Universidad Politécnica de Cartagena

GRADO EN INGENIERÍA TELEMÁTICA
PROYECTO FIN DE GRADO

Diseño e implementación de algoritmos, procedimientos de gestión de software y mejoras de visualización en Net2Plan

Autor: Jorge San Emeterio Villalaín
Director: Pablo Pavón Mariño
Fecha: 2 de junio de 2017

Resumen:

Net2Plan es una herramienta de planificación de redes desarrollada en la Universidad Politécnica de Cartagena originariamente pensada como utensilio de enseñanza para el alumnado. Pronto, la aplicación comenzó a mostrar su potencial fuera del entorno académico, consiguiendo con ello atravesar esta barrera e introducirse dentro del mundo de la industria. Este proyecto fin de grado describe el proceso de modernización que el programa experimentó durante el salto de un entorno a otro a través de la introducción de dos herramientas de gestión y desarrollo de proyectos, como son *Git* y *Maven*. Al mismo tiempo, se detalla la inclusión de mejoras de visualización que tienen como objetivo dotar de más funcionalidad práctica a la aplicación.

Lista de palabras clave:

Gestión de Software, Git, Maven, Ciclo de vida, POM, Repositorio, Control de versiones, *Net2Plan*, *OpenStreetMaps*.

Índice general

1. Introducción **8**

1.1. Net2Plan . . . 9

1.2. Objetivos . . . 10

I Gestión de Software **11**

2. Introducción a la gestión de Software **12**

2.1. Motivación . . . 13

2.1.1. Los inicios . . . 13

2.1.2. El concepto de gestión de software . . . 14

2.1.3. Aplicación de la gestión de software en *Net2Plan* . . . 15

3. *Git* **18**

3.1. ¿Qué es *Git*? . . . 19

3.1.1. *Git* contra *Subversion* . . . 20

3.2. *Git* en *Net2Plan* . . . 23

3.2.1. Situación previa de *Net2Plan* . . . 23

3.2.2. Implementando *Git* . . . 23

4. *Maven* **28**

4.1. ¿Qué es *Maven*? . . . 29

4.1.1. El *POM* . . . 30

4.1.2. El ciclo de vida de *Maven* . . . 34

4.1.3. Los repositorios de *Maven* . . . 35

4.2. *Maven* en *Net2Plan* 37
4.2.1. Necesidades 37
4.2.2. Implementando *Net2Plan* 38

II Mejoras de visualización en Net2Plan **45**

5. Interfaz con *OpenStreetMaps* **46**
5.1. Introducción 47
5.2. Objetivos 47
5.3. Glosario de términos 48
5.4. Implementación 48
5.4.1. Añadiendo el fondo 48
5.4.2. Sincronizando *JUNG* y *OSM* 50
5.4.3. Interacción con el usuario 51

6. Conclusiones **53**
6.1. Situación final del proyecto 54
6.2. Pasos futuros 54

Estructura del proyecto

A continuación, se resumen las secciones en las cuales este proyecto ha sido dividido:

Introducción
Introducción de la aplicación *Net2Plan*. Detalla sus características y funcionalidades para situar el contexto del proyecto. Junto a ello, se definen los objetivos que presenta el mismo.

Parte I

Capítulo 1.
Introducción a la gestión del *software*. Se detallan conceptos básicos sobre la misma junto con la motivación que hay detrás de su adaptación. Además, se refleja el estado inicial de *Net2Plan* dentro de este marco y se nombran algunas herramientas del entorno.

Capítulo 2.
Presentación del gestor de versiones *Git*. Detalles elementales sobre su funcionamiento y concepción. Se detalla el proceso mediante el cual el gestor fue implementado en *Net2Plan*.

Capítulo 3.
Muestra del gestor de dependencias y construcción *Maven*. Entrada en detalle sobre su funcionamiento e definición. Se explica como este gestor entró dentro del proyecto *Net2Plan* y cómo se produjo su proceso de implantación.

Parte II

Capítulo 4.
Descripción del procedimiento que se siguió para construir una interfaz con respecto al proveedor de cartografía *OpenStreetMaps*. Se señala la secuencia de pasos que se dio para conseguir un panel de visualización de mapas dinámico dentro de *Net2Plan*.

Conclusión
Se da conclusión estableciendo la situación final del programa junto con los pasos que se dieron una vez que este proyecto fue finalizado.

Capítulo 1

Introducción

1.1. Net2Plan

Net2Plan es una aplicación de código libre escrita en el lenguaje de programación *Java* que tiene como objetivo actuar como herramienta de planificación, optimización y evaluación de redes de comunicaciones. *Net2Plan* se desarrolla actualmente en la Universidad Politécnica de Cartagena y es utilizado como recurso principal de enseñanza de múltiples asignaturas impartidas en esta universidad. A su vez, en los últimos tiempos la aplicación está comenzando a dar sus primeros pasos en el ámbitos de la industria e investigación.

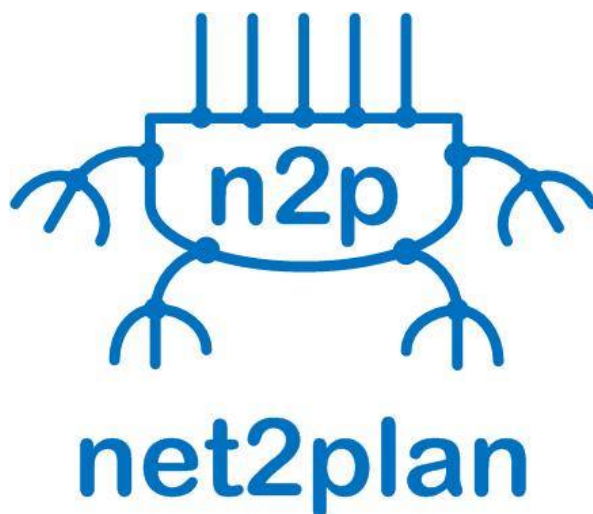


Figura 1.1: Logo de la aplicación

El funcionamiento de *Net2Plan* está basado en el concepto del *NetPlan*, una representación abstracta de una red real que está compuesta por una biblioteca de componentes tales como nodos, enlaces, demandas, etc. Esta representación abstracta permite definir redes de forma genérica, es decir, ignorando la tecnología que conforma la red real.

Una vez queda definida una red, se introduce la parte más interesante de la aplicación: la ejecución de algoritmos e informes de red. Ambas partes son de libre creación, de tal manera que un usuario puede construir su propio algoritmo de optimización y ejecutarlo sobre la aplicación para ver su resultado.

Junto a lo anterior, la aplicación también proporciona una serie de herramientas tales como simuladores de eventos y generación de matrices de tráfico.

Finalmente, *Net2Plan* está planteado desde el principio para actuar como una plataforma modular. De esta forma, si el usuario echa en falta alguna funcionalidad que no viene de serie, este es capaz de definir su propio entorno dentro del programa para que este cumpla sus requisitos.

1.2. Objetivos

Los objetivos principales del proyecto son los que siguen a continuación:

1. Estudiar e implementar la herramienta de gestión de versiones *Git* junto a la filosofía de trabajo *Git Flow*.
2. Estudiar e introducir la aplicación de gestión de proyectos *Maven* y generar una nueva estructura de programa acorde con ella.
3. Implementar un servicio de cartografía para *Net2Plan*.

Parte I

Gestión de Software

Capítulo 2

Introducción a la gestión de Software

2.1. Motivación

2.1.1. Los inicios

En su inicio, *Net2Plan* era un proyecto desarrollado por un grupo cuyo tamaño no excedía los dos miembros. Este hecho conllevaba que el desarrollo del software fuera controlado mayoritariamente de manera manual, puesto que los cambios y avances que se producían sobre el código eran lo suficientemente disjuntos y claros como para poder ser tratados mediante mano humana. Este trato afectaba pues a las tres grandes partes de la gestión del software, las cuales son:

- Unión, mecanismo por el cual el trabajo de múltiples personas es unido en una única versión de código.
- Generación, método por el cual se transforma el código en el proyecto final proporcionable al usuario.
- Despliegue, forma con la que lo generado es proporcionado de forma pública.

En primer lugar y con respecto a la unión, cuando una única persona trabajaba sobre el proyecto esta parte no suponía un problema, puesto que el código generado por dicha persona serviría como la versión final del mismo. Por otro lado, cuando más de una persona trabajaba paralelamente sobre el mismo código, la unión de este conllevaba la presencia de dichas personas para poder juntar de forma controlada lo realizado. En segundo lugar, la generación se realizaba mediante un conjunto de tareas descritas mediante *Apache Ant*[1], cuya configuración requería de intervención previa puesto que era dependiente del ordenador en el que se ejecutara. Por último, la versión preparada de *Net2Plan* era desplegada en la página web del proyecto[2], de tal manera que cualquier usuario pudiera descargar los últimos cambios a través de la misma.

Lo anteriormente descrito resultó ser viable durante la época en la que *Net2Plan* era aún un programa desarrollado por una o pocas personas. Pero en cuanto el número de estas personas aumenta a los tres o cuatro miembros, este mecanismo se muestra completamente anticuado e insostenible.

[1] Librería de *Java* encargada de ejecutar tareas descritas mediante *XML* con el fin de construir aplicaciones de este lenguaje.

[2] http://www.net2plan.com

2.1.2. El concepto de gestión de software

La idea fundamental del desarrollo de código es el concepto de que eventualmente todas las ramas que se han ido produciendo paralelamente en este tienen que converger en un único conjunto verificable que pueda ser entregado finalmente al usuario o cliente. Para esto, es necesario que todo el equipo que trabaja sobre el programa comparta un mismo medio en el que cada uno pueda trabajar por separado pero al mismo tiempo siguiendo un orden. Lo anterior conlleva a un estudio de los tres pilares de la gestión de software establecidos anteriormente.

Para comenzar, el ser humano no es bueno a la hora de mantener un histórico de los cambios que ha realizado a lo largo del tiempo, ya que que una misma tarea puede ser abierta y cerrada múltiples veces si la situación lo necesita; y si la persona no mantiene constancia de dichos eventos, se puede perder completamente el hilo de lo que se ha realizado. Esto afecta posteriormente a la unión del código de esta persona con el de sus compañeros, pues si no se sabe exactamente en que difieren cada uno de ellos, difícilmente se va a conseguir una fusión correcta y libre de errores.

A lo anterior se le ha de sumar además el hecho de la heterogeneidad de los entornos de desarrollo en los cuales trabajan los miembros de un equipo. Esto afecta especialmente a aplicaciones multi-plataforma como *Net2Plan*, en donde el programa se debe mantener constante a lo largo de múltiples sistemas operativos, y en menor medida a aquellas que son mono-plataforma. La situación obliga a que cada persona trabaje en un ámbito que en su mayor parte es similar al de los demás compañeros, pero que presenta las suficientes diferencias como para dar lugar a la aparición de una serie de fallos no controlables por el desarrollador. Lo anterior se puede extender no solo a la acción de usar distintos sistemas operativos, si no también al hecho de que cada trabajador pueda utilizar programas de desarrollo distintos. Considerando lo anterior, es necesario mantener un control sobre este problema, pues si no es posible enfrentarse a la situación en la que el trabajo de cada miembro del grupo sea incompatible con el de los demás.

Con todo esto en mente, una vez que todos los desarrolladores de una aplicación consigan trabajar en armonía con sus compañeros, es necesario hablar sobre el último problema del trabajo en equipo. A partir de un código común comprobado y verificado, es necesario que bajo cualquier tipo de entorno de desarrollo se produzca siempre el mismo proyecto final. Esto es, el mecanismo de compilado y empaquetado debe estar preparado de tal manera que bajo cualquier circunstancia sea capaz de producir siempre un mismo resultado. En el caso de que existan diferencias insalvables entre los entornos de desarrollo, tal como ocurre en los lenguajes de programación nativos como *C*, el mecanismo de empaquetado debe proporcionar las herramientas para que el proyecto se construya de manera independiente en cada uno de estos entornos. Proporcionando el mismo resultado a cada miembro que trabaja en dicho entorno de desarrollo.

En cuanto al despliegue de la aplicación, este es el paso de la gestión de software que menos depende del trabajo en equipo, pues por lo habitual es una persona a la que dada una

aplicación final es la encargada de subir dichos archivos al servidor que corresponda. En muchas ocasiones, esta tarea no es ni siquiera llevaba a cabo por una persona pues a día de hoy existen programas que se encargan de controlar y manejar este tipo de tareas.

2.1.3. Aplicación de la gestión de software en *Net2Plan*

Llegado un momento en su trayectoria, el número de personas que empezó a trabajar sobre *Net2Plan* aumentó, y por ello, la idea de implementar mecanismos con los cuales el desarrollo en este fuera más cómodo empezó a tener cierto peso. Por todo lo anterior, este proyecto se centro entonces en la aplicación de dichos medios para que la gestión de este software fuera más controlable y robusta.

La primera cuestión surgió de forma natural, era necesario una manera sencilla con la cual se pudiera mantener un control sobre el trabajo de cada uno de los nuevos miembros del equipo. Con este problema se ha encontrado absolutamente toda persona que haya desarrollado alguna vez alguna aplicación en equipo, y es por ello que no es de extrañar que ya existan alternativas que se encargan de hacer de esto algo sencillo. Estas alternativas se refieren principalmente a **Git** y **Subversion**.

(a) Git

(b) Subversion (SVN)

Figura 2.1: Programas de control de versiones

Ambos programas se hacen llamar software de control de versiones y, pese a que muestran ciertas diferencias, tienen entre otros como objetivo facilitar el trabajo en paralelo y la unión de todo el trabajo en uno solo. Esto es justamente una de la administraciones que *Net2Plan* necesitaba y por ello, se decidió la idea de comenzar a utilizar uno de estos controladores de versión. A día de hoy, incluso si una única persona es la encargada de desarrollar un proyecto entero, los software de control de versiones ofrecen tantas ventajas a cambio de una configuración tan poco costosa que es difícil negar su uso.

La elección de cual controlador de versiones sería usado no fue complicada de alcanzar. Pese a que *Subversion* es el software más veterano, *Git* es a día de hoy la solución más aceptada y soportada. Y a raíz de ello, junto con otra serie de razones de las cuales se hablará próximamente, fue elegido para su uso en *Net2Plan* debido a su facilidad de integración y su buena documentación. Esto no quiere decir que *Subversion* sea una opción incorrecta, puesto que aún siguen habiendo grupos que defienden sus ventajas sobre *Git*. Considerando lo anterior, este proyecto se centrará principalmente en como *Git* fue implementado en *Net2Plan*, además de la nueva filosofía que trajo su implantación.

Seguidamente, la segunda cuestión que surge cuando se comienza a trabajar en equipo tiene que ver con el entorno de programación que se proporciona, es decir, tiene que tratar con la cuestión de que añadir o reinstalar un puesto de trabajo debe ser una tarea atómica y simple. Todo esto se refiere al hecho de que la tarea de cargar y arrancar el código del programa desde un [3]*IDE* debe ser automática y completamente transparente a la persona que lo arranque. Para gestionar esta cuestión existen las herramientas de software conocidas como gestores de construcción y dependencias de proyectos, los cuales, tal como describe su nombre se encargan de descargar y manejar las dependencias del proyecto así como de construir el mismo. Ejemplos de este tipo de software son **Gradle**, **Ivy** y el que más nos vamos a centrar, **Maven**.

Figura 2.2: Gestor de dependencias y construcción *Maven*

[3]Integrated Development Environment: Es un software que proporciona las herramientas para que un desarrollador pueda crear sus propios proyectos. Para *Java*, un ejemplo clásico es el *IDE Eclipse*.

Los tres gestores anteriormente mencionados son usados en múltiples entornos de programación, aunque en este proyecto nos centraremos en *Java*, ya que *Net2Plan* se desarrolla en este. Los tres gestores se pueden usar de manera nativa sobre *Java* sin necesidad de utilizar extensiones.

La toma de la decisión sobre cual de los gestores debía ser implementado en *Net2Plan* fue, otra vez, rápida de decidir. En el entorno *Java*, es indiscutible que *Maven* es actualmente el gestor más utilizado y extendido, pese a que *Gradle* está haciendo últimamente un esfuerzo por intentar quitarle el trono. La gran mayoría de los *IDEs* de a día hoy traen soporte para esta plataforma o bien de forma nativa o bien a través de extensiones, y en el caso de que no lo hagan *Maven* es capaz de utilizarse de forma propia independientemente de las demás herramientas de desarrollo que se estén usando. Es por todo esto, junto con la que razón de que relativamente fácil de instalar, que *Maven* se vuelve la elección más sensata que instalar a *Net2Plan*.

Para terminar, el despliegue de la aplicación se revuelve mediante una combinación de las dos integraciones anteriormente descritas. Por un lado, la construcción del proyecto esta manejada en su totalidad por el gestor, por lo que lo único de lo que debe encargarse el desarrollador es de subir los archivos resultantes al servidor correspondiente. Pese a que *Maven* es capaz de realizar esta tarea por su cuenta, para *Net2Plan* se decidió que se realizaría a mano, cogiendo los ficheros y subiéndolos al mismo servidor en el que se encuentra el [4]repositorio de *Git*: *GitHub*. *GitHub* es una página web que entre otros ofrece servicios de [5]*hosting* de repositorios *Git* y de almacenamiento de datos y transmisión de mensajes relacionados con el proyecto correspondiente. De esta forma, se unifican todos los elementos de la gestión de software, girando todos ellos en torno al proyecto que esta publicado en la nube bajo los servidores de *GitHub*.

Figura 2.3: Almacenamiento de repositorios en el red: *GitHub*

[4]Área de trabajo de un proyecto bajo *Git*.
[5]Almacenamiento de datos en la *web*.

Capítulo 3

Git

3.1. ¿Qué es *Git*?

Git es un software de control de versiones creado originariamente en el año 2005 por *Linus Torvalds* como herramienta para el desarrollo del sistema operativo *Linux*. Actualmente, *Git* es distribuido como una aplicación gratuita protegida bajo la licencia [1]GNU que funciona bajo los sistemas operativos más comunes tales como *Windows* o los basados en *Unix*. *Git* es proporcionado principalmente como una herramienta que funciona bajo la consola de comandos del sistema operativo, aunque a día de hoy hay múltiples implementaciones que proporcionan una interfaz gráfica con la que hacer el programa más intuitivo y visual.

Git presenta en su filosofía dos características principales que lo separan de sus competidores.

1. **La gestión del código se realiza a través de un formato distribuido:**
   En lugar de tener un servidor central en el que los usuarios se conectan para trabajar y modificar el proyecto, cada usuario posee una copia propia de este sobre la que pueden trabajar, y es unicamente en el momento que quieran centralizar su trabajo cuando deban contactar con el servidor que presenta el código de referencia.

2. **Plantea un flujo de trabajo no lineal:**
   Git establece una estructura de desarrollo basada en ramas. Dicho de otra forma, el código del proyecto no presenta una realidad única si no que existen del mismo múltiples versiones paralelas (ramas) cada una de ellas con su propia historia. Más adelante, todas estas ramas son unificadas en una única que contendrá toda la funcionalidad de las ramas que la forman.

Con todo esto, el flujo de trabajo en *Git* esta formado de dos componentes: el entorno local y el entorno remoto. En cada uno de estos entornos se encuentra lo que se conoce como el repositorio, el cual actúa como el área de trabajo y almacenamiento del proyecto. El repositorio remoto es el servidor anteriormente mencionado que actúa como el contenedor del código central que todos los miembros del desarrollo utilizan como referencia, mientras que el repositorio local es la copia que cada uno de estos miembros almacena en su propio ordenador y que es independiente de la de los demás. Esto es, si un miembro trabaja sobre su repositorio local pero nunca sincroniza sus cambios con el repositorio remoto, a ojos del restos de los desarrolladores esta persona nunca realizó trabajo alguno.

[1]Licencia para la protección del software gratuito.

Como se puede deducir por lo visto hasta ahora, un entorno de trabajo basado en *Git* precisa de dos elementos principales: un ordenador con el que trabajar en local y un servidor *Git* con el que trabajar en equipo y tener copias de seguridad. A día de hoy, múltiples páginas web tales como *GitHub* o *Bitbucket* ofrecen servicios para el almacenamiento de repositorios *Git*. Estas páginas ofrecen los servicios básicos de un repositorio remoto de forma gratuita y presentan la forma más sencilla y rápida de preparar un entorno *Git*. Sin embargo, los desarrolladores del proyecto *Git* ofrecen las herramientas necesarias para que una persona pueda construir su propio servidor de *Git*, obteniendo así una versión de este más personal y privada.

Pese a que conlleva perder una gran cantidad de ventajas, no es estrictamente necesario tener un servidor de *Git* cuando el trabajo es realizado por una única persona, puesto que al no tener compañeros que precisen de su trabajo el entorno local ofrece todas las funciones que esta pueda necesitar. Pese a ello, es altamente recomendable tener siempre la opción remota disponible debido principalmente a la cualidad de tener una copia del código disponible en la nube a través de las propias herramientas que este controlador de versiones ofrece.

Para terminar, *Git* fue originariamente pensado para ser una herramienta que trabaje principalmente con textos, pero ello no conlleva que no sea capaz de trabajar con otros tipo de archivos. La forma de trabajar de *Git* es genérica, lo cual le permite tratar de la misma forma a un archivo de texto que a uno de sonido. Lo anterior ayuda al controlador de versiones a ser extendido más allá del ámbito del software, puesto que puede ser también utilizado por ejemplo para la edición de vídeo o de imágenes. Sin embargo, el repositorio remoto tiene un límite y por ello no es recomendable almacenar archivos que ocupen un gran tamaño.

3.1.1. *Git* contra *Subversion*

La principal competencia del controlador *Git* es *Subversion (SVN)*, un controlador de versiones más veterano que el que aquí se trata que hizo su aparición en el año 2000 de la mano de la fundación *Apache* y que en estos momentos tiene un uso tan o casi tan extendido como *Git*.

A continuación se presenta una comparativa entre los dos gestores de versiones con las características y desventajas de cada uno de ellos:

- Git:
  - Características:
    1. Distribución del código distribuida.
    2. Posibilidad de trabajar sin conexión a Internet
    3. Trabajo dividido en ramas.
    4. Seguimiento de los cambios basado en contenidos.
    5. Derecho de acceso a todas las partes del proyecto.
    6. Organización del proyecto a través de archivos.
    7. Rápido.
    8. Posibilidad de elegir específicamente que cambios subir.
  - Desventajas:
    1. Mayor curva de aprendizaje debido a un mayor número de comandos y conceptos.
    2. Código descentralizado, imposibilidad de saber que esta haciendo cada desarrollador.
    3. Lento a la hora de trabajar con archivos grandes o que no son de texto.
- Subversion:
  - Características:
    1. Distribución de código centralizada.
    2. Trabajo dividido en ramas.
    3. Seguimiento de cambios basado en ficheros.
    4. Posibilidad de limitar el acceso a ciertas partes del código.
    5. Organización del proyecto a través de directorios.
    6. Curva de aprendizaje baja.
    7. Sencillo e intuitivo.
  - Desventajas:
    1. Lento.
    2. Necesidad constante de conexión a Internet para trabajar.

Viendo la comparativa, se puede apreciar que los dos gestores de versiones tienen ciertos puntos en común, pero difieren lo suficiente en otros aspectos como para que cada uno de ellos tenga su identidad propia. Estos programas presentan al mismo tiempo una serie de ventajas y otra de desventajas, y es por ello, por lo que no es fácil situar a uno por encima del otro. A la hora de la verdad la decisión sobre cual elegir depende únicamente de las necesidades del usuario final.

Por ejemplo, si se presenta un grupo de trabajo en el que existe una jerarquía definida tal como puede ser una oficina, seguramente sea preferible utilizar *SVN* debido a la posibilidad de controlar ciertos accesos. Si por otro lado tenemos un grupo de desarrolladores físicamente distribuidos y en el que todos los miembros tengan los mismos derechos, entonces *Git* parece la mejor opción.

En el caso de *Net2Plan* se eligió a *Git* sobre su competidor por la misma razón recién mencionada. Simplemente, para el grupo de desarrollo que *Net2Plan* presenta las condiciones que este controlador de versiones presenta son más adecuadas que las de su competidor. Con motivo de profundizar en este dato, a continuación se presenta una ristra de argumentos de por qué *Git* fue elegido sobre *Subversion* para *Net2Plan*:

- Mientras que existen servicios web tales como *GitHub* para *SVN*, la filosofía de este controlador conlleva que el servidor donde se encuentra el repositorio remoto sea de propiedad privada. Esto conlleva la necesidad de compra, instalar y mantener un servidor, lo cual para un grupo de pequeño tamaño puede ser una inconveniencia grande.
- El desarrollo que se realiza en el ambiente de *Net2Plan* requiere en múltiples ocasiones de viajes y trabajo a distancia, por lo que la posibilidad de poder trabajar durante estas estancias sin la necesidad de conexión a Internet es otro punto importante a tener en cuenta.
- La jerarquía que el grupo muestra no es lo suficientemente rígida como para precisar el límite de acceso a ciertos contenidos del proyecto.
- *Git* es actualmente el programa de moda, y por ello existen más herramientas e información a la hora de empezar a trabajar con él.

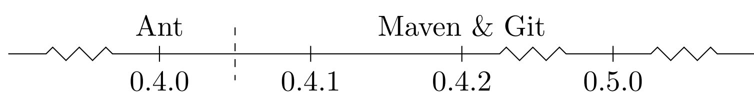
Tomada una vez la decisión y teniendo en mente todo lo anteriormente descrito, en la siguiente sección se procede a explicar como se implementó el entorno *Git* para *Net2Plan* y cual fue su impactó en la filosofía de trabajo del grupo.

3.2. *Git* en *Net2Plan*

3.2.1. Situación previa de *Net2Plan*

Antes de comenzar a mostrar como se produjo la implantación del controlador de versiones sobre este proyecto, conviene establecer previamente cual era la situación en la que se hallaba el programa.

Net2Plan se encontraba en su versión número 0.4.1, la primera versión resultante de un plan de mejora del entorno de desarrollo basado en dos pasos: primero, incluir un gestor de versiones y segundo, incluir el proyecto en el ámbito de *Maven*. Esta versión supuso un punto de inflexión en el proyecto, puesto que fue la que marcó la diferencia en el tamaño del grupo de trabajo que paso de ser apenas de uno o dos miembros a ser de tres o cuatro. Este cambio parece sutil, pero en la realidad produce una gran diferencia en la metodología de trabajo.



La nueva versión se encargaba principalmente, junto a una serie de cambios menores, de introducir *Net2Plan* 0.4.0 a las nuevas dos tecnologías ya presentadas. Pese a que dichas tecnologías no dependen entre ellas de ninguna forma, *Net2Plan* 0.4.1 no podía realizar su despliegue al público hasta que el susodicho plan fuera terminado por completo. De esta forma se realizaba la renovación entera de un solo golpe, ahorrando confusión y problemas surgidos por la mezcla entre las distintas herramientas que conforman el proyecto.

Con todo esto, se abandonaba la metodología de trabajo antigua, la cual estaba basada en el control de cambios manual y la generación del proyecto basada en *scripts Ant*, y se daba comienzo a la integración de las nuevas herramientas empezando por el controlador de versiones *Git*.

3.2.2. Implementando *Git*

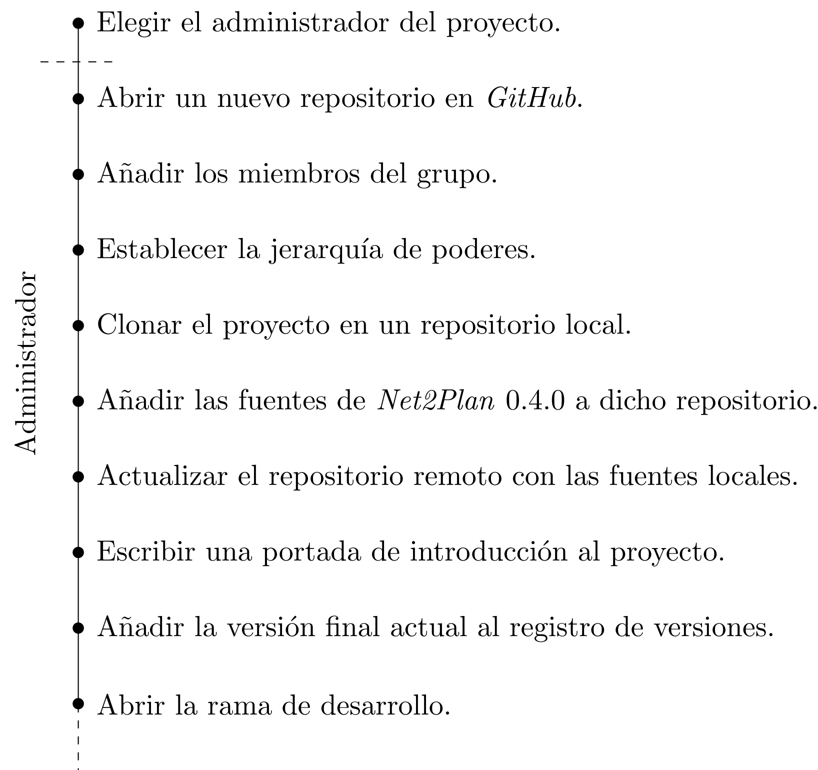
Git es capaz de trabajar bajo una gran cantidad de condiciones, por ello, lo primero que hay que establecer de forma clara son las necesidades que presenta el proyecto sobre el gestor de versiones. En el caso de *Net2Plan*, al haber más de una persona trabajando sobre el proyecto es necesario contar con un punto de referencia global, lo cual se proporciona a través de un repositorio remoto. Por otro lado, *Net2Plan* no empieza desde cero si no que han habido múltiples versiones públicas de este antes de la aparición de *Git*, lo cual conlleva a la necesidad de que se comience a trabajar sobre un código base. Finalmente, es conveniente tener un mecanismo por el cual se pueda establecer una cierta jerarquía de poderes para los miembros del grupo.

Afortunadamente, los servicios que ofrece *GitHub* cumplen todas las necesidades anteriores, ofreciendo un repositorio remoto en la nube de libre acceso pero que proporciona herramientas para el control de poderes. Además, todos estos servicios son otorgados de manera gratuita para aquellos proyectos que sean [2] *Open Source* tales como *Net2Plan*, lo cual termina por dar la última razón para el uso de este servicio.

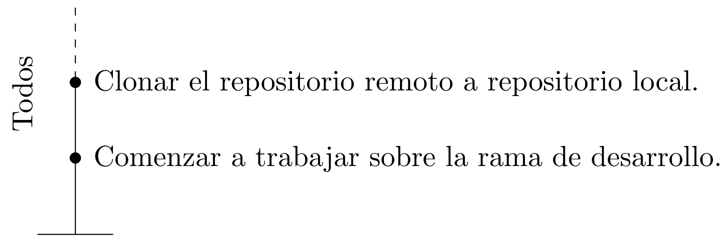
A modo de resumen, se presentan las necesidades de *Net2Plan* sobre *Git* a través del siguiente esquemático:

- GitHub:
  1. Repositorio remoto: Punto de referencia global para el código del proyecto.
  2. Código base basado en *Net2Plan* 0.4.0.
  3. Sistema de permisos de acceso y modificación: Jerarquía de poderes.
  4. Opcionales:
     - Página principal: Enlazar a contenidos relacionados con el proyecto.
     - Almacenamiento de versiones finales: Punto de descarga y punto de código en el mismo lugar.
     - Línea temporal: Recordar cambios pasados.

Decidido todo lo anterior, lo único que queda por hacer es comenzar con el alojamiento de *Net2Plan* sobre *GitHub*. Para realizar dicha tarea, se siguieron los siguientes pasos:



[2] Proyectos de código libre



A continuación, se procede a tratar en mayor profundidad los puntos anteriores que guardan una mayor importancia:

- **Establecer la jerarquía de poderes**

  GitHub ofrece tres tipos de poderes: el administrador, de escritura y de lectura. El administrador es la persona encargada de controlar la configuración del repositorio y el uso que se le da a este. Las personas con permisos de escritura son aquellas que tienen la capacidad de escribir e introducir nuevo código al proyecto, aunque si el administrador lo desea, es necesario previo permiso de este para que la persona pueda hacer sus contenidos finales. En último lugar, los derechos de lectura permiten únicamente observar el código pero sin modificarlo.

- **Añadir las fuentes de *Net2Plan* 0.4.0 a dicho repositorio**

  Todo el código que se sube al repositorio remoto proviene de un repositorio local, por lo que es necesario clonar el proyecto localmente para poder así introducir las fuentes desde las que se quiera comenzar el proyecto.

- **Abrir la rama de desarrollo**

  Hay dos ramas principales en todo proyecto de *Git*: *master* y *develop*. Por defecto, solo la rama *master* es creada junto al repositorio, por lo que es necesario que el administrador cree la otra rama paralela. Por su lado, la rama *master* contiene las fuentes finales que se proporcionan como producto al exterior, mientra que la rama *develop* contiene las fuentes en desarrollo entre versiones. Pese a que no es estrictamente necesario, es recomendable que únicamente el administrador sea capaz de modificar estas ramas con nuevo contenido.

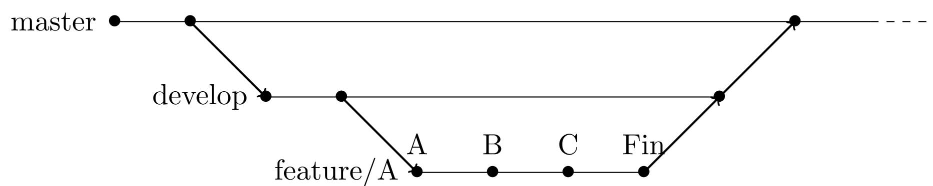
Una vez terminada la implementación de *Git*, el siguiente consiste en poner a todos los miembros del grupo de desarrollo de acuerdo para llegar a una filosofía de trabajo que permita coexistir todos los avances que se realicen en paralelo de una forma pacífica.

En el caso de *Net2Plan*, se optó por la adopción de una filosofía de trabajo muy común en los entornos de desarrollos pequeños conocida como ***Git Flow***, la cual será explicada en el siguiente apartado.

3.2.2.1. *Git Flow*

No es posible comenzar a describir una filosofía de trabajo como *GitFlow* sin saber previamente en que consiste la herramienta sobre la que se apoya. Con este fin, se procede a explicar de forma breve como es el flujo de trabajo habitual en una aplicación bajo *Git*.

Comenzando desde el inicio, en un caso habitual de trabajo en *Git* se plantean tres versiones o ramas paralelas que se relacionan entre si hasta alcanzar un punto único entre ellas. Estas ramas son las anteriormente mencionadas *master* y *develop* junto con una tercera cuyo nombre corresponde con la función que se esté implementando, por ejemplo, "feature/A". De esta manera, *develop* nace a partir de *master* y la "función/A"nace a su vez de *develop*, creando así una relación en árbol.

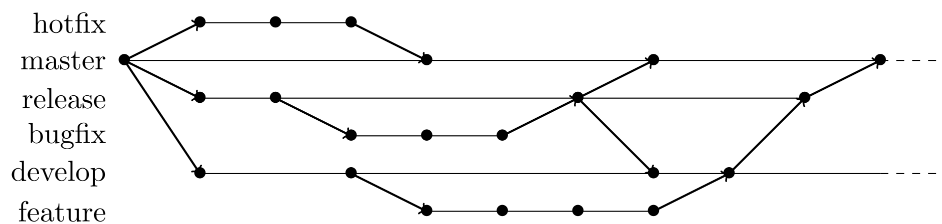


Ya que el objetivo final es el de juntar todas las ramas en una sola, este proceso se lleva a cabo en el sentido contrario al que estas nacieron. Es decir, una vez se haya terminado de trabajar en una nueva función, *feature/A* fusiona su contenido con *develop* y cuando este esté listo a su vez, se unirá a *master*.

Esta es la filosofía de trabajo más simple que se puede aplicar a *Git*. Con todo esto, la intención entonces de *GitFlow* es la de mejorar y expandir esta metodología para ofrecer una forma de trabajo más flexible y capaz. Con ello y en base a los visto anteriormente, *GitFlow* basa su operación en el uso conjunto de los siguientes tipos de ramas:

- **master**: La línea principal del proyecto. En esta rama se almacena únicamente aquel código que sea final y mostrable al usuario.
- **develop**: Rama de referencia para todo aquel código que esté en desarrollo. Nace desde *master*.
- **feature/***: Ramas que contienen una función en desarrollo. Estas ramas son usadas para crear nuevos contenidos sin producir molestias a las demás ramas.
- **release/***: Rama intermedia entre *master* y *develop* utilizada para tener una forma de desarrollar sobre el contenido de *master* sin interferir con *develop*. Cuando *release* tiene un nuevo código que ofrecer, esta es juntada tanto *master* como con *develop*.
- **bugfix/***: Ramas similares a *feature/** que son usadas para solventar problemas en el código.
- **hotfix/***: Ramas que nacen y mueren en *master* utilizadas para la corrección de un error puntual grave sobre la versión pública.

De esta forma, el flujograma anteriormente presentado es extendido de la siguiente manera:



La anterior forma de trabajo es actualmente la mayor aceptada para aquellos proyectos de pequeño o mediano tamaño que utilicen *Git*. Gracias a esto, a día de hoy existen una gran cantidad de aplicaciones que ofrecen una serie de herramientas para facilitar al usuario la implementación y el uso de esta filosofía.

Por supuesto, *GitFlow* es únicamente una filosofía de trabajo, es decir, no es más que un conjunto de recomendaciones a seguir para obtener un beneficio. Por esta razón, es posible tomarse dichas instrucciones con un mayor o menor grado de rigidez, ofreciendo así la posibilidad de tomar ciertas licencias personales sobre el uso de la metodología.

Para terminar y como apunte, *GitFlow* es utilizado en muchas ocasiones junto a una filosofía de trabajo en equipo, que no de trabajo sobre *Git*, conocida como *Scrum*. La anterior idea propone la creación de una serie de cartas o tareas que definen los objetivos del proyecto de una manera escalable. Esto es altamente compatible con *GitFlow* debido a que es posible abstraer estas cartas como ramas *feature* y el conjunto de las mismas, conocido como *Sprint*, como la rama *develop*.

Capítulo 4

Maven

4.1. ¿Qué es *Maven*?

Maven es una herramienta de gestión de dependencias y construcción de proyectos para *Java* nacida en 2002 de la mano de la fundación *Apache*. En la actualidad, *Maven* es distribuido como una herramienta gratuita funcional bajo los principales sistemas operativos que está protegida por la licencia [1]*Apache 2*. El uso de *Maven* se realiza originariamente a través de la utilización de comandos específicos introducidos mediante una consola de texto. Sin embargo, a día de hoy existen una gran cantidad de editores que permiten arrancar dichos comandos de una forma más intuitiva mediante la utilización de interfaces gráficas. De hecho, tal es el uso de *Maven* en el entorno de *Java* que la mayoría de *IDEs* traen de forma implícita la aplicación, eliminando incluso la necesidad de instalar y configurar el programa para su uso.

De una forma introductoria, las principales características que distinguen a *Maven* de sus similares son las siguientes:

1. **Funcionalidad basada en módulos:**

   Maven como tal no ofrece utilidad alguna, si no que presenta una caja en la que introducir funciones que se encargan de realizar trabajos específicos. De esta forma, el programa presenta una gran escalabilidad gracias a que cualquiera es capaz de escribir su propia tarea. Además, este mecanismo se asegura de que el usuario nunca use más de lo que verdaderamente necesite.

2. **Listo para trabajar conectado:**

   Maven es capaz de darse cuenta sobre la marcha de las dependencias y funciones que el usuario requiere, descargando de una forma transparente los elementos requeridos.

3. **Formato descriptivo mediante [2]*XML*:**

   Mientras que otros gestores utilizan *scripts* y lenguajes propios, *Maven* utiliza un lenguaje de descripción estandarizado basado en *XML*. Esto permite que la especificación sea mas legible a la persona, creando así una forma relativamente sencilla de escribir las funciones del gestor.

En síntesis, la implementación de *Maven* en un proyecto consiste principalmente en la creación de un archivo definidor conocido como el *POM*. Este fichero es el encargado de esclarecerle a la herramienta las funciones de las cuales esta se debe encargar. En él, se incluyen datos tales como las dependencias del proyecto, la versión actual de este o los módulos a instalar.

A continuación, debido a la naturaleza esencial que presenta el *POM* en *Maven* se va a proceder a ver dicho archivo en una mayor profundidad.

[1] Licencía para uso de *Software* libre.
[2] Lenguaje de definición de entornos basado en el uso de etiquetas.

4.1.1. El *POM*

Como se mencionó anteriormente, el fichero *POM*, o ***P**roject **O**bject **M**odel* (Objeto modelador de proyectos), es un fichero escrito por el usuario en el lenguaje *XML* que define las tareas y funciones de las cuales *Maven* se debe encargar. El fichero *POM* es de una naturaleza extraña, puesto que puede presentar una complejidad muy simple o muy alta dependiendo de las necesidades del proyecto. Por ello, siempre que se habla de *Maven* se establece que su configuración es de una dificultad relativa. Además, un proyecto no está limitado a tener un único *POM*, si no que es posible tener múltiples ficheros relacionados entre sí con el fin de formar una jerarquía de sub-proyectos en forma de árbol.

Teniendo esto en cuenta, no hay mejorar manera de explicar el funcionamiento de este fichero si no viendo como este está compuesto. Por esta razón, se va a proceder a echar un vistazo a los principales campos que forman un *POM*:

```
1          <!-- Cabecera -->
2          <project xmlns="http://maven.apache.org/POM/4.0.0"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5          http://maven.apache.org/xsd/maven-4.0.0.xsd">
6
7          <modelVersion>4.0.0</modelVersion>
```

La cabecera del fichero le hace saber a *Maven* que estamos ante un archivo de este tipo y que debe buscar de la página indicada el lector correspondiente para poder procesar la versión señalada.

```
1          <!-- Información del proyecto -->
2          <!-- Nombre -->
3          <name></name>
4          <!-- Descripción -->
5          <description></description>
6          <!-- Página Web -->
7          <url></url>
8          <!-- Licencias -->
9          <licenses>
10                 <license></license>
11         </licenses>
12         <!-- Desarrolladores -->
13         <contributors>
14                 <contributor></contributor>
15         /contributors>
16         <!-- Nombre organización -->
17         <organization></organization>
18
```

```xml
	<!-- Propiedades Llave:Valor -->
	<properties>
		<key>value</key>
	</properties>
```

La anterior sección define los datos básicos del proyecto. Ninguno de estos campos son obligatorios de definir puesto que no ofrecen información al programa, si no más bien a la persona que lo lee.

```xml
	<!-- Dirección web invertida donde se hospeda el proyecto -->
	<groupId></groupId>
	<!-- Nombre -->
	<artifactId></artifactId>
	<!-- Versión -->
	<version></version>
```

Por otro lado, estos campos son lo que realmente le proporcionan dentro del gestor un nombre al programa. Son unos campos esenciales puesto que marcan la ubicación, nombre y estado del proyecto. Además, son utilizados por *Maven* para generar las relaciones anteriormente mencionadas entre los distintos *POMs* de un ensamblado. Como ejemplo, para *Net2Plan* estos campos quedan de la siguiente forma:

```xml
	<groupId>com.net2plan</groupId>
	<artifactId>net2plan</artifactId>
	<version>0.4.1-SNAPSHOT</version>
```

El sufijo *-SNAPSHOT* sirve para indicar que la versión aún está en desarrollo. Cuando la versión *0.4.1* sea terminada, es necesario quitar la extensión.

```xml
	 <!-- Define relación Padre-Hijo -->
	 <modules>
		<module></module>
	</modules>

	<!-- Define relación Hijo-Padre -->
	<parent>
		<groupId></groupId>
		<artifactId>/artifactId>
		<version></version>
	</parent>
```

Maven ofrece la posibilidad de anidar múltiples proyectos de distintos tamaños en uno único con el fin de generar un gran ensamblado de una forma ordenada y modular. Para esto, se crea una jerarquía entre los módulos en forma de árbol en donde los nodos superiores son conocidos como los padres y los inferiores como los hijos. La construcción de este ensamblado se lleva a cabo entonces siempre desde arriba hacia abajo, es decir, desde los padres a los hijos. Como nota, un mismo módulo puede ser al mismo tiempo padre e hijo de otros módulos.

Hasta el momento solo se han definido campos que son utilizados por *Maven* para definir el entorno del programa, pero aún no se ha tratado ninguna de sus dos funciones principales: la gestión de dependencias y de construcción. A continuación se muestran las opciones utilizadas para describir estas tareas.

```
1          <!-- Dependencias del proyecto -->
2          <dependencies>
3                  <dependency>
4                          <!-- Identificador de la dependencia -->
5                          <groupId></groupId>
6                          <artifactId></artifactId>
7                          <version></version>
8                          <!-- Alcance de la dependencia -->
9                          <scope></scope>
10                 </dependency>
11         </dependencies>
```

Todas las dependencias son definidas de la misma forma dentro de la pila marcada por *dependencies*. Como se puede observar, todas las relaciones entre elementos de *Maven*, tanto para la gestión de dependencias como para la jerarquía padre-hijo, se crean siempre mediante el uso de los tres campos identificadores, puesto que para el gestor estos atributos son los que identifican de forma inequívoca a un elemento de la solución.

El campo más especial de este código es el *scope*. Una etiqueta encargada de limitar la transitividad de una dependencia, o dicho de otra forma, el alcance dentro del proyecto que esta tiene. Para ver en mayor detalle, los valores más comunes que este parámetro acepta son los siguientes:

- **Compile**: Valor por defecto, permite el uso de la dependencia en el proyecto y en aquellos relacionados con él.
- **Provided**: La dependencia será proporcionada de forma independiente durante la ejecución del programa.
- **Runtime**: Similar a **Provided**, pero la dependencia no es necesaria para compilar.
- **Test**: Solo se permite el uso de la dependencia dentro del entorno de prueba de la aplicación.

- **System**: Similar a **Provided**, pero a que indicar la localización local de la dependencia.

Para terminar de hablar del *scope*, es recomendable hacer un hincapié sobre la transitividad de una dependencia. La transitividad está definida por dos conceptos: el alcance y la propagación. El alcance está definido por el recién mencionado *scope*, mientras que la propagación surge de una manera más implícita. Esto se refiere al hecho por el cual la dependencia de una dependencia se vuelve a su vez la dependencia del proyecto base:

$$A \to B \wedge B \to C \Rightarrow A \to C$$

Este dato puede parecer lógico y trivial. Sin embargo, es importante tenerlo en mente puesto que es origen de una comedida cantidad de errores de difícil detección producidos por la aparición de ambigüedades y dependencias repetidas.

```
1          <!-- Opciones de construcción -->
2          <!-- Tipo de empaquetado -->
3          <!-- zip/jar/pom... -->
4          <packaging></packaging>
5          <!-- Perfiles de construcción -->
6          <profiles>
7                  <profile>
8                          <!-- Condiciones de activación -->
9                          <activation></activation>
10                         <!-- Herramientas de ensamblado -->
11                         <build>
12                                 <plugins>
13                                         <plugin></plugin>
14                                 </plugins>
15                         </build>
16                 </profile>
17         </profiles>
```

Finalmente, con estos campos se definen las labores del gestor de empaquetado. Estos trabajos están divididos en lo que se conoce como perfiles, un conjunto de funciones y propiedades cuyo producto final es el ensamblado del proyecto. Un mismo *POM* es capaz de definir múltiples perfiles de trabajo, dando así la posibilidad de empaquetar el proyecto bajo una serie de condiciones: empaquetado final, de producción, etc.

Además, una definición de construcción bajo el campo *build* que no esté dentro de un perfil se considerá la opción por defecto, usada cuando ninguno de los otros perfiles ha sido activado. De hecho, *Maven* ya proporciona este *build* internamente, ejecutando una serie de tareas que tiene definidas en su interior.

Por otro lado, la parte que realmente hace el trabajo en esta sección son los [3]plug-ins, los cuales se encargan de realizar una tarea concreta bajo una serie de propiedades descritas previamente por la persona. Mientras que los *plug-ins* se van a usar son definidos por el usuario, la instalación y mantenimiento de estos es realizado de una forma autónoma y transparente al desarrollador, proporcionando así un mayor agilidad a la hora de integrar o eliminar estas herramientas.

Para poner un ejemplo, un *plug-in* de común aparición en los ensamblados de *Maven* puede ser el siguiente:

```
1         <!-- Comprime el proyecto en un fichero jar -->
2         <groupId>org.apache.maven.plugins</groupId>
3         <artifactId>maven-jar-plugin</artifactId>
4         <version>3.0.2</version>
5         <configuration>
6                 <!-- Configuración de la herramienta -->
7         </configuration>
```

En resumen, se puede observar como *Maven* no tiene ninguna funcionalidad de base para crear empaquetados, todo su trabajo esta basado en el uso de código externo compatible con él.

4.1.2. El ciclo de vida de *Maven*

El ciclo de vida de *Maven* se refiere al proceso completo por el cual el gestor compila, prueba y ensambla el proyecto o artefacto, tal como es conocido en la nomenclatura propia. Este ciclo está compuesto de una serie de pasos o fases que se ejecutan secuencialmente con el fin de generar el empaquetado y despliegue final. Como se puede esperar, el ciclo de vida es altamente configurable, no siendo incluso necesario su fin completo para ser considerado válido. Por defecto, las fases que se siguen en este proceso son las siguientes:

- **Clean**: Limpía aquellos ficheros que hayan sido producidos por un ciclo anterior.
- **Validate**: Comprueba que toda la información necesaria está bien definida y disponible.
- **Compile**: Compila las fuentes del proyecto.
- **Test**: Ejecuta las tareas de prueba sobre las fuentes compiladas.
- **Package**: Empaqueta y da forma al ensamblado.
- **Verify**: Comprueba que los pasos anteriores cumplen una serie de pruebas de integración.

[3]Módulos que añaden funcionalidad a *Maven*.

- **Install**: Instala el ensamblado en un directorio local.
- **Site**: Actualiza la página web del proyecto.
- **Deploy**: Instala el ensamblado en un directorio remoto.

Si *Maven* completa la última fase con éxito, entonces considerá al ciclo como válido. Si tan solo uno de los pasos falla, todos los demás lo harán puesto que cada sección depende de lo generado por las anteriores.

Volviendo a la definición de *plug-ins* vista anteriormente, es posible determinar dentro de la configuración de una herramienta la fase en la cual esta se va a ejecutar. En caso de que dicha fase no esté descrita, cada *plug-in* trae preestablecido consigo una fase en la cual integrarse. Además, si varios *plug-ins* están marcados para funcionar en una misma sección estos son ejecutados en el mismo orden por el que han sido leídos dentro del *POM*.

Para terminar, con la intención de dar una mayor versatilidad al ciclo de vida de *Maven* existen definidos una serie de pasos antes y después de una fase con los cuales es posible preparar su realización.

4.1.3. Los repositorios de *Maven*

A la hora de manejar las dependencias, *Maven* funciona de una forma muy similar a *Git* ya que ambos utilizan paralelamente un repositorio remoto y uno local. Sin embargo, en este caso se plantea una gran diferencia, *Maven* no se espera que el usuario tenga que darse cuenta de este factor.

Por el lado local, el gestor se encarga de construir de forma automática un repositorio oculto en algún lugar del sistema. En él, se genera una biblioteca con las librerías indicadas dentro del *POM* para que sean directamente accesibles en el entorno del proyecto. De esta forma, existe una copia real pero en un principio intocable del archivo *JAR* de cada una de las librerías, así como de sus dependencias.

Por otro lado, el repositorio remoto es un servidor que contiene una gran colección de librerías expuestas de forma pública, junto con la definición en *Maven* de cada una de ellas.

Con todo esto, la parte de gestión de dependencias se plantea sencilla de entender. Para el siguiente código en el *POM*:

```
1    <dependency>
2        <groupId>org.apache.logging.log4j</groupId>
3        <artifactId>log4j-core</artifactId>
4        <version>2.8.2</version>
5    </dependency>
```

Se le indica a *Maven* que debe acceder al repositorio remoto en busca del *JAR* indicado por los tres campos anteriores. El repositorio remoto contiene una base de datos que relaciona cada librería con estos tres campos, de forma que cada *JAR* queda identificado de forma unívoca. Como se puede deducir, el servidor es capaz de almacenar múltiples versiones de una misma dependencia, permitiendo así la retro-compatibilidad con aquellos proyectos que no las hayan actualizado. Una vez encontrado el archivo, este es descargado y almacenado en el repositorio local.

Por supuesto, este sistema no funciona si no existieran repositorios remotos ahí fuera a los que conectarse. A día de hoy existen dos que son altamente utilizados:

- **Maven Central**: Mantenido por la fundación *Apache*. Es el repositorio utilizado comúnmente por defecto y el que mayor número de librerías contiene. No obstante, es muy estricto a la hora de autorizar nuevos contenidos.
- **SonaType Central**: Proporcionado por la compañía *Sonatype*. Es un repositorio mucho más permisivo gracias a que sus requerimientos son muchos más livianos.

Por otro lado, es posible añadir otros repositorios, bien sean propios u otros públicos menos comunes, mediante el uso de los siguientes campos:

```
1   <repositories>
2       <repository>
3           <id></id>
4           <name></name>
5           <url></url>
6       </repository>
7    </repositories>
```

4.2. *Maven* en *Net2Plan*

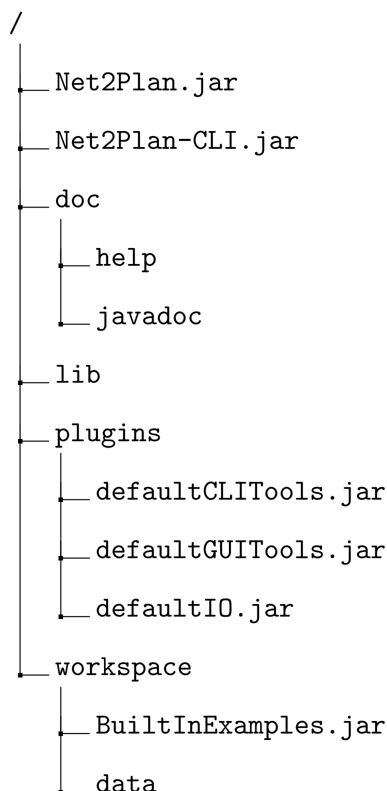
4.2.1. Necesidades

Lo que *Net2Plan* buscaba con la integración de *Maven* era conseguir apoyo a su estado de proyecto *Open Source*. Es decir, se pretendía añadir un mecanismo mediante el cual se pudiera entregar el código fuente con seguridad e independencia. Buscando que el nuevo desarrollador del proyecto sobre deba preocuparse del código en sí y no de lo que lo rodea.

Con este fin, se planteaba la eliminación de *Apache Ant* como elemento empaquetador del proyecto y se borraba la necesidad de entregar explícitamente las dependencias. En su lugar, *Maven* se encarga de integrar abstractamente las librerías externas, además de ofertar todos los mecanismos requeridos tal que el resultado del ensamblado sea exactamente el mismo que se tenía hasta el momento. En síntesis, *Net2Plan* pedía de la integración con *Maven* los siguientes apartados:

1. Proporcionar dependencias de forma automática y transparente.
2. Mecanismo de ensamblado independiente de la plataforma.

Como ya se ha mencionado, el ensamblado con *Maven* tiene que producir exactamente el mismo resultado que el realizado con *Apache Ant*. Por ello, el objetivo del empaquetado es obtener un proyecto con la siguiente estructura:



4.2.2. Implementando *Net2Plan*

La implantación de *Maven* en *Net2Plan* fue un proceso realizado en dos etapas separadas mutuamente en el tiempo. Dichas etapas corresponden respectivamente con la salida de las versiones 0.4 y 0.5 del programa, dando una distinción y evolución al uso que se le daba al gestor en cada una. A si mismo, estos dos pasos se relacionan a su vez con los dos grandes modelos de implantación que la aplicación proporciona: el *Super-POM* y la estructura multi-modular.

Con esto, a continuación se va a proceder a realizar un estudio de estos dos métodos siguiendo el orden cronológico con el que fueron hechos.

4.2.2.1. El Super-POM

A un proyecto se le dice que contiene un estructura en forma de *Super-POM* cuando presenta un único descriptor *POM* para todo el proyecto. Este descriptor se encuentra siempre en el directorio base del código, actuando como un punto de referencia y de entrada a este. En él, se detalla todo aquello que sea relativo al uso de *Maven*, desde la indicación de dependencias hasta la configuración de las tareas de construcción. Dicho de otra forma, un *Super-POM* es un fichero *POM* en el que se integran todos los campos que han sido descritos en los apartados anteriores.

Cuando se comienza a trabajar con *Maven*, esta distribución se presenta como la más intuitiva y lógica de implementar puesto que es similar a lo que un desarrollador está acostumbrado. Es decir, una carpeta en donde se contiene todo el código fuente y un conjunto de ficheros base explicativos. Además, para lo relativamente sencillo que resulta su instalación la gran mayoría de proyectos de pequeño y mediano tamaño no necesitan dar un paso más allá, puesto que esta estructura cubre todas las necesidades habituales. Con todo ello, se comenzó a integrar *Maven* en *Net2Plan* mediante la descripción de un fichero *Super-POM*.

Para comenzar, antes de anda es principal darle forma al proyecto. Como ya se ha visto, la configuración en *Super-POM* difiere muy poco de como un proyecto está originariamente planteado, variando unicamente en el añadido del nuevo archivo. Con esto, el árbol con el que se presenta *Net2Plan* tiene la siguiente forma:

```
/
├── pom.xml
├── ReadMe.md
├── .gitignore
└── src
    ├── assembly
    └── main - /* Código del programa */
```

Una vez se tiene esto, el segundo paso consiste en empezar a dar información sobre el proyecto. En este caso, la cabecera queda de la siguiente forma:

```
1                   <!-- Identificador del artefacto -->
2                   <groupId>com.net2plan</groupId>
3                   <artifactId>net2plan</artifactId>
4                   <version>0.4.1</version>
5
6                   <!-- Información adicional -->
7                   <name>Net2Plan</name>
8                   <url>http://www.net2plan.com</url>
9
10                  <packaging>jar</packaging>
```

Siendo la última línea la de mayor interés en este caso. La etiqueta *packaging* indica la intención final del *POM* presente, dicho de otra forma, su razón de ser. Este campo recibe por lo general formatos de empaquetado tales como *zip* o *jar*. Sin embargo, existe un valor especial simplemente indicado como *pom*, el cual marca que el *POM* actual solo contiene información y no un objetivo. Este campo es completamente incompatible con la estructura en *Super-POM* puesto que pese a que *Maven* es capaz de realizar las tareas de construcción estas no producen un objeto final, dejando todo el proceso por inútil.

El siguiente paso consiste en definir todas las dependencias del programa bajo la pila *dependencies*. Estas dependencias son conocidas ya a que hasta este momento se han entregando de forma directa bajo la carpeta *lib*. El objetivo consiste entonces en encontrar los identificadores de *Maven* para cada una de estas librerías e introducirlos en el *POM*. Para esta tarea, viene de ayuda utilizar buscadores de repositorios tales como "MvnRepository", los cuales se encargan de buscar automáticamente las librerías entre las bases de datos mas conocidas.

Durante la búsqueda de las bibliotecas, aparecen dos librerías que presentan un problema para la gestión por *Maven*: Una librería no almacenada en repositorio y otra contenida en el propio sistema. Con respecto al primer caso, cuando una dependencia no se encuentra almacenada en un repositorio es posible utilizar *GitHub* como repositorio remoto de *Maven* a través del uso de JitPack. Esta página se encarga de leer el *POM* de un proyecto hospedado en *GitHub* y de realizar los correspondientes pasos para poder dar unos identificadores a partir de él. Para poder utilizar este servicio, es necesario declararlo en el *POM* de la siguiente manera:

```
1           <repositories>
2                   <repository>
3                           <id>jitpack.io</id>
4                           <url>https://jitpack.io</url>
5                   </repository>
6           </repositories>
```

A fin de reflejar la estructura final del programa, a las dependencias del proyecto se les asigna un *scope* de alcance *provided* debido a que se plantea seguir proporcionándolas dentro de la carpeta *lib* anterior.

Por el lado de la gestión del ensamblado, se construyen dos perfiles similares entre sí que presentan el uso de los siguiente *plug-ins*:

1. **Maven-Compiler-Plugin**: Compila las fuentes del proyecto bajo la versión 1.8 de *Java*.
2. **Maven-Dependency-Plugin**: Descarga y traslada las dependencias de las fuentes.
3. **Maven-Assembly-Plugin**: Empaqueta los archivos *jar* y los posiciona en sus rutas correspondientes.
4. **Maven-Antrun-Plugin**: Ejecuta labores de limpieza.
5. **Maven-Javadoc-Plugin***: Crea el *JavaDoc* del proyecto. Usado por un perfil, ignorado por el otro.

De esta forma, el ciclo de vida de *Maven* para el ensamblado completo de este proyecto es el siguiente:

- Limpiar el anterior ciclo de vida.
- Descargar las dependencias a una carpeta temporal.
- Compilar las fuentes.
- Desempaquetar *Javadoc*s adicionales.
- Construir el *Javadoc* del proyecto.
- Crear el esqueleto del proyecto.
- Empaquetar las fuentes en los respectivos *jar*.
- Mover todo el resultado a sus carpetas correspondientes.
- Empaquetar el proyecto en un fichero *zip*.
- Labores de limpieza finales.

Tras este proceso, *Maven* proporciona bajo su ruta de salida un único fichero *zip* que contiene todo el proyecto preparado para ser distribuido. En caso de que el ensamblado sea una versión final, es necesario actualizar el valor de la versión del proyecto en el *POM* de acuerdo con la situación.

Como conclusión, en una primera instancia el modelo en *Super-POM* cumple con todas las necesidades de *Net2Plan*. Sin embargo, presenta una serie de inconvenientes que hacen de este un uso incomodo:

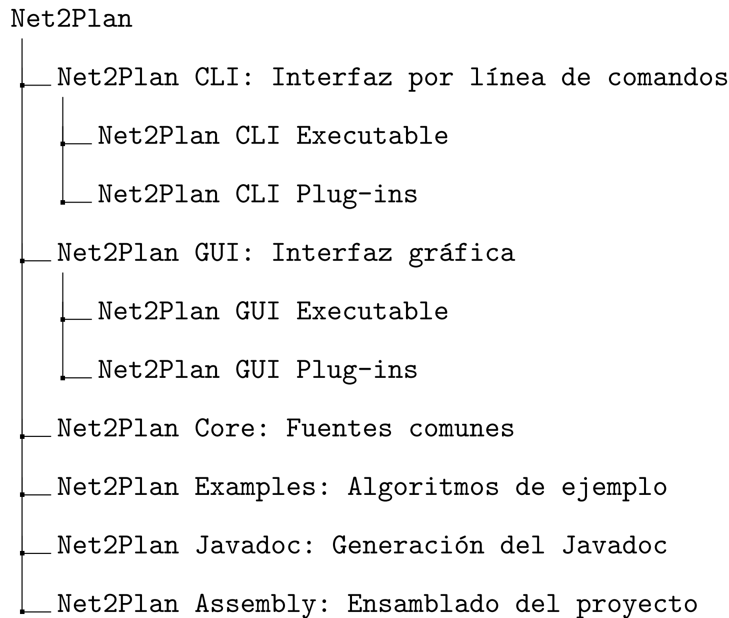
- Debido a la naturaleza de la gestión de dependencias, *Maven* no es capaz de generar un listado de las mismas durante el ensamblado, necesitando de mano humana. Esto provoca que cada vez que se defina una nueva dependencia esta tenga que ser añadida a un listado en el *POM*.
- El *Javadoc* es incapaz de relacionarse con otros como él debido al anterior problema. Esto produce que las dependencias sean tratadas por su cuenta haciendo por ello que el proceso sea más lento.
- La selección de las fuentes para cada *jar* está realizada a mano, siendo por esta razón más propensa a errores y menos escalable.

Existe una solución dentro de *Maven* para todos estos problemas. Sin embargo, dicha solución conlleva un cambio completo de la filosofía de uso. Con ello, se replantea la implementación de *Maven* junto con el siguiente paso: el modelo multi-modular.

4.2.2.2. El modelo multi-módular.

Como ya se ha podido apreciar anteriormente, hasta el momento todas las fuentes han sido tratadas de una sola vez sin distinción. La separación y elección de cada una de estas para los *jar*s que forman el ensamblado es un proceso de filtrado manual que se produce dentro del ciclo de vida. Sin embargo, *Net2Plan* posee la cualidad de estar formado por una serie de códigos fuentes altamente identificables y agrupables. Aprovechando esta característica, se plantea implementar una nueva estructura de proyecto basada en la separación de cada uno de estos conjuntos en un propio módulo de *Maven* contenedor de su gestión de dependencias y construcción personal. Con esto, la separación del código queda reflejada sobre la propia distribución de estos módulos, facilitando así la identificación y ensamblado de los mismos.

Por todo esto, el código de *Net2Plan* queda separado entonces en el siguiente conjunto de sub-proyectos:



Una vez se establezca esta organización, el siguiente paso consiste en crear las relaciones necesarias entre los miembros para que estos puedan funcionar tal como antes. Estas relaciones son divisibles entre dos tipos distintos: padre-hijo e inter-modular.

Con respecto al primer suceso, la relación padre-hijo se realiza en dos pasos. Por la parte del hijo, este marca su módulo padre mediante el uso de la etiqueta *parent*. Para el caso del módulo *Core*, su *POM* contiene estos valores:

```
1         <parent>
2                 <groupId>com.net2plan</groupId>
3                 <artifactId>net2plan/artifactId>
4                 <version>0.5.0</version>
5         </parent>
```

Por el lado del padre, se introducen sin embargo una serie de nuevos conceptos. En primer lugar, un módulo padre no debe contener por si mismo código fuente, este está proporcionado por sus hijos. La razón de este hecho es que un *POM* padre siempre está marcado con:

```
1         <packaging>pom</packaging>
```

Significando que el padre no trae contenidos por su cuenta, si no unicamente información para sus hijos y el proyecto.

En segundo lugar, un *POM* padre indica cuales son sus hijos mediante el uso de los siguientes campos:

```
<!-- Para el caso de Net2Plan-CLI -->
<modules>
        <module>Net2Plan-CLI-Exec</module>
        <module>Net2Plan-CLI-Plugins</module>
</modules>
```

Y para terminar, se introduce un nuevo tipo de declaración conocido como *management*, el cual es posible utilizar tanto para las dependencias como para el ensamblado. Como ejemplo, un *POM* padre tiene dos posibilidades distintas de gestionar el ensamblado:

```
<pluginManagement>
        <plugins>
                <plugin>
                        <groupId></groupId>
                        <artifactId></artifactId>
                        <version></version>
                </plugin>
        </plugins>
</pluginManagement>

<plugins>
        <plugin>
                <groupId></groupId>
                <artifactId></artifactId>
                <version></version>
        </plugin>
</plugins>
```

Siendo la diferencia entre ellos que *management* declara y *plugins* utiliza. En otras palabras, *management* indica a sus hijos que en el caso de que utilicen la herramienta declarada, estos reciben dicha herramienta con una configuración predeterminada aquí. Por el otro lado, *plugins* indica que los hijos tienen que utilizar la herramienta dada y además con la configuración proporcionada.

Como nota final, un hijo siempre tiene la oportunidad de sobrescribir la información que proviene de su padre. Por lo tanto, la última palabra sobre la gestión de un módulo siempre recae sobre el eslabón más bajo, es decir, él mismo.

La relación inter-modular es un proceso mucho más directo de ejecutar. Estos vínculos se crean mediante el mecanismo de dependencias visto tantas veces hasta el momento. Para el caso de un módulo que depende del *Core*:

```
1            <dependencies>
2                    <dependency>
3                            <groupId>com.net2plan</groupId>
4                            <artifactId>net2plan-core</artifactId>
5                    </dependency>
6            </dependencies>
```

Visto todo esto, la integración final de este modelo dentro de *Net2Plan* se ejecutó siguiendo estas características:

- Todas las dependencias no modulares son declaradas en el *POM* base bajo un *scope compile*. No se hace uso de *management* para la gestión de dependencias.
- Todos los miembros dependen de *Net2Plan Core*.
- Los módulos *Plug-ins* dependen de su correspondiente módulo ejecutable.
- El *POM* base indica las próximas herramientas bajo *management*:
    - **Maven-Compiler-Plugin**: Ejecuta labores de compilación.
    - **Maven-Jar-Plugin**: Construye el *jar* de cada sub-proyecto de forma independiente.
    - **Maven-Assembly-Plugin**: Recoge los *jar* del *plug-in* anterior y los coloca en su ubicación final.

Finalmente, con el uso de este modelo se han solventado todos los problemas relacionados con el *Super-POM* debido a que se hace un mejor aprovechamiento del funcionamiento y administración de *Maven*.

Parte II

Mejoras de visualización en Net2Plan

Capítulo 5

Interfaz con *OpenStreetMaps*

5.1. Introducción

OpenStreetMaps, también conocido como *OSM*, es un servicio de cartografía en línea similar al mayormente conocido *Google Maps*. La principal diferencia de este proyecto con sus semejantes es su filosofía de código abierto que fomenta la participación de la comunidad para su desarrollo. De esta forma, los usuarios son capaces de subir sus propias rutas y puntos de interés con el fin de enriquecer la experiencia para el resto de usuarios. La naturaleza abierta de *OpenStreetMaps* permite el uso de su base de datos de mapas bajo una serie de términos y condiciones de carácter laxo descritas dentro de su licencia *ODbL*. Debido a lo anterior, existe una extensa biblioteca de librerías, la cual abarca un importante número de lenguajes, que permiten el uso de estos mapas a varios niveles.

Este apartado se centra pues en el aprovechamiento de las cualidades de *OSM* para su integración dentro del entorno de *Net2Plan*.

5.2. Objetivos

El principal objetivo detrás de esta mejora consiste en la creación de una interfaz entre *Net2Plan* y *OpenStreetMaps* tal que los mapas proporcionados por este servicio puedan ser usados como imágenes de fondo en la visualización de topologías.

De esta manera, se pide:

1. Mostrar el servicio de mapas como fondo del panel de visualización de topologías.
2. Ubicar los nodos de la topología sobre sus puntos geográficos correspondientes dentro del mapa.
3. Permitir que *OSM* responda antes eventos tales como el desplazamiento de la pantalla o el cambio de *zoom*.

Como última condición, ninguno de los anteriores pasos debe modificar en ningún momento los datos de la topología. Produciendo con ello que la interfaz sea independiente en todo momento de los datos de entrada.

5.3. Glosario de términos

- **OpenStreetMaps (OSM)**: Servicio de cartografía en línea de libre distribución.
- **Swing**: Entorno predeterminado de *Java* para la creación de interfaces gráficas.
- **JUNG**: Librería de dibujo de grafos utilizada en *Net2Plan*.
- **JXMapViewer2**: Librería de control de *OSM* dentro de *Swing*.
- **JPanel**: Panel contenedor de elementos de visualización en *Swing*.

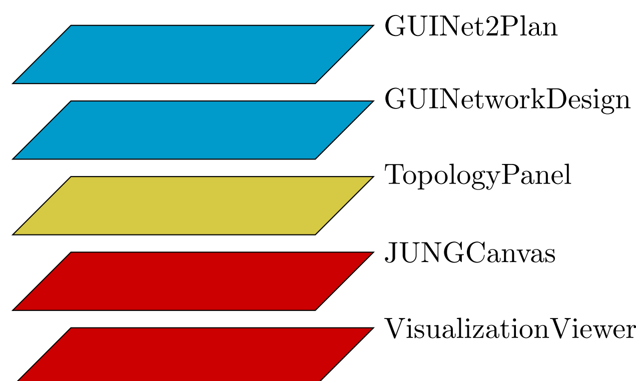
5.4. Implementación

5.4.1. Añadiendo el fondo

Obviamente, no es posible trabajar sobre una cartografía si no se posee acceso al registro de mapas. Por ello, el primer paso de la implementación con *OSM* consiste en conseguir que sus mapas sean mostrables al usuario dentro del entorno *Swing* de *Net2Plan*.

Con esta meta en mente, se decide hacer uso de una librería de código abierto conocida como [1]*JXMapViewer2*. Lo que esta librería proporciona es un *JPanel* capaz de contactar con los servidores de *OpenStreetMaps* para la descarga y visualización de sus mapas en tiempo real. Además, proveé una serie de controles mediante los cuales el programador es capaz de interactuar con el servidor. Estas características abarcan todas las necesidades que *Net2Plan* plantea, lo que ahorra trabajo de integración.

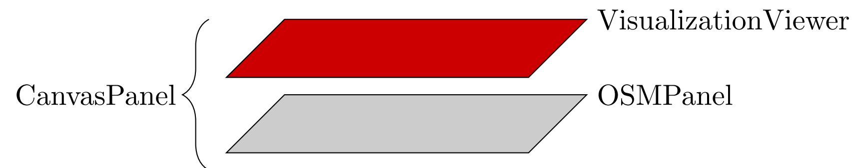
Antes de continuar con el añadido del nuevo panel, es conveniente echar un vistazo a la distribución del dibujado que presentan la topologías en *Net2Plan*:



Siendo el *VisualizationViewer* el lugar donde se dibujan los nodos y enlaces.

[1] *JXMapViewer2* en *GitHub*.

Con ello, la idea es crear una nueva capa de dibujo que se encuentre por debajo del *VisualizationViewer*. Para ello, se crea un nuevo panel expositor de mapas cuyo nombre será *OSMPanel* y a su vez un panel contenedor llamado *CanvasPanel*. A este *CanvasPanel* se le asigna una disposición en forma capas de manera que sus componentes se ordenen con dicha estructura. Finalmente, *OSMPanel* se ubica en la parte inferior del contenedor mientras que *VisualizationViewer* lo hace en la superior:



Finalmente, se especifica que *VisualizationViewer* sea transparente para que la capa inferior pueda ser visible.

El resultado de este proceso se refleja en la siguiente imagen:

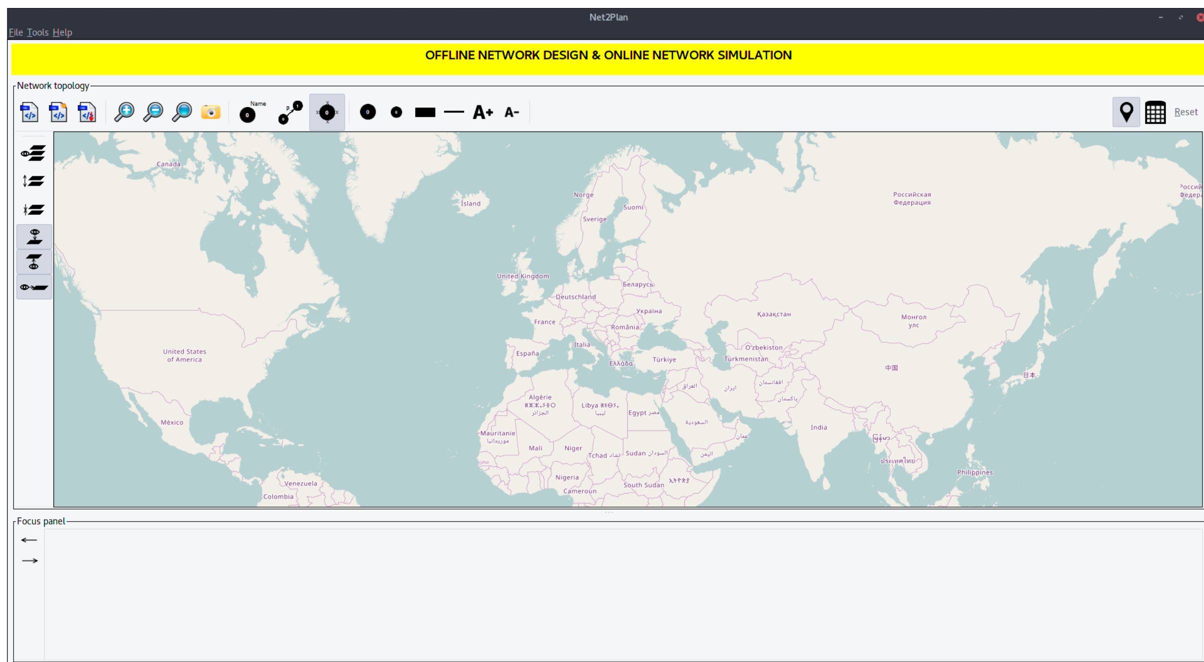


Figura 5.1: Mapa *OpenStreetMaps* en *Net2Plan*

5.4.2. Sincronizando *JUNG* y *OSM*

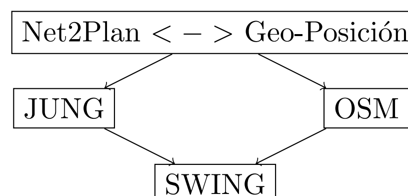
Para establecer una concordancia entre los paneles de *JUNG* y *OSM*, primero es necesario conocer los sistemas de coordenadas entre los que estos se mueven. Así pues, los tipos de coordenadas que aparecen en este caso son los siguientes:

- **Coordenada geográfica**: Punto sobre la tierra definido mediante la latitud y longitud. El eje X circula entre -180 y 180 mientras que el eje entre el -90 y 90.
- **Coordenada *Net2Plan***: Punto mediante el que se define la ubicación de un nodo en *Net2Plan*.
- **Coordenada *JUNG***: Ubica un punto dentro del lienzo de *JUNG*. Sistema de coordenadas no absoluto, una misma ubicación puede ser definida por múltiples puntos. El sistema depende pues del nivel de enfoque en cada momento.
- **Coordenada *Swing***: Define un punto dentro de la interfaz gráfica de *Swing*.

Junto a esto, se conocen además los siguientes datos que ayudan y permiten crear la interfaz entre estos sistemas de coordenadas:

1. *JUNG* contiene mecanismos para transformar un punto de *Net2Plan* en un punto propio.
2. Las coordenadas de *Net2Plan* deben ser estrictamente puntos geográficos.
3. *JXMapViewer2* permite convertir una coordenada geográfica en un punto de *Swing* y viceversa.
4. Cuando *JUNG* no presenta ningún tipo de *zoom*, sus coordenadas casan con las de *Swing*.

De este modo, se define un conjunto de interacciones suficiente tal que se puedan relacionar todos los sistemas de coordenadas. Por lo tanto, las coordenadas de los nodos en *Net2Plan* proporcionan el punto base en el que los demás sistemas se apoyan. *JUNG* y *OSM* se encargan de transformar dicho punto por su cuenta a su propio entorno. Finalmente, *JUNG* y *OSM* deben adaptarse a *Swing* para que ambos correspondan.



Así pues, el algoritmo con el que se ejecuta todo este proceso es el que se puede ver a continuación:

I.- **JUNG**: Eliminar el zoom actual.

II.- **OSM**: Transformar las coordenadas geográficas dadas por los nodos en puntos de *SWING*.

III.- **JUNG**: Mover los nodos a los puntos calculados en el punto anterior.

IV.- Calcular la diferencia de desplazamiento entre el centro del lienzo y del mapa.

V.- **JUNG**: Desplazar el lienzo tal que los centros coincidan.

Y con ello, el producto resultante es el siguiente:

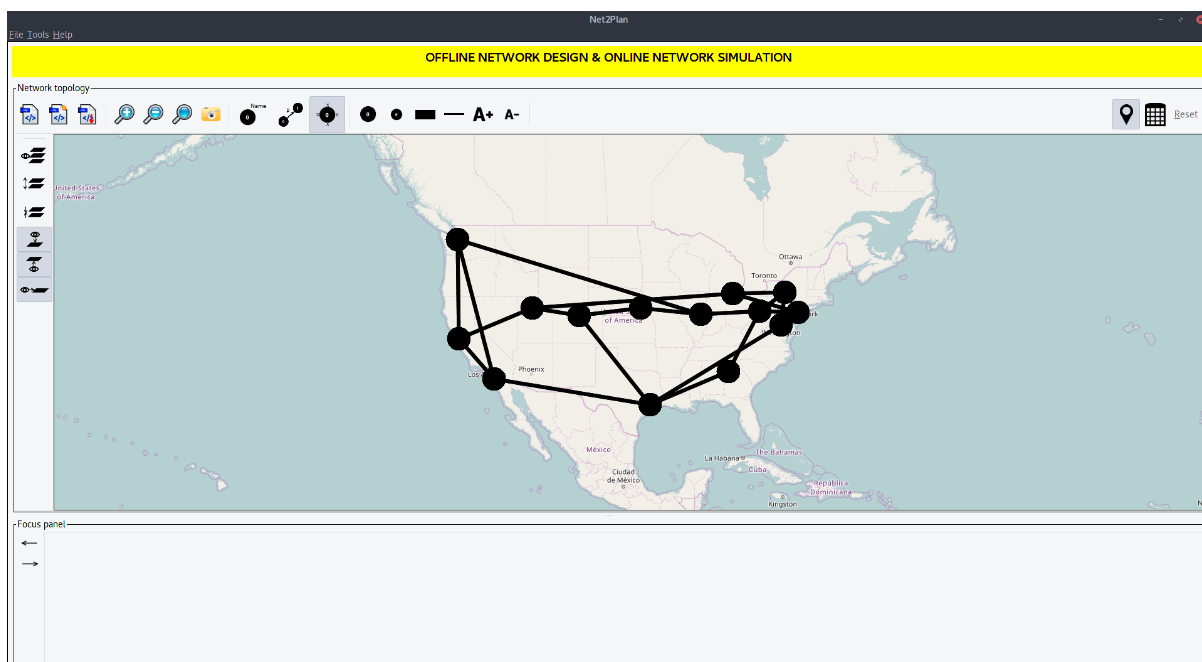


Figura 5.2: Topología sobre mapa de *OpenStreetMaps*

5.4.3. Interacción con el usuario

En la actualidad, *JUNG* proporciona al usuario dos tipos distintos de interacción: el desplazamiento del lienzo y el incremento/decremento del enfoque. El primero caso permite que el usuario pueda moverse sobre el espacio del lienzo con el fin de que este pueda ver topologías de gran tamaño que no entran dentro de una pantalla. El segundo permite cambiar el nivel de zoom para que la topología sea vista desde más cerca o más lejos.

Al mismo tiempo, *OSM* presenta unos mecanismos similares que actúan directamente sobre el mapa que se presenta. Por ello, el objetivo de este apartado consiste entonces en relacionar los medios de ambos sistemas tal que la acción del usuario produzca un cambio sincronizado en los dos.

En síntesis, las interacciones que van a ser tratadas son la que siguen a continuación:

- **Pan**: Desplazamiento del lienzo dentro del espacio de *JUNG*.
- **Zoom In**: Acercar el enfoque para ver la topología más cerca.
- **Zoom Out**: Alejar el enfoque para ver la topología más lejos.

5.4.3.1. Panning

En el caso del *Panning*, la implementación se ha realizado de tal forma que *OSM* es el miembro que recibe el evento y *JUNG* el que se adapta a este. Por ello, el evento del ratón es recibido por el controlador de *OSM*, quién realiza el desplazamiento correspondiente sobre el panel del mapa. A continuación, *JUNG* se da cuenta de dicho movimiento, se encarga de calcular la diferencia entre su centro y el nuevo y de producir el movimiento correspondiente.

5.4.3.2. Zoom

El zoom presentado por *OSM* y por *JUNG* son de distinta naturaleza. Es decir, los niveles de enfoque que proporciona *OSM* sobre sus mapas son de valores discretos mientras que los de *JUNG* son continuos. Para poder construir una interfaz entre ambos, es necesario conocer una expresión que relacione estos sistemas. Afortunadamente, la documentación de *OpenStreetMaps* nos indica la siguiente correspondencia entre *OSM* y *SWING*:

$$ZOOM_{SWING} = 2^{-ZOOM_{OSM}}$$

A sabiendas de que *JUNG* en su nivel de enfoque base corresponde con *SWING*. Para que ambos sistemas correspondan es necesario aplicar el valor resultante de la expresión anterior al lienzo de *JUNG*.

Capítulo 6

Conclusiones

6.1. Situación final del proyecto

Tras experimentar todos los cambios que han sido descritos en este proyecto, la aplicación se encuentra en un estado final tal como se refleja a continuación:

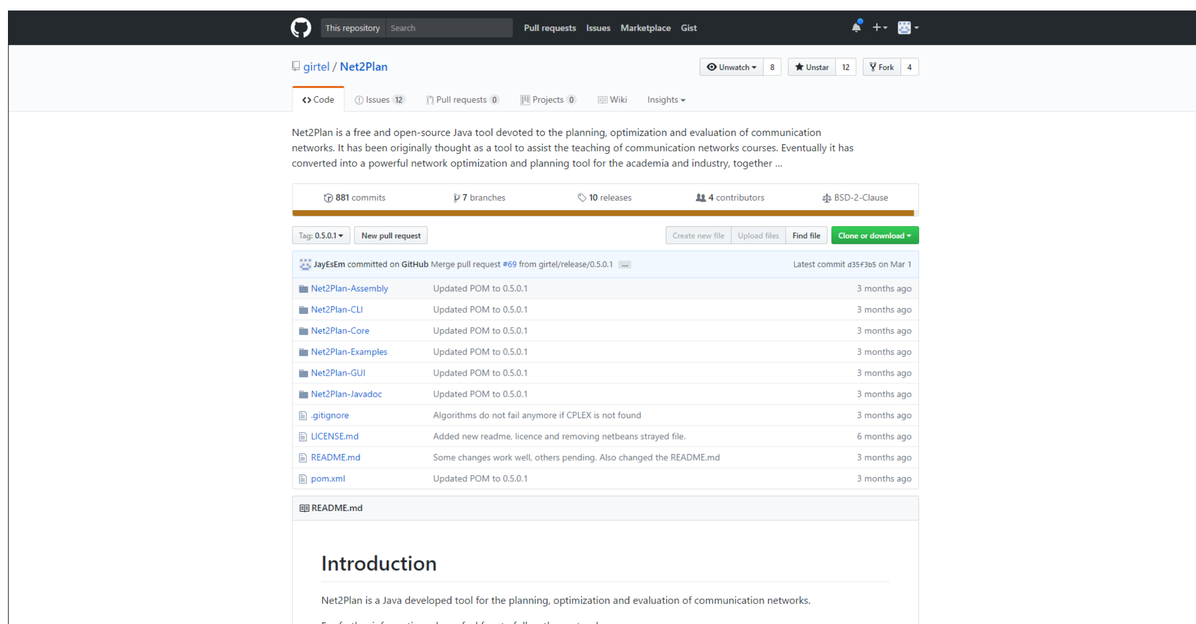


Figura 6.1: *Net2Plan* en *GitHub*

En esta figura se puede apreciar tanto la implementación de *Git* como la de *Maven*. Para el primer caso, la imagen muestra la página principal del repositorio remoto, el cual está actualmente hospedado en *GitHub*. En esta página, usuarios de todo el mundo son capaces de descargar la última versión de *Net2Plan*, acceder al código o incluso hacer sus propias copias del mismo. Por el otro lado, la estructura de las carpetas junto con el *POM* reflejan la distribución multi-modular de la que la aplicación hace gala actualmente. Todo ello puede ser visto en el repositorio de *GitHub* a partir de la versión 0.5 del proyecto.

6.2. Pasos futuros

Net2Plan ha dado un gran salto hacia el desarrollo de *software* moderno a través del uso de estas nuevas herramientas. Sin embargo, el mundo de la ingeniería del *software* es tan grande a día de hoy que está modernización no tiene por qué acabar aquí.

A partir de este punto, *Net2Plan* se ha dedicado a continuar su evolución mediante la adaptación de nuevas herramientas tales como la integración continua o la filosofía de desarrollo basado en *tests* (TDD). A día de hoy, utensilios tales como *Travis CI* y *JUnit* forman parte de la rutina de trabajo diaria dentro de la aplicación.

Bibliografía

[1] Página web net2plan. [Online]. Available: http://net2plan.com

[2] Descripción pom. [Online]. Available: https://maven.apache.org/pom.html

[3] Proyecto maven. [Online]. Available: https://maven.apache.org

[4] Proyecto git. [Online]. Available: https://git-scm.com

[5] Filosofía gitflow. [Online]. Available: https://danielkummer.github.io/git-flow-cheatsheet/

[6] Librería jxmapviewer2. [Online]. Available: https://github.com/msteiger/jxmapviewer2

[7] Openstreetmaps. [Online]. Available: https://www.openstreetmap.org/

[8] Recursos openstreetmaps. [Online]. Available: https://wiki.openstreetmap.org/wiki/Main_Page