UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

etsinf

PROYECTO FIN DE CARRERA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GENERACIÓN DE INTERFACES GRÁFICOS AUTOMÁTICOS A PARTIR DE ONTOLOGÍAS, APLICACIÓN A GUÍAS CLÍNICAS.

Autor: Alfonso María Pérez González

Director: Juan Miguel García Gómez

Co-Director: Carlos Sáez Silvestre

Director de empresa: Carlos Angulo Fernández

Valencia, Septiembre 2011

Agradecimientos

Me gustaría agradecer a mi tutor de proyecto Juan Miguel García por su ayuda y rápidas contestaciones a lo largo del proyecto. A todo el grupo de IBIME por y en especial a Carlos Sáez por estar allí siempre que surgían dudas y a Montserrat Robles por darme la oportunidad de trabajar con todos ellos.

También querría agradecer a mi familia por apoyarme incondicionalmente y a mi novia por aguantarme en las épocas de estrés.

Finalmente agradecer a mi perro por todos estos años de cariño incondicional, esté donde esté seguro que se está echando unas siestas de órdago.

Contenido

Tabla de figuras 5

Glosario 6

Acrónimos y abreviaturas 6

1. Introducción 7

1.1. Motivación 7

1.2. Objetivos del proyecto 7

1.3. Plan de trabajo 8

1.4. Organización de la memoria 8

2. Guías clínicas electrónicas 10

2.1. Introducción a las guías clínicas 10

2.2. Necesidad de las guías clínicas electrónicas 11

2.3. Estado del arte 11

2.4. PRO*forma* 13

2.4.1. Porqué PRO*forma* 13

2.4.2. Trasfondo 14

2.4.3. Descripción del sistema 15

3. Desarrollo web 19

3.1. Opciones disponibles 19

3.2. Vaadin 19

3.2.1. Porqué Vaadin 20

3.2.2. Beneficios 20

3.2.2.1. Ejecución en el servidor 20

3.2.2.2. Eficiencia 21

3.2.3. Problemas de rendimiento 21

3.2.3.1. Programa a desarrollar 21

3.2.3.2. Número de peticiones 22

3.2.4. *Addons* usados en Vaadin 23

4. Implementación del proyecto 25

4.1. Esquema de ejecución de Vaadin 25

4.2. Esquema general de la aplicación 26

4.3. Estructura del proyecto 27

4.3.1. Relación del motor de PRO*forma* con la aplicación y el SADC 27

4.3.2. Creación de la interfaz gráfica 29

4.3.3. Creación del componente grafo 31
4.4. Gestión de las sesiones de usuario 33
5. Guía clínica estudiada (CHF- COPD) 36
5.1. Introducción a la guía 36
5.2. Modelo de espirometría en el subplan EPOC 36
5.2.1. Calculo de los parámetros requeridos (FEV1 predicted y FEV1/FVC) 37
5.3. Cambios realizados a la guía 38
6. Conexión del sistema ayuda a la decisión clínica 39
6.1. SADC utilizado 39
6.2. Sistema de enlace creado entre la aplicación y el SADC 39
7. Conclusiones 41
7.1. Resumen del trabajo realizado 41
7.2. Futuras líneas de investigación 41
Bibliografía 43

Tabla de figuras

Figura 1: El modelo de tarea en PROforma 14
Figura 2: Conjunto de componentes de PROforma 16
Figura 3: Relación de los datos en PROforma 16
Figura 4: Transiciones entre los estados de las tareas 17
Figura 5: Aplicaciones de internet enriquecidas disponibles (RIA, Rich Internet Applications) 19
Figura 6: Arquitectura cliente-servidor de Vaadin 25
Figura 7: Arquitectura de la aplicación desarrollada 26
Figura 8: Conjunto de clases creadas para la gestión del motor de PROforma 27
Figura 9: Estructura del patrón de diseño Adapter, utilizado para un componente genérico 28
Figura 10: Clases creadas al diseñar la interfaz de usuario 29
Figura 11: Patrón de diseño Composite utilizado por los componentes de la aplicación 29
Figura 12: Arquitectura de los componentes de la interfaz de usuario 31
Figura 13: Funciones de las clases implementadas al crear un nuevo componente 31
Figura 14: Estructura de clases y funciones llamadas en la creación del componente VPROformaGraph. 32
Figura 15: Clases creadas para representar el grafo de PROforma 32
Figura 16: Cambio de referencia durante un procesamiento secuencial de peticiones 34
Figura 17: Cambio de variables locales durante un procesado de peticiones concurrente 34
Figura 18: Clases encargadas de la gestión de los datos del usuario 35
Figura 19: Sub-plan elegido 36
Figura 20: COPD_diagnosis_plan original 36
Figura 21: Fórmula utilizada en el SADC 38
Figura 22: COPD_diagnosis_plan modificado 38
Figura 23: Esquema XML utilizado para definir los datos externos de la guía clínica 39

Glosario

Acrónimos y abreviaturas

API Application Programming Interface

COPD Chronic Obstructive Pulmonary Disease

CSS Cascading Style Sheets

DOM Document Object Model

DSS Decision Support System

EPOC Enfermedad Pulmonar Obstructiva Crónica

GWT Google Web Toolkit

HTML Hypertext Markup Language

JSON JavaScript Object Notation

SADC Sistema de Ayuda a la Decisión Clínica

SVG Scalable Vector Graphics

UML Unified Modeling Language

VML Vector Markup Language

XML Extensible Markup Language

1. Introducción

Este proyecto se ha llevado a cabo dentro del marco de unas prácticas en empresa realizadas en la asociación ITACA[1] entre octubre de 2010 y septiembre de 2011, en colaboración con el grupo de investigación de minería biomédica[2] perteneciente al grupo de trabajo IBIME[3].

La aplicación puede ser probada en la dirección web[4]. También se dispone de un manual de usuario online.

1.1. Motivación

A lo largo de las últimas dos décadas se han creado cientos de guías clínicas con la intención de ayudar a los médicos y a los pacientes en la decisión acerca del tratamiento apropiado ante unas circunstancias clínicas concretas (1) (2) (3). Sin embargo, este proyecto surge tras comprobar que, a pesar de la enorme cantidad de guías clínicas disponibles actualmente, ningún sistema de ejecución de éstas era capaz de relacionarse con un sistema de ayuda a la decisión de manera sencilla, por lo que la ejecución de dichas guías se limitaba a la introducción de los datos requeridos por parte del personal médico y a la aplicación de un diagrama de flujo de datos en el que las decisiones tomadas se basaban exclusivamente en datos estáticos, ya fuesen introducidos por el médico o tomados del propio historial del paciente.

Dados los actuales problemas que tiene el personal médico y los pacientes con el acceso a las guías clínicas y la ejecución de éstas (4) (5) (6), también se planteó que este proyecto sirviera como un puente que facilite el acceso del personal a las guías clínicas electrónicas. Debido a que existe ya una gran cantidad de lenguajes disponibles para definir correctamente éstas guías, y que crear un nuevo lenguaje para definir guías quedaba fuera del ámbito del trabajo dada la complejidad y tiempo que esto supone, en este proyecto se planteó la creación de una interfaz de usuario que utilizase alguno de los lenguajes de definición de guías clínicas electrónicas disponibles actualmente. Además esta interfaz permitirá un manejo sencillo y cómodo, y mostrará toda la información disponible durante la ejecución de la guía y el acceso a ella de una manera rápida y precisa.

1.2. Objetivos del proyecto

En este proyecto se implementará un generador automático de interfaces gráficas desde la especificación formal de guías clínicas teniendo en cuenta las reacciones de las acciones/decisiones y el estado del paciente. Este sistema se relacionará con el motor de clasificación del sistema de ayuda a la decisión médica Curiam permitiendo el uso de modelos predictivos para ayudar a las decisiones en determinados nodos de la guía.

[1] http://www.itaca.upv.es/view.php
[2] http://www.ibime.upv.es/bmg
[3] http://www.ibime.upv.es/
[4] http://www.ibime.upv.es/bmg/guidelineguidemo

Ya existen aplicaciones que cargan y ejecutan guías clínicas permitiendo seguir así la evolución del paciente a lo largo de esta (p. ej. Protégé[5], que sirve para editar y ejecutar lenguajes tales como EON, GLIF, PRODIGY o PRO*forma,* de los cuales se habla más adelante en esta memoria). Este proyecto va un paso más allá y relaciona la ejecución de estas guías con un sistema de ayuda a la decisión clínica, de manera que el sistema de ayuda a la decisión tomará los datos de entrada proporcionados por la guía clínica, los clasificará y el resultado de esta clasificación será ofrecido a la guía como sugerencia. Posteriormente la guía podrá utilizar dichos datos como valores de entrada, posibilitando así una mayor automatización en la toma de decisiones. Todo este proceso estará supervisado por la persona que utilice la guía, siendo éste, en todo momento, el que tome la decisión final, y se llevará a cabo a través de una interfaz de usuario sencilla que facilitará el manejo y que funcionará vía web.

El proyecto se divide en tres grandes partes:

1. El estudio del estado del arte sobre lenguajes de ejecución de guías clínicas.
2. La ampliación del lenguaje de guías clínicas y de su motor de ejecución al acceso a la ayuda a la decisión.
3. La creación de una interfaz gráfica web que permita ver la evolución de las guías a lo largo de su desarrollo.

La descripción del trabajo desarrollado en cada una de estas partes se realiza en los capítulos posteriores de la memoria.

1.3.Plan de trabajo

El plan de trabajo utilizado en la realización de este proyecto fue el siguiente:

- Estudio del estado del arte de las guías clínicas electrónicas y elección de un tipo de éstas
- Estudio de la viabilidad para enlazar el motor de ejecución del formato de guía seleccionado con un sistema de ayuda a la decisión o de crear uno nuevo
- Estudio de las tecnologías requeridas para el desarrollo del proyecto (lenguaje de programación, entorno de trabajo, instalación en máquina o a través de web,…)
- Aprendizaje de los sistemas elegidos y especificación de requisitos para modelar correctamente la aplicación
- Implementación del proyecto para que fuese accesible vía web y despliegue de éste en un servidor dedicado
- Documentación del proyecto

1.4.Organización de la memoria

En este apartado se resumen brevemente los capítulos de la memoria:

[5] http://protege.stanford.edu/

1) Breve explicación del proyecto realizado, sus objetivos y cómo se ha trabajado para alcanzarlos
2) Explica la necesidad de las guías clínicas electrónicas en la actualidad, presenta el estado del arte de los lenguajes existentes para definir guías clínicas electrónicas y describe la sintaxis y la semántica de PRO*forma* y porqué se eligió éste como lenguaje de especificación a utilizar
3) Se explica el entorno de programación de Vaadin, porqué se eligió éste frente a otras opciones disponibles y las consecuencias de esta decisión
4) Se detalla la creación del proyecto, se explican brevemente las clases utilizadas más importantes y se muestran esquemas UML y de relación de las clases. Además se explica los patrones de diseño utilizados durante el desarrollo y los beneficios que éstos nos ofrecen
5) Descripción de la guía clínica seleccionada y de las modificaciones que hubo que hacer para adaptarla al proyecto. También se explica brevemente como se calculan los parámetros que requerirá dicha guía y porqué estos son importantes dentro del diagnóstico de la enfermedad pulmonar obstructiva crónica.
6) Detalla cómo se conecta la aplicación implementada con el sistema de ayuda a la decisión clínica, muestra un ejemplo del XML creado y como se relacionan a través de dicho esquema la aplicación desarrollada y el sistema de ayuda.
7) Conclusiones de la memoria y futuras líneas de investigación que han ido surgiendo durante el desarrollo del proyecto
8) Bibliografía

2. Guías clínicas electrónicas

2.1.Introducción a las guías clínicas

Las guías clínicas son un poderoso método para la estandarización y la mejora de la calidad del cuidado médico. Suelen definirse como un conjunto de planes, organizados de manera esquemática en varios niveles, que gestionan a pacientes que tienen una condición clínica en particular (p. ej. enfermedad pulmonar obstructiva crónica (7)).

Una posible manera de interpretar una guía clínica es como un conjunto de restricciones que se utilizan en el proceso de aplicación de la guía (es decir, las acciones que debe realizar el médico) junto con sus terminaciones deseables (es decir, los estados del paciente). Estas restricciones son, generalmente, temporales o como mínimo tienen una dimensión temporal significativa (ya que casi todas las guías clínicas se centran en el cuidado a pacientes crónicos), o por lo menos especifican un conjunto de cuidados que deben ser aplicados a lo largo de un determinado espacio de tiempo. Otra manera de interpretarlas sería como planes esquemáticos reutilizables que se aplican a un paciente determinado y necesitan ser ejecutadas por el médico a lo largo de significativos periodos de tiempo, a la vez que ofrecen cierta flexibilidad para la consecución de metas específicas.

Actualmente se reconoce que las guías clínicas basadas en evidencias médicas, han demostrado ser capaces de apoyar mejoras en la calidad y la consistencia de la sanidad, un hecho que ha sido demostrado en (8) y (9), pudiendo incluso llegar a reducir los costes de los cuidados médicos en determinadas ocasiones (10).

2.1.1. Propósito de las guías clínicas

Para poder comprender mejor las guías clínicas y porqué estas pueden llegar a ser tan importantes en la práctica de la medicina si se aplican habitualmente, a continuación se describen una serie de propósitos que intentan cumplir las guías:

- Describir los cuidados médicos apropiados basados en pruebas científicas y en el consenso general
- Reducir las variaciones no justificadas que se puedan dar en la práctica médica.
- Proveer de una base de referencias más racional, basadas en las mejores prácticas clínicas
- Proveer de un foco para una educación continua
- Promover el uso eficaz de los recursos
- Actuar como foco para los controles de calidad, incluidas las auditorías
- Resaltar las deficiencias que puedan existir en la literatura y sugerir las correspondientes futuras investigaciones

2.1.2. Guías clínicas electrónicas

Las guías clínicas electrónicas codifican las recomendaciones basadas en ensayos clínicos y son capaces de ofrecer nuevas recomendaciones acerca de los procedimientos médicos a seguir de acuerdo a un paciente. Este tipo de guías presentan bastantes beneficios en comparación con las guías clínicas escritas en papel, como se puede ver a continuación:

- Ofrecen referencias a las que se pueden acceder de manera sencilla, dotando así de acceso selectivo al conocimiento de la guía
- Ayudan a revelar errores en el contenido de la guía
- Ayudan a mejorar la claridad de la guía, por ejemplo, respecto a las decisiones y las recomendaciones clínicas
- Ayudan a describir de manera más precisa los posibles estados de un paciente
- Pueden proponer de manera automática las soluciones oportunas para un caso
- Ayudan a la decisión específica para un paciente y en los recordatorios con respecto a éste
- Pueden comprobar inconsistencias en la guía clínica

2.2.Necesidad de las guías clínicas electrónicas

La gran mayoría de las guías que existen actualmente están escritas en papel y son inaccesibles para los médicos que más las necesitan (11). Incluso cuando las guías existen en formato electrónico, y éstas están disponibles a través de la web, los médicos rara vez tienen el tiempo y los medios necesarios para decidir cuáles de las guías disponibles se ajusta mejor a su paciente y si al final deciden usar una, no están seguros de que podrían conseguir aplicando esa guía a ese paciente en particular debido a la actual sobrecarga de información a la que se enfrentan (12). Debido a esto, estos profesionales necesitan procesar más datos ahora que nunca y cada vez en periodos más cortos de tiempo.

Para apoyar las necesidades de los profesionales médicos, así como las de los administradores, y asegurar la calidad continua de la atención, son necesarias las herramientas de procesamiento de información más sofisticadas. Debido a las limitaciones de las tecnologías de última generación, no es factible el análisis de guías textuales sin una estructura definida, por lo que hay una urgente necesidad para facilitar la difusión y aplicación de guías clínicas electrónicas que se puedan leer y ejecutar de manera automatizada.

Existe una gran cantidad de las tareas que se engloban en el cuidado basado en guías clínicas que se podrían beneficiar de la ejecución automática de éstas, entre ellas están: la especificación y el mantenimiento de dichas guías, la selección de las guías apropiada para cada paciente, la ejecución en tiempo real de la guía y la evaluación en retrospectiva de la calidad conseguida al aplicar la guía. Debido a la enorme cantidad de campos a los que se pueden aplicar la guías clínicas electrónicas y que estas son definidas por grupos de trabajo independientes alrededor de todo el mundo, no todos los lenguajes creados para definirlas poseen las mismas características y por lo tanto, teniendo en cuenta el caso en el que se vayan a aplicar, es conveniente usar un lenguaje para definir una guía u otro.

2.3.Estado del arte

Si analizamos lo comentado anteriormente, nos damos cuenta de que hoy en día hay una clara necesidad de herramientas de ayuda que resulten efectivas en el área de la salud. Dichas herramientas podrían ayudar al procesamiento de la gran cantidad de información que se genera para los médicos y administradores. Para que sean efectivas, estas herramientas necesitan estar relacionadas con el registro del paciente, además deben usar el vocabulario médico estándar, deben tener una semántica clara, deben facilitar el mantenimiento y la distribución del conocimiento y tienen que ser suficientemente expresivas para capturar el

diseño racional (proceso e intención de resultado) del autor de la guía mientras permite flexibilidad al médico que lo use y a sus métodos preferidos.

Numerosos métodos que permiten la ejecución de guías electrónicas han sido o están siendo desarrollados por la comunidad de informática médica en las últimas décadas (como se verá a lo largo de los siguientes párrafos), entre ellos están los métodos que se enfocan a la tarea de apoyar el cuidado médico basado en guías clínicas y que usan los datos del paciente. Éstos codifican las guías como tablas simples de estados/transiciones o como reglas de acción/situación dependientes del registro médico electrónico (13). Un estándar definido por el equipo de HL7 (*Health Level Seven*)[6], de representación del conocimiento médico, es Arden Syntax (14), representa el conocimiento médico como unidades independientes denominadas módulos de lógica médica (Medical Logic Modules, MLMs) y, lo que es más importante, separa la lógica médica que define la guía (codificada con la sintaxis de Arden), de los componentes dependientes de las organizaciones o centros donde se define (se codifican como el lenguaje de términos y preguntas que esté definido en la base de datos local). Aun así, las implementaciones basadas en reglas presentan ciertas desventajas:

- No suelen incluir una manera explícita e intuitiva de representar la lógica médica general de la guía.
- No hacen distinciones semánticas acerca de los diferentes tipos de conocimiento clínico representado.
- Carecen de la habilidad para representar y reutilizar de manera sencilla las guías y sus componentes.
- No son compatibles con la aplicación de guías a lo largo de extensos periodos de tiempo, como por ejemplo, cuando es necesario cuidar de los pacientes crónicos.

A lo largo de los últimos 20 años, han existido varios intentos para crear complejos sistemas de cuidado basados en guías clínicas, formadas por planes bien definidos y que funcionasen a lo largo de extensos periodos de tiempo de manera automática. Ejemplos de estas arquitecturas y lenguajes de representación son: EON (15), lenguaje que actualmente está abandonado por falta de fondos, aunque SAGE (16) continúa con parte de este proyecto, Asbru (17) integrado dentro del proyecto Asgaard (18), PRO*forma* (19), GLIF (20) que en su versión más actual es conocido como GLIF3 (21), GUIDE (22), HELEN (23), GLARE (24) y Prodigy (25).

Casi todo los lenguajes definidos para ser utilizados dentro de las guías clínicas electrónicas y en particular los que basan su arquitectura en la definición de planes, pueden ser descritos como preceptivos, lo que quiere decir que especifican las acciones que necesitan ser realizadas y cómo se deben realizar éstas acciones. Sin embargo, algunos sistemas como por ejemplo Asbru, usan además de una metodología preceptiva para especificar o prescribir intervenciones, un enfoque crítico, en la cual el médico sugiere un tratamiento específico y recibe respuestas por parte de la aplicación. Asbru en este caso usa el enfoque crítico para poder evaluar la calidad del tratamiento de manera retrospectiva.

Por otro lado, tenemos el modelo de elementos de guías (*Guideline Elements Modeling, GEM*) (26) cuyo esquema más actual es GEM II (27) un estándar aprobado por la sociedad americana

[6] http://www.hl7.org/

de materiales y pruebas (*American Society for Testing and Materials,* ASTM)[7]. Dicho modelado permite estructurar un documento de texto que contenga una guía clínica como un documento que contenga un lenguaje de marcas extensible (*Extensible Markup Language, XML*), usando un esquema XML bien definido. Sin embargo, GEM, se ejecuta en una máquina independiente y no soporta herramientas que puedan interpretar el resultado del texto semiestructurado que genera como resultado, ya que no incluye un lenguaje formal que provea de un modelo computacional claro.

2.4.PRO*forma*

PRO*forma* es un lenguaje de representación formal del conocimiento para el desarrollo y ejecución de guías clínicas y permite capturar el contenido y la estructura de las guías clínicas en una forma que puede ser interpretada por un ordenador. Este lenguaje crea las bases de un método y una tecnología para el desarrollo y la publicación de guías clínicas ejecutables. Las aplicaciones que se construyen usando PRO*forma* están diseñadas para ayudar en la gestión de procedimientos médicos y en las decisiones clínicas aplicadas al tratamiento.

2.4.1. Porqué PRO*forma*

Para la realización de este proyecto se ha elegido PRO*forma*. Este lenguaje ha sido desarrollado por el laboratorio de computación avanzada del departamento de investigación oncológica del Reino Unido (Advanced Computation Laboratory, Cancer Research UK), y crea las bases de un método y una tecnología para el desarrollo y la publicación de guías clínicas ejecutables.

Este lenguaje empezó a desarrollarse en 1992 y actualmente existen dos aplicaciones disponibles que hacen uso de su motor de inferencia y están en continuo desarrollo (Arezzo[8] y Tallis[9]). A continuación se enumeran las razones por las cuales PRO*forma* ha sido elegido como el lenguaje de especificación para ser utilizado en este proyecto:

- Es un lenguaje bien documentado (28) (29) (30). Se ofrecen su sintaxis y su semántica de manera gratuita de forma que si se necesitase un motor de ejecución, éste podría ser construido siguiendo las reglas definidas del lenguaje
- Tallis ofrece en su página web, una versión del motor de ejecución de PRO*forma*, de esta forma se ofrece a los investigadores y estudiantes que quieran utilizarlo una manera sencilla de ejecutar guías clínicas definidas en el lenguaje de PRO*forma*. Además el motor está definido en Java, que es el mismo lenguaje usado para la implementación de la aplicación
- PRO*forma* puede ser ejecutado a través de la web o en una máquina independiente
- Actualmente existen aplicaciones clínicas en uso y en desarrollo (como se en el punto 2.4.2.1 de esta memoria), lo cual quiere decir que es un lenguaje que no va a ser abandonado a corto plazo

Como se puede observar, que Tallis ofreciese de manera libre su motor de ejecución decantó la balanza hacia PRO*forma*, ya que el trabajo de desarrollar un motor de ejecución que fuese

[7] http://www.astm.org/
[8] http://www.infermed.com/index.php/arezzo
[9] http://www.cossac.org/tallis

capaz de analizar la sintaxis y la semántica de las guías clínicas, su validación y su ejecución habría llevado un tiempo y unos recursos que no estaban disponibles para la realización de este proyecto. Además durante la elección del lenguaje, se estableció contacto con otros grupos de investigación que también estaban desarrollando en PRO*forma*, abriendo así un camino a posibles colaboraciones.

2.4.2. Trasfondo

2.4.2.1. El lenguaje PROforma y métodos

PRO*forma* ha sido utilizado para desarrollar varias aplicaciones, entre las que se incluyen:

- **CAPSULE:** desarrollada para ayudar a los médicos generalistas en la prescripción de medicamentos para enfermedades comunes (31).
- **RAGs:** un sistema de evaluación de riesgos en genética cancerígena (32).
- **LISA:** sistema de ayuda a la decisión incluido en una base de datos clínica para ayudar a los médicos a cumplir con las dosis para un protocolo de pruebas para niños con leucemia linfoblástica aguda (33).

En PRO*forma* la guía clínica se modela como un conjunto de datos y tareas. La noción de una tarea es central, es decir, como se puede ver en la Figura 1, el modelo de tarea de PRO*forma* divide una **tarea** (componente ***task***) genérica en cuatro tipos distintos de tareas: planes, decisiones, acciones y consultas, que serán los utilizados posteriormente para definir la guía.

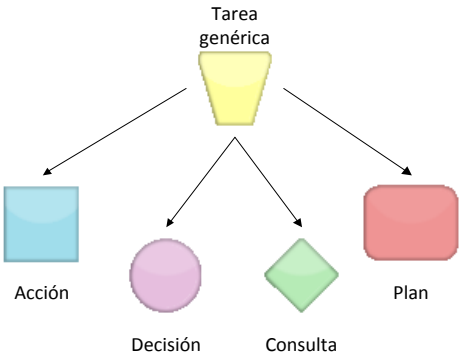

Figura 1: El modelo de tarea en PRO*forma*

Los **planes** (componente ***plan***) son los bloques de construcción básica de la guía y pueden contener cualquier número de tareas de cualquier tipo, incluyendo otros planes, las tareas que contienen están agrupadas por una razón, ya sea porque comparten una meta común, usan algún recurso en común o necesitan ser ejecutadas al mismo tiempo. Las **decisiones (**componente ***decision*****)** son tomadas en puntos de la guía donde se presentan opciones, por ejemplo, si empezar a aplicar un tratamiento a un paciente o seguir con las pruebas médicas. Las **acciones** (componente ***action***) son procedimientos clínicos típicos (como administrar una inyección) que necesitan ser realizados. Las **consultas** (componente ***enquiry***) son peticiones de información extra o de datos que son requeridos por la guía clínica antes de poder continuar.

Los procesos de PRO*forma* pueden ser representados de manera esquemática como grafos dirigidos en los que los nodos representan las tareas y los arcos representan restricciones en la planificación. Cada guía clínica contiene como mínimo un plan raíz único que puede contener de manera recursiva a otros planes.

2.4.2.2. Propiedades de las tareas y los datos

Cada tarea tiene un número de propiedades cuyos valores indican cómo tiene que ser interpretada. El valor de una propiedad puede ser un valor escalar, una expresión o incluso un objeto el cual, a su vez, puede tener propiedades. A continuación se van a describir las propiedades principales que tiene estos elementos.

Las propiedades compartidas tanto por los datos como por las tareas son:

- **Título y descripción:** PRO*forma* está definido de tal manera que cualquier elemento que pueda tener cualquier tipo de propiedad puede tener un espacio de texto para un nombre y una descripción, los cuales se pueden usar para escribir comentarios como, por ejemplo, la explicación del propósito de la tarea

Todas las **tareas** comparten las siguientes propiedades:

- **Precondición:** es una expresión que puede ser evaluada como verdadera o falsa y que debe ser cumplida para que la tarea pueda comenzar
- **Restricciones:** son restricciones lógicas que impiden a una tarea empezar antes de que otra tarea o conjunto de tareas haya terminado

Las subclases de las tareas también tienen propiedades distintivas, las **decisiones** tienen las siguientes propiedades:

- **Candidatos:** son objetos que representan las opciones que hay que tener en cuenta cuando se toma una decisión. Los candidatos también tienen propiedades por su cuenta, incluyendo reglas de recomendación, que es una expresión de PRO*forma* usada para reflejar las condiciones bajo las cuales sería aconsejable elegir al candidato
- **Argumentos:** son objetos que representan los argumentos que existen a favor, en contra o que son relevantes para un candidato en particular. Un argumento consiste en una expresión que se evaluará a verdadero o falso y en un nombre y una descripción que explicará la expresión

Además de las decisiones, los planes también tienen una propiedad exclusiva:

- **Condiciones de cancelación o terminación:** son expresiones lógicas que representan de manera suficiente, pero no necesaria, las condiciones en las cuales un plan puede acabar de manera satisfactoria y continuar el resto de tareas que puedan quedar a la espera o las condiciones en las cuales un plan necesita ser cancelado y con ello todas las tareas que podrían haber sido ejecutadas a continuación

2.4.3. Descripción del sistema

2.4.3.1. La sintaxis de PROforma

La sintaxis de PRO*forma* se define utilizando una notación de Backus Naur (28)(Backus Naur Form, BNF), por lo tanto, las guías que no cumplan con esta construcción del lenguaje no pueden ser validadas. La sintaxis de PRO*forma* puede ser dividida en dos partes, la de las expresiones, que será la encargada de definir qué condiciones lógicas y que expresiones

matemáticas se aceptan y la que define como los componentes de la guía deben ser organizados y separados. Esta especificación además define un algoritmo de inferencia para los tipos de las expresiones e impone restricciones de tipo en las expresiones que son valores de propiedades, por ejemplo, el valor de la propiedad precondición debe ser siempre una expresión lógica.

2.4.3.2. *La semántica de PROforma*

La semántica de PRO*forma* trata las guías como si fuesen un conjunto de objetos a los que se refiere como componentes de PRO*forma*. Las distintos tipos de clases que pueden ser usadas para instanciar estos objetos están organizados en una jerarquía que se puede observar en la Figura 2.

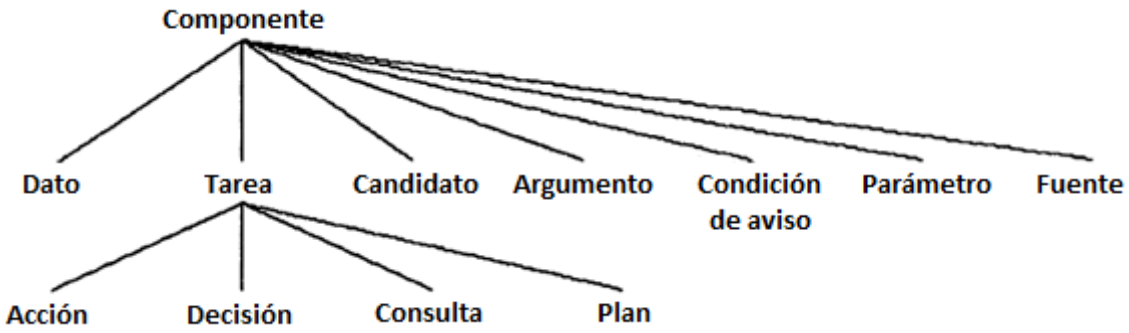

Figura 2: Conjunto de componentes de PROforma

La mayor parte los componentes que se pueden ver en la Figura 2 han sido descritos con anterioridad, excepto la condición de aviso (*Warning condition*, que describe condiciones que deberían ser comprobadas cuando un valor es proporcionado para un dato), el parámetro (*Parameter*, que son utilizados para almacenar datos específicos de una instancia en particular de una tarea) y las fuentes (*Sources*, que son utilizadas para describir la información que debería ser recogida por una tarea del tipo petición).

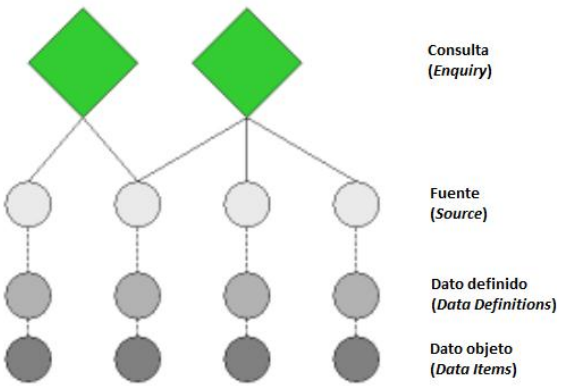

Figura 3: Relación de los datos en PROforma

Como se puede ver en la Figura 3, una consulta (*enquiry*) puede tener varios datos requeridos, cada uno de ellos conocido como fuente (*source*). Cada fuente está basada en un dato definido en la guía (*data definition*), el cual determina la estructura del dato que la consulta está requiriendo. La clave para diferenciar entre una fuente y un dato es que la fuente solo existe dentro del contexto de una consulta y por lo tanto tiene un atributo que representa su estado. El estado de la fuente determina si es obligatorio rellenar el dato para que una consulta pueda ser completada. Por defecto su estado es obligatorio, es decir, que se tiene que proporcionar un valor para el dato requerido. Si el valor de una fuente es opcional, la consulta que lo contenga puede ser completada sin que ese valor sea rellenado. Cuando un valor es recogido durante la ejecución de la guía, se

crea una instancia del dato que lo contenga. Esa instancia será conocida como el dato objeto (*data item*).

Como se ha mencionado antes, cada clase tiene cierta cantidad de propiedades, y además cada una heredará esas propiedades de sus superclases, por lo que, una instancia de una clase tendrá valores en las propiedades descritas en esa clase incluyendo aquellas heredadas por su superclase. Por ejemplo, cuando añadimos una instancia de la clase Tarea a una guía, debemos especificar qué valores tiene esa instancia para cada una de las propiedades de la clase Tarea, incluyendo aquellas propiedades que hereda de la clase Componente, como por ejemplo, el nombre y la descripción. Además cada instancia tendrá por su parte, una identificador de componente que será único para esa instancia, es decir, dos instancias no podrán tener el mismo identificador incluso si son instancias de distintas clases.

Los valores que pueden tener las propiedades tienen que ser valores admitidos por PRO*forma*, lo cual quiere decir que deben ser algo de los siguientes tipos:

- Un número, ya sea real o entero
- Una cadena de texto
- Una expresión de PRO*forma*
- Un identificador de componente
- Alguna de las constantes definidas por PRO*forma*, como por ejemplo *true*, *false, dormant, in_progress, discarded* o *completed*.
- Una secuencia finita de los valores definidos por PRO*forma*. Para definir esta lista de valores se usarán los símbolos <>, por ejemplo, para una lista de enteros <1, 2, 3, 4>.

2.4.3.2.1. Estados posibles de las tareas

Todas las tareas tienen la propiedad conocida como estado, la cual puede tomar cuatro valores diferentes: dormida (*dormant*), en progreso (*in_progress*), descartada (*discarded*) o completada (*completed*). Inicialmente todas las tareas se encuentran en el estado dormido. La semántica de PRO*forma* no impone ninguna interpretación a estos estados, pero se podría decir que una tarea está dormida si todavía no ha empezado y no es posible decir cuándo va a empezar, estará en progreso si se ha iniciado, descartada si la lógica de la guía implica que esa tarea no debería ser empezada o completada y el estado de una tarea será completado cuando dicha tarea haya finalizado.

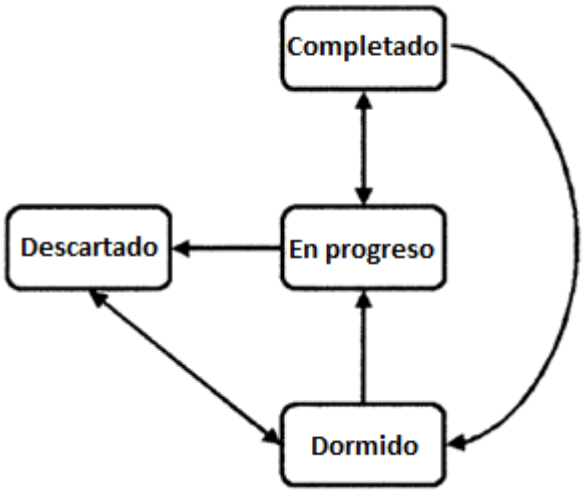

Figura 4: Transiciones entre los estados de las tareas

La Figura 4 muestra las transiciones permitidas entre los estados de las tareas. La razón por la cual existen transiciones que salen de los estados completado y descartado es porque las tareas pueden ser cíclicas, lo cual quiere decir, que se pueden ejecutar varias veces a lo largo de un mismo plan. La transición de completado a en progreso ocurre cuando la propia tareas

es cíclica y la transición de completado o descartado a dormido ocurre cuando es el plan el que es cíclico.

2.4.3.3. El motor abstracto de PROforma

La semántica de PRO*forma* está definida en términos de un motor abstracto y dicho motor es responsable de la ejecución de al menos una guía. El estado de este motor abstracto está definido por las cuatro variables descritas a continuación:

- **La tabla de propiedades:** es una tabla de tres columnas que contiene los valores actuales de todas las propiedades de todos los componentes de la guía. Cada fila de la tabla tiene la forma (C, P, V), donde C es un identificador único para un componente de la guía, P es el identificador de la propiedad que pertenece a dicho componente y V es el valor actual de esa propiedad. La tabla de propiedades no puede contener dos filas con los mismos valores en C y P
- **La tabla de cambios:** es una tabla que contiene los nuevos valores que tienen que ser asignados a las propiedades de los componentes como resultado de ciertas operaciones que han sido realizadas en la guía. Las filas de la tabla de cambios tienen la misma estructura que la de la tabla de propiedades, es decir (C, P, V), pero es esta tabla pueden existir varias filas con los mismos valores en C y P
- **El indicador de excepción lógica:** será verdadero si ha ocurrido algún evento anormal durante el procesado de una operación en la guía, y será falso en caso contrario. Antes de enviar los cambios a la tabla de propiedades desde la tabla de cambios, se consulta este valor y se avisa al usuario si hay algún problema.
- **El tiempo del motor:** esta variable, que será un número real, solo podrá ser modificada por el propio motor. La semántica definida no requiere que el tiempo del motor corresponda con ninguna medida de tiempo que tenga equivalencia en el mundo real, aun así, esta variable tomará como valor el número de milisegundos que han pasado desde la medianoche del 1 de enero de 1970

3. Desarrollo web

3.1. Opciones disponibles

Para la realización de este proyecto se ha optado por el diseño de una aplicación accesible al usuario a través de la web. Debido a la complejidad del proyecto afrontado, se quiso elegir un entorno de programación que facilitara el trabajo.

En la Figura 5 se puede observar una tabla que compara los entornos (*frameworks*) más utilizados actualmente el desarrollo web:

Feature	Ext GWT 2.0	Flex 3	GWT 1.7	ICE Faces 1.8	JQuery 1.3.2	Smart GWT 1.3	Vaadin 6.1	Wicket 1.4
Widget diversity & richness	✶✶✶	✶✶✶	✶✶	✶✶	✶✶	✶✶✶	✶✶✶	✶
No browser plug-in required	✔		✔	✔	✔	✔	✔	✔
No JavaScript programming required	✔		✔	✔		✔	✔	✔
Framework extensions are done in Java	✔		✔			✔	✔	
No HTML required	✔	✔	✔			✔	✔	
No XML configuration required	✔		✔		✔	✔	✔	✔
Only server-side implementation required				✔			✔	✔
User interface logic is kept securely on server				✔			✔	✔
Web-page oriented					✔			✔
Multimedia oriented		✔						
Backed up by a corporation	✔	✔	✔	✔		✔	✔	
Commercial support & guarantees available	✔	✔		✔		✔	✔	
Vendor offers experts services to complement the framework				✔		✔	✔	
Vendor offers add-on components to complement the framework		✔				✔	✔	
Free for commercial use		✔	✔	✔	✔	✔	✔	✔
License	GPL v3/Commercial	MPL 1.1	Apache 2.0	MPL 1.1	MIT/GPL v2	LGPL	Apache 2.0	Apache 2.0

Figura 5: Aplicaciones de internet enriquecidas disponibles (RIA, Rich Internet Applications)

Para poder elegir un entorno de entre los disponibles, éstos tendrán que cumplir ciertas condiciones:

- La licencia debe permitir que la aplicación desarrollada utilizando dicho *framework* pueda ser distribuida sin tener que distribuir el código fuente con ella y si se decidiese comercializar la aplicación, dicha licencia debería poder permitirlo de manera gratuita
- A ser posible, se tiene que poder desarrollar exclusivamente en Java, tanto la aplicación en sí como las posibles extensiones que se diseñen. Esto es debido a que tanto el motor de inferencia de PRO*forma*, como la lógica creada después se han programado en Java y de esta manera se simplificaría el trabajo de desarrollo
- Dado que se tratarán con datos médicos, tanto la parte del servidor como la parte del cliente deben poder ofrecer seguridad de base, aunque posteriormente se puedan reforzar estos métodos

3.2. Vaadin

Vaadin es un entorno de trabajo para el desarrollo de aplicaciones web en Java y está diseñado para crear aplicaciones interactivas que funcionen sobre el navegador sin necesitar ningún tipo de añadido, es decir, no requiere instalación en el navegador. Vaadin usa una arquitectura orientada al servidor junto con un modelo de componentes reutilizables para simplificar la

programación de las aplicaciones y para ofrecer una mayor seguridad de cara a la aplicación web.

3.2.1. Porqué Vaadin

Una vez contemplados los requisitos que debe cumplir el entorno de programación seleccionado y si volvemos a consultar la Figura 5, podemos ver que Vaadin es la mejor opción disponible que cumplen estos requisitos, ya que aunque GWT y SmartGWT también podrían haber sido seleccionados, la propiedad de Vaadin de no necesitar implementación en el cliente, le da un extra de seguridad, ya que toda la lógica se ejecutará en el servidor y podrá estar mejor protegida. Frente a Wicket, Vaadin permite que toda la aplicación esté escrita en Java, incluso las extensiones de dicho entorno, lo cual le conferirá al código de dicha aplicación más estabilidad y legibilidad.

Además Vaadin está diseñado para construir aplicaciones web, no solo páginas web. La forma de programar se parece mucho a la forma en que se programaría una aplicación para el escritorio y se quería mantener la apariencia de una aplicación de escritorio para no confundir excesivamente a los usuarios ya acostumbrados a manejar otro tipo de aplicaciones de estilo similar.

Vaadin tiene una licencia de desarrollo Apache 2.0 lo cual permite al usuario del software desarrollado la libertad de usarlo para cualquier propósito, distribuirlo, modificarlo, y distribuir versiones modificadas de ese software. Esta licencia no exige que las obras derivadas (versiones modificadas) del software se distribuyan usando la misma licencia, ni siquiera que se tengan que distribuir como software libre u *open source*. La Licencia Apache sólo exige que se mantenga una noticia que informe a los receptores que en la distribución se ha usado código con la Licencia Apache.

También permite, si fuese necesario, el uso de *Google Web Toolkit* además de *JavaScript*, *HTML*, *XML* y *CSS* para el desarrollo de componentes en la parte del cliente, los cuales ofrecen mucha más potencia a este entorno y permiten personalizar más aún las aplicaciones que van a ser desarrolladas.

En las siguientes páginas se pasa a explicar los beneficios y los problemas que acarreó la elección de Vaadin como entorno de trabajo para el desarrollo de este proyecto.

3.2.2. Beneficios

3.2.2.1. Ejecución en el servidor

Ejecutar la interfaz de usuario en el servidor puede ser controvertido, pero tiene tremendos beneficios:

- **Se puede usar la propiedad de java "Reflection":** mientras que en GWT no se puede utilizar, en el código ejecutado en el servidor se puede. Esto es muy útil, por ejemplo, cuando quieres instanciar clases sin declarar su nombre explícitamente, en vez de eso puedes usar un archivo de propiedades que puede cambiar lo largo de la ejecución del programa, lo cual confiere mucha flexibilidad al código

- **Se respetan los parámetros de seguridad:** en cualquier momento se pueden usar listas de control de acceso (*Access control list*, ACLs) para definir lo que es y no es posible en la interfaz
- **Menos servicios web:** La interfaz de usuario se construye y ejecuta en el servidor, pero es dibujada en el cliente, por lo cual no es necesario crear servicios adicionales para recibir diálogos y descriptores
- **Agilidad:** Con GWT es difícil mantener la agilidad, ya que no permite cambiar el código final sin tener que compilarlo a JavaScript de nuevo. Vaadin permite combinar y utilizar cualquiera de sus componentes ya compilados sin necesidad de tener que volver a generar el código JavaScript
- **Testeo:** todo el código puede ser testeado con test jUnit normales, incluso la interfaz de usuario

3.2.2.2. Eficiencia

Como este proyecto ha sido creado dentro de un entorno de trabajo con tiempo y recursos limitados, la eficiencia al desarrollar es muy importante, y Vaadin ofrece:

- **Documentación:** La documentación de Vaadin (34) es muy completa y de calidad, el libro de Vaadin (35), el cual se puede descargar de manera gratuita, simplifica mucho el comienzo del desarrollo
- **Compilación:** no se necesita recompilar el código que vayamos implementando en el servidor a menos que se añada algún componente nuevo a la parte del cliente, lo cual ocurre en contadas ocasiones
- **Estilos:** como muchas otras funcionalidades, los estilos son servicios proporcionados por Vaadin. Además una vez decidido uno distinto a los proporcionados por defecto, es muy fácil implementarlo
- **Colaboración de la comunidad:** además de las funcionalidades proporcionadas, también se ofrece en la página web de Vaadin una recopilación de los componentes (*Addons*) creados por la comunidad desarrolladora de este entorno de trabajo. Esto generalmente simplifica mucho el trabajo, ya que las necesidades que puedan surgir debido a la falta de algún componente en Vaadin quedan cubiertas por estos añadidos

3.2.3. Problemas de rendimiento

En esta sección se explica por qué los posibles defectos o problemas de rendimiento que en un principio se pueden plantear a Vaadin no están justificados, o por lo menos, por qué no neutralizan los beneficios.

3.2.3.1. Programa a desarrollar

En este proyecto se va a desarrollar un generador automático de interfaces gráficas que tiene como finalidad principal, ser utilizado por un reducido grupo de personas y ser ejecutado generalmente en intranets, lo cual implica un bajo número de peticiones al servidor y una gran conexión (ancho de banda) entre la máquina del cliente y el servidor. Aun así, una vez finalizada la aplicación y para comprobar la escalabilidad del proyecto y el consumo en memoria de éste en el servidor, se ejecutarán una batería de pruebas llevadas a cabo con

Jmeter[10], que es una aplicación de software libre diseñada para medir la eficacia de las aplicaciones web y el rendimiento de éstas.

3.2.3.2. Número de peticiones

Uno de los mayores problemas que se plantean cuando se utiliza Vaadin es que gestionar los eventos en la parte del servidor llevará a más peticiones y añadirá el retardo de la red a la mayor parte de las interacciones del usuario. Este problema puede echar para atrás la elección de Vaadin como entorno de trabajo, pero antes de rechazarlo por completo estudiaremos varios escenarios y la respuesta de Vaadin en comparación con GWT.

Escenario 1) Pregunta por datos (GWT) = enviar evento (Vaadin)

Casi todas las interacciones del usuario acabarán con una petición al servidor. La única diferencia es la razón para enviarla. Mientras que GWT preguntará por los datos para mostrarlos o enviarlos a otro lado, Vaadin enviará información sobre un evento. Algunos ejemplos de interacciones del usuario con la interfaz son:

- Crear una nueva página
- Abrir un diálogo
- Pulsar el botón de guardado

Escenario 2) Una petición para dominarlas a todas

Hay que tener en cuenta que cualquier interfaz de usuario que se ejecute en la parte del cliente en algún momento requerirá datos del servidor. Por lo que es probable que una aplicación de este estilo tenga que enviar más peticiones para reunir los datos que necesita que una aplicación que se ejecuta en el servidor.

Un caso de uso típico es la apertura de un cuadro de diálogo para editar texto que necesita información adicional para rellenar una tabla o para leer información de la página a la que estamos enlazando. Vamos a ver cómo se comporta GWT y Vaadin en este caso:

GWT (2 – 3 peticiones):

1. Evento: ejecución de la acción de apertura del cuadro de diálogo
2. Preguntar al servidor por el cuadro de diálogo a abrir y por su descriptor
3. Rellenar el cuadro de diálogo: petición para recoger los datos requeridos
4. Si necesitamos información adicional para rellenar la tabla: petición al servidor de los datos

Vaadin (1 petición):

1. Evento: petición inmediata al servidor
2. El manejador de eventos crea el cuadro de diálogo (es muy rápido ya que no incluye la parte de dibujado en el cliente)
3. Los datos requeridos son leídos. Leer información adicional como por ejemplo metadatos es inmediato
4. Se enlazan los datos con el cuadro de diálogo

[10] http://jakarta.apache.org/jmeter/

5. Vaadin envía al cliente los cambios en la interfaz a través de JSON (*JavaScript Object Notation*)
6. El cliente aplica los cambios en su interfaz

En este escenario se puede ver claramente como nos ahorramos hacer peticiones de más al servidor gracias al protocolo de comunicación de Vaadin.

Escenario C) Cambios en la visibilidad.

A pesar de funcionar principalmente gracias al manejo de los eventos entre el cliente y el servidor, Vaadin también ofrece la posibilidad de gestionar exclusivamente los eventos del cliente. En algunos casos de uso, como por ejemplo hacer *click* sobre una pestaña o navegar por los menús, ni GWT ni Vaadin tienen la necesidad de enviar eventos al servidor. En ambos casos es el cliente el que gestiona la visibilidad de los elementos.

Conclusión a los posibles problemas de rendimiento

Dado que Vaadin se ejecuta en el servidor, se puede esperar un aumento de las peticiones entre éste y el cliente, pero como se ha podido ver en los casos anteriores, ni si quiera llegarían a duplicar el número de peticiones e incluso en algunos casos mejoraría, por lo que se puede decir que el modelo adoptado por Vaadin para comunicarse entre cliente y servidor no supone un descenso en la eficiencia de la web que lo implemente.

3.2.4. *Addons* usados en Vaadin

Como se ha comentado anteriormente, el entorno de trabajo de Vaadin permite la inclusión de *addons* diseñados por la comunidad de desarrolladores. Éstos simplifican ampliamente el trabajo de la aplicación final y ayudan a Vaadin a dar un aspecto más pulido.

A continuación se describirán los *addons* usados en este proyecto y su utilidad:

- **easyuploads-0.4.9.jar:** librería encargada de gestionar la carga de archivos en el servidor. El componente *Upload* de Vaadin ya implementa estas funciones, pero las funciones que usa son de muy bajo nivel y a veces es difícil utilizarlo para tareas simples. Esta librería contiene funciones que simplifican la implementación de este servicio y además provee de funciones que permiten la carga múltiple de archivos, lo cual reduce el tiempo de subida de varios archivos al servidor
- **gwt-graphics-1.0.0.jar:** librería gráfica de vectores que provee de soporte para múltiples navegadores. Para poder ofrecer este soporte, la librería usa lenguajes basados en XML: SVG y VML. La implementación en VML se utilizará para Internet Explorer y SVG para el resto de navegadores. Este *addon* ha hecho posible que la representación del grafo se vea de igual manera en todos los navegadores. Aunque internamente el *addon* utilice las librerías de GWT, Vaadin lo utiliza ya compilado y a través del sistema de gestión de mensajes comentado anteriormente
- **listbuilder-0.6.0.jar:** librería que añade funcionalidad al componente de Vaadin llamado selección en columnas gemelas (*TwinColSelect*). Además de arreglar varios *bugs* existentes, permite añadir estilos y mantiene el orden de los elementos seleccionados

- **vaadin-treetable-1.2.2.jar:** es una librería que extiende la funcionalidad del componente tabla de Vaadin, permitiendo que los elementos que componen dicha tabla estén relacionados entre sí como si de un árbol se tratara.

4. Implementación del proyecto

Este proyecto ha sido desarrollado utilizando el entorno de trabajo Eclipse[11] y el entorno de desarrollo Vaadin. La aplicación se ha desarrollado íntegramente en Java haciendo uso de las librerías estándar de este lenguaje así como basándose en patrones de diseño software (36).

4.1.Esquema de ejecución de Vaadin

Para comprender como ha sido desarrollada la aplicación, primeramente se procederá a explicar cómo está relacionada con su entorno. En la Figura 6, se puede ver la arquitectura que tendrán todas las aplicaciones desarrolladas con este *framework*.

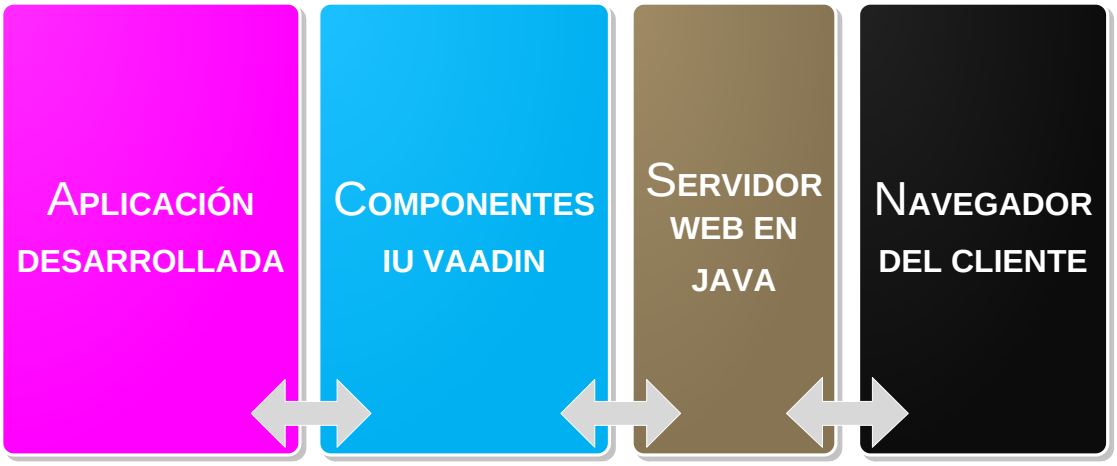

Figura 6: Arquitectura cliente-servidor de Vaadin

La aplicación implementada se encontrará entre los diferentes servicios de conexión con los que se quiera relacionar (p. ej. Servicios web, bases de datos, *Enterprise Java Beans* EJBs) y los componentes definidos por Vaadin para crear las interfaces gráficas. Esto permitirá, como más adelante se explica, conectar la aplicación con diversas fuentes de datos, lo cual nos permite dotar de nuevos recursos a las guías clínicas en ejecución.

Todo esto será ejecutado en la parte del servidor, mientras que la interacción con el usuario será manejada por el motor que se ejecutará en el navegador del cliente. Las comunicaciones cliente-servidor y cualquier tecnología que implique ser utilizada en la parte del cliente son completamente invisibles a la hora de desarrollar la aplicación. Como el motor en el lado del cliente se ejecuta como JavaScript en el navegador, no existe la necesidad de instalar nada en dicho navegador.

[11] http://www.eclipse.org/

4.2.Esquema general de la aplicación

En la Figura 7**¡Error! No se encuentra el origen de la referencia.** se puede ver la estructura que poseen todas las aplicaciones desarrolladas usando el entorno de Vaadin. Como se puede observar, explica con más detalle la arquitectura presentada en la Figura 6.

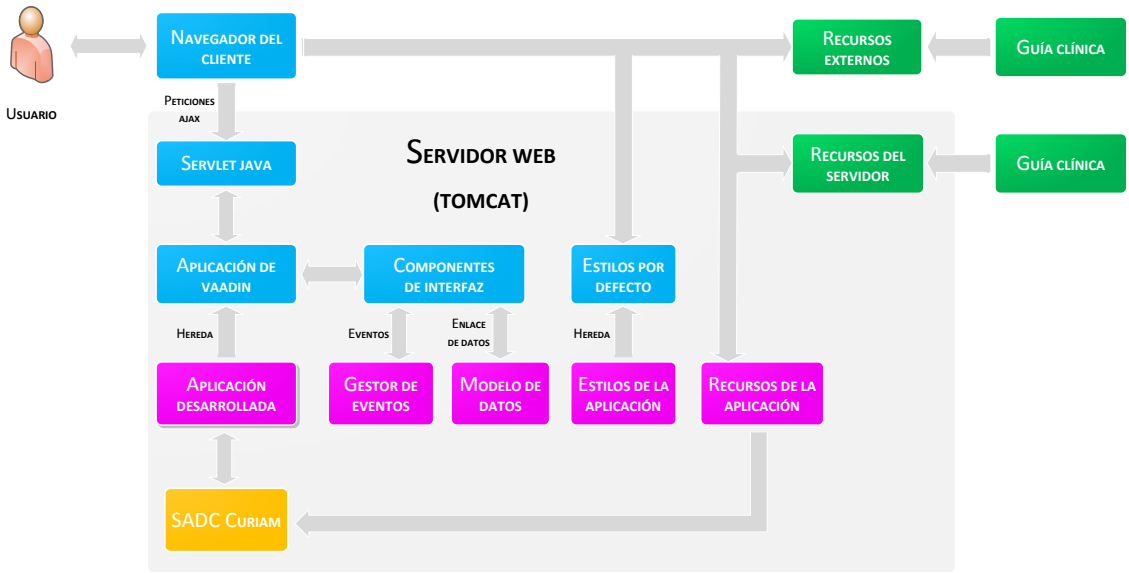

Figura 7: Arquitectura de la aplicación desarrollada

El esquema de colores ayuda a comprender como está estructurada la arquitectura interna de la aplicación. El color azul muestra los componentes que son ofrecidos por el entorno de desarrollo Vaadin y que son invisibles durante la implementación de la aplicación. El color verde muestra los recursos externos a nuestra aplicación, ya sean recursos accedidos a través de otros servicios o conexiones (p. ej. Bases de datos, servicios web u otro tipo de recursos online), o recursos que se pueden encontrar dentro del propio servidor (p. ej. Ficheros). El color magenta indica lo que ha sido desarrollado de manera particular para esta aplicación, todos las aplicaciones desarrolladas usando Vaadin, están obligadas a heredar la clase aplicación (mostrada en este esquema como un rectángulo verde del que hereda Aplicación de usuario), ya que en esta clase se define el punto de entrada al programa, que aquí se puede ver en el rectángulo magenta marcado como aplicación desarrollada. El color naranja representa los sistemas utilizados que se utilizan sin conocer cómo funcionan internamente, es decir, sólo nos interesa su resultado. En la Figura 7 se muestra la relación del sistema de ayuda a la decisión Curiam con la aplicación y con las guías clínicas ejecutadas. La descripción del sistema de conexión completo y de cómo se creó, se presenta en el punto 6 de esta memoria.

Analizando la Figura 7 podemos deducir que el desarrollo de una aplicación en web con Vaadin acaba convirtiéndose en algo muy parecido al desarrollo de una aplicación de escritorio creada en Java y utilizando las librerías de Swing, ya que el programador solo se tiene que preocupar por la aplicación y no por cómo se conecta ésta con el cliente.

4.3.Estructura del proyecto

Para el desarrollo de este proyecto se utilizó el lenguaje de programación Java. Aprovechando que este es un lenguaje orientado a objetos se hizo uso de una de las propiedades básicas de este tipo de lenguajes, la modularidad. La aplicación ha sido dividida en tres módulos y posteriormente éstos han sido relacionados entre ellos. Los módulos creados consistieron en: primero, crear una forma de enlazar el motor con la aplicación y con el sistema de ayuda a la decisión clínica (SADC). Segundo, crear una interfaz gráfica que permita manejar la ejecución de la guía de manera sencilla y tercero, crear un nuevo componente en Vaadin que permita representar la guía clínica cargada como un grafo dirigido que sea interactivo.

4.3.1. Relación del motor de PRO*forma* con la aplicación y el SADC

Cuando se planteó la relación del motor de PRO*forma* creado por Tallis con la aplicación se llegó a la conclusión de que el diseño creado debía ser capaz de admitir un cambio en el motor de ejecución de la guía clínica sin que esto afectase al resto del programa.

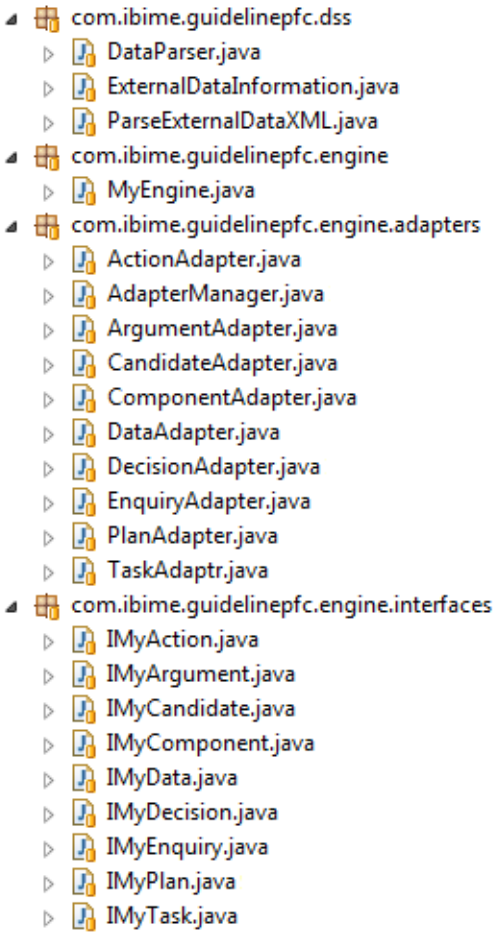

Figura 8: Conjunto de clases creadas para la gestión del motor de PROforma

Para conseguir eso, y como se puede ver en la Figura 8 se han creado una serie de interfaces y adaptadores que nos permitirán gestionar las funciones del motor de Tallis sin tener que utilizar sus propias llamadas en nuestro código. El patrón de diseño utilizado se conoce como *Adapter*. Este patrón ofrece a través de la creación de una o varias interfaces, la posibilidad de encapsular el código que vaya a cambiar dentro de los adaptadores, de esta manera, para poder relacionar el resto de la aplicación con el motor se crearán llamadas a dichas interfaces sin importarnos lo que se encuentre por debajo, es decir en la clase adaptadora.

Se ha creado una interfaz y un adaptador por cada componente del motor que necesitaba ser gestionado por la aplicación. Esto se ha hecho así para gestionar mejor las posibles actualizaciones del motor de PRO*forma*, pero también quiere decir que si en un futuro se cambiase el motor subyacente este debería, como mínimo, tener una estructura interna que fuese capaz de representar la guía clínica utilizando dichos componentes.

Además se ha creado, en la clase *AdapterManager.java* un gestor de dichos adaptadores, lo cual nos permite crear cualquier tipo de adaptador a través de la llamada a la función *getAdapter (PFComponent component)*. Este gestor de adaptadores recibirá un componente

del motor definido por Tallis y nos devolverá un componente ya adaptado, guardándose en una estructura interna la interfaz creada, por si fuese requerida posteriormente en la aplicación. Debido a esto, antes de poder usar cualquier componente en la aplicación se tiene que llamar a esta función para que nos devuelva el componente adaptado, por lo que la clase *AdapterManger* implementa además una versión modificada para aplicaciones web, del patrón de diseño *Singleton* (como se verá en el apartado 4.4).

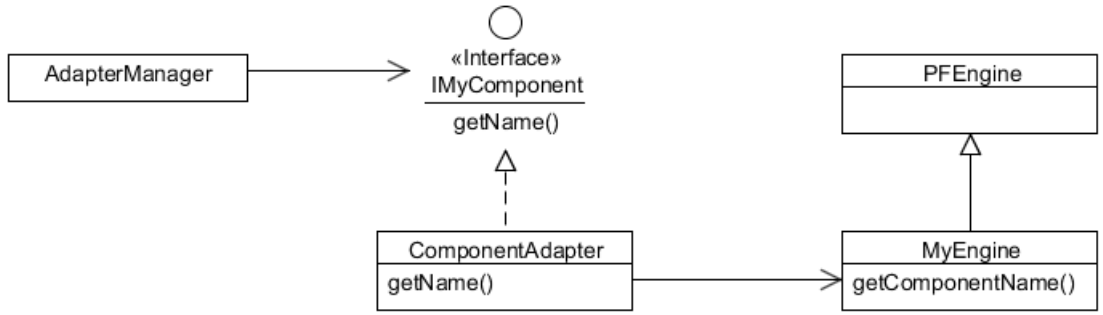

Figura 9: Estructura del patrón de diseño Adapter, utilizado para un componente genérico

En la Figura 9 se puede ver una muestra de la implementación realizada para la conexión del motor con la aplicación. El resto de componentes se llamarán de igual manera y adaptarán sus diversas funciones al programa. Como se puede observar en la figura anterior, se ha creado además una clase llamada *MyEngine*, esto fue debido a que para gestionar los datos externos, el motor proporcionado por Tallis que se encuentra dentro de la clase *PFEngine* no era suficiente y hubo que extender su funcionalidad.

En la Figura 9 se presenta un paquete de datos llamado *com.ibime.guidelinepfc.dss*, dentro de ese paquete se encuentran las clases encargadas de gestionar los datos externos y de relacionarlos con el motor en tiempo de ejecución. La clase *DataParser* se llama nada más cargar la guía clínica en el motor y antes de empezar la ejecución de ésta. Esta clase recibe en su constructor el nombre del archivo XML que tendrá las referencias a los datos externos, si este archivo no existiese, quiere decir que la guía correspondiente no necesita datos que estén relacionados con el sistema de ayuda a la decisión clínica. Si el archivo existe, se crea una estructura de datos definida dentro de la clase *ExternalDataInformation* que será consultada cuando el dato externo sea requerido.

El motor a lo largo de la ejecución de la guía, irá comprobando si los datos que necesita se encuentran o no definidos en dicha estructura y cuando sean requeridos averiguará su valor a través del método *getDSSResult ()* que creará una nueva instancia del sistema de ayuda y devolverá los datos en formato de XML. Para poder analizar este resultado, se ha creado la clase *ParseExternalDataXML* que a través de su método *getValue ()* analiza el XML dado como resultado y nos devuelve el dato requerido en un formato comprensible para el resto de la aplicación.

Una explicación más detallada del sistema de enlace creado con el motor de Tallis y el SADC, así como una muestra del XML generado para representar los datos que serán requeridos por el motor de manera externa se puede encontrar en el punto 6.2 de esta memoria.

4.3.2. Creación de la interfaz gráfica

La creación de la interfaz gráfica se ha desarrollado (a excepción del grafo), usando los componentes definidos en el entorno de Vaadin. La interfaz consta de siete componentes principales, los cuales son:

- La barra del menú (*BarraMenu.java*)
- La ventana de carga de archivos (*Uploader.java*)
- El árbol que representa la guía (*GuidelineTree.java*)
- El grafo que muestra la guía (tiene un paquete aparte, y será comentado en el siguiente punto de la memoria)
- Las tablas (tabla de datos, tabla de tareas y tabla de decisiones) (*MyTabSheet.java* y el paquete *com.ibime.guidelineofc.ui.tables* completo)
- La ventana de información del nodo (*NodeInformation.java*)
- La ventana con información sobre la guía (*GuideState.java*)

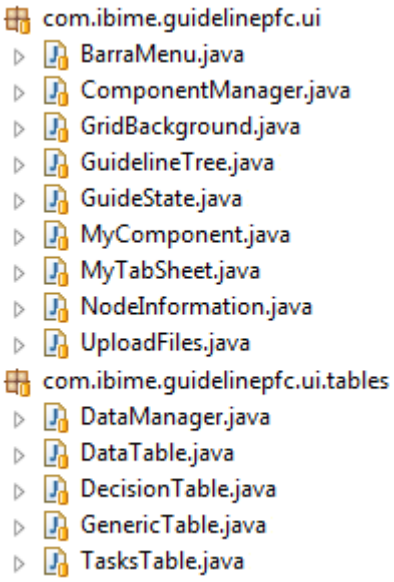

Figura 10: Clases creadas al diseñar la interfaz de usuario

Como muestra la Figura 10, toda la interfaz de usuario está definida dentro del paquete *com.ibime.guidelinepfc.ui*. También se puede observar que existen tres clases más que en principio no pertenecen a la interfaz.

La clase *GridBackground.java* define los paneles que contendrán a los demás componentes de la aplicación, así como su situación dentro del marco del navegador y su estilo de cara al usuario.

La clase *MyComponent.java* define una interfaz que será usada por todos los componentes de la aplicación (excepto el grafo), y que contiene la función *reload ()*. El resto de componentes implementarán esta interfaz y posteriormente definirán la función *reload ()*, siendo el comportamiento de esta función distinto para cada uno de ellos. Como se puede ver en la Figura 11, se está haciendo uso del patrón de diseño *Composite*.

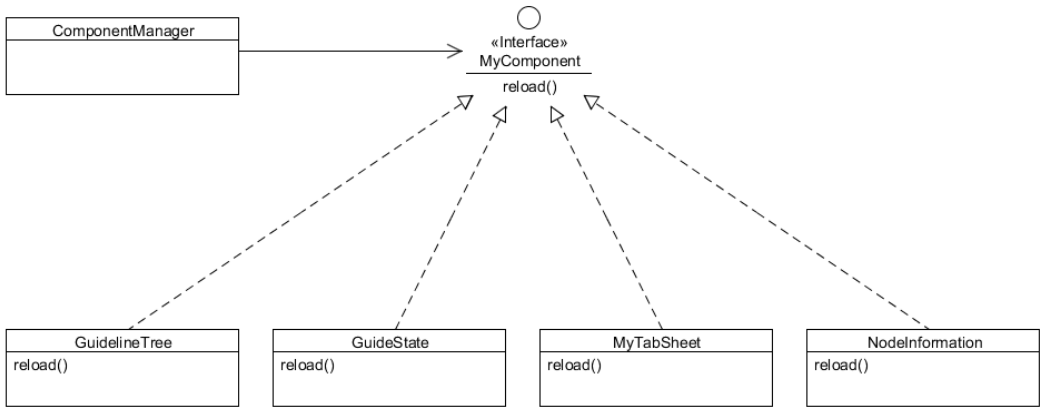

Figura 11: Patrón de diseño Composite utilizado por los componentes de la aplicación

La clase *ComponentManager.java* almacenará los componentes en una estructura de datos y cuando sea necesario actualizarlos, llamará a la función *reload ()* definida en *MyComponent* sin importarle cual sea el comportamiento individual de cada componente.

Además en la clase *ComponentManager*, se usan otros dos tipos de patrones. A nivel de aplicación dicha clase se trata como si fuese un *Singleton* (como se ha comentado anteriormente, esto no es exactamente así, pero se explica más adelante el funcionamiento, en el punto 4.4), y dentro de la propia clase, se utiliza el patrón *FactoryMethod* para crear los distintos componentes que formarán la aplicación con una misma llamada siempre. La clase implementa la función *getComponent (String componentId)*, que nos devolverá, dependiendo de con que identificador se llame, una instancia a uno de los componentes definidos anteriormente.

Como se muestra en la Figura 10, se ha creado un paquete específico para gestionar la creación y el control del flujo de datos hacia y desde las tablas. Este paquete se llama *com.ibime.guidelinepfc.ui.tables*. La clase encargada de gestionar las tablas, su creación y su actualización es *MyTabSheet.java*. Cada una de las tres tablas hereda unas variables y funciones básicas creadas en la clase *GenericTable.java* y que serán comunes a las tres tablas.

Además la clase *DataTable.java* debe mostrar los diferentes tipos de datos a introducir, y como esto es bastante complejo debido a que existen siete tipos distintos de datos (texto, entero, real, booleano, conjunto de texto, conjunto de reales, conjunto de enteros) y en un futuro podría haber más, se ha dejado la gestión de los datos a una clase extra denominada *DataManager.java*. De esta manera desligamos los distintos tipos de datos de la tabla de datos en sí, lo cual nos ofrece un código más fácil de mantener y actualizar de cara al futuro, ya que bastaría con introducir un nuevo conjunto de datos en la clase *DataManager* y automáticamente la clase *DataTable* los aceptaría.

La última clase que se utiliza dentro de este apartado es *UploadFiles.java*, esta clase nos ofrece un gestor y selector de archivos que nos permite cargar en el servidor los archivos que contienen la definición de la guía clínica. Ya que estamos usando una arquitectura general de cliente-servidor, si el cliente necesitase cargar una guía que no estuviese definida en el servidor, lo haría a través de esta clase. Posee la función *handleType (…)* que se llamará cada vez que el usuario decida subir un fichero al servidor. Esta función nos indica el tipo de fichero que el usuario está intentando subir, así como su tamaño y localización donde éste se encuentra. Cuando un fichero se ha guardado en el servidor de manera satisfactoria se actualizan el resto de componentes y el motor de PRO*forma* a través de las llamadas a una funciones globales que se encontrarán definidas dentro de la clase AppData (la explicación de esta clase y su funcionamiento se ven en el punto 4.4 de esta memoria).

4.3.3. Creación del componente grafo

Para la creación de esta aplicación se ha desarrollado un componente nuevo (requerido para crear el grafo), ya que actualmente no existe ninguno (ni como componente genérico de Vaadin, ni como componente desarrollado por la comunidad) que cumpla los requisitos que se pedían para este proyecto.

Antes de avanzar en la explicación del desarrollo, se explicará cómo se estructuran los componentes en Vaadin y como se conectan éstos internamente, ya que cada componente tiene una parte en de código en el servidor y otra en el cliente. En la Figura 12**¡Error! No se encuentra el origen de la referencia.** se puede observar la arquitectura de los componentes desarrollados en Vaadin.

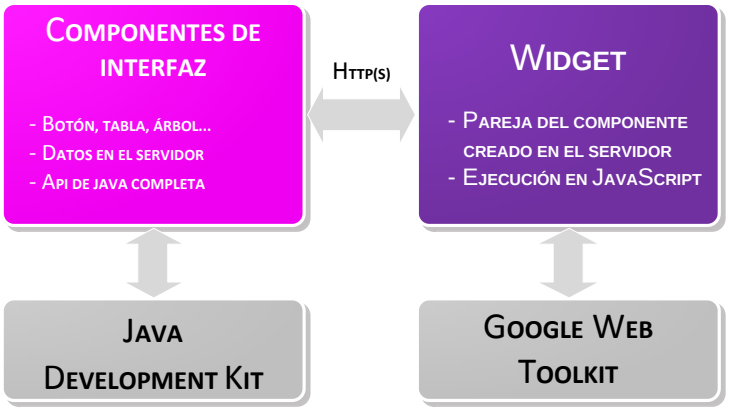

Figura 12: Arquitectura de los componentes de la interfaz de usuario

Todos los componentes encargados de dibujar algo (p.ej. botones, tablas, paneles…) siguen esta arquitectura. Vaadin los ofrece ya compilados y no es necesario que el desarrollador sepa cómo se conectan con el servidor, pero cuando es necesario crear uno nuevo hay que compilar el código creado en el cliente, ya que, como se ha comentado antes, el navegador ejecutará la aplicación como JavaScript. Por otro lado, la parte desarrollada en el servidor, se compilará de manera normal a través del JDK de Java.

Tal y como se muestra en la Figura 13, cuando creamos un componente nuevo, tendremos dos clases principales bien definidas, una en el servidor y otra en el cliente. La primera tiene que definir una interfaz de programación (*Application Programming Interface, API*) y debe contar con unos métodos que gestionen los eventos del cliente y que puedan enviarle los posibles cambios que se puedan producir en el servidor. Por otro lado, la clase definida en el cliente debe ser capaz de recoger los eventos enviados por el servidor, y debe poder transformar estos eventos en un lenguaje comprensible por los navegadores. Esto lo hará a través de código JavaScript que utilizará las funciones del modelo de objetos del documento (*Domain Object Model, DOM*).

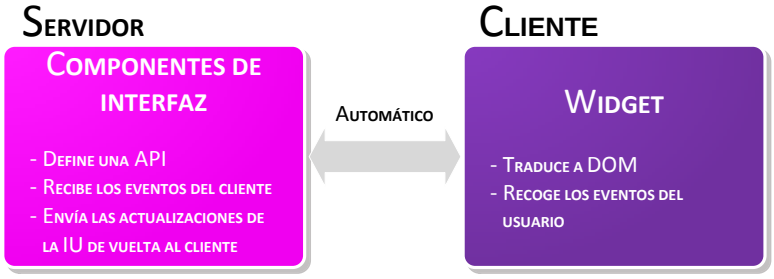

Figura 13: Funciones de las clases implementadas al crear un nuevo componente

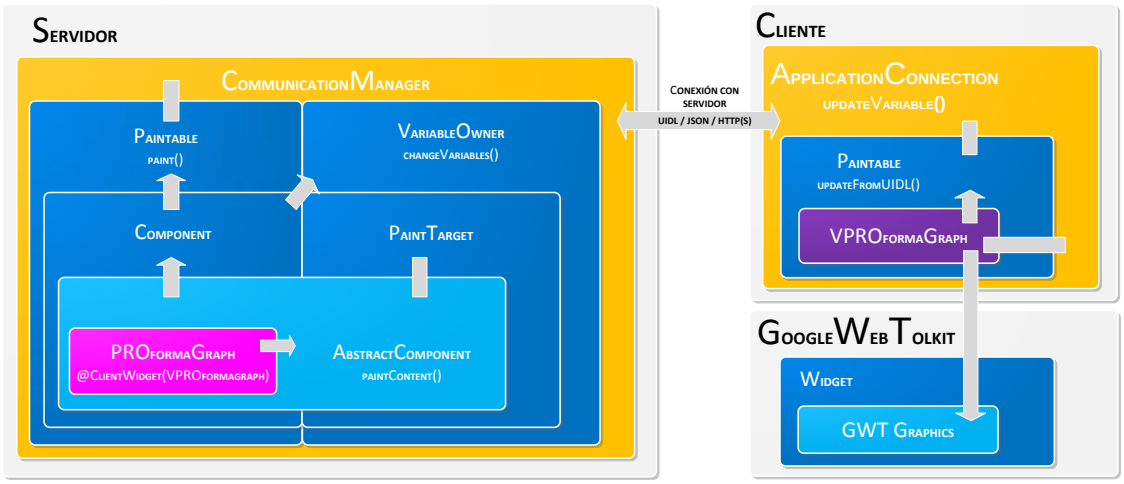

Figura 14: Estructura de clases y funciones llamadas en la creación del componente VPROformaGraph.

Todas estas relaciones y métodos utilizados para establecer la comunicación entre el cliente y el servidor se muestran resumidos en la Figura 14. Como se ve en la figura superior, Vaadin ofrece una solución a este problema de comunicación que funciona de manera automática. Basta con indicar en la clase que se defina en el servidor, el nombre de la clase que se utilizará en el cliente y el sistema de gestión de eventos se encargará del resto. El desarrollador solo se tiene que preocupar de enviar y recibir los datos utilizando las funciones definidas por el *framework*.

Se puede observar en la Figura 14 que Vaadin ofrece una serie de interfaces (en la figura representadas en rectángulos de color azul oscuro) que nos indican las funciones que tienen que ser utilizadas para que el gestor de eventos definido serialice y deserialice (en inglés *marshall* y *unmarshall* respectivamente) las variables entre el cliente y el servidor. Esto quiere decir que a través de estas funciones se codificarán los objetos creados en el servidor o el cliente, se enviarán a través de la red y se descodificarán en el lado contrario.

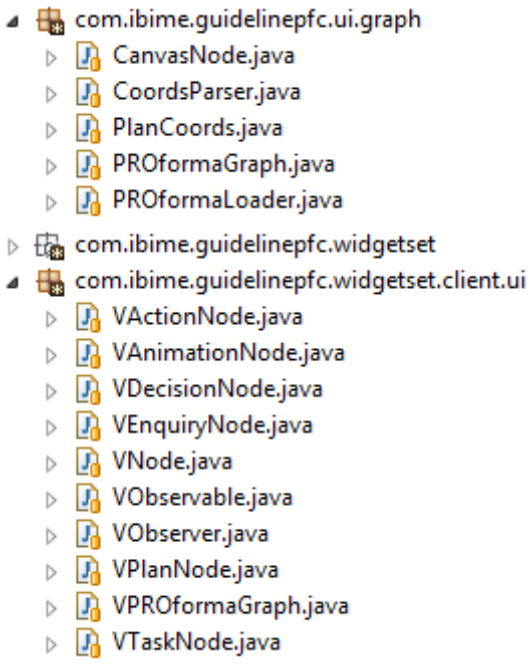

Figura 15: Clases creadas para representar el grafo de PROforma

Las clases representadas en los rectángulos magenta son las clases básicas que se han definido para crear el grafo. Por su parte cada una de ellas se relaciona con otros conjuntos de clases definidos en el cliente y el servidor, pero que por claridad de la Figura 14 se han dejado fuera del esquema y se representan en Figura 15.

En la figura adjunta se pueden ver las clases definidas que se utilizan para construir el grafo. Las clases que se encuentran dentro del paquete *widgetset* serán las que el cliente utilice. Como se puede comprobar, se ha definido una clase por tipo de nodo a representar (*action, decision, enquiry, plan* y *task*) y una clase genérica *Vnode* que contendrá

las variables comunes a todos los nodos. Además se ha creado una clase para gestionar la animación de dichos nodos, que los actualizará dependiendo de en qué estado se encuentren (dormidos, en ejecución o terminados).

Este tipo de implementación de los componentes permite que en la parte del servidor se utilice cualquier *widget* definido en GWT , como se puede observar si volvemos a la Figura 15**¡Error! No se encuentra el origen de la referencia.** la clase definida en el cliente *VPROformaGraph* se relaciona con el *widget* de GWT *gwt-graphics*. Esto era necesario debido a que los distintos navegadores existentes (Chrome, Firefox, Opera, Safari, Explorer…) usan distintos motores de representación, lo cual puede provocar que una misma imagen se vea de maneras distintas en todos ellos. Para poder evitar eso, se utiliza el lienzo de dibujo (*Canvas*) de HTML5 cuyas funciones básicas se encuentran definidas en el *widget* de GWT previamente nombrado.

También en la Figura 16 se puede observar que hay dos clases denominadas *VObserver* y *VObservable*, esto es debido a que GWT limita las clases de Java que se pueden utilizar en el cliente. Como para relacionar los nodos con la clase de animación se utiliza el patrón *Observer*, se tuvieron que crear en el cliente las clases correspondientes que permitieran dicha implementación.

El patrón de diseño *Observer* es utilizado en el grafo para notificar la finalización de la animación de los nodos. Cada nodo observará la clase *VAnimationNode*, que indicará la duración de la animación del nodo. Cuando la animación termine, *VAnimationNode* avisará al nodo correspondiente de que ha terminado, pero si el nodo sigue en activo, esto es, que su estado no ha cambiado, la animación empezará de nuevo.

4.4.Gestión de las sesiones de usuario

Como paso final en la implementación de esta aplicación hubo que crear un sistema que fuese capaz de gestionar diferentes sesiones de usuario. Esto fue debido a que la aplicación necesitaba acceder a varios objetos desde prácticamente cualquier clase del código y usar variables estáticas o el patrón de diseño *Singleton* proporciona acceso global a los datos, pero no solo a cada sesión, sino en general. Esto provocaba que los datos fuesen compartidos por todos los usuarios que iniciaban la aplicación, lo cual generaba un comportamiento no deseado durante la ejecución de la guía clínica.

Para atajar este problema se guardarán las referencias a estos datos globales usando el patrón de hilo local (*ThreadLocal Pattern*). Este patrón ofrece la solución al acceso de datos global resolviendo a su vez dos problemas más.

El primer problema se plantea debido a que el contenedor de *servlets* procesa las peticiones para múltiples usuarios de manera secuencial, por lo que si un usuario establece una variable estática con un valor, esta puede ser leída o modificada por la siguiente petición perteneciente al siguiente usuario. Como muestra la Figura 16 se puede resolver esta situación estableciendo una referencia global que apunte a los datos del usuario actual al principio de cada petición HTTP. En este caso usamos la interfaz *TransactionListener* estableciendo las referencias a través de la funcione *transactionStart ()* y eliminándolas con la función *transactionEnd ()*.

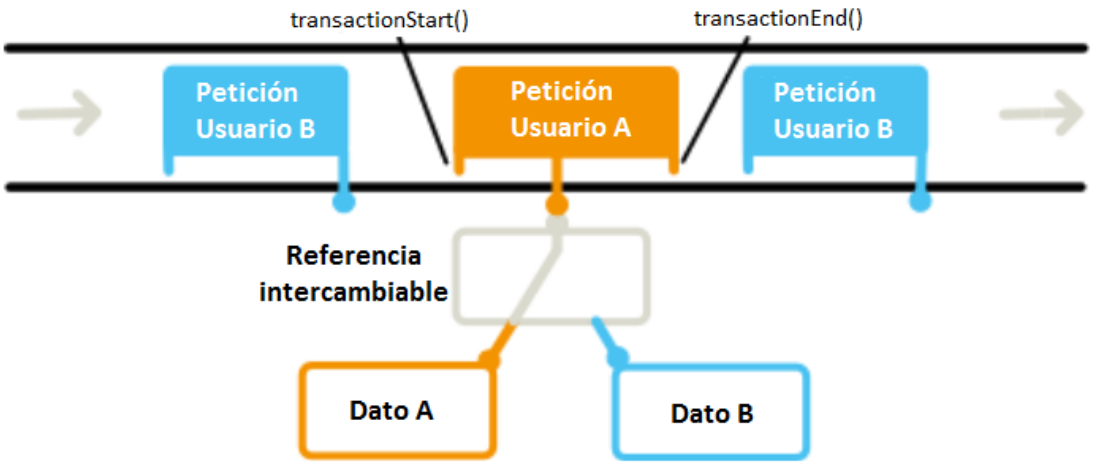

Figura 16: Cambio de referencia durante un procesamiento secuencial de peticiones

El segundo problema se presenta cuando los contenedores de servlets almacenan múltiples hilos de ejecución que requieren datos. Si en esta situación cambiamos una variable estática, esta variable cambiaría en todos los hilos de ejecución simultáneamente presentando problemas si otro usuario desea acceder a dicha variable. La solución es almacenar la referencia en una variable local del propio hilo (*thread-local variable*), lo cual se puede hacer en java usando la clase *ThreadLocal*. Este problema se muestra en la Figura 17, así como el acceso a las variables locales si se utiliza la solución planteada.

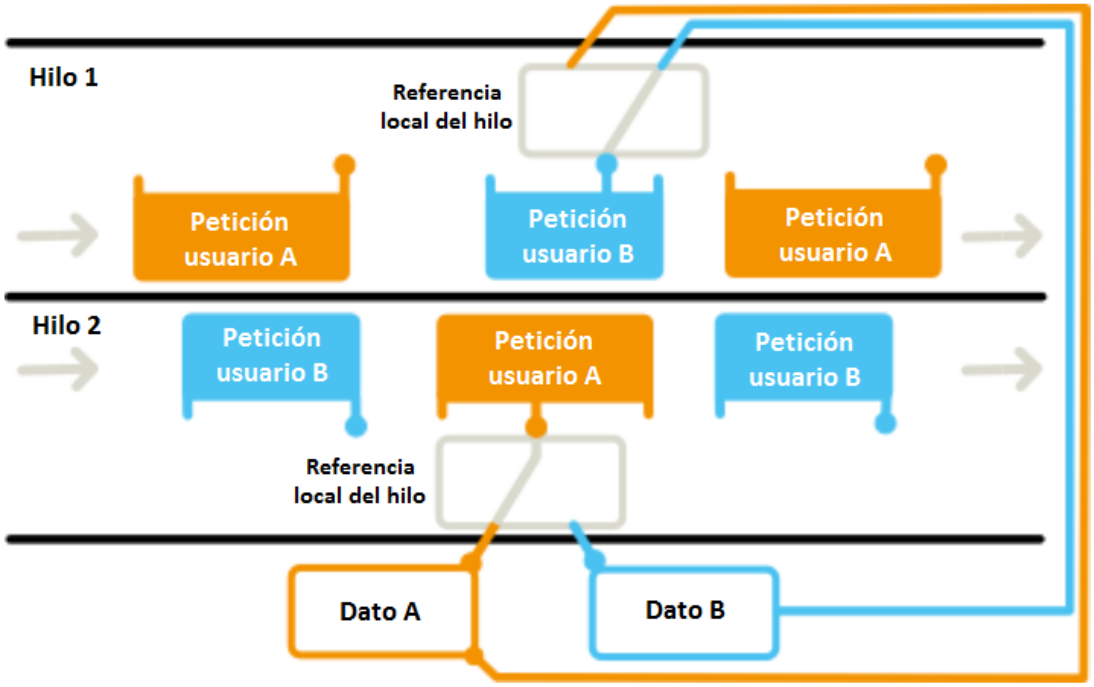

Figura 17: Cambio de variables locales durante un procesado de peticiones concurrente

Cuando se utiliza la interfaz *TransactionListener*, los *listeners* se usarán dentro del contexto de la aplicación (o sesión de usuario), no dentro de la instancia de la aplicación. En la práctica, esto quiere decir que si se utiliza más de una aplicación de Vaadin a la vez en la misma sesión (lo cual puede pasar, porque el *framework* lo permite) las peticiones se realizarían sin diferenciar entre las distintas aplicaciones, por lo que el dato accedido debería comprobar a que aplicación pertenece cuando las transacciones se inicien o finalicen.

Este proyecto se ha desarrollado como una aplicación única, por lo que esas comprobaciones de datos no son necesarias. Además siempre que una transacción termina, se limpian las referencias a los datos estableciendo su valor a *null*, con lo que se evitan fugas de memoria y el acceso por error a dichos datos por la siguiente petición de datos del siguiente usuario.

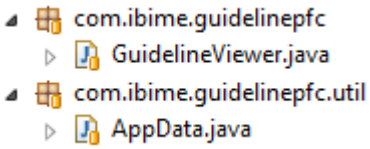

Figura 18: Clases encargadas de la gestión de los datos del usuario

En la Figura 18 se pueden observar las clases que gestionan todo lo comentado en este punto. Dentro de *AppData* se guardan las variables que serán accedidas de manera local usando la clase *ThreadLocal* y en la clase *GuidelineViewer*, que es el punto de acceso a la aplicación, se crean una instancia de AppData y se enlaza con los datos de la aplicación que serán locales y dependerán de la sesión de usuario.

5. Guía clínica estudiada (CHF- COPD)

5.1.Introducción a la guía

Para la realización de este proyecto se ha seleccionado una guía clínica que combina los tratamientos para COPD/EPOC (*Chronic Obstructive Lung Disease*, enfermedad pulmonar obstructiva crónica) y CHF/FCC (*Chronic Heart Faliure*, fallo cardíaco crónico FCC) La creación y desarrollo de esta guía combinada se pueden consultar en (37).

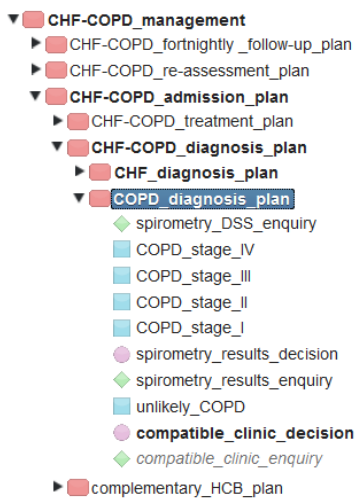

Figura 19: Sub-plan elegido

Como el tamaño de esta guía es considerable, para realizar el enlace de los datos definidos en la guía junto con el sistema de ayuda a la decisión elegido se ha seleccionada una subguía definida dentro de la principal. La subguía elegida se define por el plan *COPD_diagnosis_plan*, como se puede ver en la Figura 19, este plan se encuentra dentro del plan de diagnóstico que a su vez se encuentra dentro del plan de admisión del paciente, lo cual quiere decir que nada más cargar la guía se ejecutará este plan y se requerirán ciertos datos acerca del paciente.

Para complementar la guía con el sistema de ayuda se han añadido unos pequeños cambios en la definición de la guía, permitiendo así la recolección de los datos necesitados por el SADC y el posterior envío de éstos de vuelta a la guía clínica. En los siguientes apartados de la memoria se describirán estos cambios y cómo afecta a la ejecución de la guía.

5.2.Modelo de espirometría en el subplan EPOC

Para la demostración de la relación entre el sistema de ayuda a la decisión y la guía clínica se ha elegido el subplan de EPOC, el cual se puede observar en la Figura 20.

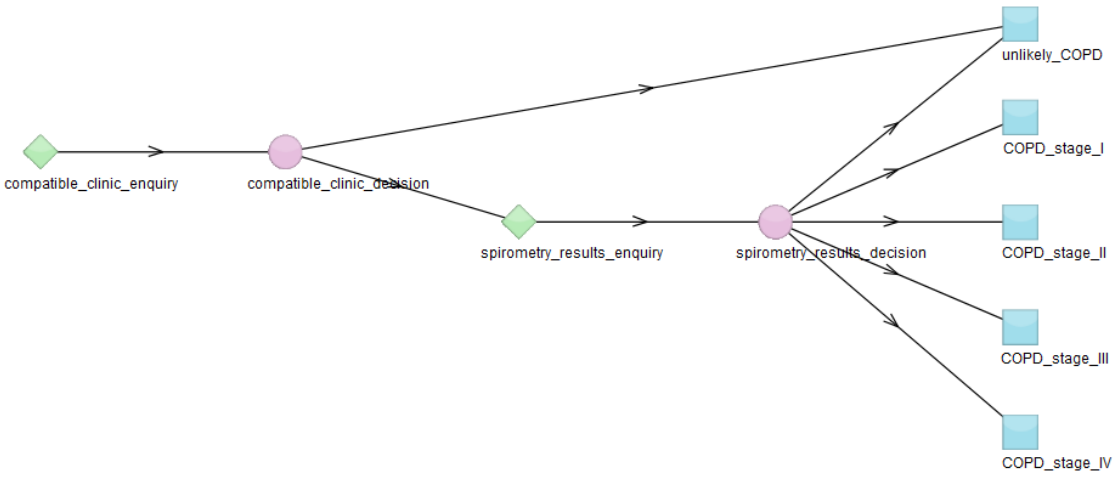

Figura 20: COPD_diagnosis_plan original

El plan cuenta con dos nodos de consulta de datos, dos nodos de decisión y cinco nodos que representarán las acciones que el médico debería tomar si siguiese las indicaciones de la guía.

En el primer nodo de consulta, *compatible_clinic_enquiry*, requiere los siguientes datos: *age*, *chronic*_cough, *chronic_respiratory_failure*, *chronic_sputum_production*, *dyspnea* y *risk_factors*. Posteriormente el nodo de decisión *compatible_clinic_decision* analizará los datos anteriormente introducidos y recomendará uno de sus candidatos, ya sea a favor de continuar con las pruebas clínicas o no. Si se llega al nodo de consulta *spirometry_results_enquiry* se nos pedirán los datos *FEV1_FVC_postbronchodilator* y *FEV1_prediction*, y basados en estos datos y en los introducidos al inicio del plan, el nodo *spirometry_results_decision* recomendará una de las cinco posibles acciones a elegir.

Como se puede ver, se ha elegido este plan porque en el nodo *spirometry_results_enquiry* requiere unos datos de predicción que actualmente se introducen a mano, o están codificados como valores por defecto dentro de la propia guía, pero que podrían ser calculados por un SADC sin ningún problema, ayudando así a agilizar la ejecución de la guía y previniendo la introducción de posibles errores por parte del personal médico.

5.2.1. Calculo de los parámetros requeridos (FEV1 predicted y FEV1/FVC)

El sistema de ayuda a la decisión calculará estos parámetros en base a los datos anteriormente introducidos, pero antes de explicar cómo se calculan estos valores, se explicará brevemente que es lo que se quiere calcular y porqué estos datos son imprescindibles si lo que se quiere es diagnosticar si el paciente padece una enfermedad pulmonar obstructiva crónica.

Para empezar, estos valores tienen sentido dentro del conjunto de pruebas pulmonares conocido como espirometría, que se encargan de medir la funcionalidad de los pulmones, específicamente miden la cantidad (volumen) y la velocidad (flujo) a la que el aire puede ser inspirado o espirado de éstos. Gracias a este conjunto de pruebas se pueden conseguir los siguientes valores, que indicarán el estado de los pulmones del paciente que realizó la prueba:

- ***Forced Vital Capacity* (FVC, Capacidad Vital Forzada):** es el volumen máximo de aire que puede ser exhalado por el paciente durante una espiración forzada, es decir, con la máxima rapidez que el paciente pueda producir, se mide en litros y es la prueba más básica dentro del conjunto de pruebas de espirometría
- ***Forced Expiratory Volume in 1 second* (FEV1, Volumen Espiratorio Forzado):** es la cantidad de aire expulsado durante el primer segundo de la espiración máxima, realizada tras una inspiración máxima
- ***Ratio FEV1/FVC* (FEV1%, relación FEV1/FVC):** es la relación, en porcentaje, de la capacidad forzada que se espira en el primer segundo, con el total exhalado para la capacidad vital forzada. Su valor normal (**FEV1% normal**) en adultos saludables suele ser entorno al 80% (38)
- ***FEV1% predicted* (FEV1% predicho):** es un valor derivado del FEV1%, está definido como el FEV1% del paciente entre la media del FEV1% de la población para una persona de edad, sexo y complexión similar

Para pacientes con enfermedades pulmonares como la EPOC, el **FEV1** será menor debido a la resistencia de las vías para el aumento del flujo espiratorio. El **FVC** también puede ser menor debido al cierre prematuro de las vías respiratorias durante la espiración, solo que este valor no va en la misma proporción que el **FEV1**. Debido a esto los pacientes con un cociente **FEV1/FVC** menor que 0.7 (7), son bastante propensos a padecer EPOC.

El sistema de ayuda a la decisión calculará los valores **FEV1/FVC** y **FEV1% predicted** en base a los valores introducidos anteriormente en la guía que serán: edad, género, peso, FVC y FEV1. Una vez con estos valores**,** el SADC calculará internamente el **FEV1% normal** y aplicará la fórmula que se puede ver en la Figura 21, posteriormente enviará los resultados a la guía clínica para que esta pueda decidir si el paciente padece o no EPOC y si lo padece, cuál sería la gravedad de dicha enfermedad.

$$\text{Fev1\% predicted} = \frac{\text{Fev1\% paciente}}{\text{Fev1\% normal}}$$

Figura 21: Fórmula utilizada en el SADC

5.3.Cambios realizados a la guía

Para poder enlazar el SADC con la guía clínica y realizar los cálculos anteriormente explicados, se han tenido que aplicar ciertos cambios en el plan elegido (*COPD_diagnosis_plan*). Estos cambios no afectan a la toma de decisiones de la guía ya que, como se puede ver en la Figura 22, únicamente se ha introducido un nuevo nodo de consulta que preguntará al personal médico por los datos requeridos para poder llevar a cabo la predicción de los valores **FEV1/FVC** y **FEV1% predicted**.

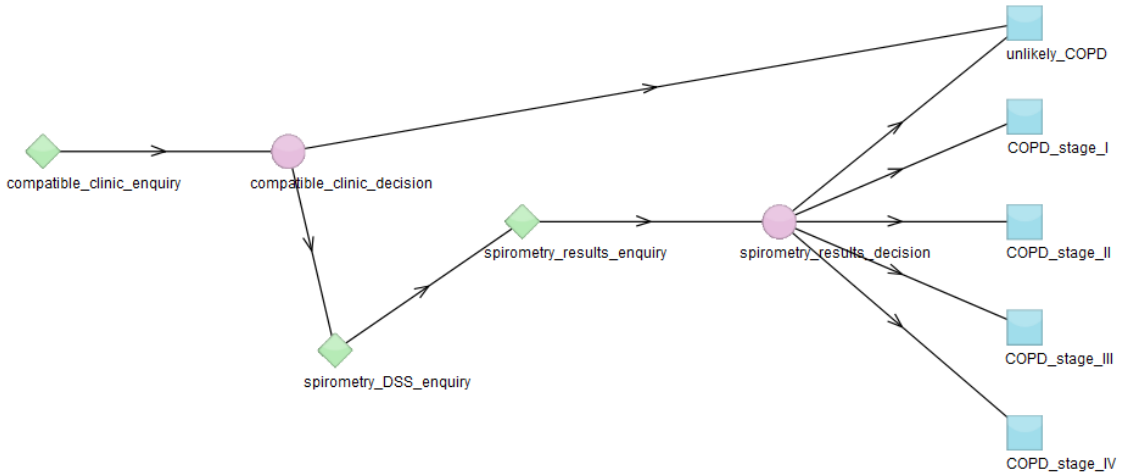

Figura 22: COPD_diagnosis_plan modificado

Para mantener la integridad de la guía y que el flujo de ejecución se vea obligado a pasar por el nuevo nodo, se ha cambiado la precondición de ejecución del nodo *spirometry_results_enquiry*, es decir, antes ese nodo se ejecutaba cuando el nodo *compatible_clinic_decision* terminaba y el candidato resultado de su decisión era *compatible_clinic*, ahora, esas precondiciones se encuentran en el nuevo nodo añadido, y la condición para que se ejecute el antiguo *spirometry_results_enquiry* es que el estado del nodo *spirometry_DSS_enquiry* sea completado.

6. Conexión del sistema ayuda a la decisión clínica

Dichos sistemas también son conocidos como SADC o en inglés *decisión support system* o DSS. De ahora en adelante cuando nos refiramos a un sistema de estos, lo nombraremos como SADC.

6.1.SADC utilizado

Los sistemas de ayuda a la decisión clínica (SADC) son sistemas computacionales que aportan conocimiento específico para la ayuda a la decisión en el diagnóstico, pronóstico, tratamiento o administración de pacientes.

Para la realización de este proyecto se ha utilizado el motor de clasificación genérico y extensible empleado en el SADC Curiam del grupo IBIME (39) (40), el cual se puede encontrar definido con más detalle en (41).

El SADC, al igual que el motor de PRO*forma*, ha sido utilizado como una caja negra (es decir, no importa cómo está diseñado, sino sus resultados y comunicación con el exterior). Debido a esto se ha trabajado en desarrollar unos mecanismos de comunicación efectivos entre este sistema y la aplicación desarrollada. A continuación se describen dichos sistemas de comunicación.

6.2.Sistema de enlace creado entre la aplicación y el SADC

Para poder relacionar el SADC con la guía clínica lo primero que se hizo fue identificar los datos que iban a ser externos a la guía clínica de una manera diferente, y ya que el lenguaje de PRO*forma* está perfectamente definido como se ha visto en el apartado 2.3 de esta memoria, se descartó introducir cambios dentro de la definición formal de la guía. Debido a esto, se decidió crear un nuevo método para relacionar las guías definidas en PRO*forma de forma no invasiva*. Esto fue llevado a cabo mediante un fichero XML externo.

```
<?xml version="1.0" encoding="utf-8"?>

    <guideline name="CHF-COPD_management_1">

        <dataId name="FEV1_FVC_postbronchodilator">
            <DSS name="00_test_epoc_predicted">
                <variable>Age</variable>
                <variable>Gender</variable>
                <variable>Weight</variable>
                <variable>FEV1</variable>
                <variable>FVC</variable>
            </DSS>
        </dataId>

        <dataId name="FEV1_prediction">
            <DSS name="00_test_epoc_predicted">
                <variable>Age</variable>
                <variable>Gender</variable>
                <variable>Weight</variable>
                <variable>FEV1</variable>
                <variable>FVC</variable>
            </DSS>
        </dataId>

    </guideline>
```

Figura 23: Esquema XML utilizado para definir los datos externos de la guía clínica

Como se puede observar en la Figura 23, el esquema XML definido indicará como primer valor el nombre de la guía clínica para el cual se definen estos valores, esto permitirá en un futuro el uso de un solo archivo XML para definir los datos relacionados con el SADC de todas las guías clínicas que se tengan disponibles.

Posteriormente se pasan a definir los datos que serán los que reciban los valores externos que generará el SADC, dichos valores se identifican por el nombre, que en PRO*forma* se estipula que será único. Dentro del nombre del dato, se definirá el modelo que usará el SADC para predecir el dato requerido y definido dentro de éste irán las variables necesarias por el modelo para conseguir dicho resultado.

Una vez definido el esquema XML, se creó un analizador de dicho esquema, este se programó en Java, y lo que hace es leer el documento XML definido, comprobar que la guía clínica que actualmente está en ejecución tenga datos externos a la guía, es decir, que requieran datos del SADC y posteriormente marca esos datos para que cuando el motor de ejecución llegue a ellos, en vez de preguntar al personal médico, se envíe una petición al modelo del SADC definido en el documento XML.

Una versión más completa y genérica de este documento XML así como su XML schema está siendo desarrollada actualmente por la grupo de minería biomédica[12] dentro del grupo de investigación IBIME[13] y permitirá conectar el lenguaje de definición de guías PRO*forma* con cualquier fuente externa de datos.

[12] http://www.ibime.upv.es/bmg/
[13] http://www.ibime.upv.es/

7. Conclusiones

7.1. Resumen del trabajo realizado

A lo largo de esta memoria se ha presentado un sistema con el aspecto de una aplicación de escritorio que funciona vía web y que relaciona sistemas de ayuda a la decisión clínica con guías clínicas electrónicas, concretamente guías clínicas electrónicas definidas en PRO*forma*.

Este sistema permite cargar un esquema XML en el que se encontrarán las relaciones entre los SADC y las guías definidas anteriormente, por lo que no es necesario modificar nada ni de las guías clínicas ni del SADC.

Una vez cargada la guía se representa esta de manera gráfica y esquemática en forma de grafo dirigido y árbol de datos, mostrando en todo momento el estado actual de los nodos que componen la guía y con capacidad de mostrar, si se requiere, información extra acerca de cada nodo.

A la vez que se muestran estas representaciones, la guía se encontrará en ejecución y el usuario podrá introducir datos y tomar decisiones dentro de ésta si así lo desea. Otra de las opciones presentadas es dejar que el motor de datos tome las decisiones que considere más oportunas basándose en los datos de los que dispone, agilizando así la ejecución de la guía.

Además gestiona de manera eficaz las distintas sesiones iniciadas por los usuarios reservando espacios en memoria únicos para cada uno lo cual facilita la gestión de los permisos de acceso dentro del servidor.

7.2. Futuras líneas de investigación

Como se ha comentado al principio de esta memoria, el proyecto se ha dividido en tres grandes apartados (estudio de las guías clínicas, relación de éstas con un sistema de ayuda a la decisión y creación de la interfaz web), por lo que las líneas de investigación podrían estar enfocadas en cualquiera de estas tres partes.

Para la primera parte (estudio de las guías clínicas) se podría plantear la relación de estas con modelos probabilísticos, lo cual por ejemplo, permitiría a las guías basarse en teoría de la decisión. Además, se podría plantear aprender a partir de unas muestras históricas. Esto se traduciría en unas guías clínicas con capacidad de tomar mejores decisiones evitando así acciones innecesarias y mejorando el trato al paciente ya que este sería más personalizado. Además una guía con capacidad de aprendizaje requeriría menos mantenimiento ya que a través de los datos y resultados del propio paciente y a través de las evidencias médicas se podría mantener actualizada.

Respecto a la segunda parte (relacionar los SADC con guías clínicas), en el punto 6.2 de esta memoria se ha comentado que actualmente está en desarrollo una versión más genérica del mecanismo de conexión desarrollado para esta aplicación, lo cual permitirá conectar un motor de ejecución de guías clínicas con cualquier fuente de datos externa a dichas guías.

En referencia a la tercera parte, se podría plantear el estudio de la utilización por parte del personal médico de las interfaces de usuario más comunes, comprobando así cuáles son sus necesidades básicas y que consideran ellos relevante de cara a la atención al paciente. Si

posteriormente se aplican los resultados de estos estudios a las interfaces creadas, esto se podría traducir en un menor rechazo por parte del personal que lo tenga que utilizar y una mayor implantación de este tipo de aplicaciones en nuestra vida diaria.

Bibliografía

1. National Guideline Clearingohuse. [Online]. [cited 2011 08 30. Available from: http://www.guideline.gov/browse/index.aspx?alpha=A.

2. Published clinical guidelines. [Online]. [cited 2011 08 30. Available from: http://www.nice.org.uk/Guidance/CG/Published.

3. Catálogo de Guías de Práctica Clínica en el Sistema Nacional de Salud. [Online]. [cited 2011 08 30. Available from: http://portal.guiasalud.es/web/guest/catalogo-gpc.

4. Cabana MD, Rand CS, Powe NR, Wu AW, Wilson MH, Abboud PA, et al. Why dont physicians follow clinical practice guidelines? A framework for improvement. JAMA : the journal of the American Medical Association. 1999 oct 20; 282: p. 1458--1465.

5. Veatch RM. Reasons Physicians Do Not Follow Clinical Practice Guidelines. JAMA: The Journal of the American Medical Association. 2000; 283: p. 1685-1686.

6. Baiardini I, Braido F, Bonini M, Compalati E, Canonica GW. Why do doctors and patients not follow guidelines? Current Opinion in Allergy and Clinical Immunology. 2009; 9: p. 228--233.

7. Guía de práctica clínica (EPOC). [Online].; 2010 [cited 2011 09 04. Available from: http://www.guiasalud.es/GPC/GPC_468_EPOC_AP_AE.pdf.

8. Field MJ. Guidelines for clinical practice : from development to use: National Academy Press; 1992.

9. Grimshaw JM, Russell IT. Effect of clinical guidelines on medical practice: a systematic review of rigorous evaluations. Lancet. 1993 nov 27; 342: p. 1317--1322.

10. Kosimbei G, Hanson K, English M. Do clinical guidelines reduce clinician dependent costs? Health Res Policy Syst. 2011; 9: p. 24.

11. Hibble A, Kanka D, Pencheon D, Pooles F. Guidelines in general practice: the new Tower of Babel? BMJ. 1998; 317: p. 862-863.

12. Hanka R. Guidelines in general practice. Information overload on GPs desks must be overcome. BMJ. 1999 May; 318: p. 1212.

13. Sherman EH, Hripcsak G, Starren J, Jenders RA, Clayton P. Using intermediate states to improve the ability of the Arden Syntax to implement care plans and reuse knowledge. Proc Annu Symp Comput Appl Med Care. 1995;: p. 238-42.

14. Health Level 7. Arden Syntax. [Online]. [cited 2011 08 27. Available from: http://www.hl7.org/implement/standards/ardensyntax.cfm.

15. Tu SW, Musen MA. Modeling data and knowledge in the EON guideline architecture. In

. Medinfo; 2001. p. 280--284.

16 Tu SW, Campbell JR, Glasgow J, Nyman MA, McClure R, McClay J, et al. The SAGE Guideline
. Model: achievements and overview. J Am Med Inform Assoc. 2007; 14: p. 589--598.

17 Miksch S, Shahar Y, Johnson P. Asbru: A Task-Specific, Intention-Based, and Time-Oriented
. Language for Representing Skeletal Plans. In ; 1997. p. 9--1.

18 Shahar Y, Miksch S, Johnson P. The Asgaard project: a task-specific framework for the
. application and critiquing of time-oriented clinical guidelines. 1998 Sep 01..

19 Fox J, Patkar V, Thomson R. Decision support for health care: the PROforma evidence base.
. Informatics in Primary Care. 2006 mar; 14: p. 49--54.

20 Ohno-Machado L, Gennari JH, Murphy SN, Jain NL, Tu SW, Oliver DE, et al. The guideline
. interchange format: a model for representing guidelines. J Am Med Inform Assoc. 1998; 5:
p. 357--372.

21 Boxwala AA, Peleg M, Tu S, Ogunyemi O, Zeng QT, Wang D, et al. GLIF3: a representation
. format for sharable computer-interpretable clinical practice guidelines. J. of Biomedical
Informatics. 2004 June; 37(3): p. 147--161.

22 Ciccarese P, Caffi E, Boiocchi L, Quaglini S, Stefanelli M. A guideline management system.
. Stud Health Technol Inform. 2004; 107: p. 28--32.

23 Skonetzki S, Gausepohl HJ, van der Haak M, Knaebel S, Linderkamp O, Wetter T. HELEN, a
. modular framework for representing and implementing clinical practice guidelines.
Methods Inf Med. 2004; 43: p. 413--426.

24 Terenziani P, Montani S, Bottrighi A, Torchio M, Molino G, Correndo G. The GLARE
. approach to clinical guidelines: main features. Stud Health Technol Inform. 2004; 101: p.
162--166.

25 Johnson PD, Tu S, Booth N, Sugden B, Purves IN. Using Scenarios in Chronic Disease
. Management Guidelines for Primary Care. 2000..

26 ASTM. ASTM E2210 - 06 Standard Specification for Guideline Elements Model version 2
. (GEM II)-Document Model for Clinical Practice Guidelines. [Online].; 2011 [cited 2011 08
07. Available from: http://www.astm.org/Standards/E2210.htm.

27 Standard Specification for Guideline Elements Model version 2. [Online]. [cited 2011 09 04.
. Available from: http://www.astm.org/Standards/E2210.htm.

28 Sutton DR, Fox J. The syntax and semantics of the PROforma guideline modeling language.
. Journal of the American Medical Informatics Association. 2003; 10: p. 433--443.

29 Fox J, Das S. Safe and Sound: Artificial Intelligence in Hazardous Applications: AAAI Press;

. 2000.

30 Knowledge Modelling. [Online].; 2011 [cited 2011 08 27. Available from:
. http://www.cossac.org/technologies/proforma/modelling.

31 Walton RT, Gierl C, Yudkin P, Mistry H, Vessey MP, Fox J. Evaluation of computer support
. for prescribing (CAPSULE) using simulated cases. BMJ. 1997; 315: p. 791-795.

32 Emery J, Walton R, Murphy M, Austoker J, Yudkin P, Chapman C, et al. Computer support
. for interpreting family histories of breast and ovarian cancer in primary care: comparative study with simulated cases. BMJ. 2000; 321: p. 28-32.

33 Bury J, Hurt C, Roy A, Cheesman L, Bradburn M, Cross S, et al. LISA: a web-based decision-
. support system for trial management of childhood acute lymphoblastic leukaemia. Br. J. Haematol. 2005 Jun; 129: p. 746--754.

34 Learn Vaadin. [Online]. [cited 2011 08 29. Available from: http://vaadin.com/learn.
.

35 Book of Vaadin. [Online]. [cited 2011 08 29. Available from: http://vaadin.com/book/-
. /page/preface.html.

36 Gamma E, Helm R, Johnson R, Vlissides JM. Design Patterns: Elements of Reusable Object-
. Oriented Software: Addison-Wesley Professional; 1994.

37 Lozano E, Marcos M, MartÃnez-Salvador B, Alonso A, Alonso J. Experiences in the
. Development of Electronic Care Plans for the Management of Comorbidities. 2010: p. 113-123.

38 Forced Expiration. [Online].; 2001 [cited 2011 08 29. Available from:
. http://oac.med.jhmi.edu/res_phys/Encyclopedia/ForcedExpiration/ForcedExpiration.HTML
.

39 C S, JM GG, J V, S T, M E, A N, et al. A generic decision support system featuring an
. assembled view of predictive models for magnetic Resonance and clinical data. In ESMRMB 25th annual meeting; 2008; Valencia.

40 Sáez C. Solución para el desarrollo de Sistemas de Ayuda a la Decisión para diagnóstico
. clínico. 2009. Tesis de Máster, Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital, DSIC, UPV.

41 Sáez C, García-Gómez JM, Vicente J, Tortajada S, Luts J, Dupplaw D, et al. A generic and
. extensible automatic classification framework applied to brain tumour diagnosis in HealthAgents. Knowledge Eng. Review. 2011; 26: p. 283-301.