de ellas son: BOSH, Siemens, QTronic y SIMPACK.

Antes de desarrollar JavaFMI, existían cerca de 50 herramientas que permitían importar FMUs pero solo una de ellas era para Java, el nombre de dicha herramienta es JFMI y es una librería desarrollada dentro del marco del proyecto Ptolemy de la Universidad de Berkeley, California. Publicaron la versión 1.0 de la librería en Junio de 2012 y la ultima versión publicada es la 1.0.2 en Abril de 2013. Esta librería es una capa añadida sobre el estándar que, aunque permite utilizar FMUs en aplicaciones Java, necesita que los sus usuarios controlen con detalle todo el estándar incluidos sus detalles más difíciles. Ademas que necesita mucho conocimiento sobre Java Native Interface y, dado que son aspectos de muy bajo nivel, no es algo normal entre los programadores Java.

Capítulo 3

Resultados

3.1. La librería JavaFMI

JavaFMI es una herramienta que permite utilizar simulaciones que cumplen con el estándar FMI en aplicaciones Java de una manera muy simple, limpia y eficiente. Es un proyecto open source con licencia LGPL V2.1H y su código fuente se encuentra disponible para ser clonado en la pagina del proyecto. A la fecha cuenta con mas de 220 descargas en sus diferentes versiones y se han resuelto 22 de las 29 incidencias reportadas por sus usuarios. Al ser un proyecto de software libre brinda la posibilidad de recibir aportaciones de personas en todo el globo y de hecho tiene contribuyentes de distintos centros de investigación de Alemania y Francia.

El proyecto se encuentra alojado en `www.bitbucket.org/siani/javafmi` y cuenta con una página de bienvenida donde se explica como se usa la librería, una página para reportar incidencias o solicitar que se implementen nuevas historias y una página donde se listan todas las versiones que hay disponibles para descargar. JavaFMI se distribuye como un fichero zip que contiene el .jar con el código compilado de la librería una carpeta lib con las dos dependencias que tiene con librerías externas y una copia de la licencia.

3.2. Versiones

El estándar FMI publicó su primera versión denominada 1.0 en 2010 y desde entonces han pasado por tres versiones intermedias antes de llegar a FMI 2.0 en Julio del 2014. JavaFMI ha ido creciendo y dando soporte al estándar de manera incremental. FMI tiene dos tipos de comunicación con una FMU y como ya se ha mencionado antes JavaFMI

solo da soporte al tipo denominado Co-simulation. El incremento en las versiones se puede observar en la lista de descargas disponibles en la página y la numeración, en su parte más significativa, se corresponde con la versión del estándar a la que da soporte y la parte menos significativa indica cambios y corrección de errores. En este apartado se hará una revisión de cambios desde la versión más antigua a la más reciente de JavaFMI donde los cambios o mejoras más importantes aparecen primero.

v1.0 beta

- Probadas las operaciones de entrada salida, computación, obtención de estado, lectura y escritura de variables
- Probada la lectura de los ficheros modelDescription.xml
- Probado extraer ficheros de un zip
- Carga la librería usando JNA
- Introduce el concepto de simulation
- Añade soporte para FMUs generadas con Dymola
- Añade ficheros de licencia y la cabecera de licencia en cada fichero con código fuente
- Movidos los TODOs de los comentarios del repositorio a la pagina de incidencias
- Los test se ejecutan en Windows y Mac OS

v1.1 beta

- Resuelve problemas de compatibilidad de model description de FMUs generadas con Dymola
- Arreglado el issue 14 relacionado con el nombre y el identificador del modelo
- Las variables de tipo enumerado se pueden leer, aunque el estándar no lo contemple
- Las constantes no se leen de la FMU por ahorrar tiempo, se leen del model description
- publicada una herramienta que permite probar las FMUs desde una terminal de linea de comandos

v1.2 beta

- Publicado FMU tester como servicio web y basado en la aplicación de linea de comandos para probar y validar FMUs
- FMU tester registra un log de las variables deseadas en cada paso de la simulación que puede ser descargado en cualquier momento

v1.3 beta

- Distintas FMUs pueden ser utilizadas en el mismo proceso
- Cuando se extrae el fichero zip de la FMU, se hace en una carpeta independiente
- Se lanza una excepción cuando se intenta leer una variable que no existe

v1.6 beta

- Desabilitado el logging porque el formato de los mensajes de error no estándar, en algunas ocasiones se invocaba con punteros nulos que terminaban la maquina virtual
- Las variables consultadas se guardan en cache en lugar de leerse cada vez de la lista
- Mejora del rendimiento
- VariableReader y VariableWriter se extraen como objetos y se les inyecta un fmu proxy

v1.7 beta

- Rendimiento mejorado considerablemente

v1.8 beta

- Cambio en la API. eliminado el sufijo slave de los métodos resetSlave e instantiateSlave
- VariableWriter hace un cast a double del valor pasado para cuando el usuario se olvida de poner el carácter punto (.)

- Los ficheros temporales generados al extraer la FMU se borran al terminar
- El model description es extraído ahora por el ModelDescriptionDeserializer
- Se usan LinkedHashMap en luhar de HashMap para mejorar el rendimiento
- La estructura de callbacks se pasa pasada por valor y no por referencia para evitar conflictos de memoria cuando se instancian varias FMUs

v1.9 beta

- Arreglado un bug importante al leer variables del tipo boolean
- Al usar el método invokeInt(...) de JNA se gana mucho en rendimiento

v1.10 beta

- Soporte parcial para FMUs de Linux. No se puede testear todo porque no se tienen FMUs suficientes
- Se separan los test de Windows y de Linux y se definen unos primeros test para linux
- VariableWriter hace el cast de todas las variables a su correspondiente tipo, no solo las double

v2.0 beta

- Soporte para FMI V2RC1
- Se crea la clase Access para permitir la manipulación avanzada de FMUs
- Se añade SharedMemmoryAllocator como gestor de la memoria compartida entre Java y código nativo
- Refactorización en el código para reducir el acoplamiento entre clases
- Creado un conjunto de excepciones propio en lugar de usar las ofrecidas por Java

v2.0.1 beta

- Corregido un error en el nombre de un método de la clase Access

v2.1 beta

- Los ficheros temporales se crean en la carpeta temporal del sistema en lugar del directorio donde se ejecuta la aplicación
- Arreglado un error al leer el fichero model description en FMUs de la versión 1

v2.2 beta

- Arreglado un problema grave al leer variables del tipo boolean en FMUs de la versión 2RC1
- Arreglado un problema de rendimiento al leer variables del model description
- Mejorado el sistema de lanzamiento de excepciones para hacerlo más expresivo
- Arreglado un problema al leer la versión de la FMU desde el model description

v2.3 beta

- Arreglados varios problemas con el model description de FMUs de la versión 2RC1
- Se relaja la exigencia en el cumplimiento del esquema del model description porque no todas las herramientas que generan FMUs lo respetan al 100 %

v2.3.1 beta

- Se prefija con una marca de tiempo la carpeta temporal donde se extrae la FMU para permitir la instancia de muchas FMUs con exactamente el mismo nombre

v2.3.2 beta

- Arreglado el error número 10 que causaba BufferOverflowException cuando se escribían variables en la FMU

v2.4 beta

- Ultima versión que da soporte a FMI V2RC1
- Corregido un importante error que provocaba un gran consumo de memoria en simulaciones donde se guardaba el estado para hacer rollback
- Corregidos varios errores en los nombres de algunos atributos del model description
- El uso de JNA queda 100 % oculto dentro de la librería, en la clase Access el estado es representado por una clase de JavaFMI en lugar de la clase Pointer de JNA.
- el FmuProxy manipula directamente la librería compartida cargada, en lugar de ello accede a ella a través de una clase
- Se cambia el mecanismo con que se leen las librerías compartidas por uno más simple
- Se refactoriza la arquitectura de la aplicación definiendo nuevos módulos y cambiando las dependencias
- Limpieza del código y eliminación de malos olores

v2.5 beta

- Soporte para la versión FMI V2RC2
- Se deja de ignorar el parámetro resourcesLocation del estándar porque aunque este vacío el directorio de recursos algunas FMUs validan la correctitud de este valor.
- Debido a que se usa el método Native.free(peer) es necesario reservar memoria con el método Native.malloc(size) porque de lo contrario se detiene la ejecución de la maquina virtual.

v2.6 beta

- Añadido soporte para FMI V2 Final.

3.3. Manual de usuario

La librería no cuenta con un manual de usuario como tal por dos razones principales. Es tan expresiva que no necesita documentación y en la web del proyecto se describen a manera de ejemplos los casos de uso típicos. Estos casos de uso están organizados según el control que el usuario necesite tener sobre la FMU. Para el caso de uso más habitual, donde se quiere ejecutar un modelo conforme avanza el tiempo, se ofrece la clase Simulation, que encapsula casi todos los detalles más complicados del estándar y brinda un enfoque muy orientado a objetos. Para usuarios más avanzados o que necesitan un acceso más profundo a la API FMI está la clase Access, que es casi idéntica al propio estándar pero ofrece algunas de las ventajas de Java y de las facilidades de la clase Simulation, esta ultima clase también es orientada a objetos aunque el estándar tenga un enfoque tan procedural.

3.3.1. La clase simulation

Listing 3.1: La clase Simulation

```
Simulation model = new Simulation("path/to/model.fmu");
model.init(startTime);
while (!model.isTerminated()) {
        Variable out = model.readVariable("outputVariable");
        model.writeVariable("inputVariable", manipulate(out));
        model.doStep(stepSize);
}
model.terminates();
```

Con el sencillo ejemplo del listing 3.1 se documentan la mayoría de los casos de uso que un usuario del estándar podría tener. Podemos observar que la clase tiene una API muy expresiva que se describe a si misma, cuando se utiliza con entornos de desarrollo modernos como NetBeans o IntelliJ, la asistencia al programar que ofrecen estos entornos facilita enormemente el aprendizaje de la librería. En muy pocas líneas conseguimos instanciar, manipular y destruir un modelo de simulación. Se hace mucho énfasis en la expresividad del código porque es una de las piedras angulares en el desarrollo del proyecto, este concepto permite brindar a los usuarios una librería que se entiende rápidamente y evita ensuciar el código con comentarios de herramientas como JavaDoc. Con esto se ha conseguido que el coste de mantenimiento de la documentación sea cero. Para que el código se lea de esta manera se requiere un esfuerzo mental adicional como desarrollador, el mantener las cosas simples normalmente indica que el diseño o desarrollo no ha sido fácil pero los beneficios están multiplicados, tanto para los usuarios como para cualquier persona que tenga que mantener el código.

3.3.2. La clase access

Como se explicó al principio de esta sección, existen usuarios que necesitan un poco más de control sobre el estándar en la ejecución de sus modelos. En ese caso, se ofrece una clase específica para cada versión del estándar llamada Access que tiene casi la misma interfaz. El ejemplo del listing 3.2 sirve para mostrar como se interactúa con la classe Acces y además para introducir un poco el trabajo de abstracción que se ha hecho en la clase Simulation, para ello se va escribir el código que hace falta para realizar exactamente el mismo ejemplo que antes. Se debe tener cuidado cuando se utiliza la clase Access de importar la implementación que corresponda con la versión del estándar con que se ha exportado la FMU.

Listing 3.2: La clase Access

```
double simulationTime = 0.;
Simulation simulation = new Simulation("path/to/model.fmu");
ModelDescription modelDescription = simulation.getModelDescription();
int valueReference = modelDescription.getValueReference("outputVariable");
Access access = new Access(simulation);
access.initialize(START_TIME, STOP_TIME);
while (simulationTime < STOP_TIME) {
        double[] outputVariable = access.getReal(valueReference);
        access.setReal(new int[]{valueReference},
                       new double[]{manipulate(outputVariable)});
        Status stepStatus = access.doStep(simulationTime, STEP_SIZE);
        if (stepStatus == Status.OK) {
                simulationTime += STEP_SIZE;
        }
}
access.terminate();
access.freeInstance();
```

Aún cuando la clase Access encapsula muchos de los detalles propios de la implementación nativa del estándar, al comparar con el ejemplo de la clase Simulation ya podemos ver la cantidad de código que se necesita para hacer la misma tarea. Como aspecto destacable en este ejemplo, tenemos el hecho de que las variables se han de leer y escribir según su tipo y que no se puede acceder a ellas por su nombre, sino por un valor de referencia que se almacena en el fichero de descripción del modelo.

Otro aspecto que añade algo de complejidad es el hecho de tener que manipular el model description. El estándar almacena en él mucha información que si bien esta estructurada por categorías y jerárquicamente, puede llegar a ser poco amigable descender por los distintos niveles hasta la información que se desea. Por esto, se han añadido métodos a esta clase que facilitan enormemente su manipulación en los casos más típicos.

3.3.3. Haciendo rollback en la simulación con access

El ejemplo anterior sirvió como introducción a la verdadera API del estándar, pero existen otros casos de uso que motivaron el desarrollo de la clase Access. Escrito como historia de usuario, este caso de uso es: ‘‘Yo como usuario de JavaFMI quiero poder obtener y guardar el estado en que se encuentra una simulación en un momento determinado, para poder volver a él cuando lo considere oportuno’’. En el listing 3.3 se muestra como podemos manipular el objeto access hasta obtener un resultado valido, o podemos devolverlo a su estado antes de la operación y manipularlo de otra manera. Esto es tremendamente útil por ejemplo cuando queremos probar con que configuración de valores converge más rápido nuestro modelo o cuando queremos hacer rollback en operaciones que fallan.

Listing 3.3: Haciendo rollback con access

```
...
State oldState = access.getState(State.newEmptyState());
if(someOperationGoesWell(access))
return Status.OK;
else
access.setState(oldState);
return tryAnotherOperation(access);
...
```

3.3.4. Obteniendo las derivadas direccionales con access

Matemáticamente hablando una FMU es un sistema de ecuaciones. El mecanismo para resolver esas ecuaciones puede estar contenido dentro de la propia FMU o puede que haga falta de un algoritmo master que maneje las entradas y salidas de una FMU. En el listing 3.4 se ve como se completa el caso de uso, escrito como historia de usuario, en que: ‘‘como usuario avanzado del estándar quiero calcular las derivadas direccionales de una determinada variable para afinar los valores que debo fijar en la FMU y hacer que el sistema converja a una solución’’.

Listing 3.4: Obteniendo las derivadas direccionales con access

```
...
access.getDirectionalDerivatives("unknownVariable", "knownVariable");
...
```

3.4. Diseño y Arquitectura de la librería

La arquitectura de una aplicación es un concepto que muchos autores intentan definir, algunas de estas definiciones podrían gustar a unos más que a otros. Martin Fowler opina

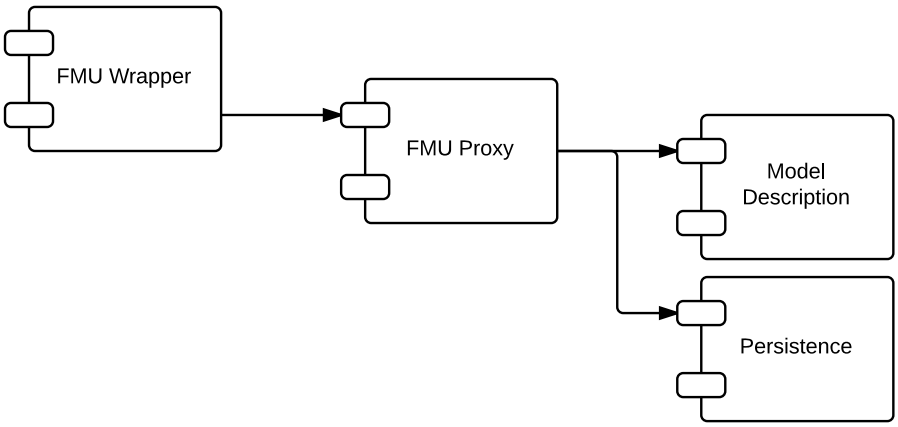
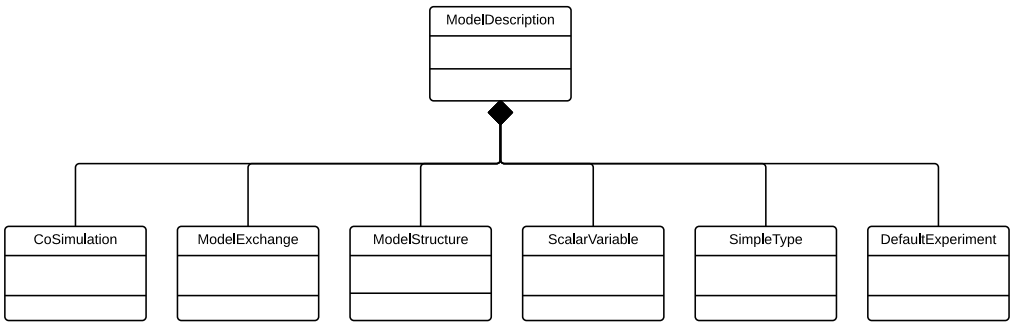


Figura 3.1:

en su libro ‘‘Patterns of Enterprise Application Architecture’’ que de todo el conjunto de definiciones que hay sobre el concepto arquitectura de aplicación se pueden sacar dos ideas en común, a) es la división de mas alto nivel de abstracción que se puede aplicar a un sistema; y b) es todo el conjunto de decisiones que es muy difícil cambiar. Sea como sea, dado que programar es un trabajo creativo tal y como lo es el arte, no existe una única manera de definir la arquitectura de un sistema, por el contrario lo que si que existen, son muchas arquitecturas dentro del sistema y la visión de lo que es considerado arquitectónicamente importante cambia con el tiempo a medida que el desarrollo del sistema avanza.

Por esta razón, y porque se ha modificado desde el primer día hasta hoy, la arquitectura de JavaFMI descrita en el listing 3.1 es una foto de la organización actual de los módulos que muy seguramente no se quedará así para siempre. La librería se compone de tres elementos principales que representan conceptos claves en el espacio de la solución. Un cuarto componente es el responsable de interactuar con el sistema de ficheros pero es menos relevante arquitectónicamente. El Model Description, FMU Wrapper y FMU Proxy se explican a continuación.

Figura 3.2: Arquitectura interna Model Description



3.4.1. Model description

El estándar FMI almacena toda la información estática de una FMU como por ejemplo los nombres de las variables, su unidad y el valor inicial en un fichero XML llamado modelDescription.xml. Este fichero se encuentra en la raíz del zip que representa la FMU. Con JavaFMI el usuario no debe preocuparse sobre como leer o analizar XML, ya que la librería se encarga de ello. La Clase ModelDescriptionDeserializer es la encargada de leer un fichero y devolver un objeto de la clase ModelDescription. Con esta configuración tan simple, es muy sencillo cambiar en cualquier momento la manera con que se hace el procesado del xml. Si apareciera una librería que lo hiciera más rápido, solo haría falta crear un nuevo ModelDescriptionDeserializer que hiciera uso de ella y todo el código de la aplicación seguiría funcionando igual.

Dada la jerarquía implícita que tienen los ficheros XML, es muy fácil establecer relaciones de composición entre los distintos elementos del fichero. Estas relaciones también se pueden transformar de manera directa en clases Java. Por eso internamente el módulo está dividido en dos paquetes, uno por cada versión del estándar implementada por JavaFMI. Lo que observamos en la figura 3.2 son algunas de las clases más importantes correspondiente a la versión 2.0 del estándar. Se han omitido en este diagrama muchas clases que en la mayoría de los casos no se utilizan, pero deben estar para dar completitud al modelado del estándar.

Por simplificar solo se ha incluido el diagrama que representa la el model description de la versión 2.0. Esta aparente duplicidad en el código está altamente justificada dado que las versiones son completamente independientes unas de otras, de hecho llegaron a haber conviviendo 3 versiones del estándar en el mismo código de JavaFMI antes del lanzamiento de la versión 2.0 estable. En cualquier momento podría aparecer una nueva completamente

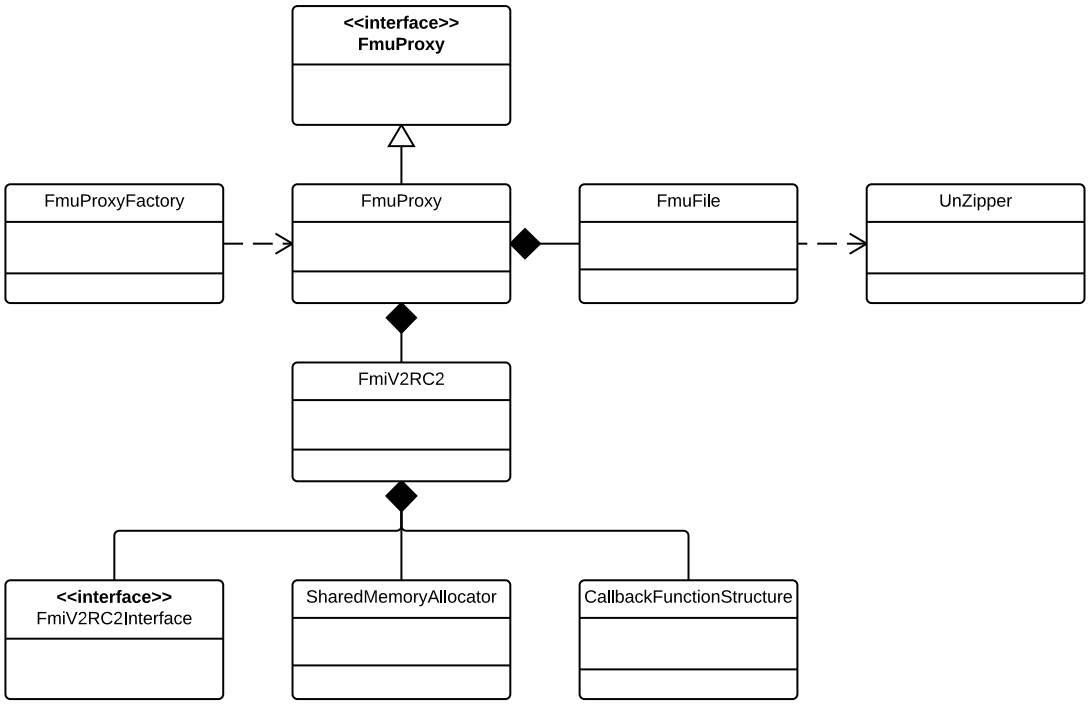


Figura 3.3: Arquitectura interna Fmu Proxy

distinta y JavaFMI podría adaptarse muy rápido extendiendo su funcionalidad y creando un nuevo paquete. A esto se le llama principio abierto cerrado que sera explicado mas adelante en la sección F.

3.4.2. FMU proxy

El módulo Fmu Proxy es la piedra angular de JavaFMI. Es el módulo responsable de orquestar detrás de cámaras como es que se realiza verdaderamente la comunicación entre Java y la librería nativa del sistema almacenada dentro del zip que representa la FMU. también está dividido en 2 paquetes donde cada uno maneja la forma en que ha de interactuar con las librerías generadas cumpliendo las distintas versiones del estándar. La organización del FMU Proxy es un poco mas compleja, pero en la figura 3.4.2 se muestran las clases mas importantes de la arquitectura y que se corresponden con el paquete de la versión 2.0.

En este módulo se ven conceptos mucho menos abstractos y que están muy cerca tanto del sistema operativo como de la implementación del estándar. Un ejemplo de ello es la clase la clase UnZipper, que es la responsable de extraer ficheros dentro de un zip, en este

caso una FMU. También tenemos la clase FmuFile, que representa un fichero que tiene extension .fmu, hace uso de un unzipper y sus métodos son los esperables en dicho rol: la obtención de la librería correspondiente al sitema operativo del usuario, el directorio donde se encuentran los recursos o el fichero xml del model description. Para este último caso vemos en el listing 3.5 que leer y analizar un fichero XML es muy simple gracias al desarrollo modular y a los principios SOLID de la orientación a objetos.

Listing 3.5: Convertir xml a clases Java

```
import static org.javafmi.modeldescription.ModelDescriptionDeserializer;
...
File modelDescriptionXml = new FmuFile("file.fmu").getModelDescriptionFile();
ModelDescription model = deserialize(ModelDescription.class,
                                     modelDescriptionXml);
```

De más bajo nivel son las clases SharedMemoryAllocator y CallbackFunctionStructure. En el primer caso, la clase SharedMemoryAllocator tiene la responsabilidad de gestionar la memoria que se usa para pasar leer y escribir desde Java, valores que se modifican en código nativo. La clase CallbackFunctionStructure alberga cuatro punteros a funciones (callbacks) que la FMU utiliza para reservar y liberar memoria, notificar la finalización de un paso en la simulación y enviar mensajes de log a la maquina que la está utilizando. Cómo se realiza la gestión de la comunicación entre Java y el código nativo será explicada más adelante en el apéndice G.

La interfaz FmiV2Interface contiene todos los métodos que define esta versión del estándar con sus firmas escritas de tal manera, que al leer la librería de la fmu con Java Native Access, se obtiene una instancia de un objeto de este tipo cuyos métodos son delegados en las funciones del estándar. Esto es posible solamente para las versiones posteriores a la 1.0 de FMI, porque en ellas los nombres de las funciones son siempre los mismos en lugar de estar prefijados con el nombre del modelo.

3.4.3. FMU wrapper

El FMU Wrapper que en inglés significa envoltorio, es el módulo con que el usuario interactúa y depende del FMU Proxy y del Model Description. Ofrece las clases Simulation y Access cuyo uso se ha explicado la sección 3.3. Este módulo es una capa de abstracción sobre el módulo FMU Proxy que ha sido añadida por comodidad para el usuario. Sus dos clases principales tienen almacenada muy poca lógica de control porque la mayoría de sus métodos son delegados. En la figura 3.4.3 podemos observar 4 conceptos clave. VariableReader y VariableWriter, que son clases de soporte para simplificar el proceso de manipulación de las variables. La clase Simulation recoge la mayoría de los casos de

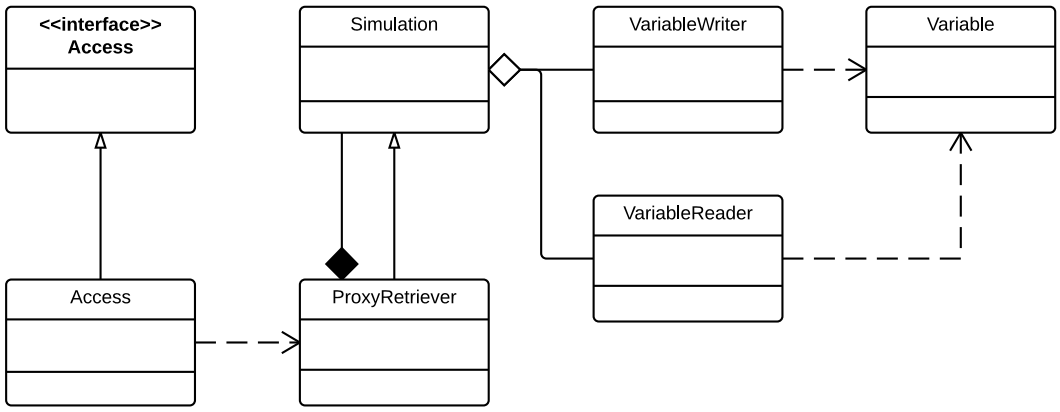


Figura 3.4: Arquitectura interna FMU Wrapper

uso que un usuario normal del estándar podría llegar a tener y por útlimo la clase Access satisface las necesidades de aquellos usuarios que desean un control más potente sobre el estándar.

Con el estándar FMI manipular las variables del modelo no es una tarea fácil o amena. Siguiendo siempre el principio simplicidad, con JavaFMI es posible realizar esta tarea utilizando el nombre de las variables en lugar de su valor de referencia, que es un numero asignado por la herramienta con que se creó la FMU y no suelen ser fáciles de recordar. Este aporte elimina el esfuerzo que un desarrollador debe invertir en recordar números que no le aportan valor y genera una sensación de comodidad con la librería que facilita su adopción.

En el módulo FMU Wrapper hay 2 clases Access, una vez más para separar por versiones del estándar. En este caso la necesidad de hacer dicha separación es mucho mayor que con el Model Description, porque de una versión para otra del FMI la API de las librerías nativas cambia. Al hacer esta separación es más fácil de extender el código para dar soporte a las nuevas versiones y se pueden mantener aislados los errores que se han de corregir en cada una de las versiones ya implementadas, esto es el principio de responsabilidad única y el principio de segregación de interfaces en acción. Un objeto de la clase Access debe crearse a partir de un objeto de la clase Simulation, internamente, como se ve en el listing 3.6 el access extrae el atributo privado FmuProxy del objeto simulation mediante un ingenioso sistema que le permite seguir ocultando ese atributo pero abre una pequeña puerta para poder obtenerlo de forma segura sin problemas de incompatibilidad a la vez que se conserva con una arquitectura muy limpia.

Listing 3.6: Obtención atributo privado FmuProxy

```
public class Access implements org.javafmi.Access {

        public Access(Simulation simulation) {
                this.proxy = new ProxyRetriever(simulation).proxy;
                ...
        }

        ...
        private class ProxyRetriever extends Simulation {
                private final FmuProxy proxy;

                public ProxyRetriever(Simulation simulation) {
                        checkVersion(simulation);
                        this.proxy = (FmuProxy) getProxy(simulation);
                }

                private void checkVersion(Simulation simulation) {
                        if(getProxy(simulation) instanceof FmuProxy) return;
                        throw new JavaFmiException(
                                ‘‘Access to FMU does not match with Simulation version’’);
                }
        }
}
```

3.5. Comparativa con JFMI

Para realizar una evaluación objetiva sobre las herramientas, se ejecutó exactamente el mismo experimento con ambas y se midieron las mismas variables para su posterior comparación. Se definieron dos variables que afectan a la ejecución de los experimentos y son: a) la usabilidad y calidad del código y b) el rendimiento en tiempo de ejecución. La usabilidad es importante porque hoy en día las interfaces son cada vez más claras y fáciles de usar, la calidad y expresividad del código trata aspectos como las reglas de diseño, respeto de las buenas prácticas y código limpio que son factores determinantes en el coste de aprendizaje y mantenimiento de la herramienta, por último, pero no menos importante el rendimiento de la librería en términos de tiempo de ejecución.

El experimento consiste en dada una FMU de la versión 1.0 de FMI, que representa un Quadrotor[1], instanciarla y fijar sus valores iniciales para iterar desde el momento cero hasta el momento mil mientras se van leyendo los valores de las variables previamente fijadas. En concreto la FMU es de un AR Drone que es un producto comercial de la empresa Parrot. Esta FMU ha sido cedida por un usuario de la librería con el propósito de realizar la comparativa entre herramientas de manera que los resultados puedan servir como punto de referencia a la hora de decidir cual de las dos utilizar para un determinado proyecto.

[1]tipo de helicóptero impulsado por cuatro motores

3.5.1. Usabilidad

Antes de medir la usabilidad se debe primero definir el concepto para posteriormente determinar las variables que nos permitan comparar los resultados. La usabilidad se puede definir como el ratio de personas que, utilizando una herramienta determinada, consiguen un objetivo con eficiencia y satisfacción. Hay que tener en cuenta que estas librerías son interfaces humano maquina, a la manera que ofrecen de interactuar con los sistemas detrás de ellas y cumplir el objetivo de los usuarios se le conoce como Application Program Interface (API) y es sobre las APIs que mediremos la usabilidad. La eficiencia en uso quiere decir que se emplea menos tiempo en conseguir el objetivo y la satisfacción es cuan cómodo se sintió el usuario con la herramienta.

Para hacer comparativas sobre herramientas necesitamos definir el perfil de personas que se espera que las utilicen, no tiene sentido por ejemplo comparar la habilidad para conducir de un piloto de rally con un ciudadano que utiliza su coche todos los días para ir y volver del trabajo. Por tanto asumimos que los desarrolladores que utilizan JavaFMI y JFMI conocen el estándar con un cierto nivel de profundidad y ademas tienen experiencia con el lenguaje Java y en general con la orientación a objetos. Habiendo definido entonces quien se espera que utilice las librerías se mide el tiempo que tarda una persona en conseguir su objetivo contando la cantidad de pasos que se necesitan para empezar a utilizar las librerías y la cantidad mínima de lineas de código con las que se puede llevar a cabo la misma tarea.

1. descargar la última versión de JavaFMI.
2. descomprimir el fichero zip.
3. añadir los ficheros JavaFMI.jar, jna.jar y simple-xml.jar descomprimidos como dependencias del proyecto.
4. instanciar la clase Simulation o la clase Access.
5. utilizar los métodos de cada clase según que se desee hacer.

Para empezar con JFMI hace falta:

1. descargar la última versión del JFMI.
2. descomprimir el zip o tar.gz según el sistema operativo.

3. copiar el código fuente de JFMI a la raíz del código del proyecto.
4. añadir jna.jar y junit.jar como dependencia del proyecto.
5. extender la clase FMUDriver e implementar el método simulate.
6. entender que es un FMUDriver y como está implementado.
7. modificar algunos atributos del FMUDriver que han de ser protected.
8. adaptar el código del ejemplo según las necesidades.

Para empezar a utilizar JavaFMI hay que completar 5 pasos contra los 8 que se necesitan para intentar empezar con JFMI. He dicho intentar porque JavaFMI no tiene apenas curva de aprendizaje para personas con el perfil descrito anteriormente porque está tan cerca del dominio de las FMUs que los métodos ofrecidos son prácticamente los que un usuario podría imaginar. Con JFMI el trabajo es mucho más complicado, en un primer acercamiento un usuario se siente abrumado porque el punto de partida no está claro. Lo primero que debe hacer es mirar los ejemplos para posteriormente darse cuenta que hay que extender de una clase. Esta clase depende de que el usuario sobreescriba atributos privados de la misma debiendo decidir si hacerlo vía setter, hacer el atributo público o hacerlo protegido. El flujo normal de trabajo con JFMI es añadir el código fuente de la librería al proyecto del usuario, dado que este código incluye pruebas, el usuario debe añadir además JUnit para que el código siquiera compile o bien borrar ese código esperando que no eliminar cosas de más. Con todos estos contratiempos, se puede decir que JFMI no está lista para empezar a usarse cuando se descarga.

Hay muchas empresas y desarrolladores escribiendo código muy malo y están ganando dinero, pero esa situación de beneficios económicos puede acabar rápidamente conforme el tiempo avanza, los requisitos aumentan, los bugs aparecen, el código se vuelve cada vez peor y al final no es posible mantenerlo. Por eso hay que destacar que dentro de la ideología ágil la calidad, expresividad y limpieza del código tienen muchísimo peso. Cuando se tienen en cuenta estos elementos las empresas pueden desarrollar y desplegar su código con confianza, pueden pivotar rápidamente en función de las necesidades del negocio, los clientes y el entorno, pueden aprovechar oportunidades competitivas y convertirse en empresas que sacan al mercado productos que se quedan por mucho tiempo.

Como se dijo en el párrafo anterior, la limpieza y la expresividad y calidad del código son elementos presentes en casi toda la literatura sobre metodologías ágiles de desarrollo de software. Robert Martin tiene un libro titulado Clean Code que se dedica por completo a tratar estos asuntos. En el párrafo anterior se ha parafraseado el prefacio del libro Implementation Patterns de Kent Beck, creador de la metodología eXtreme Programming.

La influencia de estos y otros autores en el desarrollo de JavaFMI se puede ver por todas partes. Quien usa la librería se siente cómodo utilizando conceptos próximos y que le resultan fáciles de entender, para nadie supone un esfuerzo adicional y mucho menos necesita recordar complicados detalles que le alejen de su objetivo que es manipular FMUs.

Mantener la interfaz de la librería simple ha sido un principio que se ha respetado en todos los aspectos del diseño. Para cada decisión que se ha tomado, se ha considerado que la consecuencia no suponga un esfuerzo adicional para ningún usuario. Unos nombres muy bien escogidos tanto en las clases, sus métodos y cada parámetro, proporcionan una sensación que todos los usuarios valoran como positiva. Por último, el rendimiento es un aspecto que también ha sido tenido en cuenta puesto que el tiempo de ejecución para simulaciones muy grandes no debe ser tan alto como para provocar el rechazo de los usuarios.

Listing 3.7: Código del experimento usando JavaFMI

```
import org.javafmi.Simulation;

import static java.lang.System.currentTimeMillis;
import static java.lang.System.out;

public class TestQuadrotor {

    private final Simulation simulation;

    public static void main(String[] args) {
        long startTime = currentTimeMillis();
        new TestQuadrotor(new Simulation("Drone.fmu")).execute();
        out.println("Simulation time: "
                + (currentTimeMillis() - startTime));
    }

    public TestQuadrotor(Simulation simulation) {
        this.simulation = simulation;
    }

    private void execute() {
        simulation.init(0, 1000);
        simulation.writeVariable("z_in", 25.0);
        simulation.writeVariable("theta_in", 0.125);
        do {
            display("z_out");
            display("pos_x_out");
            simulation.doStep(0.01);
        } while ((!simulation.isTerminated()));
        simulation.terminate();
    }

    private void display(String variable) {
        out.println("at time "
                + simulation.getSimulationTime() + ": "
                + variable + " = "
                + simulation.readVariable(variable).getValue());
    }
}
```

Listing 3.8: Código del experimento usando JFMI

```
package testsjfmi;

import com.sun.jna.Function;
import com.sun.jna.NativeLibrary;
import com.sun.jna.Pointer;
import org.ptolemy.fmi.*;
import org.ptolemy.fmi.driver.FMUDriver;

import java.io.File;
import java.util.List;

public class TestsJFMIMini extends FMUDriver {

        public static void main(String[] args) throws Exception {
                new TestsJFMIMini().simulate("Drone.fmu", 1000, 0.01, false, '\t', "test.txt");
        }

        @Override
        public void simulate(String fmuFileName, double endTime, double stepSize,
                             boolean enableLogging, char csvSeparator, String outputFileName)
        throws Exception {

                FMUDriver._setEnableLogging(enableLogging);
                FMIModelDescription fmiModelDescription = FMUFile.parseFMUFile(fmuFileName);
                String sharedLibrary = FMUFile.fmuSharedLibrary(fmiModelDescription);
                _nativeLibrary = NativeLibrary.getInstance(sharedLibrary);
                _modelIdentifier = fmiModelDescription.modelIdentifier;
                String fmuLocation = new File(fmuFileName).toURI().toURL().toString();
                FMICallbackFunctions.ByValue callbacks =
                new FMICallbackFunctions.ByValue(
                                                new FMULibrary.FMULogger(),
                                                new FMULibrary.FMUAllocateMemory(),
                                                new FMULibrary.FMUFreeMemory(),
                                                new FMULibrary.FMUStepFinished());

                Function fmiInstantiateSlaveFunction = getFunction("_fmiInstantiateSlave");
                Pointer fmiComponent = (Pointer) fmiInstantiateSlaveFunction.
                invoke(Pointer.class, new Object[]{_modelIdentifier,
                        fmiModelDescription.guid,
                        fmuLocation,
                        "application/x-fmu-sharedlibrary",
                        1000.,
                        (byte) 0,
                        (byte) 0,
                        callbacks,
                        (byte) 0});
                if (fmiComponent.equals(Pointer.NULL))
                throw new RuntimeException("Could not instantiate model.");

                double startTime = 0;

                invoke("_fmiInitializeSlave", new Object[]{fmiComponent, startTime,
                        (byte) 1, endTime}, "Could not initialize slave: ");
                double time = startTime;

                List<FMIScalarVariable> var = fmiModelDescription.modelVariables;
                FMIScalarVariable z_in_var = null;
```

```
58                  FMIScalarVariable theta_in_var = null;
59                  FMIScalarVariable z_out_var = null;
60                  FMIScalarVariable pos_x_out_var = null;
61                  for (FMIScalarVariable v : var) {
62                          if (v.name.equals("z_in"))
63                          z_in_var = v;
64                          if (v.name.equals("theta_in"))
65                          theta_in_var = v;
66                          if (v.name.equals("z_out"))
67                          z_out_var = v;
68                          if (v.name.equals("pos_x_out"))
69                          pos_x_out_var = v;
70                  }
71                  assert z_in_var != null;
72                  assert theta_in_var != null;
73                  assert z_out_var != null;
74                  assert pos_x_out_var != null;
75                  z_in_var.setDouble(fmiComponent, 25);
76                  System.out.println("z_in_set_to_25");
77                  theta_in_var.setDouble(fmiComponent, 0.125);
78                  System.out.println("theta_in_set_to_0.125");
79
80                  Function doStep = getFunction("_fmiDoStep");
81                  while (time < endTime) {
82                          invoke(doStep,
83                                 new Object[]{fmiComponent, time, stepSize, (byte) 1},
84                                 "Could_not_simulate,_time_was_" + time + ":_");
85                          time += stepSize;
86                          System.out.println(time
87                                     + "\tz_out=" + z_out_var.getDouble(fmiComponent)
88                                         + "\tx_out=" + pos_x_out_var.getDouble(fmiComponent));
89                  }
90                  z_out_var.getDouble(fmiComponent);
91                  pos_x_out_var.getDouble(fmiComponent);
92                  invoke("_fmiTerminateSlave",
93                         new Object[]{fmiComponent},
94                         "Could_not_terminate_slave:_");
95                  Function freeSlave = getFunction("_fmiFreeSlaveInstance");
96                  int fmiFlag = freeSlave.invokeInt(new Object[]{fmiComponent});
97                  if (fmiFlag >= FMILibrary.FMIStatus.fmiWarning)
98                  new Exception("Warning:_Could_not_free_slave_instance:_" + fmiFlag)
99                          .printStackTrace();
100         }
101 }
```

Comparando el código de los listings 3.7 y 3.8 salta a la vista lo complejo que es conseguir el mismo resultado con JFMI. Es prácticamante imposible seguir el hilo del programa y mirando en las entrañas de la librería el código no es que mejore. Durante mucho tiempo se valoró el software por la cantidad de lineas de código que lo componían pero es evidente que es una métrica errónea. Con JavaFMI se necesita escribir un 61 % menos. Además no se cumplen las guías de estilo de Java y tampoco los principios de la orientación a objetos, de hecho obliga a usar un paradigma procedural. Se revelan detalles de implementación y el usuario tiene que conocer la API de una de las librerías de las que

	Tiempo de ejecución (ms)	
Tiempo de simulación (ms)	**JavaFMI**	**JFMI**
1000	1122	2309
3000	1832	5874
5000	3025	9619
7000	4136	13486
9000	5498	17217
11000	6569	20650
13000	8236	24193

Figura 3.5: Tiempos de ejecución JavaFMI y JFMI

depende, esta dependencia también está presente en JavaFMI pero el usuario no se da cuenta de ello porque queda completamente oculta.

De cara a escoger entre una de las dos librerías, en el apartado de usabilidad y calidad del software JavaFMI es claramente la herramienta ganadora con tres pasos menos para empezar a usarla y 61 % menos lineas de código para conseguir el mismo objetivo. Podríamos hacer un análisis mucho más extensivo sobre la evaluación de la usabilidad. Podríamos acudir a ISO para mirar sus criterios y aplicar métodos mucho mas estrictos pero eso sería un documento por si mismo y no el principal objetivo de esta memoria.

3.5.2. Rendimiento

El apartado del rendimiento se va a simplificar midiendo únicamente el tiempo de ejecución. Para poder realizar una observación que aporte información interesante de cara a tomar una decisión, se ha repetido la ejecución del experimento varias veces variando en cada una el tiempo de finalización de la FMU y registrando para cada ejecución la diferencia en milisegundos del tiempo de final y el tiempo de inicio. Para evitar alteraciones en los tiempos de ejecución debidas al uso de la entrada/salida, se han comentado en los experimentos las llamadas a imprimir por pantalla. Hay que decir que con JFMI hubo que tocar el código interno de la librería para conseguirlo porque se realizan salidas por consola aún cuando las FMUs se instancian sin posibilidad de hacerlo.

Detalladamente el experimento consiste en avanzar en el tiempo la simulación de un drone donde en cada salto de tiempo se leen dos de sus variables una denominada

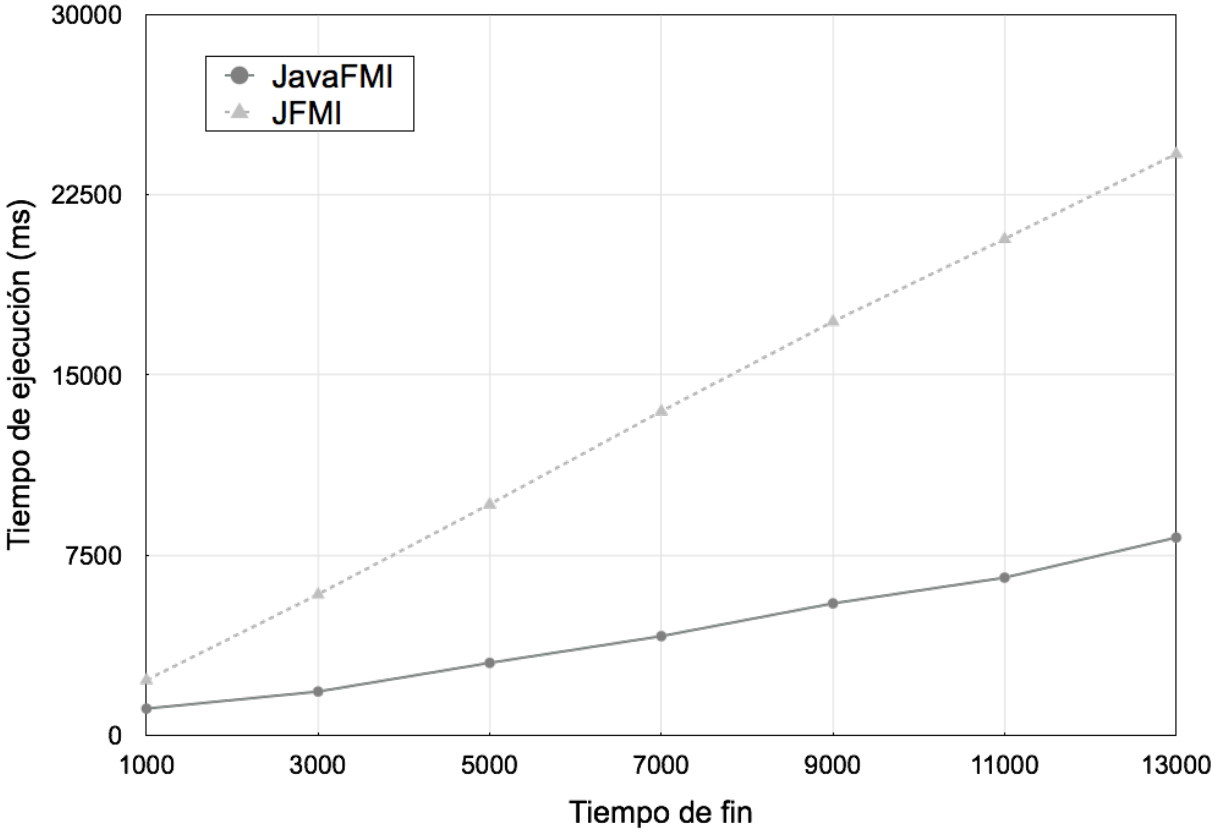


Figura 3.6: JavaFMI vs JFMI

`z_out_var` y otra denominada `pos_x_out`. El resultado de la ejecución se recoge en la figura 3.5.2 pero para hacer evidentes las diferencias, se ha incluido la figura 3.6 donde cuanto más arriba, mayor el tiempo que tarda en terminar, por lo tanto es peor.

El comportamiento que se observa es lineal en ambos casos, pero la pendiente de las dos rectas distan mucho de ser parecidas. Aunque desde un primer momento JavaFMI ya demuestra ser más rápida, la diferencia entre las dos librerías podría considerarse irrelevante. Sin embargo con solo aumentar el tiempo de la simulación a tres segundos, los resultados son aplastantes y conforme se hace mayor la simulación más evidente resulta que JavaFMI rinde mucho mejor con una reducción del 65.95 % del tiempo para la simulación de once segundos. Con este dato resulta difícil decantarse por JFMI cuando de rendimiento se trata.

El experimento se ha ejecutado en las siguientes condiciones:

- Maquina virtual Virtual Box:
  - Sistema operativo Windows 7 de 32 bits
  - 4096MB de Memoria RAM
  - Procesador Intel Core i5 2.4GHz
- Oracle JDK 1.80.25
- Java HotSpot build 25-25-b02
- JavaFMI v2.6 beta
- JFMI 1.0.2
- jna-4.1.0

3.5.3. Soporte y ayuda a los usuarios

Existe un tercer parámetro menos relacionado con la interacción con la herramienta y es el soporte a los usuarios que es sobre la interacción entre desarrolladores. El manifiesto ágil B dedica dos de sus cuatro principios a la comunicación porque sin ella no es viable desarrollar un producto con esta ideología. Por esta causa se abrió para JavaFMI un canal de comunicación con los usuarios de la librería que les permite reportar incidencias, solicitar requisitos y contribuir con el código necesario para hacer mejoras. Por este canal se han resuelto 22 incidencias y sugerencias de los usuarios y se intenta corregir los bugs en un plazo máximo de una semana. Por el contrarío JFMI aclara en su pagina web que la librería se distribuye sin algún tipo de soporte y que quizá se pueda recibir ayuda en la lista de correo electrónico de otra herramienta. Esto hace que se dificulte aún mas si cabe el proceso de aprendizaje de un desarrollador que empieza con JFMI.

A la luz de los resultados, con menos lineas de código, una API limpia, expresiva y auto documentada y un rendimiento que es un 66 % mejor, JavaFMI es objetivamente y hasta la fecha la mejor herramienta Java que existe para manipular FMUs de la versión 1.0 y 2.0 del estándar FMI.

Capítulo 4

El proceso de desarrollo

El desarrollo ágil de software es un enfoque muy diferente al modelo clásico de desarrollo conocido también como modelo en cascada. En lugar de destinar espacios de tiempo fijos para el análisis, diseño, implementación, despliegue y pruebas donde en cada una de las fases se tiene como salida un conjunto de artefactos que sirven como entrada para la fase siguiente, se propone avanzar de manera iterativa e incremental partiendo de un conjunto mínimo de historias que permiten crear una primera aproximación que entregue valor al cliente y sobre la cual se puedan construir mejoras. El desarrollo ágil es una ideología, un conjunto de principios y valores sobre los que han surgido unas prácticas y en torno a las cuales se han inventado metodologías. Muy a menudo la gente se refiere al desarrollo ágil nombrando solamente las metodologías, pero en realidad se puede desarrollar software de manera ágil sin aplicar ninguna de ellas. Lo más importante son las personas y la interacción entre ellas, después vienen los procesos y las herramientas[1]. Para el desarrollo de JavaFMI no se ha escogido una metodología concreta, sin embargo está altamente influenciado por eXtreme Programming (C) y Lean (D). A lo largo del siguiente capítulo se irá describiendo el proceso iterativo que ha llevado a JavaFMI hasta lo que es hoy mientras que también se irán describiendo como se han aplicado determinadas practicas y herramientas.

4.1. Primera iteración

Al empezar el desarrollo de JavaFMI me encontré con muchos conceptos que no entendía, por ejemplo nunca había cargado dinámicamente una librería compartida en una aplicación y ni siquiera imaginaba que era posible hacerlo desde un lenguaje que

[1] primer principio del manifiesto ágil

no fuera de la familia C. Los ficheros XML eran algo relativamente nuevo para mi y no entendía conceptos como serializar o analizar datos xml. El buen diseño, el código limpio y la orientación a objetos eran algo que acababa de descubrir así que me frenaba mucho al desarrollar pensando en detalles a los que antes no les daba la importancia que tenían. Pero sin duda el mayor de todos los retos fue hacer desarrollo dirigido por pruebas C, porque esta manera de trabajar era una cuestión más aptitudinal y no de falta de conocimiento. Había escuchado muy buenos testimonios sobre esta práctica y todos los desarrolladores que admiraba en ese momento estaban, y aún lo están, involucrados de alguna manera con ello.

Debido a todos los conceptos que debía aprender, el desarrollo de JavaFMI empezó con un acercamiento tanto al problema como a la solución analizando el código de JFMI. Por desconocimiento escribí un test de integración que era demasiado ambicioso para empezar y me iba a tomar mucho tiempo en pasar, rápidamente definí otras pruebas de más bajo nivel y conseguí refactorizar C el código JFMI para extraer el fichero FMU pero de una manera ordenada y limpia. Una vez que supe como extraer un fichero zip, me preocupe de deserializar un fichero XML y replicar su estructura en Java. Poco a poco a lo largo de la iteración fui creando nuevos test y muy rápidamente fui comprobando los beneficios de trabajar de esta manera.

Para la tarea de convertir un documento XML en una jerarquía de clases de Java utilicé la librería Simple XML. Con esta librería es muy fácil serializar y deserializar ficheros en este formato, se vale de anotaciones para especificar los elementos y atributos que se encuentran en el documento. Tiene una interfaz muy limpia y está muy bien documentada en su web, su licencia es Apache versió 2.0 lo que permite su utilización y redistribución.

4.2. Segunda iteración

En la primera iteración, de todo lo desconocido, había abordado lo que menos difícil parecía, pero había llegado el momento de profundizar en el estándar y las herramientas y adentrarse en la carga dinámica de librerías compartidas del sistema. Hablando de software una librería es un agregado de recursos de los que puede disponer un programa ya sean, funciones, estructuras de datos, clases, documentación, etc. En todos los sistemas operativos mayoritarios (GNU/Linux, Windows y Mac) existe el concepto de librería estática, y quiere decir que cuando se enlazan los componentes de un programa, el contenido de esas librerías pasa a estar contenido en el propio programa, de esta manera se asegura que las funciones o elementos que se importen de ellas estarán disponibles en tiempo de ejecución, pero con el consecuente aumento del tamaño del binario. Existe también otro tipo de

librería llamada dinámica (Windows y Mac) o compartida (GNU/Linux), cuyo contenido no se copia al programa que utiliza sus funciones sino que se hospedan en memoria y el programa que las necesita espera que estén disponibles en tiempo de ejecución.

Dentro del espacio de librerías dinámicas, que llamaremos desde ahora con la nomenclatura Windows dll, tenemos a su vez dos maneras de cargarlas en el espacio de memoria de una aplicación. Es posible especificar a la hora de enlazar los componentes de un programa que se hará uso de una librería dinámica, con lo cual el sistema operativo buscará esta librería en el instante en que cargue el programa que la necesita. Pero existe una manera aún mas flexible de hacerlo, que es cargar tanto la librería como sus funciones explícitamente justo antes de utilizarlas en tiempo de ejecución. JavaFMI explota esta posibilidad para cargar en el espacio de memoria de la aplicación Java la dll de la FMU.

Java ofrece desde la versión 1.2 del JDK la posibilidad de invocar métodos nativos desde las clases con la API de Java Native Interface (JNI). Sin embargo esto exige que en la implementación nativa de estos métodos, se escriban unas directivas especiales y de una determinada manera, además de que es obligatorio incluir la cabecera `jni.h`. Como era de esperar, la API del estándar FMI no está pensada para ser invocada así y por lo tanto no es compatible con JNI. Sería posible de todos modos cargar las clases e invocar estos métodos pero el esfuerzo que esto llevaría podría ser el desarrollo de un trabajo de fin de grado en si mismo. Para simplificar enormemente esta tarea se hace uso de la librería Java Native Access (JNA) G, que es una capa añadida sobre JNI y permite cargar cualquier librería compartida e invocar de manera sencilla sus métodos. JNA también se distribuye bajo licencia LGPL V2.1 y la segunda iteración la dediqué enteramente a aprender todo esto permitiéndome profundizar cuanto fuera necesario para poder avanzar de manera continuada y a un buen ritmo en las iteraciones siguientes.

4.3. Tercera iteración

Con lo hecho en la primera iteración y aprendido en la segunda, solo hacía falta ir encajando las piezas del puzzle para que la primera versión de JavaFMI viera la luz. Cuando el conjunto de historias implementadas fue suficiente, y todos los tests estaban en verdeC, empezó la preparación de la entrega o release que es el termino que se usa en inglés. Antes de publicar la entrega, teníamos que tomar la importante decisión de la licencia con que se iba a publicar JavaFMI. La licencia escogida fue LGPL V2.1 porque permitía a la librería ser incluida sin que el trabajo derivado del uso de la misma tuviera que ser liberado tambien.

En los primeros días se pensó en Github para hospedar el proyecto, pero por esos momentos no tenían muy bien definido el sistema de paginas web y de información, mientras que Bitbucket ofrecía ademas del gestor de incidencias un sistema de wiki para la documentación y una página web de bienvenida. Se confeccionó dicha pagina y se configuró una cuenta de Google Analytics para monitorizar aspectos relacionados con el trafico de la página. Cuando estuvo todo listo para el primer commit de la rama master, añadí la cabecera de licencia a todas las clases de la librería y se publicó la primera versión de JavaFMI.

Monitorizar el tráfico de la página era una tarea fundamental para obtener información demográfica sobre los futuros usuarios de la librería. Crear bucles rápidos de información es parte fundamental del desarrollo ágil, nos permite aprender sobre quienes son nuestros clientes y en que cosas están interesados y tomar decisiones informadas que les aporten valor. Trabajar con Google Analytics ha sido una de las experiencias mas enriquecedoras y motivacionales del desarrollo del proyecto, porque se puede ver como la librería es utilizada en Francia, Reino Unido, Austria, Italia y Alemania entre otros paises.

4.4. Cuarta iteración

Ya el código y la arquitectura de JavaFMI estaban muy bien encausados así que era hora de dedicar una iteración entera al análisis y mejora del rendimiento. Para llevar a cabo esta tarea hicimos uso de la herramienta de ‘‘profiling’’ de Oracle denominada jvisualvm. Con esta herramienta es posible monitorizar la ejecución de una aplicación Java para determinar entre otras cosas que partes del código toman más tiempo en ejecutarse. Existen dos maneras de monitorizar la ejecución de una aplicación, una de ellas menos invasiva y que afecta relativamente poco al tiempo real de ejecución y se le conoce como hacer un muestreo. Esto consiste en inspeccionar cada cierto tiempo en que método se encuentra la ejecución del programa, y luego muestra al usuario una estimación del tiempo que dura la aplicación en cada llamada de cada método. La otra manera se denomina instrumentación completa, es decir que se mide el tiempo que tardan todas y cada una de las llamadas de los métodos del programa. Esto como es de suponer aumenta el tiempo de ejecución, pero lo hace de manera constante para todos los métodos, con lo cual aunque el tiempo global suba, aporta información de mucho valor sobre el tiempo relativo que tarda una llamada a un método.

Con esta herramienta fue posible corregir muchos errores que repercutieron positivamente en el rendimiento de la librería y son visibles en la entrega de JavaFMI v1.7 beta. Por ejemplo: la manera en que se obtenían las funciones de la dll no era la mejor, así que

se añadió una cache para evitar obtener la misma función más de una vez; las llamadas al método `invoke(...)` de la clase Function de JNA cuando la función devuelve un valor entero, encapsulan una llamada a otro método de esta misma clase, `invokeInt(...)`, como este segundo método es público, para las funciones en las que se devuelve un entero se ahorra una llamada y un cast, que es lo verdaderamente costoso. Otro problema de rendimiento que se corrigió, también con una cache, fue el iterar sobre la lista de todas las variables de una FMU para leer los valores de referencia cada vez que el usuario quería leer una. Tras esta iteración se consiguió superar en rendimiento a JFMI.

4.5. Quinta iteración

Hasta el momento el desarrollo se había centrado en Windows y dado que utilizo Mac OSX para desarrollar, esta plataforma también está soportada aunque apenas y hay usuarios de este sistema operativo utilizando el estándar. Hacia falta entonces probar como se comportaba la librería en sistemas operativos basados en Linux. Nos pusimos en contacto con varias personas solicitando que por favor nos enviaran FMUs compatibles con este sistema y una vez que tuve una, adpaté los test que habían escritos para Windows de tal manera que se ejecutaran en Linux. En mi cabeza el proceso era más fácil que en la realidad. No se podían aplicar exactamente los mismos tests porque no teníamos exactamente las mismas FMUs, con lo cual hubo que definir una nueva batería de tests que nos permitiera al menos tener un mínimo de confianza en que JavaFMI podía ser compatible con todos los sistemas operativos.

Para la mayoría de las aplicaciones, Java es una alternativa muy potente por esa premisa de ‘‘escribe una vez ejecuta en todas partes’’, pero en el caso de JavaFMI al hacer uso de tecnologías de tan bajo nivel las cosas son diferentes. Hubo que corregir algunos problemas de compatibilidad antes de hacer la entrega de JavaFMI para Linux, pero en general no fue un problema mayor. Fueron apareciendo pequeños bugs y errores en la manera en que usaba JNA pero fui capaz de corregirlo, y de esta manera se añadió soporte a Linux de 32 bits en la entrega de JavaFMI v1.10 beta.

4.6. Sexta iteración

El estándar FMI hasta la fecha había lanzado una versión 2.0 beta 4 pero como no era una versión final, los cambios no estaban claros y pocas herramientas habían decidido implementarla, no aportaba valor añadir soporte a esta versión.Las cosas fueron

distintas cuando se publicó la versión 2RC1 porque algunos de los usuarios de la librería demandaron soporte para ella. Implementarla supuso cambios importantes, algunos a mejor y más de los esperados a peor. Entre los aspectos positivos de esta versión fue la eliminación del prefijo del nombre del modelo en el nombre de las funciones de la dll. Es decir si una FMU se llamaba bouncingBall, entonces el método para instanciarla se llamaba `bouncingBall_instantiateSlave(...)`. Esta limitación impedía cargar las funciones del estándar con una interfaz homogénea para todas las FMUs y obligaba a leer primero el fichero modelDescription para averiguar el nombre del modelo. Como aspecto negativo aparecieron dos nuevas funciones que forzaron al FmuProxy a lidiar con el estado de la FMU para mantener la interfaz de la clase Simulation agnóstica de la versión de la FMU.

La implementación de esta versión trajo consigo un aporte muy importante que fue proporcionar al usuario la clase Access. Esta clase le dio la capacidad de manipular una FMU con un nivel de detalle mucho mayor que con la clase Simulation, siendo responsable también de manejar todos los detalles que ya la clase Simulation manejaba por él. Entre otros cambios, modifiqué la arquitectura de la librería, con una aproximación muy similar a la que tiene hoy en día, se empezaron a utilizar excepciones propias y se iban eliminando acoplamientos entre clases como en el caso del FmuProxy con las clases VariableReader y VariableWriter; renombré algunas clases para mantener la coherencia en los paquetes de las distintas versiones y la clase FmuFile era ahora capaz de proporcionar la ruta a la carpeta de recursos aunque estuviese vacía.

Uno de los cambios más importantes en la arquitectura de JavaFMI fue la desagregación del pquete `org.javafmi.kernel` en tres partes diferentes. Por un lado extraje todas las clases relacionadas con el model description a un nuevo módulo que solo dependía de la librería simple-xml. Por otro lado todo lo relacionado con manipular directamente la dll, saqué un módulo llamado FMU Proxy que depende el model description y la librería JNA. Por último con las clases que interactúa el usuario creé un último módulo llamado FMU Wrapper que contiene únicamente las clases Simulation, Access y sus clases auxiliares.

4.7. Septima iteración

En esta iteración se avanzo en muchos frentes, empezaba a surgir la necesidad entre los usuarios de no solo importar FMUs a sus aplicaciones Java sino también de exportar simulaciones desarrolladas en Java como FMUs. Poco a poco se fueron recogiendo historias hasta que se tuvo una idea inicial sobre lo que pasaría a ser el Java FMU Builder. Con esta herramienta el usuario sería capaz de crear un fichero FMU a partir de un conjunto

de clases Java utilizando un framework. La característica mas importante de esta FMU poder ser ejecutada en el ordenador de un usuario que no tenga instalada la maquina virtual de Java.

Mientras se maduraba la idea de un módulo capaz de generar una FMU con una aplicación Java, por dentro se fueron resolviendo problemas con JavaFMI, rápidamente se lanzo un parche para la versión 2.0 porque había un error en el nombre de uno de los métodos públicos del Access, añadí detalles como dejar de extraer los ficheros temporales en la carpeta donde se ejecutaba la aplicación para hacerlo en la carpeta temporal del sistema, corregí varios problemas relacionados con el model description entre los cuales uno que no dejaba identificar la versión de la FMU por estar escrito todo el model description en una sola linea. Y así corrigiendo varios errores se fue avanzando en las entregas de JavaFMI hasta que salió una nueva versión del estándar. FMI V2RC2 supuso de nuevo un cambio en la API de la dll que provocó inmediatamente una incompatibilidad con las versiones anteriores. Según algunos usuarios, la versión 2RC2 del estándar estaba muy cerca de la V2 final así que implementé esta versión y estuvo disponible en la entrega de JavaFMI v2.5 beta for FMI V2RC2.

4.8. Octava iteración

El salto a FMI V2 final fue relativamente sencillo. No habían cambios significativos desde la versión FMI V2RC2 así que no supuso esto un gran problema, se duplicaron los test para ir haciendo la migración poco a poco. El verdadero reto en esta última iteración vino por otras partes. Al principio JavaFMI era un proyecto relativamente pequeño pero la base de usuarios creció, las versiones soportadas aumentaron y los usuarios estaban reportando algunos bugs. Como todo el desarrollo de JavaFMI ha estado dirigido por pruebas C, en esta iteración ya era inviable ejecutar manualmente todos los test, para todas las versiones en todos los sistemas operativos que daba soporte JavaFMI, así que tuve que buscar una manera de automatizar todo ello y la solución fue utilizar Gradle y Jenkins. Migrar la construcción y las pruebas desde un entorno asistido por el IDE a un sistema de scripts que se ejecutan en distintos servidores orquestados por un controlador fue todo un reto y aún queda mucho por explorar.

El otro gran reto de esta iteración fue la primera aproximación del FMU Builder. Decidí mantener la coherencia con la librería desarrollada así que el usuario tenía libertad de implementar dos interfaces en función del nivel de detalle del estándar que le interesara. Creé entonces las clases FmiSimulation y FmiAccess donde FmiSimulation es un subconjunto de operaciones disponibles con FmiAccess. Otro detalle para mantener la coherencia fue

crear una clase Zipper encargada de empaquetar la FMU de la misma manera que la clase UnZipper tenia la responsabilidad de extraerla. Como quizá has imaginado, para generar el fichero del model description creé una clase llamada ModelDescriptionSerializer. De esta manera el módulo FMU Builder es arquitectónicamente muy fácil de entender una vez que estas familiarizado con JavaFMI.

Desarrollar el builder ha sido una de las tareas más difíciles de todo el proyecto y aún esta en una fase muy experimental. El hecho de ejecutar una aplicación Java sin que el usuario de esa aplicación deba tener Java instalado implica que la maquina virtual ha de estar empotrada en la aplicación, que en realidad no es tal sino una librería compartida del sistema. Esta librería debe estar en contacto permanente con la maquina virtual para transferir a ella la llamada de los métodos que el usuario hace sobre la FMU. Para conseguir esto se ha hecho un uso extensivo de JNI que permite manipular el entorno de la maquina virtual desde código nativo. Otro reto quizá menos complicado que compilar y embeber una maquina virtual en una librería compartida fue el importar en tiempo de ejecución clases contenidas en un fichero jar para posteriormente instanciarlas y acceder sus métodos. Aún con todos los problemas que surgieron en esta última iteración se publicaron los entregables de JavaFMI v2.6 beta y JavaFmuBuilder v1.0 beta.

Capítulo 5

Aportaciones, conclusiones y trabajo futuro

El aporte más evidente de este trabajo es sin duda el producto JavaFMI en si mismo, pero la manera en que se ha desarrollado aporta un valor que merece ser descrito: Una FMU es una representación de un modelo de un objeto o de un sistema, poder tratar estos objetos como tales es un gran aporte; tener opciones a la hora de escoger herramientas es muy positivo y JavaFMI es objetivamente la mejor elección por su facilidad de uso, su rápido rendimiento y su expresividad; el ofrecer una API cercana al dominio del problema es un aporte importante porque para el usuario es más fácil recordar como se usa y no tiene que realizar el esfuerzo de aprender una nueva tecnología; al ser software libre y poder disponer de todo esto de manera gratuita y pudiendo adaptar la librería cada uno a sus propias necesidades tiene un impacto muy positivo en el ecosistema de aplicaciones FMI a la vez que engrosa la lista de aplicaciones software libre.

El desarrollo de este trabajo me ha aportado mucho conocimiento y experiencia en muchos aspectos diferentes al ámbito de la programación. Además he aprendido sobre muchísimas herramientas y metodologías que han mejorado drásticamente mi manera de trabajar. Al principio de este documento en la sección 1 de motivación y objetivos iniciales se plantearon una serie de preguntas para las cuales se ha alcanzado una respuesta y a continuación escribo mis conclusiones.

Es posible medir el impacto de las decisiones del diseño en los usuarios, y además es muy positivo para corregir rápidamente los errores y para afinar el diseño, pero para ello se han de abrir los canales de comunicación adecuados. En este caso la comunicación vía Bitbucket y correo electrónico han sido muy eficientes dado que los usuarios de la librería se encuentran en diversos países.

A lo largo de este documento no se encuentran especificaciones de casos de uso o apartados similares porque la manera en que hemos gestionado los requisitos no los contempla. Las historias de usuario, que se escriben de la forma: ‘‘Yo como [rol], quiero [característica] para [motivación o valor que aporta]’’. En esta sencilla frase se transmite quizá menos información que en una especificación de casos de uso pero precisamente por lo compacta que resulta, es mucho más efectiva porque leerlas no es una molestia ni un gran esfuerzo. Un detalle muy importante de las historias es que proporcionan el por qué se deben implementar. Esta visión permite priorizar a la hora de implementar según el valor que aporte y no según el criterio que se definió en el momento de hacer la especificación. En conclusión las historias de usuario son más fáciles de escribir y leer, son más efectivas y combinan a la perfección con metodologías Kanban y Getting Things Done.

Tras la realización de este trabajo, respecto al desarrollo dirigido por pruebas debo decir que no conozco una manera más efectiva de desarrollar. La cantidad de ventajas que aporta justifica todos los problemas que se puedan tener durante los primeros meses en que se adopta esta práctica, si bien para algunas personas es más sencillo que para otras lo importante es poco a poco irse documentando, buscando comunidades y personas a las que puedas consultar y te puedan servir de referencia. Los beneficios propiamente del desarrollo dirigido por pruebas se explican en el apéndice de C

JavaFMI ahora mismo ofrece soporte para el tipo Co-simulation de las versiones 1.0 y 2.0 del estándar FMI. En relación al cumplimiento del estándar aún queda por implementar el tipo Model Exchange y continuar con el soporte de las siguientes versiones del estándar. Durante el desarrollo de la librería surgió la necesidad de crear una herramienta que permitiera exportar a una FMU una simulación escrita en Java. El framework Java FMU Builder esta en una fase muy experimental pero ya cuenta con trece descargas y por el interés que ha despertado es una linea muy interesante para continuar el desarrollo de este proyecto. Queda muchísimo por hacer pero ese producto podría ser utilizado por grandes empresas para integrarlo en sus entornos de modelado y se podría incluso generar un modelo de negocio interesante a partir de ello. Mas allá de la propia librería se podría implementar todo un ecosistema de herramientas y aplicaciones que permitan explotar el poder de Java en modelos FMU, por ejemplo un sistema distribuido de simulación o la monitorización en tiempo real de FMU en dispositivos empotrados

Apéndice A

El estándar Functional Mockup Interface

Este apéndice es para describir detalles más técnicos o de bajo nivel acerca del estándar como pueden ser los tipos definidos, la estructura interna de una FMU y algo de la API nativa. Junto con la documentación del estándar vienen unos ficheros .h necesarios para implementar una FMU en código C, en concreto vienen tres ficheros uno llamado `fmi2TypesPlatform.h` que contiene la definición de los parámetros de entrada y salida de las funciones del estándar como se describe en el listing A.1. El fichero `fmi2FunctionTypes.h` contiene la definicion como tipos de los prototipos de las funciones del estándar, en el listing A.2 se algunos prototipos de las funciones que son comunes a los tipos Co-simulation y Model Exchange. El último fichero llamado `fmi2Functions.h` contiene las funciones que han de ser llamadas desde la propia simulación, para este fichero no he incluido ningun listing porque es demasiado extenso y hace uso de macros que no permiten simplificarlo para que se entienda.

Listing A.1: fichero fmi2TypesPlatform.h

```
typedef void* fmi2Component;
typedef void* fmi2ComponentEnvironment;
typedef void* fmi2FMUstate;
typedef unsigned int fmi2ValueReference;
typedef double fmi2Real   ;
typedef int fmi2Integer;
typedef int fmi2Boolean;
typedef char fmi2Char;
typedef const fmi2Char* fmi2String;
typedef char fmi2Byte;

#define fmi2True  1
#define fmi2False 0
```

Listing A.2: parte del fichero fmi2FunctionTypes.h

```
typedef fmi2Status fmi2SetDebugLoggingTYPE(fmi2Component,
                                           fmi2Boolean,
                                           size_t,
                                           const fmi2String[]);

typedef fmi2Component fmi2InstantiateTYPE(fmi2String,
                                          fmi2Type,
                                          fmi2String,
                                          fmi2String,
                                          const fmi2CallbackFunctions*,
                                          fmi2Boolean,
                                          fmi2Boolean);
typedef void fmi2FreeInstanceTYPE(fmi2Component);

typedef fmi2Status fmi2SetupExperimentTYPE(fmi2Component,
                                           fmi2Boolean,
                                           fmi2Real,
                                           fmi2Real,
                                           fmi2Boolean,
                                           fmi2Real);

typedef fmi2Status fmi2EnterInitializationModeTYPE(fmi2Component);
typedef fmi2Status fmi2ExitInitializationModeTYPE(fmi2Component);
typedef fmi2Status fmi2TerminateTYPE(fmi2Component);
typedef fmi2Status fmi2ResetTYPE(fmi2Component);
```

Respecto de la estructura interna de una FMU, recordemos que es un fichero zip sin compresión y que tiene la extensión .fmu. Una herramienta que intenten usar una FMU se espera encontrar dentro de ella determinados elementos en unos directorios específicos. En la figura A.1 tomada de la documentación del estándar se muestra como debe ser la estructura interna para una FMU de la versión FMI 2.0.

```
// Structure of zip file of an FMU
modelDescription.xml          // Description of FMU (required file)
model.png                     // Optional image file of FMU icon
documentation                 // Optional directory containing the FMU documentation
   index.html                 // Entry point of the documentation
   <other documentation files>
sources                       // Optional directory containing all C sources
   // all needed C sources and C header files to compile and link the FMU
   // with exception of: fmi2TypesPlatform.h , fmi2FunctionTypes.h and fmi2Functions.h
   // The files to be compiled (but not the files included from these files)
   // have to be reported in the xml-file under the structure
   // <ModelExchange><SourceFiles> … and <CoSimulation><SourceFiles>
binaries                      // Optional directory containing the binaries
   win32                      // Optional binaries for 32-bit Windows
      <modelIdentifier>.dll   // DLL of the FMI implementation
                              // (build with option "MT" to include run-time environment)
      <other DLLs>            // The DLL can include other DLLs
      // Optional object Libraries for a particular compiler
      VisualStudio8         // Binaries for 32-bit Windows generated with
                            // Microsoft Visual Studio 8 (2005)
         <modelIdentifier>.lib   // Binary libraries
      gcc3.1                // Binaries for gcc 3.1.
        ...
   win64    // Optional binaries for 64-bit Windows
      ...
   linux32  // Optional binaries for 32-bit Linux
      <modelIdentifier>.so  // Shared library of the FMI implementation
      ...
   linux64  // Optional binaries for 64-bit Linux
      ...
resources  // Optional resources needed by the FMU
   < data in FMU specific files which will be read during initialization;
     also more folders can be added under resources (tool/model specific).
     In order for the FMU to access these resource files, the resource directory
     must be available in unzipped form and the absolute path to this directory
     must be reported via argument "fmuResourceLocation" via fmi2Instantiate.
   >
```

Figura A.1: Estructura interna de una FMU

Apéndice B

Metodologías ágiles

Estamos descubriendo formas mejores de desarrollar
software tanto por nuestra propia experiencia como
ayudando a terceros. A través de este trabajo hemos
aprendido a valorar:

Individuos e interacciones sobre procesos y herramientas
Software funcionando sobre documentación extensiva
Colaboración con el cliente sobre negociación contractual
Respuesta ante el cambio sobre seguir un plan

Esto es, aunque valoramos los elementos de la derecha,
valoramos más los de la izquierda.

En Febrero del 2001 un conjunto de diecisiete desarrolladores muy hábiles y con muchísima experiencia entre los cuales estaban Martin Fowler, Kent Beck y Robert Martin entre otros se reunieron en Iuta para firmar este corto pero importante documento llamado el manifiesto ágil. Cada una de estas personas había estado trabajando en grandes empresas de desarrollo de software y tras el informe del Standish Group llamado CHAOS Report y la denominada crisis de las punto com, estos señores aportaron un paradigma de desarrollo centrado en las personas y la comunicación. Con esta nueva manera de trabajar se daba prioridad a lo verdaderamente importante y poco a poco fueron saliendo a la luz distintas

metodologías que se usaban desde hacía años en algunas empresas y además iban en esta linea. Todas las metodologías ágiles son iterativas e incrementales porque no es un secreto para nadie que las historias de una aplicación cambian de un día para otro y las tecnologías que hoy pueden resultar validas, se ven rápidamente superadas por otras que emergen con más fuerza.

Existen muchas metodologías ágiles pero entre las más conocidas e importantes se pueden destacar cuatro: eXtreme Programming, Lean, Kanban y Scrum. Todas ellas están guiadas por un conjunto de principios y valores sin los cuales no se puede practicar ninguna pero que muchas veces se dejan de lado creyendo que lo importante son los procesos o practicas que cada una recomienda. A lo largo de los apéndices siguientes se explica con un poco más de detalle en consisten las metodologías que se han utilizado para el desarrollo de este proyecto.

Apéndice C

Programación extrema

Programación extrema o eXtreme Programming (XP) es el nombre de la metodología que inventó Kent Beck y se explica en el libro ‘‘eXtreme Programming explained embrace the change’’. Define una disciplina para el desarrollo de software que centra la atención en alcanzar los objetivos comunes del equipo. Las practicas descritas en esta metodología premian la creatividad a la vez que entienden que el desarrollo de software involucra personas y que las personas se equivocan. Un equipo que usa esta metodología es capaz de producir código de muy alta calidad y con un ritmo sostenible. XP es un conjunto de principios, valores y practicas que hacen de ella una metodología que abarca casi todos los aspectos del desarrollo de software.

Los valores son los que dan un norte a toda metodología. Si cada miembro del equipo se centra en lo que es importante para el equipo, ¿en que debe centrarse cada uno?. XP aporta con sus valores la respuesta a esta pregunta: comunicación, simplicidad, retroalimentación, coraje y respeto. La comunicación es sin duda el más importante de todos por muchas razones, cuando no sabes como hacer algo casi seguro que algún miembro del equipo si lo sabe.

Los principios aportan un nivel más de concreción que los valores. Mientras estos últimos guían el comportamiento los principios aportan que se debe hacer, además que ayudan a entender las prácticas. Son quince los principios de XP: humanidad, economía, beneficio mutuo, mejora continua, diversidad, reflexión, flujo constante, ver oportunidad en los problemas, redundancia, falla pero aprende, calidad, avanza en pequeños pasos y responsabilidad aceptada. De todas formas los principios dependen del contexto en que se encuentra el equipo y la formación de las personas, así que pueden surgir nuevos o no ser todos aplicables en un determinado caso. Explicar todos los principios de forma detallada llevaría mucho tiempo y ya para eso está el libro.

El ultimo componente de XP son sus prácticas, que están organizadas en dos grupos: primarias y corolarias. a continuación menciono sólo aquellas prácticas que han afectado al desarrollo del trabajo:

Prácticas	**Descripción**
Equipo completo	Estamos todos juntos, nos ayudamos unos a otros, crecemos y aprendemos
Trabajo energizado	Trabajar únicamente las horas que te sientes productivo y aquellas que estas en capacidad de aguantar.
Historias	Haz el planning en unidades de tiempo y en un lenguaje que tu cliente entienda
Ciclo semanal	La organización del trabajo se ha de hacer cada semana
Construcción en diez minutos	Los test y el empaquetado de cada nueva versión debería durar máximo 10 minutos
Desarrollo dirigido por pruebas	Escribe un test automático que falle antes de cambiar cualquier cosa, después el código necesario para pasar
Diseño incremental	Dedica tiempo todos los días para mejorar el diseño de la aplicación.

C.1. Desarrollo dirigido por pruebas

Me gustaría terminar este apéndice hablando sobre lo que originalmente Kent Beck bautizó como Test-First Programming. El desarrollo dirigido por pruebas o Test-Driven Development (TDD) es en La teoría bastante fácil de explicar: escribe un test automático que falle antes de tocar cualquier cosa en el código de la aplicación. Los retos vienen a la hora de adoptar esta práctica porque es un cambio de mentalidad que te obliga a salir de la denominada zona de confort. Al principio la curva es empinada pero puede ser dominada como todas las técnicas mediante la práctica reiterada. No vale tirar la toalla cuando no sabemos de que manera escribir el test, a veces hay que dedicar algo de tiempo y todo tiene al final su recompensa. Programando de esta manera se consiguen muchas cosas a la vez, como por ejemplo:

1. Tienes confianza suficiente para reordenar, limpiar y refactorizar el código
2. Se mantiene la concentración únicamente en el test que estas pasando evitando poner código ‘‘por si acaso’’.

3. Cuando escribir un test se hace complicado el problema seguramente es de diseño, no del test. La cohesión y el acoplamiento se detectan inmediatamente.

4. Es difícil confiar en el autor de algo que no funciona. Cuando se desarrolla de esta manera sube la confianza del equipo en ti.

5. No se pierde el norte programando sin rumbo durante horas, haciendo TDD siempre sabes que tienes que hacer y que cosas están pendientes.

6. Aumenta la robustez del código

Apéndice D

Lean

Principio	**Descripción**
Eliminar desperdicios	Esto incluye muchas cosas como por ejemplo: código innecesario, requisitos que no están claros, repetición de trabajo manual, burocracia, comunicación interna lenta y en general cualquier cosa que no aporte valor al cliente
Amplifica el aprendizaje	Antes de escribir costosos y lentos planes para las nuevas ideas, se pueden programar nuevos experimentos y validar el conocimiento mucho más rápido. Aprende de todo cuanto puedas y comparte ese conocimiento con el equipo
Decide tan tarde como sea posible	No se deben tomar decisiones a menos que se tenga la suficiente información para hacerlo, un ejemplo claro de esto es por ejemplo no empezar el desarrollo de la aplicación por la base de datos porque se sabe que forma o tecnología va a tener finalmente.
Entrega tan rápido como puedas	Hoy en día si te paras a perfeccionar una idea antes de salir al mercado, alguien sacara una beta primero que tu y se llevara tus usuarios. Si la idea no está completa no importa, experimenta con ella para adquirir conocimiento que te permita mejorar en la linea adecuada. Por eso JavaFMI sigue en fase beta tal y como lo hizo Google con Gmail durante tantos años.

Dentro de la comunidad ágil Mary Poppendieck y Tom Poppendieck importaron de manera modificada los principios del Lean Manufacturing originales de la empresa Toyota a

la metodología Lean Software Development. Esta metodología estña ampliamente explicada en un libro del mismo nombre. En la tabla anterior se destacan, dada su importancia, los principios que han influenciado el desarrollo del proyecto.

Apéndice E

Git flow

Cuando se trabaja con sistemas de control de versiones (SCV) es muy habitual hablar del concepto de rama. Cada equipo de desarrollo puede utilizar el SCV como le parezca o como mejor le funcione, pero existen lo que se denominan modelos de ramas. El modelo de ramas utilizado en el desarrollo de este proyecto es el denominado Git Flow que fue inventado por Vincent Driessen. Este modelo de ramas como se puede sospechar de su nombre esta fundamentado en git y se encuentra perfectamente documentado en la pagina del autor `http://nvie.com/posts/a-successful-git-branching-model/`. Como una imagen dice más que mil palabras la mejor manera de explicar en que consiste este modelo de ramas es con una gráfica que se recoge en la figura E.1

Cuenta con dos ramas principales una develop donde se llevan a cabo las operaciones diarias de desarrollo, y una master donde se hospedan las versiones estables de la aplicación. Las otras tres son auxiliares y tienen un nombre muy auto descriptivo. hotfix es una rama que nace de los commits en master y arregla cosas puntuales y que no se pueden posponer, las ramas feature son para implementar sobre ellas nuevas historias, pero esto entra en conflicto con el la práctica de integración continua propuesta en XP porque se posponen las acciones de merge al momento en que se termina la historia y puede que se produzcan muchos conflictos que relenticen el proceso. Por último en las ramas release se prepara el código si hiciera falta para su integración con la rama principal master.

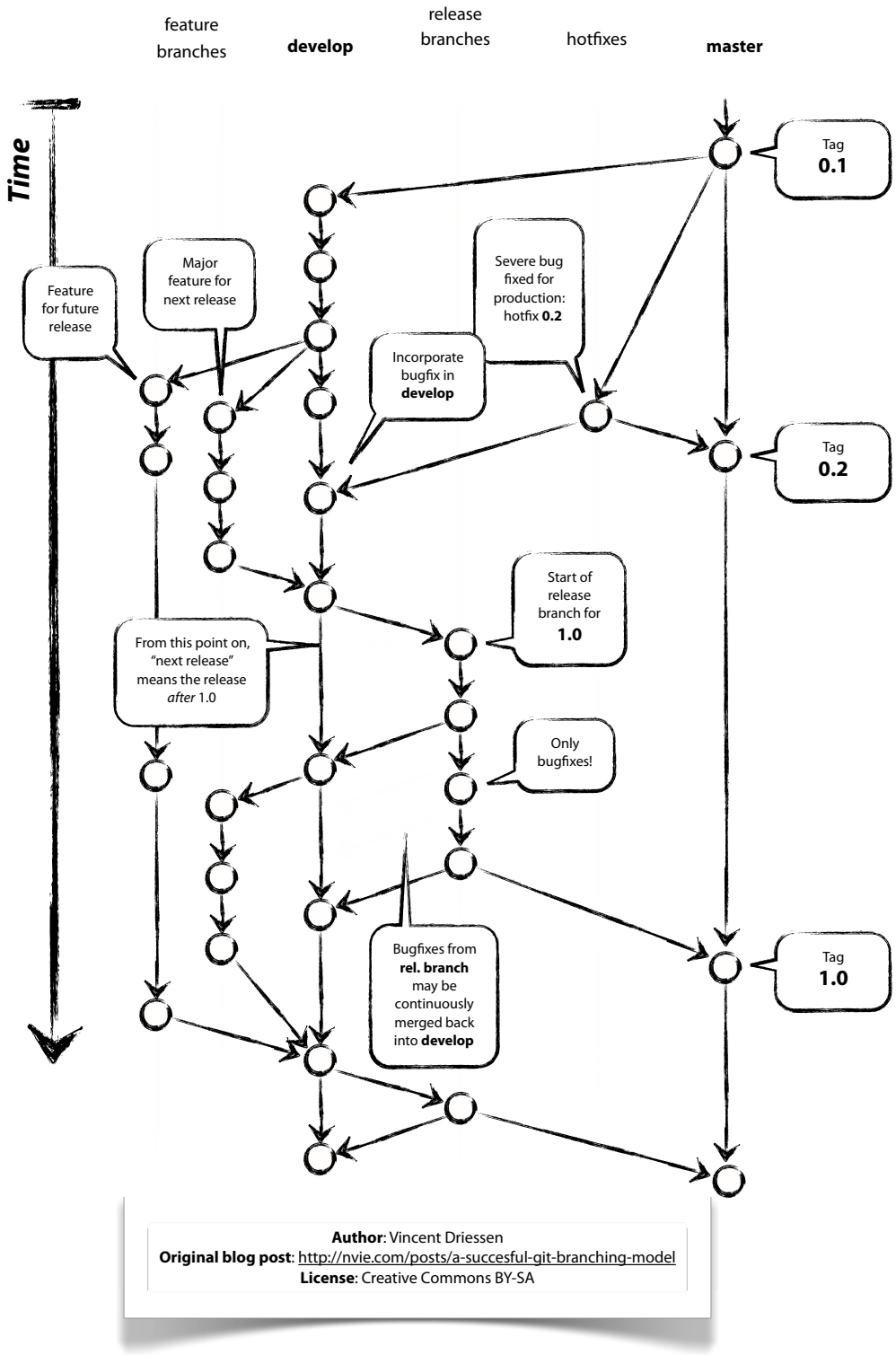


Figura E.1: Modelo de ramas Git Flow

Apéndice F

Principios de la orientación a objetos

A principios del 2000 Robert Martin puso el nombre de SOLID a los cinco primeros principios descritos por Michael Feathers. SOLID es un acrónimo para un conjunto de principios que aplicados juntos consiguen que un desarrollador cree software que es fácil de mantener y extender en el tiempo. Sirven como guía para eliminar lo que en inglés se conocen como Code Smells o malos olores en el código mediante técnicas de refactorización. Están presentes en practicamente cualquier desarrollo ágil en el que se utilicen lenguajes que apliquen este paradigma.

Un olor en el código es un síntoma de que algo no esta bien. Según Martin Fowler es un indicador en la superficie de que hay problemas en las profundidades de la aplicación. Otra manera de verlos es como el indicador de la violación de los principios de diseño que tiene un impacto bastante negativo en la calidad del software. Los code smells no son bugs porque no en realidad no hay nada técnicamente incorrecto ni impiden el funcionamiento del programa pero incrementan exponencialmente la el riesgo de cometer errores e introducir bugs

Como técnica para eliminar los code smells se pueden aplicar refactorizaciones del código, que es el proceso de reestructurar el código existente cambiando su composición sin alterar su comportamiento. Cuando se hace un refactor se mejoran carcterísticas no funcionales de la aplicacion pero se mejoran aspectos muy importantes como la legibilidad, la expresividad de la arquitectura y la extensibilidad de la aplicación. Normalmente una operación de refactor esta compuesta de varias operaciones de minirefactor, todas ellas estan documentadas en el libro de Joshua Kerievsky Refactoring to Patterns.

Tras esta explicación de porque son importantes los principios procedo a mencionar en que consiste cada uno:

Inicial	**Acrónimo**	**Principio**
S	SRP	Single responsibility principle Una clase debe ser responsable de una sola cosa por ejemplo solo un cambio potencial en la especificaión del programa debería afectar la espcificación de la clase.
O	OCP	Open closed principle Las entidades de la aplicación deben estar abiertas a extensiones pero cerradas a modificaciones
L	LSP	Liskov substitution principle Los objetos de un programa, deberían poder ser reemplazados por instancias de sus subtipos sin alterar la correctitud del programa
I	ISP	Interface segregation principle Es mejor tener muchas interfaces una para cada cliente que una super interfaz generica
D	DIP	Dependency inversion principle Uno solo debería depender de abstracciones, no de implementaciones concretas, la inyección de la dependencia (dependency inyection) es un resultado de aplicar este principio pero a menudo estos términos se confunden.

Apéndice G

Java Native Access

En una aplicación Java se puede declarar un método como nativo y su ejecución se delega en la librería dinámica que lo implementa, esta librería ha de ser cargada justo en el momento en que se carga la clase que tiene el método. También es posible crear una instancia de la maquina virtual de java desde una aplicación escrita en lenguajes de la familia C. En este último caso es posible acceder cualquier clase su métodos y sus atributos. Todo esto posible gracias a Java Native Interface pero no es una tarea trivial.

Para el caso de la invocación de funciones nativas desde aplicaciones Java existe una librería que facilita enormemente el trabajo llamada Java Native Access. Con esta herramienta un usuario es capaz hacer estas invocaciones utilizando Java de la manera en que esta acostumbrado, sin el ruido de JNI o alguna clase de código generado y mucho menos código nativo. JNA usa una pequeña libreria JNI que invoca el código nativo de manera dinámica. El desarrollador utiliza una interfaz Java para describir funciones y estructuras que están dentro de la librería que se desea invocar. Es interesante leer en su página que aunque también el rendimiento se ha tenido en cuenta, la correctitud y la facilidad de uso tienen prioridad.

En el listing G.1 se muestra un ejemplo modificado de la web de JNA donde se ve como se puede hacer uso de la llamada `printf(const char* format, ...)` de Windows desde Java de una manera bastante natural. En este caso tenemos una interfaz que contiene los métodos deseados con exactamente la misma firma que los nativos. JNA en su llamada `Native.loadLibrary(name, interfface)` devuelve una instancia de esta interfaz con el mapeo de las funciones encontradas en la librería con aquellas que se han sido definidas.

Listing G.1: LLamando al printf() del sistema desde java

```
package com.sun.jna.examples;

import com.sun.jna.Library;
import com.sun.jna.Native;
import com.sun.jna.Platform;
public class HelloWorld {

    public interface CLibrary extends Library {
        CLibrary INSTANCE = (CLibrary)
            Native.loadLibrary("msvcrt", CLibrary.class);

        void printf(String format, Object... args);
    }

    public static void main(String[] args) {
        CLibrary.INSTANCE.printf("Hello,␣%s\n", "World");
    }
}
```

Apéndice H

Licencia LGPL

El tipo de licencia que se aplica a un producto como JavaFMI puede ser determinante en la adopción del mismo. La Free Software Foundation (FSF) tiene distintos tipos de licencia General Public License (GPL) pero son bastante restrictivos porque cualquier aplicación derivada del uso de otra bajo licencia GPL debe ser distribuida con la misma licencia. Para algunos casos la FSF, aunque no es recomendable y desaconsejan su uso, ofrece un tipo de licencia Lesser GPL más permisiva con el animo de permitir el uso de las aplicaciones en otras no libres. Se ha optado por esta última para proteger el posible trabajo derivado de las personas y empresas que hagan uso de JavaFMI.

JavaFMI se distribuye con copia de la licencia LPGL y en cada fichero del código fuente se incluye una cabecera de copyrigth. A continuación se anexan estos ficheros

GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author’s reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy,

and you may at your option offer warranty protection in exchange for a fee.

  2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

    a) The modified work must itself be a software library.

    b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

    c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

    d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

    (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

  3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

  Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

  This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

  4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

  If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not

compelled to copy the source along with the object code.

  5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

  However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

  When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

  If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

  Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

  6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

  You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

    a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

    b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user’s computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

    c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

    d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

    e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made

generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  12. If the distribution and/or use of the Library is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Library under this License may add
an explicit geographical distribution limitation excluding those countries,
so that distribution is permitted only in or among countries not thus
excluded. In such case, this License incorporates the limitation as if
written in the body of this License.

  13. The Free Software Foundation may publish revised and/or new
versions of the Lesser General Public License from time to time.
Such new versions will be similar in spirit to the present version,
but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library
specifies a version number of this License which applies to it and
"any later version", you have the option of following the terms and
conditions either of that version or of any later version published by
the Free Software Foundation. If the Library does not specify a
license version number, you may choose any version ever published by
the Free Software Foundation.

  14. If you wish to incorporate parts of the Library into other free
programs whose distribution conditions are incompatible with these,
write to the author to ask for permission. For software which is
copyrighted by the Free Software Foundation, write to the Free
Software Foundation; we sometimes make exceptions for this. Our
decision will be guided by the two goals of preserving the free status
of all derivatives of our free software and of promoting the sharing
and reuse of software generally.

NO WARRANTY

  15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW.
EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR
OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY
KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE
LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN
WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY
AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU
FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE
LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING
RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A
FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

  If you develop a new library, and you want it to be of the greatest
possible use to the public, we recommend making it free software that
everyone can redistribute and change. You can do so by permitting
redistribution under these terms (or, alternatively, under the terms of the
ordinary General Public License).

  To apply these terms, attach the following notices to the library. It is
safest to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least the
"copyright" line and a pointer to where the full notice is found.

    <one line to give the library's name and a brief idea of what it does.>
    Copyright (C) <year> <name of author>

```
    This library is free software; you can redistribute it and/or
    modify it under the terms of the GNU Lesser General Public
    License as published by the Free Software Foundation; either
    version 2.1 of the License, or (at your option) any later version.

    This library is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
    Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public
    License along with this library; if not, write to the Free Software
    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the library, if
necessary.  Here is a sample; alter the names:

  Yoyodyne, Inc., hereby disclaims all copyright interest in the
  library ‘Frob’ (a library for tweaking knobs) written by James Random Hacker.

  <signature of Ty Coon>, 1 April 1990
  Ty Coon, President of Vice

That’s all there is to it!
```

Bibliografía y fuentes de información

- **Herramientas y conceptos**

1. **Functional Mockup Interface**
   www.fmi-standard.org/

2. **JFMI**
   ptolemy.eecs.berkeley.edu/java/jfmi

3. **Simple XML**
   simple.sourceforge.net/

4. **Java Native Interface**
   The Java Native Interface Programmer's Guide and Specification, Shen Liang
   docs.oracle.com/javase/7/docs/technotes/guides/jni/
5. **Java Native Access**
   github.com/twall/jna

6. **JUnit**
   junit.org/

7. **Gradle**
   www.gradle.org/

8. **VisualVM**
   visualvm.java.net/

9. **Maquina Virtual Avian**
   github.com/ReadyTalk/avian

10. **Git flow**
    nvie.com/posts/a-successful-git-branching-model/

11. **Principios de la orientación a objetos**
    sites.google.com/site/unclebobconsultingllc/getting-a-solid-start
    Design Principles and Design Patterns, Robert C. Martin

- **Metodologías ágiles**

1. Agile Manifesto, Varios autores, agilemanifesto.org/
2. eXtreme Programming explained embrace the Change, Kent Beck, 2da edición
3. Test-Driven Development by Example, Kent Beck
4. Diseño ágil con TDD, Carlos Ble
5. Patterns of Enterprise Application Architecture, Martin Fowler
6. Refactoring. Improving the Design of Existing Code, Martin Fowler
7. Clean Code, Robert C. Martin
8. Lean Software Development, Mary and Tom Poppendieck