Proyecto de Grado

Ingeniería en Computación

Generación Automática de Interfaces de Usuario para Software Médico

Laura Cuadrado
lauraccibic@gmail.com

Arianne Palau
luarinblue@gmail.com

Tutor

Ing. Antonio López Arredondo

Cliente

Ing. Pablo Pazos Gutiérrez

Facultad de Ingeniería - Universidad de la República
Instituto de Computación
Montevideo - Uruguay
18 de diciembre de 2013

Resumen

Los sistemas de información en salud sufren muchas modificaciones en cortos períodos de tiempo. La mayoría de estos cambios son demandados sobre la estructura de los registros clínicos (por ejemplo, agregar un campo o modificar una restricción). La rigidez de la gran mayoría de los sistemas existentes hace que los tiempos y costos asociados a su modificación sean altos y las herramientas se atrasen con respecto a los requerimientos actuales. Este proyecto absorbe el impacto de los cambios requeridos de forma automática o semi-automática en la capa de interfaz de usuario.

El foco de la solución planteada en este trabajo se centra en la especificación y generación de interfaces de usuario para el registro clínico, basándose en los modelos de información y de gestión del conocimiento. El primero se implementa dentro de las aplicaciones de forma genérica y estable, mientras que el otro representa la parte dinámica del registro clínico permitiendo crear especificaciones sobre el modelo de información genérico.

Durante el proyecto se realizó un estudio sobre los conceptos relacionados con la generación de código y las pautas para el diseño de interfaces. Dado que uno de los principales requerimientos del proyecto fue incluir la estandarización de definiciones de conceptos clínicos (arquetipos) establecida por OpenEHR, se hizo un relevamiento de su modelo de información y modelo refinado.

Para el desarrollo del diseñador y generador de interfaces de usuario se trabajó con el lenguaje de programación Groovy, logrando generar IU en HTML5 para aplicaciones web y en XAML para aplicaciones de escritorio.

Estas herramientas permitieron validar el proceso de generación automática de interfaces. El mismo estuvo ordenado por los cinco pasos básicos para construir un generador por nivel, lo que implicó especificar los metadatos, crear un modelo conceptual y semántico de definiciones de interfaces y especificar las correspondencias o mapeos entre los elementos distinguibles de la definición y los componentes de los artefactos finales en las diferentes tecnologías.

El resultado obtenido representan un paso más en una posible evolución de los sistemas clínicos, dado que incorpora estándares y acelera los tiempos asociados a la prototipación y generación de interfaces de usuario para software clínico.

Palabras claves

Sistemas de información en salud, generación de código, OpenEHR, arquetipos, interfaz de usuario.

Tabla de Contenido

Resumen 1
Tabla de Contenido 3
1 Introducción 7
1.1 Motivación 7
1.2 Objetivos 8
1.3 Organización del documento 9
2 Estado del arte 10
2.1 Generación de código 10
2.1.1 Conceptos básicos 10
2.1.2 Herramientas analizadas 12
2.1.2.1 GeneXus 12
2.1.2.2 CodeSmith 12
2.1.2.3 Iron Speed 13
2.1.2.4 AJGenesis 13
2.1.2.5 Evaluación general de herramientas 13
2.1.3 Generación de interfaces de usuario 14
2.2 Diseño de interfaz de usuario 14
2.2.1 Modelos mentales 15
2.2.2 Componentes de un formulario 16
2.2.3 Percepción visual 16
2.2.4 Principios básicos del diseño de IU 17
2.3 Tecnologías 18
2.3.1 XML 18
2.3.2 XML Schema 19
2.3.3 Groovy 20
2.3.4 HTML5 20
2.3.5 XAML 21
3 OpenEHR 22
3.1 Modelo de información 22
3.1.1 Tipos de dato 24
3.2 Modelo de Conocimiento 25
3.2.1 Arquetipos 25
3.2.1.1 Lenguaje de definición de arquetipos 26
3.2.2 Template 26
3.3 Herramienta de edición y visualización 26
3.3.1 Archetype Editor 27

3.3.2 ADL Workbench 27
3.3.3 ADL - syntax highlight for Notepad++ 27
3.3.4 Template Designer 27
3.4 EHRGen 28
3.4.1 Metodología de trabajo 29
3.4.2 Template 29
3.4.2.1 Limitaciones del modelo 30
4 Organización del trabajo 32
4.1 Alcance 32
4.2 Metodología de trabajo 32
4.3 Gestión del proyecto 33
5 Presentación de la solución 35
5.1 Generador de interfaces de usuario 35
5.1.1 Diseño del generador 36
5.1.1.1 Especificación de requerimientos sobre los metadatos 36
5.1.1.2 Diseño del formato de serialización de instancias del modelo 37
5.1.2 Construcción del código de prueba 42
5.1.3 Desarrollo del analizador de datos de entrada 43
5.1.4 Desarrollo del mapeador a partir del código de prueba 44
5.1.4.1 Definiciones de mapeo 45
5.1.5 Desarrollo del generador del código de salida 48
5.1.5.1 Componentes del generador 49
5.2 Evolucionando el generador de interfaces de usuario 52
5.2.1 Multiplicidad 52
5.2.2 Generación de IU para una nueva tecnología 53
5.3 Diseñador de interfaces de usuario 54
6 Verificación y validación 55
6.1 Verificación 55
6.2 Validación 55
6.2.1 Diseñador de interfaces 55
6.2.2 Generador de interfaces de usuario 56
6.2.3 Aplicación integradora 58
7 Planificación, Esfuerzo efectivo 62
8 Conclusiones 63
8.1 Conocimientos adquiridos 64
8.2 Trabajo futuro 64
Glosario 66
Referencias 68

Índice de Ilustraciones 71
Índice de Gráficos 71
Índice de Tablas 72
Anexos 73
Anexo A. ADL Archetype Definition Language: Caso de estudio 73
Anexo B. Template: Descripción de componentes 75
Anexo C. Tipos de controles 76
Anexo D. UITemplate: XML Schema 79
Anexo E. Variables de sustitución 85
Variables Globales 89
Anexo F. Manual de usuario técnico 90
Agregar nuevos tipos de datos 91
Agregar nuevo tipo de control par un tipo de dato existente 91
Agregar una nueva tecnología para generar interfaces de usuario 92
Agregar un nuevo tipo de lectura 92
Multiplicidad en HTML5 93
Multiplicidad a nivel de ArchetypeFieldReference 93
Anexo G. Manual de Usuario Funcional 93
Requerimientos del sistema 94
Estructura del entregable 94
Compilación 95
Configuración 96
Archivo de configuración del generador 96
Archivos de mapeo 96
Ejecución 97
DISM 97
GISM 97
Ejemplo 98
Errores Conocidos 102
DISM 102
GISM 102
Anexo H. Entrevista a Gastón Milano 102
Anexo I. Entrevista Juan José Spinetti 107

1 Introducción

La informática médica es la aplicación de la informática y las comunicaciones al área de la salud, mediante el uso del software médico. Su objetivo principal es prestar servicio a los profesionales de la salud para mejorar la calidad de la atención sanitaria (1). Entre los grandes aportes que la informática médica ofrece a la comunidad encontramos la oportunidad y calidad de la información, la efectividad y la eficiencia. Por su parte, al personal médico y le da la posibilidad de manejar grandes cantidades de información en forma ágil y organizada, ayudando así a mejorar la calidad de sus diagnósticos. En consecuencia, el paciente ve mejorada la calidad de los servicios que les son prestados, lo cual le brinda mayor información, control y seguridad sobre los temas relacionados con su salud.

En contraparte, la complejidad de dichos sistemas no es menor. Los mismos deben manejar grandes cantidades de información, lo que complica su mantenimiento a largo plazo, y resultan muy costosos a la hora de su construcción.

La falta de aplicación de estándares y buenas prácticas en su diseño y construcción repercute negativamente en la calidad global del producto final. Esta situación pone en riesgo la viabilidad de los proyectos de informatización del área clínica, a gran escala y largo plazo. (2)

A raíz de este problema, es que surgen comunidades de profesionales que buscan lograr una mejor atención sanitaria mediante la aplicación de tecnologías de la información y comunicación (TIC), centrándose en la aplicación de estándares y buenas prácticas. Este es el caso de OpenEHR, una fundación sin fines de lucro, creada para permitir el desarrollo de especificaciones, software y recursos de conocimiento para los sistemas de información de la salud.

Creemos en la definición, adopción y adaptación de estándares libres, porque es la única forma de lograr una estandarización a gran escala, disminuir costos, y aprovechar los pocos recursos disponibles al máximo. (3)

En este proyecto queremos lograr hacer un aporte a la comunidad de la mano de OpenEHR, buscando una manera de sobrellevar parte los costos y de los tiempos invertidos en el mantenimiento de sistemas médicos.

1.1 Motivación

Los sistemas de información en salud sufren muchas modificaciones en cortos períodos de tiempo. La mayoría de estos cambios son demandados sobre la estructura de los registros clínicos (por ejemplo agregar un campo o modificar una restricción). La rigidez de la gran mayoría de los sistemas existentes hace que los tiempos y costos asociados a su modificación sean altos y las herramientas se atrasen con respecto a los requerimientos actuales.

Este proyecto busca acortar los tiempos de prototipación, construcción, modificación y validación del software, absorbiendo el impacto de los cambios requeridos de forma automática o semi-automática en la capa de interfaz de usuario.

1.2 Objetivos

En este proyecto se aplicarán experiencias previas sobre generación de interfaz de usuario a partir de metadatos, resolviendo problemas pendientes y formalizando una metodología de aplicación general. Ésta se basará en un paradigma de diseño multi-nivel definido por distintos aspectos de los sistemas de información mediante artefactos computables representados mediante formatos estándar (como XML, XSD, ADL, JSON), donde cada nivel depende solo del anterior tal como se puede ver en la *Ilustración 1.1*. En esta imagen se distinguen el modelo de información en donde se definen las entidades de dominio orientadas a la documentación clínica, el modelo refinado el cual es utilizado para crear modelos detallados de los distintos componentes del registro clínico (en forma de especificaciones y restricciones), el modelo de interfaz de usuario a partir del cual se crean las interfaces de usuario que permitan ingresar, corregir, editar y visualizar datos del registro clínico de un paciente, teniendo en cuenta que todos los datos cumplen los modelos detallados previamente, y finalmente el modelo de workflow, lugar donde se definen distintas formas de combinación de registros clínicos para distintos usuarios o roles, integrando las diferentes interfaces de usuario.

Los cambios de requerimientos del sistema se verán reflejados a nivel del modelo refinado y visualizados en la interfaz de usuario. Por otro lado el modelo de workflow permitirá contextualizar el uso de las interfaces de usuario, como ser el orden, las condiciones, los modos de visualización, etc.

Ilustración 1.1: Diagrama multinivel

El trabajo se acotará al dominio de la salud. El modelo de información a utilizar será el de OpenEHR (4) y el refinado será el de arquetipos (5). El modelo de interfaz de usuario será uno de los resultados del proyecto. Se deberán definir formatos computables para el modelo de interfaz de usuario y se creará un Diseñador de Interfaces de Usuario capaz de tomar como insumo los modelos de información y restricciones, y generar los formatos antes mencionados.

A partir de esto, se creará una segunda herramienta para generación de artefactos de interfaz de usuario sobre tecnologías específicas, basándose en los formatos computables. Por ejemplo el generador podría crear HTML para interfaces web de PCs de escritorio, HTML para interfaces web de Smart Phones, XAML para aplicaciones .Net, XUL para aplicaciones de escritorio (especificación de Mozilla), SwiXML para aplicaciones de escritorio Java/Swing.

Como prueba de concepto se creará un prototipo minimal de aplicación concreta que utilice los artefactos de interfaz de usuario creados por el generador.

1.3 Organización del documento

En el próximo capítulo se presentará el estudio del estado del arte que se realizó en las primeras etapas del proyecto, el cual incluirá un análisis sobre los conceptos relacionados con la generación de código y pautas para el diseño de interfaces, así como también, una sección dedicada a la evaluación de tecnologías relevantes para el proyecto.

El tercer capítulo estará dedicado al estudio de OpenEHR, el cual proporciona la base de información y estándares sobre las que se basa el proyecto. En dicho capítulo se describirá a grandes rasgos los fundamentos de OpenEHR y se presentarán aplicaciones relacionadas al mismo.

En el cuarto capítulo se realizará una presentación formal del proyecto en cuanto alcance, metodología de trabajo y gestión. El quinto capítulo describirá de forma detallada el proceso de desarrollo completo de la solución lograda, mientras que en el sexto capítulo, se presentarán las etapas de verificación y validación realizadas sobre la misma.

Finalmente, el séptimo capítulo incluirá las conclusiones alcanzadas al cierre del proyecto, así como también, la visión a futuro del mismo.

La completitud del documento está apoyada en diez anexos que se adjuntan al final del mismo, con el fin de que dicho conocimiento esté al alcance de los lectores, de forma tal de poder ampliar cualquiera de los capítulos que crea necesario.

2 Estado del arte

El estudio del estado del arte se dividió en dos ramas que hacen al desarrollo final del proyecto. Por un lado, se relevaron los conceptos más importantes acerca de la generación de código a partir de descripciones de alto nivel, con el fin de hallar pautas de desarrollo a tener en cuenta para crear dicho generador. Por otro lado, se investigó acerca de las pautas y recomendaciones a seguir a la hora de diseñar las interfaces que construirá el generador. Finalmente, para acompañar las líneas de estudio planteadas, se realizó un relevamiento de tecnologías de uso relevante para el proyecto.

2.1 Generación de código

En el libro "Code Generation in Action" (6), Jack Herrington afirma que las dos constantes principales en la ingeniería de software son: que el tiempo del programador es valioso y que al programador no le gustan las tareas repetitivas y aburridas.

La generación de código busca alivianar la tarea de escribir código repetitivo, dejando más tiempo disponible para aquellas tareas críticas que merecen la atención del programador. Alguna de las ventajas que podemos destacar son:

Portabilidad

Las reglas de negocio se abstraen en archivos que sean independientes del lenguaje o detalles del framework, con el fin de no obstaculizar la portabilidad.

Calidad

Grandes cantidades de código escrito a mano tienden a tener calidad inconsciente, debido a que los programadores encuentran nuevos y mejores enfoques mientras avanzan en el desarrollo. La generación de código crea instantáneamente una base de código consistente, y si la configuración del generador es cambiada, solo basta con correr nuevamente el generador para que las correcciones de defectos o las mejoras en el código sean aplicadas consistentemente sobre todo el código base, en cuestión de segundos.

Conocimiento centralizado

Un cambio en el archivo de definición del esquema se propagará en cascada a través del generador en todo el sistema.

Por todo esto, podemos decir que la generación de código es una herramienta muy valiosa que puede tener un gran impacto en la productividad y la calidad de los proyectos de ingeniería de software.

Citando nuevamente a Jack Herrington: "*Cuando usted puede trabajar más inteligentemente en vez de más duro y utilizar la computadora para alivianar parte de su trabajo, su proyecto ira mejor*".

2.1.1 Conceptos básicos

Un generador de código es una herramienta que toma como entrada metadatos, combina los mismos con un motor de template, y produce una serie de archivos de código fuente como salida.

En términos generales, hay dos tipos diferentes de generadores de código: pasivos y activos. Los generadores de código pasivos generan código una vez y luego se desligan de toda responsabilidad sobre el código generado. Son buenos para crear código base que luego el desarrollador utilizará para personalizar. Pero una vez que se ha creado el código, el generador de código pasivo no puede regenerarlo con los cambios realizados posteriormente por el programador.

Por el contrario, los generadores de código activos están diseñados para mantener un vínculo con el código que se genera en el largo plazo, permitiendo que el generador se pueda ejecutar varias veces sobre el mismo código. Para que un generador de código activo sea eficaz, debe proporcionar una manera para que el desarrollador pueda personalizar el código de salida, y luego permitir la regeneración del código sin sobrescribir las personalizaciones.

Modelo por nivel

Un generador de código que siga el modelo por nivel, genera el código para todo un nivel o sección de una aplicación. Los datos de entrada de un generador de este tipo provienen de un archivo de definición desde donde reúne la información necesaria para construir las clases y funciones correspondientes al nivel. Luego usa esa información en conjunto con templates para construir el código de salida. Estos archivos de salida ya estarán listos para consumir y a la espera de ser integrados con el resto de la aplicación.

Para construir un generador de nivel, se deben seguir los siguientes pasos:

1) *Construcción del código de prueba*

Un aspecto interesante sobre la construcción de un generador de código, es que se empieza por el final. El primer paso es crear un prototipo de lo que se desea generar, escribiendo a mano el código deseado. Esto se llamará código de prueba.

2) *Diseño del generador*

Una vez que se tiene definida la salida, es necesario determinar cómo se va a construir el generador que va a crear el código de prueba como salida. El trabajo más importante es especificar los requerimientos sobre los datos. ¿Cuál es el menor conjunto posible de conocimientos que se necesitan para construir la salida? Una vez que se tengan los requisitos sobre los datos, se deberá diseñar el formato de archivo de definición de entrada.

3) *Desarrollo del analizador de datos de entrada*

El primer paso de la implementación es construir la definición del analizador del archivo de entrada. En el caso de XML, esto significa leer del XML y almacenarlo en alguna estructura interna para poder ser procesado más fácilmente.

4) *Desarrollo de los templates a partir del código de prueba*

Un vez que los datos de entrada estén dentro del generador, se deben crear los templates que generarán los archivos de salida. La manera más fácil de hacer esto es tomando el código de prueba.

5) *Desarrollo del constructor de código de salida*

El paso final es crear el código ensamblador, responsable de correr los templates sobre la especificación de entada desde el archivo de definición y construir los archivos de salida. Al final del desarrollo, debería ser posible correr el generador sobre los

archivos de definición y obtener una salida que se corresponda con el código de prueba creado en el comienzo del proceso.

2.1.2 Herramientas analizadas

Dado que hoy en día existe varias herramientas útiles para realizar generación automática de código, durante este capítulo se realiza una breve reseñan sobre aquellas que fueron estudiadas.

2.1.2.1 GeneXus

Desarrollada por Artech, GeneXus es una herramienta de desarrollo de software ágil, multiplataforma, basada en conocimiento, orientada principalmente a aplicaciones web empresariales, plataformas Windows y dispositivos móviles o inteligentes.

La idea básica de GeneXus es automatizar todo aquello que es automatizable: normalización de los datos y diseño, generación y mantenimiento de la base de datos y de los programas de aplicación. De esta manera se evita que el analista deba dedicarse a tareas rutinarias y tediosas, permitiéndole poner toda su atención en aquello que nunca un programa podrá hacer: entender los problemas del usuario.

Partiendo del conocimiento sistematizado, GeneXus genera automáticamente la aplicación (base de datos y programas) para la plataforma que se haya escogido. Desde un punto de vista lógico, lo que debe hacerse es independiente de la plataforma. Físicamente, sin embargo, no lo es: cada lenguaje, cada sistema de gerencia de base de datos, cada sistema operativo, cada arquitectura, tiene comportamientos diferentes. GeneXus resuelve este problema dividiendo la generación en dos: lógica, común a todas las plataformas, y física, construida de modo de optimizar los programas generados para cada una de las plataformas.

GeneXus genera código para múltiples lenguajes, incluyendo: Cobol, RPG, Visual Basic, Visual FoxPro, Ruby, C#, Java, y para múltiples plataformas móviles, incluyendo Android o Blackberry, y Objective-C para dispositivos Apple. Los DBMSs más populares son soportados, como Microsoft SQL Server, Oracle, IBM DB2, Informix, PostgreSQL y MySQL.

GeneXus y la herramienta a desarrollar para este proyecto de grado tiene grandes puntos en común. El principal, es en cuanto a la generación de código para múltiples tecnologías en base a una única definición de la realidad a representar.

Debido a que GeneXus es un producto con una gran trayectoria, se realizó una entrevista a Gastón Milano, gerente de desarrollo, para indagar sobre cuál es el secreto detrás de GeneXus. La entrevista se encuentra transcripta en el *Anexo H.*

2.1.2.2 CodeSmith

Otra herramienta similar y con buena reputación es CodeSmith (7). El generador CodeSmith es una herramienta de desarrollo que ayuda a realizar el trabajo de los desarrolladores de manera más eficiente.

Técnicamente hablando, se trata de un generador de código basado en templates que automatiza la creación de código común de aplicaciones para cualquier lenguaje (C#, Java, Visual Basic, PHP, ASP.NET, SQL, etc.). La herramienta incluye muchas plantillas útiles, así como sets de templates para la generación de arquitecturas que ya han sido probadas. El desarrollador puede fácilmente modificar los templates o escribir el suyo propio para generar el código de la manera deseada.

2.1.2.3 Iron Speed

Iron Speed Designer (8) acelera el proceso de desarrollo mediante la creación de aplicaciones fácilmente personalizables y listas para poner en producción. Las aplicaciones generadas tienen un look-and-feel consistente, y es posible crear pantallas complejas con componentes que tengan conexión a base de datos, incluyendo formularios, reportes, filtros, y navegación.

Con esta herramienta podremos crear rápidamente una aplicación web en .NET que cumple con las siguientes características:

- Comienza desde cero con el modelo de datos específicos de la aplicación.
- Crea todo lo necesario para proveernos de una aplicación ejecutable libre de errores.
- Crea automáticamente código fuente nativo para la plataforma destino
- Proporciona código fácil de modificar.
- Conserva modificaciones/personalizaciones durante la regeneración de aplicaciones posteriores.

2.1.2.4 AJGenesis

Por último, se evaluará una herramienta de código abierto llamada AJGenesis (9). Ésta es un generador de código y archivos de texto basado en modelos y plantillas definibles.

Los principales puntos a destacar son:

- El uso de un modelo libre en memoria, permitiendo el manejo y el acceso como si fuera un objeto común en un lenguaje de programación.
- El uso de un lenguaje simple y dinámico para ejecutar tareas simples o complejas, dando lugar a una completa librería de clases.
- El uso de plantillas con soporte de expresiones y comandos escritos usando el mismo lenguaje dinámico.

2.1.2.5 Evaluación general de herramientas

Lo primero que vamos a evaluar es el costo de utilizar las herramientas. Las tres evaluadas en primera instancia son pagas, oscilando su costo entre los dos mil dólares anuales. Este hecho no satisface la necesidad del proyecto de proporcionar un sistema de código abierto.

Por tanto nos deja como único candidato a AJGenesis. Si bien dicha herramienta es de código abierto y muy personalizable, AJGenesis tiene una documentación muy pobre y la última versión liberada es del año 2006. La herramienta

fue desarrollada por una sola persona y no da garantía de tener soporte y actualizaciones frecuentes.

2.1.3 Generación de interfaces de usuario

La mayor ventaja de generar interfaces de usuario es que se puede crear una capa de abstracción entre las reglas de negocio de la interfaz y su implementación. El código asociado a interfaces de usuario puede ser particularmente complicado, en el cual se pueden encontrar parte de las reglas de negocio y los estándares de usabilidad de la aplicación.

A continuación, presentaremos algunas de las principales ventajas aplicadas a la generación de interfaces de usuario:

Consistencia

La generación de pantallas en masa, crea pantallas que son consistentes y que presentan estándares de usabilidad uniformes para el usuario.

Flexibilidad

Usar una herramienta para construir interfaces significa que es posible responder rápidamente a los cambios de requerimientos.

Portabilidad

Tener una representación de alto nivel de los requerimientos de negocio asociados a las interfaces de la aplicación, permite reconfigurar el generador a cualquier número de tecnologías diferentes para generar dichas interfaces.

Una interfaz de usuario puede ser pensada como dos piezas. La primera consiste en el aspecto visual de la interfaz. El código generado puede verse tan bien como el de interfaces codificadas a mano, dependiendo siempre de la calidad del templates utilizado para generarlas. La otra pieza es la facilidad de uso de la interfaz. Se necesita un gran cuidado para construir interfaces de usuario “usables” generadas con la técnica de generación de código. La generación automática no hará que la interfaz de usuario sea mejor. Su tarea consiste en crear la interfaz de forma más rápida y más consistente que si se utilizara la codificación manual.

En el siguiente capítulo se presentarán una serie de características de diseño que buscan lograr una interfaz de usuario de calidad en términos de usabilidad y visualización que sele proporcionará al usuario.

2.2 Diseño de interfaz de usuario

El estudio del diseño de interfaces de usuario es un subconjunto del campo de estudio llamado interacción persona-computadora (HCI por sus siglas en ingles) (10). La interacción persona-computadora comprende el estudio, la planificación y el diseño de cómo las personas y las computadoras trabajan juntas para que las necesidades de las primeras se satisfagan de la manera más eficaz. Para ello, los diseñadores HCI deben tener en cuenta una variedad de factores: lo que la gente desea y espera, las limitaciones físicas y las capacidades que las personas poseen, cómo trabaja su

percepción y su sistema de procesamiento de información, y lo que la gente encuentra agradable y atractivo.

La interfaz de usuario tiene esencialmente dos componentes: la entrada y la salida. La entrada es cómo una persona comunica a una computadora sus necesidades o deseos. La salida es cómo la computadora transmite el resultado de sus cálculos y requisitos al usuario.

El diseño adecuado de una IU proporcionará una combinación de mecanismos de entrada y salida que satisfará las necesidades del usuario, sus capacidades y limitaciones de la forma más eficaz posible. La mejor interfaz es una que no es percibida y que permite al usuario centrarse en la información y en la tarea, en lugar del mecanismo utilizado para presentar la información y llevar a cabo la tarea.

¿Por qué se debería poner tanto empeño en diseñar una interfaz de usuario?

En el ambiente de la salud, los profesionales ya tienen un método eficaz para registrar su trabajo. Ellos ya cuentan con planillas en papel, que rellenan a mano con gran fluidez. El profesional siente que ya tienen automatizado y dominado este método, y por ello es muy difícil competir con él.

Para que un médico prefiera usar un método nuevo para hacer su trabajo, como lo es el de la historia clínica electrónica, debemos proveerle un ambiente donde se sienta cómodo para interactuar, que esté familiarizado con los elementos con que trabajará y a su vez, que sienta que el sistema le proporciona un gran beneficio a cambio de poca dedicación. Necesitamos que dicho usuario se sienta atraído por el cambio. En el *Anexo I* se presenta una entrevista realizada al Ing. Juan José Spinelli, Jefe de desarrollo de K2BHealth, una solución integral de salud que abarca todas las áreas de la empresa tanto médica como comercial. En dicha entrevista le consultamos su opinión en relación a cómo diseñar IU para registros clínicos.

¿Cómo generar interfaces de usuario intuitivas, atractivas y usables?

Ser fiel a los modelos mentales de los usuarios, seguir los principios básicos para lograr pantallas visualmente perceptivas, comprender los elementos básicos que tendremos que manejar en nuestras pantallas y seguir recomendaciones generales para el desarrollo de IU, son las cosas que todo diseñador debe tener en mente antes de comenzar su trabajo. Las pantallas médicas están compuestas mayoritariamente por formularios, y por lo tanto, enfocaremos la investigación a pantallas de este tipo.

2.2.1 Modelos mentales

Un modelo mental representa el proceso de pensamiento de una persona para saber cómo funcionan las cosas (por ejemplo, la comprensión de una persona del mundo que lo rodea). Los modelos mentales se basan en hechos incompletos, experiencias pasadas, e incluso percepciones intuitivas. Ellos ayudan a darles forma a las acciones y comportamientos, y definen cómo las personas enfocan y resuelven problemas. Los usuarios hacen uso de los modelos mentales para predecir lo que el sistema, el software o el producto va ha hacer o debería hacer con.

Por otra parte, un modelo conceptual es el modelo real que se le da al usuario a través de la interfaz del producto. Alguien diseña una interfaz de usuario y esa interfaz comunica al usuario el modelo conceptual del producto.

Cuando buscamos que el usuario tenga una mejor experiencia con el producto, lo que en realidad estamos buscando es mejorar la concordancia entre los modelos mentales de los usuarios y el modelo conceptual del producto. Si el modelo conceptual del producto no coincide con el modelo mental del usuario, el usuario encontrará el producto difícil de aprender y usar (11).

2.2.2 Componentes de un formulario

En nuestra vida cotidiana hacemos uso frecuente de los formularios. Ellos son necesarios para procesar/ingresar información de forma más efectiva, pero a menudo también son una fuente de dolor de cabeza tanto para los diseñadores, como para los usuarios. Con el tiempo, los usuarios han creado expectativas de cómo un formulario debe verse y comportarse. Por lo general se espera encontrar los siguientes seis componentes: (12)

Etiquetas
Indican a los usuarios qué significan los campos de entrada correspondientes.

Campos de entrada
Los campos de entrada permiten a los usuarios ingresar información. Estos pueden ser campos de texto, campos cifrados para contraseñas, check boxes, radio buttons, etc.

Acciones
Se trata de enlaces o botones que, al ser presionado por el usuario, realizan una acción. Por ejemplo enviar los datos ingresados en un formulario.

Ayuda
Esto proporciona asistencia sobre cómo llenar el formulario.

Mensajes
Los mensajes proporcionan información al usuario en base a sus acciones. Pueden ser positivos (indicando que el formulario se ha enviado con éxito) o negativos (indicando, por ejemplo, que un valor no ingresado es obligatorio).

Validación
Estas medidas garantizan que los datos ingresados por el usuario cumplan los requerimientos definidos para el formulario.

2.2.3 Percepción visual

Gestalt es un término de la psicología, que significa "todo unificado" y se refiere a las teorías de la percepción visual desarrolladas por psicólogos alemanes en la década de 1920. Los principios de la Gestalt más relevantes para el diseño de los formularios son: el color, tamaño, forma, figura / fondo, proximidad y similitud (13).

Color, tamaño y forma son atributos de los elementos individuales de la pantalla. Usaremos estos atributos para destacar (por ejemplo, añadiendo un asterisco rojo para señalar qué preguntas son obligatorias), comunicar (el tamaño de un cuadro de texto informa a los usuarios en cuanto a la cantidad esperada de contenido) y maximizar la capacidad de aprendizaje (por ejemplo, círculos para radio buttons y cuadrados para check boxes).

Los principios de figura / fondo, proximidad y similitud, se refieren a la relación entre los elementos. La figura / fondo nos dice que la gente naturalmente busca figuras en su entorno. Para los formularios, esto significa que las personas esperan detectar elementos como botones y cuadros de texto, por lo que estos elementos no necesitan mucha decoración.

El principio de proximidad nos dice que la gente ve los objetos que están muy juntos como si estuvieran relacionados. Un gran número de formularios difíciles de usar descuidan este principio; Por ejemplo, colocando etiquetas más cerca de otras etiquetas en lugar de ubicarlas en los campos correspondientes a las etiquetas. Cuando se está diseñando el formulario, hay que pensar en las maneras en que los diferentes elementos se relacionan, y asegurarse de que la disposición refleje estas relaciones.

El principio de similitud nos dice que la gente ve los objetos que son similares como si estuvieran relacionados. El principio es relevante para formularios de la misma manera como lo es para los sitios web. En una web, nos aseguramos que el diseño de diferentes tipos de elementos sea consistente (todas los encabezados H1 están en negrita y de 20px, todos los enlaces son de color azul y subrayado, etc.). Con los formularios, la similitud es importante en el mismo sentido. Ser consistente en el diseño ayudará al usuario a comprender y avanzar en la tarea de llenarlos más rápidamente.

2.2.4 Principios básicos del diseño de IU

Al momento de diseñar una interfaz de usuario se deben tener en cuenta una serie de aspectos en relación al usuario y cómo este interactúa con la misma. Se tratan de pautas necesarias para lograr un equilibrio entre la completitud y la usabilidad de las IU. Las características más relevantes se presentan a continuación. (14)

Conocer al usuario

Los objetivos del usuario son los objetivos del diseñador. Se debe aprender acerca de las habilidades y experiencia de los usuarios, qué es lo que necesitan y qué interfaces prefieren, mediante la observación de cómo las utilizan.

Prestar atención a los patrones

Los usuarios pasan mucho tiempo utilizando interfaces que no son las de su software: redes sociales, correo electrónico, banca on-line, etc. No hay necesidad de reinventar la rueda. Esas interfaces pueden resolver algunos de los mismos problemas que los usuarios perciben en la interfaz que se está diseñando. El uso de patrones de IU familiares ayuda a que los usuarios se sientan cómodos.

Ser consistente

Los usuarios necesitan consistencia. Ellos necesitan saber que una vez que aprenden a hacer algo, van a ser capaces de hacerlo de nuevo. Idioma, distribución y diseño son sólo algunos elementos de la interfaz que necesitan consistencia. Una

interfaz consistente permite que los usuarios tengan una mejor comprensión de cómo funcionan las cosas, aumentando su eficiencia.

Utilizar la jerarquía visual

Diseñar la interfaz de una manera que permita al usuario centrarse en lo que es más importante. El tamaño, el color, y la ubicación de cada elemento trabajarán conjuntamente, facilitando la comprensión de la interfaz. Una jerarquía clara podrá ser utilizada para reducir la apariencia de complejidad (incluso cuando las propias acciones son complejas).

Proporcionar retroalimentación

La interfaz debe hablar en todo momento con el usuario. Ya sea cuando sus acciones son correctas, incorrectas o mal entendidas. Siempre se debe informar a los usuarios sobre las acciones, cambios de estado y errores o excepciones que se producen. Las indicaciones visuales o mensajes simples pueden mostrar al usuario si sus acciones han llevado al resultado esperado.

Ser tolerante

No importa cuán claro sea el diseño, la gente comete errores. La interfaz de usuario debe permitir y tolerar el error del usuario. Siempre que sea posible, es conveniente diseñar de modo que el usuario pueda deshacer sus acciones. También es valorable que el sistema sea tolerante con respecto a datos de entrada variados (a nadie le gusta empezar de nuevo porque se puso en el formulario la fecha de nacimiento equivocada). Además, si el usuario provoca un error, se pueden utilizar mensajes para enseñarle al usuario qué acción no fue correcta, y asegurar de que sepa cómo prevenir el error si se vuelve a producir.

Mantenerlo sencillo

Los mejores diseños de interfaz son invisibles. Cuando las cosas van bien en un diseño, no le prestamos atención. Sólo prestamos atención a las cosas que nos molestan. Es como un aire acondicionado en una sala de conferencias. Nadie interrumpe una reunión para decirnos que la temperatura está perfecta. Nadie se da cuenta. Por eso decimos que cuanto mejor sea el diseño, en más invisible se convierte.

2.3 Tecnologías

En esta sección se presentarán las tecnologías estudiadas y sobre las cuales se llevó a cabo el desarrollo de la implementación final asociada al proyecto.

2.3.1 XML

XML, por sus siglas en inglés de eXtensible Markup Language (lenguaje de marcas extensible), es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos en forma legible.

XML es un metalenguaje. En él se definen las reglas por las que se crean los lenguajes específicos de marcado XML, pero no dice nada sobre qué tipos de elementos se utilizan. Cada documento contiene una variedad de tipos de información significativa. Cuando usamos etiquetas XML para codificar, podemos etiquetar partes del documento para distinguir los diferentes tipos de contenido. Estas distinciones son

puramente conceptuales, y estos bits de contenido no tienen ningún tipo de formato o presentación inherentes a ellos.

XML es extensible. Permite la creación de nuevos conjuntos de etiquetas para el contenido específico del dominio. Esencialmente no hay límite para los tipos de elementos que un documento XML puede contener, y los elementos se nombran a menudo para sugerir el significado del contenido.

XML es una tecnología interoperable. En pocas palabras, si las plataformas de envío y recepción pueden leer y escribir archivos codificados con Unicode, y un analizador XML está disponible para esas plataformas, un archivo XML puede ser procesado sin ningún problema. Las diferencias entre los sistemas operativos, o en cuestiones tales como la conversión de conjunto de caracteres, se convierten en preocupaciones mínimas.

Un modelo de datos es una colección de datos de información definidos, relacionados y metadatos. El modelo de datos describe estructuras de metadatos, contenedores y sus características, pero no incluye los valores de datos reales. XML es una sintaxis conveniente para la codificación de los modelos, pero XML en sí, no ayuda a crear buenos modelos. Lo que realmente importa en este caso, es la calidad del análisis y diseño que posee la representación en los modelos conceptuales, antes de codificarlos en vocabularios XML.

2.3.2 XML Schema

XML Schema define los posibles tipos de contenido en un documento y las reglas que rigen la estructura y los valores de ese contenido. Cada esquema XML contiene las definiciones de los tipos de elementos y también especifica los atributos que pueden estar asociados con los elementos. El significado de los elementos se representa a través de las restricciones y reglas que controlan la disposición estructural de elementos y los valores que los elementos y atributos pueden tener. Los tipos de reglas expresadas en definiciones de elementos XML incluyen relaciones de contención, secuencia y cardinalidad y recursividad, entre otros.

El primer lenguaje de esquema XML era Document Type Definition (DTD). DTD tiene una sintaxis muy simple y compacta, y es suficiente para describir los modelos cuyos contenidos sea principalmente texto.

Para los tipos de documentos transaccionales (que se utilizan principalmente por las aplicaciones de negocios) el lenguaje de esquema más útil es el recomendado por el W3C, denominado XSD o XML Schema. XSD fue desarrollado para cumplir con un conjunto de requisitos más amplios y más orientados a sistemas computacionales que los DTD. XSD incluye todos los tipos de datos básicos comunes en lenguajes de programación y bases de datos (string, booleano, número entero, punto flotante, etc.), así como los mecanismos para crear nuevos tipos de datos. Una facilidad muy importante en XSD es su soporte para namespace, que permite distinguir vocabularios XML con el fin de que un esquema pueda reutilizar definiciones y evitar conflictos entre los elementos con el mismo nombre que significan cosas diferentes.

2.3.3 Groovy

Lenguaje dinámico orientado a objetos, muy íntimamente ligado a Java. EL 99% del código Java existente puede ser compilado mediante Groovy, y el 100% del código Groovy es convertido en bytecode Java, y ejecutado en una JVM. Groovy simplifica la sintaxis de Java expresar realmente lo que queremos hacer, y además añade una serie de métodos muy útiles al JDK (15).

Groovy fue construido en base a las fortalezas de Java pero también posee características muy potentes inspiradas por leguajes tales como Python, Ruby y Smalltalk (16).

2.3.4 HTML5

El lenguaje de marcas de la World Wide Web ha sido siempre HTML. HTML fue diseñado principalmente como un lenguaje para describir semánticamente documentos científicos. A pesar de su diseño general las adaptaciones en los últimos años han permitido que sea utilizado para describir una serie de otros tipos de documentos.

HTML5 se creó con la colaboración entre el World Wide Web Consortium (W3C) y el Web Hypertext Application Technology Working Group (WHATWG), y se estableció que HTML5 debería cumplir las siguientes reglas:

- Las nuevas características se deben basar en HTML, CSS, DOM y JavaScript.
- La necesidad de plugins externos (como Flash) debe ser reducida.
- El manejo de errores debe ser más fácil que en las versiones anteriores.
- El scripting tiene que ser sustituido por más etiquetas de marcado.
- HTML5 debe ser independiente del dispositivo.
- El proceso de desarrollo debe ser visible para el público.

.

HTML5 todavía no es un estándar oficial, y ninguno de los navegadores tiene soporte completo. Sin embargo, los principales navegadores (Safari, Chrome, Firefox, Opera, Internet Explorer) continúan agregando nuevas características de HTML5 a sus últimas versiones.

HTML5 es multiplataforma y está diseñado para ofrecer casi todo lo que se quiere hacer en línea, sin necesidad de plugins adicionales. Lo hace todo, desde la animación de aplicaciones, música para películas, y también se puede utilizar para crear aplicaciones complejas que se ejecutan en el navegador, incluso para que operen offline.

Algunas de las novedades que ofrece HTML5 son:

- Incorpora etiquetas (canvas 2D y 3D, audio, video) con códecs para mostrar los contenidos multimedia.
- Etiquetas para manejar grandes conjuntos de datos: Datagrid, Details, Menu y Command. Permiten generar tablas dinámicas que pueden filtrar, ordenar y ocultar contenido en cliente.
- Mejoras en los formularios. Nuevos tipos de datos (email, number, url, datetime, etc.) y facilidades para validar el contenido sin JavaScript.

- Visores: MathML (fórmulas matemáticas) y SVG (gráficos vectoriales). En general se deja abierto a poder interpretar otros lenguajes XML.
- Drag & Drop. Nueva funcionalidad para arrastrar objetos como imágenes.

En (17) se puede comprobar cuantas de las nuevas características de HTML5 soportan los diferentes navegadores.

HTML5 introduce muchas características de vanguardia que permiten a los desarrolladores crear aplicaciones y sitios web con la funcionalidad, velocidad, rendimiento y experiencia de aplicaciones de escritorio. Pero a diferencia de las aplicaciones de escritorio, las aplicaciones basadas en la plataforma web puede llegar a un público mucho más amplio con una gama más amplia de dispositivos.

2.3.5 XAML

El lenguaje de marcado de aplicaciones extensible (XAML por sus siglas en inglés) es un lenguaje declarativo basado en XML, desarrollado por Microsoft y optimizado para describir gráficamente interfaces de usuarios visuales ricas desde el punto de vista gráfico.

XAML forma parte de la Microsoft Windows Presentation Foundation (WPF). WPF permite el desarrollo de interfaces de interacción en Windows tomando características de aplicaciones Windows y de aplicaciones web.

WPF usa XAML para crear interfaces de usuario de una excelente calidad visual en lenguaje de marcado en lugar de lenguaje de programación, como por ejemplo C#. Es posible crear documentos elaborados de IU totalmente en XAML con elementos tales como controles, texto, imágenes, formas, animación, entre otros. Como XAML es declarativo (al igual que HTML), se requerirá la adición de código si fuera necesario agregar lógica en tiempo de ejecución a la aplicación. Por ejemplo, si la aplicación sólo usa XAML, se pueden crear y animar elementos de la interfaz de usuario y configurarlos para que respondan de un modo limitado a los datos proporcionados por el usuario (mediante desencadenadores de eventos). Pero la aplicación no podrá realizar cálculos ni responderlos, ni podrá crear espontáneamente nuevos elementos de IU sin la adición de código. El código de una aplicación XAML se almacena en un archivo distinto del documento XAML. Este código se almacena en un archivo con el mismo nombre, pero con la extensión adicional .cs o .vb. El hecho de que el diseño de la interfaz de usuario esté separado del código subyacente permite a programadores y diseñadores trabajar juntos en el mismo proyecto sin interferir mutuamente en su trabajo.

3 OpenEHR

OpenEHR es una comunidad que trabaja sobre la definición de estándares y modelos útiles para aplicar sobre los sistemas e-Salud. Principalmente, se busca que los sistemas de salud registren en forma electrónica los datos clínicos de la población de pacientes.

El modelo de desarrollo que plantea OpenEHR (18), es conocido como un modelo dual, en el cual se diferencian el modelo de información y el modelo de conocimiento. El primero permite definir cualquier estructura de información clínica para una aplicación. En tanto, el modelo de conocimiento (también llamado modelo de arquetipos) representa una base de conocimiento creada por profesionales de la salud. A través de los arquetipos se especifican las estructuras de información que representan los diferentes conceptos clínicos concretos (presión arterial, frecuencia cardíaca, prescripción de medicamentes, etc.), que posteriormente serán parte de un registro médico. Los arquetipos se definen por fuera del software, mediante una sintaxis formal llamada Archteype Definition Language (ADL).

El objetivo principal del modelo dual, es lograr separar la gestión del conocimiento clínico, de la gestión del software. Esto presenta varias ventajas. Si ocurre un cambio en el conocimiento clínico, éste no repercute de forma drástica sobre la implementación asociada, porque el cambio ocurre a nivel del modelo de arquetipos mantenido por fuera del software. Esto permite que el software se adapte fácilmente a cambios en los requerimientos referentes a la estructura de los registros clínicos. Además hay una participación más activa de los profesionales sanitarios, dado que forman parte fundamental en el modelado de arquetipos. Esto implica una metodología de trabajo que difiere de la que estamos acostumbrados a realizar en los procesos de creación de software. Por ejemplo, el relevamiento de requerimientos.

3.1 Modelo de información

El modelo de información (IM) de OpenEHR engloba todos los conceptos, relaciones y restricciones que permiten especificar la semántica de los datos utilizados en el dominio de la salud y su registro electrónico. El IM está pensado para “durar en el tiempo”, es decir, debe ser una versión lo suficientemente estable, evitando así sufrir constantes cambios a nivel de software.

Para comprender mejor esta organización, se debe tener en cuenta que el modelo planteado por OpenEHR se basa en el contexto en donde se desarrollan los sistemas sanitarios. Cada situación clínica que se plantee en la realidad, tiene un nivel en el modelo de información, definiendo una correspondencia directa.

De acuerdo a esto, toda la información que se genera en un registro médico es expresado en una instancia de la clase ENTRY (*ver Ilustración 3.1*), la cual podríamos describir como capaz de representar un “hecho clínico”.

El package ENTRY está modelado por OpenEHR según el proceso de registro clínico (Clinical Investigator Recording) (19). En dicho proceso se plantea seguir el flujo típico de un sistema de salud (*Ilustración 3.2*), el cual se compone de los siguientes cuatro tipos de eventos:

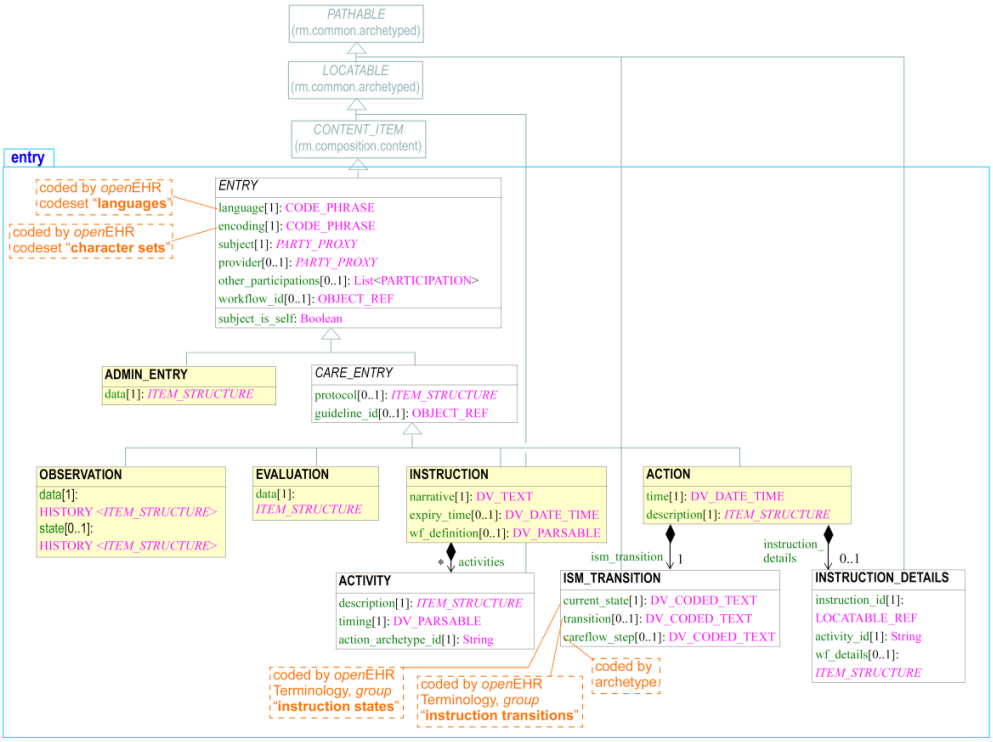


Ilustración 3.1: Diagrama del ENTRY Package

Observación (Observation)

En el contexto de la salud, una observación se define como un hecho clínico en el cual el personal sanitario registra el resultado de una medición o control que efectúa sobre un paciente, con el fin de obtener una clasificación inicial del mismo.

Evaluación (Evaluation)

Encierra todas las conclusiones y diagnósticos que un médico puede inferir a partir de las observaciones y del conocimiento clínico propio o adquirido sobre la materia.

Instrucción (Instruction)

Hace referencia a cualquier indicación clínica que surja en consecuencia de la interacción médico-paciente.

Acción (Action)

Registro de cualquier acto clínico, en general derivado de una instrucción.

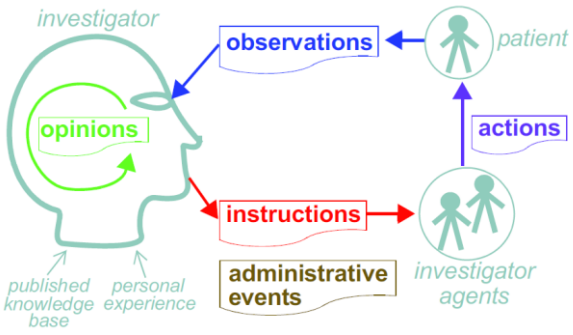


Ilustración 3.2: Proceso del registro clínico (Clinical Investigator Recording)

3.1.1 Tipos de dato

Los tipos de dato que define OpenEHR (20) están diseñados para abastecer los requerimientos propios de un registro clínico, facilitando la expresión e implementación del mismo, y permitiendo la interoperabilidad con tipos de datos definidos en otros estándares.

El package DATA_TYPE está estructurado en cinco sub-package: BASIC, TEXT, TIME_SPECIFICATION, URI, QUANTITY y ENCAPSULATED. A continuación se puede ver una breve descripción de los package más relevantes.

BASIC

Maneja las definiciones para tipos de datos de doble estado (DV_BOOLEAN), estados de máquina (DV_STATE) y datos de identificación única (DV_IDENTIFIER).

ENCAPSULATED

Permite el manejo de datos cuya estructura interna se encuentra definida fuera de OpenEHR, tal como es el caso de archivos multimedia.

QUANTITY

Incluye varias clases útiles para representar diversos valores y cantidades numéricas que conviven en un registro médico, tales como:

- *Valores simbólicos:* Existen valores que no son cuantificados. Estos se modelan a través de símbolos que representan una magnitud. Un ejemplo puede ser "*Grado de una quemadura*", la cual podrá tomar los valores: "*primer orden*", "*segundo orden*", etc.

- *Elementos contables:* Son en general valores enteros que representan por ejemplo: "Cantidad de dosis", "Cantidad de cirugías previas", etc.

- *Unidades dimensionadas:* Formada por un valor real y la dimensión que representa. Ejemplo: "*Altura: 170 cm*"

- *Relaciones y proporciones*: Las relaciones son comúnmente definidas con valores enteros, por ejemplo "*Concentración de Sodio en potación NA: K*". Mientras que las proporciones se expresan en valores de porcentaje (%), como por ejemplo "*Porcentaje de distribución de glóbulos rojos en sangre (RDW)*"

- *Fórmulas:* Se trata de un concepto similar a las relaciones. Ejemplo "*250 mg / 500 ml*"

- *Rangos numéricos:* Son utilizados para definir un rango de valores que puede ser de referencia, para definir valores como "colesterol deseable < 5.5 mmol/L" o también para definir rangos de administración de sustancias.

TEXT

Contiene las clases que permiten representar campos de texto en un registro médico. Esto puede incluir tanto texto plano como codificación de términos.

TIME_SPECIFICATION

A través de clases de éste tipo, es posible representar indicaciones temporales que se repiten a lo largo del tiempo o que tiene una fecha límite. A modo de ejemplo "Una vez cada 8 horas", "El primer domingo de cada mes", etc.

URI

Permite referenciar objetos de información en la web. Incluso es posible definir para cualquier elemento de un registro de OpenEHR un identificador de tipo URI. Por ejemplo, a través de un tipo de dato URI, es posible desde un documento clínico hacer referencia a otro relacionado.

3.2 Modelo de Conocimiento

El modelo de conocimiento, también llamado Modelo de Arquetipos (AOM por sus siglas en inglés) (5) representa la estructura y semántica de los arquetipos. Para lograr esto, se cuenta con un lenguaje de definición de arquetipos (ADL) (21), un modelo de objetos para arquetipos (AOM) y un perfil de arquetipos (oAP). El ADL permite expresar los arquetipos a través de una sintaxis abstracta. Con el AOM es posible crear modelos de objetos equivalentes, tal como si se estuviese trabajando con diagramación UML. Y el oAP define las restricciones de uso sobre los objetos del AOM.

3.2.1 Arquetipos

Un arquetipo representa un concepto clínico, como puede ser "Administración de Sustancia", "Transfusión", etc. Desde el punto de vista de OpenEHR, un arquetipo se puede entender como una composición de diferentes niveles de nodos, que pueden ser tanto objetos como atributos. Un atributo es una propiedad de datos de una clase que puede representar una relación o un dato primitivo.

Los arquetipos se pueden ver como "restricciones sobre el modelo de información". Restricciones sobre cómo se estructura la información clínica y sobre qué valores son válidos al momento de que un usuario ingrese datos.

Existen diversas características que vale la pena destacar sobre los arquetipos:

- Es posible componer varios arquetipos, creando conceptos más complejos a partir de otros más simples. Estructuralmente, un arquetipo puede contener otro arquetipo a través del uso de slots.
- Dado un arquetipo es posible modificarlo de manera de obtener una especialización del mismo. Es decir que, teniendo un concepto clínico definido a través de un arquetipo, el mismo puede "adecuarse" a necesidades o requerimientos más restrictivos.
- Los arquetipos soportan versionado
- Dentro de un arquetipo los nodos son referenciados de forma única por una ruta (path). Con esto se logra que las restricciones se declaren específicamente para cada nodo de forma individual.

Cada arquetipo tendrá un tipo asociado, dependiendo de la actividad a la que pertenezca: OBSERVATION, EVALUATION, INSTRUCTION o ACTION. A nivel del modelo de información, las clases que modelan cada uno de estos tipos se hallan por

fuera de la definición de los arquetipos, aun cuando estas están estrechamente relacionadas con los mismos.

3.2.1.1 Lenguaje de definición de arquetipos

Tal como fue presentado, el ADL es un lenguaje formal que permite representar la estructura de los arquetipos. Este lenguaje utiliza a su vez otras tres sintaxis: cadl (constraint form of ADL) que expresa la definición del arquetipo, dADL (data definition form of ADL) que permite representar los datos que se hallan bajo las secciones de lenguaje, descripción, ontologías y revisiones históricas, y FOPL (first-order predicate logic) necesaria para describir restricciones sobre los datos instanciados desde el modelo de información.

Se debe tener en cuenta que el AOM define la semántica del arquetipo y las relaciones entre sus componentes, mientras que el ADL define la sintaxis formal para poder expresar dichas estructuras.

En el *Anexo A "ADL Archetype Definition Language: Caso de estudio",* se puede ver desglosado un ejemplo de una descripción en ADL en la cual se detallan cada una de sus secciones. La creación de un ADL para un arquetipo dado puede resultar una tarea poco sencilla a medida que crecen, tanto los elementos que lo componen (estructura) o datos extra, como las traducciones a diferentes lenguajes. Por esto existen herramientas que facilitan esta tarea, y en las cuales no es necesario trabajar a bajo nivel. En la sección 3.3 se describen alguna de estas herramientas.

3.2.2 Template

Un template permite agrupar diferentes definiciones de arquetipos bajo alguna "regla de negocio", o como se define en el área de la salud, un "servicio médico". Dicho de otra manera, el template permite referenciar varios arquetipos que para determinado hecho clínico, tienen una relación lógica.

La sintaxis que propone OpenEHR aún está en desarrollo (22), pero maneja ciertas ideas con respecto a la estructura y funcionalidad de un template, que vale la pena destacar:

- *Composición:* acción de combinar arquetipos en estructuras más complejas.
- *Selección de elementos o nodos*: selección y especialización de elementos o nodos de los arquetipos que pertenecen al template final. La especialización permite describir cuando un nodo es opcional u obligatorio.
- *Restricciones:* definición de reglas o restricciones sobre elementos de arquetipos
- *Valores por defecto:* control sobre los valores de los nodos en tiempo de ejecución

3.3 Herramienta de edición y visualización

Durante esta sección hablaremos de diferentes herramientas relacionadas con el manejo de arquetipos y templates (23).

3.3.1 Archetype Editor

Archetype Editor (24) es una herramienta que permite diseñar arquetipos. Esta tarea de diseño se desarrolla a alto nivel, lo que significa que el usuario a través de una intuitiva pantalla puede definir un arquetipo, enfocándose simplemente a construir el concepto clínico que representa. Como resultado, se generará de forma automática la definición del arquetipo en formato ADL con la estructura de nodos descrita en el *Anexo A "ADL Archetype Definition Language: Caso de estudio"* .

Existen varias ventajas a destacar de Archetype Editor. La primera, y más importante es que la construcción de un arquetipo no requiere crear y editar un archivo ADL, sino todo lo contrario: la interfaz permite declarar todos los elementos que lo componen, restricciones sobre estos datos, etc., y la herramienta genera paralelamente el ADL correspondiente que luego podrá ser utilizado por otro sistema. Otra característica a destacar es que a medida que se construye el arquetipo es posible visualizar esta definición no solo como el ADL final, sino también como como un formulario web, XML, etc., permitiendo al usuario tener una visión semejante de cómo se verá el concepto que está definiendo una vez se esté utilizando en el sistema final.

3.3.2 ADL Workbench

Se trata de una herramienta que facilita la navegación entre los diferentes arquetipos pertenecientes a un modelo dado. Para cada arquetipo se pueden visualizar los siguientes componentes (25):

- Descripciones generales como autor, lenguaje, propósito, uso, etc.
- Definición del arquetipo, que se representa visualmente en forma jerárquica de modo el mismo se puede desglosar nodo a nodo hasta llegar al tipo de dato asociado al valor de dicho nodo. .
- Terminologías, ontologías, paths (rutas con la que se identifica cada nodo dentro de un arquetipo)
- Definición ADL del arquetipo
- Diferentes estadísticas sobre elementos que se utilizan

Como ventaja a destacar de esta herramienta es que permite agrupar bajo una misma interfaz todos los componentes que forman parte de la estructura de un arquetipo, así como también la relación de éste en un modelo dado.

3.3.3 ADL - syntax highlight for Notepad++

Este plugin para Notepad permite que esta herramienta identifique la gramática de un ADL, facilitando así la visualización de la estructura de un arquetipo.

3.3.4 Template Designer

Asociado a la herramienta EHRGen (*ver capítulo 3.4 EHRGen*), se cuenta con un diseñador web de templates, Template Designer (TD) (26), mediante el cual pueden

definirse diferentes plantillas. Dado que básicamente un registro clínico se puede ver como un formulario, la herramienta permite crear este tipo de formulario agrupando los registros de acuerdo a unos esquemas predefinidos.

En la interfaz del TD se presenta cuatro secciones:

Arquetipos

Despliega el árbol de arquetipos predefinidos, los cuales podrán ser utilizados en la creación del registro clínico.

Arquetipo Raíz

Una vez seleccionado el arquetipo a utilizar de Arquetipos, se puede visualizar la estructura interna del mismo.

Template

Muestra el esqueleto base donde se partirá para construir registro clínico. El TD, ya define templates básicos con subsecciones definidas, para que a partir de allí se agrupen los diferentes elementos. Desde el arquetipo seleccionado, se podrán desplazar hacia el template los elementos que se quieren estén presentes en el registro, indicando el orden relativo entre ellos. La desventaja en este punto radica en que cada vez que se cambia a un nuevo arquetipo raíz se pierden los elementos ya agregados en el template.

Menú

Menú con las acciones necesarias para Crear un nuevo template, Abrir un ya existente, Guardar, etc.

La ventaja de esta herramienta, similar a los que sucedía con el Archetype Editor, es que permite diseñar el template a alto nivel, y haciendo uso de utilidades gráfica como drag and drop, hace que la interfaz resulte bastante intuitiva y amigable. Una desventaja a señalar de esta herramienta es que está sujeta al modelo de template definido en la EHRGen que se presentará en la sección *3.4.2 Template.*

3.4 EHRGen

En el marco de OpenEHR se relevaron aplicaciones que hicieran uso de las estandarizaciones que realiza esta comunidad sobre los sistemas de salud (27), con especial énfasis en aquellas herramientas de código abierto.

EHRGen (28) es una herramienta de código abierto basada en los estándares establecidos por OpenEHR, y como tal permite la generación de sistemas de historia clínica electrónica orientados a la gestión de conocimiento. Esta herramienta toma como base la arquitectura propuesta por OpenEHR y agrega cinco módulos más: Information Model, Data Binder, GuiGen, Demographic Access, Knwolwdge Access y Workflow Manager (18).

3.4.1 Metodología de trabajo

La metodología de trabajo en esta herramienta se puede dividir en cuatro etapas: modelado de la base de conocimiento, generación de los registros clínicos, creación de las interfaces gráficas y utilización del sistema final.

Modelado de la base de conocimiento

En esta etapa se lleva a cabo la creación de los arquetipos. Se deben identificar qué conocimientos simples van a formar parte de nuestro sistema que se generará tomando como base dichos elementos (arquetipos). El proceso que se realiza en esta etapa se basa en el desarrollo orientado al conocimiento. Personal con perfil médico será responsable de modelar la base de conocimiento para su posterior utilización.

Generación de registros clínicos

Consiste en tomar varios arquetipos predefinidos y agruparlos bajo cierta lógica en un template. EHRGen permite definir múltiples registros, sin tener que alterar el código fuente de la aplicación. Posteriormente, los datos que son ingresados en los formularios de registros generados, serán validados contra las restricciones de datos definidas en los arquetipos que lo componen.

Generación de interfaces gráficas

El generador automático de interfaces toma como entrada los template definidos y a partir de allí genera las interfaces web en HTML El componente responsable de realizar esta tarea (GuiGen) es capaz de generar un componente grafico para cada uno de los modos del registros (create, show y edit).

Uso del sistema final

Por último, la utilización del sistema está gestionada por un WFM, encargado de recibir las solicitudes de los clientes para retornar la interfaz relacionada.

3.4.2 Template

Dado que la definición de template presentada por OpenEHR resultaba compleja y aún estaba en pleno desarrollo, para la aplicación EHRGen se definió un modelo de templates que se ajustara a las necesidades de este framework (29).

Como se puede ver en la *Ilustración 3.3*, se destacan las siguientes características:

- Inclusión de arquetipos
- Inclusión y exclusión de nodos particulares de los arquetipos
- Selección de controles que se visualizarán en la interfaz de usuario
- Otra información para generar la interfaz de usuario, tal como la zona donde desplegará determinado control asociado a un nodo.

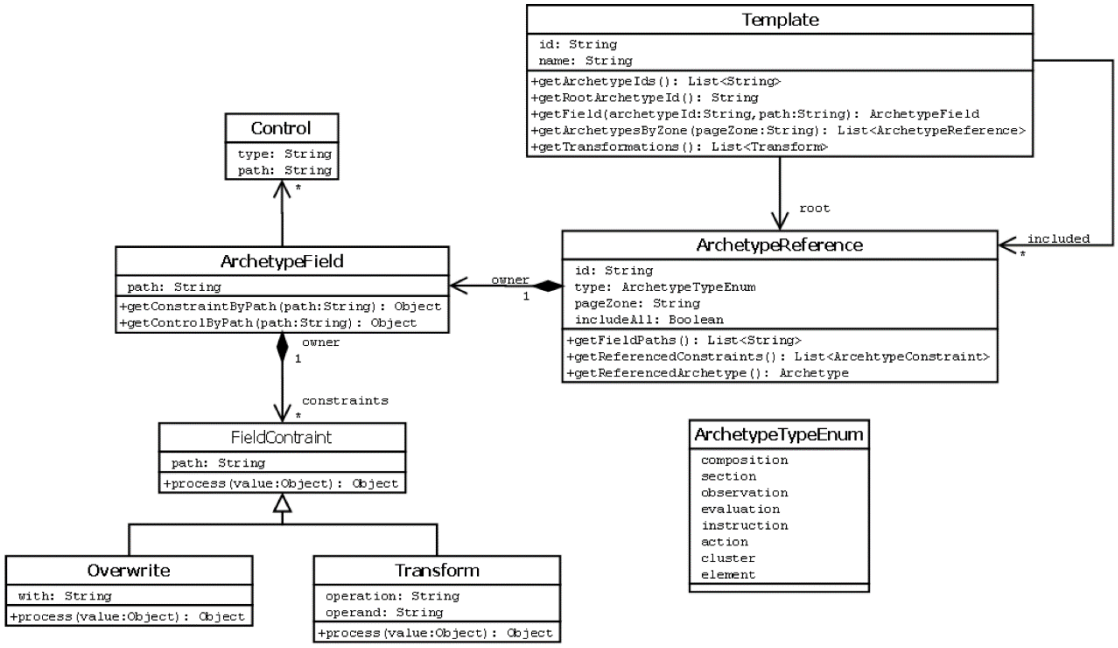


Ilustración 3.3: Modelo UML del Template Object Model de EHRGen

3.4.2.1 Limitaciones del modelo

Este modelo template presenta varias limitaciones. Una de ellas es con respecto a la representación del Layout, dado que a los nodos de arquetipos referenciados solo se les puede asociar una posición en la IU final (right, left, content, button) tal como se ve en la *Ilustración 3.4*. Esta característica impide hacer diseños de interfaces más complejas.

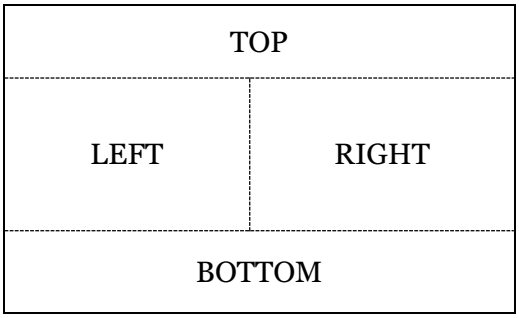


Ilustración 3.4: Layout de template

Por otra parte este modelo de template no permite dar soporte a otros componentes del modelo de información relacionados con los arquetipos, tales como las clases OBSERVATION, EVALUATION, INSTRUCTION, y ACTION. Esto se resuelve a nivel de la herramienta.

A demás no es posible definir para un mismo nodo, varios Overwrite, uno por cada lenguaje al que se quiera dar soporte. Se debe tener en cuenta que dentro de

arquetipo las notaciones pueden estar definidas en diferentes lenguajes, por lo que sería conveniente que en el mismo template se pudieran agregar, para cada referencia (ArchetypField), los Overwrite para todos los lenguajes incluidos.

Si se crea un template que hace referencia a un arquetipo de este tipo, sólo se podrá definir un Overwrite para un lenguaje, lo que

En EHRGen, se pueden ver múltiples IU que fueron generadas a partir de diferentes definiciones de template, e integradas a una misma IU. Esto ocurre porque el modelo de template planteado no permite definir varios formularios y unificarlos bajo una mima interfaz. Esto es algo útil ya que es al definir un template, se puede pensar como un conjunto de registros relacionados, y que dicha definición no dependa de la aplicación final que haga uso de estas definiciones de IU.

Otra limitante del modelo es que a nivel de referencias de arquetipos no define identificadores únicos dentro de un template.

4 Organización del trabajo

En esta sección se presentarán los principales puntos que definen el alcance del proyecto, y se realizará una breve descripción de la metodología y gestión del trabajo llevado a cabo.

4.1 Alcance

Para arribar al alcance final del proyecto, se cumplió con un conjunto de requerimientos que detallamos a continuación:

- Estudiar del estado del arte en tres líneas de trabajo independientes: generación de código, interfaces de usuario y OpenEHR.

- Lograr una descripción del paradigma multinivel que incluyó la especificación de interfaces de usuario en función del modelo OpenEHR y el modelo de arquetipos. Dicha especificación tomó como base la ya definida en la herramienta EHRGen.

- Especificar la generación de artefactos de interfaz de usuario en tecnologías concretas, lo que se conoce como Implementation Technology Specifications (ITS), partiendo de la descripción de IU desarrollada en el punto anterior. Esta generación en tecnologías concretas, como XAML, HTML, SwiXML, XUL, etc., deberán ser generadas de forma automática o semi-automática.

- Desarrollar un conjunto de herramientas que permitan probar y validar las especificaciones logradas en los puntos anteriores. Las tres herramientas básicas que serán las siguientes:
  - Diseñador de interfaces de usuario que para generar las definiciones de IU (o templates) que luego serán procesadas por el generador.
  - Generador de interfaces de usuario.
  - Prototipo de aplicación que utiliza las interfaces generadas.

4.2 Metodología de trabajo

El proyecto tuvo una duración de nueve meses, y a grandes rasgos, se conformó de 3 etapas. La primera etapa consistió en el estudio del estado del arte presentado en el capítulo 2 y 3, insumiendo aproximadamente 2 meses de trabajo. La segunda etapa se correspondió a la implementación de las herramientas presentadas en el capítulo 5, con una duración de trabajo cercana a los 6 meses. Por último se culminó la documentación, la cual tuvo una duración de aproximada de 2,5 meses, solapándose con los últimos meses de desarrollo de la segunda etapa. A continuación se presenta un diagrama de los hitos más importantes del proyecto.

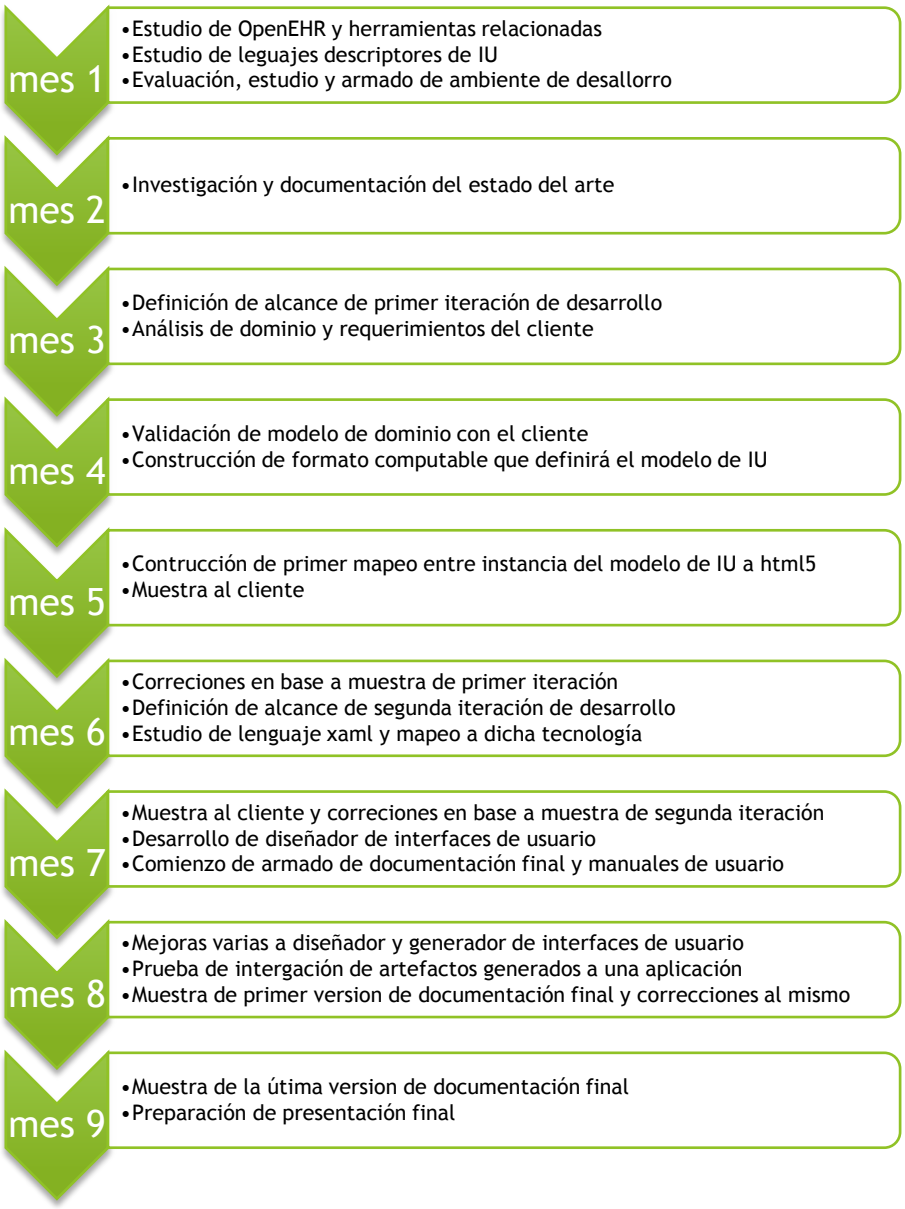


Ilustración 4.1: Metodología de trabajo

4.3 Gestión del proyecto

En cuanto a la organización, se creó un grupo de Google conformado por todos los participantes del proyecto, con el objetivo de centralizar el flujo de mails. Se abrieron directorios en Google Drive para organizar y compartir archivos de en forma ordenada. Por último, para compartir el código fuente del proyecto, se usó SVN.

En materia de comunicación, se concretaron numerosas reuniones presenciales entre las integrantes del proyecto (especialmente en etapas de diseño y modelado), y diarias puestas a punto de manera remota. A su vez, en la durante la etapa de desarrollo e implementación se llevaron a cabo reuniones quincenales con la presencia de tutor y del cliente, con el objetivo de ajustar el alcance del proyecto y llegar a un acuerdo ecuánime entre ambas partes.

La comunicación con el cliente fue muy fructífera para el proceso de desarrollo gracias a su constante disposición, interés y gran conocimiento en los temas estudiados. Además, su amplia visión del proyecto y sus constantes aportes, siempre daban resultado a un producto de mayor calidad.

El tiempo dedicado al proyecto se clasificó de acuerdo a las siguientes categorías:

- Análisis y diseño _ Diseño de CU, arquitectura, prototipo.
- Comunicación _ Reuniones, seguimiento.
- Documentación _ Estado del arte, manuales, documentación final.
- Implementación – Implementación.
- Investigación _ Auto estudio.
- Requerimientos _ Especificación de requerimientos, alcance.
- Verificación _ Pruebas del sistema.

En el *7 "Planificación, Esfuerzo efectivo*", se presenta un gráfico de con la distribución horaria total por categoría.

5 Presentación de la solución

OpenEHR plantea una metodología de diseño multi-nivel de software para el dominio de la salud, donde se busca expresar metadatos semánticos que describen distintas partes del software mediante artefactos computables estándar, utilizando herramientas de diseño y generación. Luego, las aplicaciones de software para usuarios finales que implementen el estándar, podrán utilizar los artefactos generados.

El foco de la solución planteada en este trabajo se centra en la especificación y generación de interfaces de usuario para el registro clínico. Esto se basa en el modelo de información, implementado en forma genérica y estable, y en el modelo de gestión del conocimiento (modelo de arquetipos) que representa la parte cambiante del registro clínico y permite crear especificaciones sobre el modelo de información genérico.

Por otra parte, el nivel de interfaces de usuario se basa en el modelo de arquetipos, especificando cómo se deberán construir las interfaces de usuario para los distintos registros clínicos.

En la *Ilustración 5.1* se puede ver un diagrama de componentes por nivel que hacen a la solución final, donde cada nivel es el resultado de la composición de los subcomponentes del nivel inferior, los cuales serán presentados en las siguientes secciones.

Ilustración 5.1: Esquema de la solución: generador y diseñador de interfaces de usuario

5.1 Generador de interfaces de usuario

El generador de IU construido, es un tipo de generador pasivo basado en un modelo por nivel. Por lo tanto cumple con las siguientes características:

- Una vez que genera el código esperado se deslinda totalmente del producto final obtenido.
- Genera código en un nivel, es decir, toma como insumo archivos de configuración y, en combinación con las plantillas definidas, retorna el código final.

Para la construcción del generador se utilizó el proceso de cinco etapas, necesario para construir un generador basado en un modelo por nivel, con la salvedad que se intercambiaron los dos primeros pasos. Como fue mencionado en la sección *2.1,* dicho proceso cuenta de las siguientes fases: construcción del código de prueba, diseño del generador, desarrollo del analizador de datos de entrada, desarrollo de los template a partir de código de prueba y desarrollo del constructor de código de salida.

Este capítulo se estructurará según las cinco etapas mencionadas, detallando en cada de ellas el trabajo realizado.

5.1.1 Diseño del generador

Objetivos principales:
Especificar los requerimientos sobre los metadatos
Diseñar el formato de serialización de instancias del modelo

Durante el diseño del generador se debieron tomar dos decisiones claves. La primera, especificar sobre qué conjunto de datos se iba a trabajar para lograr la salida esperada. Mientras que la segunda decisión se basó en definir cómo codificar los datos de entrada, los cuales serán tomados como insumo por el generador para construir las IU en las diferentes tecnologías.

5.1.1.1 Especificación de requerimientos sobre los metadatos

Los requerimientos se especificaron sobre los metadatos, tanto a nivel de definiciones de estructura de información y texto en diferentes idiomas, como de restricciones.

El modelo utilizado es el de arquetipos de OpenEHR. Para acotar este conjunto, se utilizarán únicamente arquetipos que cumplan con las siguientes características:

Arquetipos planos
Los arquetipos (*ver 3.2.1 "Arquetipos"*) planos son aquellos que no hacen referencia a otros arquetipos, o a parte de ellos, a través de slots.

Arquetipos con tipos de datos acotados
Se hará uso de arquetipos que únicamente utilicen alguno de los siguientes tipos de datos:

- *DV_TEXT:* texto de largo variable (palabra, frase, etc.).
- *DV_CODEDTEXT:* texto dependiente de una terminología definida.
- *DV_COUNT:* datos para tipos numerables.

- *DV_QUANTITY:* representación de cantidades científicas, como "magnitud/medida". También se puede utilizar para representar medidas de tiempo en las que, por ejemplo, sólo es necesario desplegar "cantidad de segundos".

- *DV_DATE:* refiere a un punto en el tiempo. La notación está regida por la norma ISO 8601 (30).

OpenEHR Entry

Soporte para las clases asociadas a los tipos de registros INSTRUCTION, con sus respectivas ACTIVITY y ACTION (*ver Ilustración 3.1*)

5.1.1.2 Diseño del formato de serialización de instancias del modelo

El principal problema a resolver en este punto fue la elaboración de un nuevo y mejorado modelo de especificación de IU al que denominamos UITemplate. Para ello se consideraron características básicas del modelo actual definido en la herramienta EHRGen, pero superando sus limitaciones (*ver sección 3.4.2.1 "Limitaciones del modelo"*). Los nuevos requerimientos a cumplir fueron los siguientes:

REQTMP01. En el modelo actual, un template equivale a un formulario. Teniendo en cuenta que varios formularios arman un documento clínico, el nuevo formato deberá permitir la definición de varios formularios bajo un template.

REQTMP02. Cuando existan más de un formulario dentro del template, será necesario generar en la IU elementos que permitan la navegación entre ellos, como por ejemplo tabs (viñetas). La posición de dichos elementos debe ser configurable (above, below, left, right).

REQTMP03. Permitir definir en un template varios layouts. Cada uno de los formularios deberá poder asociarse con alguno de los layouts definidos.

REQTMP04. Un formulario podrá incluir referencias a arquetipos o partes de ellos. Cada referencia a un nodo de un arquetipo, deberá manejar información acerca de cómo el mismo se visualizará en la IU final. Las restricciones sobre las referencias deberán ser las siguientes:

a. Posición de los campos a generar en un layout.
b. Tipo de control.
c. Valor por defecto.
d. Propiedad nullflavor indicando que se genera un campo para el que no se tiene valor.
e. Si un valor es requerido u opcional.

Nuevo modelo de template

A continuación se detalla el diseño conceptual del nuevo modelo de template. En la *Ilustración 5.2* se presentan los conceptos y relaciones definidas, mientras que en

la *Ilustración 5.3* se describen los tipos enumerados que se utilizan en el diseño presentado. Luego se realizará una breve descripción y análisis de los componentes para verificar que los requerimientos planteados anteriormente se cumplan en la nueva especificación del modelo.

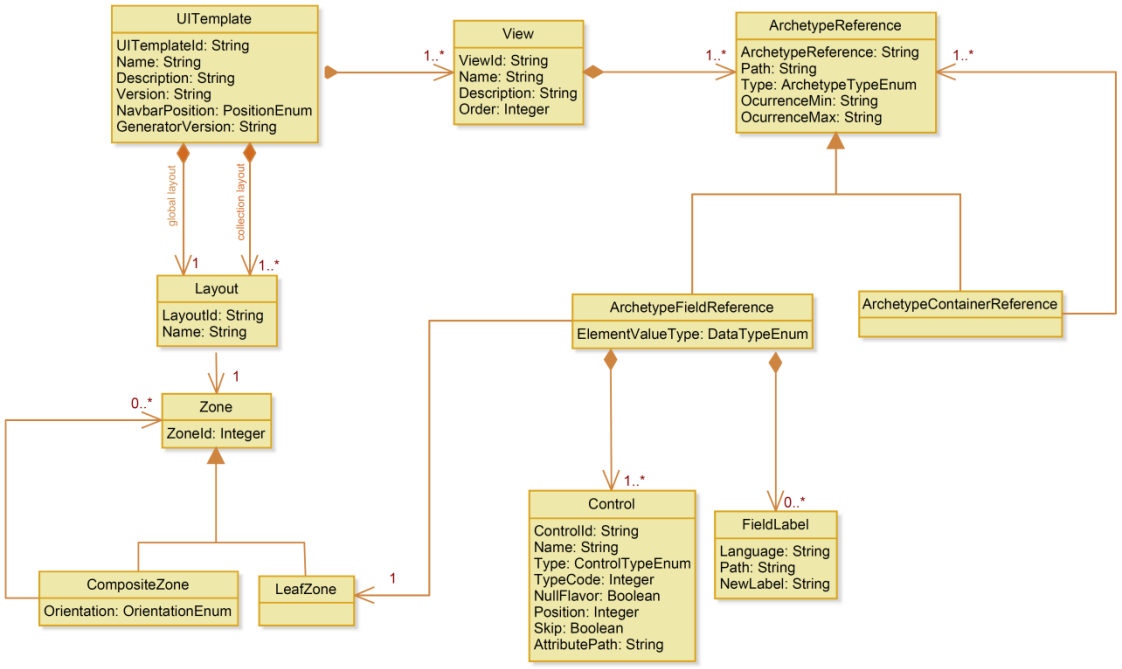


Ilustración 5.2: Modelo de diseño del template

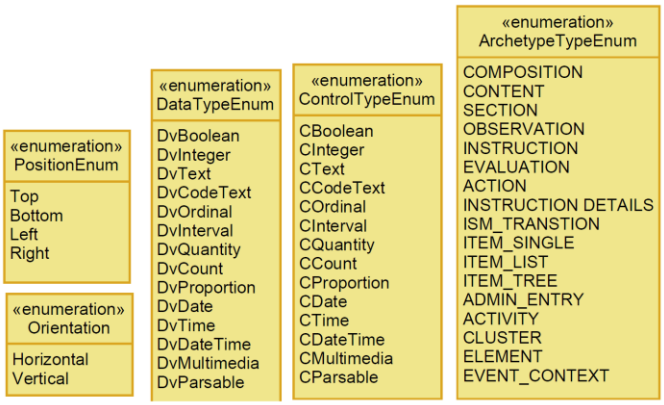


Ilustración 5.3: Modelo de diseño del template - Tipo de datos

UITemplate

Plantilla sobre la cual se diseña la estructura de las diferentes interfaces de usuario (IU). Cada UITemplate está integrado por un conjunto de Views (REQTMP01).

Para un UITemplate se pueden definir una colección de layouts, pero un UITemplate tiene asociado un único layout global (REQTMP03). Éste será heredado por los Views que pertenezcan a dicho UITemplate.

View

Representa el diseño de una IU. La misma contendrá todos los datos/campos que puedan visualizarse en la IU que se esté diseñando. A su vez un View puede tener asignado un Layout diferente al layout global definido en el UITemplate. Los View tienen un orden relativo dentro del UITemplate al que pertenecen (REQTMP02).

Layout

Un Layout indica la disposición general de los elementos de una IU, definiendo grandes zonas dentro de la interfaz de usuario.

Zone, CompositeZone, LeafZone

Las Zone permiten dividir un Layout. Una Zone puede ser de tipo hoja (LeafZone) o compuesta (CompositeZone).

Una CompositeZone puede estar dividida en más de una zona para lograr la estructura final del Layout. La orientación de corte de la misma puede ser vertical u horizontal, variando con respecto a la orientación de sus zonas hijas. Esta restricción mantiene la consistencia del diseño de interfaces, dado que impide que dos definiciones diferentes de Layout generen el mismo diseño. Por otra parte, una Zone que no admite divisiones va a ser de tipo LeafZone.

En la *Ilustración 5.4* se puede ver cómo dividir un Layout en diferentes zonas. Partiendo desde el Zone 1 se muestra el paso a paso hasta llegar a la estructura final.

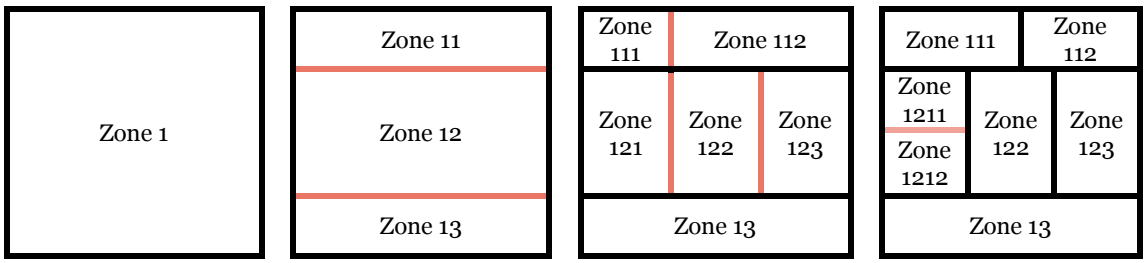


Ilustración 5.4 : Diseño de template

El Layout representado se ajusta a la estructura de árbol planteada en la *Ilustración 5.5*. Allí se puede ver cómo cada nodo representa una instancia de las clases involucradas en el diseño del Layout (CompositeZone y LeafZone).

ArchetypeReference, ArchetypeFieldReference, ArchetypeContainerReference

ArchetypeReference representa una referencia a un arquetipo. Define los elementos para los que se deberán desplegar controles en la IU final, generada a partir de un UITemplate. Indica qué arquetipo (ArchetypeReference) y qué partes del mismo (ArchetypeReferencePath) se utiliza. El ArchetypeReference puede ser de tipo field (ArchetypeFieldReference) o container (ArchetypeContainerReference).

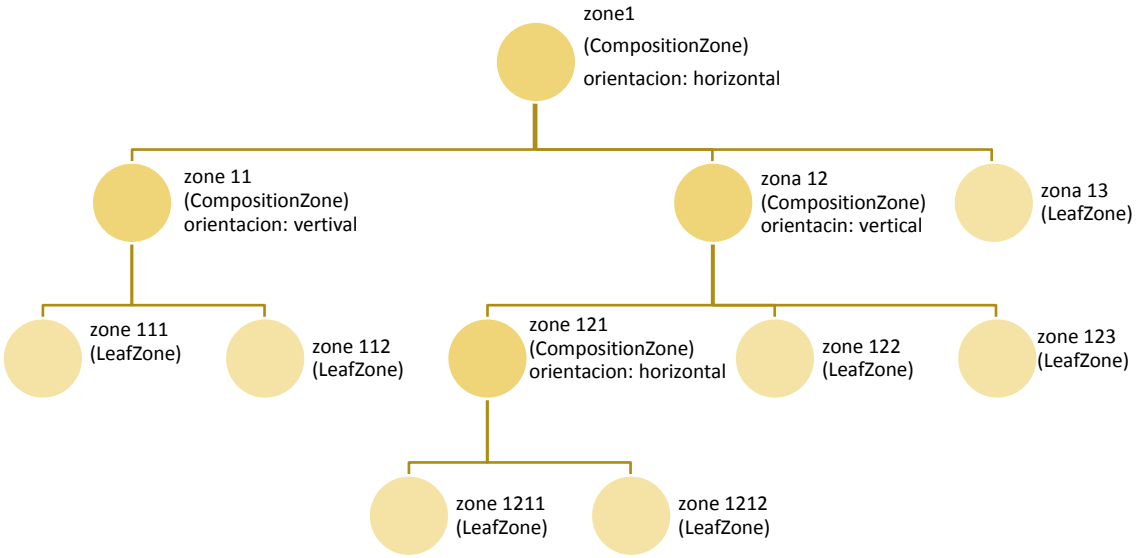


Ilustración 5.5: Árbol de instancias para Layout

Un ArchetypeFieldReference es una referencia a un nodo de tipo ELEMENT (ArchetypeReferencesType) dentro del arquetipo identificado. Éste podrá tener asociado más de un FieldLabel y uno o más Control (uno por cada atributo del DataType en el nodo ELEMENT.value del arquetipo)

Por otra parte, un ArchetypeContainerReference hará referencia a nodos que no sean de tipo ELEMENT. Estos podrán ser de tipo COMPOSITION, OBSERVATION, ITEM_TREE, CLUSTER, INSTRUCTION, ACTION, etc.

Será posible crear dos ArchetypeReference distintos dentro del mismo UITemplate que usen el mismo identificador de arquetipo (ArchetypeReference) pero tengan distintas rutas (ArchetypeReferencePath). No pueden existir dos ArchetypeReference con el mismo ArchetypeReference y una misma ArchetypeReferencePath.

FieldLabel

Permite sobrescribir la etiqueta de un ArchetypeFieldReference. Por defecto se toma como etiqueta el nombre asociado al nodo del arquetipo al que hace referencia el ArchetypeFieldReference. Se debe tener en cuenta que dentro de un arquetipo se definen nombres descriptivos para cada nodo, los cuales pueden estar traducidos a diversos idiomas.

Para cada ArchetypeFieldReference se puede definir más de un FieldLabel, uno por cada lenguaje que se quiera soportar.

Control

Describe como se visualizará un ArchetypeFieldReference dentro de un View (REQTMP04). Tiene restricciones sobre la posición del campo, el tipo de control a mostrar en la IU, el valor por defecto y si el valor es requerido u opcional.

Cada Control tendrá un identificador único (ControlId) dentro del UITemplate.

En el *Anexo C "Tipos de controles"* se puede ver el estudio realizado para definir qué controles son posibles asociar a cada tipo de dato definido en OpenEHR.

Restricciones de modelo

i. *El layout global pertenece a la colección de layouts asociadas al UITemplate dado.*

ii. *Si un ArchetypeReference tiene ocurrencia máxima 0, la cantidad de veces que el campo asociado puede visualizarse en la IU es ilimitada.*

iii. *La ocurrencia máxima de un ArchetypeReference debe ser mayor o igual que la ocurrencia mínima, exceptuando el caso en que la ocurrencia máxima tome el valor 0.*

iv. *Si un ArchetypeReference tiene definido un valor para OccurrenceMin, dicho valor no puede ser menor que el mínimo definido en el nodo del arquetipo referenciado por ArchetypeRefernce.Path (siempre que exista este valor). Análogamente para el caso de OccurrenceMax.*

v. *La ruta (path) de un ArchetypeContainerReference debe ser prefijo de todas las rutas de los ArchetypeReference hijos.*

vi. *Los ArchetypeFieldReference que pertenezcan a un View, harán referencia a elementos LeafZone que formen parte del Layout asociado a dicho View.*

vii. *Todos los ArchetypeFieldReference que formen parte de un ArchetypeContainerReference con multiplicidad (OccurrenceMax != 1) deben pertenecer a la misma LeafZone.*

Representación del modelo y modelo validador

Se seleccionó XML para representar las instancias del modelo. La elección se basó en sus características de sencillez, extensibilidad, interoperabilidad, y capacidad de poder ser entendido a simple vista por personas con perfil técnico. Además mantiene una concordancia con EHRGen, dado que también representa en XML las definiciones de templates.

De la mano de esta condición, una vez definido el modelo conceptual, fue necesario plantear un modelo que fuera capaz de validar las diferentes instancias de UITemplate. Es decir, un modelo a través del cual se pudiese describir: estructura, elementos, relaciones y restricciones. Es así que, en base a XML Schema (31), se creó el modelo semántico que puede verse en el *Anexo D "UITemplate: XML Schema*"

Para crear el modelo XML Schema (XSD) se utilizó Eclipse Modeling Framework Project (EMF) (32). Con esta herramienta, una vez diseñado el modelo conceptual (diagrama UML), se autogeneró un modelo en XSD. Dicho esquema lo tomamos como base para crear nuestro modelo final.

5.1.2 Construcción del código de prueba

Objetivos principales:
Crear el prototipo de lo que se desea generar, escribiendo a mano el código deseado. Esto se llamará código de prueba.

Para generar el código de prueba de una interfaz de usuario, hubo que definir qué se quiere mostrar y con qué tecnología se quiere trabajar.

En cuanto a lo primero, y teniendo en cuenta lo analizado en la sección anterior, se definió que la IU que representará a una instancia de UITemplate constará de:

- una barra de navegación principal desde la cual se podrán acceder a los diferentes View definidos.
- una sección en donde mostrar el contenido del View seleccionado.

Luego, cada uno de los View deberá contener:

- un título.
- una tabla que se corresponderá con la disposición definida en el Layout.
- etiquetas y campos para el ingreso de datos.
- un botón para el envío de los datos ingresados.

Con respecto a la tecnología, se eligió HTML5 para desarrollar el primer código de prueba. Su facilidad de uso, los resultados obtenidos, y la buena recepción de sus aplicaciones web entre los usuarios, fueron algunos de los puntos que se tuvieron en cuenta para tomar esta decisión.

La *Ilustración 5.6* muestra una primera aproximación a la estructura definida:

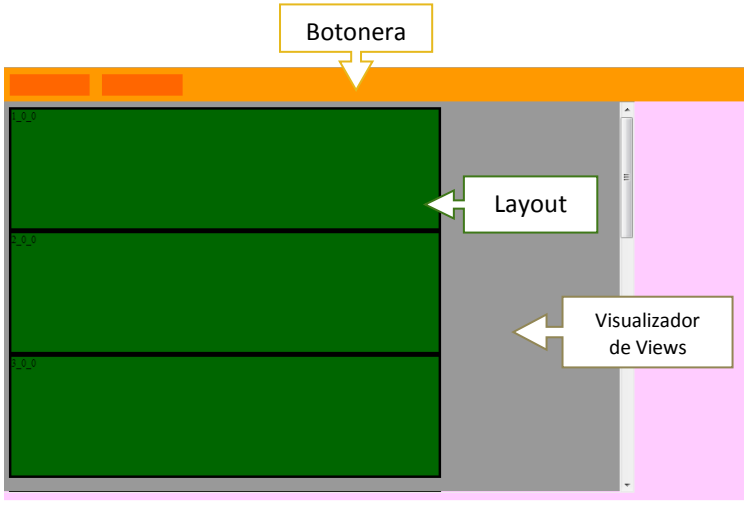


Ilustración 5.6: Estructura básica de IU

La dificultad que se encontró en esta etapa fue poder generar una tabla con la estructura de un Layout, de manera que cada celda correspondiente a una determinada fila tuviera el mismo tamaño que las otras celdas de la fila. El requisito fue no tener que recurrir a cálculos sobre el ancho total de la tabla y la cantidad de celdas de cada fila.

De esta manera, se utilizó una nueva característica de CSS2, en dónde una etiqueta *div* se puede configurar para que se visualice en forma de tabla mediante el

atributo *display* (33). A partir de esta solución, se pudieron generar desde Layouts simples a más complejos como se ve en la *Ilustración 5.7*.

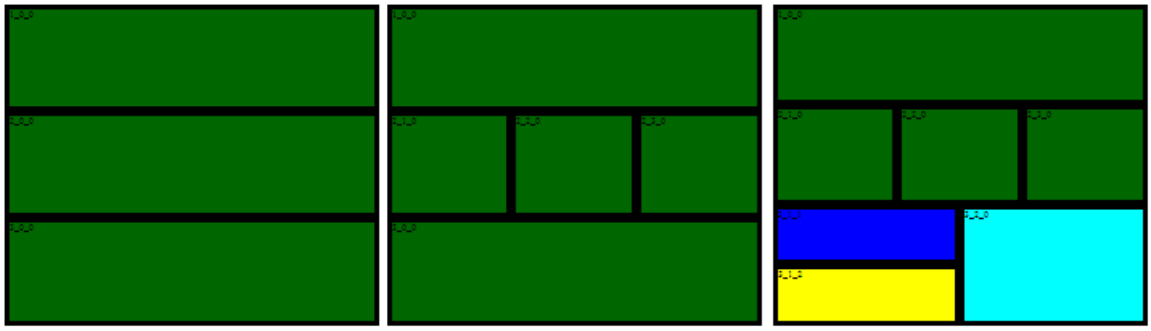


Ilustración 5.7: Estructura de Layout

Finalmente se llegó a un código de prueba, el cual representa la IU que se puede observar en la *Ilustración 5.8*.

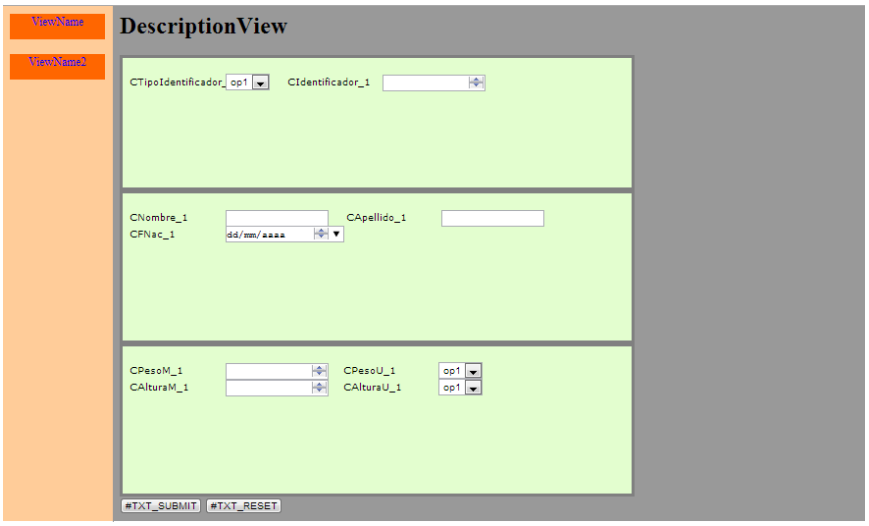


Ilustración 5.8: IU generada a partir del código de prueba

5.1.3 Desarrollo del analizador de datos de entrada

Objetivos principales
Desarrollar un componente capaz de procesar la metadata serializable que tendrá como entrada el generador.

En este proyecto se decidió utilizar Groovy (34) (*ver sección 2.3.3 "Groovy"),* beneficiándonos su potencia y su sencilla curva de aprendizaje, para llevar a cabo el proceso de implementación. Cabe destacar, que si bien GeneXus comparte estas mismas características, no era una opción para desarrollar la aplicación por no ser open source. Otra de las razones que llevó a utilizar Groovy fue que EHRGen, aplicación que define el modelo de template a mejorar, está construido en el mismo lenguaje. Esto facilita la transmisión de conocimiento y reutilización de código, en especial sobre el manejo de arquetipos.

Para procesar y gestionar en memoria las instancias UITemplate expresadas en XML se evaluaron dos alternativas. La primera se centraba en recorrer el XML mediante funciones provistas por alguna librería de Java, como lo es JDOM (35), y almacenar los datos en una estructura de clases asociada. Dicha estructura se generaría de forma automática en base a la definición del modelo XML Schema con la herramienta EMF. Esto nos permitiría navegar sobre instancias de dichas clases tal como si se recorriéramos una definición de UITemplate.

La segunda alternativa se basaba en la utilización de la API de Groovy, XMLSlurper (36). Esta API permitiría parsear un archivo XML, generando una estructura jerárquica o de árbol. Dicha estructura podría ser analizada mediante expresiones XPath (*XML Path Language)* (37), facilitando la selección y referencia a textos, elementos, atributos y cualquier otra información contenida dentro de un XML.

En conclusión, para el procesamiento de definiciones de UITtemplate, se optó por la segunda alternativa dado que no era necesario mantener una estructura complementaria de clases. La sintaxis para recorrer el XML es fácil e intuitiva debido a la potencialidad de las funciones que brinda la API. En la sección *5.1.1, Diseño del generador* se puede ver cómo se integra XMLSlurper al resultado final de implementación.

5.1.4 Desarrollo del mapeador a partir del código de prueba

Objetivos principales
Definir un componente capaz de realizar la correspondencia entre instancias de UITemplate y una tecnología de salida específica

Partiendo del código de prueba se identificaron diferentes secciones que agrupaban una misma área de conocimiento (*Ilustración 5.9*), haciendo que cada uno de los componentes de un una instancia de UITemplate se correspondieran con el código fuente específico. A modo de ejemplo, en la *Ilustración 5.10*, se puede ver un sencillo código en HTML donde se marcan las diferentes secciones encontradas.

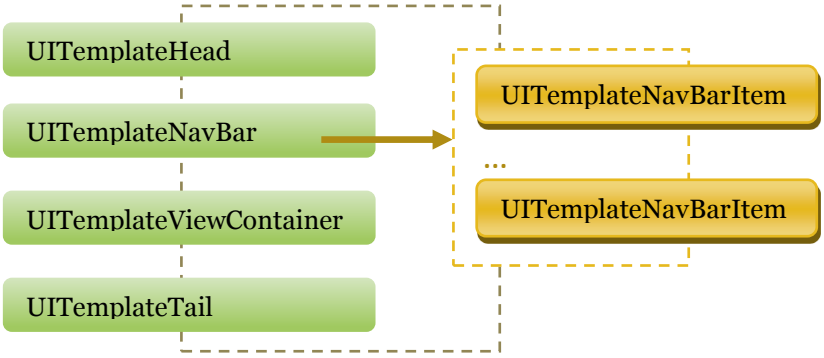


Ilustración 5.9: Secciones de identificadas en el código de prueba

UITemplateHead

```
<html>
<head>
<title>Description 1</title>
<link href="uitemplate_css.css" rel="stylesheet" type="text/css" />
</head>
<body>
```

UITemplateNavBar

```
<div id="navbarLeft">
<a target="loader" href="view_1.html"><div class="navbar_item">ViewName</div></a>
<a target="loader" href="view_2.html"><div class="navbar_item">ViewName2</div></a>
<div id="main">
```

UITemplateViewContainer

```
<div id="view_containerLeft">
<iframe id="loader" src="view_1.html" width="100%" height="100%"></iframe>
</div>
<div id="main">
</div>
```

UITemplateTail

```
</body>
</html>
```

Ilustración 5.10: Código HTML por secciones

5.1.4.1 Definiciones de mapeo

El objetivo durante esta etapa fue definir el código fuente en la tecnología de salida para cada una de las secciones identificadas a partir del código de prueba y los sucesivos cambios que se realizaron para aumentar la complejidad del diseño. Estas secciones pasarán a tomar el nombre de variables de sustitución.

Una variable de sustitución permite establecer la estructura configurable de una interfaz de usuario. Partiendo de ella es posible definir el código fuente en una tecnología específica para una sección en particular de la IU. El conjunto de todas las variables de sustitución conforman el código final de la IU. Opcionalmente, en una variable de sustitución es posible incluir variables a las que denominaremos globales.

A través de una variable global se pueden referenciar valores externos definidos en la instancia UITemplate o en los arquetipos asociados a la misma. Las correspondencias entre cada una de las variables globales y sus respectivos valores externos fueron especificadas e incluidas en el generador.

A modo de ejemplo podemos tomar el caso de la variable de sustitución Fieldccount1Head, utilizada para describir el cabezal de un control tipo count. En uno de los archivos de mapeo creados, el código fuente para dicha variable se define de la siguiente forma:

```
Fieldccount1Head=
        <input class="form" type="number"
        Id="#VIEW_MODE#_#FIELD_CONTROLID#_o#FIELD_OCCURRENCEINDEX#"
        name="#FIELD_CONTROLID#_o#FIELD_OCCURRENCEINDEX#"
        min="#FIELD_MINVALUE#"
        max="#FIELD_MAXVALUE#"
        value="#FIELD_DEFAULTVALUE#"
        #REP_#FIELD_MANDATORY##required="required"#REP#>
```

Las variables globales utilizadas cumplen con la sintaxis ***#<NOMBRE_VARIABLE>#***, ejemplo #VIEW_MODE#, #FIELD_MINVALUE#, etc. A su vez, la definición de variable global, puede contener distintos valores que se representarán como ***var.*** En el ejemplo la variable global #REP_num#$code$#REP# tiene la finalidad de repetir num veces el código $code$.

En la IU generada el código asociado a la variable de sustitución ser verá como

```
<input class="form" type="number" id="edit_c2_01" name="c2_01" min="1" max="3" value="1" >
```

Es así que, en el archivo de mapeo se listarán todas las variables de sustitución que el generador de código será capaz de procesar. En los esquemas representados en la *Ilustración 5.11* y la *Ilustración 5.12* se pueden ver cómo estas variables se agrupan para darle forma al código.

El *Anexo E "Variables de sustitución"* describe cada una de las variables de sustitución y globales definidas en el generador.

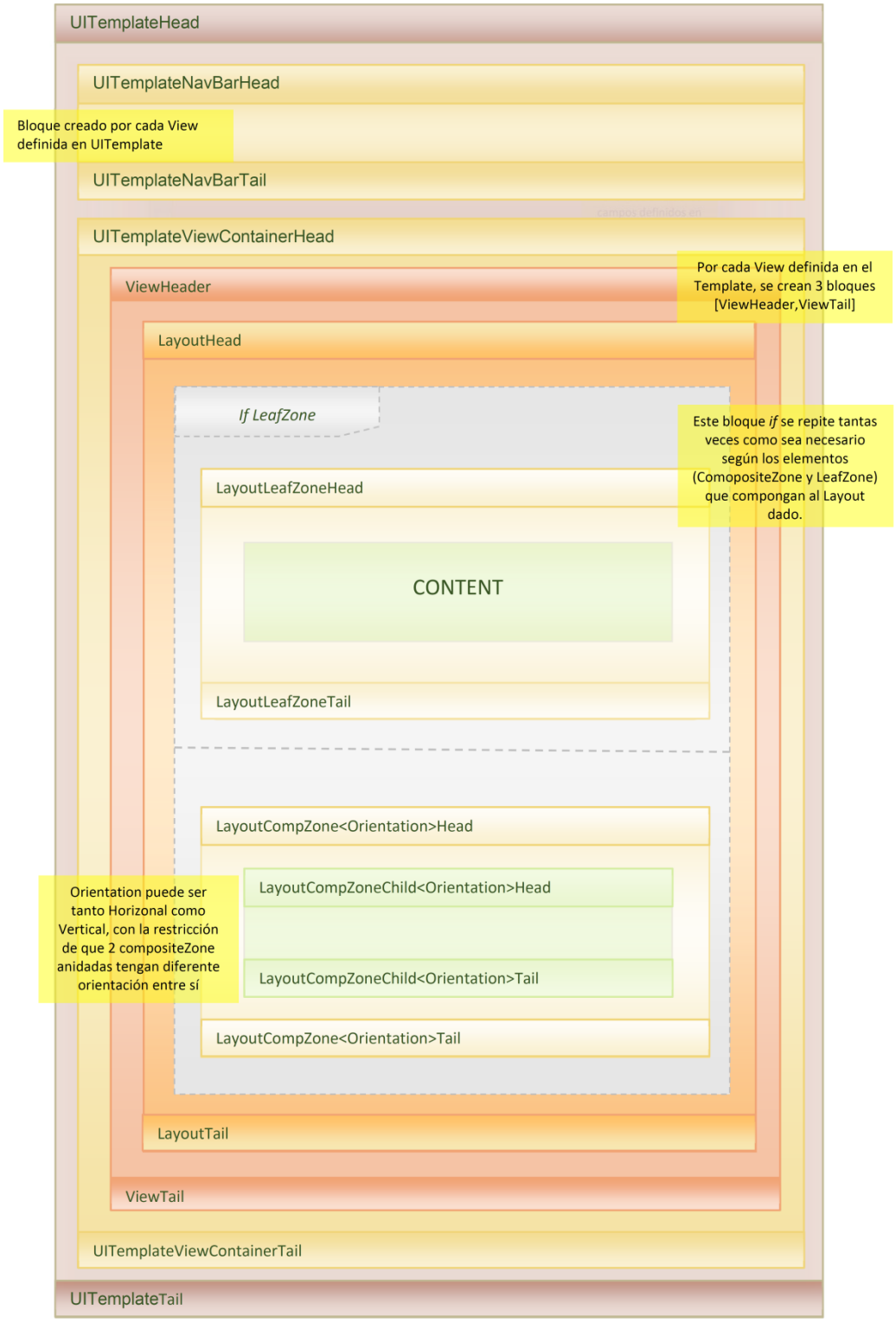


Ilustración 5.11: Plantilla UITemplate

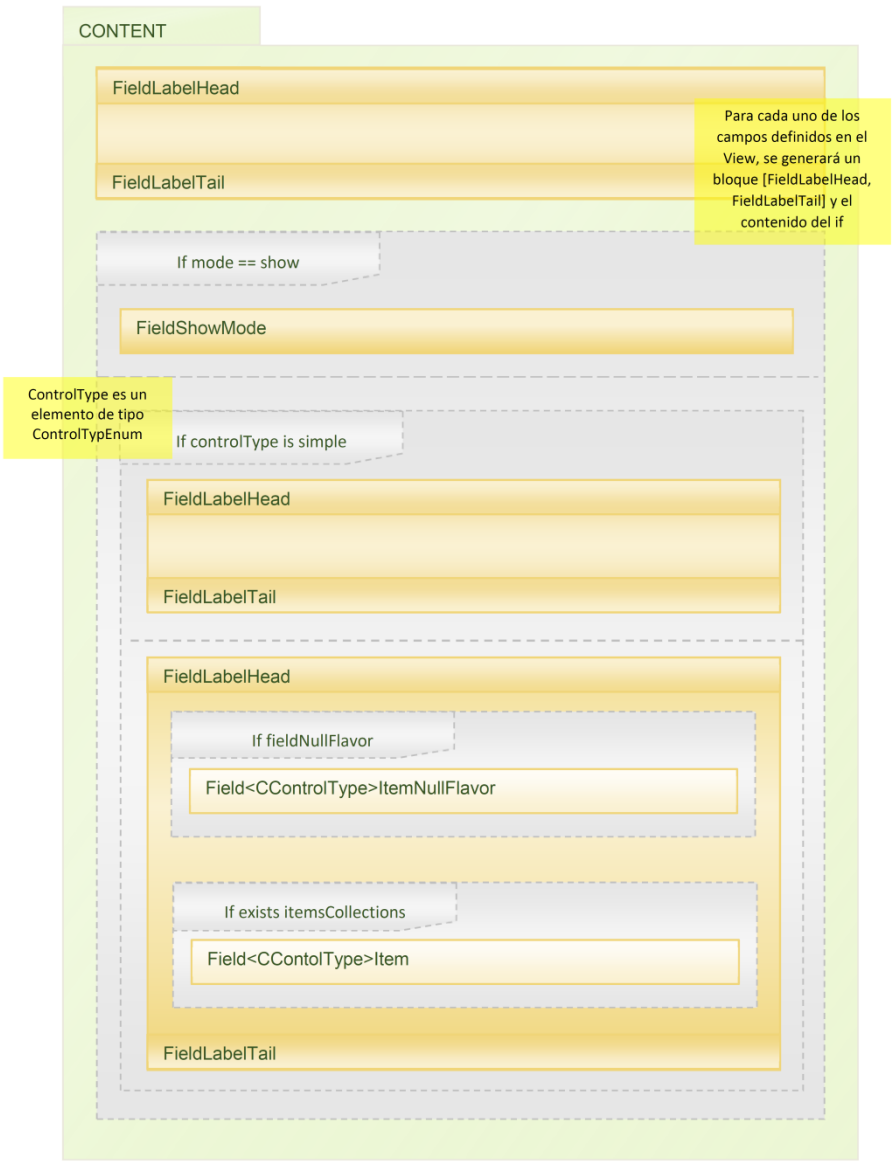


Ilustración 5.12: Plantilla UITemplate - contents

5.1.5 Desarrollo del generador del código de salida

Objetivos principales:
Ensamblar los diferentes componentes del generador definidos en secciones anteriores, tal que dada una instancia de UITemplate, se genere la IU correspondiente en la tecnología de salida específica.

Una vez definido el analizador de datos de entrada y el mapeador se realizó la formalización de los componentes responsables de cada una de estas tareas. Así, se hallaron otros elementos complementarios que también se integraron con el desarrollo final de generador.

5.1.5.1 Componentes del generador

Para desarrollar el generador de interfaces, se definieron un conjunto de componentes básicos: Loader, Parser, Mapper, ArchetypesManager y Generator. En la *Ilustración 5.13* se puede ver el diseño y las relaciones entre los componentes que se describen a continuación.

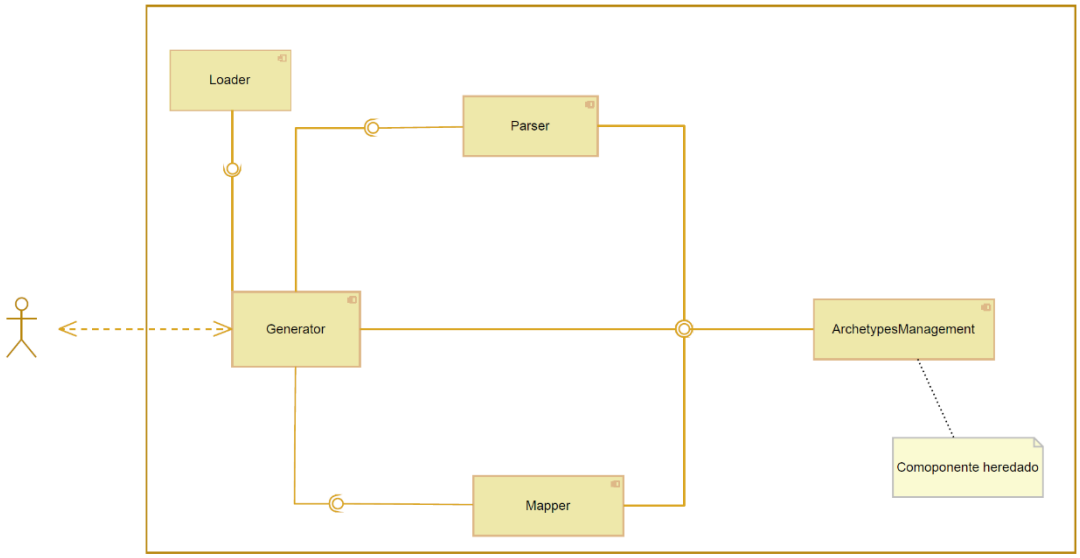


Ilustración 5.13: Diseño de componentes del generador de interfaces

Generator

Punto inicial donde se da comienzo al proceso de generación de interfaces. Tiene interacción con el usuario de la aplicación, quien es el encargado de configurar las propiedades del generador de interfaces (*ver Ilustración 5.17*) y de invocar la aplicación a través de este componente para que se generen las IU en una tecnología final dada.

El Generador será responsable de cargar en memoria la instancia del UITemplate y archivos de configuración a través de Loader. Luego se comunicará con el Parser para validar la información a procesar. Una vez validada la instancia de UITemplate, se comunicará con el Mapper para iniciar el proceso de generación propiamente dicho.

Loader

Componente responsable de realizar las lecturas de archivos cualesquiera sea su origen. Actualmente soporta únicamente la lectura desde disco, pero puede ser fácilmente extensible a otros tipos de carga (ej. vía ssh, web services, etc.). Cada nuevo tipo de lectura se deberá desarrollar en una clase que implemente la interfaz definida.

Parser

Agrupa el conjunto de controles que se realizan sobre una definición de UITemplate, antes de que la misma sea procesada por el Mapper. Los controles que se llevan a cabo son tanto a nivel de estructura como de restricciones de dominio.

Las tareas de validación de la estructura y de la sintaxis se realizan contra el esquema definido (*ver Anexo D "UITemplate: XML Schema"*).

Además brinda funciones para recuperar las rutas de nodos de arquetipos a partir de identificadores presentes en las interfaces de usuario generadas. Esto es útil a nivel de gestión de las interfaces en las aplicaciones integradoras (*ver 6.2.3 Aplicación integradora*)

ArchetypesManagament

ArchetyepesManagement es un componente formado por una clase (ArchetypeManager.groovy) heredada de la aplicación EHRGen (2). Las tareas que permite llevar a cabo son las de buscar y cargar arquetipos desde un repositorio a memoria. Dado que los arquetipos son accedidos frecuentemente por tres de los componentes que se presentan, es de gran utilidad que esta clase mantenga los arquetipos cargados en la caché, para mejorar así los tiempos de acceso a los mismos.

Mapper

Lleva a cabo los mapeos necesarios entre la instancia de UITemplate y la interfaz de usuario esperada en una tecnología seleccionada. Todos los mapeos se resuelven de forma automática basándose en archivos de configuración que definen estas asociaciones, tal como se vio en el capítulo anterior. Un usuario con perfil técnico deberá ser capaz de configurar, según la tecnología, las variables de sustitución.

Interacción de componentes

Con el fin de comprender cómo se comporta el generador de interfaces, una vez que éste sea ejecutado por un usuario, se presenta el diagrama de secuencia de la *Ilustración 5.14*. Se muestra un caso positivo de acción, en la que se parte de la llamada inicial al generador, y se finaliza con la creación de todos los componentes de interfaz de usuario en la tecnología final seleccionada.

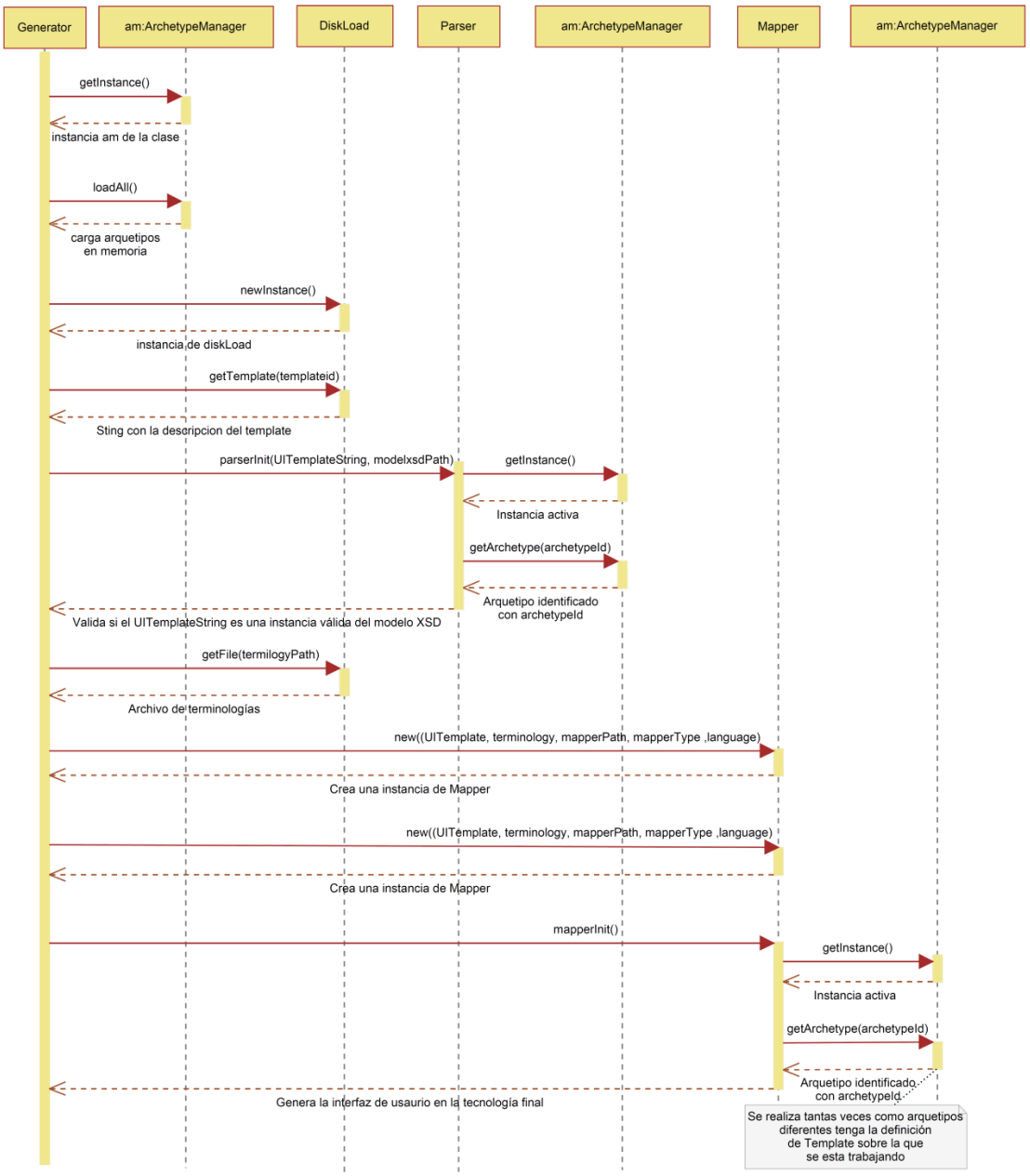


Ilustración 5.14: Diagrama de secuencia del diseñador de interfaces

5.2 Evolucionando el generador de interfaces de usuario

Una vez validado el cumplimiento de los requisitos mínimos del generador, se decidió ampliar sus funcionalidades. En esta instancia se incorporó el concepto de multiplicidad y la generación para una nueva tecnología.

5.2.1 Multiplicidad

Cada nodo de un arquetipo, tanto simple como compuesto, maneja el concepto de ocurrencia mínima y máxima. Si la ocurrencia mínima es “m”, dicho nodo tendrá “m” ocurrencias dentro de una IU generada, y el ingreso de datos para los campos asociados a dicho nodo será obligatorio. Por otro lado, si la ocurrencia máxima es "M", significará que dicho nodo puede tener hasta "M" ocurrencias dentro de la IU generada. Para ello, se deberá proporcionar los medios por los cuales el usuario pueda decidir en tiempo de ejecución, si desea “repetir” la ocurrencia de dicho nodo en la IU. Además en el UITemplate se pueden restringir los valores de ocurrencia máxima y mínima definidos en el arquetipo a partir de los atributos OccurrenceMin y OccurrenceMax de la entidad ArchetypeReference.

Dadas estas condiciones se consideró que el manejo de la multiplicidad era un tema importante a resolver. Para ello se destinaron dos grupos de variables de sustitución para permitir configurar el código necesario para incluir dicha funcionalidad tanto a nivel del campo (field) como de componente (container) (FieldMultiplicityHead, FieldMultiplicityHead, FieldMultiplicityHead). En la *Ilustración 5.15* se puede ver cómo se resuelve tanto la ocurrencia mínima como máxima en HTML5 mediante la utilización de JavaScript. A su vez, la *Ilustración 5.16* muestra una resolución hallada para el caso de componente.

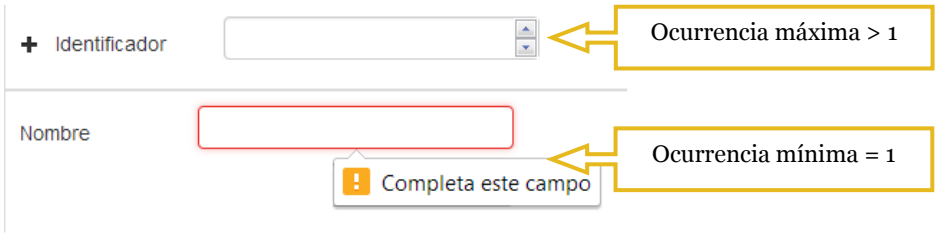


Ilustración 5.15: Multiplicidad de campo

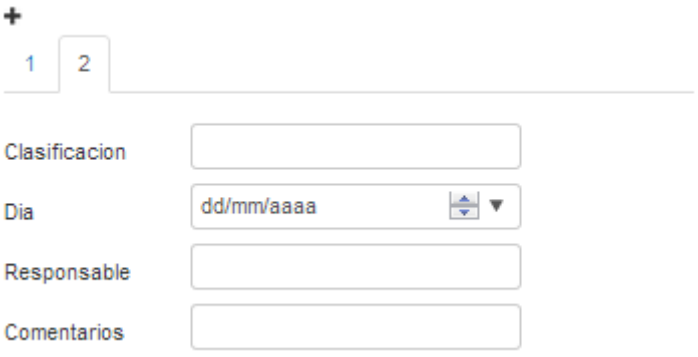


Ilustración 5.16: Multiplicidad de componente

En el UITemplate se pueden restringir los valores de ocurrencia máxima y mínima definidos en el arquetipo a partir de los atributos OccurrenceMin y OccurrenceMax de la entidad ArchetypeReference. El ingreso de dichos valores es opcional, si no se definen se usarán los predeterminados en el arquetipo. En el *Anexo F "Manual de usuario técnico",* se explica cómo se resuelve a nivel de implementación esta funcionalidad.

5.2.2 Generación de IU para una nueva tecnología

Culminada la etapa de generación de código para HTML5, nuestro objetivo se enfocó en incorporar el mapeo a otra tecnología. Teniendo en cuenta que uno de los requisitos del cliente era poder generar IU para aplicaciones de escritorio, se selección XAML como la nueva tecnología para cumplir dichos objetivos.

Para realizar este proceso se volvió a iterar sobre el ciclo de generación de IU partiendo desde la etapa *5.1.2 "Construcción del código de prueba"*. Esto nos sirvió para verificar que el diseño del generador construido fuese lo suficientemente sólido y desacoplado como para incluir el mapeo en la tecnología elegida.

Como resultado final los mapeos definidos para ambas tecnologías, fueron:

- *html.map*

Genera las interfaces de usuario asociadas al UITemplate y a sus Views en diferentes archivos .html. En la IU asociada al UITemplate se utiliza iframe para desplegar el contenido del View seleccionado.

- *html2.map*

La estructura del UITemplate y Views se generan en el mismo archivo de salida. Utiliza JavaScript para controlar dinámicamente la visualización de una u otra View integrada en la IU.

- *htmlPHP.map*

Genera solo el código de las Views para ser acoplado a la aplicación integradora.

- *xaml.map*

Genera el código XAML para ser compilado con Visual Studio, y poder ser ejecutado como una aplicación de escritorio.

- *xamlWeb.map*

Se trata del mismo formato de visualización que el xaml.map pero para aplicaciones web. Para poder ser ejecutado en un navegador, debe ser compilado previamente en Visual Studio.

- *xamlWebIE.map*

El código generado no necesita ser compilado en Visual Studio, y puede ser ejecutado directamente en Internet Explorer. Dicho código tienen integrada la definición de estilos, no considera eventos asociados a las acciones de multiplicidad de código y no utiliza el elemento datepicker para desplegar datos de tipo date.

5.3 Diseñador de interfaces de usuario

El diseñador de interfaces de usuario logrado, es una herramienta que toma como insumo los modelos de información y restricciones. Generará formatos computables que definan el modelo de interfaz de usuario (modelo UITemplate). Se trata de una aplicación que corre en línea de comando, proporcionando una manera interactiva de crear una instancia de UITemplate para posteriormente ser consumida por el generador de IU.

El principal objetivo de la herramienta es proporcionar al usuario una manera amigable y genérica de ingresar los datos para definir el UITemplate. Puede no existir un conocimiento previo acerca del formato utilizado para almacenar los datos, es decir, no es necesario conocer en profundidad el modelo (*ver Ilustración 5.2: Modelo de diseño del template*) para interactuar con la aplicación. La creación de UITemplates a partir del diseñador, garantiza que dichas instancias cumplan con la especificación del modelo, siendo insumos válidos para ser procesados posteriormente por un generador.

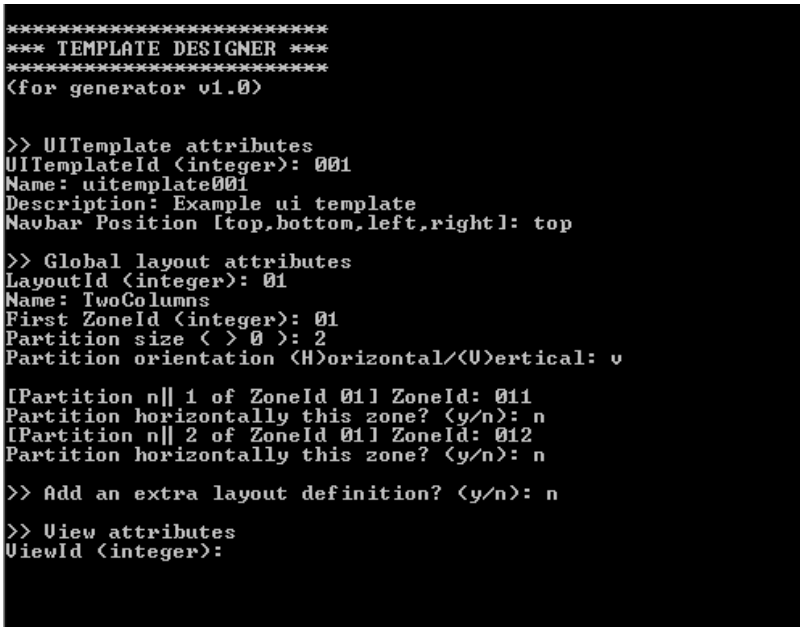


Ilustración 5.17: Ejemplo de ejecución del diseñador

Una vez en ejecución, se guía al usuario mediante sencillos pasos para lograr una correcta definición de un UITemplate. A grandes rasgos, se requerirá el ingreso de datos para los siguientes componentes:

- UITemplate: atributos
- Layouts: atributos
  - Composición de zonas (Zones): atributos
- Views: atributos
  - Arquetipos (Container) : atributos y propiedades
  - Arquetipos (Field): atributos y propiedades
    - Controls: atributos

A pesar de la simplicidad de esta herramienta, nos fue muy útil para la generación de definiciones de UITemplate a ser utilizadas como insumos en el generador.

6 Verificación y validación

En este capítulo se pretenden mostrar los aspectos más relevantes en relación a la verificación y validación de los resultados obtenidos a lo largo del proyecto.

Cada una de las etapas que integran la solución se construyeron siguiendo un proceso iterativo incremental, desarrollado en base a un proceso de sucesivos refinamientos sobre la definición inicial. En cada iteración se contó con la validación del cliente, mitigando así el riesgo en etapas posteriores y logrando que el proceso se desarrollara con éxito.

6.1 Verificación

El proceso de verificación se llevó a cabo controlando los hitos relevantes del proyecto, verificando que el trabajo realizado en cada etapa se correspondiera con los requerimientos iniciales. A continuación se destacan los principales hechos que se tuvieron en cuenta para esta etapa:

- Relevamiento de requerimientos.
- Definición de un nuevo modelo de UITemplate.
- Validación del modelo de template con diferentes definiciones de UITemplate.
- Desarrollo del generador de interfaces de usuario en HTML5.
- Definición de diferentes configuraciones de mapeo para HTML5.
- Testeo y correcciones del generador de IU.
- Ajustes del generador para generar IU en XAML.
- Testeo y correcciones del generador de IU.
- Definición de diferentes configuraciones de mapeo para XAML.
- Desarrollo del diseñador de interfaces de usuario.
- Integración de las interfaces generadas en una aplicación para dicho propósito.

6.2 Validación

La validación del prototipo final se dividió en dos etapas. Inicialmente se validó el diseñador de interfaces de usuario, creando diferentes instancias de UITemplate. Estas definiciones luego fueron utilizadas en una segunda instancia, en donde se realizó la validación del generador de IU tanto para HTML5 como para XAML.

6.2.1 Diseñador de interfaces

El diseñador creado permite generar definiciones de UITemplate en base al modelo de template planteado, accediendo también a las estructura de arquetipos. Una definición de UITemplate creada con esta herramienta, garantiza que la misma sea una instancia válida con respecto al modelo de UITemplate definido, y que las paths de arquetipos estén correctamente referenciadas. Eso último se debe a que se realizan los chequeos necesarios de existencia de los nodos que se integran a cada definición. En el

Anexo G "Manual de Usuario Funcional" se puede ver un ejemplo de ejecución del diseñador de interfaces.

6.2.2 Generador de interfaces de usuario

En la herramienta presentada es posible integrar cualquier arquetipo plano que contenga alguno de los siguientes tipos de datos de OpenEHR: DV_TEXT, DV_CODEDTEXT, DV_COUNT, DV_QUANTITY, DV_DATETIME. En el manual de usuario técnico presentado en el *Anexo F "Manual de usuario técnico"* se especifica como ampliar el generador para dar soporte al resto de los tipos de datos de OpenEHR.

A partir de un archivo de definición de interfaz y un archivo de mapeo, se logra la generación automática de IU, tanto en HTML5 para aplicaciones web (*ver Ilustración 6.1),* como en XAML para aplicaciones de escritorio (*ver Ilustración 6.2*). Como se puede ver en ambas imágenes, se partió de la misma definición de UITemplate pero, modificando el archivo de mapeo se logró tener la misma interfaz en cuanto a contenido para ambas tecnologías.

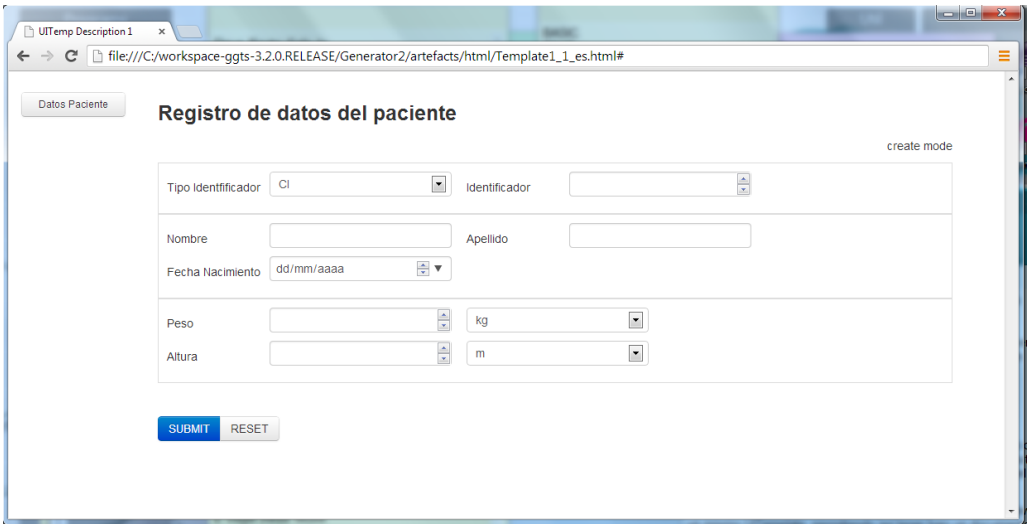


Ilustración 6.1: Interfaz de usuario generada en HTML5

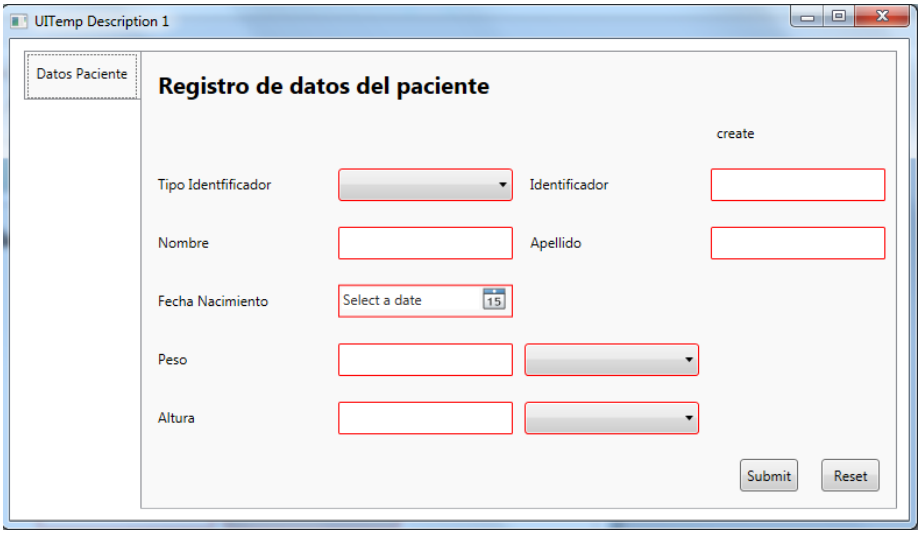


Ilustración 6.2: Interfaz de usuario generada en XAML (desktop)

```
<!-- Apellido -->
        <ArchetypeFieldReference
        ArchetypeReference="openEHR-EHR-COMPOSITION.arquetipo_prueba_001.v1"
        Path="/context/other_context[at0001]/items[at0014]/items[at0010]"
        Type="ELEMENT"
        OccurrenceMin="1"
        OccurrenceMax="1"
        ZoneId="12"
        >
        <Control
             ControlId="tmp0000004"
             Name="CApellido_1"
             Type="CText"
             TypeCode="1"
             NullFlavor="0"
             Position="2"
             Skip="0"
             AttributePath="/context/other_context[at0001]/items[at0014]/items[at0010]/value"
        />
        </ArchetypeFieldReference>
```

Ilustración 6.3: Definición de campo "Apellido" en el UITemplate

La decisión de cómo se visualiza un campo está dada por el o los usuarios destinados al diseño de la IU. A modo de ejemplo, en las imágenes anteriores, los campos “Nombre” y “Apellido” están alineados. Esto se debe a que en su UITemplate (*ver Ilustración 6.3*), el atributo Skip de la definición del Control asociado tiene el valor 0. En cambio si se modifica dicho valor a 1, y se vuelve a ejecutar el generador con dichos cambios, se obtendrá una IU como la de la *Ilustración 6.4,* en donde los campos “Nombre” y “Apellido” ya no se hallan en la misma línea.

Por otra parte también soporta la definición de diferentes tipos de controles. Solo basta con definirlos en los archivos de mapeo correspondientes y referenciarlos desde la definición de UITemplate (*ver Anexo F “Manual de usuario técnico”*).

Para validar las instancias de interfaces generadas en HTML5 se utilizó el Markup Validation Service (38). Gracias a este validador, se pudieron hallar errores de sintaxis que fueron corregidos en la especificación del código HTML5 de las variables de sustitución.

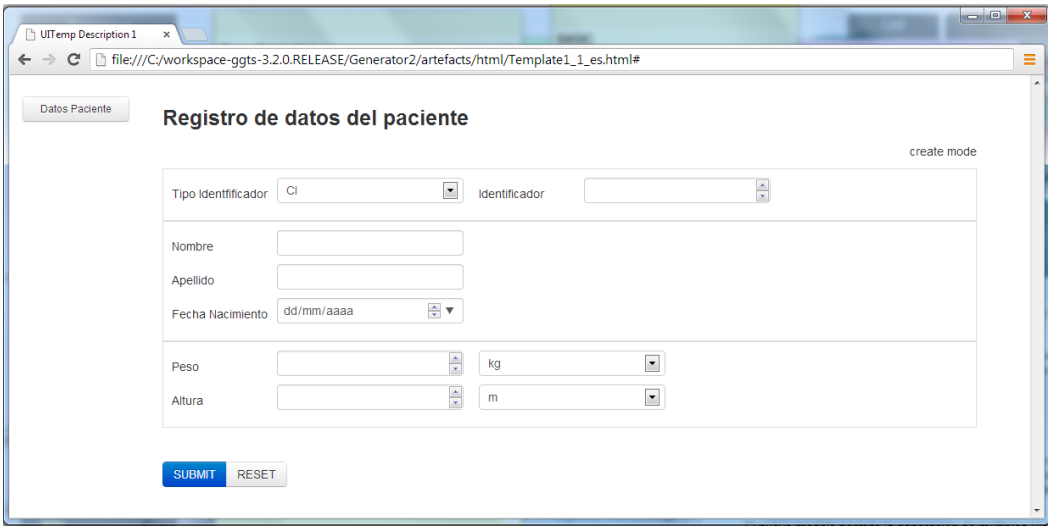


Ilustración 6.4: Interfaz de usuario generada en HTML5

6.2.3 Aplicación integradora

Tanto el diseñador como el generador de interfaces tienen como fin agilizar los tiempos de prototipación y generación de interfaces de usuario para uso clínico. En la *Ilustración 6.5* se puede ver cómo es posible integrar en el proceso de diseño y especificación de una aplicación las herramientas creadas.

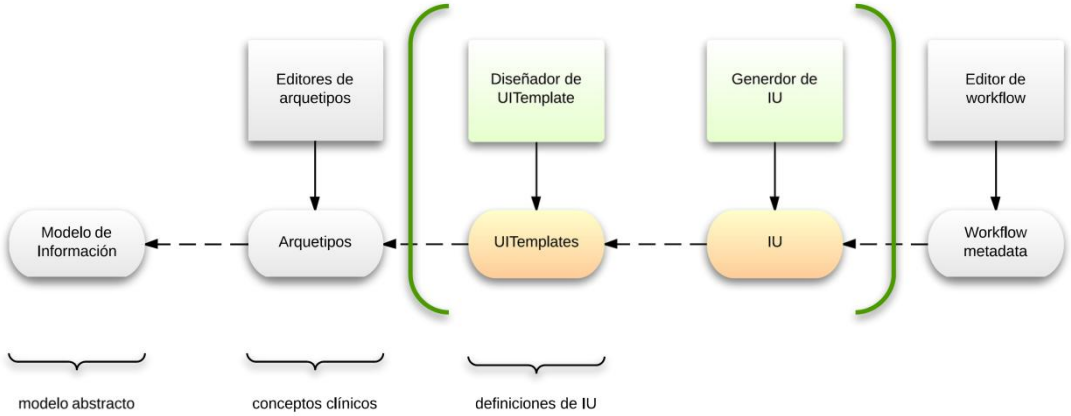


Ilustración 6.5: Integración de herramientas: tiempo de diseño y especificación de artefactos (metadata)

En la *Ilustración 6.6* se describe el flujo normal de interacción del usuario final con una aplicación que hace uso de las interfaces diseñadas y generadas automáticamente.

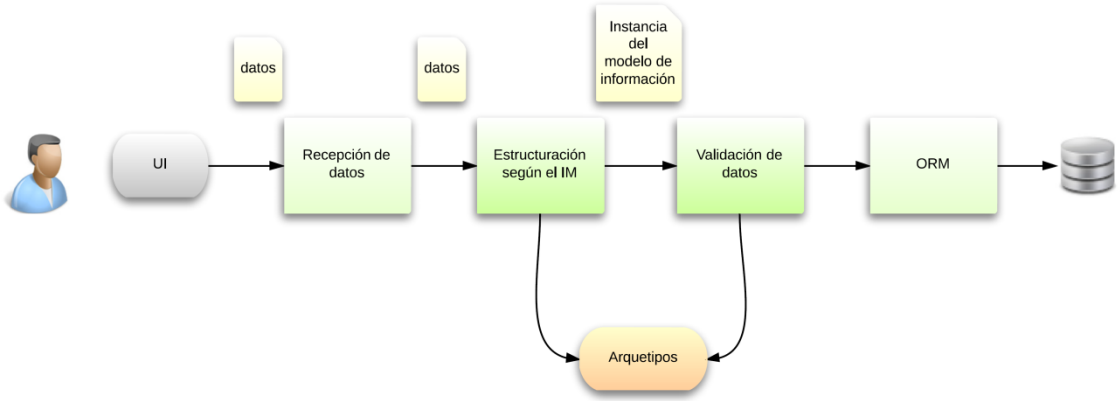


Ilustración 6.6: Diagrama de uso de registro clínico electrónico

Considerando el caso de uso típico, se destacan los siguientes pasos:

i. El usuario ingresa a la aplicación para ingresar los datos en un formulario.
ii. El usuario confirma el ingreso de datos.
iii. DATA BINDER

iv. Se controlan los datos ingresados. La validación se realiza teniendo en cuenta las restricciones de los nodos de los arquetipos asociados a cada campo, y si la validación fue correcta,
v. Los datos se almacenan físicamente para poder ser recuperados más adelante para su consulta y o modificación.

Para que la herramienta pueda ejecutar el paso iv, necesita contar con la información que le permitan asociar cada campo de la IU con un nodo específico de un arquetipo. Esto es posible debido a que cada elemento de la IU generada tiene un identificador único en el UITemplate. De esta forma la aplicación puede, dados una UITemplate y un identificador, recuperar el nombre del arquetipo y la ruta del nodo (path). Actualmente las funciones necesarias para realizar esta tarea se hallan bajo la clase Parser (*ver 5.1.5.1 "Componentes del generador").*

Se utilizó una aplicación encargada de integrar interfaces de usuario generadas automáticamente con el fin de simular casos de uso típico. Dicha herramienta toma los datos ingresados y los almacena en archivos JSON. Siempre que se quiera, dado un formulario, es posible recuperar y modificar los datos persistidos. En la *Ilustración 6.7*, *Ilustración 6.8*, *Ilustración 6.9* e *Ilustración 6.10*, se presentan capturas de pantalla de dicha aplicación, mostrando los casos de uso identificados previamente.

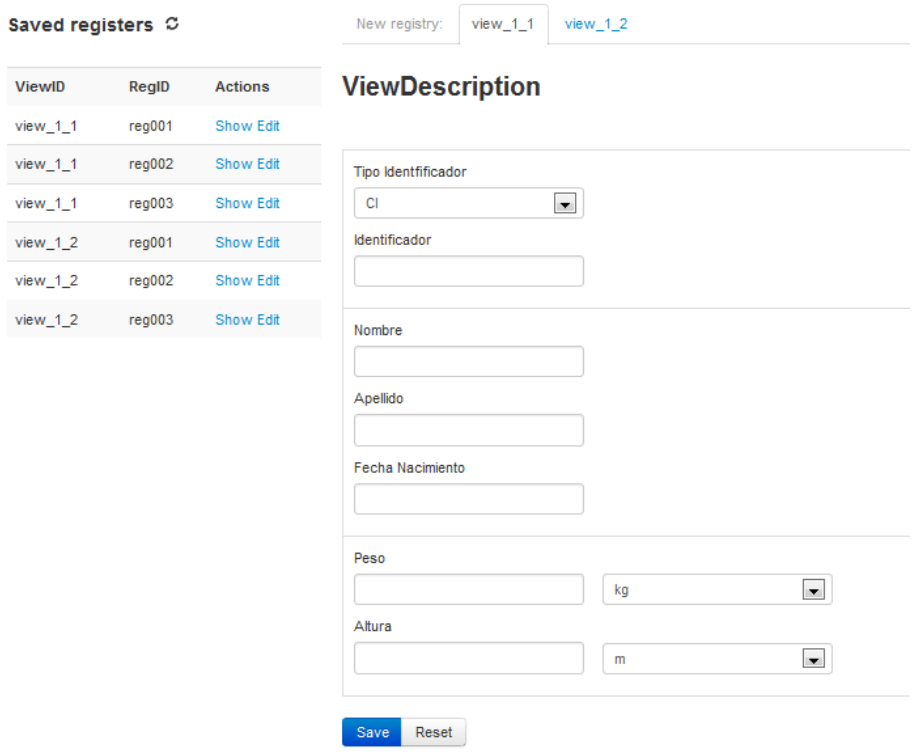


Ilustración 6.7: Ingreso de datos

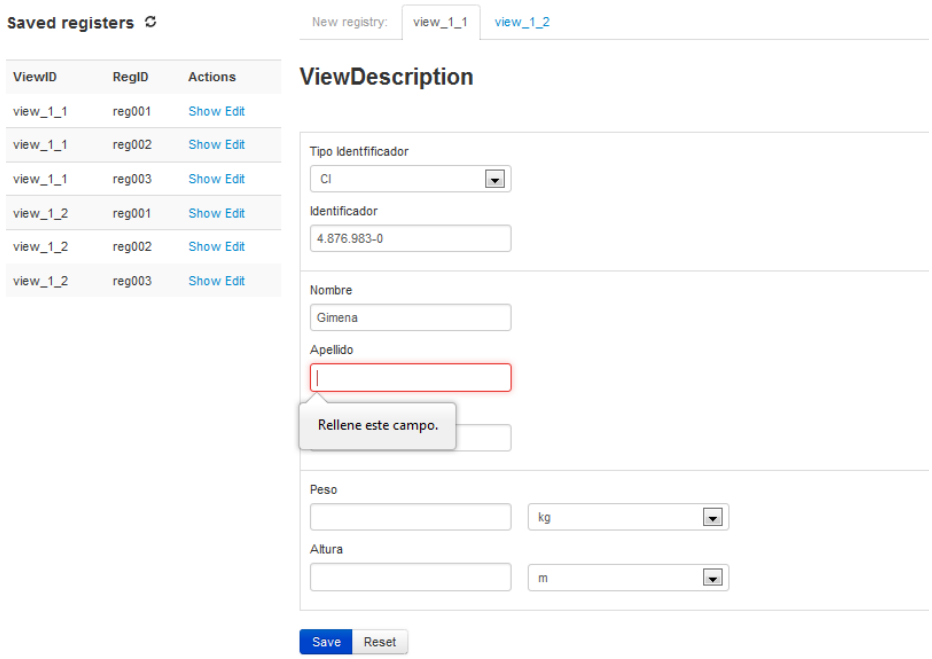


Ilustración 6.8: Validación de datos

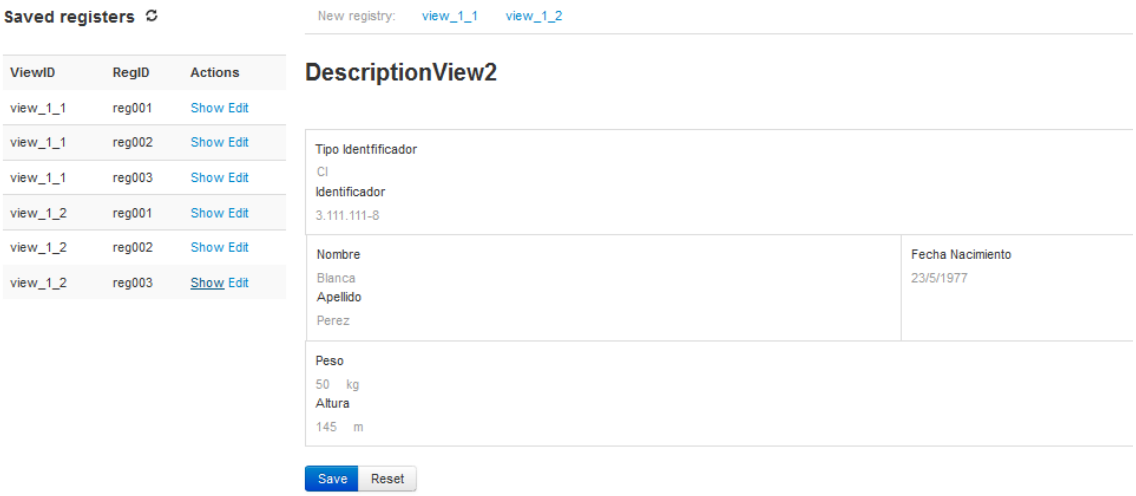


Ilustración 6.9: Visualización de datos ingresados

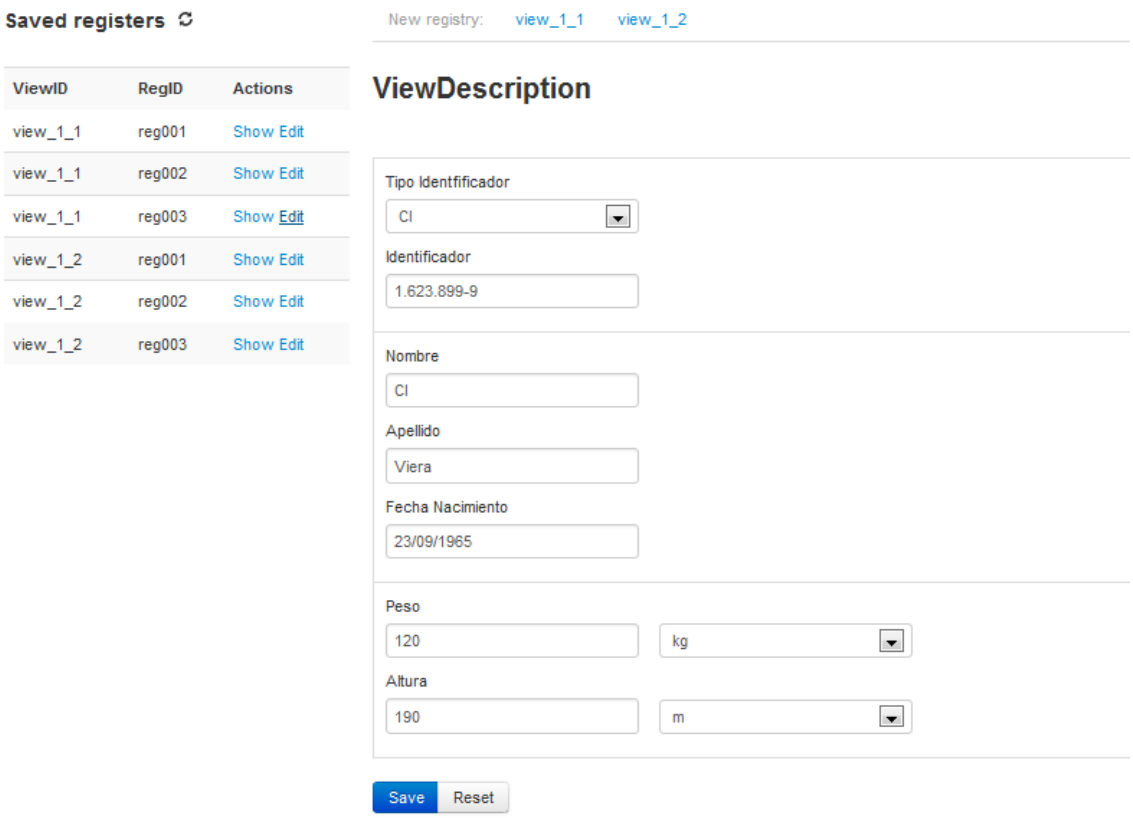


Ilustración 6.10: Edición de datos previamente ingresados

7 Planificación, Esfuerzo efectivo

En este capítulo se pretende mostrar la distribución del tiempo empleado para el proyecto (*ver Gráfico 1*), así como también detallar las actividades involucradas y el tiempo destinado para cada una de ellas (*ver Gráfico 2*).

El registro de horas se realizó mediante planillas compartidas en GoogleDrive. Cada registro se asociaba a una de las nueve categorías definidas: análisis y diseño, comunicación, documentación, gestión, implementación, investigación, requerimientos, verificación y armado de la presentación.

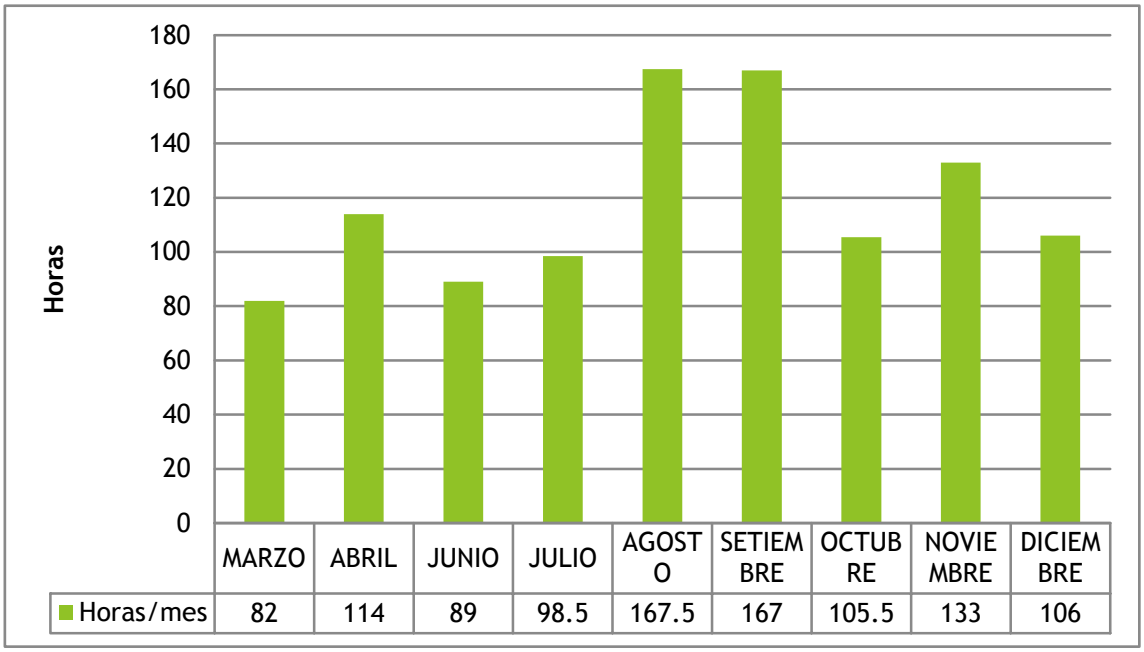


Gráfico 1: Distribución del tiempo (horas) durante los meses de duración del proyecto

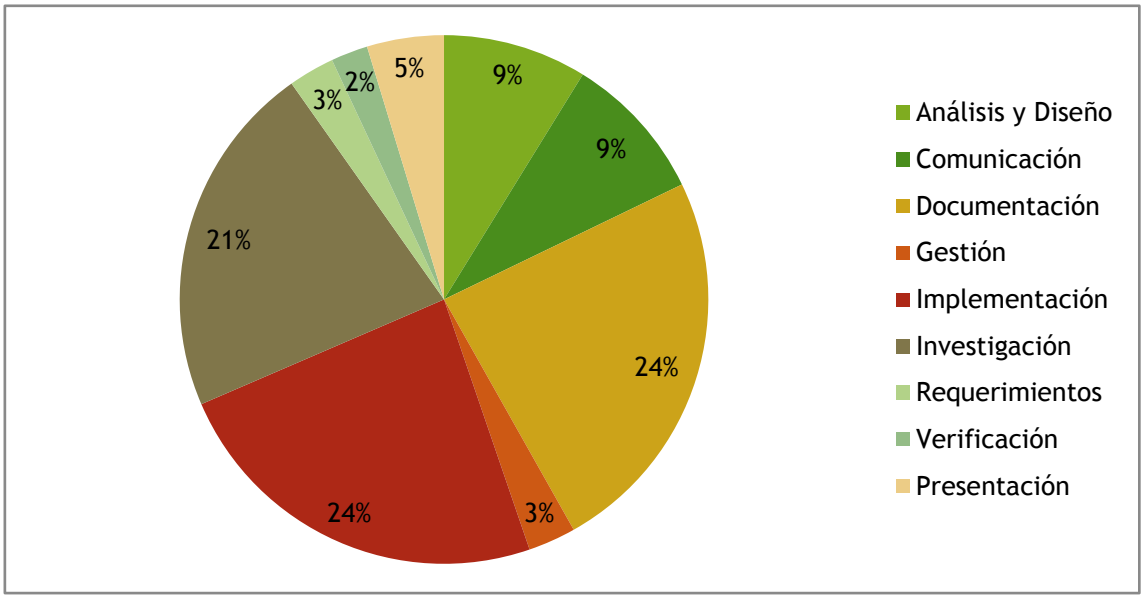


Gráfico 2: Distribución del tiempo (%) en áreas del proyecto

8 Conclusiones

Basándonos en los resultados obtenidos, se considera que los objetivos del proyecto fueron alcanzados.

Se logró construir un prototipo que sirve como herramienta para validar el proceso de definición de interfaces de usuario en base a especificaciones de OpenEHR, y de generación automática de las IU definidas en diferentes tecnologías. Por un lado, se desarrolló el generador de interfaces de usuario basado en un modelo multi-nivel, capaz de generar, a partir de definiciones UITemplate y archivos de mapeo correspondientes a una tecnología dada, artefactos para visualizar las interfaces. En relación a esto, se logró dar soporte a la generación de IU en HTML5 y en XAML. Además, se incluyó un diseñador en línea de comandos que permite crear definiciones de IU en formato XML. En esta herramienta se hallan las reglas básicas de construcción de los UITemplate, así como también la integración (lectura) de los arquetipos utilizados para dicha tarea.

Existen dos puntos fuertes en la herramienta creada. El primero es la integración del modelo propuesto por OpenEHR. La utilización de arquetipos garantiza que el conocimiento clínico sea creado y gestionado por personal de la salud, facilitando el relevamiento de requerimientos para software médico. De esta forma, cuando se define una interfaz de usuario, se hace sobre definiciones de conceptos clínicos (arquetipos) aprobados por personas capacitadas en la materia. El segundo punto a destacar es la generación automática o semi-automática de interfaces de usuario, lo cual permite mejorar los tiempos de prototipación y creación de IU. Este aspecto es favorable para la creación de aplicaciones médicas, ya que permite desacoplar la "lógica de la aplicación" de la "interfaz de usuario", para que puedan evolucionar en caminos diferentes y así adaptarse con más facilidad a los cambios frecuentes que suele sufrir el conocimiento clínico.

La primera etapa del proyecto resultó compleja, principalmente, por no tener bien definido el enfoque deseado para el mismo. Se destinó mucho tiempo en el estudio de diseñadores de interfaces de usuario, el cual no era el objetivo principal del proyecto. Una vez detectado este problema y junto con los actores involucrados en el proyecto, se definió finalmente que el principal objetivo era especificar y prototipar una primera versión del generador. Tras sucesivas iteraciones éste fue evolucionando sus funcionalidades.

Para interiorizarnos en el modelo dual de OpenEHR y comprender cómo integrar los conceptos de arquetipos al proyecto, se destinaron horas de investigación que pueden verse reflejadas en la etapa del estado del arte e inicios de la implementación. A este nivel, nos fue de gran ayuda incorporar una librería desarrollada en EHRGen, la cual se acopló correctamente a nuestro desarrollo.

Además, la formalización del proceso de creación de generadores de código estructuró y ordenó la definición e implementación del mismo.

Se concluye que este proyecto es un primer eslabón para modificar el paradigma de construcción de sistemas médicos planteados por OpenEHR, permitiendo nuestro aporte estar un paso más cerca de lograr ese objetivo.

8.1 Conocimientos adquiridos

La realización del proyecto nos permitió afirmar y profundizar conocimientos adquiridos en diferentes cursos a los que hemos asistido:

- Ingeniería de Software _ aportándonos conocimiento en materia de gestión de proyecto de mediada y gran envergadura.
- Diseño de Compiladores _ facilitándonos el entendimiento del proceso de creación de un generador de código.
- Programación 4 _ proporcionándonos el conocimiento de diferentes patrones de diseño a la hora de modelar el dominio y arquitectura del sistema.
- Relaciones Personales en Ingeniería de Software _ aportándonos técnicas en el área de relacionamiento dentro del equipo y con el cliente.
- Fundamentos de la Web Semántica _ ayudándonos en la definición de metadata, utilizando conocimientos sobre XML, XML Schema.
- Sistemas de Información en Salud _ motivándonos a la realización de este proyecto e introduciéndonos en el concepto de OpenEHR.

Las investigaciones realizadas sobre OpenEHR, generación automática de código y fundamentos de diseño de IU, jugaron un papel muy importante en todo el proceso, resultando sumamente interesantes ya que ampliaron nuestros horizontes como profesionales.

A nivel técnico la implementación se realizó con un lenguaje que no dominábamos, pero que, gracias a su flexibilidad y su gran parecido a Java pudimos adaptar ágilmente. Su potencia y versatilidad nos resultaron muy útiles en el desarrollo de las herramientas, así como también las librerías XMLSlurper y Markup Builder que facilitaron enormemente la manipulación de XML.

Otras tecnologías que formaron parte de nuestro estudio fueron:

- XAML, para la generación de IU de escritorio.
- HTML5, para la generación de IU web.
- Bootstrap, front-end framework para un desarrollo web más rápido y sencillo.
- JavaScript y JQuery, lenguajes para dar dinamismo a las IU web.
- XML y XSD, para la construcción del formato computable.
- PHP, para desarrollar una aplicación que vincule las IU generadas.
- Apache Ant, librería de Java para la automatización de compilación.

8.2 Trabajo futuro

Tanto a nivel del generador como el diseñador de interfaces de usuario, existen aspectos a extender y mejorar.

- Sería deseable que el generador fuera de tipo activo, de forma que se mantenga un vínculo a largo plazo con el código generado, permitiendo que el generador se pueda ejecutar varias veces sobre el mismo código. Como se mencionó en el estado del arte, para que un generador de código activo sea eficaz, debe proporcionar una manera para que el desarrollador pueda personalizar el código de salida, y luego permitir la regeneración del código sin sobrescribir las personalizaciones.

- Soportar arquetipos no planos (con slot). Una alternativa posible sería integrar en el generador una herramienta capaz de "aplanar" los arquetipos. Actualmente existe una versión en desarrollo por el cliente del proyecto.

- Integrar todo los tipos de datos de OpenEHR restantes. En el *Anexo F "Manual de usuario técnico"*, se describe cómo es posible integrar más tipos de datos al Mapper.

- Definir mappers para otras tecnologías, cómo puede ser SwiXML para aplicaciones de escritorio.

- Evolucionar mappers parar las tecnologías trabajadas.

- Permitir otro tipo de lectura. Actualmente las definiciones de templates y arquetipos se leen desde disco, pero por ejemplo, sería deseable leer archivos desde un svn, promoviendo así la gestión de artefactos compartidos. En el *Anexo F "Manual de usuario técnico*", se explica cómo es posible extender esta funcionalidad.

- Gestionar el versionado de UITemplates y artefactos generados, mejorando el control sobre el ciclo de vida de los mismos. A modo de ejemplo, si la lectura y almacenamiento se gestionara desde un repositorio remoto como gitHub, el tema del versionado se resolvería por los mismos mecanismos del repositorio.

- Soportar multiplicidad en XAML a nivel de componentes.

- Generar una aplicación integradora para las IU generadas XAML y evaluar el comportamiento de las mismas, tal como se realizó para HTML5

- Evolucionar el diseñador de definiciones de interfaces de usuario para que:
  - ejecute de forma interactiva.
  - muestre las estructuras de los arquetipos en pantalla.
  - lea definiciones de IU desde del repositorio local/remoto considerando versionado (ejemplo, cargando la última versión por defecto).
  - edite una definición de UITemplate existente en el repositorio local/remoto.

Glosario

ADL
(Archetype Definition Language) Lenguaje formal que permite representar la estructura de los arquetipos., 8, 22, 25, 26, 27, 74, 75

Archetype Editor
Herramienta que permite diseñar arquetipos., 27

arquetipo
Concpeto clínico (ej. "administración de sustancia"). Se puede entender como una composición de diferentes nivels de nodos que pueden ser tanto objetos comoa atibutos., 25, 26, 27, 28, 37, 39, 40, 41, 52, 53, 56, 57, 74, 75, 77, 78, 79, 92, 93, 94, 99

arquetipo plano
Arquetipo que no hace referenca a otros arquetipos, o a parte de ellos, a través de slots., 36

diseñador de interfaces de usuario
Herramienta que permite crear definiciones de interfaces de usuario que luego serán procesadas por un generador., 32, 54

EHRGen
Herramienta de código abierto basada en los estándares establecidos por OpenEHR. Permite la generación de sistemas de historia clínica electrónica orientados a la gestión del conocimiento., 27, 28, 29, 30, 31, 32, 37, 41, 50

generador de código
Herramienta que toma como entrada metadatos, combina los mismos con un motor de templates, y produce una serie de archivos de código fuente como salida., 10, 11, 13, 46, 64

generador de interfaces de usuario
Herramienta que permite generar interfaces de usuario en diferentes tecnologías, tomando como insumos definiciones de IU., 32, 35, 56

GeneXus
Herramienta de desarrollo de software ágil, multiplataforma, basada en conocimiento, orientada principalmente a aplicaciones web empresariales, plataformas Windows y dispositivos móviles o inteligentes., 12, 43

Gestalt
Término de la psicología, que significa "todo unificado" y se refiere a las teorías de la percepción visual., 16

Groovy
Lenguaje dinámico orientado a objetos, muy íntimamente ligado a Java. Simplifica la sintáxis Java y añade una serie de métodos útiles al JDK., 1, 20, 43, 44

informática médica
Aplicación de la informática y las comunicaciones al área de la salud, mediante el uso del software médico., 7, 109

modelo conceptual
Modelo real que se le da al usuario a través de la interfaz del producto., 1, 16, 41

modelo de conocimiento
(OpenEHR) Representa una base de conocimiento creada por profesionales de la salud. Tambié es conocido como modelo de arquetipos., 22, 25

modelo de interfaz de usuario
Estructura a partir de la cual se crean las interfaces de usuario que permitan ingresar, corregir, editar y visualizar datos del registro clínico de un paciente., 8, 54

modelo de workflow
Estructura donde se definen distintas formas de combinación de registros clínicos para distintos usuarios o roles, integrando las diferentes interfaces de usuario., 8

modelo dual
Modelo de desarrollo de OpenEHR. Con el mismo se lograr separar la gestión del conocimiento clínico de la gestión del software., 22

modelo mental
Representa el proceso de pensamiento de una persona para saber cómo funcionan las cosas., 15, 16

modelo refinado
Estructura a través de la cual se crean modelos detallados de los distintos componentes del registro clínico (especificaciones y restricciones)., 8

OpenEHR

Fundación sin fines de lucro creada para permitir el desarrollo de especificaciones software y recursos de conocimiento para los sistemas de información de la salud., 1, 7, 8, 9, 22, 24, 25, 26, 28, 29, 32, 35, 36, 37, 41, 56, 63, 64, 65, 77, 91, 92, 94

template

Estructura que permite agrupar diferentes definiciones de arquetipos bajo alguna "regla de negocio". En salud, un template permite referencias diferentes arqeuetipos tienen una relación lógica bajo determinado hecho clínico., 26, 28, 29, 30, 31, 36, 37, 38, 39, 55, 76, 92, 93, 96, 97, 98, 99, 103, 108

UITemplate

Nuevo modelo de especificación de interfaces de usuario para software médico., 37, 38, 39, 40, 41, 42, 44, 47, 48, 49, 50, 53, 54, 55, 56, 57, 59, 63, 76, 80, 83, 86, 90, 91, 98

variable de sustitución

Permite establecer la estructura configurable de una interfaz de usuario. El conjunto de todas las variables de sustitución conforman el código final de la IU., 45

XAML

(eXtensible Application Markup Language) Lenguaje declarativo basado en XML que permite describir interfaces de usaurio para la Windows Presentation Fundation (WPF)., 8, 21, 32, 53, 55, 56, 63, 65, 94, 95, 96

XML

(eXtensible Markup Language) es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos en forma legible., 8, 11, 18, 19, 21, 27, 41, 44, 50, 63, 64, 80, 106

XML Schema

Utilizado para describir el tipo de contenidos, la estructura y restricciones de documentos XML., 19

Referencias

1. Wikipedia. *Informática en salud.* [En línea] 2013. http://es.wikipedia.org/wiki/Inform%C3%A1tica_en_salud.

2. Open EHR-Gen Framework . *Generador de sistemas de historia clínica electrónica basado en el estándar OpenEHR.* [En línea] [Citado el: 13 de 8 de 2013.] https://code.google.com/p/open-ehr-gen-framework/.

3. **Margolis, Dr. Alvaro.** *Sistemas de información en Salud. Características y funcionalidades para la asistencia médica.* 2013.

4. **Beale, T., y otros, y otros.** Open EHR. *EHR Information Model.* [En línea] 2008 de 8 de 16. [Citado el: 2013 de 5 de 11.] http://www.openehr.org/releases/trunk/architecture/rm/ehr_im.pdf.

5. **Beale, T.** Open EHR. *Archetype Object Model.* [En línea] 2008 de 11 de 20.

6. **Herrington, Jack.** *Code Generation in Action.* Greenwich : s.n., 2003. 1930110979.

7. CodeSmith Tools. [En línea] 2013. http://www.codesmithtools.com/product/generator#overview.

8. Iron Speed. [En línea] 2013. http://www.ironspeed.com/.

9. Codeplex. *AjGenesis.* [En línea] 7 de 11 de 2103. http://ajgenesis.codeplex.com/documentation.

10. **Galitz , Wilbert O.** *Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques.* s.l. : Wiley John + Sons, 2007. 978-0470053423.

11. UX Magazine. *The Secret to Designing an Intuitive UX : Match the Mental Model to the Conceptual Model.* [En línea] 8 de 10 de 2011. http://uxmag.com/articles/the-secret-to-designing-an-intuitive-user-experience.

12. **Mifsud, Justin .** Smashing Magazine. *An Extensive Guide To Web Form Usability.* [En línea] 8 de 11 de 2011. http://uxdesign.smashingmagazine.com/2011/11/08/extensive-guide-web-form-usability/.

13. Creative Bloq. *How to design great web forms.* [En línea] 30 de 8 de 2013. http://www.creativebloq.com/how-design-great-web-forms-8134254.

14. **Sollenberger, Kyle.** Treehaouse Blog. *10 User Interface Design Fundamentals.* [En línea] 7 de 8 de 2012. http://blog.teamtreehouse.com/10-user-interface-design-fundamentals.

15. davidmarco.es. *Introducción a Groovy.* [En línea] 20 de 6 de 2010. http://www.davidmarco.es/blog/entrada.php?id=181.

16. Groovy. [En línea] http://groovy.codehaus.org/.

17. The HTML5 test. [En línea] 11 de 2012. http://html5test.com/.

18. **Pazos, Pablo.** 41JAIIO, Jornadas Argentinas de Informática, 3º Congreso Argentino de Informática y Salud, CAIS 2012. *EHRGen: Generador de Sistemas*

Normalizados de Historia Clínica Electrónica Basados en openEHR. [En línea] 2012. http://www.41jaiio.org.ar/sites/default/files/15_CAIS_2012.pdf.

19. **Beale, Thomas y Heard MD, Sam.** *An Ontology-based Model of Clinical Information*. s.l. : IOS Press, 2007.

20. **Beale, T. y Heard, S.** OpenEHR. *Data Types Information Model*. [En línea] 4 de 2007.

21. —. OpenEHR. *Archetype Definition Language*. [En línea] 12 de 12 de 2008.

22. **Beale, T.** OpenEHR. *OpenEHR Templates*. [En línea] 12 de 1 de 2012. http://www.openehr.org/releases/trunk/architecture/am/tom.pdf.

23. OpenEHR. *ADL Text Editor*. [En línea] 30 de 3 de 2013. http://www.openehr.org/wiki/display/dev/ADL+Text+Editors.

24. Archetype Editor Home. [En línea] 27 de 2 de 2013. http://www.openehr.org/downloads/archetypeeditor/home.

25. OpenEHR. *ADL Workbench*. [En línea] 9 de 9 de 2013. [Citado el: 9 de 10 de 2013.] http://www.openehr.org/downloads/ADLworkbench/home.

26. Google code. *template-editor-open-ehr-gen* . [En línea] http://gc.codehum.com/p/template-editor-open-ehr-gen/.

27. OpenEHR. *Who is using openEHR?* [En línea] 2013. http://www.openehr.org/who_is_using_openehr/#!.

28. OpenEHRGen. *open-ehr-gen-framework*. [En línea] https://code.google.com/p/open-ehr-gen-framework/.

29. **Pazos Gutiérrez, Pablo.** Open EHR-Gen Framework. *Sintaxis y configuración de plantillas*. [En línea] 30 de 10 de 2010. https://code.google.com/p/open-ehr-gen-framework/downloads/detail?name=Template%20sintax%20and%20configuration%20v0.1%20%28Sintaxis%20y%20configuraci%C3%B3n%20de%20plantillas%20%29.pdf&can=2&q=.

30. W3C. *Date and Time Formats*. [En línea] 1997. http://www.w3.org/TR/NOTE-datetime.

31. W3C . *World Wide Web Consortium*. [En línea] http://www.w3.org/XML/Schema.

32. Eclipse. *Eclipse Modeling Framework Project (EMF)*. [En línea] 2013. http://www.eclipse.org/modeling/emf/.

33. iguazoft. [En línea] 23 de 7 de 2012. http://blog.iguazoft.com/maquetar-con-divs-como-si-fuese-una-tabla/.

34. Spring. *Groovy/Grails Tool Suite*. [En línea] 2013. http://spring.io/tools/ggts.

35. jdom.org. [En línea] http://www.jdom.org/.

36. Groovy. *Class XmlSlurper*. [En línea] 2013. [Citado el: 4 de 8 de 2013.] http://groovy.codehaus.org/api/groovy/util/XmlSlurper.html.

37. W3C. *XML Path Language (XPath) 2.0 (Second Edition)*. [En línea] 3 de 1 de 2011. http://www.w3.org/TR/xpath/.

38. W3C. *Markup Validation Service*. [En línea] http://validator.w3.org/.

39. Código final del proyecto. [En línea] https://subversion.assembla.com/svn/gaiusm/Generator.

40. **Carrasco, Leandro y Pazos, Pablo.** *TRAUMAGEN: historia clínica electrónica con acceso telemático a imágenes médicas de pacientes de trauma*. 2010.

41. **Gamma, Erich, y otros, y otros.** *Design Patterns*. 1998.

42. **Nielsen, Jakon.** Nielsen Norman Group. *10 Usability Heuristics for User Interface Design*. [En línea] 1 de 1 de 1995. http://www.nngroup.com/articles/ten-usability-heuristics/.

43. **Spool, Jared M.** User Interface Enginnering. *Great Designs Should Be Experienced and Not Seen*. [En línea] 14 de 5 de 2009. http://www.uie.com/articles/experiencedesign.

44. Developer Apple. *The Philosophy of UI Design: Fundamental Principles*. [En línea] 23 de 7 de 2012. https://developer.apple.com/library/mac/documentation/userexperience/conceptual/applehiguidelines/HIPrinciples/HIPrinciples.html.

45. **Bean, Jame.** *XML for data architects – designing for reuse and integration*. USA : s.n., 2003.

46. **Glushko, Robert J.** *Document Engineering - Analyzing and designing documents for business informatics & web services*.

47. Assembla. [En línea] 2013. https://www.assembla.com/home.

48. W3C. *HTML 5.1 Nightly* . [En línea] , 7 de 11 de 2013. http://www.w3.org/html/wg/drafts/html/master/.

49. w3school.com. *HTML5 Introduction*. [En línea] 2013.

50. Wikipedia. *Ciencias de la Salud*. [En línea] 2008 de 5 de 30.

Índice de Ilustraciones

Ilustración 1.1: Diagrama multinivel 8
Ilustración 3.1: Diagrama del ENTRY Package 23
Ilustración 3.2: Proceso del registro clínico (Clinical Investigator Recording) 23
Ilustración 3.3: Modelo UML del Template Object Model de EHRGen 30
Ilustración 3.4: Layout de template 30
Ilustración 4.1: Metodología de trabajo 33
Ilustración 5.1: Esquema de la solución: generador y diseñador de interfaces de usuario 35
Ilustración 5.2: Modelo de diseño del template 38
Ilustración 5.3: Modelo de diseño del template - Tipo de datos 38
Ilustración 5.4 : Diseño de template 39
Ilustración 5.5: Árbol de instancias para Layout 40
Ilustración 5.6: Estructura básica de IU 42
Ilustración 5.7: Estructura de Layout 43
Ilustración 5.8: IU generada a partir del código de prueba 43
Ilustración 5.9: Secciones de identificadas en el código de prueba 44
Ilustración 5.10: Código HTML por secciones 45
Ilustración 5.11: Plantilla UITemplate 47
Ilustración 5.12: Plantilla UITemplate - contents 48
Ilustración 5.13: Diseño de componentes del generador de interfaces 49
Ilustración 5.14: Diagrama de secuencia del diseñador de interfaces 51
Ilustración 5.15: Multiplicidad de campo 52
Ilustración 5.16: Multiplicidad de componente 52
Ilustración 5.17: Ejemplo de ejecución del diseñador 54
Ilustración 6.1: Interfaz de usuario generada en HTML5 56
Ilustración 6.2: Interfaz de usuario generada en XAML (desktop) 56
Ilustración 6.3: Definición de campo "Apellido" en el UITemplate 57
Ilustración 6.4: Interfaz de usuario generada en HTML5 57
Ilustración 6.5: Integración de herramientas: tiempo de diseño y especificación de artefactos (metadata) 58
Ilustración 6.6: Diagrama de uso de registro clínico electrónico 58
Ilustración 6.7: Ingreso de datos 59
Ilustración 6.8: Validación de datos 60
Ilustración 6.9: Visualización de datos ingresados 60
Ilustración 6.10: Edición de datos previamente ingresados 61

Ilustración A. 1: Estructura de ADL 73

Ilustración H. 1: Estructura final del aplicativo 94
Ilustración H. 2: Ejemplo de UI creada por el Generador (HTML) 102

Índice de Gráficos

Gráfico 1: Distribución del tiempo (horas) durante los meses de duración del proyecto 62
Gráfico 2: Distribución del tiempo (%) en áreas del proyecto 62

Índice de Tablas

Tabla 1: UITemplate descripción *75*
Tabla 2: View descripción *75*
Tabla 3: Layout Descripción *75*
Tabla 4: Zone Descripción *75*
Tabla 5: CompositeZone descripción *75*
Tabla 6: ArchetypeReference descripción *76*
Tabla 7: ArchetypeFieldReference descripción *76*
Tabla 8: Control descripción *76*
Tabla 9: Descripción de tipos de controles *78*
Tabla 10: Descripción gráfica de controles HTML para interfaces de usuario *79*
Tabla 11: Variables de sustitución *89*
Tabla 12: Variables globales *90*
Tabla 13: Archivo generator.config – Parámetros *96*

Anexos

Anexo A. ADL Archetype Definition Language: Caso de estudio

De acuerdo a la *Ilustración A. 1* podemos distinguir diferentes secciones que componen la definición de un ADL y las cuales veremos usando como ejemplo el ADL asociado al arquetipo TRANSFUSION.

En lo que concierne al cabezal del ADL, se hallan las secciones Archetype, Specialise, Concept y Language.

Bajo la Archetype Section, se halla el identificador del arquetipo que se está representando de acuerdo a la nomenclatura dada para estos

```
openEHR-EHR-<entry>.<nombreClínico>.v<númeroVersion>
```

Opcionalmente, puede tener la flag controlled (uncontrolled) la cual indica que la definición también tendrá la sección Revision_History.

```
archetype (adl_version=1.4, uncontrolled)
        openEHR-EHR-INSTRUCTION.transfusion.v1
```

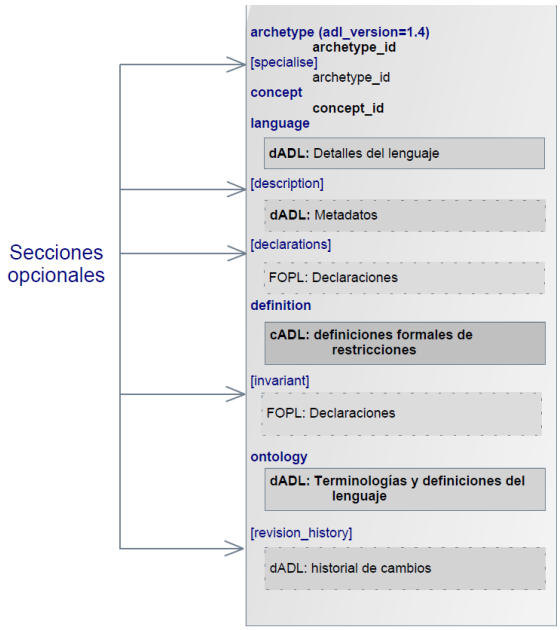


Ilustración A. 1: Estructura de ADL

En caso de que el arquetipo sea una especialización de otro, se incluye la sección Specialise el identificador de éste último. La nomenclatura es la misma que para el caso anterior.

Para que cada arquetipo pueda ser traducido a diferentes idiomas, necesita que el concepto clínico que representa sea identificado de forma única por algún código. Este valor es el que se muestra en la sección de Concept

```
    concept
        [at0000] -- Instructions for transfusion
```

Luego, en la sección Language, se tiene tanto la descripción del lenguaje original en el cual se generó el arquetipo y la lista de lenguajes a los que puede llegar a ser traducido. Se debe tener en cuenta que existe un solo lenguaje original.

```
language
    original_language = <[ISO_639-1::en]>
    translations = <
        ["de"] = <
            language = <[ISO_639-1::de]>
            author = <
        ["name"] = <"Jasmin Buck, Sebastian Garde">
        ["organisation"] = <"University of Heidelberg, Central Queensland University">
            >
        >
    >
```

Finalmente, la última sección que integra el cabezal de un ADL, es Description Section. En ella se halla la información descriptiva del arquetipo (metadatos)

```
description
    original_author = <
        ["name"] = <"unknown">
            >
    details = <
        ["de"] = <
            language = <[ISO_639-1::de]>
        `       purpose = <"Zur Dokumentation von Anweisungen bezüglich einer
Transfusion.">
            use = <"">
            keywords = <"Transfusion", "Blut">
            misuse = <"">
            copyright = <"© openEHR Foundation">
        >
        ["en"] = <
            language = <[ISO_639-1::en]>
            purpose = <"For recording the instructions relating to transfusion">
            use = <"">
            keywords = <"transfusion", "blood">
            misuse = <"">
            copyright = <"© openEHR Foundation">
        >
    >
    lifecycle_state = <"Initial">
    other_contributors = <>
    other_details = <
        ["references"] = <"">
            >
```

Anexo B. Template: Descripción de componentes

UITemplate			
Atributo	Tipo	Opcional	Descripción
UITemplateId	Integer	No	Identificador del template
Name	String	No	Nombre del template
Description	String	No	Breve descripción del template (metadata)
Version	String	No	Versión del template. Por razones de versionado, puede existir un template con el mismo nombre pero otra versión.
NavbarPosition	PositionEnum	No	Posición en la UI del menú navegable entres views
GeneratorVersion	String	No	Indica bajo qué versión de modelo de template fue creada dicha instancia,

Tabla 1: UITemplate descripción

View			
Atributo	Tipo	Opcional	Descripción
ViewId	String	No	Identificador de la vista. En un mismo template no pueden existir dos vistas con el mimos ViewId
aName	String	No	Nombre de la vista
Description	String	Si	Breve descripción de la vista (metadata)
Order	Integer	Si	Orden relativo dentro del template

Tabla 2: View descripción

Layout			
Atributo	Tipo	Opcional	Descripción
LayoutId	String	No	Identificador del Layout. En un mismo template no pueden existir dos Layout con el mimos LayoutId
Name	String	No	Nombre del layout (metadata)

Tabla 3: Layout Descripción

Zone			
Atributo	Tipo	Opcional	Descripción
ZoneId	Integer	No	Identificador del ZoneId

Tabla 4: Zone Descripción

CompositeZone			
Atributo	Tipo	Opcional	Descripción
Orientation	OrientationEnum	No	Indica el tipo de orientación de la zona, es decir la orientación bajo la cual se van a disponer la zonas hijas en dicho CompositeZone.

Tabla 5: CompositeZone descripción

ArchetypeReference			
Atributo	Tipo	Opcional	Descripción
ArchetypeReference	String	No	Nombre del arquetipo padre
Path	String	Si	Indica la ruta del nodo seleccionado en el arquetipo. Cada nodo en un arquetipo tiene a path única
Type	ArchetypeTypeEnum	No	Tipo de nodo (metadata)
OcurrenceMin	String	Si	Cantidad mínima que debe ocurrir el campo en la interfaz de usuario
OcurrenceMax	String	Si	Cantidad máxima que puede ocurrir el campo en la interfaz de usuario

Tabla 6: ArchetypeReference descripción

ArchetypeFieldReference			
Atributo	Tipo	Opcional	Descripción
ElementValueType	DataTypeEnum	Si	Tipo de ELEMENT (metadata)

Tabla 7: ArchetypeFieldReference descripción

Control			
Atributo	Tipo	Opcional	Descripción
ControlId	String	Si	Identificador único del Control
Name	String	No	Nombre descriptivo del Control
Type	String	No	Identificación del <CControlType> asociado
TypeCode	Number	No	Identificación del <CControlType> asociado
NullFlavor	Boolean	No	Admite valor nulo
Position	Number	Si	Indica la posición del field en el leafzone asociado
Skip	Number	Si	Indica que el field asociado al control, estará en línea con los campos inmediatamente anteriores dentro de una LeafZone dada
AttributePath	String	Si	Ruta del nodo del arquetipo que se presentará en al IU

Tabla 8: Control descripción

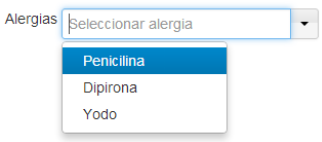
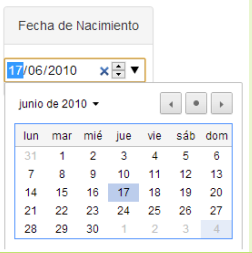
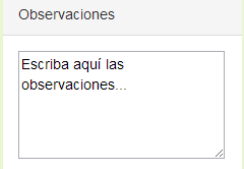
Anexo C. Tipos de controles

Tipo de dato OpenEHR	Atributo	Descripción	#Ctrl.	Tipo de Control
DV_BOOLEAN	value	RadioButton {yes, no, nullflavor(opcional)}	001	CBoolean
	value	ComboBox {yes, no, nullflavor(opcional)}	002	
	value	CheckBox {yes, no}	003	
DV_IDENTIFIER	*issuer*	TextBlock, no editable. Depende de type.	001	CIdentifier
	assigner	TextBlock, no editable. Depende de type.		
	id	TextBlock, editable. No editable si es autogenerado.		
	type	ComboBox		

	issuer	TextBlock, no editable. Depende de type	002	
	assigner	TextBlock, no editable. Depende de type		
	id	TextBlock, editable. No editable si es autogenerado.		
	type	RadioButton. Útil en caso de que se traten de pocos valores (<=5)		
DV_TEXT	*value*	TextBlock	001	CText
	value	TextArea	002	
	Value	TextEditor	003	
DV_CODED_TEXT	*defining_code*	ComboBox en caso de que estén definidos los valores en el arquetipo. No permite la selección múltiple.	001	CCodedText
	defining_code	RadioButton en caso de que estén definidos los valores en el arquetipo y se traten de pocos valores (<=5). No permite la selección múltiple	002	
	defining_code	CheckBox en caso de que estén definidos los valores en el arquetipo y se traten de pocos valores (<=5). Permite la selección múltiple. Permite selección múltiple	003	
DV_ORDINAL	*symbol*	ComboBox en caso de que estén definidos los valores en el arquetipo. No permite la selección múltiple.	001	COrdinal
	symbol	RadioButton en caso de que estén definidos los valores en el arquetipo y se traten de pocos valores (<=5). No permite la selección múltiple	002	
	symbol	CheckBox en caso de que estén definidos los valores en el arquetipo y se traten de pocos valores (<=5). Permite la selección múltiple	003	
DV_QUANTITY	*magnitude*	SelectNumber	001	CQuantity
	units	ComboBox		
DV_COUNT	*magnitude*	TextBlock	001	CCount
	magnitude	SelectNumber		
DV_DURATION	*value*	SelectNumber para cada campo. La cantidad y tipos de campos dependerán de los especificados en el arquetipo. Los posibles campos serán: Año, Mes, Semana, Día, Hora, Minuto y Segundo. Se debe almacenar el valor final siguiendo el estándar ISO 8601.	001	CDuration
DV_PROPORTION	*numerator*	SelectNumber.	001	CProportion
	denominator	SelectNumber		
	numerator denominator	TextBlock con formato (ejemplo “numerator/denominator”)	002	
DV_DATE	*value*	DatePicker	001	CDate
	value	SelectNumber para cada campo. La cantidad y tipos de campos dependerán de los especificados en el arquetipo. Los posibles campos serán: Año, Mes, Día	002	

DV_TIME	*value*	SelectNumber para cada campo. La cantidad y tipos de campos dependerán de los especificados en el arquetipo. Los posibles campos serán: Hora, Minuto y Segundo.	001	CTime
DV_DATE_TIME	*value*	DateTimePicker	001	CDateTime
	value	DataPicker y SelectNumber para Hora, Minuto y Segundo	002	
	value	SelectNumber	003	
DV_MULTIMEDIA	*data*	FileSelect		|CMultimedia
	uri *alternate_text*	TextBlock TextBlock	001	
DV_INTERVAL <DV_ORDERED>	*Range*	Los tipos de controles posibles son CControl, CProportion, CQuantity, CCount, CDuration, CDate, CTime, CDateTime. Ver características en las definiciones anteriores	001	CInterval<CControl>

Tabla 9: Descripción de tipos de controles

Tipo Control	Expresión Gráfica	Descripción
CheckBox		Permite al usuario seleccionar cero o más opciones de un número limitado de opciones.
ComboBox		Permite al usuario seleccionar una única opción entre un número limitado de opciones. Representa un control de selección con una lista desplegable que se puede mostrar u ocultar haciendo clic en la flecha del control.
DatePicker		Permite seleccionar una fecha válida a través un calendario o a través del ingreso manual.
FileSelect		Permite seleccionar una ruta a un archivo de nuestro sistema para ser enviado a un servidor.
RadioButton		Permite al usuario seleccionar una única opción entre un número limitado de opciones.
SelectNumber		Permite ingresar un número en un campo de texto permitiendo incrementar o decrementar su valor haciendo clic en las flechas del control. También podemos especificar el máximo y el mínimo de datos aceptados
TextArea		Define un campo de entrada multilínea para introducir texto

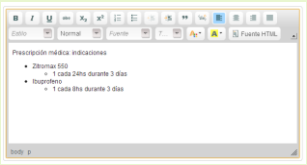
TextBlock		Define un campo de entrada de una línea para introducir texto.
TextEditor		Define un campo de entrada multilínea para introducir texto y funcionalidades para agregarle estilo a mismo. De esta forma es fácil generar texto en negrita (bold), cursiva (Italic), modificar el color, alineación, etc.

Tabla 10: Descripción gráfica de controles HTML para interfaces de usuario

Anexo D. UITemplate: XML Schema

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- UITemplate Definition -->
<xs:element name="UITemplate">
        <xs:complexType>
                <xs:sequence>
                        <xs:element maxOccurs="unbounded" name="Layout" type="Layout"/>
                        <xs:element maxOccurs="unbounded" name="View" type="View"/>
                </xs:sequence>
                <xs:attribute name="GeneratorVersion" type="xs:string" use="required"/>
                <xs:attribute name="UITemplateId" type="xs:integer" use="required"/>
                <xs:attribute name="Name" type="xs:string"/>
                <xs:attribute name="Version" type="xs:string" use="required"/>
                <xs:attribute name="Description" type="xs:string"/>
                <xs:attribute name="NavbarPosition" type="PositionEnum"/>
                <xs:attribute name="LayoutId" type="xs:integer" use="required"/>
        </xs:complexType>
        <!-- Unique Index -->
        <xs:unique name="LayoutUnique">
                <xs:selector xpath="Layout"/>
                <xs:field xpath="@LayoutId"/>
        </xs:unique>
        <xs:unique name="ViewUnique">
                <xs:selector xpath="View"/>
                <xs:field xpath="@ViewId"/>
        </xs:unique>
</xs:element>


<!-- View Definition -->
<xs:complexType name="View">
        <xs:sequence>
                <xs:element maxOccurs="unbounded" name="ArchetypeContainerReference"
type="ArchetypeContainerReference"/>
        </xs:sequence>
        <xs:attribute name="ViewId" type="xs:integer" use="required"/>
        <xs:attribute name="Name" type="xs:string"/>
        <xs:attribute name="Description" type="xs:string"/>
        <xs:attribute name="Order" type="xs:integer"/>
        <xs:attribute name="LayoutId" type="xs:integer"/>
</xs:complexType>

<!-- Layout Definition   -->
<xs:element name="Layout"/>
<xs:complexType name="Layout">
        <xs:choice>
                <xs:element name="CompositeZone" type="CompositeZone"/>
                <xs:element name="LeafZone" type="LeafZone"/>
        </xs:choice>
        <xs:attribute name="LayoutId" type="xs:integer" use="required"/>
        <xs:attribute name="Name" type="xs:string"/>
</xs:complexType>
```

```
<!-- Zone Definition -->
<xs:element abstract="true" name="Zone"/>
<xs:complexType name="Zone">
        <xs:attribute name="ZoneId" type="xs:integer"/>
</xs:complexType>

<!-- LeafZone Definition -->
<xs:element name="LeafZone" substitutionGroup="Zone"/>
<xs:complexType name="LeafZone">
        <xs:complexContent>
                <xs:extension base="Zone"/>
        </xs:complexContent>
</xs:complexType>

<!-- CompositeZone Definition -->
<xs:element name="CompositeZone" substitutionGroup="Zone"/>
<xs:complexType name="CompositeZone">
        <xs:complexContent>
                <xs:extension base="Zone">
                        <xs:sequence>
                                <xs:element maxOccurs="unbounded" ref="Zone"/>
                        </xs:sequence>
                        <xs:attribute name="Orientation" type="OrientationEnum"/>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>


<xs:complexType name="Control">
        <xs:attribute name="ControlId" type="xs:string" use="required"/>
        <xs:attribute name="Name" type="xs:string" use="required"/>
        <xs:attribute name="Type" type="ControlTypeEnum" use="required"/>
        <xs:attribute name="TypeCode" type="xs:integer" use="required"/>
        <xs:attribute name="NullFlavor" type="xs:boolean"/>
        <xs:attribute name="Position" type="xs:integer"/>
        <xs:attribute name="Skip" type="xs:boolean"/>
        <xs:attribute name="AttributePath" type="xs:string"/>
</xs:complexType>

<xs:complexType name="FieldLabel">
        <xs:attribute name="Language" type="xs:string" use="required"/>
        <xs:attribute name="Path" type="xs:string"/>
        <xs:attribute name="NewLabel" type="xs:string"/>
</xs:complexType>

<!-- ArchetypeReference Definition -->
<xs:element abstract="true" name="ArchetypeReference"/>
<xs:complexType name="ArchetypeReference">
        <xs:attribute name="ArchetypeReference" type="xs:string" use="required"/>
        <xs:attribute name="Path" type="xs:string" use="required"/>
        <xs:attribute name="Type" type="ArchetypeTypeEnum"/>
        <xs:attribute name="OccurrenceMin" type="xs:string"/>
        <xs:attribute name="OccurrenceMax" type="xs:string"/>
</xs:complexType>

<!-- ArchetypFieldReference Definition -->
<xs:complexType name="ArchetypeFieldReference">
        <xs:complexContent>
                <xs:extension base="ArchetypeReference">
                        <xs:sequence>
                        <xs:element maxOccurs="unbounded" name="Control" type="Control"/>
                        <xs:element maxOccurs="unbounded" minOccurs="0" name="FieldLabel"
type="FieldLabel"/>
                        </xs:sequence>
                        <xs:attribute name="ElementValueType" type="DataTypeEnum"/>
                <xs:attribute name="ZoneId" type="xs:integer" use="required"/>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>

<!-- ArchtypeContainerReference Definition -->
<xs:complexType name="ArchetypeContainerReference">
        <xs:complexContent>
                <xs:extension base="ArchetypeReference">
```

```
                        <xs:choice>
                        <xs:element maxOccurs="unbounded" name="ArchetypeFieldReference"
type="ArchetypeFieldReference"/>
                        <xs:element name="ArchetypeContainerReference"
type="ArchetypeContainerReference"/>
                        </xs:choice>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>

<xs:simpleType name="PositionEnum">
        <xs:restriction base="xs:string">
                <xs:enumeration value="top"/>
                <xs:enumeration value="bottom"/>
                <xs:enumeration value="left"/>
                <xs:enumeration value="right"/>
        </xs:restriction>
</xs:simpleType>

<xs:simpleType name="OrientationEnum">
        <xs:restriction base="xs:string">
                <xs:enumeration value="horizontal"/>
                <xs:enumeration value="vertical"/>
        </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ArchetypeTypeEnum">
        <xs:restriction base="xs:string">
                <xs:enumeration value="COMPOSITION"/>
                <xs:enumeration value="CONTENT"/>
                <xs:enumeration value="SECTION"/>
                <xs:enumeration value="OBSERVATION"/>
                <xs:enumeration value="INSTRUCTION"/>
                <xs:enumeration value="EVALUATION"/>
                <xs:enumeration value="ACTION"/>
                <xs:enumeration value="INSTRUCTION_DETAILS"/>
                <xs:enumeration value="ISM_TRANSITION"/>
                <xs:enumeration value="ITEM_SINGLE"/>
                <xs:enumeration value="ITEM_LIST"/>
                <xs:enumeration value="ITEM_TREE"/>
                <xs:enumeration value="ADMIN_ENTRY"/>
                <xs:enumeration value="ACTIVITY"/>
                <xs:enumeration value="CLUSTER"/>
                <xs:enumeration value="ELEMENT"/>
                <xs:enumeration value="EVENT_CONTEXT"/>
        </xs:restriction>
</xs:simpleType>

  <xs:simpleType name="DataTypeEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="DvBoolean"/>
      <xs:enumeration value="DvIdentifier"/>
      <xs:enumeration value="DvText"/>
      <xs:enumeration value="DvCodedText"/>
      <xs:enumeration value="DvOrdinal"/>
      <xs:enumeration value="DvInterval"/>
      <xs:enumeration value="DvQuantity"/>
      <xs:enumeration value="DvCount"/>
      <xs:enumeration value="DvProportion"/>
      <xs:enumeration value="DvDate"/>
      <xs:enumeration value="DvTime"/>
      <xs:enumeration value="DvDateTime"/>
      <xs:enumeration value="DvMultimedia"/>
      <xs:enumeration value="DvParsable"/>
    </xs:restriction>
 </xs:simpleType>

<!-- Control Type Definition  -->
<xs:simpleType name="ControlTypeEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="CBoolean"/>
      <xs:enumeration value="CIdentifier"/>
          <xs:enumeration value="CText"/>
      <xs:enumeration value="CCodedText"/>
```

```
      <xs:enumeration value="COrdinal"/>
      <xs:enumeration value="CQuantity"/>
      <xs:enumeration value="CCount"/>
      <xs:enumeration value="CDuration"/>
      <xs:enumeration value="CProportion"/>
      <xs:enumeration value="CDate"/>
      <xs:enumeration value="CTime"/>
      <xs:enumeration value="CDateTime"/>
      <xs:enumeration value="CMultimedia"/>
      <xs:enumeration value="CParsable"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema><?xml version="1.0" encoding="UTF-8" standalone="no"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- UITemplate Definition -->
<xs:element name="UITemplate">
        <xs:complexType>
                <xs:sequence>
                        <xs:element maxOccurs="unbounded" name="Layout" type="Layout"/>
                        <xs:element maxOccurs="unbounded" name="View" type="View"/>
                </xs:sequence>
                <xs:attribute name="GeneratorVersion" type="xs:string" use="required"/>
                <xs:attribute name="UITemplateId" type="xs:integer" use="required"/>
                <xs:attribute name="Name" type="xs:string"/>
                <xs:attribute name="Version" type="xs:string" use="required"/>
                <xs:attribute name="Description" type="xs:string"/>
                <xs:attribute name="NavbarPosition" type="PositionEnum"/>
                <xs:attribute name="LayoutId" type="xs:integer" use="required"/>
        </xs:complexType>


        <!-- Unique Index -->
        <xs:unique name="LayoutUnique">
                <xs:selector xpath="Layout"/>
                <xs:field xpath="@LayoutId"/>
        </xs:unique>

        <xs:unique name="ViewUnique">
                <xs:selector xpath="View"/>
                <xs:field xpath="@ViewId"/>
        </xs:unique>


</xs:element>

<!-- View Definition -->
<xs:complexType name="View">
        <xs:sequence>
                <xs:element maxOccurs="unbounded" name="ArchetypeContainerReference"
type="ArchetypeContainerReference"/>
        </xs:sequence>
        <xs:attribute name="ViewId" type="xs:string" use="required"/>
        <xs:attribute name="Name" type="xs:string"/>
        <xs:attribute name="Description" type="xs:string"/>
        <xs:attribute name="Order" type="xs:integer"/>
        <xs:attribute name="LayoutId" type="xs:integer"/>
</xs:complexType>

<!-- Layout Definition   -->
<xs:element name="Layout"/>
<xs:complexType name="Layout">
        <xs:choice>
                <xs:element name="CompositeZone" type="CompositeZone"/>
                <xs:element name="LeafZone" type="LeafZone"/>
        </xs:choice>
        <xs:attribute name="LayoutId" type="xs:string" use="required"/>
        <xs:attribute name="Name" type="xs:string"/>
</xs:complexType>

<!-- Zone Definition -->
<xs:element abstract="true" name="Zone"/>
<xs:complexType name="Zone">
        <xs:attribute name="ZoneId" type="xs:string"/>
```

```
</xs:complexType>

<!-- LeafZone Definition -->
<xs:element name="LeafZone" substitutionGroup="Zone"/>
<xs:complexType name="LeafZone">
        <xs:complexContent>
                <xs:extension base="Zone"/>
        </xs:complexContent>
</xs:complexType>

<!-- CompositeZone Definition -->
<xs:element name="CompositeZone" substitutionGroup="Zone"/>
<xs:complexType name="CompositeZone">
        <xs:complexContent>
                <xs:extension base="Zone">
                        <xs:sequence>
                                <xs:element maxOccurs="unbounded" ref="Zone"/>
                        </xs:sequence>
                        <xs:attribute name="Orientation" type="OrientationEnum"/>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>

<xs:complexType name="Control">
        <xs:attribute name="ControlId" type="xs:string" use="required"/>
        <xs:attribute name="Name" type="xs:string" use="required"/>
        <xs:attribute name="Type" type="ControlTypeEnum" use="required"/>
        <xs:attribute name="TypeCode" type="xs:integer" use="required"/>
        <xs:attribute name="NullFlavor" type="xs:boolean"/>
        <xs:attribute name="Position" type="xs:integer"/>
        <xs:attribute name="Skip" type="xs:boolean"/>
        <xs:attribute name="AttributePath" type="xs:string"/>
</xs:complexType>

<xs:complexType name="FieldLabel">
        <xs:attribute name="Language" type="xs:string" use="required"/>
        <xs:attribute name="Path" type="xs:string"/>
        <xs:attribute name="NewLabel" type="xs:string"/>
</xs:complexType>

<!-- ArchetypeReference Definition -->
<xs:element abstract="true" name="ArchetypeReference"/>
<xs:complexType name="ArchetypeReference">
        <xs:attribute name="ArchetypeReference" type="xs:string" use="required"/>
        <xs:attribute name="Path" type="xs:string" use="required"/>
        <xs:attribute name="Type" type="ArchetypeTypeEnum"/>
        <xs:attribute name="OccurrenceMin" type="Nat"/>
        <xs:attribute name="OccurrenceMax" type="Nat"/>

</xs:complexType>

<!-- ArchetypFieldReference Definition -->
<xs:complexType name="ArchetypeFieldReference">
        <xs:complexContent>
                <xs:extension base="ArchetypeReference">
                        <xs:sequence>
                                <xs:element maxOccurs="unbounded" name="Control"
type="Control"/>
                                <xs:element maxOccurs="unbounded" minOccurs="0"
name="FieldLabel" type="FieldLabel"/>
                        </xs:sequence>
                        <xs:attribute name="ElementValueType" type="DataTypeEnum"/>
                <xs:attribute name="ZoneId" type="xs:integer" use="required"/>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>

<!-- ArchtypeContainerReference Definition -->
<xs:complexType name="ArchetypeContainerReference">
        <xs:complexContent>
                <xs:extension base="ArchetypeReference">
                        <xs:choice>
                                <xs:element maxOccurs="unbounded" name="ArchetypeFieldReference"
type="ArchetypeFieldReference"/>
```

```
                                <xs:element name="ArchetypeContainerReference"
type="ArchetypeContainerReference"/>
                        </xs:choice>
                </xs:extension>
        </xs:complexContent>
</xs:complexType>

<xs:simpleType name="PositionEnum">
        <xs:restriction base="xs:string">
                <xs:enumeration value="top"/>
                <xs:enumeration value="bottom"/>
                <xs:enumeration value="left"/>
                <xs:enumeration value="right"/>
        </xs:restriction>
</xs:simpleType>

<xs:simpleType name="OrientationEnum">
        <xs:restriction base="xs:string">
                <xs:enumeration value="horizontal"/>
                <xs:enumeration value="vertical"/>
        </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ArchetypeTypeEnum">
        <xs:restriction base="xs:string">
                <xs:enumeration value="COMPOSITION"/>
                <xs:enumeration value="CONTENT"/>
                <xs:enumeration value="SECTION"/>
                <xs:enumeration value="OBSERVATION"/>
                <xs:enumeration value="INSTRUCTION"/>
                <xs:enumeration value="EVALUATION"/>
                <xs:enumeration value="ACTION"/>
                <xs:enumeration value="INSTRUCTION_DETAILS"/>
                <xs:enumeration value="ISM_TRANSITION"/>
                <xs:enumeration value="ITEM_SINGLE"/>
                <xs:enumeration value="ITEM_LIST"/>
                <xs:enumeration value="ITEM_TREE"/>
                <xs:enumeration value="ADMIN_ENTRY"/>
                <xs:enumeration value="ACTIVITY"/>
                <xs:enumeration value="CLUSTER"/>
                <xs:enumeration value="ELEMENT"/>
                <xs:enumeration value="EVENT_CONTEXT"/>
        </xs:restriction>
</xs:simpleType>

  <xs:simpleType name="DataTypeEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="DvBoolean"/>
      <xs:enumeration value="DvIdentifier"/>
      <xs:enumeration value="DvText"/>
      <xs:enumeration value="DvCodedText"/>
      <xs:enumeration value="DvOrdinal"/>
      <xs:enumeration value="DvInterval"/>
      <xs:enumeration value="DvQuantity"/>
      <xs:enumeration value="DvCount"/>
      <xs:enumeration value="DvProportion"/>
      <xs:enumeration value="DvDate"/>
      <xs:enumeration value="DvTime"/>
      <xs:enumeration value="DvDateTime"/>
      <xs:enumeration value="DvMultimedia"/>
      <xs:enumeration value="DvParsable"/>
    </xs:restriction>
 </xs:simpleType>

<!-- Control Type Definition  -->
<xs:simpleType name="ControlTypeEnum">
    <xs:restriction base="xs:string">
      <xs:enumeration value="CBoolean"/>
      <xs:enumeration value="CIdentifier"/>
          <xs:enumeration value="CText"/>
      <xs:enumeration value="CCodedText"/>
      <xs:enumeration value="COrdinal"/>
      <xs:enumeration value="CQuantity"/>
      <xs:enumeration value="CCount"/>
```

```
      <xs:enumeration value="CDuration"/>
      <xs:enumeration value="CProportion"/>
      <xs:enumeration value="CDate"/>
      <xs:enumeration value="CTime"/>
      <xs:enumeration value="CDateTime"/>
      <xs:enumeration value="CMultimedia"/>
      <xs:enumeration value="CParsable"/>
    </xs:restriction>
  </xs:simpleType>

<!-- Control positive numbers (>=0)  -->
  <xs:simpleType name="Nat">
        <xs:restriction base="xs:string">
                <xs:pattern value="[0-9]+"/>
        </xs:restriction>
</xs:simpleType>
</xs:schema>
```

Anexo E. Variables de sustitución

A continuación se presenta el conjunto de variables de sustitución y variables globales soportadas por el generador.

Variable de sustitución	Descripción	Variables globales asociadas
UITemplateHead	Código inicial relacionado a los datos de la entidad UITemplate	#UITEMPLATE_ID# #UITEMPLATE_NAME# #UITEMPLATE_DESCRIPTION# #UITEMPLATE_VERSION #UITEMPLATE_NAVBARPOSITION# #UITEMPLATE_FIRSTVIEWID# #UITEMPLATE_LANGUAGE#"
UITemplateTail	Código final relacionado a los datos de la entidad UITemplate	#UITEMPLATE_ID# #UITEMPLATE_NAME# #UITEMPLATE_DESCRIPTION# #UITEMPLATE_VERSION# #UITEMPLATE_NAVBARPOSITION# #UITEMPLATE_FIRSTVIEWID# #UITEMPLATE_LANGUAGE#
UITemplateNavBarHead	Código inicial relacionado a la barra de navegación para las Views definidas	#UITEMPLATE_ID# #UITEMPLATE_NAME# #UITEMPLATE_NAVBARPOSITION# #UITEMPLATE_FIRSTVIEWID# #UITEMPLATE_LANGUAGE#
UITemplateNavBarTail	Código final relacionado a la barra de navegación para las Views definidas	#UITEMPLATE_ID# #UITEMPLATE_NAME# #UITEMPLATE_NAVBARPOSITION# #UITEMPLATE_FIRSTVIEWID# #UITEMPLATE_LANGUAGE#
UITemplateNavBarItem	Código de cada ítem de la barra de navegación.	#UITEMPLATE_ID# #UITEMPLATE_NAME# #UITEMPLATE_NAVBARPOSITION #UITEMPLATE_FIRSTVIEWID# #UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_NAME# #VIEW_MODE#
UITemplateViewContainerHead	Código inicial del contenedor donde se mostrará la interfaz asociada al View que se seleccione en la barra de navegación	#UITEMPLATE_ID# #UITEMPLATE_NAME# #UITEMPLATE_NAVBARPOSITION# #UITEMPLATE_FIRSTVIEWID# #UITEMPLATE_LANGUAGE#
UITemplateViewContainerTail	Código final del contenedor donde se mostrará la IU asociada al View	#UITEMPLATE_ID# #UITEMPLATE_NAME# #UITEMPLATE_NAVBARPOSITION# #UITEMPLATE_FIRSTVIEWID# #UITEMPLATE_LANGUAGE#

ViewHeader	Código inicial relacionado a la entidad View	#VIEW_LOCALID# #VIEW_GLOBALID# #VIEW_NAME# #VIEW_DESCRIPTION# #VIEW_ORDER# #VIEW_MODE# #UITEMPLATE_LANGUAGE#
ViewTail	Código final relacionado a la entidad View	#VIEW_LOCALID# #VIEW_GLOBALID# #VIEW_NAME# #VIEW_DESCRIPTION# #VIEW_ORDER# #VIEW_MODE# #UITEMPLATE_LANGUAGE#
LayoutHead	Código inicial relacionado a la entidad Layout. Junto al definido par LayoutTail, envolverá al código generado a partir del árbol de Zones asociado al Layout	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #LAYOUT_ID# #LAYOUT_NAME#
LayoutTail	Código final relacionado a la entidad Layout.	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #LAYOUT_ID #LAYOUT_NAME#
LayoutCompZone<Orientation>Head	Código inicial relacionado a la entidad CompositeZone	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #LAYOUT_ID# #LAYOUT_NAME# #ZONE_CHILDRENSIZE# #ZONE_CHILDRENINDEX# #REP_num#$code$#REP#
LayoutCompZone<Orientation>Tail	Código final relacionado a la entidad CompositeZone	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #LAYOUT_ID# #LAYOUT_NAME# #ZONE_CHILDRENSIZE# #ZONE_CHILDRENINDEX# #REP_num#$code$#REP#
LayoutCompZoneChild<Orientation>Head	Dada una entidad CompositeZone, para cada una de sus Zone hijas se genera su código inicial (independientemente del tipo de Zone)	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #LAYOUT_ID# #LAYOUT_NAME# #ZONE_CHILDRENSIZE# #ZONE_CHILDRENINDEX# #REP_num#$code$#REP#
LayoutCompZoneChild<Orientantion>Tail	Dada una entidad CompositeZone, para cada una de sus Zone hijas se genera su código final	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #LAYOUT_ID# #LAYOUT_NAME# #ZONE_CHILDRENSIZE# #ZONE_CHILDRENINDEX# #REP_num#$code$#REP#
LayoutLeafZoneHead	Código inicial relacionado a la entidad LeafZone	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #LAYOUT_ID# #LAYOUT_NAME# #ZONE_ID# #ZONE_ORIENTATION# #ZONE_CHILDRENINDEX#

LayoutLeafZoneTail	Código final relacionado a la entidad LeafZone	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #LAYOUT_ID# #LAYOUT_NAME# #ZONE_ID# #ZONE_ORIENTATION# #ZONE_CHILDRENINDEX#"
FieldLineHead	Código final que encapsula la representación del conjunto de fields que se configuren para visualizarse *in line*	#UITEMPLATE_LANGUAGE#" #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE#
FieldLineTail	Código final que encapsula la representación del conjunto de fields que se configuren para visualizarse *in line*	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE#
FieldHead	Genera el código inicial que encapsula la representación generada para la entidad ArchetypeFieldReference y su Control asociado	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE#
FieldTail	Genera el código final que envolverá a la representación generada para la entidad ArchetypeFieldReference y su Control asociado	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE#
FieldLabel	Código del label que será asociado a una entidad ArchetypeFieldReference	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_OCCURRENCEINDEX#
FieldShowMode	Código asociado a una entidad ArchetypeFieldReference y su Control asociado para una View en modo show	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_OCCURRENCEINDEX#
Field<ControlType><id>Head	Código inicial relacionado a la entidad ArchetypeFieldReference y a su Control asociado. ControlType es de tipo ControlTypeEnum e id es un identificador único del ControlType definido	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_POSITION# #FIELD_DEFAULTVALUE# #FIELD_NULLFLAVOR# #FIELD_MANDATORY# #FIELD_MINOCCURRENCE# #FIELD_MAXOCCURRENCE# #FIELD_OCCURRENCEINDEX#

Field<ControlType><id>Tail	Código final relacionado a la entidad ArchetypeFieldReference y a su Control asociado.	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_POSITION# #FIELD_DEFAULTVALUE# #FIELD_NULLFLAVOR# #FIELD_MANDATORY# #FIELD_MINOCCURRENCE# #FIELD_MAXOCCURRENCE# #FIELD_OCCURRENCEINDEX#
Field<ControlType><id>Item	Código relacionado a cada valor definido para un ArchetypeFieldReference cuyo data type asociado soporte múltiples valores (ejemplo DV_CODEDTEXT)	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_OCCURRENCEINDEX# #FIELDITEM_ID# #FIELDITEM_NAME#
Field<ControlType><id>ItemSelected	Código relacionado al valor definido por defecto de un ArchetypeFieldReference cuyo data type asociado soporte múltiples valores (ejemplo DV_CODEDTEXT)	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_OCCURRENCEINDEX# #FIELDITEM_ID# #FIELDITEM_NAME#
Field<*ControlType*><id><attribute>Head	Código final relacionado a la entidad ArchetypeFieldReference y a su Control asociado para la magnitud	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_POSITION# #FIELD_DEFAULTVALUE# #FIELD_MINVALUE# #FIELD_MAXVALUE# #FIELD_NULLFLAVOR# #FIELD_MANDATORY# #FIELD_MINOCCURRENCE# #FIELD_MAXOCCURRENCE# #FIELD_OCCURRENCEINDEX#
Field<*ControlType*><id><attribute>Tail	Código final relacionado a la entidad ArchetypeFieldReference y a su Control asociado para la magnitud	UITEMPLATE_LANGUAGE# VIEW_GLOBALID# VIEW_LOCALID# VIEW_NAME# VIEW_MODE# FIELD_NAME# FIELD_FULLTYPE# FIELD_PATH# FIELD_POSITION# FIELD_DEFAULTVALUE# FIELD_MINVALUE# FIELD_MAXVALUE# FIELD_NULLFLAVOR# FIELD_MANDATORY# FIELD_MINOCCURRENCE# FIELD_MAXOCCURRENCE# FIELD_OCCURRENCEINDEX#
FieldNullFlavorHead	Código inicial para la opción Nullflavour que será asociada a una entidad ArchetypeFieldReference y su Control asociado, donde Control.Nullflavor=true	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_POSITION# #FIELD_OCCURRENCEINDEX#

FieldNullFlavorTail	Código final para la opción Nullflavour que será asociada a una entidad ArchetypeFieldReference y su Control asociado, donde Control.Nullflavor=true	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_POSITION# #FIELD_OCCURRENCEINDEX#
FieldNullFlavorItem	Código para cada ítem asociado al control de selección para la opción Nullflavour de una entidad ArchetypeFieldReference	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_NAME# #FIELD_FULLTYPE# #FIELD_PATH# #FIELD_POSITION# #FIELD_OCCURRENCEINDEX#
MultiplicityHead	Código inicial para la funcionalidad de multiplicidad de la entidad ArchetypeReference	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_PATH# #FIELD_MINOCCURRENCE# #FIELD_MAXOCCURRENCE# #FIELD_OCCURRENCEINDEX#
MultiplicityTail	Código final para la funcionalidad de multiplicidad de la entidad ArchetypeReference	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_PATH# #FIELD_MINOCCURRENCE# #FIELD_MAXOCCURRENCE# #FIELD_OCCURRENCEINDEX#
MultiplicityButton	Código del botón que permitirá al usuario ejecutar la funcionalidad de multiplicidad de la entidad ArchetypeReference	#UITEMPLATE_LANGUAGE# #VIEW_GLOBALID# #VIEW_LOCALID# #VIEW_NAME# #VIEW_MODE# #FIELD_PATH# #FIELD_MINOCCURRENCE# #FIELD_MAXOCCURRENCE# #FIELD_OCCURRENCEINDEX#

Tabla 11: Variables de sustitución

Variables Globales

Variable global	Descripción
#UITEMPLATE_ID#	Retorna el valor del atributo UITemplateId de la entidad UITemplate.
#UITEMPLATE_NAME#	Retorna el valor del atributo Name de la entidad UITemplate.
#UITEMPLATE_DESCRIPTION#	Retorna el valor del atributo Description de la entidad UITemplate.
#UITEMPLATE_VERSION#	Retorna el valor del atributo Version de la entidad UITemplate.
#UITEMPLATE_NAVBARPOSITION#	Retorna el valor del atributo NavbarPosition de la entidad UITemplate.
#UITEMPLATE_FIRSTVIEWID#	Retorna el valor del atributo ViewId de la entidad View tal que View.Order=1.
#LAYOUT_ID#	Retorna el valor del atributo LayoutId de la entidad Layout.
#ZONE_ID#	Retorna el valor del atributo ZoneId de la entidad Zone.
#ZONE_ORIENTATION#	Retorna el valor del atributo Orientation de la entidad CompositeZone.
#ZONE_CHILDRENINDEX#	Dada una entidad CompositeZone, retorna el índice del hijo de tipo Zone que se está procesando. Valor inicial = 0.

#ZONE_CHILDRENSIZE#	Dada una entidad CompositeZone, retorna la cantidad de hijos de tipo Zone que tiene.
#VIEW_LOCALID#	Retorna el valor del atributo ViewId de la entidad View.
#VIEW_GLOBALID#	Retorna el valor del atributo ViewId de la entidad View concatenado con el valor del atributo UITemplateId de la entidad UITemplate. Ejemplo: UITemplateId + "_" + ViewId.
#VIEW_NAME#	Retorna el valor del atributo Name de la entidad View.
#VIEW_DESCRIPTION#	Retorna el valor del atributo Description de la entidad View.
#VIEW_ORDER#	Retorna el valor del atributo Order de la entidad View.
#FIELD_NAME#	Retorna el valor del atributo Name de la entidad Control.
#FIELD_FULLTYPE#	Retorna el valor del atributo Type concatenado con el valor del atributo TypeCode de la entidad Control. Ejemplo: Type + TypeCode
#FIELD_PATH#	Retorna el valor del atributo Path de la entidad ArchetypeReference.
#FIELD_DEFAULTVALUE#	Retorna el valor del atributo DefaultValue de la entidad Control.
#FIELD_MINVALUE#	Retorna el valor mínimo definido en el Arquetipo para determinado elemento. Solo aplica para campos de tipo numéricos.
#FIELD_MAXVALUE#	Retorna el valor máximo definido en el Arquetipo para determinado elemento. Solo aplica para campos de tipo numéricos.
#FIELDITEM_NAME#	Retorna el nombre asociado a uno de los valores definidos para un campo de tipo selección. Ejemplo: un campo cuyo tipo sea DVCodeText.
#FIELDITEM_ID#	Retorna el identificador asociado a uno de los valores definidos para un campo de tipo selección. Ejemplo: un campo cuyo tipo sea DVCodeText.
#UITEMPLATE_LANGUAGE#	Retorna el valor del atributo Language de la entidad UITemplate.
#FIELD_MAXOCCURRENCE#	Retorna el valor del atributo OccurrenceMax de la entidad ArchetypeReference.
#FIELD_MINOCCURRENCE#	Retorna el valor del atributo OccurrenceMin de la entidad ArchetypeReference.
#FIELD_MANDATORY#	Retorna el valor 0 si el campo NO es obligatorio. Retorna el valor 1 en el caso contrario.
#FIELD_NULLFLAVOR#	Retorna el valor del atributo NullFlavor de la entidad Control.
#FIELD_POSITION#	Retorna el valor del atributo Position de la entidad Control.
#FIELD_OCCURRENCEINDEX#	Retorna el número de ocurrencia de un campo en la pantalla.
#REP_num#$code$#REP#	Repite num veces el codigo $code$.

Tabla 12: Variables globales

Anexo F. Manual de usuario técnico

Mediante este documento se detallará cómo extender la versión actual del generador de interfaces para software médico (GISM) para soportar nuevos componentes a nivel de arquetipos, a componentes de OpenEHR e interfaces de usuario en una tecnología final seleccionada. A demás se destina una breve sección a describir como se implementa la multiplicidad para HTML5 en las definiciones de mapper actuales.

Las configuraciones que se presentarán en este documento serán las siguientes:

- Agregar nuevos tipos de datos
- Agregar nuevos tipos de controles para un tipo de datos existente
- Agregar una nueva tecnología para generar interfaces de usuario
- Agregar un nuevo tipo de lectura

Agregar nuevos tipos de datos

Actualmente la aplicación soporta cinco tipos de dato de OpenEHR: DV_TEXT, DV_CODEDTEXT, DV_COUNT, DV_QUANTITY, DV_DATETIME.

Para dar soporte a un nuevo tipo de dato simple, por ejemplo DV_BOOLEAN se deberá hacer lo siguiente:

i. En la función Mapper.getArchetypeFieldReferenceCode está la discriminación para cada uno de los controles permitidos en el modo *edit*. Se deberá añadir el siguiente código

```
if (control.@Type.text().startsWith("CBoolean")) {
     field  = getFieldData(node, control, true, true)
     codeaux += getValue("Field" + field.fieldFulltype + "Head")
     codeaux += getValue("Field" + field.fieldFulltype + "Tail")
}
```

En cambio, si el tipo de dato que se quiere agregar es complejo, es decir tiene más de un atributo que necesita necesariamente acceder a datos del arquetipo, se deberá complementar el código anterior con los accesos a los nodos necesarios del arquetipo en cuestión.

Agregar nuevo tipo de control par un tipo de dato existente

En el archivo de configuración de una tecnología dada, los tipos de controles para un tipo de datos dato tiene el siguiente formato:

- Variables requeridas:

```
Field<ControlType><id>[<name_attribute>]Head
Field<ControlType><id>[<name_attribute>]Tail
```

- Variables opcionales (depende del tipo de dato en cuestión):

```
Field<ControlType><id>[<name_attribute>][Item][Selected]
Field<ControlType><id>[<name_attribute>][Selected]
```

Donde:

ControlType (requerido)

Tipo de control definido en el modelo de template, ejemplo CBoolean asociado DV_BOOLEAN, CText asociado a DV_TEXT, etc. Cada *ControlType* sigue la sintaxis **C<DataType>.**

Id (requerido)

Identificador del tipo de control. Para un mismo tipo de dato se pueden expresar más de un tipo de control que podrán ser luego referenciados desde la definición del template. El identificador deberá ser único dentro de las configuraciones para un mismo tipo de dato.

nameatribute (opcional)

Algunos de los tipos de datos tienen más de un atributo asociado, por ejemplo es el caso de DV_QUANTITY que tiene los atributos units, y maginitude. Para ambos atributos se va a tener que tener una representación diferente en el interfaz de usuario.

Item (opcional)

Útil en caso de que el tipo de dato obligue a representar un conjunto de datos, como ocurre en el caso del tipo DV_CODEDTEXT, el que tiene definidos en el arquetipo un conjunto de códigos.

Selected (opcional)

En caso de representar una colección de elementos, puede ser necesario indicarle al generador que en la interfaz de usuario deber estar seleccionado algún elemento en particular.

Agregar una nueva tecnología para generar interfaces de usuario

Para agregar una nueva tecnología se deberá realizar lo siguiente:

- Crear un archivo de configuración para la tecnología dada que contenga la especificación de código para cada uno de las variables de sustitución a utilizar. El nombre del archivo deberá ser

```
<nombre_tecnologia>.config
```

- En el archivo de configuración del generador (*generator.config*) se deberá agregar la siguiente línea:

```
mapper<nombre_tecnologia>=”path/<nombre_tecnologia>.config"
```

Otra forma de hacer lo anterior, sin necesidad de modificar el generator.config, es desde la invocación del generador desde línea de comandos:

```
java – jar generator <template_name>  -t <nombre_teconologia>  -p path/<nombre_tecnologia>.config
```

Agregar un nuevo tipo de lectura

Para agregar un nuevo tipo de lectura, como por ejemplo lectura desde ssh, se deberá agregar en el package **Loader** una clase que implemente las operaciones definidas por la clase **Loader**.

En el archivo de configuración (*generator.config*) se debe configurar la clase que implementa la nueva lectura

```
accessType="<package.nombre_clase>"
```

Multiplicidad en HTML5

En esta sección se pretende describir que elementos están involucrados en la gestión de multiplicidad, de modo que, si se quiere trabajar con algún método similar, se pueda tener una base para comenzar a trabajar en ello.

Vamos a presentar la multiplicidad para entidades de tipo ArchetypeFieldReference y ArchetypeContainerReference.

Multiplicidad a nivel de ArchetypeFieldReference

El código asociado a un field que soporta mutlplicidad máxima se inserta entre las variables de sustitución específicas FieldMultiplicitHead, FieldMultiplicityTail.

En la UI resultante, se identifica que un campo es múltiple porque se le asocia un botón (add). Luego, mediante .js, se gestionan tanto los eventos de multiplicidad como el control de máximo permitidos.

Para ser consistentes con la unicidad de identificadores de elementos en HTML5, dado que el identificador configurado en los mapper en el siguientes

```
id="#VIEW_MODE#_#FIELD_CONTROLID#_o#FIELD_OCCURRENCEINDEX#"
```

se utiliza la variable global #FIELD_OCCURRENCEINDEX# para crear un identificador único para cada field. Es decir, que a medida que, en tiempo de ejecución el usuario adiciona nuevos campos, se incrementa dicha variable.

Multiplicidad a nivel de ArchetypeContainerReference

La solución es similar a la plantead con los tipo field, salvo que el código de multiplicidad asociado es diferente. Cada bloque de controles incluidos en un container estará contenido entre las siguientes etiquetas:

En este caso, la variable global que identifica al containter es #FIELD_PATH#, dado que es única dentro de un arquetipo. El .js en tiempo de ejecución es responsable de multiplicar el bloque y los elementos internos, con la precaución de mantener la unicidad de los identificadores de dichos elementos (similar al caso field).

Anexo G. Manual de Usuario Funcional

En esta sección se presentarán los componentes de software principales creados para el desarrollo del generador de interfaces para software médico (GISM). El GISM permite convertir definiciones de interfaces de usuario (IU) fuertemente basadas en el modelo de referencia de OpenEHR, en artefactos HTML y XAML.

Conjuntamente con el GISM, se presentará el Diseñador de Interfaces para Software Médico (DISM) en su versión de línea de comando, que facilitará la creación

de los archivos de definición de IU (también llamados UItemplates) que serán tomadas como insumos por el GISM.

Requerimientos del sistema

Para la compilación del DISM y GISM:

- Apache Ant 1.9.2

Para la ejecución del DISM y GISM:

- Java 7 o superior

Para la visualización de artefactos generados en HTML:

- Navegador (Google Chrome recomendado)

Para la visualización de artefactos generados en XAML:

- Microsoft .NET Framework 4.5
- Microsoft Visual Studio
- Internet Explorer / Kaxaml

Estructura del entregable

El directorio del proyecto se encuentra disponible en (39).Tal como muestra la figura, en el directorio del entregable se podrá hallar la siguiente estructura:

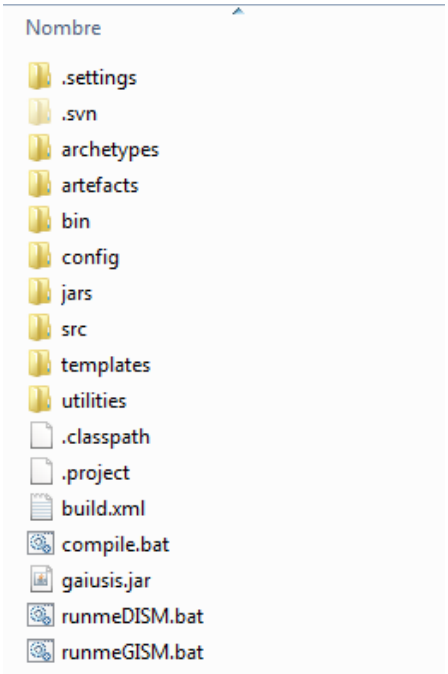


Ilustración H. 1: Estructura final del aplicativo

archetypes

Contiene los arquetipos (archivos de extensión .adl) que podrán ser utilizados en los diferentes archivos de definición de IU.

config

Contiene los archivos de configuración necesarios para ejecutar la aplicación y que se detallarán más adelante. Bajo esta carpeta, también se halla la definición del

modelo de dominio (archivos de extensión .xsd) para poder validad los template generados.

artefacts

Es la carpeta destino de los artefactos finales generados. Los artefactos generados por una misma tecnología, se almacenarán dentro de una carpeta que llevará el nombre de la tecnología utilizada. Ejemplo: todos los artefactos generados para html, se almacenarán en el directorio *artefacts/html*.

Adicionalmente contiene hojas de estilo en cascadas (archivos de extensión .css) y archivos .xaml para brindarle estilo a los artefactos generados de forma desacoplada, scripts Javascript (archivos de extensión .js) y archivos .xaml.vb para brindarle dinamismos a las IU.

templates

En dicha carpeta se hallan por defecto las definiciones de IU (archivos de extensión .xml) a partir de las cuales se generarán los correspondientes artefactos de salida.

GISM.jar

Aplicación para generar a partir de un template, los artefactos asociados en formato HTML o XAML. Junto con su archivo de ejecución runmeGISM.

DISM.jar

Aplicación para construir las definiciones de IU. Junto con su archivo de ejecución runmeDISM.

utilities

En dicha carpeta se hallan los instaladores de herramientas que se han utilizado en el proyecto y el directorio *integrator-app* correspondiente a la herramienta que ejemplifica como se integrarían los artefactos de interfaz de usuario generados por el GISM, con la lógica de una aplicación.

En la raíz del directorio se encuentran los archivos necesarios para la compilación y ejecución de las herramientas presentadas, los cuales explicaremos en el siguiente capítulo.

Compilación

Nota: Antes de comenzar con el proceso de compilación debemos tener instalado apache-ant y haber definido las variables de entorno correspondientes.

Para compilar el proyecto, en la raíz del directorio principal, se deberá ejecutar el archivo compile.bat. Dicho ejecutable será el encargado de compilar los archivos fuente del proyecto (ubicados en el directorio src), y generar el jar con las clases correspondientes (gaiusis.jar). Las clases generadas se almacenarán en el directorio bin. Para llevar a cabo este proceso, se utiliza el software ant-apache y el archivo que define el proceso es bulid.xml.

Configuración

Existen dos tipos de archivos de configuración para el GISM. El archivo de configuración del generador, a través de cual se configuran por defecto los parámetros de la herramienta, y por otra parte, los archivos de mapeo para las tecnologías específicas a las cuales se quiere generar.

Archivo de configuración del generador

En el archivo genearator.config, bajo el directorio config/, se encontrarán una serie de parámetros configurables que serán insumo de la herramienta de generación. En la *Tabla 13*, se podrá ver la descripción de los diferentes parámetros. Los valores allí definidos, serán los valores por defecto, muchos de los cuales se podrán modificar al momento de invocar el generador.

Nombre del parámetro	Descripción
language	Define el lenguaje por defecto con el que se visualizarán los textos en la IU (en formato ISO 639-1)
accessType	Indica la clase .groovy que implementa el tipo de lectura actual.
saveAccessType	Indica la clase .groovy que implementa el tipo de escritura actual
loaderSrc	Ruta por defecto donde se hallan los template a cargar para su posterior conversión
ArchetypeSrc	Ruta donde se hallan los arquetipos a consumir para las definiciones de template
modelXSDPath	Ruta donde se halla la definición (xsd) del modelo para validar los xml de entrada
terminologyPath	Ruta donde se halla el archivo de terminologías
mapperType	Tipo de mapeo por defecto
mapper<mapperType>	Ruta donde se halla el archivo (.config) de mapeo para la tecnología <mapperType>

Tabla 13: Archivo generator.config – Parámetros

Archivos de mapeo

Para que la conversión de un UItemplate al artefacto final (correspondiente a la tecnología elegida) se lleve a cabo, será necesario proporcionarle al generador un archivo de mapeo donde se definirán las correspondencias entre elementos del modelo definido y el lenguaje final. Para ello se definieron un conjunto de variables especiales que serán necesarias configurar para que la generación sea correcta. Dichas variables se dividirán en dos tipos: variables de sustitución y variables globales.

Ejecución

DISM

El DISM proporciona una manera interactiva de crear un archivo de definición de IU (UItemplate) válido, para ser consumido posteriormente por el GISM.

La especificación del xml asociado al UItemplate se encuentra en *config/modeloXSD_v<nro>.xsd*. El principal objetivo del DISM es proporcionar al usuario una manera genérica de crear dicho xml sin tener previo conocimiento de su especificación.

Para correr la aplicación se deberá ejecutar el archivo *runmeDISM.bat* ubicado en la raíz del directorio principal. Una vez en ejecución ser irá guiando al usuario mediante sencillos pasos, para lograr una correcta definición de un UItemplate. A grandes rasgos, se requerirá el ingreso de datos para los siguientes componentes del UItemplate:

- UITemplate: atributos
- Layouts: atributos
- Composición de zonas (Zones): atributos
- Views: atributos
- Arquetipos (Container) : Propiedades
- Arquetipos (Field): Propiedades, LeafZone
- Controles: Propiedades

GISM

El GISM es el encarado de generar automáticamente, los artefactos correspondientes a las interfaces de usuario asociadas a un UItemplate para una tecnología dada. Los archivos necesarios para su ejecución son:

- *Archivo de configuración del generador:* se encuentra en config/ generator.config y está detallado en el *Anexo G "Manual de Usuario Funcional"*
- *Archivo de mapeo a la tecnología seleccionada:* se encuentra en config/<tecnología>.map y está detallado en la sección. Los elementos que los componen está definidos en el *Anexo E "Variables de sustitución"*
- *Instancia de UItemplate*: archivo xml generado por el DISIM y el cual deberá encontrarse en el directorio *template/*

Para correr la aplicación se deberá ejecutar el archivo *runmeGISM.bat,* el cual puede ser llamado desde línea de comando:

runmeGISM.bat <template_name> [-t <type_name> | -p <path_type_name> | -c <configuration_path>]

Dónde:

- *template_name:* Nombre de la definición .xml a generar. Esta definición se buscará en la ruta loaderSrc del archivo de configuración por defecto (o del que se pase por parámetro)

- *type_name:* Indica el tipo de mapeo a realizar, ejemplo HTML, XAML, etc. En caso de que no se pase este parámetro, se utiliza el definido en la variable mapperType del archivo de configuración por defecto (o del que se pase por parámetro)
- *path_type_name:* Ruta al archivo de mapeo a la tecnología dada. En caso de que se realice la ejecución del generador sin este parámetro, toma por defecto desde el archivo de configuración el valor de mapper<tipo>
- *configuration_path:* Ruta a un nuevo archivo de configuración que suplanta al definido por defecto

Una vez en proceso el GISM realizará los siguientes pasos:

- Carga del template *UItemplate_name* desde la ruta por defecto.
- Control de restricción de dominio, identificadores y otros, sobre el .xsd leído.
- Generación del de los artefactos finales que representarán al template elegido en la tecnología configurada.

Ejemplo

En esta sección mostraremos como crear una IU simple desde cero partiendo del arquetipo openEHR-EHR-COMPOSITION.arquetipo_prueba_001.v1

Paso 1) Ejecutamos el DISM e ingresamos la siguiente información:

```
************************
*** TEMPLATE DESIGNER ***
************************
(for generator v1.0)


>> UITemplate attributes
Name: Template1
Description: Datos paciente
Navbar Position [top,bottom,left,right]: left

>> Layout attributes (global definition)
Name: Layout1
Partition size ( > 0 ): 2
Partition orientation (H)orizontal/(V)ertical: h

  > Zone # 1
  Partition vertically this zone? (y/n): n
  Zone z2 has been created
  > Zone # 2
  Partition vertically this zone? (y/n): n
  Zone z3 has been created

Add an extra layout definition? (y/n): n

>> View attributes
Name: Datos Paciente
Description: Datos Paciente
Order ( > 0 ): 1

Link with Layout (if empty, default value l1)
  >l1: Layout1
  Select a value:
```

Cantidad de particiones en que se dividirá la zona inicial de una Layout. La orientación de la partición podrá ser horizontal o vertical.

```
Link with Archetype file
  >1: archetypes\openEHR-EHR-ACTION.procedure.v1.adl
  >2: archetypes\openEHR-EHR-ACTION.transfusion.v1.adl
  >3: archetypes\openEHR-EHR-COMPOSITION.arquetipo_prueba_001.v1.adl
  >4: archetypes\openEHR-EHR-EVALUATION.triage.v1.adl
  >5: archetypes\openEHR-EHR-EVALUATION.triageI.v1.adl
  >6: archetypes\openEHR-EHR-INSTRUCTION.imaging.v1.adl
  >7: archetypes\openEHR-EHR-INSTRUCTION.request.v1.adl
  >8: archetypes\openEHR-EHR-INSTRUCTION.transfusion.v1.adl
  >9: archetypes\openEHR-EHR-OBSERVATION.pulse.v1.adl
  >10: archetypes\openEHR-EHR-SECTION.adhoc.v1.adl
  Select a value: 3

>> ArchetypeContainerReference attributes
Possible path values for container node:
  >1: / (RmTypeName COMPOSITION)
  >2: /context (RmTypeName EVENT_CONTEXT)
  >3: /context/other_context[at0001] (RmTypeName ITEM_TREE)
  >4: /context/other_context[at0001]/items[at0014] (RmTypeName CLUSTER)
  Select a value: 4

Minimal occurrence (if empty, default value 1):
Maximal occurrence (if empty, default value 1):

Add a child-node for path /context/other_context[at0001]/items[at0014] (y/n): y
Select a type for the new node (C)ontainer/(F)ield: f

>> ArchetypeFieldReference attributes
Possible path values for field node :
  >1: /context/other_context[at0001]/items[at0014]/items[at0003] (RmTypeName ELEMENT)
  >2: /context/other_context[at0001]/items[at0014]/items[at0004] (RmTypeName ELEMENT)
  >3: /context/other_context[at0001]/items[at0014]/items[at0009] (RmTypeName ELEMENT)
  >4: /context/other_context[at0001]/items[at0014]/items[at0010] (RmTypeName ELEMENT)
  >5: /context/other_context[at0001]/items[at0014]/items[at0011] (RmTypeName ELEMENT)
  >6: /context/other_context[at0001]/items[at0014]/items[at0012] (RmTypeName ELEMENT)
  >7: /context/other_context[at0001]/items[at0014]/items[at0013] (RmTypeName ELEMENT)
  Select a value: 2

Minimal occurrence (if empty, default value 0):
Maximal occurrence (if empty, default value 1):

Link with Zone from layout Layout1
  > z2
  > z3
  Select a value: z2

>>Control attributes
Name: CTipoIdentificador_1
TypeCode for ccodedtext (type a number): 1
NullFlavor (y/n): n
Position (type a number): 1
Display in new line (y/n): n

Add another field for /context/other_context[at0001]/items[at0014] (y/n): y
Possible path values for field node :
  >1: /context/other_context[at0001]/items[at0014]/items[at0003] (RmTypeName ELEMENT)
  >2: /context/other_context[at0001]/items[at0014]/items[at0004] (RmTypeName ELEMENT)
  >3: /context/other_context[at0001]/items[at0014]/items[at0009] (RmTypeName ELEMENT)
  >4: /context/other_context[at0001]/items[at0014]/items[at0010] (RmTypeName ELEMENT)
  >5: /context/other_context[at0001]/items[at0014]/items[at0011] (RmTypeName ELEMENT)
  >6: /context/other_context[at0001]/items[at0014]/items[at0012] (RmTypeName ELEMENT)
  >7: /context/other_context[at0001]/items[at0014]/items[at0013] (RmTypeName ELEMENT)
  Select a value: 1

Minimal occurrence (if empty, default value 0):
Maximal occurrence (if empty, default value 3):

Link with Zone from layout Layout1
  > z2
```

Cuando hay un valor indicando como default, no es obligatorio ingresar valor

```
  > z3
  Select a value: z2

>>Control attributes
Name: CIdentificador_1
TypeCode for ccount (type a number): 1
NullFlavor (y/n): n
Position (type a number): 2
Display in new line (y/n): n

Add another field for /context/other_context[at0001]/items[at0014] (y/n): y
Possible path values for field node :
  >1: /context/other_context[at0001]/items[at0014]/items[at0003] (RmTypeName ELEMENT)
  >2: /context/other_context[at0001]/items[at0014]/items[at0004] (RmTypeName ELEMENT)
  >3: /context/other_context[at0001]/items[at0014]/items[at0009] (RmTypeName ELEMENT)
  >4: /context/other_context[at0001]/items[at0014]/items[at0010] (RmTypeName ELEMENT)
  >5: /context/other_context[at0001]/items[at0014]/items[at0011] (RmTypeName ELEMENT)
  >6: /context/other_context[at0001]/items[at0014]/items[at0012] (RmTypeName ELEMENT)
  >7: /context/other_context[at0001]/items[at0014]/items[at0013] (RmTypeName ELEMENT)
  Select a value: 3

Minimal occurrence (if empty, default value 0):
Maximal occurrence (if empty, default value 1):

Link with Zone from layout Layout1
  > z2
  > z3
  Select a value: z3

>>Control attributes
Name: CNombre_1
TypeCode for ctext (type a number): 1
NullFlavor (y/n): n
Position (type a number): 1
Display in new line (y/n): n

Add another field for /context/other_context[at0001]/items[at0014] (y/n): n

Add a child-node for /context/other_context[at0001]/items[at0014] (y/n): y

Select a type for the new node (C)ontainer/(F)ield: f

>> ArchetypeFieldReference attributes
Possible path values for field node :
  >1: /context/other_context[at0001]/items[at0014]/items[at0003] (RmTypeName ELEMENT)
  >2: /context/other_context[at0001]/items[at0014]/items[at0004] (RmTypeName ELEMENT)
  >3: /context/other_context[at0001]/items[at0014]/items[at0009] (RmTypeName ELEMENT)
  >4: /context/other_context[at0001]/items[at0014]/items[at0010] (RmTypeName ELEMENT)
  >5: /context/other_context[at0001]/items[at0014]/items[at0011] (RmTypeName ELEMENT)
  >6: /context/other_context[at0001]/items[at0014]/items[at0012] (RmTypeName ELEMENT)
  >7: /context/other_context[at0001]/items[at0014]/items[at0013] (RmTypeName ELEMENT)
  Select a value: 4

Minimal occurrence (if empty, default value 0):
Maximal occurrence (if empty, default value 1):

Link with Zone from layout Layout1
  > z2
  > z3
  Select a value: z3

>>Control attributes
Name: CApellido_1
TypeCode for ctext (type a number): 1
NullFlavor (y/n): n
Position (type a number): 2
Display in new line (y/n): y
```

```
Add another field for /context/other_context[at0001]/items[at0014] (y/n): n

Add a child-node for /context/other_context[at0001]/items[at0014] (y/n): n

Add another ArchetypeContainerReference definition? (y/n): n

Add another view definition? (y/n): n

File Template1v1.0.xml has been created in templates/ directory
```

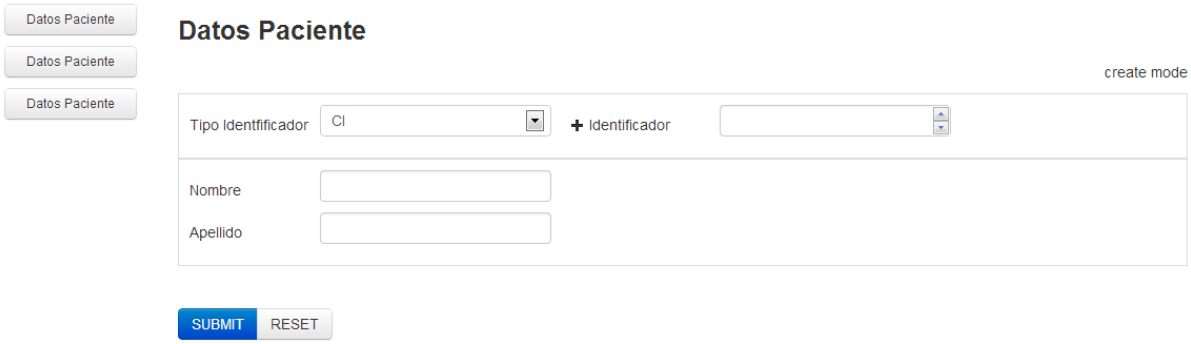
Paso 2) El DISM generará el siguiente archivo de extensión xml:

```
<UITemplate UITemplateId='Template1' Name='Template1' Version='1' Description='Datos paciente'
NavbarPosition='left' LayoutId='l1' GeneratorVersion='1.0'>

  <Layout LayoutId='l1' Name='Layout1'>
    <CompositeZone ZoneId='z1' Orientation='horizontal'>
      <LeafZone ZoneId='z2' />
      <LeafZone ZoneId='z3' />
    </CompositeZone>
  </Layout>

  <View ViewId='v1' Name='Datos Paciente' Description='Datos Paciente' Order='1' LayoutId='l1'>
    <ArchetypeContainerReference
        ArchetypeReference='openEHR-EHR-COMPOSITION.arquetipo_prueba_001.v1'
        Path='/context/other_context[at0001]/items[at0014]'
        Type='CLUSTER'>
      <ArchetypeFieldReference
        ArchetypeReference='openEHR-EHR-COMPOSITION.arquetipo_prueba_001.v1'
        Path='/context/other_context[at0001]/items[at0014]/items[at0004]'
        Type='ELEMENT' ZoneId='z2'>
                <Control ControlId='c1' Name='CTipoIdentificador_1' Type='ccodedtext'
                TypeCode='1' NullFlavor='0' Position='1' Skip='0'
                AttributePath='/context/other_context[at0001]/items[at0014]/items[at0004]/value'
                />
      </ArchetypeFieldReference>
      <ArchetypeFieldReference
        ArchetypeReference='openEHR-EHR-COMPOSITION.arquetipo_prueba_001.v1'
        Path='/context/other_context[at0001]/items[at0014]/items[at0003]' Type='ELEMENT'
        ZoneId='z2'>
                <Control ControlId='c2' Name='CIdentificador_1' Type='ccount' TypeCode='1'
                NullFlavor='0' Position='2' Skip='0'
                AttributePath='/context/other_context[at0001]/items[at0014]/items[at0003]/value'
                />
      </ArchetypeFieldReference>
      <ArchetypeFieldReference
        ArchetypeReference='openEHR-EHR-COMPOSITION.arquetipo_prueba_001.v1'
        Path='/context/other_context[at0001]/items[at0014]/items[at0009]'
        Type='ELEMENT' ZoneId='z3'>
                <Control ControlId='c3' Name='CNombre_1' Type='ctext' TypeCode='1'
                NullFlavor='0' Position='1' Skip='0'
                AttributePath='/context/other_context[at0001]/items[at0014]/items[at0009]/value'
                />
      </ArchetypeFieldReference>
      <ArchetypeFieldReference
        ArchetypeReference='openEHR-EHR-COMPOSITION.arquetipo_prueba_001.v1'
        Path='/context/other_context[at0001]/items[at0014]/items[at0010]' Type='ELEMENT'
        ZoneId='z3'>
                <Control ControlId='c4' Name='CApellido_1' Type='ctext' TypeCode='1'
                NullFlavor='0' Position='2' Skip='1'
                AttributePath='/context/other_context[at0001]/items[at0014]/items[at0010]/value'
                />
      </ArchetypeFieldReference>
    </ArchetypeContainerReference>
  </View>
</UITemplate>
```

Paso 3) Ejecutamos el GISM pasándole por parámetro en nombre del archivo xml generado en los pasos anteriores (Template1v1.0.xml) y configuramos el generador para generar la interfaz de usuario en html (generator.config).

El generador validará el template con su archivo de validación (modeloXSD_v1.3) y generará la IU en el directorio artefacts/, la cual consistirá en una pantalla con la barra de navegación a la izquierda que contendrá tres botones correspondientes a la View definida en sus tres modos (create, edit y show) y un visor de Views a la derecha de la pantalla.

Ilustración H. 2: Ejemplo de UI creada por el Generador (HTML)



Errores Conocidos

DISM

- No se pide el ingreso de los datos correspondientes a la entidad FieldLabel.

GISM

- No se valida que las variables de sustitución y globales utilizadas en los archivos de mapeo sean válidas.
- No se valida que en el archivo de mapeo estén obligatoriamente todas las variables de sustitución definidas.

Anexo H. Entrevista a Gastón Milano

Entrevista realizada en el mes de abril, a Gastón Milano, CTO de Artech Consultores, para indagar en los principios básicos de la generación de código.

¿Qué opinas sobre los pasos que pensamos seguir para lograr la generación de interfaces en diferentes tecnologías?

La línea de generación de código, en general tiene ese *approach,* que parte de un modelo independiente de la plataforma (PIM: *platformindependentmodel*) donde se define algo que para uno es independiente. No tengo que saber de ese algo. Por

ejemplo, ustedes quieren desarrollar una herramienta para diseñar pantallas médicas, sin tener conocimiento de todos los conceptos clínicos.

Entonces, lo primero a definir es, ¿cuál es el modelo independiente de la plataforma que van a usar? ¿Luego, sobre ese modelo independiente, cual es el tipo de lenguaje que van a utilizar? ¿Es un lenguaje visual o un lenguaje textual? ¿Qué herramientas van a utilizar para construir ese designer? Y por último, cómo paso del PIM al PEM (*platformespecificmodel)?* Ahí es donde viene la parte de los generadores. El generador que pasa de un modelo abstracto, a un modelo especifico, que es lo que termina siendo artefactos de código de diferentes lenguajes. Ese es un *approach* totalmente valido y estándar. En el medio de ese pasaje, puede haber numerosas etapas. Ahí depende de cuánto tiempo tengan, lo que van a poder hacer. Y depende de a donde quieran llegar, y cuantas transformaciones quieran hacer del modelo; si vale la pena hacerlo de un paso o N. Por ejemplo, ¿quieren realmente tener un generador muy avanzado en cuanto a cómo modela el código que generan? Para lograr esto, el generador tiene que ir pasando por diferentes etapas. Por ejemplo, se pasan de un modelo, a un modelo específico, pero a su vez ese modelo lo pasa a otro árbol que no tiene bien armada la sintaxis, después lo transformo para sacarle el código muerto, chequear sintaxis, o imprimirlo lindo etc. , hasta que en un momento decido imprimirlo a fuente. Sin embargo, muchas veces, esos pasos en el medio los salteas. Y pasas de tu modelo original, directo a código fuente. Desde el punto de vista del proyecto parece que eso seguro es el camino que hay que seguir. O sea, les conviene plantear estas posibilidades y en la documentación fundamentar que por razones de tiempo se decide generar en un paso, y después quedará en un futuro para mejorar.

Una cosa que deben definir es cuál va a ser conceptualmente el modelo que van a capturar del usuario final o de sus usuarios. Sin hablar de lenguajes, decir que queremos capturar. Estas capturas pueden ser a nivel gráfico, como el diseñador que ustedes plantean, o un lenguaje natural, o ambos.

Nosotras queremos que esta herramienta la maneje un usuario sin experiencia. Pero sí con experiencia en el dominio. Como por ejemplo un doctor que no tiene experiencia en programación.

A la persona que va a usar la herramienta se le llama *domainespecificexpert*. Esa persona es experta en el tema en el que se basa el dominio de la herramienta. Ahora, para ese experto también muchas veces también el lenguaje textual es mejor. No es que sea programación. Pero eventualmente para el médico es mejor escribir algo que estar haciendo "*drag and drop"* de cosas. Hay que ver qué es lo que se desea capturar. Porque, por ejemplo, si voy a recetar algo, capaz que es mejor un lenguaje natural. Sin duda, no creo que haya una bala de plata para todo. Ni los diseñadores son buenos para todo, ni textual es bueno para todo. Una mezcla de ambas técnicas puede ser una buena solución. Si lo que yo voy diseñar es una pantalla, lo mejor es claramente un diseñador. Pero si a su vez después le tengo que indicar alguna acción que tiene que hacer ese formulario, ahí es mejor un lenguaje natural.

Según lo que me cuentan ustedes sobre lo que se imaginan de la herramienta, lo que van a estar armando son formularios dinámicos. Si es eso, y no hay una acción de enviar a algún lado o hacer algo, es correcto que se haga una herramienta gráfica como el diseñador que tienen pensado. Un formulario tiene varias partes. Por una parte los datos y que datos están en el formulario. Y luego está el *layout* de eso: ¿cómo se va a organiza esa información en el formulario? Eventualmente también podemos tener metadata de información; de si tal campo es requerido no, etc. Otro punto que tienen los formularios es el tema de la acciones. Por ejemplo, yo quiero cargar con ciertos

datos tal campo, y quiero apretar un botón, y ese botón tiene que tomar alguna acción o llamar a otro formulario. ¿Voy a poder llamar a un formulario desde otro formulario? ¿Cómo voy a hacer la navegación? La parte de navegación la puedo hacer visualmente también. Para esto es necesario definir acciones. Con esto de las acciones lo que pasa por lo general, es que empezás por definir algunas y después empezás a tener varias porque te empiezan a pedir cosas. La primera y más básica que va s a tener es un link a otro formulario. Sí, todo esto hasta ahí es visual. A no ser que empiecen a meterse en lenguajes de acciones entre cosas, que ahí se empieza a poner más escabroso para seguir forzando al diseñador gráfico a hacer cosas que en general son mucho más fácil hacerlas de otra forma.

¿Cómo se modela este proceso en GeneXus?

Es bastante sencillo. Tenemos el modelo, y en el modelo tenés que definir antes que nada conceptualmente que conceptos vas a manejar. Por ejemplo, hablando del formulario, decir: en el formulario tenemos los datos, *layout* y acciones. Dentro de los datos va a estar el dominio de los datos, que como vos decías, ya están preestablecidos. Todo eso es conceptual, y hasta ahí nadie dice cómo vas a guardarlo, por ejemplo. Hay que establecer bien caro cuales son todos los componentes del modelo. Al modelo lo estableces totalmente abstracto de almacenamiento y de implementación. Una vez cumplida esta etapa, imaginamos que ya tenemos eso hecho. En su caso, en designer. Entonces la siguiente pregunta es: ¿a qué quiero llegar como resultado final? Ustedes dirán, por ejemplo, quiero llegar a una aplicación para web y a una para *smartdevices*. Ahí llegamos a lo tangible, a lo específico, de lo que quiero llegar, y sé de donde quiero partir. Ahora sí hay que empezar a hacer el camino y decir: vamos a llevarlo a la realidad pragmática de cómo bajo esto. ¿Cómo voy a pasar de A a B? ¿Qué camino tengo que hacer? ¿Cómo tengo que comunicarme con otro? ¿Tengo que tener un dialogo con algún lenguaje? Ahora sí tengo que definir cómo voy a almacenar ese modelo en algo. El modelo puede estar almacenado en infinidad de cosas, una base de datos, un archivo, etc. En *GeneXus*, al modelo lo almacenamos en una base datos, con un modelo universal, llamado base de conocimiento, de una forma específica. Es una base de datos con ciertas estructuras. Por lo general, no se decide en la base de datos. Por un tema de simplicidad lo que se hace es serializarlo y guardarlo en una estructura (un *XML*, un *json*, etc.). Otra cosa que hay que tener en cuenta es que siempre se trata de no ser redundante en la información que se almacena. Los mismos conceptos de tratar de reutilizar y no poner en cada lugar todo lo repetido. Es básico el tratar de hacer de que quede bien separado y no todo redundante en los diferentes lugares, porque después te trae problemas en la transformación o inconsistencias de lo que generas.

Ahora imagínense que de aquello conceptual que tenían, ahora ya tienen un archivo. Por ejemplo, tienen un formulario almacenado con toda la información que necesitan. Nosotros en *GeneXus* tenemos de todo. En realidad, como hacemos varias transformaciones en el medio, nosotros partimos de un modelo que está en una base de conocimiento (base de datos), de ahí generamos cierta especificación que está en *Prolog* (que es como si fuera otro modelo), que lo va a tomar otro y va a empezar a generar a partir de eso, y a su vez, para los dispositivos móviles, antes que nada, generamos metadata en *json* que después toma el generador de pantallas. Puede haber varios pasos, pero para simplificar, podemos decir que para su caso no creo que vayan a necesitar tantos. Ese es un poco el camino en total y es más o menos lo que hace *GeneXus*. Luego ese camino empieza a tener sus variantes según lo que a cada proyecto

le quede mejor. Por ejemplo, vos podes decir, en mi modelo no quiero escribir un *json* porque en realidad quiero que también se pueda leer más fácil. Si eso es un objetivo en sí mismo, tenés que elegir cierta herramienta. Sí ese no es un objetivo, no te importa si está en *json* o en jeroglíficos. Puede ser lo que quieras. Ahí empiezan a decidir en cada etapa que herramienta es la que mejor y se adecua para los objetivos que tienen.

¿Qué aspectos deberíamos tener en cuenta para poder desarrollar en diferentes tecnologías?

Tenés que conocer de la tecnología. No se cuales *target* tienen ustedes. El de moda hoy podría ser *html5*. Podría ser su primer objetivo ya que es *crossplatformcrossdevice*.Con eso ya cubren todos los dispositivos. Y se podría llegar a generar que sea adaptable a cada uno. No la misma pantalla, sino diferentes formularios para cada uno. Después para cada dispositivo, ¿qué sería lo mejor de hacer? Habría que generar lo nativo de cada dispositivo. *ObjetiveC* para *iOS*, *Android* para los dispositivos de *Google* (que es *Java),* o *Java* en *Blackberry*. Ahora, ¿qué es lo que tienen que saber de eso? Tiene que estudiar la tecnología. La técnica para generar código es ir y escribir a mano a lo que yo quiero llegar. No hay una técnica que, sin saber a lo que quiero llegar, sin haberlo hecho, genera código. Para generar hay que saber bien a lo que se quiere llegar. Nosotros utilizamos ese método de escribir a mano la pantalla que queremos generar y siempre estamos descubriendo que cosas faltan. Veo aplicaciones que yo con *GeneXus* no las puedo hacer, y digo, lo que tengo que hacer es en el modelo, incorporar esta funcionalidad. Tratar de capturarla de la forma de ir incorporándole cosas. Yo puedo decir: mi primer objetivo es llegar a esto; un formulario escrito para abajo, etc. Vas y lo escribís en *html5*. Después de eso, tengo que generarlo. Y ahí empezás a hacerte bien la imagen de que es exactamente lo que tenés que tomar, capturar y a lo que vas a llegar.

Los generadores se construyen en base a dos cosas. Una capa de generación de código, pero abajo una base de cosas que ya están escritas porque son genéricas (nosotros le llamamos clases estándar). Por ejemplo, para transformar un *string* a *date*, no lo voy a generar. Agarro y ya tengo la clase escrita, y el código generado la llama. Cuanto más tenga de base, y genere la menor cantidad de código, tengo varias ventajas. El tiempo de generación va a ser menor y también la cantidad de código escrito. Además voy a estar reutilizando código que se utiliza en diferentes aplicaciones y esto lleva a que va a estar muy probado y no voy a tener tanta casuística de diferentes casos dependiendo de cómo lo generaba. Todo lo que sea código, que va a ser para todas las aplicaciones iguales, lo dejan ya escrito, y después generar todas esa cosas que varían. Todo lo que varía, lo generan. Y todo lo que sea constante, lo dejan constante ya escrito. Ustedes estarían haciendo una forma de generación *oneway*. Generan y pueden romper todo, y vuelven a generar y reconstruyen todo.

Nosotras vimos que para algunas tecnologías, existen un tipo de XML que si nosotros convirtiéramos nuestro propio tipo de XML, en ese formato específicode la tecnología, la tecnología se encarga de agarrar eso generar código. Ejemplo ZUL.

Ahí lo que estás haciendo es transformando un lenguaje. Vos para llegar a tu tecnología final, en realidad, si hay alguna herramienta que te da un lenguaje que a su vez después genera, está perfecto. Hay que parase en los hombros del más alto. El ZUL no lo recomiendo porque es una cosa que ya está muerta y no te va a andar para

lo que vos querés. También podes elegir generar *GeneXus* y *GeneXus* genera la aplicación. Como habíamos mencionado, nuestro modelo es en la base de datos; pero tiene un modelo de intercambio que es un *xml*. Si vos escribís el *xml* que nosotros agarramos y después creas la base de conocimiento con ese *xml* y después, mandas a generar, te va a genera una aplicación *html*, otra O*bjetive C*, otra *Android* y otra *Blackberry*.

Tiene que definir hacia donde quieren llegar. No tienen por qué llegar directamente, pueden llegar indirectamente generando el lenguaje de alguna otra herramienta que a su vez genere lo que desean. Eso es válido. Tienen que evaluar qué cosas hay e investigar sobre la viabilidad en el largo plazo de eso. Si el que desarrolla esa herramienta que van a utilizar, es una persona que lo hizo por "chiviar" y que capaz que no lo va a mantener, estás matando el proyecto antes de arrancar. Tienen que mirar muchas herramientas y definir primero que nada que *targets* de tecnologías/dispositivos van a tener. Si realmente son para el proyecto todos esos targets… porque es una cosa bastante grande. Yo creo que lo más viable, porque es una transformación que en algunos aspectos va a ser uno a uno, es el concepto de html. Porque en el formulario vas a tener una tabla, y en el html está la tabla. En el formulario tenés un campo y hay un equivalente… no van a tener q hacer una transformación muy sofisticada para obtenerlo y podes mostrar que funciona. Después obviamente tendrás que mostrar en la tecnología, que tu herramienta, con la que generas código, es independiente del html y mostrar que en general podes generar otras cosas que diferentes a html.

¿Qué opinas de las aplicaciones de escritorio?

Para escritorio puede llegar a ser el formulario mismo en web. Hoy por hoy, la interactividad que tenés con web no deja nada que desear comparada con las de aplicaciones de escritorio. La otra alternativa cercana en Windows 8 es con html5. La aplicación nativa de escritorio se genera con html5.

En realidad, al final lo que va a pasar es que seguramente estas cosas lo van a estar usando en otros dispositivos que no sean el de escritorio. Y si es en escritorio, lo están usando como está Summum, con la pantalla del explorador, y usan ahí el formulario, y la interactividad está perfecta. Yo no me mataría mucho. En mi opinión para escritorio estaría perfecto hacerlo con html y eventualmente darle la posibilidad de un trabajo offline del propio html, y con eso está más que suficiente.

La idea de nuestro proyecto es hacer una herramienta open source. Parados en eso, te queríamos preguntar si este proyecto es posible hacerlo en GeneXus.

Tu proyecto sí lo vas a poder hacer open source. Lo que pasa es que el que lo va a usar eventualmente va a tener que comprar *GeneXus*. Yo he hecho proyectos open source arriba de Windows y ahí tienen que tener Windows, y Windows no es gratuito. Cada una de las cosas que desarrollamos podemos elegir hacerlas open source porque es decisión de uno. Pero eso no significa que el que va a utilizar lo que vos ofreces, le salga gratis. Cualquier herramienta open source puede tener requerimientos de cosas que utiliza. Si lo que se quiere es que el costo sea cero para el usuario final, no vas a tener que usar ni *GeneXus* ni Windows, etc.
Por ejemplo, yo podría ofrecerle esta herramienta a Summum, y Summum tiene que comprar una licencia *GeneXus* para poder generar para diferentes dispositivos. Pero no para todos sus usuarios. Los usuarios finales nunca se enteran. Para ellos es gratis.

Pero si lo que se busca es que sea totalmente gratuita, ahí es claro que tenés que ir a un conjunto de herramientas que sean “open” ellas mismas para que funcione arriba de cualquier plataforma sin tener que pagar nada. Eso es básicamente lo que deberían hacer. Me suena lo más coherente en base a eso.

¿Qué van a usar para generar? ¿Han mirado algo?

Yo les recomiendo que miren ANTLR y ExtremeTemplate También está XText donde se definen reglas de cómo pasar de un lenguaje a otro lenguaje y con un parser van transformando el formato que definieron para capturar la pantalla que quieren crear, hasta llegar a un HTML, por ejemplo.

Otra alternativa, un poco más pragmática y capaz que más fácil para no tener que hacer el parser, es: ese xml, lo levanto a un modelo de objetos. Por ejemplo, el modelo de objetos, si son campos, va a ser: “formulario” que adentro tiene un array de “campo”, que adentro tiene “dato”. Entonces, lo leo desde archivo y me construyo ese modelo; y ahora tengo un modelo en memoria. Y a partir de eso que tengo en memoria, se lo paso a lo que se le llama un “template”. Que es un template? Yo quiero, por ejemplo, generar un html que tiene que decir: html, body, tabla... y seguramente para cada tabla van a querer escribir el tr, td, tr, td, que básicamente es una fila. Esa parte es la variable. Una vez definido eso, vos escribís un template, que es que va a generar el código. En el template escribís todo menos la parte de las filas de la tabla, la parte variable. Eso no lo escribís, pero escribís todo el esqueleto. En la parte de las filas escribís algo así como “$form.campos:createRows()$”. Luego, definís esa función createRow que lo que hace es agarrar un campo de esos y escribe para cada campo “label”, en vez de poner, por ejemplo, presión (el dato no variable) se pone genérico de la siguiente manera:

```
<label>$it.caption$</label>
<input/> ...
```

Escribo exactamente lo mismo que escribiría en el código html y las cosas que son variables las tomo de los datos que me van a venir como input a este template.

En resumen, ustedes capturaron la info de un archivo json u otro formato, y lo pasan al modelo de objetos, que básicamente es una class. Luego crean una instancia de esta variable en memoria, le van cargando los campos, y después llaman al template este, pasándole como parámetro, una variable de esta clase. Por último genero. Con esto lo que van haciendo es poniendo ciertas reglas basadas ya en los datos de cómo se genera cada una de las partes.
También existen otras tecnologías, como TForm o Velocity, que hacen exactamente lo mismo. Yo miró a que es lo que quiero llegar y lo escribo. Cuando me enfrento a una parte variable, lo saco. Es como dejar una página con huecos. Y los huecos los voy a llenar con los datos que me los dan de parámetros. Es bastante intuitivo de ver qué es lo que voy a generar.

Anexo I. Entrevista Juan José Spinetti

Entrevista realizada a Ing. Juan José Spinelli, Jefe de desarrollo de K2BHealth, en relación al diseño de interfaces gráficas de usuario para registros clínicos.

A la hora de diseñar interfaces gráficas de usuario para registros clínicos, identifica algún patrón de diseño en ellas?

Sinceramente no identifico un patrón claro, la experiencia me dice que depende mucho del cliente para el cuál estás haciendo el sistema. Nos ha pasado que hicimos la pantalla de historia clínica para un cliente de acuerdo a lo que nos pidió, este cliente quedó súper conforme, y luego cuando utilizamos esa pantalla como Demo para otro cliente, nos pidió cambiar todo de lugar.

Lo que si tratamos de hacer siempre y puede llegar a considerarse un patrón en la historia clínica, es tener un cabezal con la información que se quiere que siempre esté visible, y luego un conjunto de pestañas que agrupan la información que está relacionada, como puede ser datos del examen físico, prescripción, etc.

Qué considera que un profesional de la salud espera al enfrentarse a dichas pantallas?

Desde mi punto de vista, los profesionales de la salud esperan que la pantalla acompañe el proceso de atención. Para que esto suceda, la pantalla tiene que ser pensada y diseñada conociendo el negocio y desde el punto de vista del médico. Existen sistemas de historia clínica que fueron pensados con una visión administrativa y no de registro clínico por lo que no acompañan el proceso de atención y por ende no tiene éxito en su uso, porque el médico no lo considera una herramienta de trabajo que ayude a la atención.

Cuando me refiero a acompañar el proceso de atención, me refiero a que la pantalla siga la metodología que el médico utiliza en la consulta. A modo de ejemplo, existe la metodología SOAP (Subjetivo, Objetivo, Aproximación diagnóstica, Planificación). El médico arranca con la parte subjetiva como lo son el motivo de consulta y la anamnesis, luego sigue con la parte objetiva que serían los controles clínicos y examen físico, luego realiza una aproximación diagnóstica asignando un diagnóstico presuntivo y por ultimo realiza la planificación, prescribiendo paraclínica, medicamentos y derivaciones.

No necesariamente se tiene que hacer esta metodología, pero creo que ejemplifica lo que quiero decir.

¿Percibe que es difícil para un profesional de la salud comunicar como le gustaría ver las pantallas?

En general sí. Desde mi punto de vista esto sucede porque de hecho los médicos no tienen formación en informática y no entienden de lo que están hablando. Hay casos particulares de médicos con orientación a la informática médica con los que sí se entiende, o por lo menos se entiende un poco más lo que pretenden.

Creo que en el proceso de entender lo que quiere el profesional de la salud, es importante que en el equipo del proyecto se cuente con un analista funcional como por ejemplo lo es Ximena Martin en nuestro equipo, ella es casi médica y es Líder de Test. Si bien es difícil conseguir una persona con ese perfil, al momento de mantener reuniones con los médicos, ayuda mucho a la comunicación y entendimiento.

Basado en su experiencia, una vez diseñada la pantalla, con qué frecuencia promedio debe pasar por un proceso de validación por el usuario antes de darle el ok?

Este es un número que no tengo del todo claro, hay veces que mostramos la pantalla solo una vez y hay veces que muchas más. Digamos que entre 2 o 3 veces sería el promedio. Quizás este es un numero bajo, pero hay que tener en cuenta que el equipo

nuestro viene hace años haciendo este trabajo, por lo que la experiencia ayuda a no cometer los mismos errores que al principio.