ARQUITECTURA PARA LA INTEROPERABILIDAD DE UNIDADES DE SIMULACIÓN BASADAS EN FMU

Proyecto Fin de Carrera

Ingeniería Informática

16/02/2015

Autora:
Sánchez Medrano, María del Carmen

Tutores:
Hernández Cabrera, José Juan
Évora Gómez, José
Hernández Tejera, Francisco M.

Escuela de Ingeniería Informática. Universidad de Las Palmas de G.C.

Proyecto fin de carrera

Proyecto fin de carrera de la Escuela de Ingeniería Informática de la Universidad de Las Palmas de Gran Canaria presentado por la alumna:

Apellidos y nombre de la alumna: Sánchez Medrano, María del Carmen
Título: Arquitectura para la interoperabilidad de unidades de simulación basadas en FMU
Fecha: Febrero de 2015
Tutores: Hernández Cabrera, José Juan
Évora Gómez, José
Hernández Tejera, Francisco M.

Escuela de Ingeniería Informática. Universidad de Las Palmas de G.C.

Dedicatoria

A MI MADRE Y A MI PADRE

Escuela de Ingeniería Informática. Universidad de Las Palmas de G.C.

Agradecimientos

A mis tutores. A José Juan Hernández Cabrera por su dirección y orientación en este proyecto. Por ser un excelente mentor que ha permitido iniciarme en la línea de investigadora y en el aprendizaje de otras filosofías de programación. A José Évora Gómez por sus explicaciones y ayuda incondicional durante todo el desarrollo. A Francisco Mario Hernández por darme la oportunidad de integrarme en este excelente equipo.

A todo el grupo de profesores que me han instruido durante todos estos años permitiéndome conseguir el objetivo deseado: ser ingeniera informática.

A todos mis compañeros y amigos de carrera, en especial a Alberto y Dani. Con ellos he compartido estos últimos años de carrera llenos de vivencias y anécdotas. Gracias a todos por estar ahí ya que sin ustedes este camino no hubiera podido ser tan fácil y divertido. He conocido a grandes amigos en estos años con los que espero seguir haciendo proyectos juntos y poderlos ayudar cuando lo necesiten.

A Edu por aguantarme en la carrera, en los momentos buenos y malos, teniendo en él un apoyo incondicional en todo momento.

Y en especial a mi familia, sobretodo a mis padres. Ellos han sido un gran apoyo, no sólo a nivel económico. Han sido el verdadero sostén moral y anímico del cual yo he salido fortalecida siempre, en cualquiera de las condiciones que me encontrase, viendo en ellos la mayor de las ayudas. Siempre han estado ahí cuando los he necesitado alegrándose cuando he conseguido mis metas y apoyándome a conseguirlas. Sé que para ellos es un orgullo que haya llegado hasta aquí sobretodo por todo el esfuerzo que hay detrás. Nunca podré agradecérselo lo suficiente.

Un recuerdo muy, muy, muy especial a mi madre que esperaba este momento con ilusión y que desgraciadamente no ha podido ver.

Me gustaría finalizar este documento diciéndoles a todos ustedes de todo corazón

¡Muchas gracias!

Escuela de Ingeniería Informática. Universidad de Las Palmas de G.C.

Índice general

Índice de figuras

Facultad de Informática. Universidad de Las Palmas de G.C.

Capítulo 1

Motivación y objetivos iniciales

En diversos sectores de la industria, como puede ser la eléctrica o la automovilística, la simulación es una parte fundamental del diseño y fabricación de los productos. Con ello, se intenta disminuir costes y asegurar que los productos sean lo más fiables posible.

Para dar soporte a estas necesidades, en el mercado existen diversas herramientas de simulación, cada una especializada en un determinado tipo de problema y con una orientación concreta al modelado del sistema: Anylogic, MatLab, Dymola...

Puede suceder, por ejemplo en la industria automovilística, que un ingeniero realizando la simulación de un motor, necesite utilizar la simulación de una bujía que ha sido realizada con otra herramienta. Aún más, el ingeniero que está realizando la simulación del motor no tiene por qué tener instalada la herramienta con la que se realizó la simulación de dicha bujía.

Por tanto existe la necesidad tanto de poder integrar simulaciones realizadas con herramientas distintas, como de distribuir una simulación en un formato ejecutable, es decir, que no requiera tener instalada la herramienta con la que se desarrolló.

En 2010, la fundación Modelisar presentó el estándar Functional Mock-up Interface (FMI) como una solución a estas necesidades de interoperabilidad entre herramientas de simulación. En FMI, se define una interfaz común (API) que permite la distribución e interoperabilidad de simulaciones. Así, una simulación puede transformarse en un formato ejecutable para su distribución con una interfaz pública conocida. En este

estándar, una simulación se empaqueta en un formato de fichero llamado Functional Mock-up Unit (FMU). Una simulación en formato FMU se define como un fichero comprimido en formato zip. Este fichero, además de incluir una descripción de la propia simulación, incluye unas librerías dinámicas que permiten la ejecución de la simulación en múltiples sistemas operativos. Para ello, se incluyen los binarios compilados de unas librerías para: win32, win64, linux32, linux64 y Mac OS.

No obstante, existen otro tipo de necesidades relacionadas con la interoperabilidad de simulaciones. En el caso de que se deseara simular el uso de la electricidad en las viviendas de una ciudad como Las Palmas de Gran Canaria, unas 200.000 aproximadamente, habría que resolver la interoperabilidad en una ejecución distribuida. En este caso, dado que es una simulación muy grande, podría empaquetarse la simulación de un edificio o un barrio en una FMU con el objetivo de repartirlas en una red de ordenadores.

En general, cuando una simulación requiere muchos componentes que tienen que ser simulados, podemos encontrarnos con un problema de falta de recursos en el ordenador: por ejemplo insuficiente memoria RAM. Para resolver este problema, se plantea la posibilidad de realizar una ejecución distribuida de los componentes.

Este problema, que se presenta habitualmente en simulaciones muy grandes, ha sido identificado por el Instituto Universitario SIANI de la Universidad de Las Palmas de Gran Canaria. Además otros institutos de investigación europeos y empresas, con los que tiene relación el SIANI, también demandan soluciones de escalabilidad en simulaciones. En este proyecto concretamente se aborda la interoperabilidad distribuida de simulaciones bajo el estándar FMI.

Objetivos iniciales El objetivo de este proyecto es el dar a los usuarios del estándar FMI la posibilidad de poder ejecutar las unidades de simulación (FMUs) de forma distribuida. Para conseguir el objetivo principal del proyecto se han desglosado los siguientes subobjetivos:

- **Estudiar el estándar FMI**.
- **Definición de una arquitectura**. La definición de la arquitectura es una parte fundamental en el desarrollo de software ya que define a grandes rasgos los módulos que van a formar parte en el proceso. A lo largo de las diversas iteraciones del

desarrollo esta arquitectura ha sufrido cambios significativos guiando el desarrollo y redefiniendo los subobjetivos del mismo.

- **Definición del protocolo de comunicación**. Se definen los métodos de serialización y deserialización así como los diversos protocolos de comunicación.
- **Asegurar el correcto funcionamiento de la ejecución de la simulación en remoto** para lo cual se realizarán pruebas.

Capítulo 2

Antecedentes y contextualización

2.1. Functional Mock-up Interface

El Functional Mock-up Interface (FMI) es un estándar abierto que ofrece la compartición de modelos de sistemas dinámicos entre aplicaciones. Proporciona una interfaz en C para que las aplicaciones que pretendan utilizarlo la implementen, y ofrece un conjunto de funciones para la manipulación de modelos. Permite que los modelos se ejecuten en máquinas de arquitectura diferentes, incluyendo sistemas empotrados. Dicha interfaz tiene un bajo nivel de abstracción; si no fuera así, dificultaría la manipulación directa de los modelos. Puede que dependiendo del modelo se necesiten integradores numéricos y mecanismos de resolución de ecuaciones. Su primera revisión se lanzó en 2010, y su versión actual es la 2.0 aunque se han lanzado las versiones Release Candidate (RC) 1 y 2 de esta última versión.

Este estándar contempla simulaciones de dos clases: model exchange y co-simulation.

- **Model exchange** tiene como objetivo el resolver numéricamente sistemas diferenciales, algebraicos y ecuaciones discretas, con lo cual gran parte del trabajo debe realizarse por quien utiliza la FMU, ya que ha de escribir código de métodos numéricos capaces de resolver las ecuaciones.

- **Co-simulation** ha sido concebido para dos tareas: el acoplamiento de modelos que forman parte de un sistema mayor y que han de ser exportados juntos incluyendo código necesario para resolver sus ecuaciones, y el acoplamiento de modelos concebidos con diversas herramientas de simulación.

El concepto básico del estándar consiste en crear una FMU (Functional Mock-up Unit) de un modelo en una herramienta cualquiera y luego importarla y usarla en un entorno cerrado. Una **FMU** consiste en un fichero comprimido con la extensión *.fmu*. En su interior podemos encontrar:

- Un fichero XML, nombrado como FmiModeldescription, que describe aspectos del modelo como las variables, los tipos de datos y los parámetros por defecto.
- El código de la simulación como un conjunto de funciones en C, el cual podemos encontrar como ficheros binarios (librerías de sistema) y/o el código fuente.
- Por último, encontramos la carpeta resources, la cual puede contener cualquier cosa que la simulación necesite.

El hecho de usar una interfaz en C hace que el compilado de las FMUs sea independiente aunque siguen siendo dependientes del sistema operativo.

En un inicio este estándar se desarrolló para la industria del automóvil; sin embargo, hoy en día existen numerosas herramientas de propósito general y de diversos dominios que ofrecen la posibilidad de trabajar con FMUs, importarlas y exportarlas. Entre los más conocidos tenemos Dymola, Open Modelica, MatLab y Energy Plus. Al igual que las herramientas que trabajan con este estándar, hay numerosas compañías que toman parte activamente en su desarrollo: Bosch, Siemens, Qtronic y Simpack.

Dentro de este contexto se ha desarrollado el proyecto de software libre JavaFMI, en el cual se ha llevado a cabo la creación del wrapper JavaFMI con el fin de permitir ejecutar FMUs desde Java. Así mismo, se han abarcado otros requisitos de interoperabilidad en el mismo ámbito. Nuestro proyecto se encuentra inmerso dentro de éste, con el objetivo de realizar una aportación a la necesidad de permitir la ejecución distribuida de FMUs.

2.2. Estado del arte

2.2.1. JFMI

Antes de desarrollar el proyecto, existían diversas herramientas que permitían importar FMUs en diversos lenguajes como Python o Java. La única herramienta compatible con Java era JFMI [3].

JFMI 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 sus usuarios controlen con detalle todo el estándar, incluidos sus detalles más complicados. Además, necesita mucho conocimiento sobre Java Native Interface y, dado que son aspectos de muy bajo nivel, no es algo normal entre los programadores Java.

2.2.2. Herramientas para la ejecución distribuida de FMUs

Las Functional Mock-Up Interfaces tienen usos en diferentes aplicaciones de simulación distribuida y cosimulación. A continuación se muestran herramientas enfocadas, en mayor o menor medida, a la ejecución distribuida de FMUs. La mayoría de ellas son herramientas comerciales por lo que no existen grandes detalles de cómo funcionan internamente. Algunas de estas herramientas son:

Método de Transmisión Lineal

El *Método de Transmisión Lineal* (Transmission Line element Method, TLM), es un método para el particionado de modelos introduciendo retardos de tiempo motivados físicamente. En sistemas físicos, la propagación de información siempre queda retrasado por capacitadores. El concepto con elementos de transmisión lineal consiste en reemplazar estos capacitadores en el modelo con dichos elementos de transmisión lineal, modelados como impedancias características. Este método hace posible mantener una

propagación de onda precisa, algo que no es posible utilizando simplemente retardos de tiempo.

Cuando dos programas se comunican utilizando TLM, es importante que las variables de cada conexión queden claramente especificadas. El usuario puede hacer eso manualmente al importar el modelo al entorno host, aunque puede resultar problemático.

Una solución alternativa consistiría en incluir esta información en la especificación XML de la FMU. Una conexión TLM queda definida con cuatro variables: intensidad, flujo, variable de onda, e impedancia característica. Dependiendo del dominio físico algunas FMUs pueden necesitar otras variables como la posición o la inercia para las conexiones mecánicas. Incluso cuando encontramos similaridades entre diferentes dominios, desafortunadamente no podemos utilizar una definición general; el conjunto de variables siempre necesitará estar codificado a priori dependiendo del tipo de conexión. Por ejemplo, un fluido incomprimible necesita sólo una variable de flujo, mientras que uno comprimible podría necesitar variables para el flujo de masa y el flujo de volumen.

Atego Ace

Atego Ace es una plataforma para la integración virtual de las funciones y componentes de software de la industria del automóvil. Este programa se encarga de realizar la co-simulación de los componentes mecánicos simulados. Permite la integración virtual de sistemas de software distribuidos en ordenadores estándar mientras que mantienen el código preparado para sistemas empotrados, expandiendo las opciones de simulación para cubrir la implementación software.

Los diseñadores de sistemas pueden validar esquemas de arquitecturas, funciones y redes de funciones antes de convertirlos en hardware. Los probadores tienen acceso a una tecnología eficiente para llevar a cabo sus pruebas en las etapas iniciales, y luego incorporarlas sucesivamente, antes de que el hardware esté disponible. Con la flexibilidad que ofrece la integración virtual, los integradores de sistemas pueden realizar las etapas de integración de forma más precisa y controlar mejor sus pruebas.

CosiMate

CosiMate utiliza una arquitectura abierta basada en un bus de co-simulación para interconectar diversas simulaciones entre sí. Esta solución ofrece dos grandes ventajas: una arquitectura abierta que permite la integración y comunicación en todos los puntos de la misma con simuladores heterogéneos, y la habilidad de simular modelos a través de la red, optimizando el uso de la CPU y el rendimiento de la simulación.

Además de esto, CosiMate proporciona, entre otras características, un entorno de testeo que integra debuggers de C/C++ además de otras herramientas especializadas; una plataforma de verificación, resultado de la capacidad de co-simular un modelo a diferentes niveles de abstracción, que proporciona un entorno completo para verificar la funcionalidad de un modelo no regresivo, y una arquitectura de red adaptada a equipos de desarrollo multidisciplinarios, que permiten particionar un modelo y distribuir la simulación de sus partes en distintos ordenadores. Está disponible para sistemas Windows y es compatible con gran cantidad de simuladores.

Hopsan

Hopsan es un entorno de simulación gratuito para sistemas de fluidos y mecatrónicos, desarrollado en la Universidad de Linköping. Fue desarrollado originalmente para la simulación de sistemas de fuerzas con fluidos, pero ha sido adaptado también a otros dominios como la corriente eléctrica, dinámicas de vuelo y dinámicas de vehículos. Utiliza líneas de retardo bidireccional (o Transmission Line Elements, TLM) para la conexión de diferentes componentes.

La generación actual de Hopsan consiste de dos partes: una interfaz gráfica y un núcleo de librerías para simulación. El usuario puede crear sus propios modelos y compilarlos como librerías separadas, que pueden ser cargadas desde Hopsan. También incluye un generador de componentes automatizado que se basa en ecuaciones que utilizan la sintaxis de Modelica. Los modelos también pueden ser generados a partir de ecuaciones usando Mathematica. Ofrece también optimización numérica a través de una

herramienta propia. Los modelos generados por Hopsan pueden ser exportados a Simulink, y los datos, exportados a XML, CSV, gnuplot y MatLab. Hopsan es un proyecto multiplataforma, con la intención de que pueda ser ejecutado en sistemas Windows, Unix y Macintosh; sin embargo, la beta actual sólo funciona en Windows.

ICOS

ICOS (Independent CO-Simulation environment) consiste en una plataforma de co-simulación, diseñada como un elemento integrador dentro de la simulación de los diferentes dominios incluidos en la automoción, permitiendo simular un vehículo virtual. Esta plataforma puede simular problemas mecatrónicos en el área de seguridad integrada o en electromovilidad, o en la compensación activa de las vibraciones del tren de transmisión y resolverlos de manera eficiente. Además, el vehículo no es el único enfoque de las simulaciones, sino también las influencias ambientales y el comportamiento del conductor con el modelo del vehículo.

La observación de la interacción entre dominios permite estimar la influencia de la calidad de modelo y la profundidad de modelado, y detectar las no linealidades en el resultado total a lo largo del proceso de desarrollo. Un enfoque interdisciplinario para las diferentes áreas que forman parte del vehículo permite comprobar cómo los diferentes elementos funcionales interaccionan entre ellos.

TWT Co-Simulation Framework

El marco de trabajo de co-simulación TWT es una herramienta para la configuración y puesta en práctica de simulaciones distribuidas. Simulaciones individuales se conectan entre sí mediante una serie de señales definidas, y por lo tanto, permiten la ejecución de las mismas en diferentes equipos de una red. Además, exportando modelos compatibles con FMI (FMUs), se pueden conectar a simulaciones externas a la red. Esto facilita, por ejemplo, la solución de problemas, ya que cada modelo se ejecuta en su respectiva herramienta de simulación. Además la simulación en línea permite la fijación de simulaciones no funcionales, tales como el análisis estructural, sistemas multicuerpo o mecánica de fluidos.

Permite la conexión de FMUs y de diferentes herramientas de simulación, como MatLab, Simulink, StarCCM+, Dymola, Modelica y Qucs. Además, posee un protocolo propio para la protección de las conexiones, TWT FMTC (Functional Mock-up Trust Center), protegiendo la propiedad intelectual.

Capítulo 3

Resultados

Con este proyecto se ha conseguido dar soporte a la ejecución de FMUs de forma distribuida. El producto software construido, llamado Remotifier, forma parte del proyecto open source JavaFMI.

JavaFMI es un proyecto de software libre que se distribuye bajo licencia LGPL V2.1 [D]. En él, se han construido diversos componentes que cubren algunas necesidades identificadas entorno al estándar FMI. Es un proyecto realizado en su mayoría en lenguaje Java que se ha desarrollado principalmente atendiendo a la práctica de ingeniería de software conocida como desarrollo dirigido por pruebas.

Para el control de versiones se ha utilizado la herramienta de software GIT. GIT es un software de control de versiones pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente. El uso de este software de control de versiones se ha realizado mediante el método GitFlow [B]. Este método está pensado para el desarrollo simultáneo de distintas tareas y se basa en la existencia de dos ramas principales (master y develop) y la creación de nuevas ramas, según funcionalidad, permitiendo así su desarrollo simultáneo.

Todo el proyecto se encuentra en un repositorio Bitbucket donde podemos encontrar tanto el código fuente del mismo, (`https://bitbucket.org/siani/javafmi/src`) como las realeases generadas de los distintos componentes que lo forman (`https://bitbucket.org/siani/javafmi/downloads`).

A lo largo de este apartado vamos a cubrir aspectos como la arquitectura del Remotifier, su diseño modular y el manual de usuario.

Los apartados que vamos a cubrir en esta sección son:

- Arquitectura.
- JavaFMI.
- Manual de Usuario.
- Diseño Modular.

3.1. Arquitectura

La definición de la arquitectura es una parte fundamental en el desarrollo de software. Define a grandes rasgos los módulos que van a formar parte en el proceso permitiendo así, alcanzar el objetivo principal del proyecto.

El resultado de este proyecto es una arquitectura para ejecutar de forma distribuida las FMUs. A esta arquitectura la hemos llamado **Remotifier**. El proceso de ejecutar una FMU se lleva a a cabo mediante la creación de Mirrors. Éstos permiten la conexión con el nodo remoto en el que se encuentra la FMU enviada por el Remote Builder.

La arquitectura actual del proyecto se representa en la figura 3.1. En ella podemos identificar los siguientes componentes:

- El **Container**, una aplicación que se encarga de gestionar las peticiones de alojamiento de FMUs en la máquina. A la vez, instancia cada una de las FMUs alojadas y asocia un Skeleton a cada una de ellas.
- El **Skeleton** es una aplicación que tiene la función principal de actuar como puerta de enlace entre un Mirror y la instancia de la FMU que tiene asociada, permitiendo las conexiones en remoto.

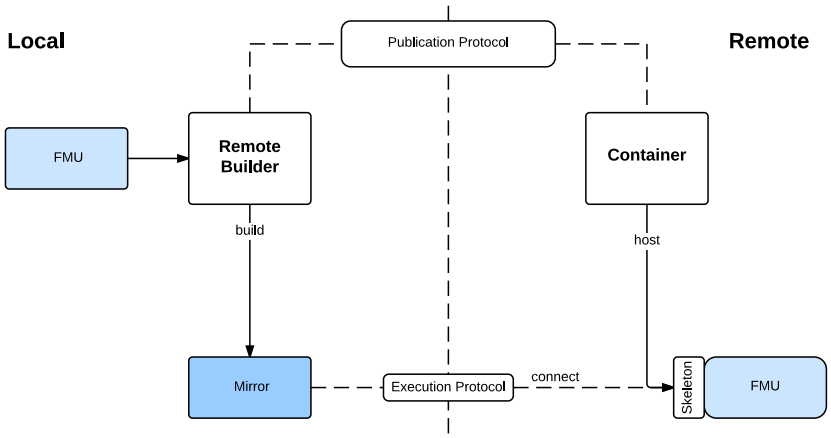

Figura 3.1: Estructura del Remotifier

- El **RemoteBuilder**, es la aplicación encargada mandar la FMU a ejecutar al ordenador remoto y crear la FMU espejo de la misma.
- El **Mirror** o FMU espejo es la FMU creada por el Remotifier que se encarga de la conexión con el Skeleton permitiendo así, la comunicación en remoto con la FMU.

Así mismo se identifican dos protocolos de comunicación:

- El **Publication Protocol**. Protocolo encargado de gestionar la comunicación entre el *Remotifier* y el *Container*.
- El **Execution Protocol**. Protocolo encargado de la comunicación entre el *Mirror* y el *Skeleton*.

3.1.1. El Remote Builder

El *Remote Builder* es una aplicación Java cuya función principal es construir una FMU espejo a partir de una FMU real. El término FMU espejo se refiere a una FMU

que a efectos del usuario se utiliza de igual manera que una FMU corriente, pero a efectos de ejecución, ésta se conecta a un nodo remoto en el que se ejecuta la FMU real. De este modo, se consigue descargar del cómputo al ordenador del usuario.

Como podemos observar en la figura 3.1, el *Remote Builder* espera como parámetros de entrada una FMU y devuelve una FMU espejo. Para llevar a cabo este procedimiento de creación de una FMU espejo, fue necesaria la construcción de lo que denominamos una FMU espejo parametrizable.

Como se ha explicado anteriormente, una FMU está compuesta fundamentalmente por una descripción de la simulación y por una serie de ficheros binarios y/o el código correspondiente en lenguaje C. Esta FMU espejo parametrizable consiste en una serie de librerías dinámicas precompiladas para cada sistema operativo, que permiten el cambio de los parámetros referentes a la dirección IP y el puerto del nodo remoto. En las librerías, se define una API con las funciones del estándar FMI para que sean capaces de gestionar el paso de mensajes. Asimismo, dentro de la aplicación *Remote Builder* existen métodos que permiten incluir el fichero XML de la FMU real en la nueva FMU espejo. Una descripción más detallada del los procesos de creación de las librerías dinámicas y de modificación de las mismas se encuentra explicados en las secciones 4.3 y 4.4 respectivamente.

3.1.2. El Container

El *Container* es la otra aplicación necesaria para poder llevar a cabo el proceso de ejecución en distribuido de una FMU. Si el Remote Builder se ejecuta en el lado cliente de la comunicación, esta aplicación se ejecuta en el lado servidor.

El Container actúa como servidor gestionando las peticiones realizadas por un Remote Builder. Dentro de estas gestiones, se encuentra tanto el almacenamiento de la FMU a ejecutar en remoto como la selección del puerto de la máquina por la que la FMU espejo resultante se vaya a conectar.

Una vez que el Remote Builder le envía la FMU que desea ejecutar en la máquina remota, el Container almacena dicho fichero e instancia un nuevo Skeleton asociándolo con la nueva FMU almacenada. De esta forma, se deja preparada la FMU para su completa ejecución en espera de los comandos.

3.1.3. Publication Protocol

Como se observa en la figura 3.1, este es el protocolo encargado de gestionar la comunicación entre la aplicación *Remote Builder* y la aplicación *Container*.

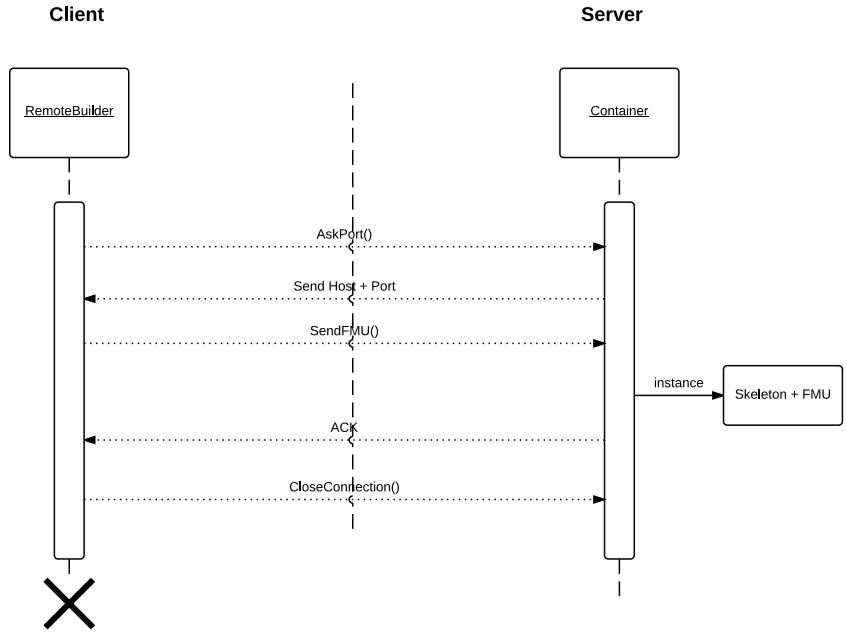

Figura 3.2: Publication Protocol

En la definición del protocolo se acordó que la obtención del puerto a la que la FMU espejo debe conectarse es gestionada por el *Container*, el cual se encarga de seleccionar un puerto libre en la máquina y enviársela al Remote Builder que ha realizado la petición.

Una vez el Remote Builder obtiene los parámetros necesarios para la creación la FMU espejo, éste envía al Container la FMU real que desea ejecutar en dicha máquina. Para esta comunicación se han definido una serie de comandos a enviar con los que se

definen los distintos tipos de peticiones que se pueden realizar. Este proceso se refleja en la figura 3.2.

3.1.4. Mirror o FMU espejo

El Mirror o FMU espejo es el componente resultante de la ejecución del Remote Builder.

Éste es un fichero comprimido, en formato zip con la misma estructura que una FMU real. La principal diferencia la encontramos en las librerías de sistema.

En una FMU real, las librerías de sistema nos ofrecen la API del estándar FMI en las que se han definido el comportamiento de una simulación. Sin embargo, en una FMU espejo nos encontramos con misma API FMI con la peculiaridad que no se define ninguna simulación. En su lugar, encontramos el código que permite el paso de mensajes entre una aplicación remota que se encuentra esperando en un puerto determinado. De esta forma, cuando se ejecutan comandos sobre esta FMU, estos son enviados a la máquina donde se encuentra la FMU real asociada.

3.1.5. Skeleton

El *Skeleton* es una aplicación que actúa como intermediaria entre la FMU real y las posibles peticiones de comandos generadas por una FMU espejo.

Esta aplicación es la encargada de instanciar la FMU asociada. Asimismo, gestiona las peticiones de comandos recibidas a través de una FMU espejo y los envía a la FMU que tiene asociada.

Para ejecutar las FMUs, el Skeleton hace uso de la librería JavaFMI [2]. Es el encargado de deserializar los comandos y ejecutarlos en la FMU que se encuentra en la misma máquina. Una vez finalizado cada comando, el Skeleton envía la salida a la FMU espejo con la que se encuentra conectado.

3.1.6. Execution Protocol

Para las conexiones entre la FMU espejo y el Skeleton asociado a la FMU real, fue necesario definir otro protocolo al que denominamos Execution Protocol.

En él, se definió el proceso de serialización/deserialización de los comandos a enviar al Skeleton. De esta manera, el Skeleton puede recibir los comandos desde la FMU espejo y ejecutarlas en la FMU del nodo remoto. Un ejemplo de comunicación de este protocolo queda plasmado en la figura 3.3.

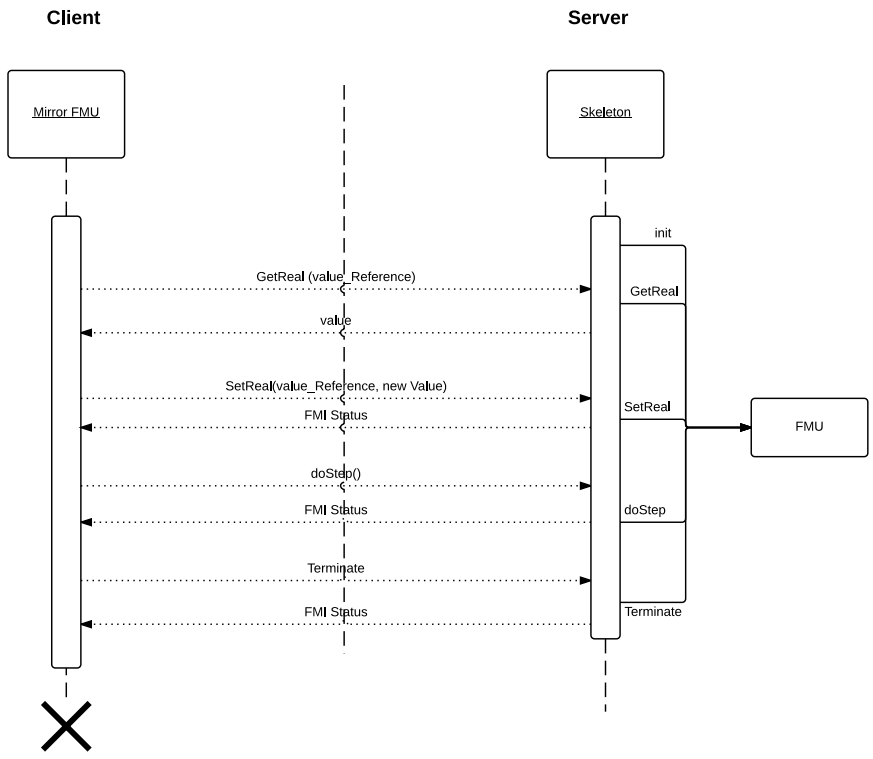

Figura 3.3: Protocolo Comunicación entre la FMU espejo y la FMU en remoto

3.2. JavaFMI

JavaFMI [2] es un proyecto open source con licencia LGPL V2.1 y se encuentra alojado en `https://bitbucket.org/siani/javafmi`. Sus inicios surgen ante la necesidad de poder ejecutar FMUs en aplicaciones Java de una forma sencilla e intuitiva.

Antes de su desarrollo, existía otra librería para ejecutar FMUs en Java llamada JFMI [3] desarrollada por la Universidad de Berkeley. Ésta ofrecía una interfaz muy poco intuitiva que dificultaba su uso. Con la intención de mejorar la experiencia del usuario se comenzó a desarrollar JavaFMI empezando con el desarrollo del Wrapper JavaFMI.

Con posteriores feedbacks y estudiando los problemas y carencias existentes en este ámbito, se empezaron a desarrollar otros componentes con el fin de solventar dichas carencias. Con todo ello, JavaFMI contiene dichos componentes:

- **Wrapper**. 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.
- **Builder**. Tiene la responsabilidad de construir FMUs a partir de simulaciones realizadas en lenguaje Java. Para ello, se ofrece un framework gracias al cual cualquier simulación en Java puede ser empaquetada como FMU.
- **Remotifier**. La arquitectura que permite la ejecución distribuida de FMUs.

3.3. Manual de Usuario

El Remotifier en sí, se compone de dos grandes bloques a nivel de uso: un servidor para gestionar las peticiones(*Container.jar*) y un constructor para generar FMUs que se conecten en remoto (*RemoteBuilder.jar*). Todo ello, se distribuye en un fichero zip que contiene ambos componentes.

3.3.1. Lanzamiento del servidor

El primero de los ficheros jar, **Container**, es lo que en sistemas de redes hace las funciones de un servidor. Éste debe ponerse en ejecución en la máquina elegida para ejecutar, también conocida como máquina remota y alojar las FMUs. El modo de uso consiste en lanzar en la máquina remota el *Container.jar*, para que éste se quede en espera de peticiones de FMUs a ejecutar en esa máquina.

```
java -jar Container.jar
```

3.3.2. Generar la FMU espejo

El segundo de ellos, el **Remote Builder**, es un constructor encargado de crear las FMUs espejo. Las FMUs espejo son FMUs que han sido personalizas para que se ejecuten en otra máquina. Los requisitos necesarios para su uso son: por un lado, tener ejecutando en la máquina remota el *Container*, y por otro, tener identificada la dirección IP de la máquina donde se ejecuta.

Para utilizar esta herramienta, es necesario ejecutar este comando:

```
java -jar RemoteBuilder.jar <FMU a ejecutar> <IP del Container> <(opcional) puerto>
```

A la hora de llamar a esta aplicación, el puerto es un parámetro opcional. Si cuando se manda ejecutar en la máquina remota, una FMU es de interés que esta conexión se haga por un puerto determinado, es cuando es necesario indicarlo. Si en la máquina remota, este puerto está ocupado, devolverá un error. Si la ejecución termina correctamente, se habrá creado en el mismo directorio donde se ha ejecutado el comando, un nuevo fichero *.fmu*, con la estructura:

```
mirror-[IP del Container]-[Puerto Asignado]-[Nombre de la FMU].fmu
```

Otro posible uso, quizá menos habitual, es utilizar el RemoteBuilder dentro del código, lo que significa utilizar el jar como librería. Para este fin, se ofrece la interfaz con una única llamada *build*, la cual devuelve un objeto tipo FILE, correspondiente a la FMU espejo. Si quisiésemos utilizarla, éste sería el código correspondiente a incluir en nuestra aplicación:

```
import org.javafmi.fmuremoter.builder.FmuRemoteBuilder;
...
   File stubFMU = new RemoteFmuBuilder().build("fmuFilePath", "192.168.1.100");
...
```

3.3.3. Utilización de las FMUs

Las FMUs resultantes, a efectos de usuario, son utilizadas de la misma forma que las FMUs no remotas. Existen varias herramientas para ejecutar dichas FMUs pero en este apartado nos centraremos en aquellas desarrolladas dentro del proyecto JavaFMI.

Consola

La forma más directa de ejecutarlas es utilizando la consola que se encuentra en el repositorio de *JavaFMI*. Hay que tener en cuenta, que en este método de ejecución, el uso de las funciones de la interfaz FMI está acotado, ya que se definen las funciones esencialmente necesarias para llevar cabo una simulación. Dichas funciones son:

- *init*. Se encarga de inicializar la simulación.
- *get [nombre variable]*. Muestra por pantalla el contenido de la variable referida en el tiempo t de la simulación.
- *set [nombre de variable] [nuevo valor]*. Inicializa la variable al nuevo valor.
- *step [Tamaño]*. Se llama a la función doStep del estándar FMI.
- *multistep [Tamaño] [ciclos de tiempo]*. Se llama a la función doStep tantas veces con ciclos de tiempo se definan.
- *reset*. Resetea la simulación al punto de inicio.
- *terminate*. Finaliza la simulación.

También existe el comando *help* para indicar los comandos y cómo utilizarlos dentro de la consola.

Wrapper JavaFMI

Si nos centramos en usar las FMUs con el wrapper JavaFMI, éste muestra dos formas de ejecución. Para la primera de ellas, se ofrece la clase *Simulation*, que da la posibilidad de ejecutar las FMUs de una manera sencilla sin entrar en detalles muy técnicos sobre la interfaz FMI. Lo único que se necesita es la dirección de la FMU a usar.

```
    ...
    Simulation simulation = new Simulation(COUNTER_FMU);
    simulation.init(0);
    simulation.writeVariable("counter", 2.3);
    for (int i = 0; i < 1000 ; i++) {
        simulation.doStep(1);
    }
    Variable counter = simulation.readVariable("counter")
    ...
```

La segunda de las opciones ofrecidas para poder ejecutar las FMUs está enfocada a usuarios más expertos que deseen utilizar un mayor nivel de detalle en la interfaz FMI. La clase que deberíamos invocar, si este fuera nuestro objetivo, es la clase *Access*. En ella se ofrece un rango más amplio de funciones en la que no se generalizan ninguna de ellas. Es decir, si en la clase Simulation tenemos definida la función writeVariable para ejecutar los setters, sin importar el tipo (String, double, int, boolean), en esta variante el usuario debe de ser consciente del tipo al que va a inicializar la variable, y llamar a la función correspondiente.

```
    ...

    Simulation simulation = new Simulation(fmu.getPath());
    Access access = new Access(simulation);
    access.enterInitializationMode();
    access.exitInitializationMode();
    access.setReal(new int[]{0}, new double[]{2.3})
    for (int i = 0; i < 1000 ; i++) {
        simulation.access.doStep(0, 1.);;
    }
    double counter = access.getReal(0)[0]);
    ...
    access.terminate();
    access.freeInstance();
...
```

3.4. Diseño Modular

El diseño modular es una de las metodologías más empleadas en programación. Está basada en la técnica de resolución de problemas: divide y vencerás. Consiste en dividir el algoritmo en unidades más pequeñas sucesivamente hasta que sean directamente ejecutables en el ordenador. Es una gran ayuda a la hora de satisfacer los requisitos y servir como guía en el desarrollo.

La estructura actual se encuentra referenciada en la figura 3.4. En ella, podemos observar como en el *Remotifier* hace uso del wrapper *JavaFMI*.

A su vez, se identifican cuatro módulos pertenecientes al Remotifier. Dentro del proyecto, vamos a diferenciar dos funcionalidades que van a explicar la existencia de cada módulo:

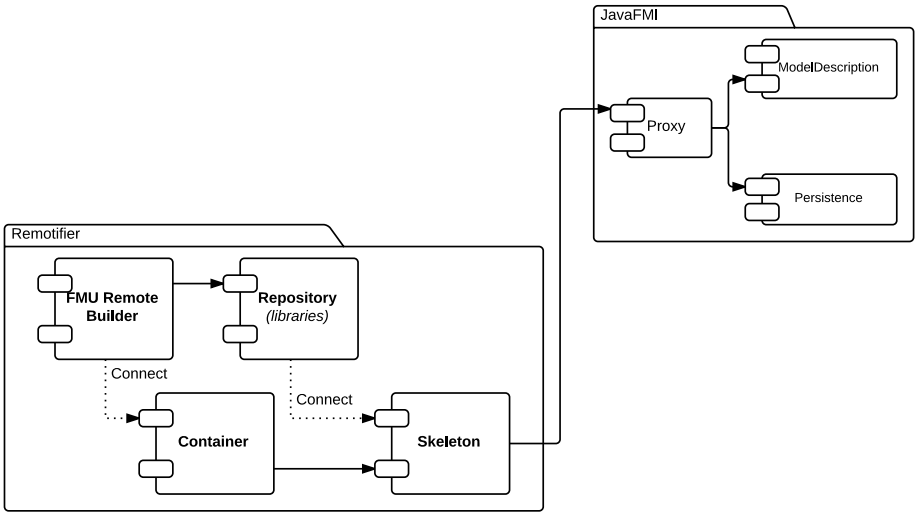

Figura 3.4: Diseño Modular del Remotifier

- **Creación de la FMU remota**. Los agentes que intervienen en la creación de las FMU remota son el módulo FMU Remote Builder y el Container.
- **Conexión y ejecución de FMUs creadas**. Los módulos encargados de dicho objetivo son los módulos Dynamic Library Repository y Skeleton.

  A continuación procederemos a la explicación de las estructuras utilizadas para cada uno de los módulos.

3.4.1. Fmu Remote Builder

Como su propio nombre indica, este módulo es el encargado de construir FMU remotas. Su uso está pensado en conjunto con el Container ya que tiene la funcionalidad de conectarse a él, para enviar la FMU que desea ejecutar en la máquina remota y consultar el puerto asignado. El Container se explica más detalladamente en la sección 3.4.3.

En la figura 3.5 se muestra el diagrama UML de este módulo. En ella podemos observar cómo las distintas funciones del proceso se encuentran repartidas en clases, haciendo uso de uno de los principios SOLID: la responsabilidad única de clases.

De esta forma, nos encontramos con la clase *FMU Remote Builder*. Ésta hace uso de la clase *SocketUtils*, encargada de aportar los procedimientos necesarios para permitir la conexión con el Container. Con gestionar la conexión nos referimos a la petición de alojamiento de la FMU no remota, obtención de la dirección y puerto de escucha y posterior envío de la FMU a ejecutar en la máquina remota al servidor (Container).

Del mismo modo, vemos como una clase FMU Remote Builder tiene agregada una del tipo FMU Builder. Esta clase tiene la responsabilidad de definir la estructura interna de una FMU (ya sea remota o no) y de crear el fichero comprimido correspondiente, es decir la FMU.

El FMU Builder nos aporta funciones para añadir las librerías de sistema correspondientes. Por ello existe la clase FmuBuilderHelper, encargada de gestionar las librerías dinámicas precompiladas, que están preparadas para la conexión en remoto a un servidor en una dirección y puerto predeterminados. Como es de esperar que dichos

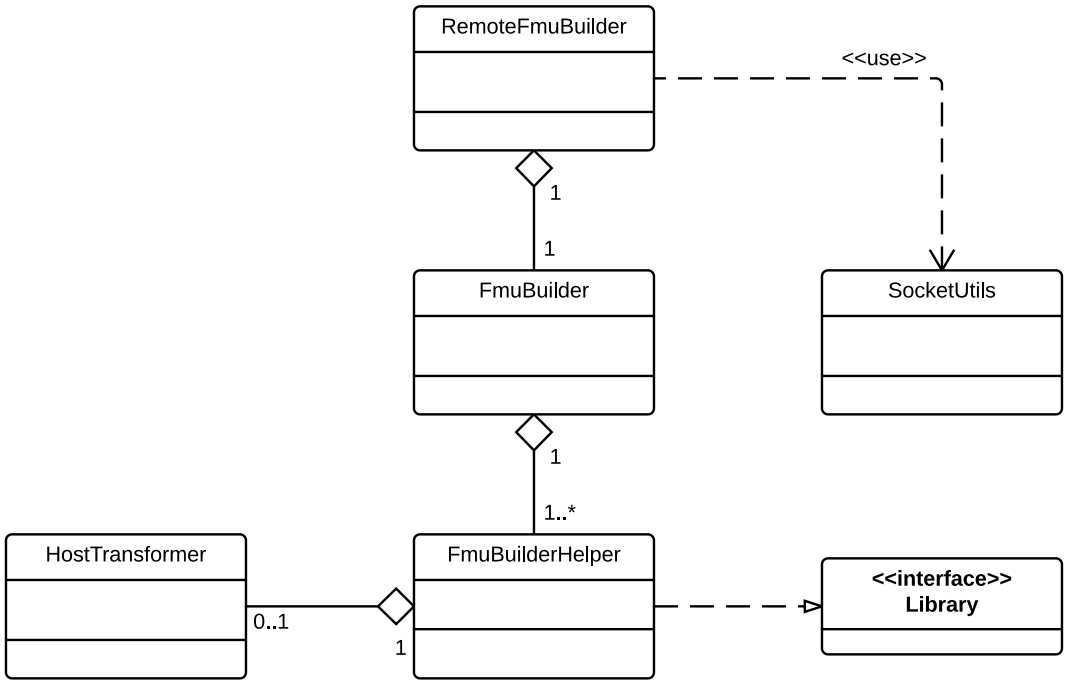

Figura 3.5: Diagrama de Clases del Fmu Remote Builder

parámetros cambien, las clases FmuBuilderHelper y HostTransformed son las encargadas de gestionar los ficheros binarios, modificándolos. Para mas información sobre este proceso, se puede consultar la sección 4.4.3 dentro del *Proceso de Desarrollo*.

3.4.2. Repositorio de las librerías dinámicas

Este módulo hace referencia a las librerías dinámicas resultantes de su compilación en C, para todos los sistemas operativos que permitan no solo el uso de una FMU, sino también su conexión a la FMU alojada en el servidor.

Es la parte del proyecto más compleja y más sensible a cambios ya que al ser C un lenguaje que se ejecuta sobre la máquina, es necesario utilizar distintos tipos para la

gestión de los sockets dependiendo de si estamos hablando de plataformas Windows o Unix.

Es verdad que existen librerías para sockets que prometen un uso idéntico para todas las plataformas tales como *libcURL*, estudiada en la elección de herramientas. Como se indica en la sección 4.3.1 en la que podemos ver la gráfica 4.1 donde se comparan los tiempos de transferencias de distintos tamaños de paquetes utilizando la librería *libCurl* frente al uso de los sockets creados con las librerías de sistema. El uso de esta herramienta quedó descartado por la lentitud que añadía al paso de mensajes, tal como se muestra en la figura anteriormente referenciada. Cuando hablamos de conexiones remotas, va implícito el hecho de que su ejecución va a ser más lenta, por lo que debemos utilizar aquellas herramientas que intenten minimizar ese problema. Con todo ello, se eligieron los sockets de las librerías de sistema, winsock para plataformas Windows y sockets para plataformas Unix.

La estructura utilizada siguió el mismo criterio que los anteriores componentes, dividiendo los contenidos en carpetas según su responsabilidad. Así nos encontramos con la siguiente estructura de carpetas:

- **fmi**. Están la totalidad de los headers referentes al estándar FMI.
- **networks**. Se encuentra el fichero cabecera Connection. Este, mediante compilación condicional, incluirá la librería correspondiente a los sockets según el sistema operativo. Del mismo modo, aquí se encuentran las implementaciones de los sockets para las dos plataformas.
- **script**. Se encuentran el fichero que permite la compilación de las librerías según el tipo de sistema operativo.
- **StubFMU**. Ésta es la implementación de las librerías referentes al estándar, rellenado cada método, con el código que hace posible la utilización de las librerías en remoto mediante los protocolos definidos.

Cabe destacar que son librerías genéricas que luego dado el caso particular, el FMU Remote Builder se encarga de particularizar.

3.4.3. Container

El módulo Container es el encargado de actuar como servidor. Este posee las funcionalidades de:

- Gestionar los puertos libres de la máquina y asignar un puerto libre a cada petición.
- Alojar las FMUs a ejecutar en remoto.
- Lanzar un Skeleton correspondiente para cada FMU.

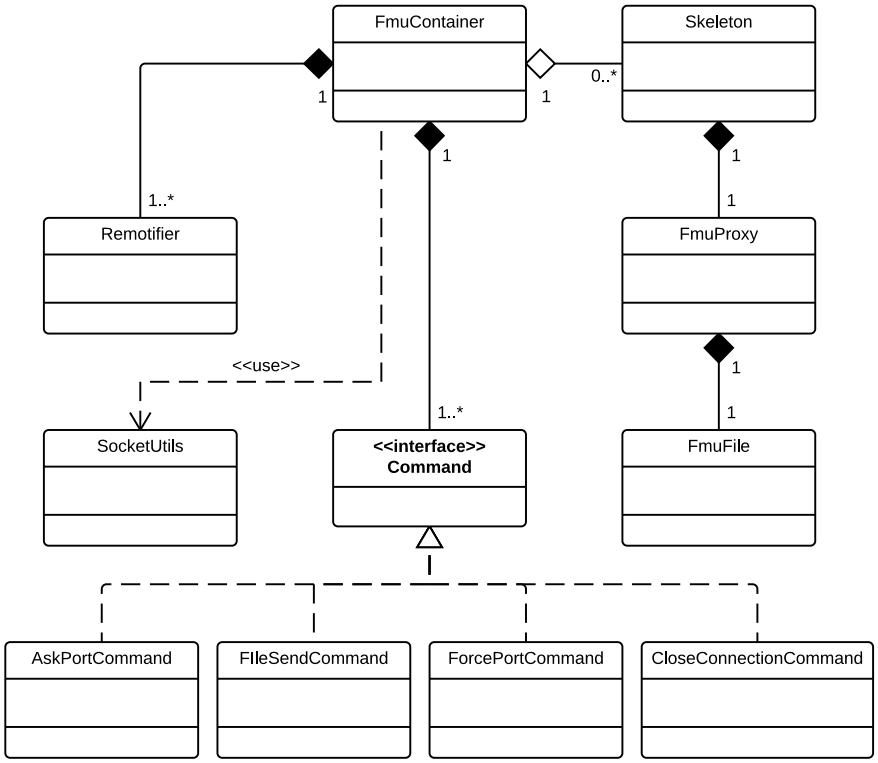

Figura 3.6: Diagrama de Clases del Container

Como podemos observar en el diagrama 3.6, el FMU Container esta compuesto de 4 tipos de implementaciones de la interfaz Command al igual que las instancias de la clase Remotifier.

La clase Remotifier tiene la función de gestionar las peticiones de conexión al servidor, creándose uno nuevo por cada petición. Del mismo modo, las peticiones que puede recibir el Container, son de la clase Command, existiendo cuatro implementaciones de la misma con distintas funcionalidades:

- **AskPortCommand**. Si se recibe dicha petición, el FMU Remote Builder esta gestionando la petición de alojamiento de la FMU, realizando una petición de puerto libre al Container. El Container procederá a la asignación y envío de la dirección IP y puerto.
- **FileSendCommand**. Si se recibe este commando, el FMU Remote Builder ha procedido al envío de la FMU a ejecutar en la máquina remota. El Container procede al almacenamiento de la misma.
- **ForcePortCommand**. Función opcional para obligar al Container a utilizar un puerto determinado, si en todo caso este se encuentra ocioso.
- **CloseConnectionCommand**. Es enviado cuando el FMU Remote Builder ya tiene todos los datos necesarios para alojar la simulación o bien cuando le ha venido denegada alguna petición.

Por otro lado vemos que existe una agregación con el Skeleton. Esto es así ya que, después de finalizar una petición, se lanza un Skeleton cargando la FMU asignada y escuchando por el puerto designado anteriormente.

3.4.4. Skeleton

El Skeleton se construyó como una puerta de enlace entre una FMU y una conexión remota, ya que una FMU no posee los medios para comunicarse entre sockets por sí sola.

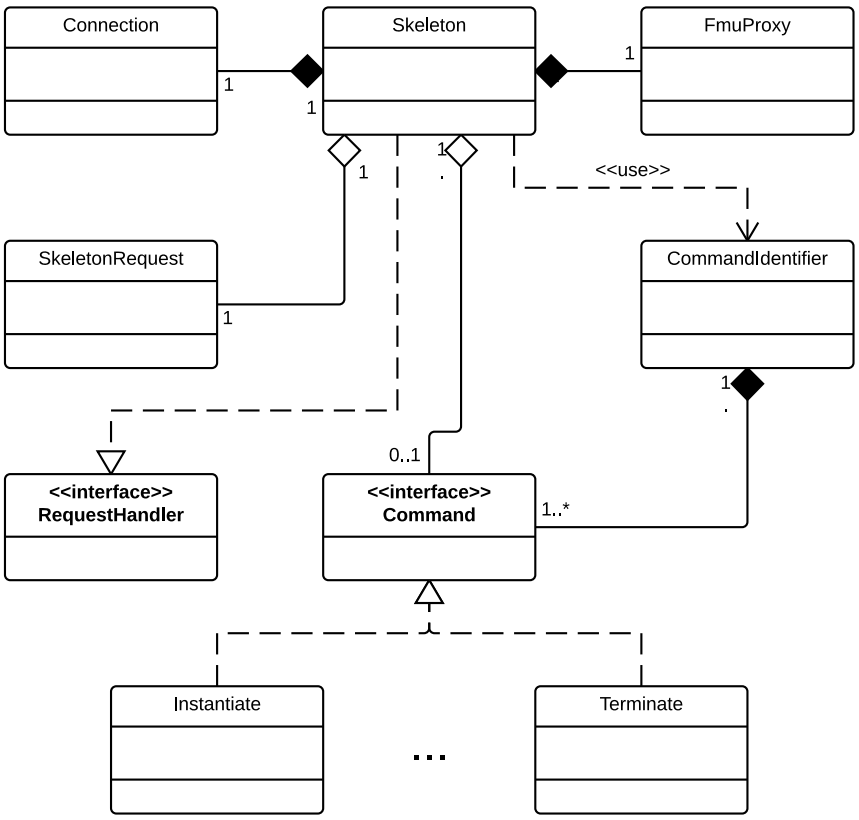

Figura 3.7: Diagrama de Clases del Skeleton

Podemos observar en el gráfico 3.7 como las clases fundamentales para el desarrollo y existencia del módulo son la clase Connection, que como su propio nombre indica, se encarga de gestionar el envío y recepción de los datos mediante sockets y la clase FMUProxy.

Esta última clase, el FMU Proxy, es la puerta de entrada a utilizar el wrapper JavaFmi que permite la ejecución de las FMUs.

También vemos como existe una agregación con la clase SkeletonRequest. Esta clase tiene la responsabilidad de deserializar el comando enviado. Una vez deserializado, el

Skeleton hace uso de la clase CommandIdentifier para identificar a cual de las implementaciones de la clase Command se hace referencia, y llamarla mediante Java reflection. Cada implementación de la clase Command posee un método execute el cual llama al método correspondiente del wrapper JavaFMI.

Capítulo 4

Proceso de desarrollo

4.1. Metodología de desarrollo

En el desarrollo del proyecto, se utilizado una filosofía ágil. El desarrollo ágil consiste en métodos de software basados en el desarrollo iterativo e incremental, donde los requisitos y soluciones evolucionan mediante la colaboración de grupos multidisciplinarios.

La mayoría de los métodos ágiles existentes minimizan los riesgos desarrollando el software en periodos cortos de tiempo. El software desarrollado en una unidad de tiempo se llama iteración y cada una incluye las etapas de planificación, análisis de requisitos, diseño, codificación, revisión y documentación. El objetivo de cada iteración es tener un producto mínimo viable y testado al que se le irá añadiendo más funcionalidad en las siguientes iteraciones del proceso de desarrollo.

4.2. Primera etapa: Familiarización con herramientas y metodologías de desarrollo

Al definir los objetivos del proyecto, acordamos que el desarrollo del mismo se realizaría principalmente en los lenguajes Java y C. A lo largo de los 5 años de carrera,

hemos trabajado intensivamente con lenguajes como C o C++, pero no tanto en Java, considerando que la experiencia en este lenguaje no era suficiente.

Una gran parte de la primera etapa consiste en una familiarización con este lenguaje, así como reconocer y llevar a cabo los principios SOLID.

SOLID En ingeniería de software, SOLID es un acrónimo introducido por *Robert C. Martin* a comienzos de la década del 2000 que representa cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar en el tiempo. Los principios SOLID son guías que pueden ser aplicadas en el desarrollo de software para eliminar código sucio, provocando que el programador tenga que refactorizar el código fuente hasta que sea legible y extensible. Debe ser utilizado con el desarrollo guiado por pruebas o TDD, y forma parte de la estrategia global del desarrollo ágil de software y programación adaptativa. Los 5 principios son:

- Principio de responsabilidad única (***S**ingle responsibility*).
- Principio abierto/cerrado (***O**pen-Closed*).
- Principio de sustitución de Liskov (***L**iskov substitution*).
- Principio de segregación de interfaces (***I**nterface segregation*).
- Principio de inversión de dependencia (***D**ependency inversion*).

Desarollo dirigido por pruebas (TDD) De la misma manera, también fue necesaria la familiarización con el **desarrollo dirigido por pruebas**, más conocido como **TDD** (Test-Driven Development). Es una práctica de programación orientada a objetos que involucra otras dos prácticas: escribir las pruebas primero (Test First Development) y refactorización (Refactoring). Para escribir las pruebas generalmente se utilizan pruebas unitarias. En primer lugar, se escribe una prueba y se verifica que las pruebas fallan. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente de la manera más simple posible y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas; de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.

4.3. Segunda etapa: Creación de las librerías dinámicas

Como se explica en el apartado de contextualización, la filosofía del estándar FMI se basa en las construcción de FMUs. Cada FMU contiene en su interior unas librerías dinámicas (y/o en su defecto el código correspondiente), que definen las funciones para llevar a cabo la simulación, compiladas para cada sistema operativo, aportando la API definida por el estándar.

Como nuestro objetivo era desarrollar una FMU espejo, es decir una FMU que se conectara remotamente, en la solución optada era necesario construir librerías dinámicas genéricas, que fuesen capaz de comunicarse en red y que a la vez permitiesen cambiar la dirección IP y el puerto.

4.3.1. Elección librerías para manejo sockets

Para ello se acordó que la conexión se realizaría por sockets TCP/IP.

A priori, ya sabíamos que lo más óptimo, en cuanto a generación de código, era que un mismo código funcionase en todos los sistemas y arquitecturas, a excepción del comando de compilación. Por ello, se barajaron varias opciones de librerías que permitían el manejo de sockets tanto en entorno Unix como Windows.

Algunas de las estudiadas fueron *C++ Sockets Library*, *POCO* o *libCurl*. Todas ellas ofrecían una interfaz común para el manejo de sockets amigable para el usuario. A pesar de ello y tras realizar algunas pruebas nos decantamos por las librerías para manejo de sockets que vienen por defecto con el sistema por varias razones. El tiempo de ejecución es menor que la de sus hermanas y no es necesario añadir ni compilar código extra en nuestras librerías, lo cual nos interesaba ya que no queríamos que nuestras librerías pesasen demasiado. La gráfica 4.1 es una comparativa entre los tiempos de ejecución entre la librería libCurl frente a sockets generados por las librerías de sistema. En ella, se puede observar una diferencia en más de un minuto en el intercambio de paquetes.

Por lo tanto se optó por utilizar *winSockets* para sistemas Windows y *sockets.h* para las librerías en sistemas Unix.

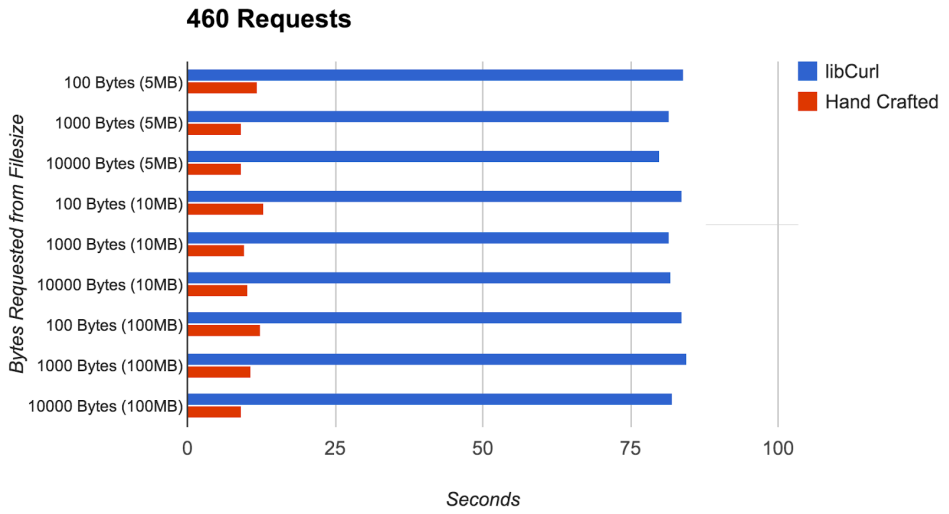

Figura 4.1: Comparativa ejecución LibCurl frente a las librerias de sistema

4.3.2. Programación orientada a objetos en C

A la hora de crear las clases para el manejo y tratamiento de los sockets en C, se optó por el procedimiento definido en el libro *Object Orientated Programming in ANSI-C* [7]. En él, se describe cómo aplicar programación orientada a objetos en lenguaje C. Este método consiste en la creación de un header (*Connection.h*) en el cual se definen las cabeceras a las funciones genéricas, sin tener en cuenta que se utilicen unos sockets u otros. Ya será a la hora de la compilación cuando se incluyan los headers necesarios.

```
Connnection.h
...
#if defined __WIN32 || defined __MSYS__
        #include <winsock2.h>
        #include <windows.h>
        #include <iphlpapi.h>
#else
        #include <sys/types.h>
    #include <sys/socket.h>
    #include <netdb.h>
#endif

extern const void* Connection;

void* newConnection(const char* host, int port);
void openConnection(void* self);
```

```
void closeConnection(void* self);
void sendMessage(void* self, const char* message);
void receiveMessage(void* self, char* buffer);
...
```

Es en la implementación de cada una de las variantes de *Connection.h* donde se define el struct que necesita cada implementación de los sockets. De esta forma, abstraemos un escalón por encima el manejo de los sockets al usuario, ya que éste sólo sabe que es de tipo Connection, independientemente de lo que haya internamente.

```
Ejemplo de unixConexion.h
#if defined __UNIX__
#include "Connection.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

struct Connection{
        const char* host;
        int port;
        int socketDescriptor;
};
...
void* newConnection(const char* host, int port){
        struct Connection* connection = malloc(sizeof(Connection));
        connection->host = host;
        connection->port = port;
        return connection;
}
...
void sendMessage(void* _self, const char* message){
        struct Connection* self = _self;
        char messageBuffer[128];
  strcpy(messageBuffer, message);
  strcat(messageBuffer, "\n");
        send(self->socketDescriptor, messageBuffer,
                                strlen(messageBuffer), 0);
}

void receiveMessage(void* _self, char* buffer){
        struct Connection* self = _self;
  char bufferToRead;
  int bufferIndex = 0;
  do{
    recv(self->socketDescriptor, &bufferToRead, 1, 0);
    buffer[bufferIndex++] = bufferToRead;
  }while(bufferToRead != '\n');
```

```
  buffer[bufferIndex] = '\0';
}
```

4.3.3. Definición del protocolo de comunicación

Al escoger cómo iba a realizarse la comunicación remota y antes de realizar su implementación, fue necesario acordar un protocolo de comunicación. Esto implicaba a su vez definir, si bien con posibilidad de cambios, una estructura global de lo que iba a ser el Remotifier en sí.

Al llegar aquí se definió la existencia de una aplicación cliente, a la que denominaríamos más adelante *Remote Builder*. Ésta debería mandar de una forma de comandos las peticiones al servidor.

Fue en este punto en donde empezamos a definir que debía existir un servidor general, el que al final denominamos (*Container*), encargado de estar siempre escuchando posibles peticiones, alojando las FMUs e informando a la FMU virtual del puerto en el que la instancia de la FMU estará esperando comandos.

A su vez debía existir otro elemento que actuase como puerta de comunicación, entre la FMU remota y la FMU alojada en el servidor, el *Skeleton*.

Conexión y obtención del puerto La obtención del puerto a la que la FMU virtual debe conectarse es gestionada por el *Container*, el cual se encarga de seleccionar un puerto libre en la máquina y enviársela al FMU Remote Builder que ha realizado la petición.

En este punto, se define que el Container será el encargado de revisar los puertos libres de la máquina y le asignará uno de ellos. Los detalles de implementación del Container se explicarán más adelante en el apartado 4.6.2, página 43.

Serialización y deserialización de la información La serialización de la información transmitida de uno a otro terminal fue otro punto a acordar. Se realizaron pruebas utilizando JSON, un formato ligero de intercambio de datos. Más concretamente, se realizaron pruebas con la librería Gson, perteneciente a Google. A pesar de que su uso está muy extendido y añade seguridad al intercambio, se consideró que la información a intercambiar no era lo suficientemente pesada y se realizó mediante el envío de una string en la cual se separa cada argumento significativo mediante espacios. Es el *Skeleton* vinculado con dicha FMU el encargado de decodificarla posteriormente e identificar el comando a llamar vía reflection.

```
public void newRequest(String requestData) {
    System.out.println(requestData);
    SkeletonRequest request = new SkeletonRequest(requestData);
    Command command = CommandIdentifier.identify(request.getCommandName());
    connection.sendResponse(command.executeOn(fmuProxy,
                                              request.getArguments()));
    if(command instanceof FreeInstance) connection.close();
}
```

4.4. Tercera Etapa. Creación del Remote Builder

Una vez generadas y probadas las librerías dinámicas genéricas cumpliendo el estándar FMI, nos centramos en el desarrollo del Remote Builder.

Para ello, nos planteamos la creación de un constructor de FMUs, al que de ahora en adelante llamaremos por su nombre en inglés, *Remote Builder*. Una FMU, como ya explicamos en el apartado de contextualización, no es más que un fichero comprimido que cumple una determinada estructura.

Teniendo en cuenta esta definición, lo primero que se llevó a cabo fue la creación de un *FMU builder*, en el cual se definió una interfaz que permitiese la creación por partes de una FMU. Funciones tales como *addLibrary*, *addModelDescription* o *addFileAsResource*, en conjunto con una llamada *buildFMU* generaban el fichero comprimido con extensión *.fmu*. Es decir, se construía una FMU.

En este punto del desarrollo, también se definió la interfaz *Library*, para permitir una mejor compatibilidad entre los futuros tipos de librerías que podían surgir a lo largo del desarrollo del proyecto en conjunto. El uso de este paquete es bastante intuitivo:

```
public void buildingFmuWithOneResource() {
        FmuBuilder fmuBuilder = new FmuBuilder();
        fmuBuilder.addLibrary(new StubFmu("libwin64.dll"), "binaries/win64");
        fmu.Builder.addModelDescription("FmiModelDescription.xml");
        fmuBuilder.addFileAsResource(new File(RESOURCES[0]));
        File fmuFile = fmuBuilder.buildFmu(FMU_NAME);
    }
```

Una vez desarrollado lo que sería el paquete encargado de la creación de una FMU según como lo dicta el estándar, nos pusimos manos a la obra en el desarrollo del constructor de FMUs espejo partiendo de una FMU determinada.

A este módulo lo llamamos FMU Remote Builder. En él, podemos encontrar dos funciones bien definidas; una primera, encargada de empaquetar la FMU con las librerías generadas en la segunda etapa del desarrollo, y la segunda, que tiene la tarea de contactar con el servidor que debemos desarrollar en la siguiente etapa. Para esta iteración, las pruebas con el servidor se hicieron mediante la librería *mockito*, con la que simulamos (moquear) una conexión servidora.

4.4.1. Empaquetado de la FMU

Para la función de empaquetar la FMU se hace uso del *Fmu Builder*. La principal diferencia que se verá reflejada entre una FMU real y una FMU espejo viene definida por los tipos de librerías dinámicas que se añadan. En este caso, se creó la implementación de la interfaz StubFmu que incluye funciones para permitir el paso de mensajes de éstas.

4.4.2. Obtención del puerto y envío de la FMU

Para la segunda función, la encargada de contactar con el servidor, se desarrolló un sistema de conexión mediante sockets en Java donde la codificación y paso de información se realizan mediante Streams. Cuando se realiza la llamada la función build, lo primero que ocurre es una conexión con los servidores, para que este le asigne un determinado puerto en su máquina. Una vez asignado, envía la FMU a ejecutar en remoto al servidor para que ésta la aloje en sus filas y la instancie en escucha en el puerto que le ha asignado.

4.4.3. Modificación de ficheros binarios

El hecho de tener precompiladas las librerías de sistema implicaba un problema a la hora de cambiar el puerto y la dirección IP de la máquina servidora. Se podía solucionar fácilmente haciendo que el usuario compilara las librerías con los nuevos valores. Pero esto era exactamente lo que no nos interesaba ya que esto aportaría poca usabilidad al producto. Por esta razón, se optó por la modificación de ficheros binarios.

En este punto, se definió en el código de las librerías dinámicas una dirección IP y puerto por defecto. Una vez compiladas, se localizaron los bytes correspondientes a la definición de la dirección IP y el puerto en distintas arquitecturas. Esta información se incluyó en la clase *StubFMU*, definiendo así la dirección IP y puerto por defecto al igual que los bytes de cada uno de estos valores. En el caso de que el servidor les asignara un puerto y/o un dirección IP distinta, las funciones *modifyPort* y *modifyhost* se encargarían de localizar los bytes correspondientes y los intercambiarían por los resultantes de la dirección IP y/o el puerto dados por el servidor. Aquí se exponen las dos funciones esenciales para este proceso:

```
private void modifyHost(InputStream originalLibrary,
                        BufferedOutputStream modifiedLibrary) {
     byte[] buffer = new byte[16];
     int readBytes;
     while((readBytes = readFromTo(originalLibrary, buffer)) != -1) {
         if(Arrays.equals(buffer, DEFAULT_HOST_BYTES)) {
             buffer = new HostTransformer().toByteArray(host);
             writeDataTo(buffer, modifiedLibrary, readBytes);
             break;
         }
         writeDataTo(buffer, modifiedLibrary, readBytes);
     }
 }
 private void modifyPort(InputStream originalLibrary,
                         BufferedOutputStream modifiedLibrary) {
     byte[] buffer = new byte[16];
     int readBytes;
     while((readBytes = readFromTo(originalLibrary, buffer)) != -1) {
         if(containsDefaultPort(buffer)) {
             writeDataTo(insertPortBytesInBuffer(buffer, port),
             modifiedLibrary, readBytes);
             return;
         }
         writeDataTo(buffer, modifiedLibrary, readBytes);
     }
 }
```

4.5. Cuarta etapa. Migración a la versión del estándar FMI v2 RC1

Cuando este proyecto empezó a desarrollarse, el estándar FMI había publicado la versión 1.0 del mismo. Por ello, el desarrollo se realizó basándose en las versión existente. Durante su desarrollo se han publicado dos versiones Realease Candidate (RC) de la versión 2.0 y la versión estable 2.0.

En un principio, no se consideró pertinente el migrar el desarrollo del Remotifier para que fuera compatible con las versiones posteriores. Pero cuando los requisitos de usabilidad empezaron a cambiar y empezaron a llegar peticiones para que existiera compatibilidad con las versiones posteriores del estándar, se consideró esta posibilidad. La portabilidad se realizó a la v2 RC1 del estándar para todos los componentes del proyecto JavaFMI (Wrapper, Builder y Remotifier).

La portabilidad nos aportó velocidad en cuanto al proceso de generar las FMUs. En la versión 1 del estándar se define que en cada función del mismo se debería añadir una etiqueta con el nombre de la FMU. De esta manera, se nos obligaba a cambiar además de la dirección IP y puerto en las librerías genéricas parametrizables, las cabeceras de cada una de las funciones del estándar, añadiendo el nombre de la FMU. Este proceso es costoso y lento, que además había que repetir en 5 librerías pertenecientes a cada sistema operativo. Por el contrario, a partir de la versión 2.0 RC1, se añadió la posibilidad de generar el código sin añadir esta cabecera a estas funciones. De esta manera, eliminamos ese procedimiento de cambiar las cabeceras de las funciones en las ficheros binarios.

Los cambios realizados incluían también modificaciones en las librerías, ya que en este nueva versión se definían nuevas funciones y desaparecían o se redefinían otras. Además, la estructura del *modelDescription.xml* cambiaba drásticamente, apareciendo nuevos campos obligatorios.

4.6. Quinta Etapa. Creación del Skeleton y Container

Una vez queda definido y probado el funcionamiento correcto de los builders, nos centramos en desarrollar la parte del servidor. En esta etapa, dividimos el problema en dos; por un lado identificamos la necesidad de encontrar una puerta de comunicación entre los sockets y una FMU estándar, y por otro la necesidad de gestionar peticiones de los FMU Remote Builders. Con estos problemas identificados, surgieron el Skeleton y el Container respectivamente.

4.6.1. Creación del Skeleton

El Skeleton nace de la necesidad de buscar una vía de comunicación entre una FMU estándar, es decir, una FMU creada por método convencional, y una comunicación en red. Por ello en este módulo fue necesario crear una conexión mediante sockets, e incluir el módulo *Proxy* del Wrapper JavaFmi para instanciar y ejecutar la FMU.

El desarrollo del skeleton se pensó de tal manera que los sockets se lanzan en un hilo distinto permitiendo así enviar más de un comando a la vez. De esta manera, se define que la forma de ejecutar e instanciar la FMU se hará por medio del Wrapper JavaFMI.

4.6.2. Creación del Container

Ante la necesidad de tener que centralizar las peticiones de FMU que se desean ejecutar remotamente, surgió el *Container*. Éste se encargaría de gestionar las peticiones de alojamiento de FMUs mediante sockets. Sus principales responsabilidades son:

- **Gestionar y asignar los puertos libres de la máquina**. Cuando el *Container*, recibe una petición de almacenar la FMU para que se ejecute en remoto, lo primero que realiza es un elección de puerto libre. En el hipotético caso que no encontrara ningún puerto libre, lanzaría un mensaje de error. La búsqueda del puerto se realiza desde Java, gestionando mediante la creación de un socket en el puerto

elegido (y posterior liberación) si éste está libre. En caso afirmativo, este puerto será enviado, de lo contrario se vuelve a realizar la búsqueda de un nuevo puerto. Entrando más en detalle, podemos decir que siguiendo la concordancia del FMU Remote Builder, la codificación de esta información se realiza en streams.

- **Almacenamiento de las FMUs**. Una vez elegido un puerto para ejecutar la FMU, el Remote FMU Builder envía la FMU a ejecutar al servidor y ésta se almacena en el directorio *workspace* que el *Container* crea al ejecutarse. Cada FMU alojada dentro del directorio se identifica por un nombre construido:

```
[Direccion IP del cliente]-[puerto]-[Nombre de la FMU].fmu
```

- **Instanciar y poner en ejecución la FMU en el puerto elegido**. La instancia de la FMU, se lleva a cabo lanzando un *Skeleton* por cada FMU que se desea ejecutar. El objetivo principal del Skeleton, como se explica en el apartado 4.6.1, es el de ejecutar y actuar como puerta de comunicación entre sockets.

4.7. Temporización

A continuación pasamos a describir los periodos de tiempo empleados en cada una de las partes de este proyecto. Podemos ver en el diagrama 4.2 el tiempo empleado por sectores:

En la figura 4.2 se muestra que el 67 % del tiempo invertido ha sido en el diseño y el desarrollo del mismo. Dentro de estos dos apartados se incluyen los tiempos dedicados al consenso llevado a cabo dentro de los miembros del equipo y los cambios referidos a la retroalimentación dada por los usuarios del proyecto JavaFMI.

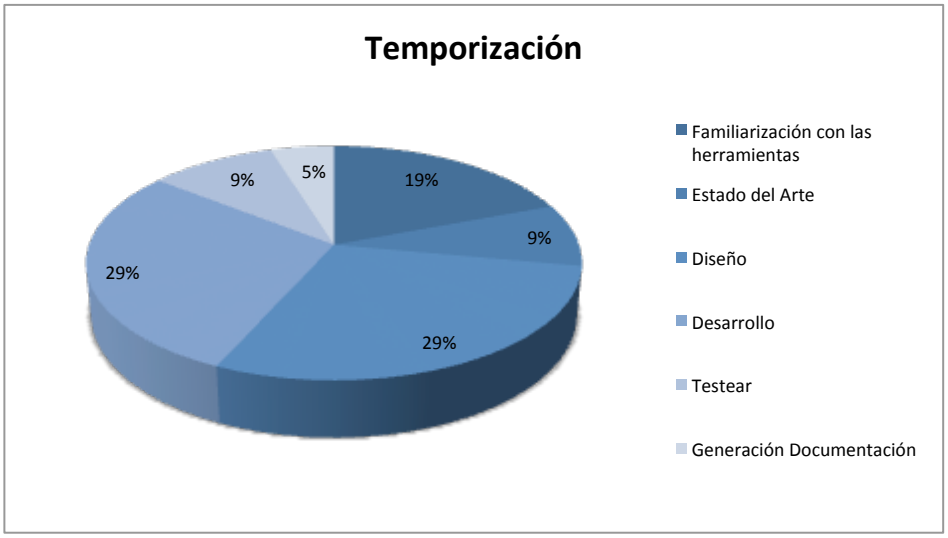

Figura 4.2: Porcentajes temporización

Capítulo 5

Herramientas Utilizadas

Durante el desarrollo del proyecto se han utilizados una serie de recursos y herramientas software y hardware que se exponen a continuación.

5.1. Recursos Hardware

Para el desarrollo del proyecto se han utilizado los siguientes equipos:

- **Equipo 1**:
  - *Sistema Operativo*: OS X 10.9.5
  - *Procesador*: Intel Core i7 2,66 GHz
  - *Memoria*: 4 GB 1067 MHz DDR3
  - *Tarjeta Gráfica*: NVIDIA GeForce GT 330M 512 MB + Intel HD Graphics 288 MB
- **Equipo 2**:
  - *Sistema Operativo*: Windows 8.1
  - *Procesador*: Intel Core i7 2,80 GHz
  - *Memoria*: 8 GB 1067 DDR 3
  - *Tarjeta Gráfica*: NVIDIA GeForce GTS 240 4Gb

5.2. Recursos Software

- **Oracle VM VirtualBox** es un software de virtualización para arquitecturas x86/amd64, creado originalmente por la empresa alemana GmbH. Actualmente es desarrollado por Oracle Corporation como parte de su familia de productos de virtualización. Por medio de esta aplicación se pueden instalar sistemas operativos adicionales, conocidos como *sistemas invitados*, dentro de otro sistema operativo *anfitrión*, cada uno con su propio ambiente virtual. La aplicación fue inicialmente ofrecida bajo una licencia de software privativo, pero en enero de 2007, después de años de desarrollo, surgió VirtualBox OSE (Open Source Edition) bajo la licencia GPL 2. Actualmente existe la versión privativa Oracle VM VirtualBox, que es gratuita únicamente bajo uso personal o de evaluación, y está sujeta a la licencia de *Uso Personal y de Evaluación VirtualBox* (*VirtualBox Personal Use and Evaluation License* o *PUEL*) y la versión Open Source, *VirtualBox OSE*, que es software libre, sujeta a la licencia GPL.

  Para el desarrollo de este proyecto esta herramienta ha sido fundamental ya que nos ha permitido realizar pruebas en los distintos sistemas operativos existentes y compilar cada librería de sistema para cada sistema operativo y arquitectura sin la necesidad de utilizar nuevos ordenadores.

- **IntelliJ IDEA** es un entorno de desarrollo integrado (Integrated Development Environment, IDE) para Java desarrollado por JetBrains. Está disponible en una edición con licencia Apache 2 y otra edición comercial. En un reportaje de la revista Infoworld del año 2010, IntelliJ obtuvo la mejor puntuación de entre las cuatro mejores herramientas de programación en Java: Eclipse, IntelliJ IDEA, NetBeans y Oracle JDeveloper.

  Este IDE ha sido el elegido para llevar a cabo el desarrollo del proyecto.

- **Cygwin** es un entorno estilo Unix e interfaz de línea de comandos para Microsoft Windows. Cygwin proporciona una navegación del sistema nativo Windows, junto con sus programas, datos y otros recursos del sistema, a la vez que aplicaciones, herramientas software y datos de un entorno tipo Unix. De esta forma es posible ejecutar aplicaciones de Windows desde el entorno de Cygwin, así como utilizar herramientas de Cygwin desde Windows. Cygwin consiste de dos partes: una DLL que actúa como una capa de compatibilidad y provee una parte sustancial

de las funcionalidades de la API de POSIX, y una gran colección de herramientas software y aplicaciones que le otorgan un aspecto y funcionalidad similar a un sistema Unix. Se trata de software libre, lanzado bajo la licencia GNU GPL versión 3. Está mantenido actualmente por los empleados de Red Hat, NetApp y otros voluntarios.

Cygwin se ha utilizado cuando hemos necesitado compilar con gcc/g++ las librerías de sistema.

- **Sublime Text** es un editor de texto y código multiplataforma, con una API para Python. Se trata de software propietario, y su funcionalidad puede ser ampliada mediante plugins. Muchos de estos plugins tienen licencias de software libre y están mantenidas por la comunidad. No utiliza cuadros de diálogo para la configuración, sino que ha de ser configurado a través de ficheros de texto.

  Se ha usado para editar todo lo necesario para las librerías de sistema en lenguaje C.

- **StarUML** es una herramienta UML (Unified Modelling Language) de código abierto, con una versión modificada de GNU GPL como licencia. El objetivo de esta aplicación era sustituir aplicaciones comerciales como Rational Rose y Borland Together. StarUML soporta la mayoría de los tipos de diagramas especificados en UML 2.0. No soporta diagramas de objetos, paquetes, tiempo e interacción, aunque los dos primeros pueden ser modelados adecuadamente a través del editor de diagramas de clases.

  Se ha usado para generar los diagramas UML que se encuentran en el documento.

- **Texmaker** es un editor LaTeXmultiplataforma de código abierto lanzado bajo una licencia GPL. Se trata de una aplicación hecha con Qt. Texmaker incluye soporte Unicode, corrección ortográfica, auto-completado, plegado de código y un visor incorporado de PDF con soporte de synctex y modo de visualización continua.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

La ejecución de una simulación compleja en la que intervienen muchas FMUs es una necesidad que puede ser inviable de realizar en un sólo ordenador por la cantidad de recursos que consume. Con el producto software desarrollado en este proyecto, se permite hacer frente a simulaciones complejas repartiendo el gasto computacional en diversas máquinas.

Así, la contribución más importante de este proyecto ha sido un producto software que da soporte a la ejecución de FMUs de forma distribuida.

A este producto software se le ha llamado **Remotifier**. El Remotifier define una estructura cliente-servidor mediante la cual se crean FMUs espejos (Mirrors) encargadas de la comunicación con FMUs que se están ejecutando en nodos remotos.

Aún no siendo uno de los objetivos iniciales, otra contribución importante ha sido el poder ejecutar una FMU en un sistema operativo distinto para el que ha sido generada. Cuando se construye una FMU, el desarrollador puede elegir para qué sistema operativo generarla. Cuanto más sistemas operativos se soporte, mayor tamaño en bytes tiene la FMU. Por esta razón, podemos encontrar FMUs ejecutables únicamente para un sistema operativo. Si quisiéramos ejecutarla en otro sistema operativo no sería posible.

Con el Remotifier hemos dado con una posible solución este problema permitiendo la conexión a otro ordenador en el que la FMU puediera ejecutarse. Además, cabe destacar la ligereza de las FMUs espejo permitiendo que sean compatibles con todos los sistemas operativos.

A nivel académico, podemos concluir que este proyecto ha cubierto las expectativas de un Proyecto Fin de Carrera. He ganado experiencia en diferentes ámbitos de la programación y he adquirido conocimientos y destrezas en el manejo de nuevas herramientas. Los conocimientos que he conseguido se han generado dentro de filosofía de trabajo que se respira en el equipo, todo ello basado en un trabajo constante, reflexivo y con una crítica constructiva que nos permite debatir, confrontar nuestras ideas y mejorar el producto. Esto aporta una satisfacción personal de mejora ya que nos permite estar preparados para esta sociedad del conocimiento.

Además, hay que destacar que las herramientas y métodos aprendidos no solamente se circunscriben al desarrollo del Remotifier sino también al resto de componentes que integran JavaFMI. Por ejemplo, se ha aprendido a usar JNI (Java Native Interface) un framework que permite que aplicaciones hechas en Java puedan interactuar con programas escritos en lenguaje C/C++. De la misma manera, el hecho de utilizar un sistema de control de versiones modificándose a la vez distintas funcionalidades del desarrollo del proyecto ha sido muy gratificante y formativo.

El incorporarme al desarrollo de un proyecto más grande como JavaFMI me ha dado la posibilidad de trabajar en grupo de investigación con todos los beneficios que ello conlleva. Me ha permitido comprobar la interdisciplinariedad que se respira en un equipo de este tipo, ampliando mi experiencia formativa en este campo. El ser un proyecto internacional con contribuciones de laboratorios de Alemania y Francia me ha aportado experiencia a la hora dar soporte al mismo. Me ha dado la oportunidad de interactuar con el equipo, escuchar las sugerencias y aplicar cambios aprendiendo que la idea inicial de un proyecto siempre es mejorable.

Antes de empezar a desarrollar el proyecto había oído de la filosofía ágil en ingeniería de software pero ni siquiera me había planteado el cómo sería el desarrollo de un proyecto aplicando algunas de sus metodologías. Tras finalizar, puedo decir que el hecho de ir realizando diferentes iteracciones cada una enfocada a un cumplir con un requisito con

sus consecuentes pruebas de funcionalidad es la manera más productiva y cómoda de desarrollar software.

Asimismo, el desarrollo dirigido por pruebas, que en un principio puede resultar difícil de asimilar y demasiado costoso, aporta unas grandes ventajas como un aumento de la robustez del código. En un proyecto en grupo como es JavaFMI, el tener una serie de pruebas que te indican el correcto funcionamiento de una clase te permite hacer cambios sobre ella y testear de una forma cómoda si dichos cambios al alterado el comportamiento de la misma.

JavaFMI es un proyecto open source que se distribuye bajo licencia LGPL. El hecho de permitirme trabajar en un proyecto de software libre aporta mucho valor al mismo, ya que no circunscribe su uso a una pequeña comunidad. Está a la mano de cualquier persona que lo desee utilizar e incluso personalizar. Muchos proyectos fin de carrera se toman como el último paso para terminar una carrera; tras su presentación, quedan archivados y no vuelven a utilizarse. En este caso, será abierto al público, y quien quiera podrá utilizarlo, haciendo que sea realmente útil tanto dentro como fuera de la universidad. El código fuente se encuentra disponible para ser utilizado en el repositorio Bitbucket del proyecto en la página referentes a sources `https://bitbucket.org/siani/javafmi/src` y las releases de los distintos componentes en el apartado de descargas `https://bitbucket.org/siani/javafmi/downloads`.

6.2. Trabajo Futuro

Todo proyecto fin de carrera tiene un fin pero siempre se pueden realizar mejoras sobre él. Algunas modificaciones que se pueden realizar como trabajo futuro pueden ser:

- Hacerlo compatible con la versión final 2.0 del estándar FMI. Como indicamos anteriormente, el Remotifier esta disponible para la versión 2.0 RC1 del estándar. Desde esta release candidate 1 se ha definido una versión final 2.0 que incluye algunas modificaciones que se podrían incluir.

- Permitir la ejecución de las simulaciones de forma asíncrona. Actualmente las FMUs se ejecutan de forma síncrona, es decir, se esperan a los comandos para realizar acciones dentro de la FMU. La nueva idea consistiría en permitir que la simulación se vaya ejecutando de forma remota, y responda a las peticiones a la vez que se va ejecutando.
- Mejorar los tiempos de respuesta. El proyecto se ha centrado el cubrir el problema de paralelizar la ejecuciones de FMUs pero no se ha centrado en estudiar y mejorar el rendimiento

Apéndice A

Functional Mock-up 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 `fmiTypesPlatform.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 `fmiFunctionTypes.h` contiene la definición como tipos de los prototipos de las funciones del estándar, y en el listing A.2 se encuentran algunos prototipos de las funciones que son comunes a los tipos Co-simulation y Model Exchange. El último fichero llamado `fmiFunctions.h` contiene las funciones que han de ser llamadas desde la propia simulación; para este fichero no he incluido ningún listing porque es demasiado extenso y hace uso de macros que no permiten simplificarlo para que sea legible a simple vista.

Listing A.1: fichero fmiTypesPlatform.h

```
typedef void*         fmiComponent;
typedef void*         fmiComponentEnvironment;
typedef void*         fmiFMUstate;
typedef unsigned int  fmiValueReference;
typedef double        fmiReal   ;
typedef int           fmiInteger;
typedef int           fmiBoolean;
typedef const char*   fmiString ;
typedef char          fmiByte   ;

#define fmiTrue  1
#define fmiFalse 0
```

Listing A.2: parte del fichero fmiFunctionTypes.h

```
typedef enum {
    fmiOK,
    fmiWarning,
    fmiDiscard,
    fmiError,
    fmiFatal,
    fmiPending
} fmiStatus;

typedef enum {
    fmiModelExchange,
    fmiCoSimulation
} fmiType;

typedef enum {
    fmiDoStepStatus,
    fmiPendingStatus,
    fmiLastSuccessfulTime,
    fmiTerminated
} fmiStatusKind;

...
   typedef const char* fmiGetTypesPlatformTYPE();
   typedef const char* fmiGetVersionTYPE();
   typedef fmiStatus   fmiSetDebugLoggingTYPE(fmiComponent,
    fmiBoolean, size_t, const fmiString[]);


   typedef fmiComponent fmiInstantiateTYPE (fmiString, fmiType,
   fmiString, fmiString, const fmiCallbackFunctions*,
    fmiBoolean, fmiBoolean);
   typedef void         fmiFreeInstanceTYPE(fmiComponent);

   typedef fmiStatus fmiSetupExperimentTYPE
      (fmiComponent, fmiBoolean, fmiReal, fmiReal, fmiBoolean, fmiReal);
   typedef fmiStatus fmiEnterInitializationModeTYPE(fmiComponent);
   typedef fmiStatus fmiExitInitializationModeTYPE (fmiComponent);
   typedef fmiStatus fmiTerminateTYPE              (fmiComponent);
   typedef fmiStatus fmiResetTYPE                  (fmiComponent);

   typedef fmiStatus fmiGetRealTYPE   (fmiComponent,
   const fmiValueReference[], size_t, fmiReal   []);
   typedef fmiStatus fmiGetIntegerTYPE(fmiComponent,
   const fmiValueReference[], size_t, fmiInteger[]);
   typedef fmiStatus fmiGetBooleanTYPE(fmiComponent,
   const fmiValueReference[], size_t, fmiBoolean[]);
   typedef fmiStatus fmiGetStringTYPE (fmiComponent,
    const fmiValueReference[], size_t, fmiString []);

   typedef fmiStatus fmiSetRealTYPE   (fmiComponent,
```

```
const fmiValueReference[], size_t, const fmiReal   []);
typedef fmiStatus fmiSetIntegerTYPE(fmiComponent,
const fmiValueReference[], size_t, const fmiInteger[]);
typedef fmiStatus fmiSetBooleanTYPE(fmiComponent,
const fmiValueReference[], size_t, const fmiBoolean[]);
typedef fmiStatus fmiSetStringTYPE (fmiComponent,
const fmiValueReference[], size_t, const fmiString []);
```

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. RC1

```
// 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

Git flow

Cuando se trabaja con sistemas de control de versiones (System Control Version, 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 los 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 página web de su 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 B.1

Cuenta con dos ramas principales una de desarrollo (develop) donde se llevan a cabo las operaciones diarias de desarrollo, y una rama principal conocida como master donde se hospedan las versiones estables de la aplicación. Las otras tres son auxiliares y tienen un nombre 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 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 ralenticen el proceso. Por último en las ramas release se prepara el código si hiciera falta para su integración con la rama principal master.

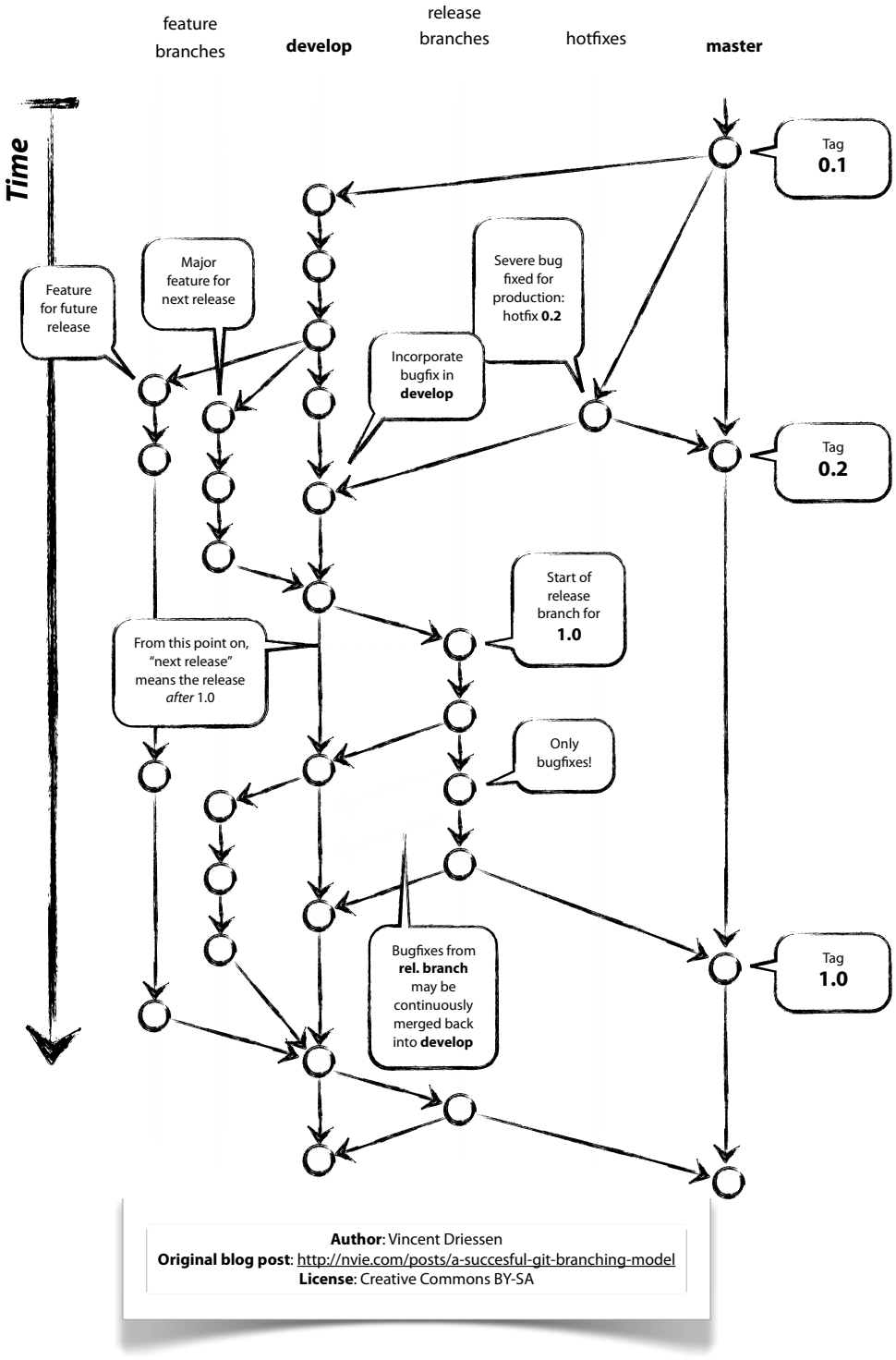

Figura B.1: Modelo de ramas Git Flow

Apéndice C

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 prácticamente 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 está 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 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 características no funcionales de la aplicación, 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 mini-refactor, todas ellas están documentadas en el libro de Joshua Kerievsky Refactoring to Patterns.

Tras esta explicación de porqué 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 especificación del programa debería afectar la especificación 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 mismo.
I	ISP	Interface segregation principle Es mejor tener muchas interfaces, una para cada cliente, que una super interfaz genérica.
D	DIP	Dependency inversion principle Uno solo debería depender de abstracciones, no de implementaciones concretas. La inyección de dependencias (dependency inyection) es un resultado de aplicar este principio, pero a menudo estos términos se confunden.

Apéndice D

Licencia LGPL

La licencia bajo la que se distribuye JavaFMI es una licencia LGPL, incluyéndose en la cabecera de cada código fuente el copyright. La elección del tipo de licencia en un producto es algo esencial ya que esto puede determinar los posibles usos del mismo. La fundación de software libre, mayoritariamente conocida como **Free Software Foundation (FSF)** pone a disposición del usuario distintos tipos de licencias **General Public License (GPL)**. Entre ellas, se ofrece la **Lesser GPL** la cual es más permisiva que sus hermanas, ya que permite el uso de las herramientas distribuidas en aplicaciones no desarrolladas bajo licencia GPL, algo que la mayoría de las licencias GPL no permiten. Por esta razón se ha elegido la licencia LGPL; para proteger el posible trabajo resultante de las personas que hagan uso de JavaFMI. A continuación se anexan estos ficheros de copyright:

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

[1] **Functional Mock-up Interface Oficial**
`https://www.fmi-standard.org`.

[2] **Repositorio del proyecto JavaFMI**
`https://bitbucket.org/siani/javafmi/wiki/Home`.

[3] **Página oficial librería JFMI**
`http://ptolemy.eecs.berkeley.edu/java/jfmi/`.

[4] **Git flow**
`nvie.com/posts/a-successful-git-branching-model/`

[5] **JUnit**
`junit.org/`

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

[7] ***Object Orientated Programming in ANSI-C***.
Axel Schreiner

[8] Leslie Lamport *LaTex : A document Preparation System*. Addison-Wesley, 1986.

[9] Christian Rolland *LaTex guide pratique*. Addison-Wesley, 1993.