

Editor de guías clínicas (GDL) basadas en arquetipos

Índice

1. Introducción	1
1.1. Contextualización	1
1.2. Objetivos	4
2. Fundamentos tecnológicos	5
2.1. Aplicaciones web	5
2.2. Node.js	6
2.3. AngularJS y patrón de arquitectura Model-View-Controller	7
2.3.1. Módulos en AngularJS	7
2.3.2. Templates	8
2.3.3. Controladores	9
2.3.4. Directivas	9
2.3.5. Objeto <code>\$scope</code> de AngularJS	11
2.3.6. Two-way databinding	11
2.4. Patrones de diseño	13
2.4.1. Patrón <i>Singleton</i>	13
2.4.2. Patrón <i>Fachada</i>	14
2.4.3. Patrón <i>Proxy</i>	15
2.4.4. Inyección de dependencias	16
3. Estado del arte	16
3.1. Estándares de interoperabilidad	16
3.2. <i>openEHR</i> : Modelo de Referencia y Modelo de arquetipos	16
3.3. Terminologías clínicas	17
3.4. Especificación GDL	17
3.4.1. Introducción y requisitos	17
3.4.2. Principios de diseño	17
3.4.3. Modelo de Objetos de Guías Clínicas	18
3.4.4. Paquete de Guías Clínicas	18
3.4.5. Paquete de Expresiones	20
4. Métodos	23
4.1. Introducción	23
4.2. Metodología de desarrollo	23
4.2.1. Metodología Scrum	23
4.2.2. Fases de Scrum	29
4.2.3. Ventajas de Scrum	29
4.3. Aplicación de Scrum en el proyecto	30

5. Desarrollo	31
5.1. Iteración 1	32
5.1.1. Análisis	32
5.1.2. Diseño	33
5.1.3. Implementación	36
5.1.4. Pruebas	42
5.2. Iteración 2	46
5.2.1. Análisis	47
5.2.2. Diseño	47
5.2.3. Implementación	47
5.2.4. Pruebas	47
6. Testing	47
6.1. Jasmine Framework	47
6.2. Tests unitarios con Karma	47
6.3. Tests end-to-end con Protractor	47
7. Material utilizado	47
8. Implantación y seguimiento (si corresponde)	48
9. Planificación y costes (planificación inicial vs. final) - (Opcional)	48
10. Resultados y discusión	48
11. Conclusiones	48
12. Futuros trabajos	48
13. Bibliografía	48
13.1. Referencias	48
14. Glosario	49
A. Apéndices	49
A.1. Gramática de GDL	49
A.2. Manual de usuario	62
A.2.1. Guidelines	62
A.2.2. Description	63
A.2.3. Definitions	63
A.2.4. Rule List	67
A.2.5. Edición de reglas	68
A.2.6. Editor de expresiones	70
A.3. Ejemplo de creación de una guía clínica	70

Índice de figuras

1.	Toma de decisiones	1
2.	Estrategia de doble modelo.	3
3.	Arquitectura cliente-servidor.	5
4.	Patrón Model-View-Controller.	7
5.	One Way Data Binding.	12
6.	Two Way Data Binding.	12
7.	Patrón Singleton.	13
8.	Patrón fachada.	14
9.	Patrón proxy.	15
10.	Paquete "guía clínica".	19
11.	Paquete de Expresiones.	21
12.	Scrum: visión general.	24
13.	El <i>Product Backlog</i>	26
14.	Gestión visual: ítems del <i>Product Backlog</i> sobre la pared.	26
15.	Gestión visual: tareas de un <i>Sprint Backlog</i> sobre la pared.	27
16.	Diagrama de despliegue del sistema.	31
17.	Casos de uso de la aplicación.	33
18.	Arquitectura de alto nivel.	34
19.	Diagrama de actividades: listado guías clínicas.	36
20.	Patrón <i>Composite View</i>	39
21.	Iteración 1: funcionalidad visualización guías clínicas.	42
22.	Guidelines: visualización de las guías disponibles.	62
23.	Description: meta-information de la guía clínica.	63
24.	Definitions: definiciones de la guía.	64
25.	Definitions: elección de arquetipo.	65
26.	Definitions: elección de una instancia de elemento.	66
27.	Definitions: Funciones predicado.	67
28.	Lista de reglas.	68
29.	Edición de reglas: Seleccionar instancia de elemento.	69
30.	Editor de reglas.	70

Índice de cuadros

1.	Clase <i>Guide</i>	19
2.	Clase <i>GuideDefinition</i>	19
3.	Clase <i>ArchetypeBinding</i>	20
4.	Clase <i>ElementBinding</i>	20

5.	<i>Clase Rule.</i>	20
6.	<i>Clase Unary Expression.</i>	21
7.	<i>Clase BinaryExpression.</i>	21
8.	<i>Clase AssignmentExpression.</i>	21
9.	<i>Clase FunctinalExpression.</i>	22
10.	<i>Clase OperatorKind.</i>	22
11.	Funciones permitidas.	22

Resumen

Expresar y compartir contenido de soporte a la decisión clínica (CDS por sus siglas en inglés) de forma automatizada e independiente del idioma y de plataformas técnicas ha sido una tarea compleja durante mucho tiempo. Los dos principales desafíos a abordar para compartir lógica de decisión entre diferentes sistemas son, por un lado, la carencia de modelos compartidos de información clínica y, por otro, la escasez de soporte flexible para los diferentes recursos terminológicos.

Para expresar la lógica que da soporte a la decisión médica se necesita un modelo de representación de guías clínicas. Para ello la comunidad *openEHR*, comunidad virtual que trabaja en el desarrollo de [especificaciones](#) para la interoperabilidad de registros clínicos electrónicos entre sistemas sanitarios, ha desarrollado la especificación del lenguaje GDL (Guideline Definition Language). GDL es un lenguaje formal diseñado para expresar lógica de soporte a la decisión de una forma interoperable, el cual, recientemente, ha pasado a formar parte del núcleo de la plataforma de *openEHR*. Ha sido diseñado para ser independiente de lenguajes naturales y de terminologías clínicas haciendo uso, para ello, de los diseños del [Modelo de Referencia](#) y del [Modelo de Arquetipos](#) de *openEHR*.

El principal objetivo de este proyecto ha sido la elaboración de una herramienta web para la edición de guías clínicas expresadas en lenguaje GDL. La comunidad no disponía de una herramienta de gestión de dichas guías con un enfoque colaborativo como el que puede ofrecer una aplicación web. Lo que se pretende es que sean los propios profesionales sanitarios—los que tienen el conocimiento necesario—los que elaboren las guías clínicas de una manera colaborativa y consensuada, con el fin de poder utilizarlas de manera exitosa en la práctica médica.

El resultado es un software capaz de crear y gestionar guías clínicas en formato GDL, las guías son validadas en el *backend* de la aplicación, denegándose la petición de envío cuando la guía posee algún error de elaboración. Se ha desarrollado una aplicación modular en AngularJS, framework JavaScript de Google para el desarrollo de *aplicaciones web de una sola página* basado en el patrón de arquitectura Modelo-Vista-Controlador (MVC). AngularJS favorece la reusabilidad de componentes, inyección de dependencias y un testeo unitario y *end-to-end* de una forma sencilla y eficiente. Al estar basado en el patrón MVC hay una clara separación de responsabilidades entre la gestión de los datos (modelo), la lógica de la aplicación (controlador) y la presentación de la información (vista). La metodología de desarrollo ágil Scrum ha ayudado a reducir considerablemente los ciclos de desarrollo y ha asegurado que al final de cada ciclo se hubiesen desarrollado ciertas funcionalidades operativas de aplicación.

Palabras clave

Guías clínicas, arquetipos clínicos, sistema de soporte a la decisión, interoperabilidad, AngularJS, REST.

1. Introducción

La toma de decisiones es el proceso mediante el que se realiza la elección de una alternativa para resolver un problema. Está presente cotidianamente tanto en la actividad humana como en la de una organización. La clave del éxito está en tomar buenas decisiones. Es necesario que se cumplan ciertos requisitos para tener éxito tomando decisiones, como son:

- Conocimiento en profundidad el dominio en cuestión
- Experiencia
- Información disponible:
 - abarcable por un conjunto de personas
 - no siempre orientable al proceso analítico
- Uso de metodologías de análisis sencillas e intuitivas

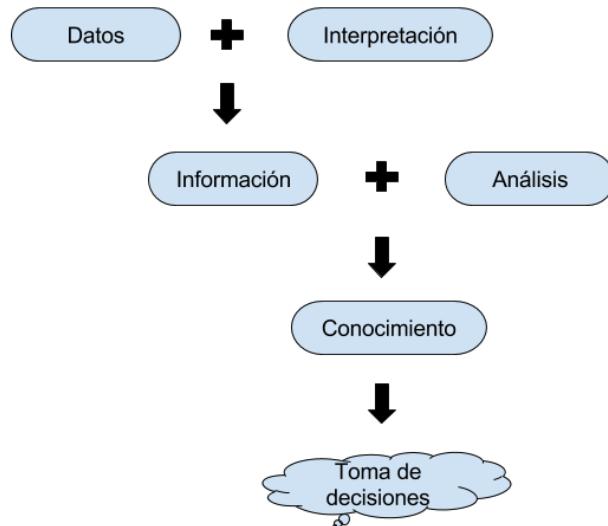


Figura 1: Toma de decisiones.

La toma de decisiones utiliza como materias primas la *información* y el *conocimiento*, ya que sin éstas sería imposible evaluar las opciones existentes o el desarrollo de nuevas alternativas. El ámbito de este TFM es el entorno médico-sanitario, el cual se encuentra sometido constantemente a la toma de decisiones y donde la representación de la información y del conocimiento juega un papel de vital importancia. Para interpretar los datos y poder transformarlos en información relevante es necesario un *sistema de información*. De esta información, con su respectivo proceso de análisis, obtendremos conocimiento de gran valor para la medicina y, por lo tanto, para la sociedad. Además de los sistemas de información, existen sistemas diseñados especialmente para ayudar en el proceso de la toma de decisiones, que se conocen como *sistemas de soporte a decisiones* o *sistemas de apoyo a la decisión*. En este TFM se presenta una herramienta que permite crear guías clínicas que serán utilizadas en este tipo de sistemas.

1.1. Contextualización

El concepto de apoyo a las decisiones es muy antiguo [keen] y ha evolucionado principalmente desde dos áreas de investigación: los estudios teóricos de organización de la toma de decisiones, desarrollados en el Carnegie Institute of Technology a principios de los años 60 y el trabajo técnico sobre sistemas informáticos interactivos, llevados a cabo principalmente en el Instituto Tecnológico de Massachusetts en la década de los 60. Posteriormente, en la década de los 70, los sistemas de apoyo a las decisiones

pasan a convertirse en un campo de investigación como tal. Durante la década de los 80 la investigación en este campo ganó intensidad, surgiendo así los sistemas de información ejecutiva (EIS), los sistemas de apoyo a la decisión en grupo (GDSS) y los sistemas organizacionales de apoyo a la decisión (ODSS). A partir de 1990 el procesamiento analítico en línea (OLAP) y los almacenes de datos fomentaron la ampliación del ámbito de los SSD. Con el cambio de milenio se introdujeron aplicaciones analíticas basadas en web.

Breve historia de los Sistemas de Soporte a la Decisión Clínica (SSDC). En el ámbito de la salud las decisiones clínicas se toman, por lo general, mediante razonamiento deductivo utilizando el conocimiento que tienen los profesionales sanitarios sobre la fisiopatología humana teniendo en cuenta la información disponible para cada caso concreto.

Parece bastante obvio que la experiencia de otros médicos y profesionales de la salud puede ayudar a tomar decisiones. A principios de los años 60 ya se empezaron a dedicar esfuerzos para crear sistemas de soporte a la decisión clínica [bonis]. Incluso una década antes, se trabajó con intensidad en la realización de diagnósticos a partir de la información que los médicos habían almacenado en sus computadoras, se basaban en reglas lógicas básicas. Dos ejemplos clásicos son el sistema de diagnóstico de alteraciones electrolíticas de Bleich [bleich] y el realizado por Warner et al. [warner] basado en la realización de razonamientos probabilísticos mediante el uso intensivo del teorema de Bayes. Hasta ese momento se fundaron lo que fueron las bases teóricas del desarrollo de los SSDC. Durante las dos décadas siguientes se desarrollaron los primeros SSDC basados en grandes ordenadores. Sin embargo no fue hasta los 80 y 90, con la llegada de los ordenadores personales y el uso generalizado de los mismos, cuando se aprovecharon potencialmente este tipo de sistemas. Hoy en día el soporte a la decisión clínica forma parte de muchos sistemas sanitarios, se está invirtiendo mucho esfuerzo por parte de las instituciones médicas y compañías de software para producir SSDC viables para apoyar en todos los aspectos de las tareas clínicas. A pesar de estos esfuerzos por parte de las instituciones para producir y utilizar estos sistemas, todavía no se ha alcanzado la adopción y la aceptación generalizada de la mayoría de las ofertas, y esto es debido principalmente a la escasez de modelos información clínicos compartidos entre sistemas sanitarios.

Las TIC en los sistemas sanitarios. La utilización de las Tecnologías de la Información y las Comunicaciones en sistemas sanitarios se ha convertido en indispensable en los sistemas actuales. Hoy en día todavía estamos en plena transición de una historia clínica convencional a una historia clínica electrónica que introduce una serie de mejoras relacionadas con la inviolabilidad de los datos, la privacidad del paciente, accesibilidad, disponibilidad, mejoras contra la pérdida de información, durabilidad, legibilidad, identificación del profesional clínico, redundancia, errores en la consignación de datos y algo muy importante: la estandarización de datos. Ya en 1990 empieza a surgir la necesidad de representación genérica para la comunicación de registros de historia clínica entre sistemas. La HCE puede almacenar, cuanto menos, la misma información que podría almacenarse en un archivo médico convencional. En los 15 últimos años se han implementado una gran cantidad de proyectos orientados a la eSalud.

Interoperabilidad. Los sistemas de salud actuales requieren para sí una característica fundamental para garantizar la continuidad asistencial¹. Para conseguir esto es inevitable que la información clínica sea representada de una manera interoperable, de modo que diferentes organizaciones puedan hacer uso de dicha información independientemente de quién, cómo y cuándo se haya generado. La continuidad asistencial del paciente es indispensable en multitud de situaciones como cambios de residencia de los pacientes, cambios organizativos, cambios de niveles asistenciales, etc. La manera de conseguir la deseada interoperabilidad es aplicando la normalización. Las normas o estándares proporcionan la base sobre la que poder crear acuerdos con un objetivo en común, en este caso la continuidad asistencial. *openEHR* es una comunidad virtual que trabaja en la estandarización de la Historia Clínica Electrónica (HCE), garantizando una interoperabilidad universal entre todo tipo de sistemas de información clínicos. Como parte de su trabajo, la Fundación *openEHR* cuenta con una plataforma de intercambio de conocimiento clínico denominado *Clinical Knowledge Manager* (CKM). En estos momentos el CKM consiste en un repositorio de arquetipos (modelos formales para la representación de conceptos del dominio médico). Recientemente, la comunidad *openEHR* ha decidido que el lenguaje GDL, lenguaje formal diseñado para representar conocimiento clínico para soporte a la decisión médica, pase a formar parte del núcleo de especificaciones de la plataforma, por lo que las guías clínicas en GDL serán incluidas dentro de la plataforma CKM.

Conseguir la deseada interoperabilidad en entornos sanitarios es una tarea compleja ya que las organizaciones actúan, en cierto modo, como silos de información aislados, es decir, son autónomos a la hora de tomar sus propias decisiones, de implantar los sistemas de información que cada una de ellas quiere. Esto se repite incluso dentro de las propias organizaciones, un ejemplo claro podrían ser los propios servicios dentro de un hospital que, en numerosas ocasiones, utilizan sistemas distintos. Surge, de esta manera, la necesidad de crear una metodología que nos permita representar el conocimiento médico y, de este modo, poder separarlo de la información clínica existente en los sistemas.

¹ **Continuidad asistencial:** concepto que define el proceso, centrado en el paciente, en el cual intervienen diferentes profesionales sanitarios en diferentes períodos de tiempo y lugares con el objetivo compartido de mejorar la calidad de la asistencia prestada.

Elementos para el cambio Lo que diferencia a la medicina de otros entornos es la complejidad del conocimiento clínico y la velocidad con la que éste varía. Esto ocasionó durante mucho tiempo que no tuviésemos un mecanismo para representar el conocimiento ni para gestionarlo correctamente, por lo tanto, lo lógico fue buscar una estrategia que permita gestionar la información clínica con sus características intrínsecas. La mayoría de sistemas actuales se desarrollan de tal forma que los conceptos del dominio médico se encuentran *hardcodeados* al software y a los modelos de bases de datos utilizados lo que impide que dicha información pueda ser compartida con otros sistemas. Es en este punto donde surge la *estrategia del doble modelo*, una estrategia que maneja de manera separada ambos tipos de conceptos: información y conocimiento, de tal manera que ambos puedan coexistir de manera independiente. Con esto se pretende proteger a los sistemas de información ante futuros cambios y delega la gestión del conocimiento a los expertos del dominio. Esta aproximación es utilizada por *openEHR* y por la norma UNE-EN ISO 13606 para la gestión de la información.

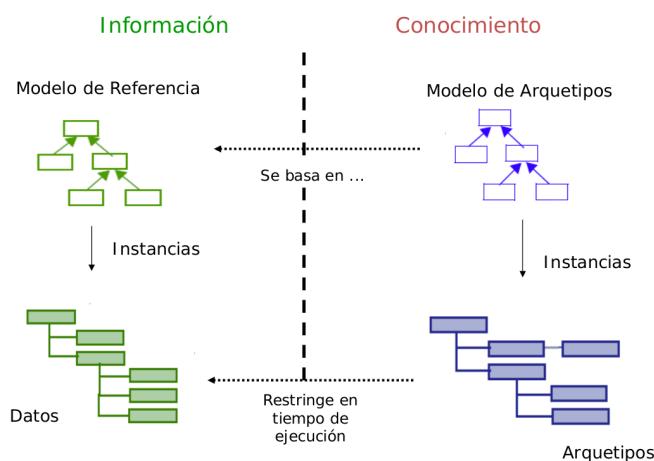


Figura 2: Estrategia de doble modelo.

Este modelo dual permite un acceso multinacional o multiempresarial teniendo en cuenta la necesidad de interacción con otros sistemas:

- Sistemas de soporte a la decisión.
- Sistemas de terminología clínica.
- Sistemas de conocimiento médico (bases de datos conceptuales).
- Sistemas demográficos
- Sistemas de seguridad

Pero el principal aporte es que permite una gestión separada e independiente de dos conceptos fundamentales que son la *información* y el *conocimiento*. La *información* entendida como los datos que no van a variar con el tipo y el *conocimiento* clínico que sí puede, y suele, variar con el tiempo, por lo tanto ofrece una característica importante como es la capacidad de adaptación a nuevos avances en la medicina. Este modelo permite representar la HCE de un paciente, o parte de ella, de manera consistente.

En la Figura 2 se muestra una representación del modelo dual, que permite que los sistemas se adapten a los cambios que se producen en el dominio clínico conservando intacta la información almacenada en los sistemas y favoreciendo su mantenimiento. Ya que la *información* y el *conocimiento* tienen naturalezas diferentes, cualquier cambio producido en el conocimiento afectaría en gran medida a la información, que no debe cambiar, por lo que este modelo hace una clara separación de ambos niveles. Las principales ventajas de esta aproximación son:

- Interoperabilidad a nivel de conocimiento: la evolución del conocimiento puede ser compartida.
- Consulta y recuperación de la información eficaces: debido al conocimiento a priori de la estructura de almacenada.
- Separación de tareas y responsabilidades: los modelos técnicos son desarrollados por ingenieros de software mientras que el dominio clínico lo elaboran los médicos, los verdaderos conocedores del mismo.
- Los sistemas pueden evolucionar y cambiar con naturalidad, lo que facilita el mantenimiento a largo plazo.

En la parte de la izquierda de la [Figura 2](#) se representa la *información* clínica, hechos u opiniones de o referidos a entidades específicas y que no varían con el paso del tiempo (*e.g.* El día 1 de enero el paciente X tenía una presión arterial de 120/80 mmHg.). El *conocimiento*, representado en la parte derecha, comprende hechos acumulados a lo largo del tiempo, procedente de muchas fuentes, que son verdad para todas las instancias de las entidades del dominio y que pueden variar con el paso del tiempo (*e.g.* La medida de la presión arterial consta de dos valores: la presión arterial sistólica y la presión arterial diastólica). Para cada uno de estos dos niveles se tiene un modelo de datos. La *información*, los datos que se almacenan en el sistema, se componen a partir de los elementos y la organización del Modelo de Referencia (MR). Este modelo necesita reflejar la estructura y la organización jerárquica de la HCE con el fin de ser fiel al contexto clínico original. El MR representa el conjunto de clases que forman los componentes básicos de cualquier HCE, además de la información jerarquizada de estas clases. Cada parte proporciona significado semántico claro a la hora de intercambiar HCE entre distintos sistemas heterogéneos. El objeto de información por antonomasia en este tipo de estándares es el extracto clínico. El *conocimiento* representa los conceptos del dominio clínico, de manera que permita representar los datos y se representa mediante el Modelo de Arquetipos. Un arquetipo clínico es una estructura de datos utilizada para representar conceptos del dominio clínico.

Los pilares básicos de este tipo de estándares son 3:

- Un vocabulario común para codificar términos: terminologías clínicas.
- Un mecanismo para representar las estructuras de datos: arquetipos.
- Un mecanismo para representar el contexto: modelo de referencia.

El modelo de arquetipos define el modelo que permite expresar arquetipos para cualquier MR, restringiéndolo (fijando nombres, tipos de datos, valores por defecto, cardinalidades, etc.) para modelar formalmente conceptos de un dominio del conocimiento. Los arquetipos son acordados en una comunidad con el objetivo de garantizar la interoperabilidad semántica, la consistencia de la información y la calidad de los datos. Este modelo consta de una serie de paquetes que se verán en el [capítulo 3](#). Los arquetipos son elementos jerárquicos formados por una serie de nodos utilizando el patrón de diseño Composite, de manera que un nodo pueda contener de manera recursiva otros nodos.

1.2. Objetivos

El objetivo principal de este proyecto ha consistido en el desarrollo de una aplicación web para la gestión de guías clínicas basadas en arquetipos y expresadas en lenguaje GDL. Al estar GDL basado en una norma de interoperabilidad, esta herramienta permitirá a los profesionales sanitarios generar conocimiento clínico interoperable. Además, al hacer uso de las tecnologías web y al ser integrada en el CKM de *openEHR*, permitirá hacerlo de una manera colaborativa y consensuada.

Hasta el desarrollo de la aplicación objeto de este Trabajo Fin de Máster, se disponía de una herramienta basada en tecnología Java Swing que permite editar y ejecutar guías clínicas en GDL. Se trata de un software de escritorio independiente de la plataforma CKM, que permite editar las guías de manera off-line. Por tratarse de un software stand-alone que no se encuentra integrado en CKM, representa una limitación en la colaboración e intercambio de conocimiento clínico entre diferentes usuarios. Por estos motivos, se demanda una herramienta de gestión de guías clínicas con un enfoque colaborativo, además de proporcionar una usabilidad más adecuada, accesible y modular que la existente actualmente. Esta herramienta estará preparada para ser integrada en la plataforma CKM y supondrá el punto de partida hacia una serie de nuevas funcionalidades de gran utilidad para la comunidad, como la edición colaborativa en tiempo real, la resolución de conflictos, la gestión de historial y de versiones, etc. La aplicación se desarrollará siguiendo un enfoque de metodología ágil, adaptado a las características y requerimientos de un Trabajo de Fin de Máster.

Objetivos específicos. Como objetivos específicos, se plantean los siguientes:

- La aplicación desarrollada permitirá crear, editar y actualizar ficheros GDL a través de un navegador web.
- La aplicación desarrollada deberá estar preparada para ser integrada con el CKM de *openEHR* (www.openehr.org/ckm). La aplicación se diseñará de manera que sus componentes puedan ser fácilmente reutilizados y de forma que resulte sencillo incorporar nuevas funcionalidades.
- Se diseñará una interfaz de usuario sencilla e intuitiva, teniendo en cuenta aspectos de usabilidad, de forma que el proceso de gestión de las guías clínicas resulte fácilmente asimilable por los expertos en el dominio clínico y disfruten de una experiencia agradable durante el uso de la herramienta.

2. Fundamentos tecnológicos

2.1. Aplicaciones web

En la actualidad millones de negocios usan Internet como canal de comunicaciones de bajo coste. Internet les permite intercambiar información con su mercado objetivo y realizar transacciones de una manera rápida y sencilla. Sin embargo un acuerdo efectivo solamente es posible cuando el negocio es capaz de capturar y almacenar todos los datos necesarios y tener un medio que permita el procesamiento de dicha información y la presentación de los resultados al usuario.

En la ingeniería de software se denomina aplicación web [web-app] a aquellas herramientas que los usuarios pueden utilizar accediendo a un servidor web a través de Internet o de una intranet mediante un navegador. Las aplicaciones web son populares debido a lo práctico del navegador web como cliente ligero, a la independencia del sistema operativo, así como a la facilidad para actualizar y mantener aplicaciones web sin distribuir e instalar software a miles de potenciales usuarios. Existen aplicaciones como los *webmails*, *wikis*, *weblogs*, tiendas en línea, buscadores etc. que son ejemplos bien conocidos de aplicaciones web.

Es importante mencionar que una página web puede contener elementos que permiten una comunicación activa entre el usuario y la información. Esto permite que el usuario acceda a los datos de modo interactivo, gracias a que la página responderá a cada una de sus acciones, como por ejemplo llenar y enviar formularios, participar en juegos diversos y acceder a gestores de base de datos de todo tipo. Las aplicaciones web usan una combinación de scripts de servidor (PHP, ASP o cualquier backend desarrollado en otros lenguajes de programación) para gestionar el almacenamiento y la recuperación de la información, y scripts del lado de cliente (Javascript, CSS y HTML) para presentar la información al usuario. Esto permite a los usuarios la interacción con el backend por medio de formularios, sistemas de gestión de contenidos, carritos de la compra, etc. Además de permitir la creación de documentos, la compartición de información colaboración en proyectos y trabajar en documentos compartidos independientemente de la localización y de los dispositivos de los usuarios.

Las aplicaciones web encajan dentro de las aplicaciones cliente-servidor, que siguen un modelo de aplicación distribuida. Las aplicaciones web se caracterizan por estar alojada en un servidor web en una URL determinada.

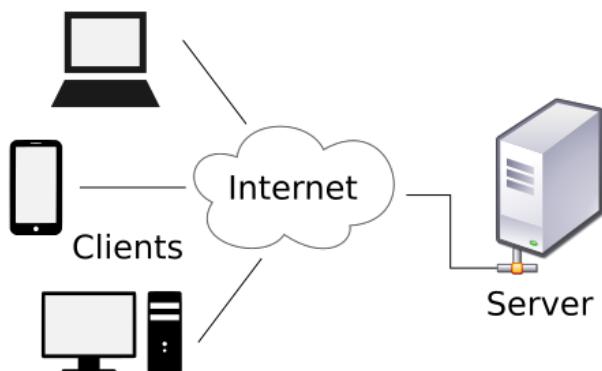


Figura 3: Arquitectura cliente-servidor.

En este tipo de arquitecturas la entidad que realiza solicitudes se denomina cliente. Las características de un cliente son:

- Es quien inicia una solicitud o petición a un servidor, por lo tanto asumen un rol activo en la comunicación.
- Se queda a la espera y recibe la respuesta del servidor.
- Se puede conectar con varios servidores a la vez por lo general.
- Suele utilizar una *interfaz gráfica de usuario* como media de interacción con el servidor.

A la entidad receptora de dichas solicitudes se le denomina servidor, cuyas características se resumen a continuación:

- Están a la espera de solicitudes por parte de los clientes, por lo tanto asumen un rol pasivo en la comunicación.
- Reciben una solicitud, la procesa y, posteriormente, envía la respuesta al cliente.

- Por lo general, acepta un número elevado de conexiones por parte de diferentes clientes.

Las aplicaciones web poseen ventajas significativas con respecto a otro tipo de aplicaciones. Al ejecutarse en un navegador web, proporcionan una gran compatibilidad entre plataformas (portabilidad) y deberían funcionar independientemente de la versión del sistema operativo instalado en el cliente. Además no requieren que el usuario realice actualizaciones, ya que estas son implementadas en el lado del servidor. Por último, este tipo de aplicaciones requieren poco o nada de espacio libre en disco, suelen ser bastante livianas.

Sin embargo también tiene algunas desventajas que, como se verá, tiene solución relativamente sencilla. Requieren navegadores web totalmente compatibles para funcionar, aunque normalmente esto se resuelve actualizando la versión de los navegadores. En numerosas ocasiones requieren que exista una conexión a Internet, que si es interrumpida, puede que la aplicación no funcione. Esto, dependiendo de la aplicación, se podría abordar implementando soluciones que permitan el funcionamiento offline.

2.2. Node.js

Node.js es un entorno de ejecución multiplataforma, de código abierto, para JavaScript construido con el motor de JavaScript V8 de Chrome [\[node\]](#). Node.js usa un modelo de operaciones E/S sin bloqueo y orientado a eventos, que lo hace liviano y eficiente. El ecosistema de paquetes de Node.js, *npm*, es el ecosistema más grande de librerías de código abierto en el mundo en estos momentos.

Para bien o para mal, JavaScript es el lenguaje de programación más popular para la Web y, debido al alcance de ésta, JavaScript ha cumplido con el sueño de "escribir una vez, ejecutar en cualquier parte" que tenía Java en la década de los noventa.

Desde el lanzamiento de Google Chrome a finales de 2008, el rendimiento de JavaScript ha mejorado a un ritmo increíblemente rápido debido [\[cantelon\]](#) a la fuerte competencia entre los proveedores de navegadores (Mozilla, Microsoft, Apple, Opera y Google). El rendimiento de estas modernas máquinas virtuales JavaScript literalmente está cambiando los tipos de aplicaciones que puede crear en la web². Un ejemplo convincente y francamente alucinante de esto es jslinux³, un emulador de PC que se ejecuta en JavaScript donde puede cargar un kernel de Linux, interactuar con la sesión de terminal y compilar un programa C, todo en su navegador.

Node usa V8, la máquina virtual que utiliza Google Chrome, para la programación del lado del servidor. V8 da a Node un gran impulso en el rendimiento, ya que elimina a los intermediarios, prefiriendo la compilación directa a código nativo sobre la ejecución de *bytecode* o el uso de un intérprete. Dado que Node utiliza JavaScript en el servidor, también hay otros beneficios:

- Los desarrolladores pueden escribir aplicaciones web en un solo lenguaje, lo que ayuda al reducir el cambio de contexto entre el desarrollo del cliente y del servidor y permitir el intercambio de código entre ambos, reutilizar el mismo código para la validación de formularios o cualquier lógica de negocio.
- JSON es un formato de intercambio de datos muy popular hoy en día y es nativo de JavaScript.
- JavaScript es el lenguaje utilizado en varias bases de datos NoSQL (como CouchDB y MongoDB), por lo que la integración con ellos se realiza de forma natural (por ejemplo, el shell y el lenguaje de consulta de MongoDB es JavaScript, el paradigma *map/reduce* de CouchDB es JavaScript).
- JavaScript es un lenguaje *target*, en el sentido de que ya hay una serie de lenguajes que se compilan a Javascript⁴.
- Node usa una máquina virtual (V8) que se mantiene actualizada con el estándar ECMAScript⁵. En otras palabras, no se tiene que esperar a que todos los navegadores se pongan al día para usar las nuevas características de lenguaje JavaScript en Node.js.

¿Quién diría que JavaScript terminaría siendo un lenguaje convincente para escribir aplicaciones del lado del servidor? Sin embargo, debido a su alcance, rendimiento y otras características mencionadas anteriormente, Node ha ganado mucho terreno. Aunque JavaScript es sólo una pieza del rompecabezas, la manera en la que Node utiliza JavaScript es aún más convincente.

² Ver la página *Chrome Experiments* para ver algunos ejemplos: <http://www.chromeexperiments.com/>.

³ JsLinux, un emulador JavaScript para PC: <http://bellard.org/jslinux/>.

⁴ Ver la *Lista de lenguajes que se compilan a JavaScript*: <https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS>.

⁵ Para más información sobre el estándar ECMAScript, ver Wikipedia: <http://en.wikipedia.org/wiki/ECMAScript>

2.3. AngularJS y patrón de arquitectura Model-View-Controller

AngularJS es un framework basado en JavaScript de código abierto, mantenido por Google y por una amplia comunidad de individuos y organizaciones para la solución de muchos de los desafíos encontrados en el desarrollo de *aplicaciones web de una sola página*. Lo que se pretende con este framework es simplificar el desarrollo y las pruebas de este tipo de aplicaciones proporcionando un framework Model-View-Controller (MVC) y Model-View-Viewmodel (MVVM) para las arquitecturas del lado cliente.

La principal idea detrás del patrón MVC es que se tiene una separación clara en el código entre la gestión de los datos (modelo), la lógica de la aplicación (controlador) y la presentación de datos al usuario (vista). Las aplicaciones MVC surgieron ya en los años 70 de la mano de Smalltalk. A partir de entonces se fueron convirtiendo cada vez más populares en casi todos los entornos de desarrollo donde estuvieran involucradas las interfaces de usuario. Hasta hace relativamente poco tiempo, era prácticamente ajeno al desarrollo web.

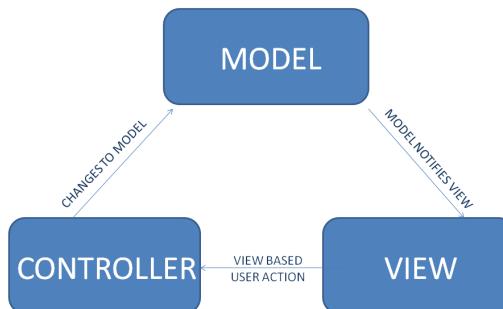


Figura 4: Patrón Model-View-Controller.

En este patrón de arquitectura, la vista obtiene los datos del modelo para ser mostrados al usuario. Cuando un usuario interactúa con la aplicación haciendo clic con el ratón o escribiendo un *input*, el controlador responde cambiando los datos en el modelo. Finalmente, el modelo notifica a la vista que se ha producido un cambio para que pueda actualizar lo que se está mostrando. En las aplicaciones Angular, la vista es el [Document Object Model \(DOM\)](#)⁶, los controladores son clases JavaScript y los datos del modelo se almacenan en propiedades de objetos. MVC es apropiado por varios motivos.. En primer lugar ofrece un modelo mental que indica dónde colocar las cosas, de tal manera que no hay que reinventar la rueda continuamente. Los colegas que colaboran en un mismo proyecto tendrán una ventaja de antemano al entender lo que ya se hubiese escrito, ya que ellos sabrán que se está utilizando la estructura MVC para organizar el código. Y quizás lo que es más importante, ofrece grandes beneficios ya que hace que las aplicaciones sean más fáciles de extender, mantener y probar.

2.3.1. Módulos en AngularJS

Un [módulo\[module\]](#) en AngularJS es un contenedor para las diferentes partes de una aplicación, como controladores, servicios, filtros, directivas, información sobre configuración, etc. Es una manera de agrupar funcionalidades. La mayoría de aplicaciones en otros lenguajes de programación como Java o incluso Python tienen un método *main* que instancia y vincula todas las partes de la aplicación. Las aplicaciones Angular no tienen dicho método. En vez de ello, tiene el concepto de módulos, que especifican de forma declarativa las dependencias de la aplicación y cómo ésta debe ser inicializada. Esta aproximación tiene varias ventajas:

- El proceso es declarativo. Esto significa que está escrito de tal manera es que más sencillo de escribir y de entender.
- Es modular. Ello fuerza a que se piense en cómo se van a definir los componentes y dependencias de la aplicación. Se puede empaquetar código como módulos reutilizables.
- Dichos módulos pueden ser cargados en cualquier orden—incluso en paralelo—ya que los módulos pueden diferir la ejecución.

⁶ Especificación técnica de w3c: <https://www.w3.org/DOM/DOMTR>

- Facilitan las pruebas, ya que los tests unitarios sólo tienen que cargar los módulos necesarios, lo cual hace que dichos tests sean muy rápidos.
- Los tests *end-to-end* pueden usar módulos para sobreescribir la configuración.

En AngularJS un módulo se declara de una manera muy sencilla.

```
1 // declare a module
2 var myAppModule = angular.module('myApp', []);
3
4 // configure the module.
5 // in this example we will create a greeting filter
6 myAppModule.filter('greet', function() {
7   return function(name) {
8     return 'Hello, ' + name + '!';
9   };
10});
```

Dicho módulo se utilizaría en la vista declarándolo con la *directiva* `ng-app`. Esta *directiva* le dice a Angular que inicie la aplicación usando el módulo `myApp`.

```
1 <div ng-app="myApp">
2   <div>
3     {{ 'World' | greet }}
4   </div>
5 </div>
```

Estos módulos se deben de definir de una manera organizada. Es recomendable tener módulos separados para los controladores, servicios, filtros, directivas, etc. Para el desarrollo de esta aplicación se ha tenido en cuenta una [guía de buenas prácticas](#) muy utilizada en la comunidad de desarrollo de AngularJS. En módulo principal podría entonces simplemente declarar el resto de módulos como dependencias, tal como se ha hecho en el desarrollo de esta aplicación. Esto hace que sea más sencilla la gestión de nuestros módulos, ya que se convierten en piezas aisladas de código, teniendo cada uno de ellos una, y sólo una responsabilidad. Esto también permite que los tests carguen el módulo (o módulos) que interesen, haciendo que sean más rápidos y que se focalicen en la funcionalidad a ser probada.

2.3.2. Templates

Las *aplicaciones web multi-página* generan su HTML ensamblándolo y uniéndolo con datos del servidor y, una vez hecho esto, se envían las páginas al navegador. Muchas *aplicaciones web de una sola página* también hacen esto de alguna manera. Angular es diferente en este sentido ya que la plantilla (el HTML) y los datos se envían al navegador para ser ensamblados allí. El servidor sólo sirve recursos estáticos: las plantillas y los datos requeridos por dichas plantillas para que se puedan mostrar de forma apropiada. Una vez recibida la plantilla en el navegador web, Angular expande dichas plantillas en la aplicación poblando la plantilla con los datos pertinentes.

El flujo básico de este procedimiento es el siguiente:

1. Un usuario pide la primera página de la aplicación.
2. El navegador del usuario hace una conexión HTTP al servidor y carga la página `index.html` que contiene la plantilla.
3. Angular se carga en la página, espera a que la página esté totalmente cargada y busca una directiva `ng-app` para definir su alcance.
4. Angular recorre la plantilla buscando directivas y vinculaciones (two-way databinding). El resultado de este proceso es que se registran los *listeners* y se hacen las manipulaciones del DOM necesarias, además de la recuperación de los datos iniciales desde el servidor. El resultado final es que se inicializa la aplicación y la plantilla se convierte en una vista como un DOM.
5. A partir de este punto, el usuario se conectará con el servidor para cargar la información adicional necesaria a medida que la vaya requiriendo el usuario.

Estructurar una aplicación con Angular permite que las plantillas se mantengan separadas de los datos que las pueblan, por lo tanto las plantillas son cacheables. Despues de la primera carga, sólo se necesita que se envíen los nuevos datos del servidor al cliente, con lo que resulta en un mejor rendimiento de la aplicación.

2.3.3. Controladores

Los controladores, en AngularJS, son objetos que permiten desarrollar la lógica de negocio de la aplicación. Enlazan el ámbito `$scope` con la vista y permite tener un control total de los datos. Este enlace con la vista se realiza a través de una directiva `ng-controller`. Un módulo necesita como mínimo un controlador para funcionar. Un controlador es una parte funcional de un módulo, con sus propios atributos, modelos y métodos. Para declarar un controlador se utiliza el método `controller()` del objeto `module` visto anteriormente. Este método acepta, como mínimo, un parámetro llamado `$scope`, que representa el alcance de variables, modelos y métodos del controlador.

```
1 var app = angular.module("MyApp", []);
2 app.controller("mainController", function($scope) {
3     // contenido del controlador
4 }) ;
```

Tanto el método `angular.module()` como el método `controller()` devuelven una referencial al módulo, es posible concatenar métodos. Como se puede observar, en el snippet anterior, declarar un controlador es bastante sencillo, tan solo basta con asignarle un nombre e injectar las dependencias necesarias, podemos injectar otros componentes dentro del controlador, sean nativos de AngularJS u otros componentes creados por nosotros.

Angular hace uso intensivo de la inyección de dependencias en todos sus componentes, para injectar nuevas dependencias simplemente se le pasan como parámetros al controlador separados por comas.

```
1 app.controller('mainController', function($scope, servicio, factorial) {
2     //contenido
3 }) ;
```

Tal como se ha comentado anteriormente, en la vista se usa la directiva `ng-controller` para asociar el controlador definido a la vista. El controlador debemos añadirlo en el nivel de anidamiento adecuado en el cual se quiere que tenga alcance.

```
1 <body>
2     <div ng-controller="mainController">
3         <h1>Hola AngularJS</h1>
4         <div/>
5     </body>
```

Dentro del alcance donde se haya declarado el controlador se tiene un objeto `$scope` que permite controlar los datos. A continuación se muestra un sencillo ejemplo de cómo se pueden mostrar los datos del controlador en la vista.

```
1 app.controller('mainController', function($scope) {
2     $scope.saludo = "AngularJS";
3 }) ;
```

```
1 <body>
2     <div ng-controller="mainController">
3         <h1>Hola {{saludo}}</h1>
4         <div/>
5     </body>
```

Se puede observar como en el controlador se ha creado la variable `saludo` dentro del `$scope`, al cual se le ha asignado un valor. Por otra parte en la vista podemos ver cómo se accede y se muestra la variable. Para hacer referencia a las variables del `$scope` se utiliza el *double curly* "`{ }{ }`" que trae incorporado Angular en su propio motor de plantillas.

En resumen, se puede decir que los módulos y controladores en AngularJS son componentes que nos permiten desacoplar el código, englobar funcionalidades y tener un código más limpio. Los controladores extienden o construyen el `$scope`, el cual se encarga de contener los datos y de transferirlos de la vista al controlador y viceversa.

2.3.4. Directivas

Las directivas[directive] son, desde un punto de vista de alto nivel, *marcadores* en un elemento del DOM (*i.e.* un atributo, un nombre de elemento, un comentario o una clase CSS) que le informan al compilador de Angular (`$compile`) que vincule un

comportamiento especial a dicho elemento del DOM (*e.g.* a través de escuchadores de eventos), o incluso que transforme el elemento del DOM y sus elementos hijos.

Angular vienen incorporado con un conjunto de directivas, como ngBind, ngModel o ngClass. Del mismo modo que es posible crear controladores o servicios propios, también se pueden crear directivas. Cuando AngularJS inicializa una aplicación, el compilador HTML recorre el DOM emparejando directivas con elementos del DOM.

Para poder crear directivas es importante saber cómo el compilador de AngularJS determina cuándo usar una directiva. Similar a la terminología usada cuando un **elemento se corresponde con un selector**, se puede decir que un elemento **se corresponde con** una directiva cuando la directiva es parte de su declaración. En el siguiente ejemplo se puede decir que el elemento <input> se corresponde con la directiva ngModel.

```
1 <input ng-model="foo">
```

El elemento <input> de a continuación también encaja con ngModel.

```
1 <input data-ng-model="foo">
```

Y el siguiente elemento <person> se corresponde con la directiva person.

```
1 <person>{{name}}</person>
```

Normalización. AngularJS normaliza el *tag* de un elemento y los nombres de atributos para determinar qué elementos encajan con qué directivas. En terminología AngularJS, se suele hacer referencia a las directivas por su nombre normalizado usando notación *camelCase* (*e.g.* ngModel). Sin embargo, ya que HTML no es *case-sensitive*, se suele hacer referencia a las directivas en el DOM usando notación *lower-case*, normalmente usando atributos **dash-delimited** en los elementos del DOM (*e.g.* ng-model). El proceso de normalización es el siguiente:

1. Eliminar x- y data- del inicio de los elementos/atributos.
2. Convertir el :, - o los nombres delimitados por _ a *camelCase*.

Por ejemplo, las siguientes formas son equivalentes y todas ellas se corresponden con la directiva ngBind.

```
1 <div ng-controller="Controller">
2   Hello <input ng-model='name'> <hr/>
3   <span ng-bind="name"></span> <br/>
4   <span ng:bind="name"></span> <br/>
5   <span ng_bind="name"></span> <br/>
6   <span data-ng-bind="name"></span> <br/>
7   <span x-ng-bind="name"></span> <br/>
8 </div>
```

Tipos de directivas. \$compile puede emparejar directivas basadas en nombres de elementos (E), atributos (A), nombres de clases (C) y comentarios (M). Las directivas nativas de Angular muestran en su documentación qué tipo de emparejamiento soportan. A continuación se muestran las diferentes maneras en las que una directiva (myDir en este caso) que cumple los cuatro tipos mencionados puede ser referenciada desde una plantilla. Una directiva puede especificar qué tipos soporta en su propiedad restrict de su definición. Si no se indica ninguno, por defecto soporta los tipos EA.

```
1 <my-dir></my-dir>
2 <span my-dir="exp"></span>
3 <!-- directive: my-dir exp -->
4 <span class="my-dir: exp;"></span>
```

Crear directivas propias. Al igual que los controladores, las directivas se registran en los módulos. Para registrar una directiva se usa la APU module.directive, que recibe el nombre de la directiva normalizada seguida de una función factoría. Esta función factoría debería de devolver un objeto con las opciones que le digan a \$compile cómo se debería de comportar la directiva cuando esté activa. La función factoría se invoca sólo una vez cuando el compilador encuentra la directiva por primera vez. Se puede realizar cualquier trabajo de inicialización en este momento. La función se invoca usando \$injector.invoke lo cual la hace inyectable como si de un controlador se tratase. De manera análoga a la declaración de un controlador, una directiva se declara de la siguiente manera:

```
1 var app = angular.module("myApp", []);
2 app.directive("myDirective", function() {
3     return {
4         template : "<h1>Made by a directive!</h1>"
5     };
6 }) ;
```

Tal como se ha descrito con anterioridad, una vez creada la directiva se puede invocar mediante el nombre de un elemento, a través de un atributo, usando una clase y mediante un comentario. Los siguientes 4 *snippets* producirían el mismo resultado.

Nombre de un elemento:

```
<my-directive></my-directive>
```

Atributo:

```
<div my-directive></div>
```

Clase:

```
<div class="my-directive"></div>
```

Comentario:

```
<!-- directive: my-directive -->
```

2.3.5. Objeto \$scope de AngularJS

El objeto **scope** es un objeto que hace referencia al modelo de la aplicación. Se trata de un contexto de ejecución para **expresiones**. Tienen una estructura jerarquizada que imita a la estructura del DOM de la aplicación. Los *scopes* pueden observar expresiones y propagar eventos.

Características del scope. Los *scopes* proporcionan APIs (**\$watch**) para observar cambios en el modelo. También proporcionan APIs ([https://docs.angularjs.org/api/ng/type/\\$rootScope.Scope#\\$apply\[\\$apply\]](https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$apply[$apply])) para propagar cualquier cambio en el modelo de la aplicación en la vista desde fuera del ámbito de AngularJS (controladores, servicios, controladores de eventos de Angular, etc.).

Los *scopes* se pueden anidar para limitar el acceso a las propiedades de los componentes de la aplicación mientras se proporciona acceso a propiedades compartidas del modelo. Los *scopes* anidados son o bien "scopes hijos" o "scopes aislados". Los primeros heredan las propiedades de su *scope* padre mientras que los segundos no lo hacen. Los *scopes* proporcionan un contexto contra el que se evalúan las expresiones (*i.e.* la expresión `{ {username} }` no tiene ningún significado a no ser que se evalúe contra un *scope* específico donde se defina el valor de la propiedad `username`).

El scope como modelo de datos. El *scope* es el nexo de unión entre el controlador y la vista de la aplicación. Durante la fase de **linkado** las directivas establecen expresiones **\$watch** en el *scope*. El **\$watch** permite que las directivas sean notificadas cuando se produce algún cambio en una propiedad, lo que permite a la directiva renderizar el valor actualizado en el DOM. Tanto los controladores como las directivas hacen referencias al *scope*, pero no entre ellas. Este acuerdo aísla el controlador de la directiva y del DOM.

2.3.6. Two-way databinding

La vinculación de datos [twdb] en las aplicaciones AngularJS es la sincronización automática de datos entre los componentes del modelo y de la vista. La forma en que AngularJS implementa la vinculación de datos permite tratar el modelo como la única fuente de datos en la aplicación. La vista es una proyección del modelo en todo momento. Cuando el modelo cambia, la vista refleja dicho cambio y viceversa.

Databinding en sistemas de plantillas clásicos. La mayoría de los sistemas de plantillas enlazan datos en una sola dirección: incorporan los componentes de plantilla y de modelo juntos en una vista. Después de que se produzca esta fusión, los cambios

del modelo o de las secciones relacionadas de la vista no se reflejan automáticamente en la vista. Peor aún, los cambios que el usuario hace a la vista no se reflejan en el modelo. Esto significa que el desarrollador tiene que escribir código que sincroniza constantemente la vista con el modelo y el modelo con la vista.

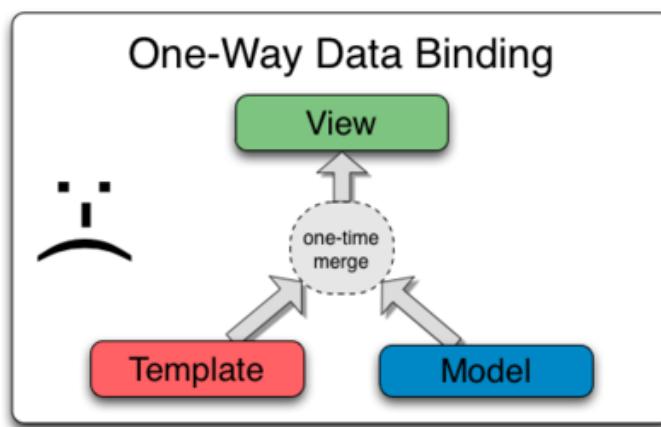


Figura 5: One Way Data Binding.

Databinding en el sistema de plantillas de AngularJS. Las plantillas de AngularJS funcionan de manera diferente. En primer lugar, la plantilla (que es el HTML sin compilar junto con cualquier marcado adicional o directivas) se compila en el navegador. El proceso de compilación produce una vista en vivo. Cualquier cambio en la vista se refleja inmediatamente en el modelo y cualquier cambio en el modelo se propaga a la vista. El modelo es una representación fiel de los datos para el estado de aplicación, simplificando enormemente el modelo de programación para el desarrollador. Se puede pensar en la vista como una simple proyección instantánea del modelo.

Debido a que la vista es sólo una proyección del modelo, el controlador está completamente separado de la vista y no tiene conocimiento de ello. Esto hace que la prueba sea un complemento porque es fácil probar su controlador de forma aislada sin la vista y la dependencia DOM / navegador relacionada.

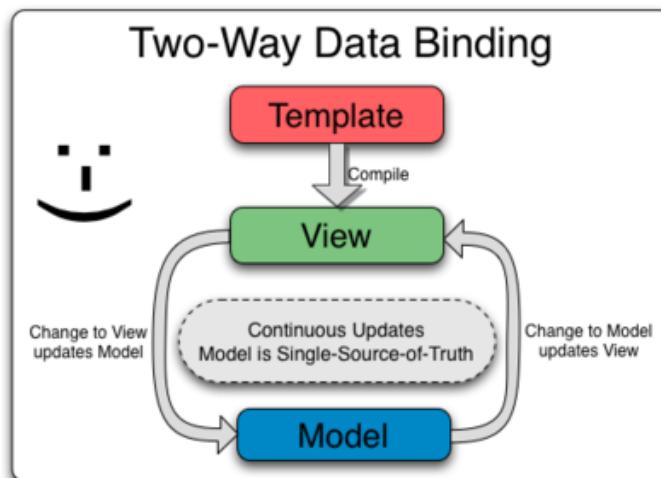


Figura 6: Two Way Data Binding.

2.4. Patrones de diseño

2.4.1. Patrón Singleton

El patrón singleton es un patrón de diseño que restringe la instanciación de una clase a un único objeto y proporciona un punto de acceso global a dicha instancia. Esto es útil cuando se necesita exactamente un objeto para coordinar acciones en todo el sistema. El concepto a veces se generaliza a sistemas que operan más eficientemente cuando sólo existe un objeto, o que restringen la instanciación a un cierto número de objetos. A continuación se muestra el diagrama UML del patrón de diseño *Singleton*.

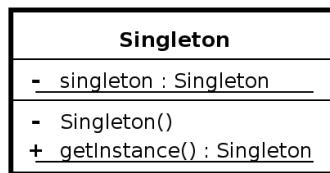


Figura 7: Patrón Singleton.

Cuando un componente requiere una dependencia, AngularJS lo resuelve utilizando el siguiente algoritmo:

1. Coge su nombre y hace una búsqueda en un *hashmap* que se define en un *closure* léxico, por lo que tiene una visibilidad privada.
2. Si la dependencia existe, AngularJS los pasa como parámetro al componente que lo requiere.
3. Si la dependencia no existe:
 - a. AngularJS lo instancia llamando al *factory method* de su *provider* (*i.e.* `$get`). Se debe tener en cuenta que instanciar la dependencia puede requerir una llamada recursiva al mismo algoritmo para resolver todas las dependencias requeridas por la dependencia dada. Este proceso puede llevar a una dependencia circular.
 - b. AngularJS lo cachea dentro del *hash map* mencionado anteriormente.
 - c. AngularJS lo pasa como parámetro al componente que lo requiere.

Se puede observar mejor el código fuente de AngularJS, que implementa el método `getService`:

```

1 function getService(serviceName) {
2   if (cache.hasOwnProperty(serviceName)) {
3     if (cache[serviceName] === INSTANTIATING) {
4       throw $injectorMinErr('cdep', 'Circular dependency found: {0}', path.join(' <- '));
5     }
6     return cache[serviceName];
7   } else {
8     try {
9       path.unshift(serviceName);
10      cache[serviceName] = INSTANTIATING;
11      return cache[serviceName] = factory(serviceName);
12    } catch (err) {
13      if (cache[serviceName] === INSTANTIATING) {
14        delete cache[serviceName];
15      }
16      throw err;
17    } finally {
18      path.shift();
19    }
20  }
21 }
  
```

Los *servicios* y las *factorías* en Angular son singlets, ya que no se instancian más de una vez. Se puede considerar la memoria caché como un gestor de *singleton*. Existe una ligera variación con el diagrama mostrado anteriormente ya que, en lugar de mantener la referencia privada estática al singleton dentro de su constructor, se mantiene dentro del gestor de *singleton* (indicado en el snippet anterior como *cache*).

2.4.2. Patrón Fachada

El patrón fachada es un patrón estructural. Se utiliza cuando se necesita proporcionar una interfaz simple de alto nivel para un subsistema complejo, o cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel. Aplicando este patrón se reduce la complejidad, minimiza las comunicaciones y dependencias entre subsistemas y desacopla un sistema de sus clientes, haciéndolo más independiente, portable y reutilizable.

Es una buena práctica en AngularJS, y en el desarrollo de software en general, situar la lógica de negocio fuera de los controladores, en servicios separados. Esto permite que el código sea más sencillo de probar y más proporciona una maneras más sencilla para el desarrollador de averiguar qué está sucediendo.

El patrón fachada se ha utilizado con frecuencia en este proyecto con el fin de mantener el código lo más limpio posible y buscando una reutilización de las funcionalidades más utilizadas, como por ejemplo el acceso a cada una de las partes de una guía clínica o para la interacción con el servidor. En este último caso se ha utilizado el patrón fachada para ocultar los detalles de la implementación de comunicación con el servidor.

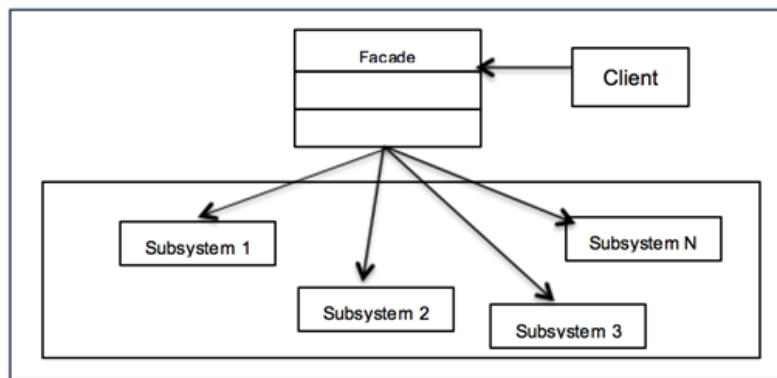


Figura 8: Patrón fachada.

En AngularJS se suelen utilizar con asiduidad las fachadas. Cada vez que se quiere proporcionar una API de alto nivel a una funcionalidad dada se suele crear una fachada. Por ejemplo, a continuación se muestra cómo podemos hacer una petición XMLHttpRequest POST:

```

1 var http = new XMLHttpRequest(),
2     url = '/example/new',
3     params = encodeURIComponent(data);
4 http.open("POST", url, true);
5
6 http.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
7 http.setRequestHeader("Content-length", params.length);
8 http.setRequestHeader("Connection", "close");
9
10 http.onreadystatechange = function () {
11     if(http.readyState == 4 && http.status == 200) {
12         alert(http.responseText);
13     }
14 }
15 http.send(params);
  
```

Pero si se quiere hacer un POST de esos mismos datos usando el servicio \$http de AngularJS, se puede hacer lo siguiente:

```

1 $http({
2   method: 'POST',
3   url: '/example/new',
4   data: data
5 })
6 .then(function (response) {
7   alert(response);
8 });

```

o, incluso, se podría hacer de la siguiente manera:

```

1 $http.post('/someUrl', data)
2 .then(function (response) {
3   alert(response);
4 });

```

La segunda opción proporciona una versión preconfigurada, la cual crea una petición HTTP a la URL dada. Incluso se ha creado una mayor abstracción utilizando el servicio `$resource`, el cual se construye sobre el servicio `$http`.

2.4.3. Patrón Proxy.

Un proxy, en su sentido más general, es una clase que funciona como una interfaz de otra cosa. El proxy podría hacer de *interface* a cualquier cosa: una conexión de red, un objeto grande en memoria, un archivo o algún otro recurso que sea costoso o imposible de duplicar.

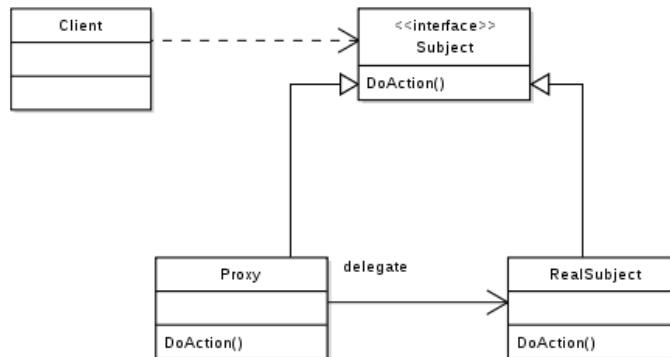


Figura 9: Patrón proxy.

Se pueden distinguir 3 tipos de proxies:

- Proxy virtual
- Proxy remoto
- Proxy protector

Se mencionará la implementación que AngularJS tiene del proxy virtual. A continuación se muestra una llamada al método `get` de la instancia `$resource`, llamada `User`:

```

1 var User = $resource('/users/:id'),
2     user = User.get({ id: 42 });
3 console.log(user); // {}

```

En el *snippet* anterior, `console.log` generaría un objeto vacío. Dado que la petición AJAX que ocurre por detrás, cuando se invoca `User.get`, es asíncrona, no se tiene el usuario real cuando se llama a `console.log`. Justo después de que `User.get` haga la petición GET, devuelve un objeto vacío y mantenga una referencia a él. Se puede pensar en este objeto como proxy virtual (un simple *placeholder*), que se llenaría con los datos reales una vez que el cliente reciba la respuesta del servidor. Esto en Angular funciona de la siguiente manera, consideremos el siguiente *snippet*:

```
1 function MainCtrl($scope, $resource) {  
2   var User = $resource('/users/:id'),  
3   $scope.user = User.get({ id: 42 });  
4 }  
  
1 <span ng-bind="user.name"></span>
```

Cuando se ejecuta el código anterior, la propiedad `user` del objeto `$scope` tendrá un valor de un objeto vacío (`{}`), lo que significa que `user.name` será indefinido y no se renderizará nada en la vista. Internamente AngularJS mantendrá la referencia a este objeto vacío. Una vez que el servidor devuelva la respuesta para la petición GET, AngularJS llenará el objeto con los datos recibidos del servidor. Durante el siguiente bucle de `$digest`, AngularJS detectará el cambio en `$scope.user`, lo que llevará a la actualización de la vista.

2.4.4. Inyección de dependencias

La Inyección de Dependencias (*DI* por sus siglas en inglés) es un patrón de diseño de software que se ocupa de cómo los componentes consiguen sus dependencias. AngularJS cuenta con un subsistema inyector que se encarga de crear componentes, resolver sus dependencias y proporcionarlos a otros componentes según lo solicitado.

La *DI* aparece continuamente en AngularJS. Puede utilizarse al definir componentes o al proporcionar bloques de ejecución y configuración para un módulo (ver los métodos `run` y `config` de un módulo).

- Los componentes como servicios, directivas, filtros y animaciones se definen mediante una *factory method* inyectable o mediante un constructor. A estos componentes se les puede inyectar un *service* o un *value* como dependencias.
- Los controladores se definen mediante una función constructor, a la que se le puede inyectar cualquiera de los componentes *service* y *value* como dependencias, pero también pueden proporcionarse con dependencias especiales (`$scope`).
- El método `run` acepta una función, a la que se le puede inyectar componentes *service*, *value* y *constant* como dependencias. No se pueden inyectar *providers* en bloques de ejecución.
- El método `config` acepta una función, a la que se le puede inyectar componentes *provider* y *constant* como dependencias. No se pueden inyectar componentes *service* o *value* en la configuración.

Para una discusión más en profundidad sobre *DI*, puede consultarse [Inyección de Dependencia](#) en Wikipedia, [Inversión de Control](#) de Martin Fowler.

3. Estado del arte

Estado de la cuestión relacionado con la interoperabilidad clínica. Normas UNE EN ISO-13606 y *openEHR*: Modelos de referencia y arquetipos, modelo de guías clínicas GDL. Arquetipos clínicos. Terminologías clínicas: SNOMED-CT, ICD10, LOINC, ICPC...

3.1. Estándares de interoperabilidad

Estándares de interoperabilidad

3.2. *openEHR*: Modelo de Referencia y Modelo de arquetipos

openEHR: Modelo de Referencia y Modelo de arquetipos

3.3. Terminologías clínicas

Terminologías clínicas

3.4. Especificación GDL

3.4.1. Introducción y requisitos

Propósito. Expresar y compartir contenido computarizado de apoyo a la toma de decisiones clínicas (CDS) a través de lenguajes y plataformas técnicas ha sido un objetivo evasivo durante mucho tiempo. La falta de modelos de información clínica compartidos compartidos y el apoyo flexible para los diferentes recursos de terminologías se han identificado como dos retos principales para compartir la lógica de decisión entre sistemas. GDL es un lenguaje formal para expresar lógica de soporte a las decisiones. Está diseñado para ser agnóstico a los lenguajes naturales y terminologías clínicas aprovechando los diseños del MR y MA de *openEHR*.

El alcance del GDL es expresar la lógica clínica como reglas de producción. Las reglas GDL discretas, que cada una de las cuales contienen declaraciones *if-then*, pueden combinarse como piezas para dar soporte a procesos de toma de decisiones sencillo y a procesos de toma de decisiones más complejos. Las reglas GDL pueden usarse para llevar a cabo la toma de decisiones en aplicaciones en consulta, así como en analíticas poblacionales retrospectivas.

DOCUMENTOS RELACIONADOS.

- Modelo de Referencia de *openEHR*
 - Modelo de Información de Tipos de Datos ([1.0.2](#))
 - Modelo de Información de Estructuras de Datos ([1.0.2](#))
 - Modelo de Información de HCE ([1.0.2](#))
 - Modelo de Información Común ([1.0.2](#))
- Modelo de Arquetipos de *openEHR*
 - Modelo de Objetos de Arquetipos (AOM) ([1.0.2](#))
 - Lenguaje de Definición de Arquetipos (ADL) ([1.0.2](#))

REQUISITOS

1. Debe ser posible expresar las reglas de CDS usando arquetipos como entrada y como salida de la ejecución de reglas.
2. Debe ser independiente del lenguaje natural y debe ser capaz de soportar traducciones a diferentes idiomas sin cambiar la definición de las reglas.
3. Debe ser independiente de las terminologías de referencia, por lo que pueden utilizarse terminologías diferentes para apoyar el razonamiento.
4. Debe ser sencillo convertir las reglas CDS en reglas expresadas en los principales lenguajes de uso común para su ejecución.
5. Debe haber suficiente meta-information acerca sobre las reglas CDS, *e.g.*, autoría, propósito, versiones y referencias relevantes.
6. Debe ser posible reutilizar las reglas CDS en diferentes contextos clínicos.
7. Debería ser posible agrupar un conjunto de normas CDS relacionadas con el fin de apoyar la toma de decisiones complejas.

3.4.2. Principios de diseño

Teniendo en cuenta los requisitos mencionados anteriormente, se han tomado las siguientes decisiones de diseño.

Arquetipos tanto como entradas como salidas de las reglas. Esto se consigue mediante la creación de enlaces entre los elementos de los datos definidos por los arquetipos y las variables utilizadas por las reglas del CDS. Cada variable de regla CDS se identifica de forma única en el contexto de una guía y se enlaza con un elemento específico definido por un arquetipo

utilizando su "Archetype ID" y su *path*⁷. Una vez definida, la variable se puede usar dentro de las sentencias *when* y *then* como entrada o como salida durante la ejecución de la regla.

Independencia del lenguaje natural. Se han utilizado varias ideas de diseño del formalismo de arquetipos de *openEHR* para lograr la neutralidad del lenguaje natural. En primer lugar, toda la meta-information dependiente del idioma sobre el propósito, el uso, el uso desaconsejado y las referencias de las reglas se agrupan bajo la sección *description* y se indexan por los códigos de idioma ISO dentro de la guía. En segundo lugar, todas las etiquetas y descripciones dependientes del lenguaje natural, *e.g.* el nombre de una variable de regla, se definen en la sección *term_definitions* de la guía y se indexan mediante códigos de idioma ISO. En tercer lugar, en las expresiones de la regla, se utilizan los identificadores únicos de variables y reglas, en lugar de sus nombres, ya que estos últimos dependen del idioma.

Independencia de terminologías de referencia. Cuando se utiliza el operador *IS_A* en las instrucciones de evaluación para la verificación de relaciones de pertenencia, se utiliza un término definido localmente en lugar de un código externo. Esta indirección hace posible modificar el código o añadir nuevos códigos de otras terminologías sin cambiar las definiciones de las reglas. Los enlaces entre los códigos definidos localmente y las terminologías de referencia externas se mantienen en la sección *term_bindings* del documento GDL.

Independencia del lenguaje de reglas. GDL sólo utiliza un conjunto de características comunes de lenguaje de reglas, como *when* y *then*. Las expresiones en las declaraciones *when* y *then* soportan cálculos aritméticos comunes, operadores lógicos y funciones.

Agrupación y reutilización de reglas. Un documento GDL (guía clínica) puede contener varias reglas que se relacionen entre sí. Cada guía es autocontenido y debe ser reutilizable en diferentes contextos clínicos. Se pueden encadenar diferentes guías para permitir el soporte de decisiones complejas. Esto se logra seleccionando la salida de una regla, como un elemento específico de un arquetipo, como entrada de otra regla.

Meta-information de las reglas CDS. La información de autoría, el estado del ciclo de vida y varias meta-informationes son compatibles con la reutilización de la clase *RESOURCE_DESCRIPTION* del diseño *openEHR*.

3.4.3. Modelo de Objetos de Guías Clínicas

Los fundamentos del diseño de GDL son los arquetipos *openEHR*, tanto como entrada como salida de las reglas CDS. Ésta es la clave para conseguir la independencia lingüística y de las terminologías de referencia. Debido a esta elección de diseño, la especificación *openEHR* juega un papel importante en el diseño GDL. En otras palabras, el diseño GDL tiene como objetivo hacer una reutilización sustancial de las especificaciones *openEHR* existentes. En áreas donde el diseño *openEHR* existente no es suficiente, se introducen diseños adicionales.

El modelo de objetos de guía (Guide Object Model, GOM), el modelo de objetos del GDL, consta de dos paquetes: el *paquete guía* y el *paquete de expresiones* descrito en detalle a través de las dos secciones siguientes.

3.4.4. Paquete de Guías Clínicas

La vista general del *paquete guía* se ilustra en la Figura 10. Las clases en color azul se basan, en líneas generales, en el diseño original de las especificaciones *openEHR*.

⁷ El *path* de un elemento de un arquetipo es la ruta única que identifica dicho elemento dentro del arquetipo.

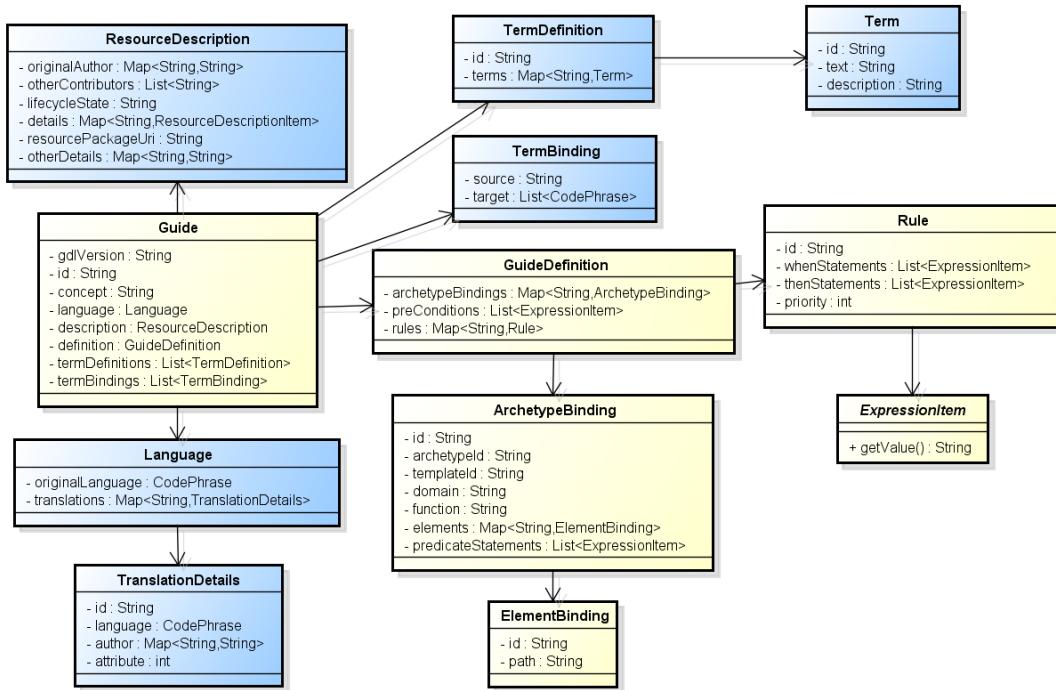


Figura 10: Paquete "guía clínica".

GUIDE. Clase principal de una guía discreta que define enlaces a arquetipos, reglas y meta-information.

Atributos	Firma	Significado
0..1	gdl_version: String	La versión en la que está escrita la guía.
1..1	id: String	Identificador de la guía clínica.
1..1	concept: String	El significado normativo de la guía en su conjunto. Expresado como un código de guía local.
1..1	language: Language	Recursos en lenguaje natural de esta guía. Incluye un idioma original y una lista opcional de traducciones.
1..1	description: RESOURCE_DESCRIPTION	Descripción de los recursos de esta guía incluyendo autoría, uso / mal uso, ciclo de vida y referencias.
1..1	definition: GUIDE_DEFINITION	La parte de definición principal de la guía. Consiste en enlaces de arquetipos y definiciones de reglas.
1..1	ontology: GUIDE_ONTOLOGY	La ontología de la guía.

Cuadro 1: Clase *Guide*

GUIDE_DEFINITION. La definición de la guía incluye una lista de enlaces de arquetipos y una lista de definiciones de reglas.

Atributos	Firma	Significado
1..1	archetype_bindings: List<ARCHETYPE_BINDING>	Lista de enlaces de arquetipos, que definen elementos específicos a ser utilizados por las reglas.
1..1	rules: Map<String, Rule>	Mapa de reglas, indexado por un código <i>gt</i> local.
0..1	pre_conditions: List<EXPRESSION_ITEM>	Lista de pre-condiciones que tienen que ser cumplidas para que una guía pueda ser ejecutada.

Cuadro 2: Clase *GuideDefinition*

ARCHETYPE_BINDING. El enlace de la lista de elementos de un arquetipo o plantilla seleccionado a los códigos *gt* locales

Atributos	Firma	Significado
1..1	archetype_id: String	El ID del arquetipo desde el que se selecciona la lista de elementos.
0..1	template_id: String	ID de una plantilla opcional que se utilizará para seleccionar elementos.
0..1	domain: String	El espacio en el que residen las variables de regla. El valor puede ser "EHR" (cuando el valor se recupera del EHR) o "CDS" (cuando el valor deriva del motor CDS). Cuando no existe. Valor por defecto: "EHR" o "CDS".
1..1	Elements: Map<String, ELEMENT_BINDING>	Mapa de <i>element bindings</i> indexados por código <i>gt</i> .
0..1	predicate_statements: List<EXPRESSION_ITEM>	Lista de predicados (restricciones) que deben cumplirse antes de que se puedan realizar las consultas de a la HCE

Cuadro 3: Clase ArchetypeBinding

ELEMENT_BINDING. La vinculación entre un elemento específico de un arquetipo y una variable local en la guía.

Atributos	Firma	Significado
1..1	id: String	El <i>código gt</i> local del elemento
1..1	path: String	El <i>path</i> para llegar a este elemento dentro del arquetipo.

Cuadro 4: Clase ElementBinding.

RULE. Una regla definida en una guía.

Atributos	Firma	Significado
1..1	id: String	El <i>código gt</i> local de la guía
1..1	when_statements: List<EXPRESSION_ITEM>	Lista de expresiones a evaluar antes de que se pueda ejecutar la regla.
1..1	then_statements: List<ASSIGNMENT_EXPRESSION>	Lista de expresiones para generar salida de la regla..

Cuadro 5: Clase Rule.

3.4.5. Paquete de Expresiones

El paquete de expresiones se muestra en la Figura 11.

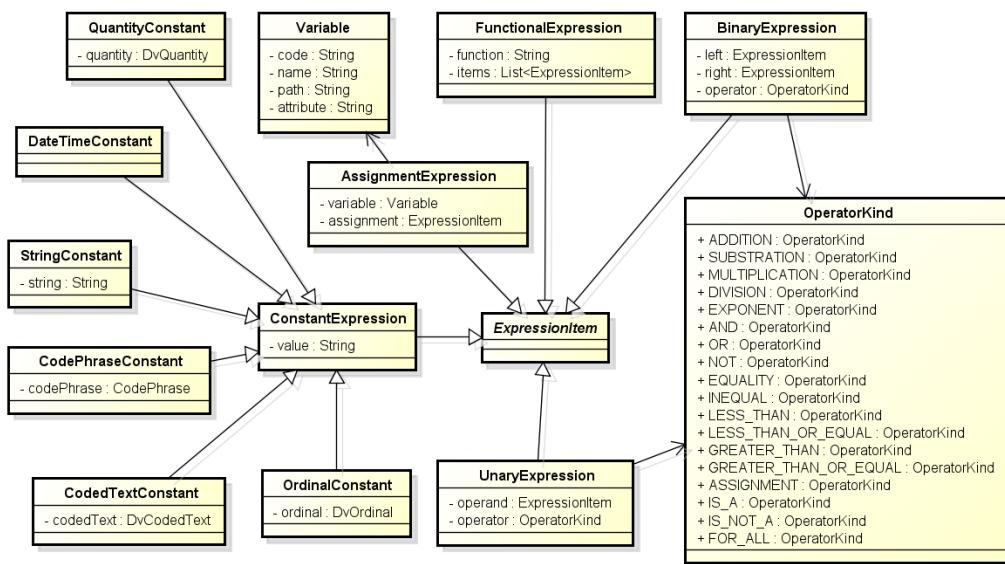


Figura 11: Paquete de Expresiones.

EXPRESSION_ITEM. Modelo abstracto de un ítem de expresión en una regla.

UNARY_EXPRESSION. Modelo abstracto de un ítem de expresión en una regla. Hereda de EXPRESSION_ITEM

Atributos	Firma	Significado
1..1	operand: EXPRESSION_ITEM	El operando de la expresión unaria.
1..1	operator: OPERATOR_KIND	El operador de la expresión unaria.

Cuadro 6: Clase *Unary Expression*.

BINARY_EXPRESSION. Modelo concreto de una expresión binaria. Hereda de EXPRESSION_ITEM.

Atributos	Firma	Significado
1..1	left: EXPRESSION_ITEM	El operando izquierdo de la expresión binaria.
1..1	right: EXPRESSION_ITEM	El operando derecho de la expresión binaria.
1..1	operator: OPERATOR_KIND	El operador de la expresión binaria.

Cuadro 7: Clase *BinaryExpression*.

ASSIGNMENT_EXPRESSION. Modelo concreto de una expresión de asignación. Hereda de EXPRESSION_ITEM.

Atributos	Firma	Significado
1..1	variable: String	El código <i>gt</i> de la variable a la cual asignar el valor.
1..1	assignment: EXPRESSION_ITEM	El <i>expression item</i> del que deriva el valor.

Cuadro 8: Clase *AssignmentExpression*.

FUNCTIONAL_EXPRESSION. Expresión concreta que modela una función. Hereda de EXPRESSION_ITEM.

OPERATOR_KIND. Enumerado que contiene todos los operadores utilizados.

FUNCTION_KIND. Tipos de funciones permitidas

Atributos	Firma	Significado
1..1	function: Kind	El tipo de función utilizada.
1..1	items: List<EXPRESSION_ITEM>	Lista de parámetros para la función.

Cuadro 9: Clase *FunctinalExpression*.

Tipo	Nombre	Símbolo
Aritmético	Suma	+
Aritmético	Resta	-
Aritmético	Multiplicación	*
Aritmético	División	/
Aritmético	Exponente	^
Lógico	And	&&
Lógico	Or	
Lógico	Not	!
Relacional	Igual	==
Relacional	Distinto	!=
Relacional	Menor que	<
Relacional	Menor o igual que	≤
Relacional	Mayor que	>
Relacional	Mayor o igual que	≥
De asignación	De asignación	=
De pertenencia terminológica	Es un	is_a
De pertenencia terminológica	No es una	!is_a

Cuadro 10: Clase *OperatorKind*.

Nombre	Función
abs	Devuelve el valor absoluto de un valor <i>double</i> .
ceil	Devuelve el valor <i>double</i> más pequeño que sea mayor o igual que el argumento e igual a un entero matemático.
exp	Devuelve el número de Euler <i>e</i> elevado a la potencia de un valor <i>double</i> .
floor	Devuelve el mayor valor <i>double</i> que es menor o igual que el argumento y es igual a un entero matemático.
log	Devuelve el logaritmo natural (en base <i>e</i>) de un valor <i>double</i> .
log10	Devuelve el logaritmo en base 10 de un valor <i>double</i> .
log1p	Devuelve el logaritmo natural de la suma del argumento más 1.
round	Devuelve el <i>long</i> más cercano al argumento.
sqrt	Devuelve la raíz cuadrada positiva correctamente redondeada de un valor <i>double</i> .
max	Se utiliza para obtener el valor máximo de un elemento.
min	Se utiliza para obtener el valor mínimo de un elemento.

Cuadro 11: Funciones permitidas.

4. Métodos

4.1. Introducción

Se ha optado por seguir una metodología ágil para la elaboración de este proyecto, en concreto se ha utilizado una versión simplificada de la metodología Scrum por ser una metodología compatible con los medios disponibles para el desarrollo de esta aplicación además de ser una metodología moderna que favorece el desarrollo rápido de aplicaciones.

Al ser una metodología ágil, se adapta perfectamente a nuestro proyecto permitiendo, mediante iteraciones, tener una aplicación funcional al final de cada una de las mismas, haciendo posible que los requisitos y las solucionen evolucionen con el paso del tiempo según las necesidades del proyecto

4.2. Metodología de desarrollo

4.2.1. Metodología Scrum

Scrum es un *framework* de desarrollo [scrum] en el que los equipos multifuncionales desarrollan productos o proyectos de forma iterativa e incremental. Scrum estructura el desarrollo en ciclos de trabajo llamados *Sprints*. Estas iteraciones no duran más de cuatro semanas cada una (lo más común son iteraciones de dos semanas), y tienen lugar una tras otra sin pausa. Los Sprints tienen una duración determinada, terminan en una fecha específica si el trabajo se ha completado o no, y nunca se extienden. Por lo general, los equipos Scrum eligen una longitud del Sprint y la utilizan para todos sus Sprints hasta que mejoren y puedan utilizar un ciclo más corto. Al comienzo de cada Sprint, un equipo multifuncional (de aproximadamente siete personas) selecciona los elementos (requisitos del cliente) de una lista de prioridades. El Equipo acuerda un objetivo colectivo de lo que ellos creen que pueden entregar al final del Sprint, algo que es tangible y que será verdaderamente "hecho". Durante el Sprint, no se pueden agregar nuevos elementos, éstos quedarían para el siguiente Sprint, ya que el Sprint corto actual está destinado a centrarse en un objetivo pequeño, claro y relativamente estable. Todos los días el equipo se reúne brevemente para inspeccionar su progreso, y ajustar los pasos siguientes necesarios para completar el trabajo restante. Al final del Sprint, el equipo revisa el Sprint con los *stakeholders*⁸, y demuestra lo que ha construido. La gente obtiene retroalimentación que se puede incorporar en el próximo Sprint. Scrum hace hincapié en que el producto tiene que ser funcional al final del Sprint. En el caso del software, esto significa un sistema integrado, completamente probado, documentado para el usuario final y potencialmente enviable. Las funciones clave, artefactos y eventos se resumen en la Figura 12.

⁸ *stakeholder*: cualquier parte interesada que es afectado o puede ser afectado por el desarrollo del producto

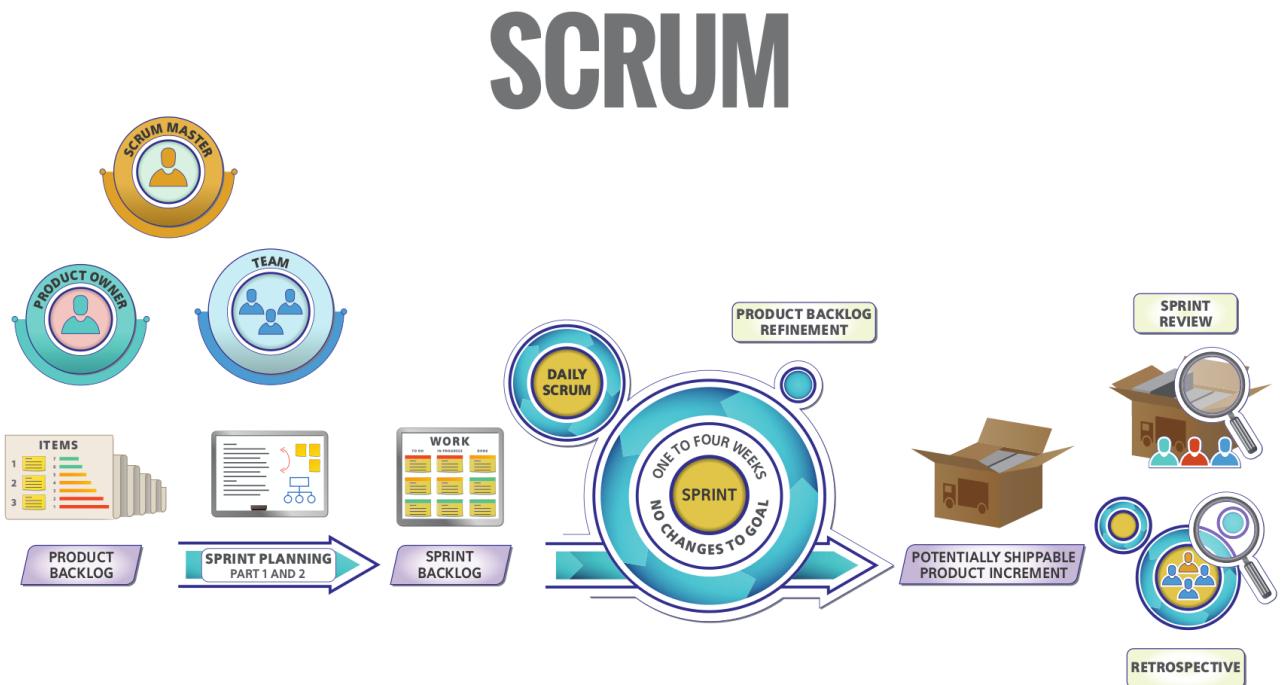


Figura 12: Scrum: visión general.

Un lema importante en Scrum es "inspeccionar y adaptarse". Dado que el desarrollo implica inevitablemente el aprendizaje, la innovación y las sorpresas, Scrum hace hincapié en dar pasos cortos de desarrollo, inspeccionando tanto el producto resultante como la eficacia de las prácticas llevadas a cabo y adaptando los objetivos del producto y las prácticas del proceso.

Roles

En Scrum existen 3 roles: el ScrumMaster, el propietario del Producto y el Equipo. Todos juntos se les conoce como en Equipo Scrum.

Propietario del Producto

Es el responsable de maximizar el retorno de la inversión (ROI) identificando las características del producto, traduciéndolas a una lista priorizada, decidiendo cuál debería estar en la parte superior de la lista para el próximo Sprint y continuamente redefiniendo y refinando la lista. El Propietario del Producto tiene la responsabilidad de las pérdidas y las ganancias del producto, asumiendo que es un producto comercial. En el caso de una aplicación interna, el propietario del Producto no es responsable del ROI en el sentido de un producto comercial (que generará ingresos), pero sigue siendo responsable de maximizar el ROI en el sentido de elegir, en cada Sprint, los ítems de más alto valor. En la práctica, "valor" es un término difuso y la priorización puede verse influida por el deseo de satisfacer a los clientes clave, la alineación con los objetivos estratégicos, la minimización de riesgos, la mejora y otros factores. En algunos casos, el propietario del Producto y el cliente son la misma persona, esto es bastante común en las aplicaciones internas. En otros casos, el cliente podría ser millones de personas con diferentes necesidades, en cuyo caso el rol de Propietario del Producto es similar al de *Product Manager* o de *Product Marketing Manager* en muchas organizaciones. Sin embargo, el Propietario del Producto es algo diferente de un Gerente de Producto tradicional porque interactúa activamente y regularmente con el Equipo, prioriza trabajando con todas las partes interesadas y revisando los resultados de cada Sprint, en lugar de delegar decisiones de desarrollo a un gerente de proyecto. Es importante observar que en Scrum hay una sola persona que sirve como Propietario del Producto, y él o ella es responsable del valor del trabajo; Aunque esa persona no tiene que trabajar sola.

El equipo

El Equipo, también llamado Equipo de Desarrollo, construye el producto que el Propietario del Producto indica: la aplicación o el sitio web, por ejemplo. El equipo en Scrum es "multifuncional", incluye toda la experiencia necesaria para

entregar el producto potencialmente en cada Sprint, y es auto-organizado, con un alto grado de autonomía y responsabilidad. El Equipo decide cuántos artículos—del conjunto ofrecido por el Propietario del Producto—debe construir en cada Sprint y la mejor manera de lograr ese objetivo.

En el Equipo no hay títulos especializados: no hay analista de negocios, ni DBA, ni arquitecto, ni jefe de equipo, ni diseñador de interfaz gráfica, ni programador. Los componentes trabajan juntos durante cada Sprint de la manera que sea más apropiada para alcanzar el objetivo que ellos mismos han fijado.

Cada persona tendrá habilidades primarias, secundarias e incluso terciarias. Los individuos también asumen tareas en áreas en las que están menos familiarizados para ayudar a completar dicha tarea. Por ejemplo, una persona cuya habilidad principal es el diseño de interfaces gráficas podría tener una habilidad secundaria en las pruebas automatizadas o alguien con habilidad primaria en escritura técnica también podría ayudar con el análisis y la programación.

El Equipo en Scrum consta de 5 a 9 personas. Para un producto de software el Equipo puede incluir personas con habilidades en análisis, desarrollo, pruebas, diseño de interfaces, diseño de bases de datos, arquitectura, documentación, etc. El equipo desarrolla el producto y proporciona ideas al Propietario del Producto sobre cómo hacer que el producto sea de la mayor calidad posible. En Scrum los Equipos son más eficientes si todos los miembros están dedicados a un único producto durante el Sprint. El equipo evita la multitarea a través de múltiples proyectos, para huir de las atenciones divididas y del cambio de contexto. Los equipos estables están asociados con una mayor productividad, por lo que es conveniente evitar cambiar a los miembros del equipo en la medida de lo posible.

ScrumMaster

El ScrumMaster ayuda a aplicar Scrum para producto comercial tenga éxito. Hace lo que esté en su mano para ayudar al Equipo, al Propietario del Producto y a la organización. El ScrumMaster no es el administrador de los miembros del equipo, ni es un jefe de proyecto, ni un líder del equipo. Ayuda a eliminar los impedimentos que puedan surgir, protege al equipo de interferencias externas y le ayuda a adoptar buenas prácticas de desarrollo. Educa, entrena y guía al Propietario del Producto, al Equipo y al resto de la organización en el uso adecuado de Scrum. El ScrumMaster es un entrenador y profesor, se asegura de que todo el mundo (incluido el Propietario del Producto y los administradores) entienda los principios y las prácticas de Scrum. Dado que Scrum hace visibles muchos impedimentos y amenazas a la eficiencia del Equipo y del Propietario de Producto, es importante tener un ScrumMaster comprometido trabajando enérgicamente para ayudar a resolver dichos problemas, de lo contrario el Equipo o el Propietario del Producto es probable que tengan dificultades para tener éxito. Un equipo pequeño podría contar con un miembro del equipo que desempeñe este papel (llevando una carga más ligera de trabajo regular cuando lo hace). Un ScrumMaster brillante puede provenir de cualquier disciplina: ingeniería, diseño, pruebas, gestión de productos, gestión de proyectos o gestión de la calidad.

El ScrumMaster y el Propietario del Producto no pueden ser el mismo individuo, ya que su enfoque es tan diferente que combinarlos a menudo lleva a la confusión y al conflicto. Un resultado común de combinar estos roles es un Propietario del Producto que se opone a la autogestión de los equipos que Scrum requiere. A diferencia de un gestor tradicional, el ScrumMaster no le dice a la gente qué hacer ni asigna tareas sino que básicamente facilita el proceso, apoya al equipo mientras se organiza y se gestiona. Si el ScrumMaster estaba previamente en una posición de gestión del equipo, tendrá que cambiar significativamente su mentalidad y estilo de interacción para que el equipo tenga éxito con Scrum.

No hay ningún rol de líder de proyecto en Scrum, esto se debe a que no es necesario. Las responsabilidades tradicionales de un director de proyecto se han dividido y reasignado entre los tres roles de Scrum, y en su mayoría al Equipo y Propietario del Producto, en vez de al ScrumMaster. Utilizar Scrum con un director de proyecto indica un mal uso fundamental de Scrum y típicamente resulta en responsabilidades conflictivas, autoridad poco clara y resultados mejorables. A veces un ex director de proyecto puede entrar en el papel de ScrumMaster, el éxito de este enfoque depende en gran medida de la persona, y lo bien que entienda la diferencia fundamental entre los dos roles, tanto en las responsabilidades del día a día como en la mentalidad necesaria para tener éxito. Una buena manera de entender a fondo el papel de ScrumMaster y comenzar a desarrollar las habilidades básicas necesarias para el éxito, es asistir a la formación de certificado ScrumMaster por Scrum Alliance.

Además de estos tres roles, existen otros *stakeholders* que contribuyen al éxito del producto como son los administradores, los cliente y los usuarios finales. Algunos *stakeholders* como los gerentes funcionales (*i.e.* un gerente de ingeniería) pueden encontrarse con que su role cambia al adoptar Scrum. Por ejemplo:

- Apoyan al equipo considerando las reglas y la esencia de Scrum.
- Ayudan a eliminar los impedimentos que identifican el Equipo y el Propietario del Producto.
- Ponen a disposición sus conocimientos y experiencia.

Documentos

La metodología Scrum establece la elaboración de una serie de documentos para apoyar, auditar y documentar el proceso.

Product backlog

Cuando un grupo planea migrar a Scrum, antes de empezar el primer Sprint, se necesita el *Product Backlog*, un documento de características centradas en el cliente priorizado y ordenado. Contiene descripciones genéricas de todos los requisitos, funcionalidades deseables, etc. Se trata de un documento de alto nivel para todo el proyecto, que existe y evoluciona durante la vida útil del producto. Es el plan de negocio del producto ([Figura 13](#) y [Figura 14](#)). En cualquier momento, el *Product backlog* ofrece una visión única y definitiva de "todo lo que podría ser hecho por el Equipo, en orden de prioridad", esta prioridad se organiza según su retorno sobre la inversión (ROI). Sólo existe un único *Product backlog* para un producto, lo que significa que el Propietario del Producto debe tomar decisiones de priorización de alto nivel, representando los intereses de todas las partes interesadas (incluido el Equipo).

Priority	Item	Details (wiki URL)	Initial Size Estimate	New Estimates at Sprint ...					
				1	2	3	4	5	6
1	As a buyer, I want to place a book in a shopping cart (see UI sketches on wiki page)	...	5						
2	As a buyer, I want to remove a book in a shopping cart	...	2						
3	Improve transaction processing performance (see target performance metrics on wiki)	...	13						
4	Investigate solutions for speeding up credit card validation (see target performance metrics on wiki)	...	20						
5	Upgrade all servers to Apache 2.2.3	...	13						
6	Diagnose and fix the order processing script errors (bugzilla ID 14823)	...	3						
7	As a shopper, I want to create and save a wish list	...	40						
8	As a shopper, I want to add or delete items on my wish list	...	20						

Figura 13: El *Product Backlog*.



Figura 14: Gestión visual: ítems del *Product Backlog* sobre la pared.

El *Product Backlog* incluye una variedad de elementos, principalmente nuevas características del cliente (e.g. "permitir a todos los usuarios colocar el libro en el carrito de compras"), pero también otros objetivos de mejora técnica (e.g. "reescribir

el sistema de C ++ a Java" o "Mejorar el rendimiento de los tests"), trabajos e investigación (e.g. "investigar soluciones para acelerar la validación de la tarjeta de crédito"), y, posiblemente, defectos conocidos (e.g. "diagnosticar y arreglar los errores del script de procesamiento de pedidos")

Los elementos *Product Backlog* se expresan de cualquier manera que sea clara y concisa. Contrariamente a los malentendidos populares, el *Product Backlog* no contiene "historias de usuarios", simplemente contiene elementos. Estos elementos pueden expresarse como historias de usuarios, casos de uso o cualquier otro enfoque de requisitos que el grupo considere útil. Pero cualquiera que sea el enfoque, la mayoría de los artículos deben centrarse en la entrega de valor a los clientes.

Sprint backlog

Muchos Equipos tienen un *Sprint Backlog* en forma de tablero de tareas de tamaño de pared (a menudo llamado también *Scrum Board*) donde las tareas, escritas en forma de notas Post-It, se van moviendo durante el Sprint a través de las columnas denominadas "To Do", "Work In Progress" y "Done". Ver [Figura 15](#)

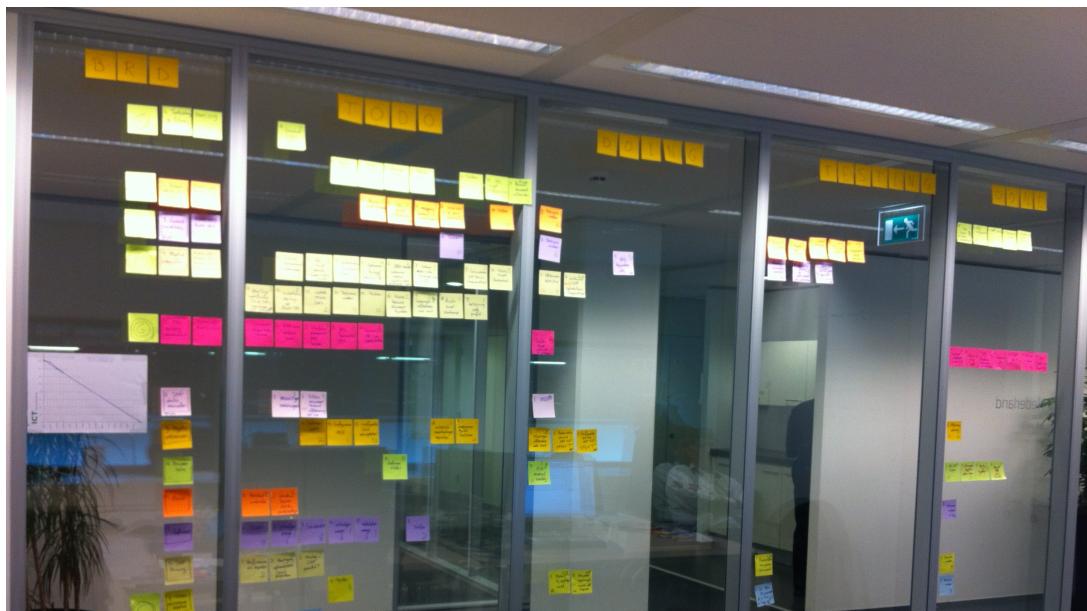


Figura 15: Gestión visual: tareas de un *Sprint Backlog* sobre la pared.

En definitiva, el *Sprint backlog* se trata de un documento detallado donde se describe cómo el equipo va a implementar los requisitos durante el siguiente Sprint. Las tareas se dividen en horas donde ninguna tarea tendrá una duración superior a 16 horas. Si una tarea es mayor de 16 horas, deberá ser dividida en subtareas de un nivel de detalle más fino. Las tareas en el *Sprint Backlog* no se asignan, sino que los miembros del equipo las van tomando de la manera que les parezca más oportuno.

Uno de los pilares de Scrum es que, una vez que el equipo establece su objetivo para el Sprint, cualquier añadido o cambio debe ser aplazado al siguiente Sprint. Esto significa que si el Propietario del Producto, en mitad de un Sprint, decide que hay un nuevo ítem en el que quisiera que el Equipo trabajara, no podría hacer el cambio hasta el comienzo del próximo Sprint. Si aparece una circunstancia externa que cambia significativamente las prioridades que significase que el Equipo estaría perdiendo el tiempo si continúa trabajando el Sprint actual, el Propietario del Producto o el Equipo puede abortar el Sprint. El equipo se detiene y se realiza una nueva reunión de planificación del Sprint, iniciándose de esta manera un nuevo Sprint. El perjuicio de hacer esto generalmente es grande, además de la posible desmotivación tanto para el Propietario del Producto como para el Equipo.

Que el Equipo esté protegido contra los cambios durante el Sprint tiene varias ventajas. En primer lugar, el Equipo se pone a trabajar sabiendo con absoluta certeza que su objetivo no va a cambiar, lo que refuerza el enfoque del Equipo en asegurar la finalización. En segundo lugar, obliga al *Propietario del Producto* a pensar realmente a través de los ítems que él prioriza en el *Product Backlog*.

Burn Down

La *Burn Down Chart* es una gráfica que se muestra públicamente y que mide la cantidad de requisitos en el *Backlog*

del proyecto pendientes al comienzo de cada Sprint. Dibujando una línea que conecte los puntos de todos los Sprints completados, se puede ver el progreso del proyecto. Lo normal es que esta línea sea descendiente (en casos en que todo va bien en el sentido de que los requisitos están bien definidos desde el principio y no varían nunca) hasta llegar al eje horizontal, momento en el cual el proyecto se ha terminado (no hay más requisitos pendientes de ser completados en el *Backlog*). Si durante el proceso se añaden nuevos requisitos la recta tendrá pendiente ascendente en determinados segmentos, y si se modifican algunos requisitos la pendiente variará o incluso valdrá cero en algunos tramos.

Reuniones

Uno de los pilares fundamentales de Scrum es la comunicación. A continuación se muestran los diferentes tipos de reuniones que tienen lugar en esta metodología:

Daily Scrum

El principal objetivo de las *Daily Scrum meetings* es llevar a cabo una actualización y una coordinación entre los miembros del Equipo. En estas reuniones son diarias durante el desarrollo de un Sprint, participan obligatoriamente todos los miembros de el Equipo, la presencia de el Propietario del Producto es opcional y el ScrumMaster suele estar presente. La duración máxima no debe exceder de los 15 minutos, para mantenerla breve se recomienda que todos estén permanezcan de pie, es la oportunidad del equipo para sincronizar su trabajo e informar de cualquier obstáculo que pudiese haber aparecido. Cada uno debe ir informando al resto del equipo de lo siguiente:

- ¿Qué se ha logrado desde la última reunión?
- ¿Qué se hará antes de la próxima reunión?
- ¿Qué obstáculos han aparecido en el camino?

Hay que destacar que el *Daily Scrum* no es una reunión de estado para informar a un gerente, se trata de un tiempo para que un equipo auto-organizado comparta entre sí lo que está pasando, se trata de una ayuda a la coordinación.

Scrum de Scrums

Suele realizarse cada día, normalmente después del *Daily Scrum*.

- Estas reuniones permiten a los grupos de equipos discutir su trabajo, enfocándose especialmente en áreas de solapamiento e integración.
- Asiste una persona asignada por cada equipo.

La agenda será la misma que la del *Daily Scrum*, además de las siguientes cuatro preguntas:

- ¿Qué ha hecho tu equipo desde nuestra última reunión?
- ¿Qué hará tu equipo antes que nos volvamos a reunir?
- ¿Hay algo que demore o estorbe a tu equipo?
- ¿Estás a punto de encargarle una tarea a otro equipo?

Sprint Planning Meeting

Se trata de la reunión para preparar cada Sprint, típicamente dividida en dos partes, la primera se centra en el *qué* y la segunda en el *cómo*. En la primera parte participa el Equipo, el Propietario del Producto y el ScrumMaster. En la segunda participa el Equipo y el ScrumMaster (el Propietario del Producto puede asistir o no, pero debe de estar disponible por si surgiesen dudas). La duración de cada una de las partes es de una hora por semana de Sprint. Esta reunión tiene lugar al principio de cada Sprint que, como se ha mencionado anteriormente, suelen durar entre 1 y 4 semanas y se tratan los siguientes puntos:

- Seleccionar qué trabajo se hará exactamente.
- Preparar, con el equipo completo, el *Sprint Backlog* que detalla el tiempo que llevará hacer el trabajo.
- Identificar y comunicar cuánto del trabajo es más probable que se pueda realizar durante el actual Sprint.
- Tiempo dedicado: ocho horas de límite.

Al final del ciclo Sprint, se llevarán a cabo dos reuniones: la *Sprint Review Meeting* y la *Sprint Retrospective Meeting*

Sprint Review Meeting

En esta reunión se trata la inspección y adaptación relacionada con el incremento de funcionalidades del producto a desarrollar. Participa el Equipo, el Propietario del Producto, el ScrumMaster y otros interesados (*stakeholders*), según corresponda, invitados por el Propietario del Producto. La duración es de una hora por semana de Sprint.

Tienen lugar cuando termina el Sprint y es el momento en el que la gente hace una revisión del Sprint. La idea principal es que esta reunión permita inspeccionar y adaptar el producto, se revisa qué trabajo fue completado y cuál no lo ha sido. Se hace una presentación del trabajo completado a todos los interesados, no permitiéndose mostrar el trabajo no completado. Un elemento crítico de esta revisión se centra en una *conversación* en profundidad entre el Equipo y el Propietario del Producto para que éste último conozca la situación, reciba consejos, etc.

Sprint Retrospective

Esta reunión es la que sigue a la *Sprint Review Meeting*, la cual se centraba en la inspección y adaptación del producto. La *Sprint Retrospective* se centra en la inspección y adaptación del proceso y del entorno. Participa el Equipo, el ScrumMaster y, opcionalmente, el Propietario del Producto. Otros *stakeholders* pueden ser invitados por el Equipo, pero no está permitida la asistencia de nadie más.

En esta reunión el Equipo discute qué está funcionando y qué no lo está y se acuerdan cambios para buscar soluciones a ciertas cosas que no funcionen. A veces, el ScrumMaster puede actuar como un coordinador eficaz para la retrospectiva, aunque en ocasiones puede ser una mejor opción invitar a una entidad externa que sea neutral para la coordinación de la reunión.

4.2.2. Fases de Scrum

Se diferencian cuatro fases en el proceso de desarrollo utilizando la metodología Scrum:

Revisión de los planes de lanzamiento y distribución, revisión y ajuste de los estándares del producto

Esta fase es llevada a cabo por los desarrolladores realizando una revisión de lo que hay que hacer y definiendo los detalles de la revisión actual (tecnologías, estándares, etc.).

Sprint

Es cada una de las iteraciones o cada uno de los ciclos repetitivos de trabajo similar que producen un incremento de las funcionalidades del producto o sistema. Se trata de la fase de desarrollo que incluye análisis, el diseño, la implementación, las pruebas, el *empaquetado*, generación de ejecutables, etc. La duración es de entre una y cuatro semanas, esta duración se fija a nivel global y todos los equipos que trabajan en el mismo sistema o producto utilizan la misma duración de ciclo. En la fase de revisión se resuelven problemas y se añaden nuevos elementos y en la fase de ajuste se utilizan las mejorías y ajustes encontrados para mejorar el producto ya se código, documentación, etc.

Revisión del Sprint

A veces se le denomina incorrectamente "la demo", concepto que no define la intención de esta revisión, cuyo objetivo es revisar el producto realizado durante el último Sprint y se añadir *Backlogs* nuevos en caso necesario. Esta fase puede admitir la participación de los clientes, los ejecutivos, etc.

Cierre

En esta fase se encuentran las típicas actividades de fin de proyecto dirigidas a obtener una versión distribuible, como el *testing*, el *debugging*, la promoción, el *marketing*, etc.

4.2.3. Ventajas de Scrum

Scrum proporciona una serie de ventajas al proyecto. Debido a la filosofía de las metodologías ágiles y a las características descritas anteriormente, dichas ventajas se pueden resumir en:

- Gestión regular de las expectativas del cliente: el cliente establece sus expectativas indicando el valor que le aporta cada requisito del proyecto y cuándo espera que esté completado. Éste puede comprobar de manera regular si se van cumpliendo sus expectativas, y puede proporcionar *feedback*. Y es desde el inicio del proyecto cuando puede tomar decisiones a partir de resultados objetivos y dirigirlos iteración a iteración hacia su meta. Se ahorran esfuerzos y tiempo al evitar las posibles hipótesis.

- Resultados anticipados (*time to market*): el cliente puede empezar a utilizar los resultados más importantes del proyecto antes de que éste haya finalizado por completo. Siguiendo la ley de Pareto (el 20% del esfuerzo proporciona el 80% del valor), el cliente puede empezar antes a recuperar su inversión (y/o autofinanciarse) comenzando a utilizar un producto al que sólo le faltan características poco relevantes, puede sacar al mercado un producto antes que su competidor, puede hacer frente a urgencias o nuevas peticiones de clientes, etc.
- Flexibilidad y adaptación: de manera regular el cliente redirige el proyecto en función de sus nuevas prioridades, de los cambios en el mercado, de los requisitos completados que le permiten entender mejor el producto, de la velocidad real de desarrollo, etc. Al final de cada iteración el cliente puede aprovechar la parte de producto completada hasta ese momento para hacer pruebas de concepto con usuarios o consumidores y tomar decisiones en función del resultado obtenido.
- Retorno de inversión (ROI): de manera regular, el cliente maximiza el ROI del proyecto. Cuando el beneficio pendiente de obtener es menor que el coste de desarrollo, el cliente puede finalizar el proyecto.
- Mitigación de riesgos: desde la primera iteración el equipo tiene que gestionar los problemas que pueden aparecer en una entrega del proyecto. Al hacer patentes estos riesgos, es posible iniciar su mitigación de manera anticipada. "Si hay que equivocarse o fallar, mejor hazlo lo antes posible". El *feedback* temprano permite ahorrar esfuerzo y tiempo en errores técnicos. La cantidad de riesgo a que se enfrenta el equipo está limitada a los requisitos que se pueden desarrollar en una iteración. La complejidad y riesgos del proyecto se dividen de manera natural en iteraciones.
- Productividad y calidad: de manera regular el equipo va mejorando y simplificando su forma de trabajar. Los miembros del equipo sincronizan su trabajo diariamente y se ayudan a resolver los problemas que pueden impedir conseguir el objetivo de la iteración. La comunicación y la adaptación a las diferentes necesidades entre los miembros del equipo son máximas (se van ajustando iteración a iteración), de manera que no se realizan tareas innecesarias y se evitan ineficiencias.

Las personas trabajan más enfocadas y de manera más eficiente cuando hay una fecha límite a corto plazo para entregar un resultado al que se han comprometido. La conciencia de esta limitación temporal favorece la priorización de las tareas y fuerza la toma de decisiones. Las iteraciones (Sprints) son regulares y de un mes para facilitar la sincronización sistemática con otros equipos, con el resto de la empresa y con el cliente. El Equipo minimiza su dependencia de personas externas para poder avanzar (depender de la disponibilidad de otros puede parar o retrasar tareas).

La estimación de esfuerzo y la optimización de tareas para completar un requisito es mejor si la realizan las personas que van a desarrollar el requisito, dadas sus diferentes especializaciones, experiencias y puntos de vista. Asimismo, con iteraciones cortas la precisión de las estimaciones aumenta. Las personas también trabajan de manera más eficiente y con más calidad cuando ellas mismas se han comprometido a entregar un resultado en un momento determinado y deciden cómo hacerlo, no cuando se les ha asignado una tarea e indicado el tiempo necesario para realizarla. El Equipo evita ocupar mucho tiempo en tareas que sigan un camino equivocado, que le obligue a realizar un gran esfuerzo para llegar al objetivo esperado. Se asegura así, la calidad del producto de manera sistemática y objetiva, a nivel de satisfacción del cliente, requisitos listos para ser utilizados y calidad interna del producto.

- Alineamiento entre cliente y Equipo: los resultados y esfuerzos del proyecto se miden en forma de objetivos y requisitos entregados al negocio. Todos los participantes en el proyecto conocen cuál es el objetivo a conseguir. El producto se enriquece con las aportaciones de todos.
- Equipo motivado: las personas están más motivadas cuando pueden usar su creatividad para resolver problemas y cuando pueden decidir organizar su trabajo. Las personas se sienten más satisfechas cuando pueden mostrar los logros que consiguen.

4.3. Aplicación de Scrum en el proyecto

Como ya se ha comentado al principio de esta sección, se ha utilizado una versión simplificada de la metodología Scrum.

Esta versión simplificada consiste en la realización de iteraciones bien diferenciadas y que permiten ir obteniendo un producto de desarrollo operativo al final de cada una. Se han realizado tres iteraciones, en las cuales, al final de cada una de ellas, se obtiene un subproducto funcional de la aplicación final. En la primera iteración se obtiene una aplicación con la funcionalidad básica, se crea la estructura del proyecto AngularJS y se desarrolla la funcionalidad de muestra de las guías clínicas que existen en el sistema. En la segunda iteración se lleva a cabo el desarrollo de la edición de las guías, permitiendo la gestión de la descripción de las mismas y de la definición de los elementos de las guías, que provienen de los arquetipos clínicos. La iteración 2 complementa a la 1 y permite tener un sistema con una funcionalidad mas avanzada. En la última iteración se finaliza la edición de las guías aumentando la funcionalidad, es decir, en esta iteración se desarrolla la funcionalidad que permite editar la lista de reglas (condiciones y acciones), las precondiciones que se tienen que cumplir para que la regla se ejecute, las terminologías clínicas, los enlaces a las diferentes terminologías y las vistas finales tanto del código en formato GDL como una vista informativa

en formato HTML. En esta última iteración se ha desarrollado también la funcionalidad de creación de una nueva guía clínica, obteniéndose de este modo la aplicación global completamente terminada y funcional.

Otra particularidad que caracteriza a esta versión simplificada de Scrum es la analogía en cuanto a las reuniones. Se han realizado reuniones periódicas entre el ScrumMaster, en este caso el director profesional del proyecto, y el Equipo, en este caso mi persona, como alumno que realiza el proyecto. No se han realizado *Daily Scrum meetings* por motivos laborales, pero sí se han realizado reuniones siempre de la misma duración y en la medida de lo posible en el mismo lugar (en su defecto se han llevado a cabo mediante sesiones remotas de Skype). En estas reuniones se trataron los asuntos referentes a qué se ha hecho (y cómo), los problemas que hubiesen podido surgir y qué se pretendía hacer hasta la siguiente reunión. En cada iteración se describieron las tareas a llevar a cabo dependiendo de la fase en la que se encontrase el proyecto de acuerdo a la iteración vigente. Al terminar cada iteración hemos procedido a la revisión de la misma, donde se ha analizado el grado de cumplimiento de los objetivos planteados para dicha iteración, realizando los cambios pertinentes en caso de que fueran necesarios. La misma persona que el ScrumMaster ha jugado el papel de cliente, ya que conoce a la perfección las necesidades que existen en la plataforma de *openEHR* donde se pretende implantar este proyecto.

Por lo tanto, se han aplicado las principales características de Scrum como la realización por iteraciones con un producto operativo al final de cada una de ellas, las revisiones periódicas y la toma de decisiones sobre el desarrollo guiado por las reuniones regulares por parte de las partes implicadas en el desarrollo de este proyecto.

5. Desarrollo

La fase de desarrollo de este editor de guías clínicas se ha llevado a cabo en X iteraciones o Sprints de Scrum, obteniéndose al final de cada una de ellas una aplicación funcional y operativa. Cada iteración se corresponde con cada una de las funcionalidades que representa cada caso de uso y consta de una fase de análisis, una de diseño, una de implementación y una de pruebas. Siguiendo la metodología Scrum, ha habido una reunión de planificación antes del primer Sprint para decidir el orden de las funcionalidades a desarrollar en cada Sprint. A mayores nos hemos reunido antes de empezar cada iteración para decidir los componentes a desarrollar durante la misma.

A continuación se muestra un diagrama de despliegue de alto nivel, que ayuda a situar cada componente de los que se mencionan en esta memoria:

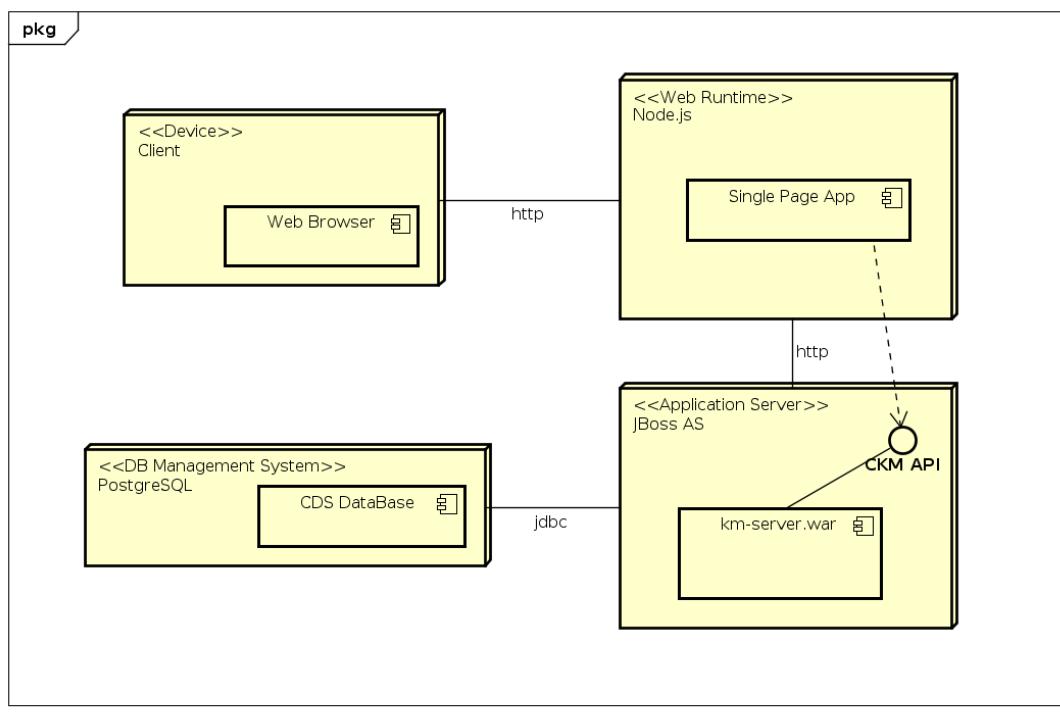


Figura 16: Diagrama de despliegue del sistema.

5.1. Iteración 1

En la primera iteración se ha establecido el objetivo básico de tener una primera versión funcional para posteriormente, ir añadiéndole el resto. De modo que lo que se pretendió llevar a cabo en este Sprint fue la decisión de la estructura de directorios de la aplicación y el desarrollo de permitir visualizar las guías clínicas existentes.

5.1.1. Análisis

En la fase de análisis del primer Sprint se han identificado dos tareas a ser tratadas. Por un lado se ha analizado la estructura organizativa de directorios de la aplicación y por otro el desarrollo de la funcionalidad básica consistente en la visualización de las guías clínicas existentes en el *backend*.

Para la primera tarea, se ha estudiado la mejor forma de estructurar la aplicación web. En numerosas ocasiones, en las primeras fases de un proyecto AngularJS, la estructura de directorios no importa demasiado y se tiende a ignorar las guías de buenas prácticas dedicadas a este cometido. Si se trata de una aplicación pequeña, esto no tendría importancia y permitiría al desarrollador tener algo funcional rápidamente, no afectando en ningún sentido. Pero si se trata de un proyecto mediano o grande con algo que complejidad, esto afectaría a la capacidad de mantenimiento de la aplicación. AngularJS es todavía relativamente nuevo y los desarrolladores todavía están averiguando cuál es la mejor forma de estructurar una SPA. Hay muchas maneras de estructurar una aplicación, se han tomado algunos principios de los *frameworks* MVC maduros existentes, pero también hemos tomado algunas cosas que son específicas de Angular.

Para construir aplicaciones que sean escalables y mantenibles en AngularJS se debe hacer uso de las guías de buenas prácticas existentes a tal efecto. La estructura de una aplicación Angular ideal debe ser modularizada en funcionalidades específicas. Se deben aprovechar las extraordinarias directivas de AngularJS para compartimentar aún más nuestras aplicaciones. las estructuras donde se usa un directorio por funcionalidad se conoce como aquellas que siguen el principio LIFT⁹. Esta aproximación es la adecuada para proyectos que no sean pequeños, aportando una serie de beneficios como los siguientes:

- Mantenibilidad del código: seguir este enfoque estructurando lógicamente las aplicaciones permite localizar u editar código fácilmente.
- Escalabilidad: la aplicación será mucho más escalable. Agregar nuevas directivas y plantillas no incrementará las carpetas existentes. La incorporación de nuevos desarrolladores también debería ser mucho más fácil una vez que se explica la estructura. Además, con este enfoque, se podrán añadir y eliminar funcionalidades de la aplicación con relativa facilidad de manera que testear nuevas funcionalidades debería ser bastante sencillo.
- Depuración: con esta aproximación modularizada es mucho más fácil depurar el código de la aplicación. Facilita encontrar las piezas de código erróneo y solucionar dichos errores.
- Pruebas: escribir tests unitarios y probar aplicaciones modularizadas es mucho más sencillo que las no modularizadas.

La segunda tarea de esta iteración se ha basado en la identificación de la funcionalidad básica consistente en la visualización de todas las guías clínicas existentes para que puedan ser seleccionadas y, a partir de ahí, se puedan modificar cada una de las secciones. Esta tarea se ha limitado a la visualización del listado de guías clínicas almacenadas en el *backend*. Para que esto sea posible, el caso de uso (ver Figura 17) que se ha de implementar es el que permite navegar entre las guías clínicas (*Browse clinical guidelines*), el cual se describe a continuación.

Descripción de los casos de uso

A continuación se describe el caso de uso comentado anteriormente.

Navegar guías clínicas

Para que el profesional sanitario pueda editar un guía clínica existente, primero debe poder visualizar las guías que ya están creadas. Cuando se le muestre el listado con las guías, el usuario simplemente seleccionará la que quiera editar.

⁹ **LIFT:** L-Localización del código sencilla, I-Identificación del código de un vistazo, F-Aplana la estructura lo máximo posible (F de *flatten*) y T-Tratar de no repetir código.

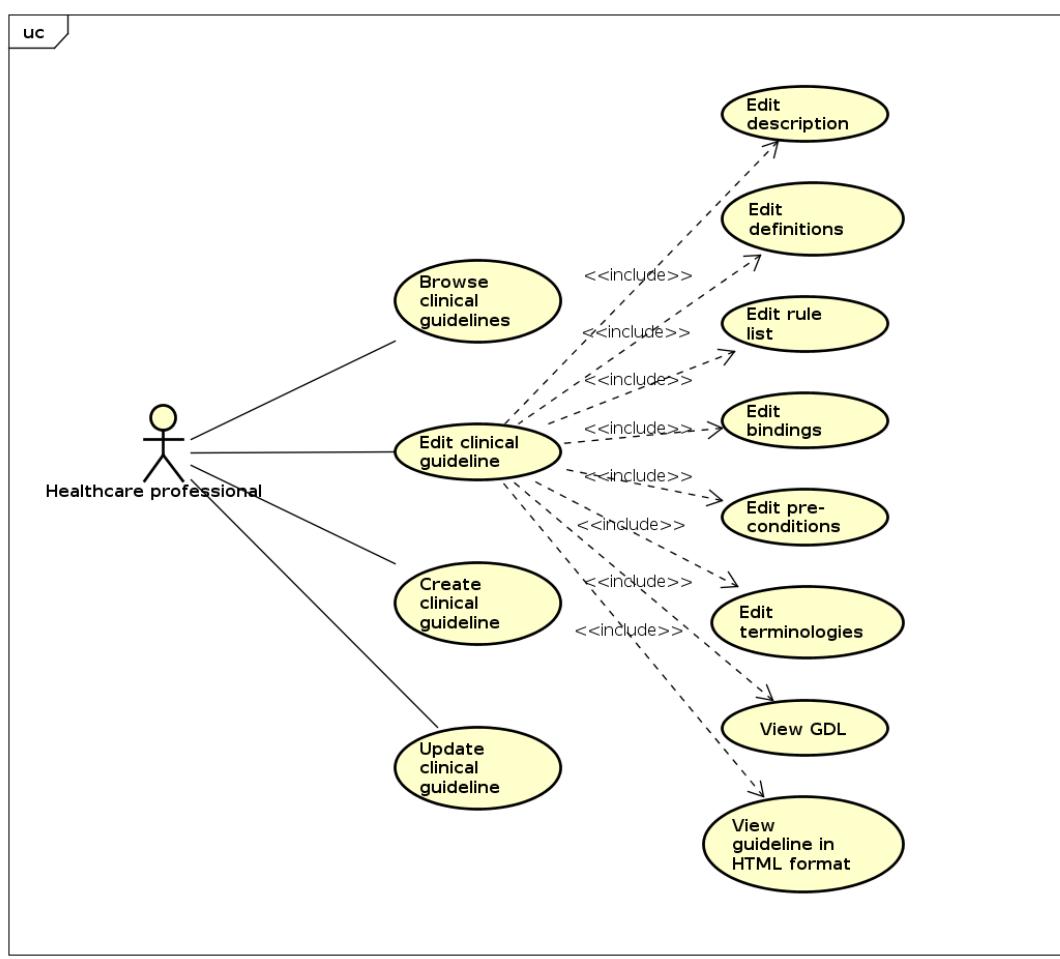
5.1.2. Diseño

Una aplicación profesional debe ir acompañada siempre de una documentación de modelado técnico y funcional hasta un cierto nivel de detalle, *i.e.* no demasiado detallado. El diseño debe ser suficiente para posteriormente encontrar más detalles en el código fuente. Si el código fuente aún no ha sido escrito, el diseño debe ser suficiente para que el desarrollador rellene los detalles a su propia discreción. Esta documentación permite ahorrar tiempo (y por lo tanto, dinero) y permite que los usuarios obtengan una aplicación de mayor calidad. El ahorro surge del hecho de que el diseño proporciona una visión de la estructura del software y por lo tanto reduce la posibilidad de una mala organización (*spaghetti code*, inconsistencias, código duplicado). Esta visión general de la estructura del software permite disminuir:

- La posibilidad de errores y, en consecuencia, el tiempo que se tarda en resolver dichos errores.
- El tiempo que se tarda en determinar la mejor manera de integrar un cambio o extensión en particular.
- El tiempo que se tarda en corregir defectos en la estructura más adelante.

En la fase de diseño de la primera iteración hemos realizado el diseño básico de toda la aplicación, esto conlleva los principales diagramas orientados a las funcionalidades generales.

En un editor de guías clínicas las funciones básicas que se deben poder realizar son las de crear una guía clínica desde el principio y el de poder navegar entre las guías existentes para poder seleccionar una y modificarla. Los casos de uso que se han de implementar son los siguientes:



powered by Astah

Figura 17: Casos de uso de la aplicación.

El único actor que tenemos en este desarrollo es el de *Profesional Sanitario* que, como conocedor del dominio médico, será el que elabore las guías clínicas.

Se ha dedicado una iteración para cada uno de los casos de uso que extienden del caso de uso base *Editar guía clínica*, por lo que la explicación de cada uno de ellos se indicara en la iteración correspondiente. La creación de una guía nueva se entiende como la edición de una guía con sus secciones vacías, excepto las obligatorias (ver Sección 3.4.4), que se rellenarán en el proceso que implementa el caso de uso *Crear nueva guía clínica*.

Como se ha comentado en la Sección 2.3.1, la arquitectura de alto nivel de una aplicación AngularJS se estructura en módulos, de tal manera que lo natural es comenzar el diseño creando un diagrama que represente los módulos de la aplicación. El símbolo pertinente de UML para representar un módulo es el paquete (*package*). Para dejar su significado absolutamente claro, se ha estereotipado el paquete como <<module>>.

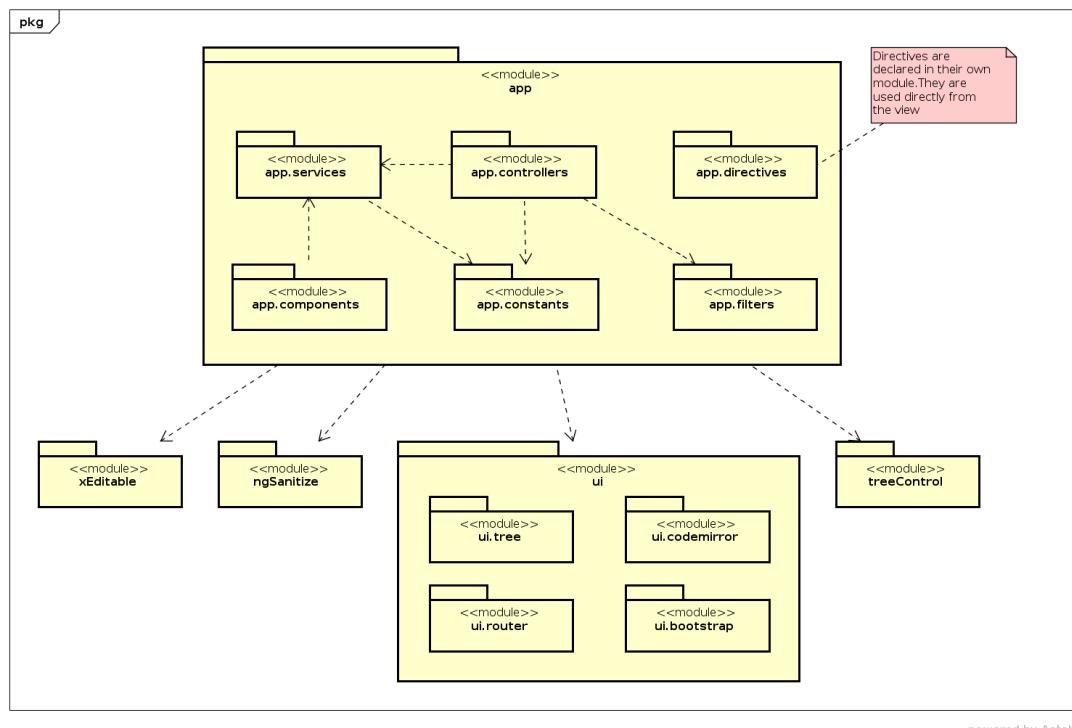


Figura 18: Arquitectura de alto nivel.

Las flechas son asociaciones de dependencia y se corresponden con las dependencias definidas en el código bien en el método `module` del objeto `angular` o bien en los propios objetos definidos en cada uno de los módulos. A continuación se describen los módulos definidos entrando un poco más en detalle e indicando el propósito y las responsabilidades de cada uno:

- **app:** es el módulo raíz de la aplicación. Se le pasa a la directiva `ngApp` para lo cargue en el `$injector` cuando se inicia la aplicación. Debe contener todo el código necesario para que la aplicación pueda funcionar o tener las dependencias sobre otros módulos que contengan dicho código.
- **app.services:** módulo que contiene los servicios personalizados de la aplicación. La idea básica de los servicios es agrupar funcionalidad que permita posteriormente ser utilizada en los controladores, mejorando la claridad y favoreciendo la reutilización del código.
- **app.controllers:** módulo que contiene los controladores personalizados de la aplicación. Se responsabilizan de proporcionar a la aplicación el comportamiento que soporte un marcado declarativo en la vista (plantilla HTML).
- **app.directives:** módulo que contiene las directivas personalizadas de la aplicación. En AngularJS cualquier acción que requiera modificar los elementos del DOM deben hacerse a través de una directiva.
- **app.components:** módulo que contiene los componentes personalizados de la aplicación, son un tipo especial de directiva que utilizan una configuración más simple que lo hace más adecuado para una aplicación con una estructura basada en compo-

nentes. Estos componentes son auto-contenidos, por lo que son reutilizables en diferentes puntos de la aplicación, fomentando las buenas prácticas del desarrollo de aplicaciones web.

- `app.constants`: módulo que contiene las constantes globales personalizados de la aplicación. Proporciona una manera de injectar valores constantes en cualquier controlador o servicio que lo necesite.
- `app.filters`: módulo que contiene filtros personalizados de la aplicación. Se usan en la vista, controladores o servicios para formatear los datos a mostrar en la vista. En esta aplicación web no se han utilizado en los servicios, motivo por el cual no existe una dependencia en el diagrama de la Figura 18.
- `ui`: módulo que actúa como contenedor de los 4 siguientes. Se han agrupado en éste debido a la relación de sus funcionalidades con la interfaz de usuario.
- `ui.tree`: componente de *drag and drop* que permite anidación en forma de árbol, utilizado en la definición de los elementos de los arquetipos, en la definición de los predicados, en las reglas y en las pre-condiciones de las guías clínicas.
- `ui.codemirror`: componente que facilita un área de texto, numerada, donde se permite editar la guía en formato GDL.
- `ui.router`: solución *de facto* para el enrutado de vistas en *Single Page Applications*. Este tipo de enrutado modela las aplicaciones como un árbol de estados jerárquico, cada estado se corresponde con una vista con su correspondiente controlador. Proporciona una **máquina de estados** que permite gestionar las transiciones entre dichos estados.
- `ui.bootstrap`: componente con un conjunto de directivas AngularJS nativas basadas en el marcado de Bootstrap y en CSS.
- `xEditable`: componente utilizado para la edición de tablas requerido en la sección de terminologías.
- `ngSanitize`: componente utilizado para evaluar como una expresión un HTML que proviene del *backend* e insertarlo de forma segura en el elemento correspondiente de la vista. Se utiliza en la vista HTML de la guía clínica.
- `treeControl`: módulo utilizado para la gestión de los árboles de la aplicación, utilizado en diferentes situaciones.

Para inicializar la aplicación AngularJS y declarar las dependencias mencionadas anteriormente lo hacemos a través un fichero en el que definimos el módulo principal (ver Sección 2.3.1), a este fichero le hemos llamado `app.js` y su contenido es el siguiente:

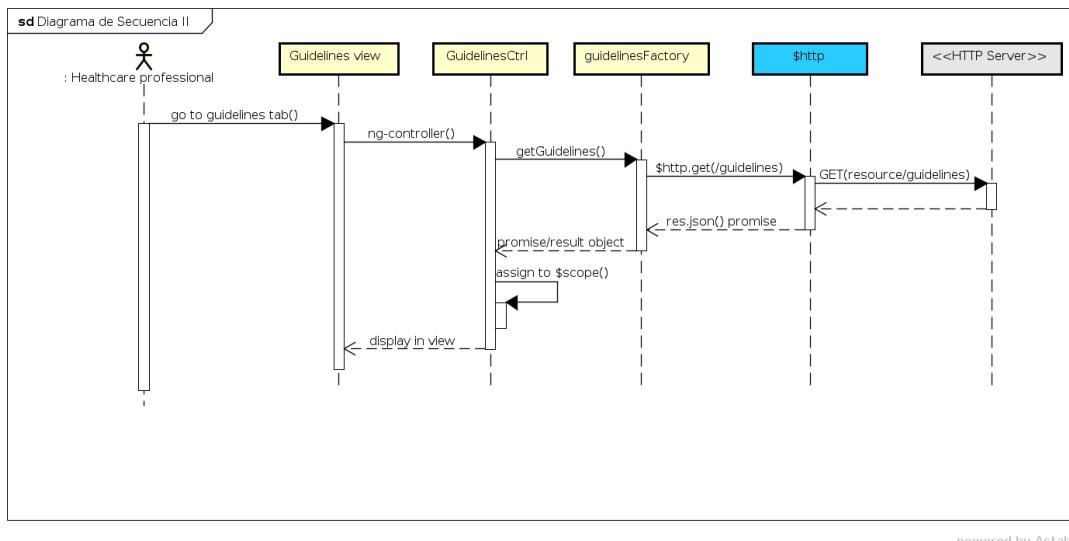
```
'use strict';

// Declare app level module which depends on views, and components
/**
 * Created by jbarros on 22/02/2015.
 */

angular.module('app', [
  'app.components',
  'app.constants',
  'app.services',
  'app.controllers',
  'app.directives',
  'app.filters',
  'app.version',
  'ui.tree',
  'xeditable',
  'ui.bootstrap',
  'ui.router',
  'ui.codemirror',
  'treeControl',
  'ngSanitize'
]);

```

La funcionalidad básica que se trató de llevar a cabo en esta primera iteración fue la de permitir mostrar las guías clínicas de manera que, en siguientes iteraciones, esta lista permitiese la navegación entre las mismas. El siguiente diagrama de secuencias muestra la forma en la que los objetos se comunican entre sí para cumplir esta tarea:



powered by Astah

Figura 19: Diagrama de actividades: listado guías clínicas.

Se diferencian 3 colores en los componentes implicados en las diferentes líneas de vida del diagrama de la Figura 19: amarillo para los desarrollados para la aplicación, azul para los que son nativos de AngularJS y en gris está representado el *backend*. Cuando se selecciona la pestaña *Guidelines* (ver manual de usuario), el enruteador de Angular automáticamente carga el controlador correspondiente (*GuidelinesCtrl*), al cual se le inyecta el servicio *guidelinesFactory*. Este controlador cuenta con un método *activate()* que es invocado al iniciarse y que llama al método *getGuidelines()* del servicio inyectado, el cual hace uso del servicio nativo de AngularJS *\$http*, que hace una petición GET al [Clinical Knowledge Manager](#) (CKM) API, el cual gestiona dicha petición y devuelve una promesa que se renderizará cuando dicha petición se haya completado.

5.1.3. Implementación

Como se ha comentado en Sección 5.1.1 se ha optado por seguir una aproximación LIFT para la estructura de directorios del proyecto.

```

app/
--- assets/                                // Recursos para la aplicación
----- css/                                 // Ficheros relacionados con los estilos
----- img/                                 // Imágenes e iconos de la aplicación
----- mocks/                               // Mocks utilizados para los tests unitarios, como ←
      respuestas de servicios web
--- components/                            // Cada componente se trata como una mini-aplicación ←
    AngularJS
----- common/
----- expression-editor/
----- layout/
----- modals/                             // Los modales de la aplicación, como componentes ←
      reutilizables
----- tabs/                               // Cada una de las principales funcionalidades del editor
-----   tab-binding/
-----   tab-definitions/
-----   tab-description/
-----   tab-gdl/
-----   tab-guidelines/
-----   tab-html/
-----   tab-preconditions/
-----   tab-rulelist/
-----   tab-terminology/
----- tabs.component.html

```

```
----- tabs.component.js
----- tabs.component.spec.js
----- version/
--- config/
----- constants.js
----- http.js
----- routes.js          // Enrutado, para que la SPA no tenga que recargar la ↵
    página entera durante la navegación
--- lib/
--- app.js
--- index.html
docs/                      // Documentación de la aplicación
--- output/
--- source/
--- diagramas.astah
e2e-tests/                 // Tests end-to-end
--- protractor.conf.js
--- scenarios.js
bower.json
package.json
```

El esquema anterior muestra una aproximación modularizada para construir aplicaciones AngularJS. Para la implementación de la primera iteración vamos a detallar los componentes que se han desarrollado para implementar la funcionalidad del listado de guías clínicas, así como los principales ficheros tanto de configuración como los necesarios para arrancar un proyecto Node.js y una aplicación AngularJS, excluyendo todo lo relacionado con las pruebas, que se verán en la siguiente sección.

En primer lugar, cabe mencionar el fichero de configuración de Node.js, denominado *package.json*. Permite la mejor manera de administrar los paquetes *npm*¹⁰ instalados localmente, entre otras funciones se destacan las siguientes:

- Sirve de documentación para los paquetes de los que depende un proyecto.
- Permite especificar las versiones de un paquete que su proyecto puede utilizar usando reglas de versionado semántico¹¹.
- Hace su compilación reproducible, lo que se traduce en una manera más sencilla de compartir con otros desarrolladores.

A continuación se muestra la estructura del *package.json* del proyecto.

```
{
  "name": "gdl-editor",
  "private": true,
  "version": "0.0.0",
  "description": "Angular GDL Editor application",
  "author": "Jesús Barros <j.barros@udc.es>",
  "repository": {
    "type": "git",
    "url": "https://github.com/jbarcas/angular-gdl-editor.git"
  },
  "license": "MIT",
  "devDependencies": {
    "bower": "^1.7.7",
    "http-server": "^0.9.0",
    "jasmine-core": "^2.4.1",
    "karma": "^0.13.22",
    "karma-chrome-launcher": "^0.2.3",
    "karma-firefox-launcher": "^0.1.7",
    "karma-jasmine": "^0.3.8",
    "karma-junit-reporter": "^0.4.1",
    "protractor": "^4.0.9"
  },
  "scripts": {
```

¹⁰ Node Package Management: <https://www.npmjs.com/>

¹¹ Versionado semántico: <http://semver.org/>

```

    "postinstall": "bower install",
    "prestart": "npm install",
    "start": "http-server -a localhost -p 8000 -c-1 ./app",
    "pretest": "npm install",
    "test": "karma start karma.conf.js",
    "test-single-run": "karma start karma.conf.js --single-run",
    "preupdate-webdriver": "npm install",
    "update-webdriver": "webdriver-manager update",
    "preprotractor": "npm run update-webdriver",
    "protractor": "protractor e2e-tests/protractor.conf.js"
  }
}

```

Este fichero, como mínimo tiene que tener los campos *name* y *version*, con el nombre y la versión del proyecto respectivamente. El campo *private*. Al establecer *private* como *true*, *npm* se negará a publicarlo. Esto impide que se publique accidentalmente en el registro público de *npm*, en *description* se indica una descripción clara y concisa del proyecto, *author* para el autor principal del proyecto. *repository* se tiliza para especificar la URL y el tipo de repositorio de código fuente, esto es útil para las personas que quieren contribuir a su módulo. Se usa el campo *license* para indicar bajo qué licencia se ha lanzado el código. Bajo *devDependencies* se idican las dependencias que sólo se destinan al desarrollo y pruebas del módulo. Por último, se tiene un campo *scripts* que consta de un objeto que expone comandos adicionales de *npm*, se asume que la clave es el comando *npm* y el valor es el *path* del script, por ejemplo, para iniciar el servidor web de desarrollo ejecutamos *npm start*.

Como gestor de dependencias para el desarrollo web *frontend* se ha utilizado Bower¹². Se cuenta con un fichero *bower.json* donde se especifican las dependencias del proyecto, de modo que se le pueda pedir a Bower que las instale todas de una vez que las actualice todas de una vez, si es que se encuentran versiones nuevas que te interese instalar y también, cuando subas a producción un proyecto, que te permita aprovisionarlo con todas las librerías externas necesarias. Este archivo es muy fácil de construir, con sintaxis JSON, indicando una serie de campos que necesita para definir tu proyecto y sus dependencias. A continuación se muestra el fichero *bower.json*.

```

{
  "name": "gdl-editor",
  "description": "Angular GDL Editor application",
  "version": "0.0.0",
  "homepage": "https://github.com/jbarcas/angular-gdl-editor.git",
  "license": "MIT",
  "private": true,
  "dependencies": {
    "angular": "~1.5.0",
    "angular-route": "~1.5.0",
    "angular-mocks": "~1.5.0",
    "html5-boilerplate": "^5.3.0",
    "angular-xeditable": "~0.1.9",
    "angular-ui-router": "~0.2.18",
    "bootstrap": "~3.3.7",
    "angular-bootstrap": "~2.5.0",
    "angular-ui-tree": "^2.22.5",
    "karma-read-json": "^1.1.0",
    "angular-ui-codemirror": "^0.3.0",
    "angular-sanitize": "1.5.0"
  },
  "resolutions": {
    "angular": "1.5.0"
  }
}

```

El formato es similar al *package.json*, sin embargo en el *bower.json* indicamos las dependencias de terceros que se han utilizado para el *frontend*, mientras que en el *package.json* se gestionan los módulos de Node.js. Para instalar las dependencias del *frontend* no es necesario ejecutar *bower install* a mano ya que se ha creado un script en el *package.json* (denominado *postinstall*) que lo hará automáticamente una vez iniciado el servidor web Node.js.

¹² Sistema de gestión de paquetes: <https://bower.io/>

Para gestionar la navegación de la aplicación entre las diferentes vistas se ha utilizado el módulo `ui-router`, el cual permite cargar varias vistas simultáneamente en una misma página. El patrón **Composite View** es uno de los clásicos a la hora de desarrollar la capa de presentación y define que la vista puede estar compuesta por varias subvistas que se actualizan de forma independiente.

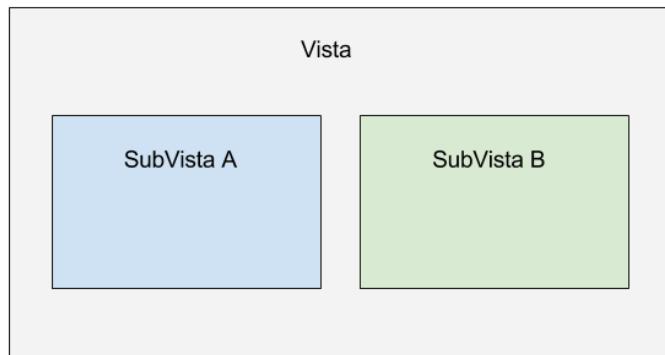


Figura 20: Patrón *Composite View*.

Para configurar AngularJS para que utilice el módulo `ui-router` lo registramos como dependencia del proyecto, tal como se ha mostrado en la sección anterior. El módulo en cuestión es el `ui.router` y se apoya en el concepto de *estado* para definir la navegación entre vistas:

- Un *estado* se corresponde a un "lugar" en la aplicación en términos de la interfaz de usuario general y de la navegación.
- Un *estado* (a través de las propiedades controlador / plantilla / vista) describe lo que muestra la interfaz de usuario y lo que se hace en ese lugar.
- Los *estados* tienen a menudo cosas en común, y la manera de refactorizar estos puntos en común en este modelo se hace a través de una jerarquía de estados.

El enrutado de la aplicación se ha definido en el fichero `routes.js`, cuyo contenido se muestra a continuación:

```

angular.module('app')
.config(function ($stateProvider, $urlRouterProvider) {
  $urlRouterProvider.otherwise("/tabs/tab-guidelines");
  $stateProvider
    .state("tab-guidelines", {
      url: "/tab-guidelines",
      templateUrl: "components/tabs/tab-guidelines/tab-guidelines.html",
      controller: "GuidelineCtrl",
      controllerAs: "vm"
    })
});
```

①
②
③
④

- ① *path* asociado al estado `tab-guidelines`.
- ② Plantilla (vista HTML) asociada a dicho estado.
- ③ Controlador vinculado al estado en cuestión,
- ④ Sintaxis que proporciona un código más claro en los controladores.

El módulo `ui-route` proporciona, entre otros, los servicios usados en el listado anterior, *i.e.* `$stateProvider` y `$urlRouterProvider`. El primero nos permite dar nombres a las rutas. Entre otras cosas, permite a cada estado asignarle un *path*, una vista y un controlador, de tal manera que, cuando se active dicho estado, rendereizará su vista correspondiente, con su controlador y su `$scope` de manera aislada garantizando de este modo un bajo acoplamiento entre las vistas. Para esta primera iteración,

solamente tendremos un estado que define la primera de las pestañas donde se muestran todas las guías clínicas y se le permite al usuario seleccionar una de ellas.

Una vez mencionados los principales ficheros a tener en cuenta a nivel de proyecto, ahora nos centramos en los relacionados en la tarea de *Navegar guías clínicas* dedicada a esta iteración. Cuando se activa un *estado* su vista asociada se inserta automáticamente en la directiva ui-view de la plantilla asociada a su estado padre, con la excepción de que si es un estado de primer nivel, la plantilla (o vista) de su estado padre es index.html.

En index.html se cuenta con dos **componentes**, uno que define un panel que indica la guía seleccionada y otro que define un conjunto de pestañas donde, cada una de ellas, va a permitir crear, editar o eliminar ciertas características de las guías clínicas.

```
1 <div class="container-fluid">
2   <gdl-panel></gdl-panel>
3   <gdl-tabs></gdl-tabs>
4 </div>
```

El componente tabs.component.js contiene la directiva ui-view donde se insertarán cada una de las pestañas que permiten editar las guías. Las guías tienen que mostrarse en la vista que se cargue al iniciar la aplicación, que es la que está asociada con el *estado* anteriormente definido (*tab-guidelines*). Este estado hace que se cargue la vista correspondiente en dicha pestaña, a continuación se muestra la vista y el controlador asociados a dicho estado.

```
1 <div class="panel panel-default">
2   <div class="panel-heading">
3     <h3 class="panel-title">Guidelines ({{vm.guidelines.length}})</h3>
4   </div>
5   <div class="panel-body gdl-columns gdl-overflow-x">
6     <div class="radio" ng-repeat="guideline in vm.guidelines | orderBy">
7       <label>
8         <input type="radio" ng-model="vm.checked" ng-click="vm.getGuideline(←
9           guideline)" ng-value="guideline">{{ guideline }}
10      </label>
11    </div>
12  </div>
13</div>
```

Tal como indica el routes.js, esta vista tiene asociado un controlador con un *scope* delimitado a la propia vista. Este controlador se llama GuidelineCtrl y es definido en tab.guidelines.controller.js, a continuación se muestra dicho controlador.

```
1 /**
2  * Created by jbarros on 16/08/16.
3  */
4
5 angular.module('app.controllers')
6   .controller('GuidelineCtrl', GuidelineCtrl);
7
8 function GuidelineCtrl(guidelineFactory, guidelinesFactory, SharedProperties) {
9
10   var vm = this;
11   vm.guidelines = [];
12   vm.getGuidelines = getGuidelines;
13
14   vm.guideline = {};
15   vm.getGuideline = getGuideline;
16
17   vm.errorMsg = false;
18
19   activate();
20
21   function activate() {
22     return getGuidelines().then(function() {
23       vm.checked = SharedProperties.getChecked();
```

```
24     })
25 }
26
27 function getGuidelines() {
28     return guidelinesFactory.getGuidelines().then(
29         function (data) {
30             vm.guidelines = data;
31             return vm.guidelines;
32         },
33         function (error) {
34             vm.errorMsg = error.error;
35         }
36     );
37 }
38
39 function getGuideline (guideId) {
40     guidelineFactory.getGuideline(guideId).then(
41         function (data) {
42             SharedProperties.setChecked(guideId);
43             vm.guideline = data;
44         },
45         function (error) {
46             vm.errorMsg = error.error;
47         }
48     );
49 }
50
51 }
```

En este controlador es donde definimos las propiedades que estarán enlazadas con la vista, en este caso `vm .guidelines` que contendrá las guías almacenadas en el *backend*. Para hacer la petición de las guías, a este controlador se le inyecta un servicio (`guidelinesFactory`), que se ha definido *a priori*, que es el encargado de realizar la petición correspondiente a la API REST proporcionada por el CKM. Como vemos a continuación, este servicio cuenta con un método `getGuidelines()` que recupera las guías clínicas.

```
1 angular.module('app.services')
2     .factory('guidelinesFactory', guidelinesFactory);
3
4 function guidelinesFactory($http, API_URL, $q) {
5
6     var guidelines = [];
7
8     return {
9         getGuidelines: getGuidelines
10    }
11
12    function getGuidelines() {
13        var deferred = $q.defer();
14        $http.get(API_URL + '/guidelines').then(
15            function (response) {
16                guidelines = response.data;
17                deferred.resolve(guidelines);
18            },
19            function (response) {
20                deferred.reject(response.data);
21            }
22        );
23        return deferred.promise;
24    }
25 }
```

A su vez, este servicio, hace uso de los servicios nativos de Angular \$http y \$q. El primero se usa para la comunicación con servidores HTTP remotos y el segundo para la ejecución de funciones asíncronas y la utilización de los valores devueltos por dichas funciones. Se trata de una implementación compatible con [Promises/A+](#), una implementación de promesas/objetos diferidos basado en [Q de Kris Kowal](#).

Lo que se ha obtenido en esta primera iteración es la definición de la estructura global del proyecto y la funcionalidad básica de la primera vista. En siguientes iteraciones, por tratarse de funcionalidades más complejas se omitirán capturas de código de servicios y controladores. Después de esta iteración, un usuario tiene la posibilidad de visualizar las guías contenidas en el servidor.

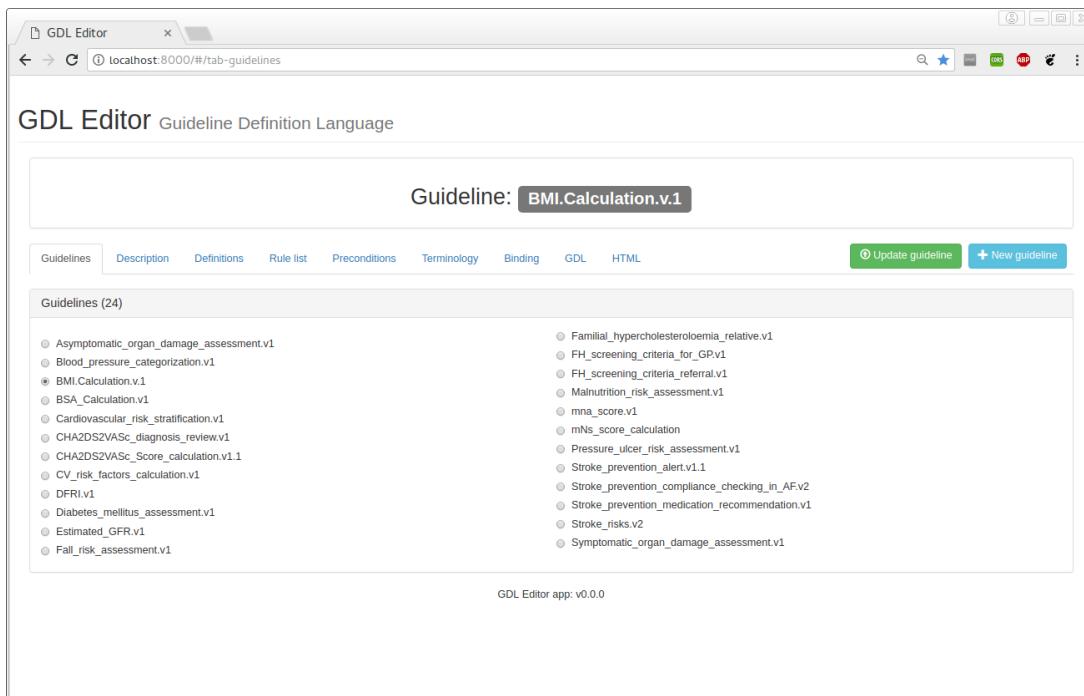


Figura 21: Iteración 1: funcionalidad visualización guías clínicas.

5.1.4. Pruebas

Una explicación más general sobre las tecnologías que se han utilizado para los tests de este proyecto se pueden ver en la Sección 6.

En esta sección nos centraremos en los tests de cada una de las iteraciones, comenzando por la configuración necesaria para la ejecución de los tests. Como se ha mencionado, para los tests unitarios se ha utilizado el *testrunner* Karma, el cual necesita un fichero de configuración denominado karma.conf.js que tiene que estar en el directorio raíz del proyecto. A continuación se muestra el contenido de dicho fichero para nuestro proyecto.

```

1 //jshint strict: false
2 module.exports = function(config) {
3   config.set({
4
5     basePath: './app',
6
7     files: [
8       // First: Load AngularJS
9       { pattern: 'bower_components/angular/angular.js', watched: false },
10      // Then angular modules
11      { pattern: 'bower_components/angular-route/angular-route.js', watched: false },
12      { pattern: 'bower_components/angular-mocks/angular-mocks.js', watched: false },
13      // Then other libraries

```

```

14     { pattern: 'bower_components/angular-ui-router/release/angular-ui-router.min.js', ←
15         watched: false },
16     { pattern: 'bower_components/angular-bootstrap/ui-bootstrap-tpls.min.js', watched: ←
17         false },
18     { pattern: 'bower_components/karma-read-json/karma-read-json.js', watched: false },
19     { pattern: 'assets/**/*.json', included: false },
20     // Then the app scripts
21     'app.js',
22     'config/**/*.js',
23     'components/**/!(*.spec).js',
24     // Finally, load the tests
25     'components/**/*.spec.js'
26   ],
27
28   autoWatch: true,
29
30   frameworks: ['jasmine'],
31
32   browsers: ['Chrome'],
33
34   plugins: [
35     'karma-chrome-launcher',
36     'karma-firefox-launcher',
37     'karma-jasmine',
38     'karma-junit-reporter'
39   ],
40
41   junitReporter: {
42     outputFile: 'test_out/unit.xml',
43     suite: 'unit'
44   }
45
46 });
47
48 );
49
50 );
51
52 );
53
54 );
55
56 );
57
58 );
59
60 );
61
62 );
63
64 );
65
66 );
67
68 );
69
70 );
71
72 );
73
74 );
75
76 );
77
78 );
79
80 );
81
82 );
83
84 );
85
86 );
87
88 );
89
90 );
91
92 );
93
94 );
95
96 );
97
98 );
99
100 );
101 );
102 );
103 );
104 );
105 );
106 );
107 );
108 );
109 );
110 );
111 );
112 );
113 );
114 );
115 );
116 );
117 );
118 );
119 );
120 );
121 );
122 );
123 );
124 );
125 );
126 );
127 );
128 );
129 );
130 );
131 );
132 );
133 );
134 );
135 );
136 );
137 );
138 );
139 );
140 );
141 );
142 );
143 );
144 );
145 );

```

En la configuración se pueden observar diferentes elementos que requieren mención. `basePath` se utiliza para indicar una ubicación de la ruta raíz que será usada para resolver todas las rutas relativas definidas en los patrones, como por ejemplo ficheros (`files`), exclusiones (`exclude`), etc. `files` es un array que contiene todos los ficheros necesarios para ejecutar un test, son los ficheros (o los patrones) que se cargarán en el navegador. `autoWatch` se utiliza para habilitar o deshabilitar la auto-observación de ficheros, de tal modo que los tests se estén ejecutando continuamente cada vez que se guarda un fichero. `frameworks` es un array que contiene la lista de `frameworks` de tests que se quieren utilizar, en nuestro caso sólo utilizaremos el framework Jasmine¹³. En `browsers` se indica una lista de navegadores para iniciar y capturar. Cuando Karma se inicia, también se iniciará cada navegador que se indique dentro de este elemento. Una vez que Karma se cierre, también cerrará estos navegadores. Puede capturar manualmente cualquier navegador abriendo el navegador y visitando la URL donde está escuchando el servidor web Karma (por defecto es <http://localhost:9876/>). El ajuste `plugins` se utiliza para indicar una lista de plugins a cargar (ver `plugins` para más información). Por último, `junit-reporter` es uno de los plugins cargados donde le indicamos donde queremos que nos almacene los informes generados.

Para la primera iteración se han realizado tests unitarios para comprobar que tanto el controlador como el servicio hacen lo que tienen que hacer, así como un test *end-to-end* sencillo que se asegura que el enruteado está funcionando correctamente.

Testeando el controlador `tab.guidelines.controller.js`

A continuación se detalla el procedimiento de pruebas para el controlador `tab.guidelines.controller.js` que se ha utilizado para la obtención de las guías clínicas. La librería `angular-mocks` que se ha añadido en el fichero `karma.conf` proporciona un servicio llamado `$controller` que se puede utilizar para testear un controlador. En el `beforeEach()` que se muestra más abajo, se inyecta el servicio `$controller` con las dependencias necesarias para instanciar dicho controlador.

¹³ Framework de tests open source para AngularJS: <https://jasmine.github.io/>

Siguiendo la guía de buenas prácticas, al fichero que contiene los tests unitarios para testear el controlador `tab.guidelines.controller.js` le hemos llamado `tab.guidelines.controller.spec.js`

```
1 'use strict';
2
3 describe('Guidelines functionality:', function() {
4
5     beforeEach(module('app.controllers'));
6     beforeEach(module('app.services'));
7     beforeEach(module('app.constants'));
8
9     // Test data
10    var mock = {};
11    mock.guidelines = readJSON('assets/mocks/guidelines.json');
12
13    /*
14     * Mock our factory and spy on methods
15     */
16    var deferred;
17    beforeEach(inject(function($q, _guidelinesFactory_) {
18        deferred = $q.defer();
19        mock.guidelinesFactory = _guidelinesFactory_;
20        spyOn(mock.guidelinesFactory, 'getGuidelines').and.returnValue(deferred.promise);
21    }));
22
23    /*
24     * Instantiate controller using $controller service
25     */
26    var scope, guidelineCtrl;
27    beforeEach(inject(function($controller, $rootScope) {
28        // Controller setup
29        scope = $rootScope.$new();
30        guidelineCtrl = $controller('GuidelineCtrl', { guidelinesFactory: mock.←
31            guidelinesFactory });
32    }));
33
34    describe('initialization', function() {
35        it('initializes with proper $scope variables and methods', function() {
36            scope.$apply();
37            expect(guidelineCtrl.guidelines).toEqual([]);
38            expect(guidelineCtrl.errorMsg).toEqual(false);
39        });
40    });
41
42    describe('getGuidelines()', function() {
43
44        var testErrorMessage = 'Error at getting the list of guidelines';
45        var testResponseSuccess = { success: true, data: mock.guidelines };
46        var testResponseFailure = { error: testErrorMessage };
47
48        it('successfully gets the list of guidelines', function() {
49            expect(mock.guidelinesFactory.getGuidelines).toBeDefined();
50            // Perform the action
51            deferred.resolve(mock.guidelines);
52            scope.$apply(function() {
53                guidelineCtrl.getGuidelines();
54            });
55            // Run expectations
56            expect(mock.guidelinesFactory.getGuidelines).toHaveBeenCalled();
57            expect(guidelineCtrl.guidelines).toBe(testResponseSuccess.data);
58        });
59    });
60});
```

```
59
60     it('fails to get the list of guidelines and displays an error message', function() {
61         deferred.reject(testResponseFailure);
62         // Perform the action
63         scope.$apply(function() {
64             guidelineCtrl.getGuidelines();
65         });
66         // Run expectations
67         expect(mock.guidelinesFactory.getGuidelines).toHaveBeenCalled();
68         expect(guidelineCtrl.errorMsg).toEqual(testResponseFailure.error);
69     });
70
71 }) ;
72
73 }) ;
```

En el anterior test se comprueban varias cosas, por un lado que la inicialización del controlador se hace correctamente, obteniéndose un array vacío en las guías antes de la ejecución del método `getGuidelines()` y que el mensaje de error en caso de fallo sea igual a `false`. Por otro lado se comprueba que el método `getGuidelines()` funciona correctamente, tanto si obtienen las guías de forma satisfactoria (`deferred.resolve`) como que se muestra el mensaje de error correspondiente cuando, por algún motivo, se ha producido un error (`deferred.reject`).

Testeando el servicio tab.guidelines.service.js

A continuación se detalla el procedimiento de pruebas para el servicio `tab.guidelines.service.js`, que hace una petición HTTP para obtener las guías clínicas disponibles en el servidor. Siempre que se utiliza el servicio nativo `$http` para hacer una llamada remota, Angular utiliza por detrás otro servicio llamado `$httpBackend` que es el que realmente hace el trabajo duro. La librería `angular-mocks` que se ha añadido al fichero `karma.conf` cuenta con su propia versión de `$httpBackend` con ciertas ventajas que nos ayudan a simular llamadas al *backend* real. Siguiendo la guía de buenas prácticas, al fichero que contiene los tests unitarios para testear el servicio `tab.guidelines.service.js` le hemos llamado `tab.guidelines.service.spec.js`.

```
1 'use strict';
2
3 describe('GuidelinesFactory', function() {
4
5     beforeEach(module('app.constants'));
6     beforeEach(module('app.services'));
7
8     var guidelinesFactory, httpBackend, q, baseUrl;
9
10    beforeEach(inject(function(_guidelinesFactory_, $httpBackend, _API_URL_) {
11        // Factory instance and dependencies
12        guidelinesFactory = _guidelinesFactory_;
13        httpBackend = $httpBackend;
14        baseUrl = _API_URL_;
15    }));
16
17    afterEach(function() {
18        httpBackend.verifyNoOutstandingExpectation();
19        httpBackend.verifyNoOutstandingRequest();
20    });
21
22
23 describe('initialization', function() {
24
25     it('is defined', function() {
26         expect(guidelinesFactory).toBeDefined();
27     });
28 })
```

```
29 });
30
31
32 describe('getGuidelines()', function() {
33
34     it('get the list of all guidelines', function() {
35         var promise, response, result;
36         // Test data
37         var testGuidelines = readJSON('assets/mocks/guidelines.json');
38         // Make the request and implement a fake success callback
39         promise = guidelinesFactory.getGuidelines();
40         promise.then(function(data) {
41             result = data;
42         });
43         response = {
44             success: true,
45             data: testGuidelines
46         };
47         httpBackend.expectGET(baseUrl + "/guidelines").respond(200, response); // Expect a ←
48             GET request and send back a canned response
49         httpBackend.flush(); // Flush pending requests
50         expect(result).toEqual(response);
51         expect(result.data).toEqual(testGuidelines);
52         expect(result.data.length).toBe(24);
53
54     });
55
56     it('fails to get the list of all guidelines', function() {
57         var promise, response, result, errorMessage;
58         errorMessage = 'Error at getting guidelines';
59         promise = guidelinesFactory.getGuidelines();
60         promise.then(null, function(error) {
61             result = error;
62         });
63         response = {
64             error: errorMessage
65         };
66         httpBackend.expectGET(baseUrl + "/guidelines").respond(500, response); // Expect a ←
67             GET request and send back a server failed canned response
68         httpBackend.flush(); // Flush pending requests
69         expect(result).toEqual(response);
70         expect(result.error).toEqual(errorMessage);
71     });
72
73 }) );
```

En el anterior test se comprueban varias cosas, por un lado que la inicialización del servicio no es `undefined`. Si se llama a `getGuidelines()` desde el test, hará una petición HTTP real al *endpoint* indicado. Sin embargo, lo que hacemos es interceptar dicha llamada con `$httpBackend` y definimos la respuesta en vez de hacer la llamada remota real. Obsérvese que es necesario hacer `$httpBackend.flush()` ya que la llamada `$http` normalmente es asíncrona, pero en el test lo queremos ejecutar de forma síncrona. La llamada a `flush()` asegura que el `.then()` de la promesa devuelta por `$http` sea ejecutada inmediatamente.

5.2. Iteración 2

Texto análisis iteración 2

5.2.1. Análisis

Texto análisis iteración 2

5.2.2. Diseño

Texto diseño iteración 2

5.2.3. Implementación

Texto implementación iteración 1

5.2.4. Pruebas

Texto pruebas iteración 2

6. Testing

6.1. Jasmine Framework

hablar del framework Jasmine (utilizado en los dos siguientes testrunners)

6.2. Tests unitarios con Karma

Utilización de Karma test runner para probar nuestros servicios y controladores.

6.3. Tests end-to-end con Protractor

Hablar del framework de tests Protractor y mencionar cómo lo hemos aplicado para realizar pruebas end-to-end en nuestro proyecto.

7. Material utilizado

- Sistema operativo Ubuntu 14.04, GNOME 3.
- IDE JetBrain WebStorm.
- NodeJS como entorno de ejecución de la aplicación cliente.
- SGBD PostgreSQL.
- Servidor de aplicaciones JBoss Server EAP 6.2.
- Astah Community Edition como herramienta de modelado UML.
- AsciiDoc como herramienta de generación de documentos.
- Git/Github como sistema de control de versiones y repositorio remoto de código.
- Open Project como software de gestión de proyectos.
- Travis CI como herramienta de integración continua.

8. Implantación y seguimiento (si corresponde)

Texto implantación y seguimiento

9. Planificación y costes (planificación inicial vs. final) - (Opcional)

Texto planificación y costes

10. Resultados y discusión

Mencionar y discutir los resultados obtenidos, tanto los que se pudieron abordar como los que no.

11. Conclusiones

Conclusiones finales sobre el proyecto.

12. Futuros trabajos

La herramienta supone el punto de partida hacia una serie de nuevas funcionalidades de gran utilidad para la comunidad, como la edición colaborativa en tiempo real, la resolución de conflictos, la gestión del historial y de versiones de las guías clínicas, etc.

13. Bibliografía

13.1. Referencias

- [1] [keen] Keen, P. G. W. Decision support systems: an organizational perspective. Addison-Wesley. ISBN 0-201-03667-3.
- [2] [node] Web oficial de Node.js. URL: <https://nodejs.org/es/>. Último acceso agosto 2017.
- [3] [cantelon] Cantelon M, Harter M, Holowaychuk TJ, Rajlich N. Node.js in Action. Manning. ISBN 9781617290572
- [4] [bonis] Bonis Julio, Sancho Juan J., Sanz Ferrán. Computer-assisted Clinical Decision Support Systems. Clinical Medicine, Elsevier Ed. 2004;122 Supl 1:39-44.
- [5] [bleich] Bleich HL. Computer evaluation of acid-base disorders. J Clin Invest, 48 (1969), pp. 1689-96 <http://dx.doi.org/10.1172/JCI106134>
- [6] [warner] Warner HR, Toronto AF, Veasey LG, Stephenson RA. Mathematical approach to medical diagnosis. JAMA, 177 (1961), pp. 75-81.
- [7] [web-app] Wikipedia en español. URL: https://es.wikipedia.org/wiki/Aplicaci%C3%B3n_web. Último acceso: junio 2017.
- [8] [module] Documentación oficial de AngularJS. URL: <https://docs.angularjs.org/guide/module>. Último acceso: junio 2017.
- [9] [directive] Documentación oficial de AngularJS. URL: <https://docs.angularjs.org/guide/directive>. Último acceso: junio 2017.

- [10] [twdb] Documentación oficial de AngularJS. URL: <https://docs.angularjs.org/guide/databinding>. Último acceso: junio 2017.
- [11] [scrum] Deemer P, Benefield G, Larman C, Vodde B. The Scrum Primer. 2012.

14. Glosario

Glosario utilizado en el documento, con respectivos significados

Historia Clínica Electrónica

Registro electrónico de información demográfica, preventiva y médica de un paciente.

Document Object Model

Interfaz, independiente del lenguaje y plataforma que permite a programas y scripts acceder dinámicamente y actualizar el contenido, estructura y estilos de documentos HTML, XHTML y XML.

Clinical Knowledge Manager

Gestor de Conocimiento Clínico. Backend con una API bien definida donde se gestiona el conocimiento clínico del sistema: arquetipos clínicos, guías médicas, terminologías, etc.

A. Apéndices

A continuación se detallan los apéndices referenciados durante la memoria de este Trabajo Fin de Máster.

A.1. Gramática de GDL

La gramática y la especificación léxica para las expresiones utilizadas por GDL se basa en líneas generales en la sintaxis de aserción en la especificación ADL. Esta gramática se implementa utilizando las especificaciones de [javaCC](#) en el entorno de programación Java. El código fuente completo del analizador GDL de Java se puede encontrar a continuación.

```
/**  
 * Expression parser  
 *  
 */  
options  
{  
    JDK_VERSION = "1.5";  
  
    LOOKAHEAD= 1;  
    DEBUG_PARSER = false;  
    DEBUG_TOKEN_MANAGER = false;  
    DEBUG_LOOKAHEAD = false;  
    UNICODE_INPUT = true;  
  
    static = false;  
}  
  
PARSER_BEGIN(ExpressionParser)  
package se.cambio.cds.gdl.parser;  
import java.io.*;  
import java.util.*;  
import se.cambio.cds.gdl.model.expression.*;  
import org.openehr.rm.datatypes.text.CodePhrase;  
import org.openehr.rm.datatypes.quantity.*;  
import org.openehr.rm.datatypes.basic.DataValue;
```

```
public class ExpressionParser
{
    private static final String CHARSET = "UTF-8";

    /* ====== public interface ===== */
    /* execute the parsing */
    public List < ExpressionItem > parseBooleanExpressions() throws ParseException
    {
        return expressions();
    }

    public List < ExpressionItem > parseArithmeticExpressions() throws ParseException
    {
        return expressions();
    }

    public ExpressionItem parse() throws ParseException
    {
        return expression_item();
    }

    /* re-initial the parser */
    public void reInit(File file) throws IOException
    {
        ReInit(new FileInputStream(file), CHARSET);
    }

    /* re-initial the parser */
    public void reInit(InputStream input) throws IOException
    {
        ReInit(new BufferedInputStream(input));
    }

    public static void main(String args []) throws ParseException
    {
    }
}

PARSER_END (ExpressionParser)

<*>
SKIP : /* WHITE SPACE */
{
    " "
    | "\t"
    | "\n"
    | "\r"
    | "\f"
}

<*>
SPECIAL_TOKEN : /* COMMENTS */
{
    < SINGLE_LINE_COMMENT : "--" (~[ "\n", "\r" ])* >
}

<*>
TOKEN : /* SYMBOLS - common */
{
    < SYM_MINUS : "-" >
    | < SYM_PLUS : "+" >
    | < SYM_STAR : "*" >
}
```

```
| < SYM_SLASH : "/" >
| < SYM_CARET : "^" >
| < SYM_DOT : "." >
| < SYM_SEMICOLON : ";" >
| < SYM_COMMA : "," >
| < SYM_TWO_COLONS : "::" >
| < SYM_COLON : ":" >
| < SYM_EXCLAMATION : "!" >
| < SYM_L_PARENTHESIS : "(" >
| < SYM_R_PARENTHESIS : ")" >
| < SYM_DOLLAR : "$" >
| < SYM_QUESTION : "?" >
| < SYM_L_BRACKET : "[" >
| < SYM_R_BRACKET : "]" >
| < SYM_INTERVAL_DELIM : "|" >
| < SYM_EQ : "==" >
| < SYM_GE : ">=" >
| < SYM_LE : "<=" >
| < SYM_LT : "<" >
| < SYM_GT : ">" >
| < SYM_NE : "!=" >
| < SYM_NOT : "not" >
| < SYM_AND :
  "and"
  | "&&" >
| < SYM_OR :
  "or"
  | "||" >
| < SYM_FALSE : "false" >
| < SYM_TRUE : "true" >
| < SYM_NULL : "null" >
| < SYM_IS_A : "is_a" >
| < SYM_IS_NOT_A : "!is_a" >
| < SYM_FOR_ALL : "for_all" >
| < SYM_MAX : "max" >
| < SYM_MIN : "min" >
| < SYM_CURRENT_DATETIME : "currentDateTime" >
| < SYM_ASSIGNMENT : "=" >
| < SYM_MODULO : "\\" >
| < SYM_DIV : "//" >
| < SYM_ELLIPSIS : ".." >
| < SYM_LIST_CONTINUE : "..." >
}

<* >

TOKEN :
{
  < #V_LOCAL_CODE_CORE : "g" [ "c", "t" ] ([ "0"--"9", "."])+ [ "0"--"9" ] >
| < V_LOCAL_CODE : < V_LOCAL_CODE_CORE > >
| < V_QUANTITY :
  (
    < V_REAL >
    | < V_INTEGER >
  )
  , " ([ "a"--"z", "A"--"Z", "$\mathbf{\mu}$", "\textdegree{ }", "%", "*", "0"--"9", "[", ←
    "] ", "/" ]) +
  ([ "a"--"z", "A"--"Z", "$\mathbf{\mu}$", "\textdegree{ }", "%", "*", "0"--"9", "[", "]", ←
    "/" ]) *
  ([ "a"--"z", "A"--"Z", "$\mathbf{\mu}$", "\textdegree{ }", "%", "*", "0"--"9", "[", ←
    "] "]) *
| < V_PROPORTION :
  (
```

```
< V_REAL >
| < V_INTEGER >
)
",
(
< V_REAL >
| < V_INTEGER >
)
",
(
["0"--"4"]) >
| < V_INTEGER :
(< DIG >)+
|
"(-" (< DIG >)+ ") "
|
(< DIG >
{
  1, 3
}
(
  ", " (< DIG >)
  {
    3
  }
)+ >
|
< V_ISO8601_DURATION: ("--") ? "P" ((<DIG>)+["Y", "Y"]) ? ((<DIG>)+["m", "M"]) ? ((<DIG>)+["w", "W" ←
  ""]) ?
((<DIG>)+["d", "D"]) ? ("T" ((<DIG>)+["h", "H"]) ? ((<DIG>)+["m", "M"]) ?
((<DIG>)+["s", "S"]) ?) >
|
< V_ISO8601_DURATION_CONSTRAINT_PATTERN: "P" ([ "Y", "Y" ]) ? ([ "m", "M" ]) ?
([ "w", "W" ]) ? ([ "d", "D" ]) ? "T" ([ "h", "H" ]) ? ([ "m", "M" ]) ? ([ "s", "S" ]) ?
|"P" ([ "Y", "Y" ]) ? ([ "m", "M" ]) ? ([ "w", "W" ]) ? ([ "d", "D" ]) ?>
|
< V_DATE: ([ "0"--"9" ]){4} "--" ( "0"["1"--"9"] | "1"["0"--"2" ] ) "--"
  ( "0"["1"--"9" ] | ["1"--"2"] ["0"--"9" ] | "3"["0"--"1" ] ) >
|
< V_HHMM_TIME: <HOUR_MINUTE> >
|
< V_HHMMSS_TIME: < HOUR_MINUTE> <SECOND> >
|
< V_HHMMSSs_TIME: < HOUR_MINUTE> <SECOND> <MILLI_SECOND> >
|
< V_HHMMSSZ_TIME: < HOUR_MINUTE> <SECOND> <TIME_ZONE> >
|
< V_HHMMSSssZ_TIME: < HOUR_MINUTE> <SECOND> <MILLI_SECOND> <TIME_ZONE> >
|
< V_TIME: <HOUR_MINUTE> <SECOND> >
|
< V_DATE_TIME_MS: <DATE_TIME> <MILLI_SECOND> >
|
< V_DATE_TIME_Z: "("<DATE_TIME> <TIME_ZONE> ")" >
|
< V_DATE_TIME: "("<DATE_TIME> ")" >
|
< V_DATE_TIME_MSZ: <DATE_TIME> <MILLI_SECOND> <TIME_ZONE> >
|
< #DATE_TIME: <V_DATE>"T"<V_TIME>>
|
< #TIME_ZONE: [ "--", "+" ] ([ "0"--"9" ]){2} ":" ([ "0"--"9" ]){2} | "Z" >
```

```
< #SECOND: ":" ["0"--"5"] ["0"--"9"] >
|
| < #MILLI_SECOND: "."(["0"--"9"]) {2, 3} >
|
| < #HOUR_MINUTE: ["0"--"9"] ["0"--"9"] ":" ["0"--"5"] ["0"--"9"] >
| < V_CODE_PHRASE : "[" (< LET_DIG_DUDSLR >)+ ":" (< LET_DIG_DUDS >)+ "]" >
| < V_CODE_PHRASE_RAW : (< LET_DIG_DUDSLR >)+ ":" (< LET_DIG_DUDS >)+ >
| < V_ORDINAL : < V_INTEGER > "|" < V_CODE_PHRASE_RAW > < V_LABEL > >
| < V_ATTRIBUTE_IDENTIFIER : [ "a"--"z" ] (< LET_DIG_U >) * >
| < V_LABEL : "|" (~[ "|" ]) * "|" >
| < V_REAL :
  (< DIG >)+ "./" ~[ ".", "0"--"9" ]
  | (< DIG >)+ "." (< DIG >) * [ "e", "E" ] ([ "+" , "-" ]) ? (< DIG >)+
  | (< DIG >) * "." (< DIG >)+
  (
    [ "e", "E" ] ([ "+" , "-" ]) ? (< DIG >)+
  ) ?
  | "(-" (< DIG >) * "." (< DIG >)+
  (
    [ "e", "E" ] ([ "+" , "-" ]) ? (< DIG >)+
  ) ? ")"
| (< DIG >)
{
  1, 3
}
(
  "_" (< DIG >)
  {
    3
  }
) +
"./" ~[ ".", "0"--"9" ]
| (< DIG >)
{
  1, 3
}
(
  "_" (< DIG >)
  {
    3
  }
) *
"."
(
  (< DIG >)
  {
    1, 3
  }
(
  "_" (< DIG >)
  {
    3
  }
) *
) ?
[ "e", "E" ] ([ "+" , "-" ]) ? (< DIG >)
{
  1, 3
}
(
  "_" (< DIG >)
  {
```

```
        3
    }
  ) *
|
(
  (< DIG >
  {
    1, 3
  }
  (
    "_" (< DIG >
    {
      3
    }
  ) *
  ) ?
  "." (< DIG >
  {
    1, 3
  }
  (
    "_" (< DIG >
    {
      3
    }
  ) *
  (
    [ "e", "E" ] ([ "+" , "-" ]) ? (< DIG >
    {
      1, 3
    }
    (
      "_" (< DIG >
      {
        3
      }
    ) *
    ) ?
  ) ? >
| < V_STRING :
  """
  (
    (
      "\\\\" ( ~[ "\\'", "\n", "\\\\" ] ) *
    )
  |
    (
      "\\\\\" ( ~[ "\\'", "\n", "\\\\" ] ) *
  |
    (
      "\n" ([ "\r", " ", "\t" ]) *
  | (~[ "\\", "\n", "\'" ] ) *
) *
"""
>
}

<*>
TOKEN : /* LOCAL TOKENS */
{
  < #DIG : [ "0"-"9" ] >
| < #LET_DIG : [ "a"-"z", "A"-"Z", "0"-"9" ] >
```

```
| < #LET_DIG_DD :
|   < LET_DIG >
|   | "."
|   | "_"
|   | "-"
| < #LET_DIG_U :
|   < LET_DIG >
|   | "_"
| < #LET_DIG_DU :
|   < LET_DIG_U >
|   | "-"
| < #LET_DIG_DUDS :
|   < LET_DIG_DU >
|   | "."
|   | "\\" >
| < #LET_DIG_DUDSLR :
|   < LET_DIG_DUDS >
|   | "("
|   | ")" >
| < V_LOCAL_TERM_CODE_REF : "[" < LET_DIG > (< LET_DIG_DD >)* "]" >
| < #PATH_SEGMENT : < V_ATTRIBUTE_IDENTIFIER > (< V_LOCAL_TERM_CODE_REF >)? >
| < V_ABSOLUTE_PATH : < SYM_SLASH > < PATH_SEGMENT > (< SYM_SLASH > < PATH_SEGMENT >)* >
}

List < ExpressionItem > expressions() :
{
    List < ExpressionItem > items = new ArrayList < ExpressionItem > ();
    ExpressionItem item = null;
}
{
    item = expression_item()
    {
        items.add(item);
    }
    (
        LOOKAHEAD(2)
        < SYM_COMMA > item = expression_item()
        {
            items.add(item);
        }
    )*
    {
        return items;
    }
}

ExpressionItem expression_item() :
{
    ExpressionItem item = null;
}
{
    (
        LOOKAHEAD(4)
        item = expression_node()
    | LOOKAHEAD(4)
        item = expression_leaf()
    )
    {
        return item;
    }
    {
        return item;
    }
}
```

```
        }
    }

CodePhrase code_phrase() :
{
    Token t;
    String lang = null;
    String langTerm = null;
    String langCode = null;
}
{
    t = < V_CODE_PHRASE >
    {
        lang = t.image;
        int i = lang.indexOf("::");
        langTerm = lang.substring(1, i);
        langCode = lang.substring(i + 2, lang.length() - 1);
    }
    {
        return new CodePhrase(langTerm, langCode);
    }
}

CodePhrase code_phrase_raw() :
{
    Token t;
    String lang = null;
    String langTerm = null;
    String langCode = null;
}
{
    t = < V_CODE_PHRASE_RAW >
    {
        lang = t.image;
        int i = lang.indexOf("::");
        langTerm = lang.substring(0, i);
        langCode = lang.substring(i + 2);
    }
    {
        return new CodePhrase(langTerm, langCode);
    }
}

/* ----- expressions ----- */
ExpressionItem expression_node() :
{
    ExpressionItem ret = null;
    ExpressionItem item = null;
    ExpressionItem item2 = null;
    OperatorKind op = null;
    List<AssignmentExpression> assignmentExpressions = null;
    boolean precedenceOverridden = false; // TODO
    Token t = null;
    String attrId = null;
}
{
    (
        (
            < SYM_FOR_ALL > item = expression_leaf()
            {
                op = OperatorKind.FOR_ALL;
            }
        )
    )
}
```

```
| < SYM_MAX > item = expression_leaf()
| {
|     op = OperatorKind.MAX;
| }
| < SYM_MIN > item = expression_leaf()
| {
|     op = OperatorKind.MIN;
| }
|
| {
|     return new UnaryExpression(item, op);
| }
|
| (
|     item = expression_leaf()
|     (
|         < SYM_EQ >
|         {
|             op = OperatorKind.EQUALITY;
|         }
|         | < SYM_NE >
|         {
|             op = OperatorKind.INEQUAL;
|         }
|         | < SYM_LT >
|         {
|             op = OperatorKind.LESS_THAN;
|         }
|         | < SYM_GT >
|         {
|             op = OperatorKind.GREATER_THAN;
|         }
|         | < SYM_LE >
|         {
|             op = OperatorKind.LESS_THAN_OR_EQUAL;
|         }
|         | < SYM_GE >
|         {
|             op = OperatorKind.GREATER_THAN_OR_EQUAL;
|         }
|         | < SYM_PLUS >
|         {
|             op = OperatorKind.ADDITION;
|         }
|         | < SYM_MINUS >
|         {
|             op = OperatorKind.SUBSTRATION;
|         }
|         | < SYM_STAR >
|         {
|             op = OperatorKind.MULTIPLICATION;
|         }
|         | < SYM_SLASH >
|         {
|             op = OperatorKind.DIVISION;
|         }
|         | < SYM_CARET >
|         {
|             op = OperatorKind.EXPONENT;
|         }
|         | < SYM_AND >
|         {
```

```
        op = OperatorKind.AND;
    }
| < SYM_OR >
{
    op = OperatorKind.OR;
}
| < SYM_IS_A >
{
    op = OperatorKind.IS_A;
}
| < SYM_IS_NOT_A >
{
    op = OperatorKind.IS_NOT_A;
}
| LOOKAHEAD(4)
< SYM_ASSIGNMENT >
{
    op = OperatorKind.ASSIGNMENT;
}
item2 = expression_leaf()
{
    return new AssignmentExpression((Variable) item, item2);
}
| assignmentExpressions = assignmentExpressions()
{
    return new CreateInstanceExpression((Variable) item, assignmentExpressions());
}
item2 = expression_leaf()
{
    ret = new BinaryExpression(item, item2, op);
}
)
)
{
    return ret;
}
}

ExpressionItem expression_leaf() :
{
    ExpressionItem item = null;
    Token t = null;
}
{
(
< SYM_L_PARENTHESIS >
(
    LOOKAHEAD(expression_node())
    item = expression_node()
| LOOKAHEAD(variable())
    item = variable()
| LOOKAHEAD(constant_expression())
    item = constant_expression()
)
< SYM_R_PARENTHESIS >
| item = constant_expression()
| item = variable()
)
{
    return item;
}
```

```
}

AssignmentExpression assignmentExpression() :
{
    ExpressionItem item = null;
    ExpressionItem item2 = null;
}
{
(
    item = expression_leaf()
    < SYM_ASSIGNMENT >
    item2 = expression_leaf()
)
{
    return new AssignmentExpression((Variable) item, item2);
}
}

List < AssignmentExpression > assignmentExpressions() :
{
    List < AssignmentExpression > items = new ArrayList < AssignmentExpression > ();
    AssignmentExpression item = null;
}
{
(
    < SYM_L_PARENTHESIS >
    item = assignmentExpression()
    {
        items.add(item);
    }
    (
        LOOKAHEAD(2)
        < SYM_SEMICOLON > item = assignmentExpression()
        {
            items.add(item);
        }
    )*
    < SYM_R_PARENTHESIS >
)
{
    return items;
}
}

ConstantExpression constant_expression() :
{
    Token t = null;
    CodePhrase code = null;
    String text = null;
    String units = null;
    Integer order = null;
}
{
(
    t = < V_STRING >
    {
        String str = t.image;
        return new StringConstant(str.substring(1, str.length() - 1));
    }
    | t = < V_ORDINAL >
```

```
{  
    String value = "DV_ORDINAL," + t.image;  
    DvOrdinal ordinal = (DvOrdinal) DataValue.parseValue(value);  
    return new OrdinalConstant(ordinal);  
}  
| t = < V_REAL >  
| t = < V_INTEGER >  
| t = < V_PROPORTION >  
| t = <V_DATE>  
| t = <V_DATE_TIME_Z>  
{  
    text = t.image;  
    text = text.replace("(", "");  
    text = text.replace(")", "");  
    return new DateTimeConstant(text);  
}  
| t = <V_DATE_TIME>  
{  
    text = t.image;  
    text = text.replace("(", "");  
    text = text.replace(")", "");  
    return new DateTimeConstant(text);  
}  
| t = <V_TIME>  
| t = <V_ISO8601_DURATION>  
| t = < SYM_NULL >  
| t = < SYM_TRUE >  
| t = < SYM_FALSE >  
| LOOKAHEAD(2)  
code = code_phrase_raw() [ text = label() ]  
{  
    if (text != null)  
    {  
        return new CodedTextConstant(text, code);  
    }  
    else  
    {  
        return new CodePhraseConstant(code);  
    }  
}  
| t = < V_QUANTITY >  
{  
    text = t.image;  
    text = text.replace("(", "");  
    text = text.replace(")", "");  
    DvQuantity q = new DvQuantity("m", 1, 0).parse(text);  
    return new QuantityConstant(q);  
}  
}  
{  
    return new ConstantExpression(t.image);  
}  
}  
}  
  
Variable variable() :  
{  
    Variable v;  
    Token t;  
    String code = null;  
    String path = null;  
    String label = null;  
    String attribute = null;
```

```
}

{
(
< SYM_DOLLAR >
(
t = < V_LOCAL_CODE >
| t = < SYM_CURRENT_DATETIME >
)
{
code = t.image;
}
[ label = label() ]
| t = < V_ABSOLUTE_PATH >
{
path = t.image;
}
)
[
< SYM_DOT > t = < V_ATTRIBUTE_IDENTIFIER >
{
attribute = t.image;
}
]
{
return new Variable(code, label, path, attribute);
}
}

String label() :
{
Token t;
String label = null;
}
{
t = < V_LABEL >
{
label = t.image;
label = label.substring(1, label.length() - 1);
return label;
}
}

double real() :
{
Token t;
String value = null;
}
{
t = < V_REAL >
{
value = t.image;
return Double.parseDouble(value);
}
}

int integer() :
{
Token t;
String value = null;
}
{
t = < V_INTEGER >
```

```

{
    value = t.image;
    return Integer.parseInt(value);
}
}

```

A.2. Manual de usuario

El editor GDL se divide en 9 pestañas:

- Guidelines: listado de todas las guías disponibles en el *backend*.
- Description: información básica sobre la guía clínica.
- Definitions: referencias a los arquetipos utilizados en las reglas y en las precondiciones.
- Rule list: permite la gestión de todas las reglas dentro de la guía.
- Preconditions: una lista de las condiciones que se tienen que cumplir para que la guía pueda ser ejecutada.
- Terminology: traducciones para cada uno de los términos usados en las guías.
- Binding: mapeo de los códigos locales utilizados en la guía a terminologías externas.
- GDL: la salida del editor (en formato GDL)
- HTML: la salida del editor (en formato HTML)

A.2.1. Guidelines

Se trata de la vista donde se muestra un listado con todas las guías clínicas, se puede seleccionar una guía para ser editada, para ello el usuario se debe mover por las pestañas que se muestran a continuación.

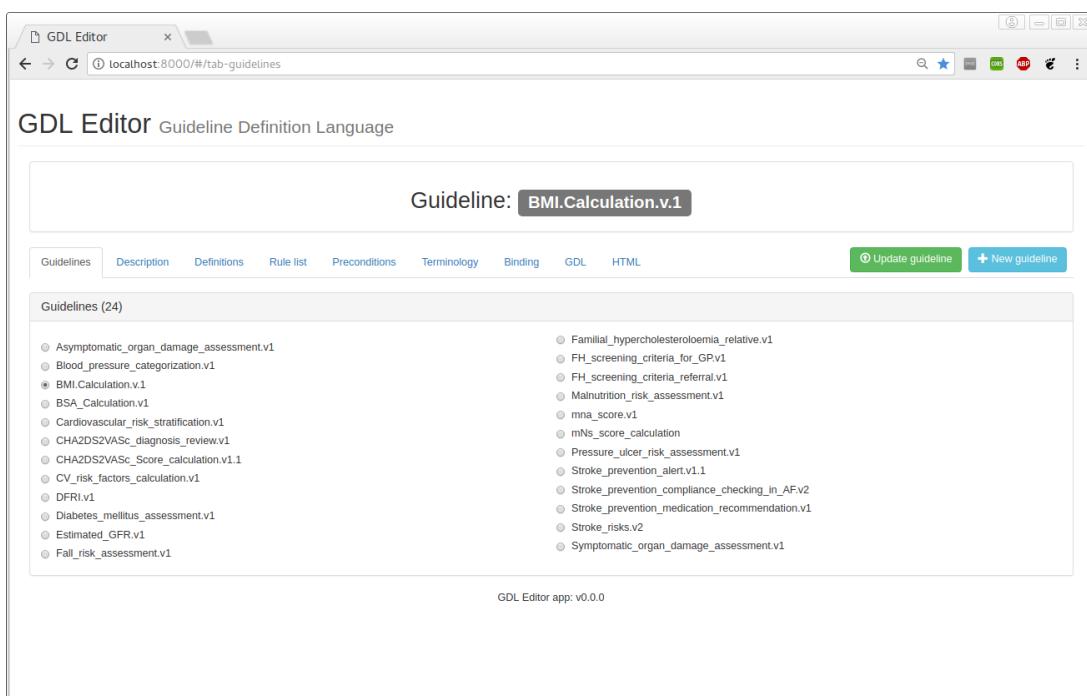


Figura 22: Guidelines: visualización de las guías disponibles.

A.2.2. Description

Esta sección define a grandes rasgos el uso y el propósito de la guía clínica con un conjunto de meta-datos. Incluye el nombre único de la guía, la identificación única del autor responsable de la descripción, que puede incluir la organización a la que pertenece y/o información de contacto; un enunciado formal, en lenguaje natural o codificado, definiendo el ámbito y el propósito clínico de la guía; una lista de palabras clave, médicas o procedimentales, así como un listado de otros colaboradores que hayan aportado trabajo a la guía clínica. Esta sección también puede incluir un enunciado sobre el uso pretendido de la guía y un enunciado sobre posibles usos erróneos o desaconsejados de la misma. La definición de una guía clínica GDL también debe indicar información del *status* de publicación (“Draft”, “Public”, “Deprecated”, etc.)¹⁴ y la fecha en la que se realizó dicha guía con este status de publicación. Por último existe un apartado donde se pueden indicar referencias relevantes para la guía.

The screenshot shows the GDL Editor interface with the 'Description' tab selected for the guideline 'BSA_Calculation.v1'. The form contains the following data:

- Name:** BSA Calculation
- Author details:**
 - Name: Rong Chen
 - Email: rong.chen@cambio.se
 - Organisation: Cambio Healthcare Systems
 - Date: 11/03/2013
- Lifecycle state:** Author draft
- Copyright:** Cambio Healthcare Systems
- Keywords:**
 - body surface area
 - Mosteller formula
- Other contributors:**
 - Konstantinos Kalliamvakos
 - Iago Corbal
- Description:** Body surface area is the measured or calculated surface area of a human body, expressed in square meters.
- Purpose:** To calculate the body surface area based on the Mosteller formula.
- Use:** Use for calculating the BSA based on the Mosteller formula ($\text{Height(cm)} \times \text{Weight(kg)} / 3600$) $^{1/2}$
- Misuse:** Do not use for calculating BSA based on other formulas.
- References:**
 - 1. Mosteller R. Simplified Calculation of Body-Surface Area. N Engl J Med. 1987;317(17):1098.

Figura 23: Description: meta-information de la guía clínica.

A.2.3. Definitions

Las definiciones establecen un enlace entre elementos de arquetipos y los términos utilizados en nuestra guía. Todas las definiciones de la guía se pueden encontrar en la pestaña *Definitions*, y pueden ser creadas desde aquí o directamente desde los paneles de precondiciones/condiciones/acciones que veremos en las siguientes subsecciones.

Para crear una nueva definición en la pestaña *Definitions* (Figura 24), simplemente arrastra y suelte las definiciones (en el lado derecho) que desea insertar. Todos los componentes editables de cada definición se mostrarán como un enlace (azul y subrayado). Para cambiar su valor, simplemente haga clic en él. Para eliminar una definición, haga clic en el segundo botón (rojo).

¹⁴ Ver todos los posibles valores en <https://openehr.atlassian.net/wiki/spaces/healthmod/pages/2949205/Archetype+Publication+Status>

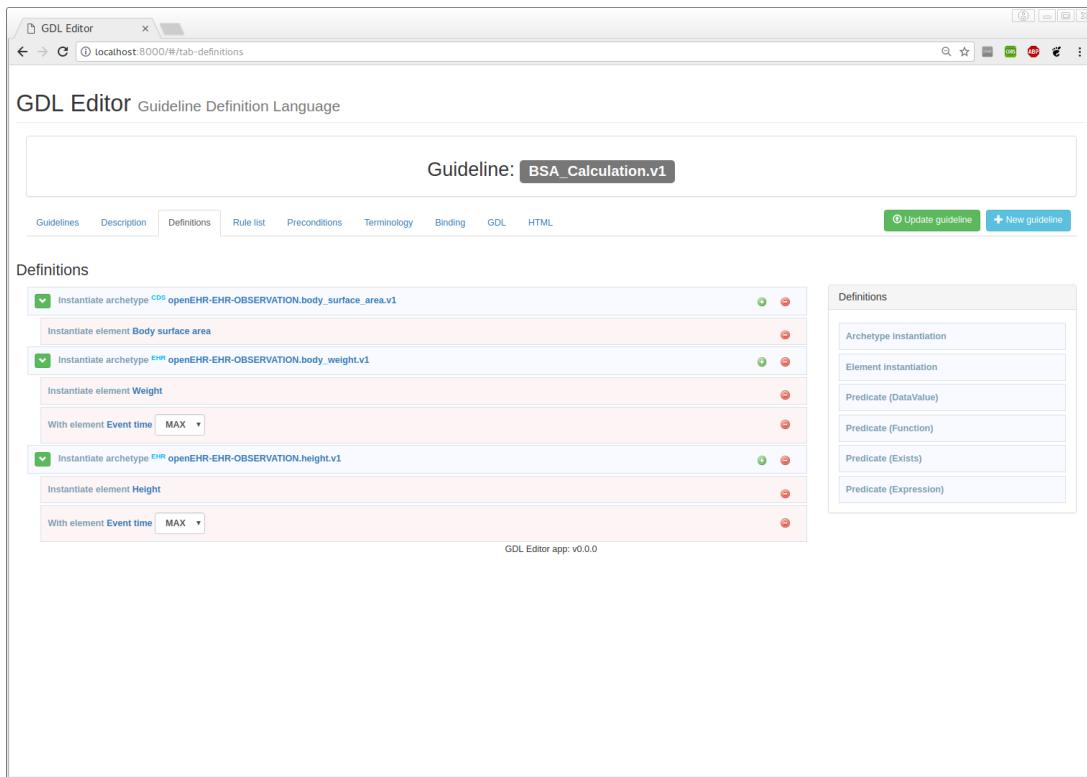


Figura 24: Definitions: definiciones de la guía.

Actualmente, GDL soporta cuatro tipos de definiciones:

- *Archetype instantiation*: crea una referencia a un arquetipo o plantilla. Para cada instanciación tendremos que definir dos parámetros:
 - *Domain*: hay tres posibles valores: *EHR*, *CDS* y *ANY* (Figura 25). Ver la especificación GDL para más información sobre cada una de ellas.
 - *Archetype/Template*: un listado con todos los arquetipos que se mostrarán.

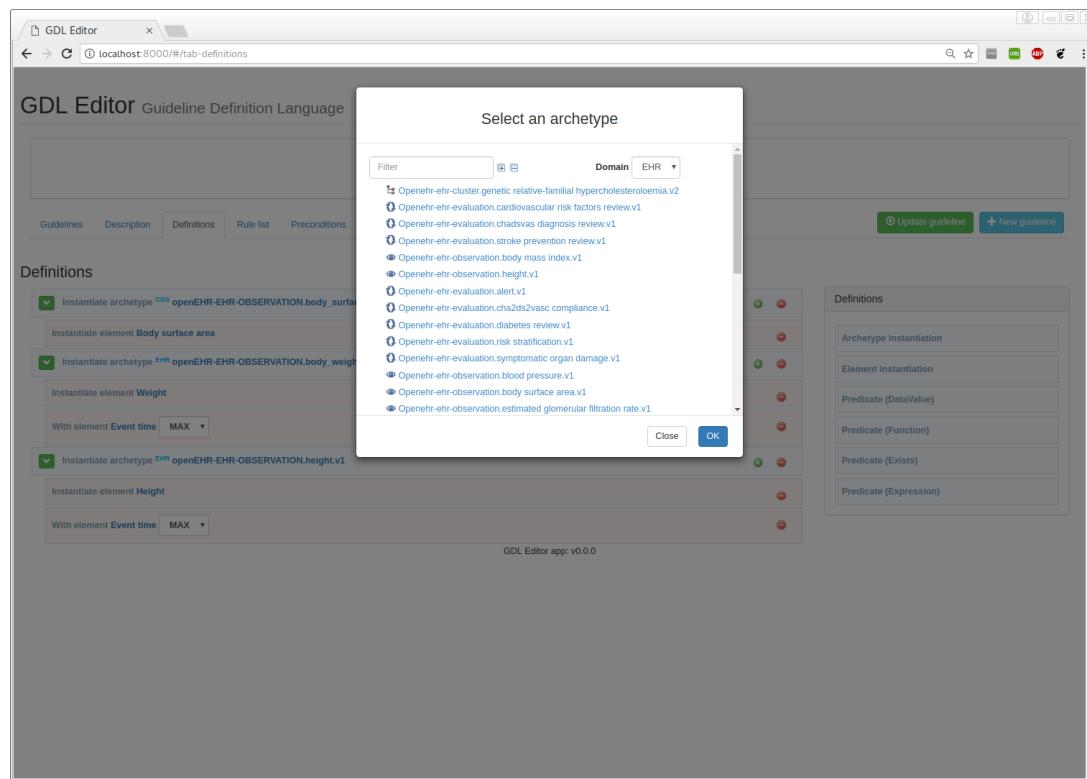


Figura 25: Definitions: elección de arquetipo.

- *Element instantiation:* crea una referencia a un elemento dentro del arquetipo o plantilla. Tiene que ser colocado dentro de una instancia de arquetipo.

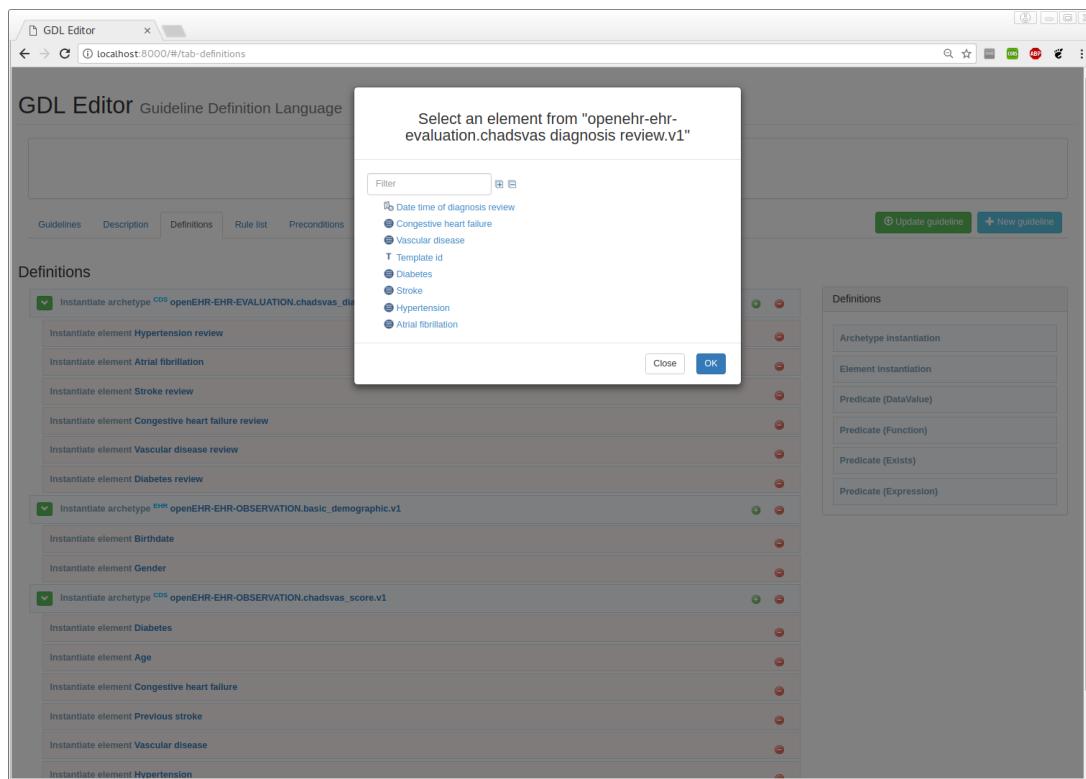


Figura 26: Definitions: elección de una instancia de elemento.

- **Predicate (DataValue):** define una restricción para la instancia del arquetipo. Tiene que ser colocado dentro de una instancia de arquetipo.
- **Predicate (Function):** añade restricciones a los elementos definidos mediante el uso de funciones de agregación.

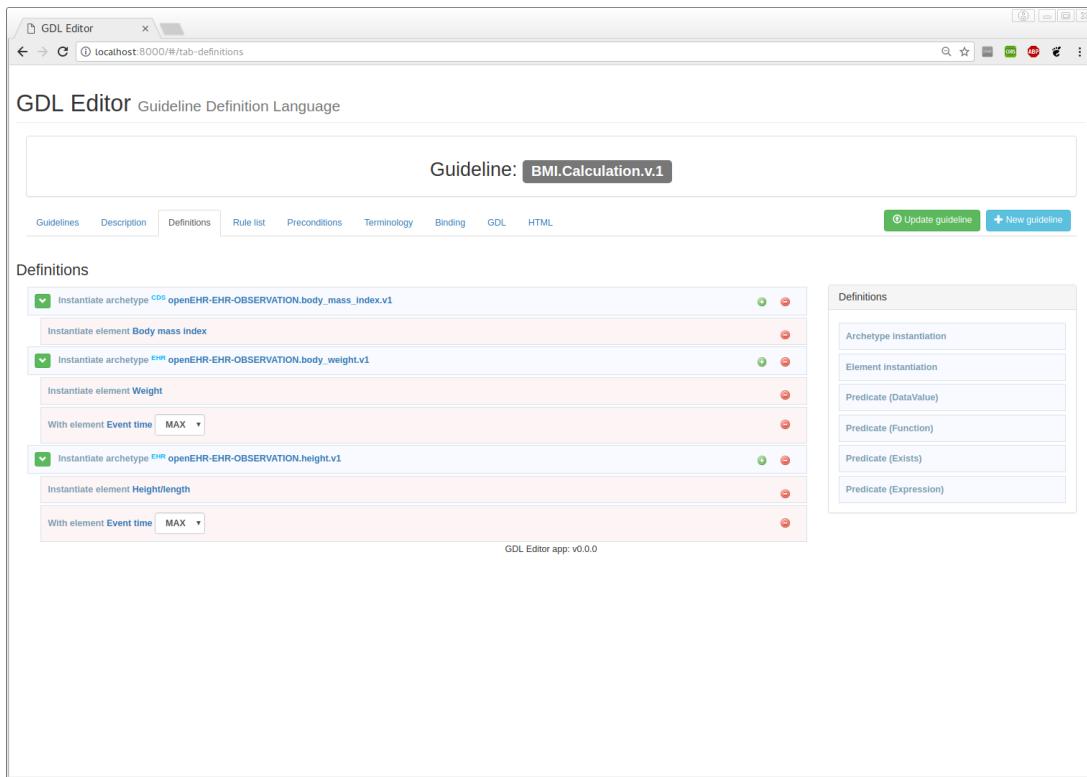


Figura 27: Definitions: Funciones predicado.

A.2.4. Rule List

En esta pestaña podremos administrar todas las reglas de la guía. Cada regla contiene un conjunto de condiciones y acciones (consultar Sección A.2.5). Para acceder a una regla, simplemente haga clic en su nombre.

La gestión de reglas es muy similar a las definiciones. Para agregar una nueva regla, utilice el botón *Add rule* situado sobre la lista de reglas. Para editar el nombre de la regla, utilice el icono de lápiz y para eliminar una regla haga clic sobre el botón rojo situado a la derecha del lápiz.

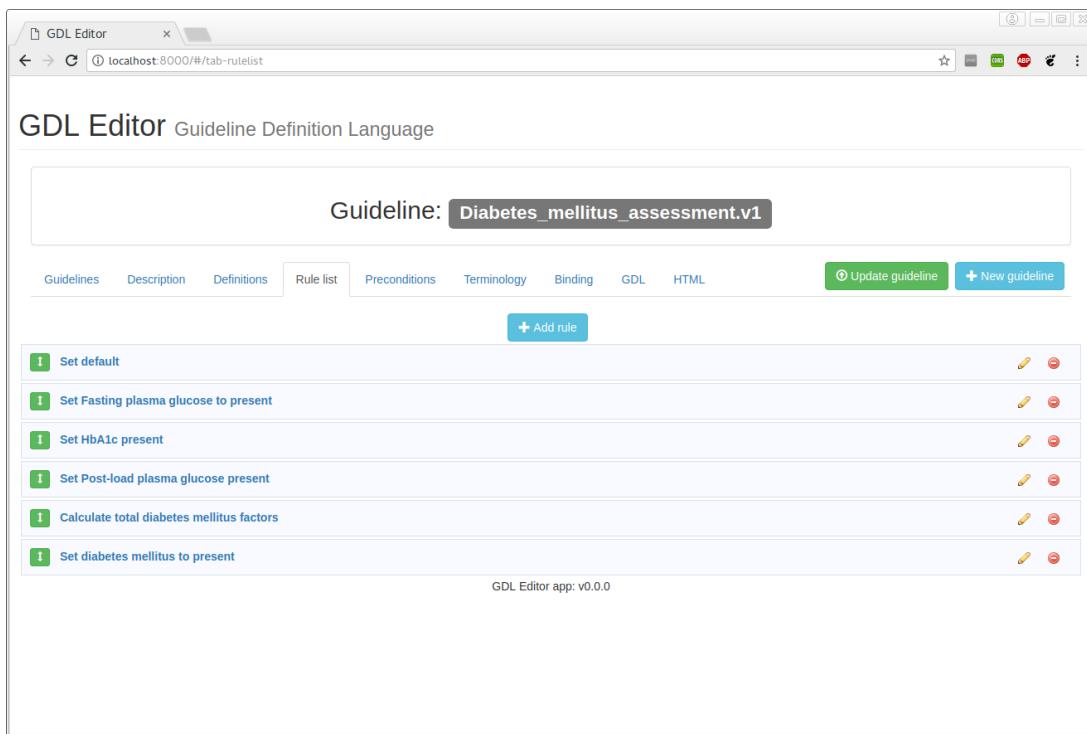


Figura 28: Lista de reglas.

A.2.5. Edición de reglas

Cuando se accede a una regla, se mostrará el editor de reglas. La parte superior muestra las condiciones necesarias para que la regla se ejecute, la parte inferior contiene las acciones que tendrán lugar una vez que se active la regla (ver Figura 30). La mayoría de las acciones y condiciones se referirán a una instancia de elemento que puede definirse previamente en la sección *Definitions* o directamente creada desde el editor de reglas. En el segundo caso, al seleccionar una instancia de elemento desde una condición o una acción, se mostrará un cuadro de diálogo para seleccionar / definir instancias de elemento. Este diálogo nos permitirá seleccionar una instancia de elemento ya definida (Figura 29), una instancia de elemento de una instancia de arquetipo ya definida (2) o añadir una nueva instancia de arquetipo (3).

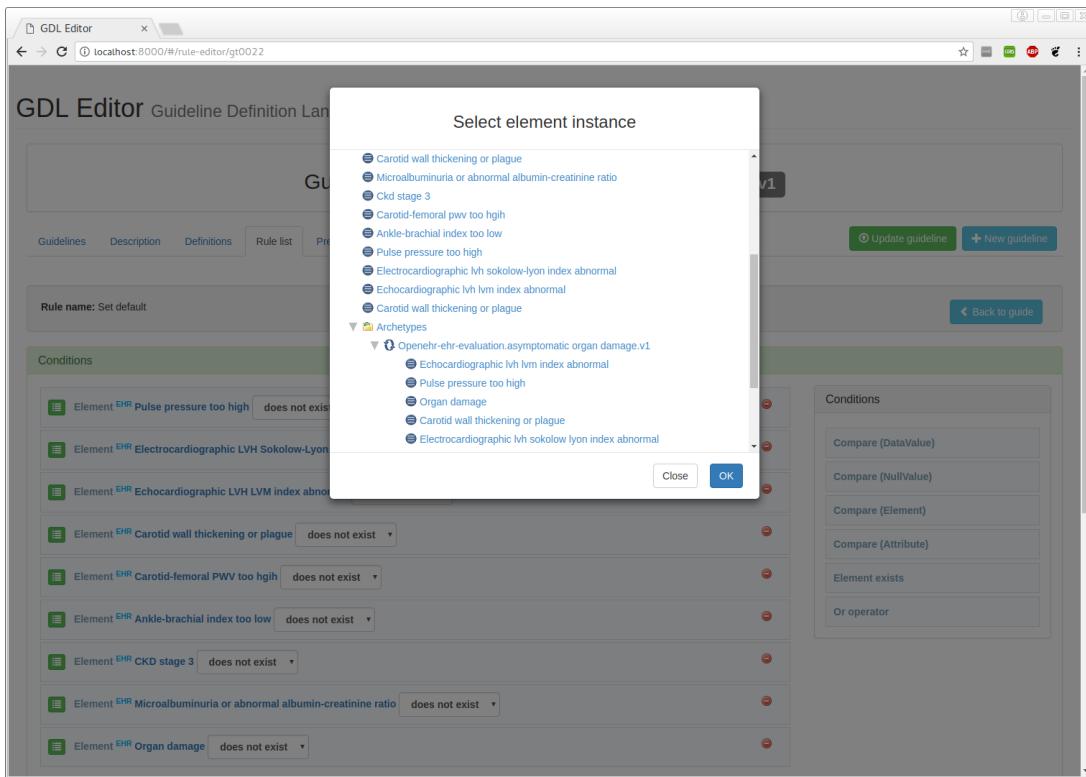


Figura 29: Edición de reglas: Seleccionar instancia de elemento.

La edición de condiciones y acciones es muy similar a la de las definiciones. La versión actual de GDL soporta seis tipos de condiciones:

- *Compare (DataValue)*: compara el valor de una instancia de elemento con un valor de datos (constante).
- *Compare (NullValue)*: compara el valor nulo de una instancia de elemento con un código *openEHR NULL_FLAVOUR*.
- *Compare (Element)*: compara el valor de una instancia de elemento con el valor de otra instancia de elemento.
- *Compare (Attribute)*: compara el atributo de una instancia de elemento con una constante o una expresión (véase [Editor de expresiones](#)).
- El elemento se inicializa: comprueba si la instancia del elemento tiene o no tiene ningún valor asignado.
- *O operator*: realiza una disyunción lógica entre dos condiciones.

Actualmente se soportan 4 tipos de acciones:

- *Set (DataValue)*: inicializa la instancia del elemento con el valor de datos seleccionado.
- *Set (NullValue)*: elimina el valor de la instancia del elemento y establece el código *NLL_FLAVOUR* seleccionado.
- *Set (Element)*: copia el valor de una instancia de elemento a otra.
- *Set (Attribute)*: establece el valor de un atributo utilizando una constante o una expresión (consulte [Editor de expresiones](#)).

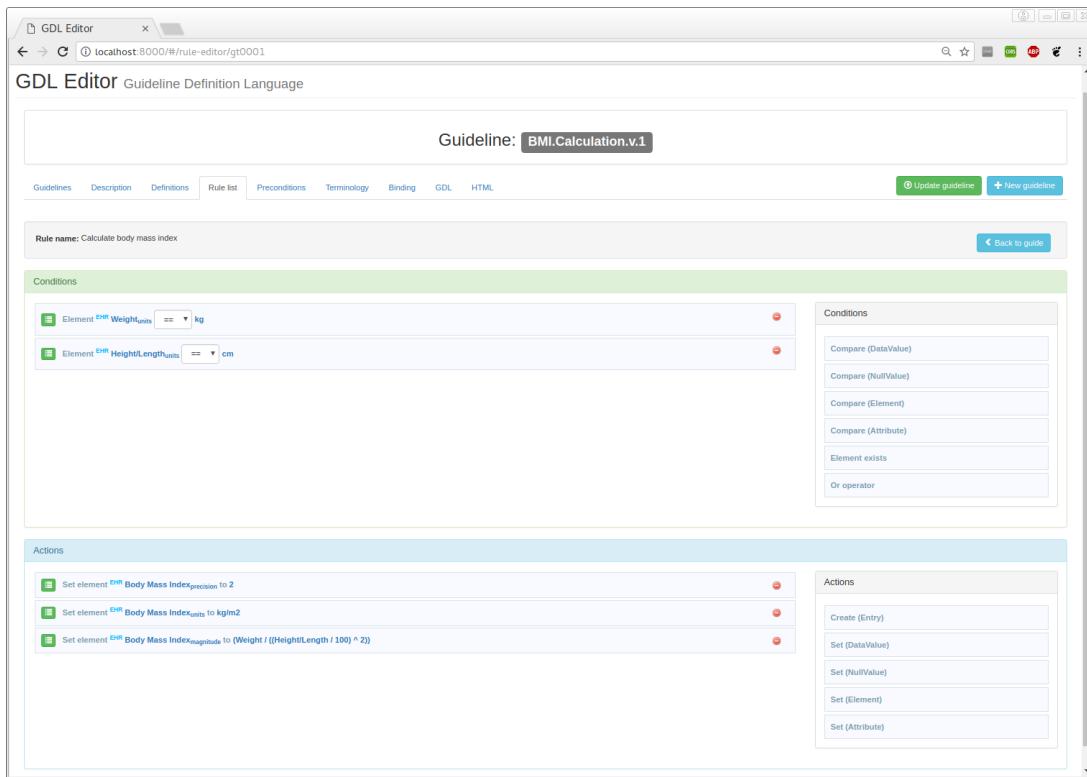


Figura 30: Editor de reglas.

Es importante tener en cuenta que sólo podremos realizar acciones en las instancias de elementos que correspondan a una instancia de arquetipo en el dominio CDS. Esto significa que el motor de reglas no puede realizar cambios directamente en los elementos de EHR.

A.2.6. Editor de expresiones

blah

A.3. Ejemplo de creación de una guía clínica

blah