Universidad de Valladolid

E.T.S Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Generador de interfaces de usuario con JSON y Vaadin

Autor:
D. Abel Martínez Martín

Tutor:
D. Miguel Ángel Laguna Serrano

Resumen

El objetivo del siguiente proyecto es el desarrollo de un framework para la generación de interfaces gráficas a partir de objetos Json. Para ello, se utilizará como apoyo el framework Vaadin, con el cual se pueden desarrollar interfaces web de una manera sencilla. De esta manera, los Json que se proporcionen al framework serán "traducidos" a una interfaz Vaadin.

Se realizará una pequeña introducción de Vaadin para ver las características que nos proporciona, y hacernos una idea de las interfaces que podremos crear con el framework que se desarrollará en el presente proyecto.

En el siguiente documento se detallarán el análisis, diseño, las metodologías de planificación y un manual de uso para el desarrollador.

Índice

Índice de Figuras

Índice de tablas

Introducción

1. Introducción

En la actualidad el uso de internet es algo habitual en la vida de la mayoría de las personas. Cada día aparecen nuevas herramientas que nos ayudan a mejorar nuestra calidad de vida, ya sea en forma de entretenimiento, información, seguridad o cualquier otro ámbito que se nos pueda ocurrir.

A medida que el uso de internet se hace cada vez más común entre la población, los desarrolladores necesitan mejores técnicas y herramientas para poder crear productos que puedan llegar al máximo número de consumidores posibles.

1.1. Motivación

Con el incesante crecimiento de internet, las aplicaciones de escritorio están tendiendo a desaparecer, convirtiéndose en aplicaciones web. El auge de este tipo de aplicaciones se debe a que se puede acceder desde cualquier navegador web, y por lo tanto desde cualquier dispositivo, teniendo todo sincronizado en la "nube".

Uno de los factores más importantes para el éxito de estas aplicaciones, es la "cara" que se ofrece al consumidor, es decir, la interfaz gráfica. Una interfaz gráfica con una usabilidad sencilla y una estética moderna y cuidada atrae mucho más al usuario.

En la actualidad existe un número bastante grande de frameworks para facilitar la creación de interfaces gráficas al desarrollador, en este caso el que nos interesa es Vaadin. La ventaja de Vaadin, es que con un lenguaje tan común como Java, un desarrollador puede construir fácilmente la capa front-end de una aplicación web.

El problema de un framework como Vaadin viene a la hora de realizar cambios en la interfaz, ya que el desarrollador tiene que añadir nuevo código Java a la aplicación y por lo tanto compilar y volver a desplegar. La idea de este framework es que a partir de un objeto Json, almacenado por ejemplo en un fichero, se pueda declarar la disposición de una interfaz gráfica de Vaadin. De esta forma se podrá modificar la interfaz sin necesidad de compilar de nuevo la aplicación, incluso pudiendo realizar cambios en "caliente".

1.2. Objetivos

El objetivo de este proyecto es el desarrollo de un framework de generación de interfaces gráficas de Vaadin a partir de objetos Json, de esta manera será mucho más flexible y sencillo construir la capa front-end en aplicaciones web.

Con esto se pretende que el desarrollador tenga ciertas facilidades a la hora del desarrollo de aplicaciones que utilicen el framework Vaadin, siendo mucho más sencilla la generación y modificación de interfaces gráficas.

Debido a que la interfaz se genera en tiempo de ejecución a partir del objeto Json, estos objetos podrían estar incluso almacenados en Base de Datos. De esta manera se podría modificar la interfaz de la aplicación en tiempo de ejecución, incluso se podrían llevar los objetos Json de las vistas a un servidor independiente que se dedique a servir los Json a la aplicación.

1.3. Estructura de la memoria

A continuación se proporciona un pequeño esquema de los temas principales que se van a tratar en los siguientes capítulos del documento.

- **Introducción a Vaadin.** Un pequeño resumen del framework Vaadin, para adelantar lo que seremos capaces de construir con el framework descrito en el presente documento.
- **Planificación.** Desglose del tiempo necesario para llevar a cabo todas las etapas del desarrollo del framework a construir, detallando así las fases y la metodología de planificación de proyectos a utilizar.
- **Análisis.** Aspectos correspondientes a la fase de análisis de la Ingeniería de Software.
- **Diseño.** Aspectos correspondientes a la fase de diseño de la Ingeniería de Software.
- **Implementación.** Aspectos correspondientes a la fase de implementación de la Ingeniería de Software.
- **Pruebas.** Detalle de las distintas pruebas realizadas para la validación del framework.
- **Manual de desarrollador.** Un pequeño manual de ayuda al desarrollador para que este aprenda a usar el framework.
- **Conclusiones.** Conclusiones obtenidas a lo largo del desarrollo del proyecto.
- **Bibliografía.** Referencias utilizadas como apoyo y ayuda a la realización del framework.

Introducción a Vaadin

2. Introducción a Vaadin

Vaadin es un framework de desarrollo de aplicaciones web en Java, diseñado para facilitar la creación y mantenimiento de interfaces gráficas. Es un framework que se ejecuta en el lado del servidor, o también conocido como "server-side". Se ocupa de controlar la interfaz de usuario en el navegador y las comunicaciones AJAX entre el navegador y el servidor.

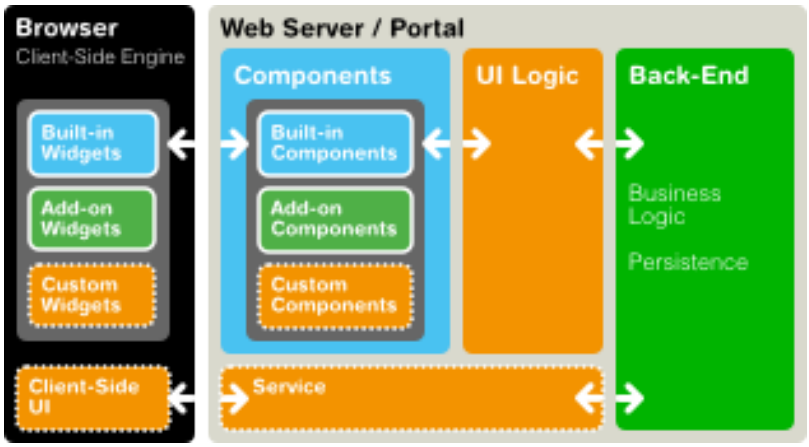

Figura 2-1. Arquitectura Vaadin

La Figura 2-1. Arquitectura Vaadin, nos muestra la arquitectura principal de Vaadin. La arquitectura está dividida en el lado de servidor o "server-side" y el lado del cliente o "client-side engine". El motor del lado del cliente se ejecuta en el navegador como código JavaScript, renderizando la interfaz de usuario y ocupándose del envío de la interacción de la interfaz con el servidor. La lógica de la interfaz se ejecuta como un Servlet Java en el servidor de la aplicación.

Como el motor del lado del cliente se ejecuta como JavaScript en el navegador, no es necesario ningún plugin para el correcto funcionamiento de las aplicaciones desarrolladas con Vaadin. Además este motor oculta al desarrollador las tecnologías relacionadas con el navegador, por lo que no hay que preocuparse por ninguna característica especial de los navegadores.

2.1. Componentes

El componente principal de Vaadin es la UI. La UI es el área visible para el usuario, y Vaadin asocia a cada UI una sesión de usuario y un estado de UI. De esta manera, una aplicación puede funcionar en distintos navegadores a la vez teniendo cada uno una instancia distinta de la misma UI. Cada UI de una aplicación va asociada a una URL, por lo que una misma aplicación puede estar compuesta por varias UI's.

Las interfaces de usuario en Vaadin están compuestas jerárquicamente, es decir, componentes dentro de componentes. Normalmente se coloca un componente de layout como el componente principal de la UI, y este se va rellenando con otros componentes. Por ejemplo:

```
UI
 `-- VerticalLayout
   |-- Label
   `-- HorizontalLayout
     |-- Tree
     `-- Table
```

Vaadin proporciona un catalogo bastante amplio de componentes, además de permitirte definir componentes propios. La Figura 2-2. Jerarquía de componentes, muestra la jerarquía de componentes existentes en Vaadin. Las interfaces se muestran en gris, las clases abstractas en naranja y las implementaciones en azul.

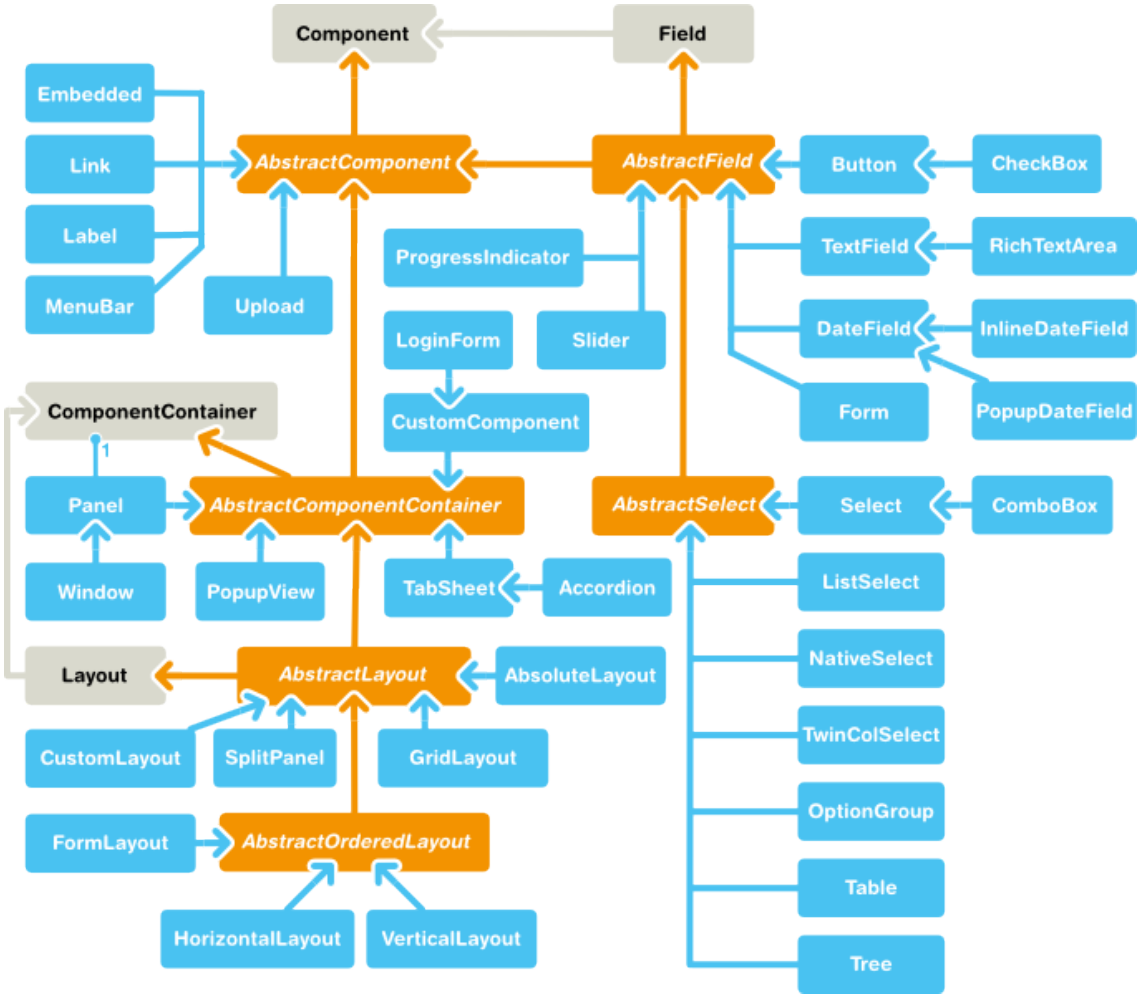

Figura 2-2. Jerarquía de componentes

Los componentes base proporcionan un gran número de características. Vamos a hacer un pequeño repaso de las características más usadas normalmente a la hora de desarrollar interfaces de usuario.

- **Caption.** Es una etiqueta identificativa del componente, normalmente situada encima, a la izquierda o dentro de este. Por ejemplo, el nombre de un botón.

- **Description,** o también conocido como Tooltip. Todos los componentes que heredan de `AbstractComponent` pueden tener una descripción. Esta descripción se muestra normalmente como un Tooltip, es decir cuando el ratón pasa por encima del componente.
- **Enabled.** Indica si el componente está activo o no. No quiere decir que sea visible o no, un componente desactivo es visible pero tiene apariencia de desactivado.
- **Icon.** Es una etiqueta gráfica aclaratoria, normalmente mostrada arriba, a la derecha o dentro del componente. Normalmente es mostrado en conjunto al caption.
- **Read-Only.** Indica si un componente es de solo lectura. Es una propiedad que se aplica a los componentes que heredan de `Field`. Por ejemplo se podría aplicar a un cuadro de texto para que este no sea modificable por el usuario.
- **Style Name.** Proporciona un nombre de clase CSS a el componente. Al indicarle un estilo propio, se puede modificar el componente por CSS.
- **Visible.** Indica si un componente es visible para el usuario.
- **Tamaños.** A cada componente se le puede indicar un ancho y una altura específica. Mas tarde hablaremos acerca de las unidades existentes y los distintos tamaños que se le pueden asignar a un componente.

2.2. Composición de la interfaz

Los componentes de Vaadin están claramente divididos en dos grupos: los componentes con los cuales el usuario puede interactuar y lo componentes de layout, para colocar otros componentes en lugares específicos de la interfaz de usuario.

La idea es sencilla, se empieza creando el contenedor para el diseño de la UI, y posteriormente se van añadiendo otros componentes de layout jerárquicamente. Finalmente se añaden los componentes de interacción como hijos de este árbol de componentes jerárquico.

Los componentes de layout más comunes son `HorizontalLayout` y `VerticalLayout`. Son diseños ordenados para la distribución de componentes horizontalmente o verticalmente, respectivamente. Por defecto, los `VerticalLayout` tienen un 100% de anchura y una altura indefinida. En cambio los `HorizontalLayout` no tienen tamaños definidos en ninguna de las dimensiones. Un uso típico de estos componentes es el siguiente:

```
VerticalLayout vertical = new VerticalLayout ();
vertical.addComponent(new TextField("Name"));
vertical.addComponent(new TextField("Street address"));
vertical.addComponent(new TextField("Postal code"));
  layout.addComponent(vertical);
```

Siendo este el resultado:

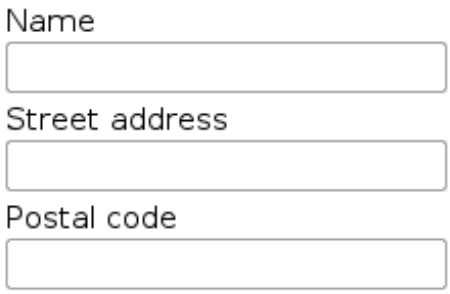

Figura 2-3. Ejemplo de VerticalLayout

Usando un `HorizontalLayout` el resultado sería el siguiente:

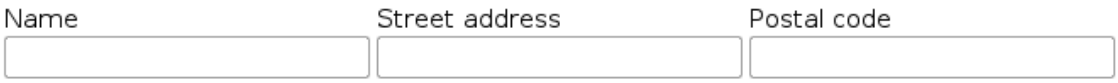

Figura 2-4. Ejemplo de HorizontalLayout

A parte de estos dos componentes de layout existen otros muchos: `GridLayout`, `FormLayout`, `Panel`, `HorizontalSplitPanel`, `VerticalSplitPanel`, `TabSheet`, `Accordion`, `AbsoluteLayout` y `CssLayout`.

2.2.1. Tamaños y espaciado.

La propiedad `spacing` permite añadir espaciado entre los elementos de un componente de layout. Para activar el espaciado basta con indicárselo al componente contenedor con `setSpacing(true)`. El tamaño del espaciado está definido por defecto en el tema de Vaadin, pero este tamaño puede ser modificado por CSS.

Todas los componentes de Vaadin tienen la propiedad `width` y `height` para indicar su tamaño. Para ello se utilizan los métodos `setWidth()` y `setHeight()`. Estos métodos reciben el tamaño como un valor float, pero además es necesario indicar la unidad. La lista de unidades está indicada en la Tabla 2-1. Unidades de tamaños.

Tabla 2-1. Unidades de tamaños

`Unit.PIXELS`	px
`Unit.POINTS`	pt
`Unit.PICAS`	pc
`Unit.EM`	em
`Unit.EX`	ex
`Unit.MM`	mm
`Unit.CM`	cm
`Unit.INCH`	in

`Unit.PERCENTAGE`	%

Los componentes que están contenidos en un layout ordenado pueden estar colocados de diferentes maneras dependiendo de la altura o anchura del componente de layout. A continuación se hace un repaso de las distintas opciones que se pueden dar.

Layout con tamaño indefinido

Si un `VerticalLayout` tiene una altura indefinida, o un `HorizontalLayout` tiene una anchura indefinida, el layout se reducirá para adaptarse a los componentes contenidos, de manera que no exista espacio extra entre ellos. El siguiente ejemplo muestra este comportamiento.

```
HorizontalLayout fittingLayout = new HorizontalLayout();
fittingLayout.setWidth(Sizeable.SIZE_UNDEFINED, 0); // Default
fittingLayout.addComponent(new Button("Small"));
fittingLayout.addComponent(new Button("Medium-sized"));
fittingLayout.addComponent(new Button("Quite a big component"));
parentLayout.addComponent(fittingLayout);
```

Figura 2-5. Ejemplo layout con tamaño indefinido

Layout con tamaño definido

Si se le indica una anchura a un `HorizontalLayout` o una altura a un `VerticalLayout`, los componentes son distribuidos para ocupar cada uno de ellos el mismo espacio.

Figura 2-6. Ejemplo layout con tamaño definido

Expandiendo componentes

A veces se desea que un componente ocupe todo el espacio que el resto de componentes dejan disponible. Para ello hay que indicarle que su tamaño sea de 100% y además aplicarle el expand-ratio con `setExpandRatio()`. El siguiente ejemplo muestra como aplicar expand-ratio a un componente.

```
HorizontalLayout layout = new HorizontalLayout();
layout.setWidth("400px");

// These buttons take the minimum size.
layout.addComponent(new Button("Small"));
layout.addComponent(new Button("Medium-sized"));

// This button will expand.
Button expandButton = new Button("Expanding component");

// Use 100% of the expansion cell's width.
expandButton.setWidth("100%");

// The component must be added to layout
layout.addComponent(expandButton);

// Set the component's cell to expand.
layout.setExpandRatio(expandButton, 1.0f);
parentLayout.addComponent(layout);
```

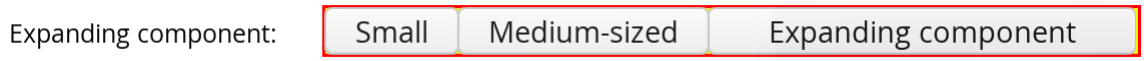

Figura 2-7. Ejemplo expand-ratio

2.3. Resumen

Este repaso de Vaadin ha servido para hacer una pequeña introducción a las interfaces de usuario que serán posibles realizar con el framework que se va a desarrollar y el cual es descrito en el presente documento.

A parte de las características descritas en esta introducción, Vaadin posee otras muchas que no se pueden resumir en este apartado. Para más información acerca de este framework tan completo es aconsejable consultar el libro oficial de Vaadin.

Gestión del proyecto

3. Gestión del proyecto

A lo largo de este apartado se detalla el proceso de planificación que se va a aplicar al desarrollo del proyecto. Después de analizar los distintos modelos de proceso de planificación de proyectos se ha decido utilizar el **Proceso Unificado**.

3.1. Proceso Unificado

El Proceso Unificado es un proceso iterativo e incremental, en el que se distinguen cuatro fases claramente diferenciadas:

- **Fase de inicio**. Se establece el ámbito y limites del proyecto. Se localizan los casos de uso críticos y se realiza la planificación temporal del proyecto.
- **Fase de elaboración**. Esta fase tiene como objetivo analizar el domino del proyecto y establecer una arquitectura base sólida. Es la parte más crítica del proceso, ya que a partir de aquí la arquitectura, los requisitos y los planes de desarrollo son estables.
- **Fase de construcción**. Durante esta fase se lleva a cabo la implementación de los casos de uso descritos durante la fase de elaboración.
- **Fase de transición**. La última fase de este proceso tiene como objetivo conseguir la aceptación por el usuario de que lo entregado es completo, y por otro lado, la elaboración de los manuales.

Cada una de las fases descritas anteriormente se dividen en iteraciones. Las iteraciones pueden considerarse como pequeños proyectos dentro del proyecto general. Su cometido es la evolución incremental del desarrollo de la fase de la que forman parte.

El número de iteraciones en las que se dividen estas fases viene establecido por la dificultad del proyecto. Este proyecto se ha caracterizado por ser de dificultad media, por lo que se aplicaran las correspondientes medidas de tiempo y esfuerzo que corresponden a los proyectos de dificultad media.

3.2. Gestión de riesgos

Uno de los puntos más importantes del Proceso Unificado es la identificación y mitigación de riegos. El estudio de los riesgos es una fase importantísima a la hora de la planificación de proyectos, ya que estos pueden llevar al fracaso el desarrollo de un proyecto.

La gestión de riesgos consiste en valorar y controlar los riesgos que afectan a un producto, proceso o desarrollo software. Es necesaria, ya que todos los proyectos poseen riesgos y alguno de ellos sucederá.

3.2.1. Identificación y análisis de los riesgos

Los riesgos en el desarrollo software se pueden categorizar en tres categorías principales:

- Riesgos de proyecto. Son aquellos riesgos que ponen en peligro al plan. Si estos se dan, el proyecto requerirá mayor esfuerzo y dinero.
- Riesgos técnicos. Estos riesgos ponen en peligro la calidad resultante del proyecto. Suelen hacer que el proyecto se convierta en más complicado de lo esperado.
- Riesgos de negocio. Ponen en peligro la realización del proyecto y podrían producir una cancelación del proyecto.

A continuación se listan los riesgos encontrados a la hora de la planificación del proyecto.

Tabla 3-1. Identificación de riesgos

Riesgo	Categoría	Probabilidad
Estimación temporal errónea.	Proyecto	Alta
Retraso en la realización del proyecto.	Proyecto	Alta
Realización de un diseño pobre inicial.	Técnico	Media
Tiempos altos en la transformación de los JSON a las vistas de Vaadin.	Técnico	Media
Problemas entre versiones de Vaadin.	Técnico	Baja

La probabilidad de riesgos se ha valorado como:

- Muy alta (<75%)
- Alta (50-75%)
- Media (25-50%)
- Baja (10-25%)

En las siguientes páginas se realiza un análisis más detallado de los riesgos listados en la tabla Tabla 3-1. Identificación de riesgos. También se expone el plan de contingencia para cada uno de ellos, por si alguno de los mismos se acaba produciendo.

Tabla 3-2. Riesgo Estimación temporal errónea

Formulario de Gestión de Riesgos 1			
Fase: Inicio	**Probabilidad:** 65%	**Consecuencia:** Alta	**Proyecto:** Generador de Layouts
Título del riesgo:	Estimación temporal errónea		
Valoración del riesgo			
Enunciado del riesgo:			
La estimación temporal es muy importante en la planificación del proyecto, pero una inexperiencia en este ámbito puede hacer que la planificación de las distintas fases y tareas se convierta en un trabajo difícil.			
Contexto del riesgo:			
Este riesgo podría ocurrir en la fase de Inicio, durante la planificación del proyecto.			
Análisis del riesgo:			
Una estimación errónea podría suponer un retraso en la realización de las tareas, por lo que supondría un aumento de las horas dedicadas a la construcción final del proyecto.			
Plan de riesgo			
Estrategia: X Prevención O Protección O Reducción O Investigación	**Plan de acción para el riesgo:** Para reducir al mínimo este riesgo, se realizará una planificación detallada de las distintas fases del proyecto.		

Tabla 3-3. Riesgo retraso en la realización del proyecto

Formulario de Gestión de Riesgos 2			
Fase: Todas	**Probabilidad:** 70%	**Consecuencia:** Alta	**Proyecto:** Generador de Layouts
Título del riesgo:	Retraso en la realización del proyecto		
Valoración del riesgo			
Enunciado del riesgo:			
Se pueden producir retrasos en la realización del proyecto debido a falta de tiempo para la realización de las tareas y el cumplimiento de las fechas acordadas en la planificación de este.			
Contexto del riesgo:			
Este riesgo podría ocurrir durante cualquier fase debido a la imposibilidad de trabajar en el proyecto por motivos laborales, personales o de enfermedad.			
Análisis del riesgo:			
Un retraso en el proyecto podría suponer el incumplimiento de las fechas acordadas en la planificación del proyecto. Esto podría desembocar en un retraso y la posibilidad de no acabar el proyecto a tiempo.			
Plan de riesgo			
Estrategia: X Prevención O Protección O Reducción O Investigación	**Plan de acción para el riesgo:**		
	Para reducir al mínimo este riesgo, se realizará una planificación de las fechas de los hitos del proyecto de acuerdo a las horas semanales que se pueden invertir en él.		

Tabla 3-4. Riesgo Diseño pobre

<table>
<tr><th colspan="4">Formulario de Gestión de Riesgos 3</th></tr>
<tr><td>Fase: Elaboración</td><td>Probabilidad:
35%</td><td>Consecuencia:
Alta</td><td>Proyecto: Generador de Layouts</td></tr>
<tr><td>Título del riesgo:</td><td colspan="3">Realización de un diseño pobre inicial</td></tr>
<tr><td colspan="4">Valoración del riesgo</td></tr>
<tr><td colspan="4">Enunciado del riesgo:</td></tr>
<tr><td colspan="4">El diseño software es una parte muy importante para el éxito de un proyecto. Un mal diseño inicial puede acarrear aplicaciones defectuosas, con un mal rendimiento o muy poco mantenibles y extensibles.</td></tr>
<tr><td colspan="4">Contexto del riesgo:</td></tr>
<tr><td colspan="4">Este riesgo podría ocurrir durante la fase de elaboración, cuando se realiza el análisis y diseño del framework a construir. Este riesgo se produciría por no haber realizado un análisis y un diseño detallado, como por ejemplo, debido a la no utilización de los patrones de diseño adecuados.</td></tr>
<tr><td colspan="4">Análisis del riesgo:</td></tr>
<tr><td colspan="4">Un mal diseño del framework a desarrollar podría conllevar problemas muy serios que derivarían en una mala implementación y por lo tanto en un funcionamiento incorrecto.</td></tr>
<tr><td colspan="4">Plan de riesgo</td></tr>
<tr><td colspan="2">Estrategia:
X Prevención
O Protección
O Reducción
O Investigación</td><td colspan="2">Plan de acción para el riesgo:
Para reducir al mínimo este riesgo, se realizará un diseño detallado, ayudándose de los patrones de diseño conocidos que sean útiles para la implementación del framework.</td></tr>
</table>

Tabla 3-5. Riesgo Tiempos altos en la transformación de los JSON

Formulario de Gestión de Riesgos 4			
Fase: Construcción	**Probabilidad:** 25%	**Consecuencia:** Media	**Proyecto:** Generador de Layouts
Título del riesgo:	Tiempos altos en la transformación de los JSON a las vistas de Vaadin		
Valoración del riesgo			
Enunciado del riesgo:			
El funcionamiento del framework consiste en la traducción de objetos JSON a vistas de Vaadin. Puede darse el caso de que no se consiga un rendimiento óptimo de estas traducciones y se generen retrasos en la generación de las vistas.			
Contexto del riesgo:			
Este riesgo podría ocurrir durante la fase de construcción. Se realizarán pruebas de rendimiento para comprobar el tiempo necesario para realizar traducciones de vistas complejas. Un malo rendimiento se podría dar por un diseño pobre del framework.			
Análisis del riesgo:			
Este riesgo produciría tiempos altos de generación de vistas, por lo que no sería un framework llamativo para su uso.			
Plan de riesgo			
Estrategia: O Prevención O Protección X Reducción O Investigación	**Plan de acción para el riesgo:** Para reducir al mínimo este riesgo, el framework se diseñará con el uso de los patrones de diseño adecuados, incluyendo paralelismo a la hora de la generación de las vistas si fuese necesario.		

Tabla 3-6. Riesgo Problema entre versiones

Formulario de Gestión de Riesgos 5			
Fase: Construcción	**Probabilidad:** 20%	**Consecuencia:** Media	**Proyecto:** Generador de Layouts
Título del riesgo:	Problemas entre versiones de Vaadin		
Valoración del riesgo			
Enunciado del riesgo:			
El framework a desarrollar se basa en el uso del framework Vaadin para la generación de las interfaces gráficas. Podrían darse incompatibilidades entre las distintas versiones de Vaadin.			
Contexto del riesgo:			
Este riesgo podría ocurrir durante la fase de construcción, por ejemplo, al ver que entre distintas versiones de Vaadin cambian los métodos utilizados o características que no se encuentran en versiones previas.			
Análisis del riesgo:			
Este riesgo produciría incompatibilidad con ciertas versiones de Vaadin, por lo que sería necesario utilizar las versiones de Vaadin compatibles.			
Plan de riesgo			
Estrategia: O Prevención O Protección O Reducción X Investigación	**Plan de acción para el riesgo:** Para reducir al mínimo este riesgo, se investigarán las versiones anteriores de Vaadin para hacer un estudio de compatibilidad.		

3.3. Recursos necesarios

Para llevar a cabo el desarrollo del framework son necesarios recursos hardware y software. A continuación se listan los recursos que van a ser utilizados para el desarrollo del presente proyecto.

Recursos hardware:

- **Ordenador**. Es una de las herramientas fundamentales, ya que para todo desarrollo software se necesita de un ordenador. En la siguiente tabla se describe el equipo a utilizar.

Tabla 3-7. Especificaciones del ordenador

Modelo	Macbook Pro Mid 2012
Procesador	2,5 GHz Intel Core i5
RAM	4 GB 1600 MHz DDR3
Disco duro	500 GB SSD
Sistema operativo	10.11 OS X El Capitan

Recursos software:

- **Eclipse**. Será el IDE (entorno de desarrollo integrado) que se utilice para la realización del framework. Se trata de una herramienta gratuita y compatible con multitud de sistemas operativos.
- **Office 365.** Se trata de un conjunto de aplicaciones del paquete Microsoft Office (Excel, Word, PowerPoint, Outlook y Access). Se utiliza una licencia gratuita proporcionada por la universidad.
- **Microsoft Project 2013.** Herramienta de administración de proyectos que se utilizará para le realización de la planificación. Se utiliza una licencia gratuita proporcionada por la universidad.
- **Astah Professional.** Herramienta de modelado UML necesaria para realizar el análisis y diseño del framework a desarrollar. Se utiliza una licencia gratuita proporcionada por la universidad.
- **Git.** Herramienta de control de versiones necesaria para el desarrollo del proyecto.
- **Bitbucket.** Repositorio git en la nube para la sincronización con el repositorio local. Se trata de un repositorio gratuito y privado.
- **Vaadin.** El framework base que se utiliza para el desarrollo del proyecto. Se trata de un framework gratuito.
- **Apache Maven.** Herramienta de software para le gestión y construcción de proyectos Java. Es una manera fácil y gratuita para la gestión de las dependencias de un proyecto.
- **Apache Tomcat 8.** Se trata de un servidor web de código abierto. Con este servidor se pueden desplegar los WAR generados.
- **Java 8.** Lenguaje de programación muy popular y el elegido para el desarrollo del framework. En este caso se ha decidido utilizar la última versión de este lenguaje, la versión 8.

3.4. Planificación

En esta sección se detalla la planificación del proyecto. Como ya se había comentado, se va a utilizar el Proceso unificado para realizar la planificación del framework a desarrollar. Debido a la utilización de este proceso, este apartado se dividirá en:

- Plan de fases.
- Planificación detallada de las fases e iteraciones.
- Seguimiento del proyecto.

3.4.1. Plan de fases

En este apartado se hace una calendarización de las fases del proyecto, indicando las fechas de inicio y fin de cada fase, y la duración de cada una de estas. Las duraciones vienen determinadas por la complejidad del proyecto. Al ser categorizado como un proyecto de dificultad media, el esfuerzo y tiempo necesario de cada fase se describe en la siguiente tabla.

Tabla 3-8. Duración y esfuerzo de las fases

Dominio	Inicio	Elaboración	Construcción	Transición
Esfuerzo	5%	20%	65%	10%
Tiempo	10%	30%	50%	10%

Una vez establecida la distribución de tiempo y esfuerzo, se define una calendarización para las mismas. Para esta calendarización se ha tenido en cuenta que hay 250 horas para la realización del proyecto, con un total de unas 30 semanas, y con una media prevista de trabajar 10 horas semanales. Las 50 horas restantes para llegar a las 300 horas serán utilizadas para la realización de la presente memoria.

Tabla 3-9. Calendarización de fases

Fase	Nº iteraciones	Duración	Fecha Inicio	Fecha Fin
Inicio	1	25 horas	14/11/2015	28/11/2015
Elaboración	1	75 horas	29/11/2015	17/01/2016
Construcción	2	125 horas	17/01/2015	16/04/2016
Transición	1	25 horas	17/04/2016	1/05/2016

3.4.2. Planificación detallada de las fases e iteraciones

A continuación se muestra la planificación detallada de las fases, indicando las tareas que se realizarán en cada fase junto con la planificación temporal de cada una de estas.

	Inicio	**25 horas**	**sáb 14/11/15**	**sáb 28/11/15**	
	Visión de objetivos	5 horas	sáb 14/11/15	sáb 14/11/15	
	Análisis inicial de Vaadin	7,5 horas	dom 15/11/15	sáb 21/11/15	2
	Análisis de riesgos	7,5 horas	sáb 21/11/15	dom 22/11/15	3
	Planificación	2 horas	sáb 28/11/15	sáb 28/11/15	4
	Calendarización	3 horas	sáb 28/11/15	sáb 28/11/15	5
	Elaboración	**75 horas**	**dom 29/11/15**	**dom 17/01/16**	**1**
	Análisis de Vaadin	10 horas	dom 29/11/15	sáb 05/12/15	
	Requisitos funcionales	10 horas	dom 06/12/15	sáb 12/12/15	8
	Requisitos no funcionales	10 horas	dom 13/12/15	sáb 19/12/15	9
	Casos de uso	10 horas	dom 20/12/15	sáb 26/12/15	10
	Diseño software	35 horas	dom 27/12/15	dom 17/01/16	11
	Construcción	**125 horas**	**dom 17/01/16**	**sáb 16/04/16**	**7**
	Iteración 1	**75 horas**	**dom 17/01/16**	**sáb 12/03/16**	
	Implementación de la traducción de objetos JSON a vistas Vaadin	55 horas	dom 17/01/16	sáb 27/02/16	
	Pruebas	20 horas	sáb 27/02/16	sáb 12/03/16	15
	Iteración 2	**50 horas**	**sáb 12/03/16**	**sáb 16/04/16**	**14**
	Implementación de la traducción de vistas Vaadin a objetos JSON	40 horas	sáb 12/03/16	sáb 09/04/16	
	Pruebas	10 horas	sáb 09/04/16	sáb 16/04/16	18
	Transición	**25 horas**	**sáb 16/04/16**	**dom 01/05/16**	**13**
	Finalización de la implementación	10 horas	sáb 16/04/16	sáb 23/04/16	
	Manual de desarrollador	15 horas	sáb 23/04/16	dom 01/05/16	21

Figura 3-1. Calendarización

3.5. Seguimiento

Una vez finalizado el desarrollo del proyecto se puede realizar el seguimiento de este. Para ello, vamos a comparar las horas y las fechas estimadas inicialmente, con las obtenidas durante el desarrollo del proyecto.

Tabla 3-10. Tabla de seguimiento del proyecto

Fase	Duración	Fecha inicio	Fecha fin
Inicio	30 horas	14/11/2015	29/12/2015
Elaboración	80 horas	30/12/2015	23/01/2016
Construcción	115 horas	24/01/2016	3/04/2016
Transición	15 horas	9/04/2016	20/05/2016

Como podemos observar después de comparar la duración, y las fechas de inicio y fin de las distintas fases, estas no coinciden con las planificadas en un inicio. Esta mala planificación, y los retrasos, se habían contemplado durante la gestión de riesgos. En concreto, los riesgos número uno, estimación temporal errónea, y número dos, retraso en la realización del proyecto.

La estrategia usada en el plan de contingencia que se eligió para estos riesgos fue de prevención, por lo que se había estudiado más detalladamente la planificación del proyecto. Aún intentado prevenir estos riesgos, no se ha conseguido evitarlos, por lo que se ha tenido que realizar un esfuerzo extra en cuanto a las horas trabajadas por semana para no demorar más la fecha de fin de este.

3.6. Estimación de costes

A continuación se realiza una estimación de los costes supone el desarrollo del proyecto. Esta estimación tiene en cuenta los elementos hardware, software y de personal que se necesitan para la finalización del proyecto.

En cuanto a la estimación de costes tanto hardware como software, se estima que el único coste a tener en cuenta es el del ordenador utilizado para el desarrollo, ya que no disponemos de ningún coste de tipo software debido a que todo el utilizado es gratuito, o con licencias proporcionadas por la universidad. El ordenador es un Macbook Pro de mediados de 2012, el cual tuvo un precio de unos 1000€.

El coste derivado del personal se va a calcular con los siguientes criterios:

- Un proyecto realizado por una única persona.
- El proyecto se ha estimado para 300 horas de trabajo.
- El precio por hora y persona estimado es de 24€, un precio utilizado entre la mayoría de empresas software españolas para programadores Junior.

Como resultado, la estimación de costes resultantes será la que se muestra en la siguiente tabla.

Tabla 3-11. Estimación de costes

Elemento	Coste
Ordenador	1000€
Personal	7200€
Total	8200€

Análisis

4. Análisis

El objetivo del análisis de un desarrollo software es adquirir un visión clara y bien definida del proyecto que se desea realizar. Para ello, uno de los puntos principales, es la identificación de todos los requisitos del desarrollo, tanto requisitos funcionales como no funcionales.

Es un fase esencial de cualquier proyecto, ya que de esta forma se pueden evitar problemas que puedan surgir durante el proceso de desarrollo. La realización de un análisis detallado y bien realizado es un signo de éxito a la hora del desarrollo software.

4.1. Requisitos funcionales

Como se decía anteriormente, uno de los principales objetivos del análisis es la identificación de requisitos. Para el desarrollo de este framework se han identificado los siguientes requisitos funcionales los cuales se describen a continuación.

FRQ-0001	**Creación de un VerticalLayout a partir de un JSON**
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear VerticalLayout's de Vaadin a partir de un objeto JSON.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-1. Requisito funcional 001

FRQ-0002	**Creación de un HorizontalLayout a partir de un JSON**
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear HorizontalLayout's de Vaadin a partir de un objeto JSON.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-2. Requisito funcional 002

FRQ-0004	Creación de un TabSheet a partir de un JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear TabSheet's de Vaadin a partir de un objeto JSON.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-3. Requisito funcional 003

FRQ-0006	Creación de un MenuBar a partir de un JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear MenuBar's de Vaadin a partir de un objeto JSON.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-4. Requisito funcional 004

FRQ-0007	Creación de un Button a partir de un JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear Button's de Vaadin a partir de un objeto JSON.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-5. Requisito funcional 005

FRQ-0008	Creación de un CheckBox a partir de un JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear CheckBox's de Vaadin a partir de un objeto JSON.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-6. Requisito funcional 006

FRQ-0009	Creación de un TextField a partir de un JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear TextField's de Vaadin a partir de un objeto JSON.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-7. Requisito funcional 007

FRQ-0010	Creación de un DateField a partir de un JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear DateField's de Vaadin a partir de un objeto JSON. También se permitirá crea su variante "PopupDateField".*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-8. Requisito funcional 008

FRQ-0011	Creación de un ComboBox a partir de un JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear ComboBox's de Vaadin a partir de un objeto JSON.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-9. Requisito funcional 009

FRQ-0012	Creación de un Table a partir de un JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear Table's de Vaadin a partir de un objeto JSON.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-10. Requisito funcional 010

FRQ-0013	Fijar tamaños a cualquiera de los componentes
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir asignar tamaños a cualquiera de los componentes que se van a poder traducir de JSON a objetos Vaadin. Se le permitirá indicar el tamaño deseado y la unidad del tamaño que se quiere aplicar.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-11. Requisito funcional 011

FRQ-0014	Asignar propiedades básicas a los componentes
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir asignar propiedades básicas a cualquiera de los componentes que se van a poder traducir de JSON a objetos Vaadin. Estas propiedades serán el "caption", "description", "icon", "styleName", "enabled", "readOnly" y "visible".*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-12. Requisito funcional 012

FRQ-0015	Fijar posiciones a los componentes
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir asignar posiciones a los componentes que estén contenidos dentro de componentes contenedores.*
Importancia	vital
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-13. Requisito funcional 013

FRQ-0016	Creación de un objeto JSON a partir de un HorizontalLayout de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un HorizontalLayout existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-14. Requisito funcional 014

FRQ-0017	Creación de un objeto JSON a partir de un VerticalLayout de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un VerticalLayout existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-15. Requisito funcional 015

FRQ-0019	Creación de un objeto JSON a partir de un TabSheet de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un TabSheet existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-16. Requisito funcional 016

FRQ-0020	Creación de un objeto JSON a partir de un MenuBar de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un MenuBar existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-17. Requisito funcional 017

FRQ-0021	Creación de un objeto JSON a partir de un Button de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un Button existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-18. Requisito funcional 018

FRQ-0022	Creación de un objeto JSON a partir de un CheckBox de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un CheckBox existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-19. Requisito funcional 019

FRQ-0023	Creación de un objeto JSON a partir de un TextField de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un CheckBox existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-20. Requisito funcional 020

FRQ-0024	Creación de un objeto JSON a partir de un DateField de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un DateField existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-21. Requisito funcional 021

FRQ-0025	Creación de un objeto JSON a partir de un ComboBox de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un ComboBox existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-22. Requisito funcional 022

FRQ-0026	Creación de un objeto JSON a partir de un Table de Vaadin
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *permitir crear un objeto JSON a partir de un Table existente de Vaadin.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-23. Requisito funcional 023

4.2. Requisitos no funcionales

NFR-0001	Validación del objeto JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *validar los objetos JSON, utilizados para crear los layouts, que se le pasen al framework.*
Importancia	importante
Urgencia	inmediatamente
Comentarios	Ninguno

Figura 4-24. Requisito no funcional 001

NFR-0002	Aviso de errores en el JSON
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *avisar con la mayor precisión posible de los errores que pueda poseer un objeto JSON que se le pase al framework.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-25. Requisito no funcional 002

NFR-0003	Tiempos de traducción aceptables
Versión	1.0 (30/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Dependencias	Ninguno
Descripción	El sistema deberá *traducir los JSON a vista de Vaadin en tiempos razonables para un adecuado uso y funcionamiento del framework.*
Importancia	importante
Urgencia	hay presión
Comentarios	Ninguno

Figura 4-26. Requisito no funcional 003

4.3. Casos de uso

Otra pieza fundamental del análisis de un proyecto software es la obtención e identificación de los casos de uso. Los casos de uso nos ayudan a describir el uso del sistema, y como los usuarios interactúan con este.

4.3.1. Actores

Debido a que se trata del desarrollo de un framework, el único actor que va a existir es el desarrollador, es decir, el usuario que va a usar el framework para crear interfaces gráficas para sus desarrollos de otras aplicaciones.

ACT-0001	**Desarrollador**
Versión	1.0 (31/12/2015)
Autores	• Abel Martínez Martín
Fuentes	• Abel Martínez Martín
Descripción	Este actor representa *el desarrollador que va a utilizar el framework para generar vistas.*
Comentarios	Ninguno

Figura 4-27. Actor desarrollador

4.3.2. Diagrama de casos de uso

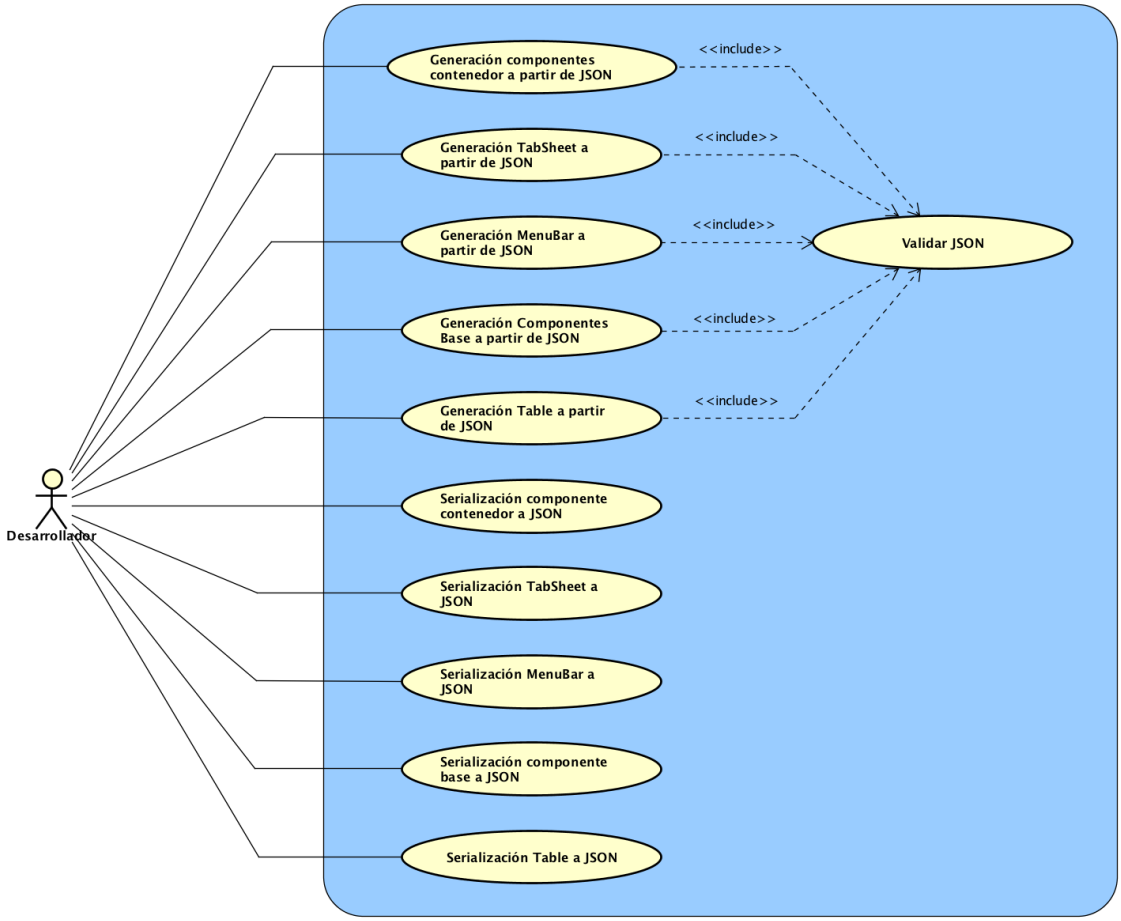

Figura 4-28. Diagrama de casos de uso

4.3.3. Descripción de los casos de uso

UC-0001	Generación de un componente contenedor de Vaadin a partir de un JSON	
Versión	1.0 (31/12/2015)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	• [UC-0007] Validar JSON	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee generar un componente contenedor de otros componentes. Estos son; "HorizontalLayout" y "VerticalLayout", los cuales permiten contener componentes de cualquier tipo.*	
Precondición	Previamente se ha tenido que realizar la validación del JSON, para ello, ejecutando el caso de uso [UC-0007]	
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *crea un objeto JSON para crear un componente contenedor de componentes. Este objeto puede ser de tipo "HorizontalLayout" o "VerticalLayout". El actor le asigna las propiedades que desee al objeto que va a crear.*
	2	El sistema *genera el componente contenedor de Vaadin que corresponde con el detallado en el objeto JSON.*
Postcondición	Se ha generado un objeto contenedor de componentes de Vaadin.	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	vital	
Urgencia	inmediatamente	
Comentarios	Ninguno	

Figura 4-29. Caso de uso 001

UC-0002	Generación de un TabSheet de Vaadin a paritr de un JSON	
Versión	1.0 (31/12/2015)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee generar un TabSheet de Vaadin. En el JSON se le indicará las propiedades que quiera que tenga el TabSheet, y se podrán añadir las pestañas que va a contener.*	
Precondición	Previamente se ha tenido que realizar la validación del JSON, para ello, ejecutando el caso de uso [UC-0007]	
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *crea un objeto JSON indicando que el tipo del componente es TabSheet. El actor le añade las propiedades que crea convenientes al TabSheet y si quiere, le puede indicar las pestañas que se van a mostrar.*
	2	El sistema *genera un TabSheet de Vaadin a partir del objeto JSON que le ha pasado el actor.*
Postcondición	Se ha creado un TabSheet de Vaadin.	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	importante	
Urgencia	hay presión	
Comentarios	Ninguno	

Figura 4-30. Caso de uso 002

UC-0003	Generación de un MenuBar de Vaadin a paritr de un JSON	
Versión	1.0 (31/12/2015)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee generar un MenuBar de Vaadin. En el JSON se le indicará las propiedades que quiera que tenga el MenuBar, y se podrán añadir los correspondientes menús y submenús que se deseen.*	
Precondición	Previamente se ha tenido que realizar la validación del JSON, para ello, ejecutando el caso de uso [UC-0007]	
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *crea un objeto JSON indicando que el tipo del componente es MenuBar. El actor le añade las propiedades que crea convenientes al MenuBar y le indica los menús y submenús que quiera que contenga.*
	2	El sistema *genera un MenuBar de Vaadin a partir del objeto JSON que le ha pasado el actor.*
Postcondición	Se ha creado un MenuBar de Vaadin.	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	importante	
Urgencia	hay presión	
Comentarios	Ninguno	

Figura 4-31. Caso de uso 003

UC-0004	Generación de componentes base a partir de un JSON	
Versión	1.0 (31/12/2015)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee generar un componente base de Vaadin. Estos componentes son; "Buttton", "CheckBox", "TextField", "DateField" y "ComboBox". Se les podrá indicar las propiedades que el actor considere necesarias.*	
Precondición	Previamente se ha tenido que realizar la validación del JSON, para ello, ejecutando el caso de uso [UC-0007]	
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *crea un objeto JSON indicando que el tipo sea uno de los descritos en la descripción. El actor le indica las propiedades que crea necesarias para el componente.*
	2	El sistema *el sistema genera uno de los componentes descritos acorde al JSON.*
Postcondición	Se ha creado uno de los objetos señalados en la descripción.	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	vital	
Urgencia	inmediatamente	
Comentarios	Ninguno	

Figura 4-32. Caso de uso 004

UC-0005	Generación de un Table de Vaadin a paritr de un JSON	
Versión	1.0 (31/12/2015)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee generar unTable de Vaadin. En el JSON se le indicará las propiedades que quiera que tenga el Table, y se podrán añadir las columnas que quiera que posea.*	
Precondición	Previamente se ha tenido que realizar la validación del JSON, para ello, ejecutando el caso de uso [UC-0007]	
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *crea un objeto JSON indicando que el tipo del componente es Table. El actor le añade las propiedades que crea convenientes al Table y le indica las columnas que quiera que posea.*
	2	El sistema *genera un Table de Vaadin a partir del objeto JSON que le ha pasado el actor.*
Postcondición	Se ha creado un Table de Vaadin.	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	importante	
Urgencia	hay presión	
Comentarios	Ninguno	

Figura 4-33. Caso de uso 005

UC-0007	Validar JSON	
Versión	1.0 (24/04/2016)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee validar un JSON que servirá para crear vistas. Todo JSON que se utilice para la generación de las vistas será previamente validado antes de intentar realizar cualquier acción de interpretación del JSON.*	
Precondición		
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *le indica al validador el JSON que se quiere interpretar.*
	2	El sistema *validará el JSON indicado. Si el JSON no es correcto, se indicará con la mayor exactitud posible la localización del error.*
Postcondición	Se obtendrá si el JSON es válido para su interpretación a vista o no.	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	importante	
Urgencia	inmediatamente	
Comentarios	Ninguno	

Figura 4-34. Caso de uso 006

UC-0008	Serialización de un componente contenedor de Vaadin a JSON	
Versión	1.0 (24/04/2016)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee serializar un componente contenedor de Vaadin a un JSON interpretable por el framework. Con este CU se permitirán serializar vistas de Vaadin, HorizontalLayout o VerticalLayout, existentes a JSON.*	
Precondición		
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *le indicará al serializador el HorizontalLayout o VerticalLayout que desea serializar*
	2	El sistema *serializará todas las propiedades soportadas por el framework incluyendolas al JSON resultante de la serialización*
Postcondición	El componente contenedor de Vaadin se habrá traducido a JSON	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	importante	
Urgencia	puede esperar	
Comentarios	Ninguno	

Figura 4-35. Caso de uso 007

UC-0009	Serialización de un TabSheet de Vaadin a JSON	
Versión	1.0 (24/04/2016)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee serializar un TabSheet de Vaadin a un JSON interpretable por el framework. Con este CU se permitirán serializar TabSheet's de Vaadin existentes a JSON.*	
Precondición		
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *le indicará al serializador el TabSheet que desea serializar*
	2	El sistema *serializará todas las propiedades soportadas por el framework incluyendolas al JSON resultante de la serialización*
Postcondición	El componente TabSheet de Vaadin se habrá traducido a JSON	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	importante	
Urgencia	puede esperar	
Comentarios	Ninguno	

Figura 4-36. Caso de uso 008

UC-0010	Serialización de un MenuBar de Vaadin a JSON	
Versión	1.0 (24/04/2016)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee serializar un MenuBar de Vaadin a un JSON interpretable por el framework. Con este CU se permitirán serializar MenuBar's de Vaadin existentes a JSON.*	
Precondición		
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *le indicará al serializador el MenuBar que desea serializar*
	2	El sistema *serializará todas las propiedades soportadas por el framework incluyendolas al JSON resultante de la serialización*
Postcondición	El componente MenuBar de Vaadin se habrá traducido a JSON	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	importante	
Urgencia	puede esperar	
Comentarios	Ninguno	

Figura 4-37. Caso de uso 009

UC-0011	Serialización de componentes base de Vaadin a JSON	
Versión	1.0 (24/04/2016)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee serializar componentes base de Vaadin a un JSON interpretable por el framework. Con este CU se permitirán serializar "Buttton", "CheckBox", "TextField", "DateField" y "ComboBox"de Vaadin existentes a JSON.*	
Precondición		
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *le indicará al serializador el componente base que desea serializar*
	2	El sistema *serializará todas las propiedades soportadas por el framework incluyendolas al JSON resultante de la serialización*
Postcondición	El componente base de Vaadin se habrá traducido a JSON	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	importante	
Urgencia	puede esperar	
Comentarios	Ninguno	

Figura 4-38. Caso de uso 010

UC-0012	Serialización de un Table de Vaadin a JSON	
Versión	1.0 (24/04/2016)	
Autores	• Abel Martínez Martín	
Fuentes	• Abel Martínez Martín	
Dependencias	Ninguno	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando *se desee serializar componentes Table de Vaadin a un JSON interpretable por el framework. Con este CU se permitirán serializar Table's de Vaadin existentes a JSON.*	
Precondición		
Secuencia normal	**Paso**	**Acción**
	1	El actor Desarrollador (ACT-0001) *le indicará al serializador el componente Table que desea serializar*
	2	El sistema *serializará todas las propiedades soportadas por el framework incluyendolas al JSON resultante de la serialización*
Postcondición	El componente Table de Vaadin se habrá traducido a JSON	
Excepciones	**Paso**	**Acción**
	-	-
Rendimiento	**Paso**	**Tiempo máximo**
	-	-
Importancia	importante	
Urgencia	puede esperar	
Comentarios	Ninguno	

Figura 4-39. Caso de uso 011

4.3.4. Modelo de dominio

Durante el análisis del proyecto, se llevó a cabo el modelo de dominio inicial que se va a describir en este punto. Se divide en dos partes, el modelo de dominio encargado de la transformación de los objetos Json a vistas de Vaadin y el modelo de dominio encargado de la serialización de objetos de Vaadin a Json. Se ha decido describir el modelo de domino por separado ya que se trata de partes sin conexión, por lo que nos facilita una mejor compresión de forma individual.

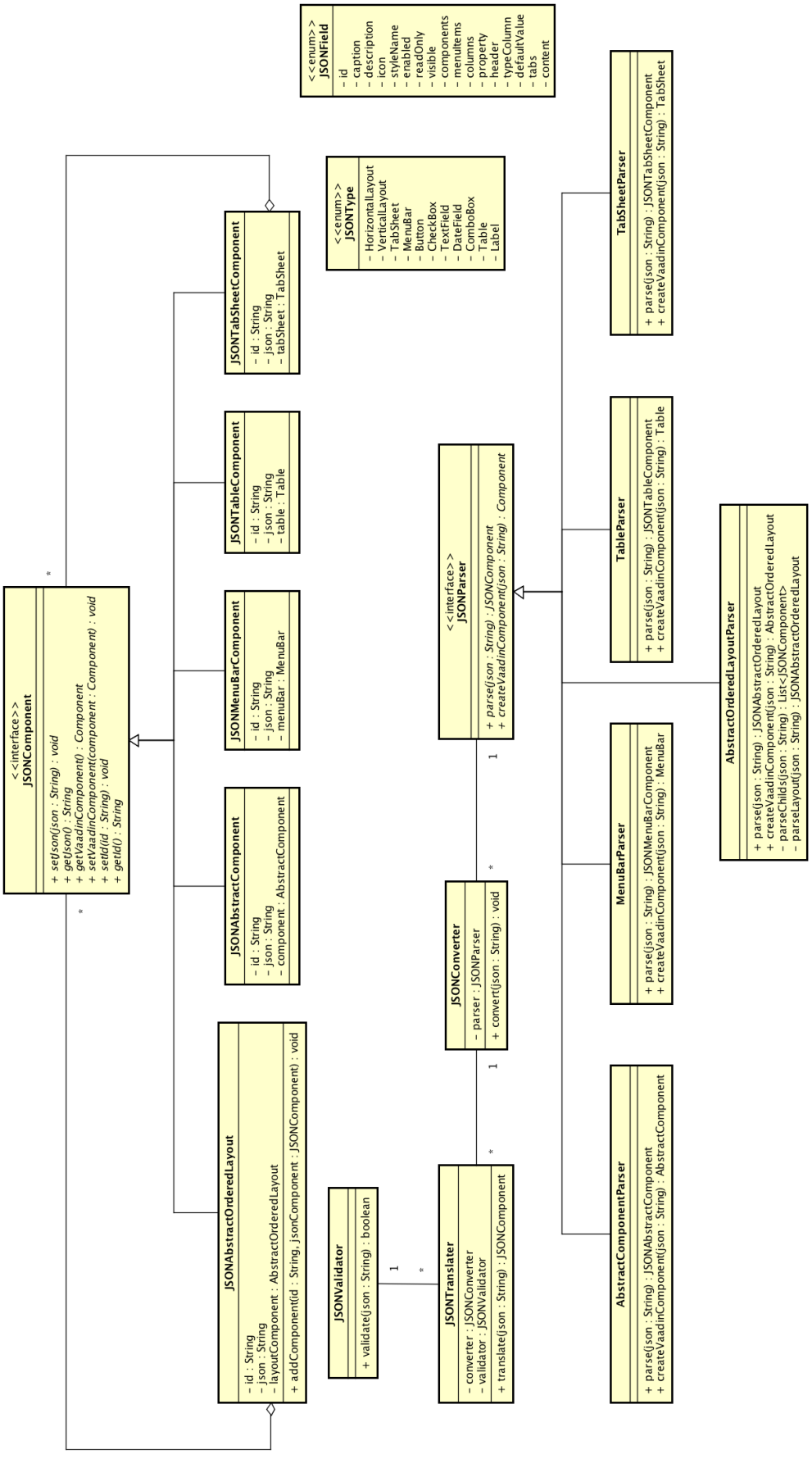

Figura 4-40. Modelo de dominio parte 1. Generación de vistas a partir de JSON.

Esta primera parte es la encargada de realizar la conversión de objetos Json a vistas de Vaadin. Como muestra el diagrama, todo componente que se quiera interpretar a partir de un Json debe implementar la clase `JSONComponent`. De esta manera facilitamos que los componentes tengan una interfaz común. Gracias a la interfaz se nos permite que para cualquier componente se pueda acceder al Json que lo forma, al componente Vaadin que corresponde y al identificador que este posee.

Los componentes `JSONAbstractOrderedLayout` y `JSONTabSheetComponent` son especiales y pueden contener a su vez cualquier tipo de componentes. Esto es debido a que se tratan de componentes contenedores.

La traducción está formada principalmente por cuatro clases, `JSONValidator`, `JSONParser`, `JSONConverter` y `JSONTranslater`.

- `JSONValidator`. Esta clase se encarga de realizar las validaciones de los Json a partir del Json Schema que se ha declarado para la validación.
- `JSONParser`. Se trata de una interfaz que define las operaciones necesarias para traducir cualquier tipo de componente. Todo componente que se quiera interpretar tiene que tener un `JSONParser` asociado para su correcta traducción.
- `JSONConverter`. Se encarga de obtener el `JSONParser` adecuado para el tipo de componente al que corresponde el Json que se quiere interpretar.
- `JSONTranslater`. Es el punto de entrada para la traducción de Json a componentes. Valida el Json correspondiente, y si este es correcto, le indica al `JSONConverter` que realice la traducción.

Como vemos, cada componente que se puede interpretar a partir de un Json necesita de un `JSONParser`. De esta forma, cuando el objeto Json le llega al `JSONConverter`, este obtiene el `JSONParser` adecuado a partir del tipo de componente.

Con este modelo es relativamente fácil añadir en cualquier momento la traducción de un nuevo tipo de componente. El nuevo componente deberá implementar la clase `JSONComponent` y se le deberá crear un `JSONParser` que realice su correcta interpretación.

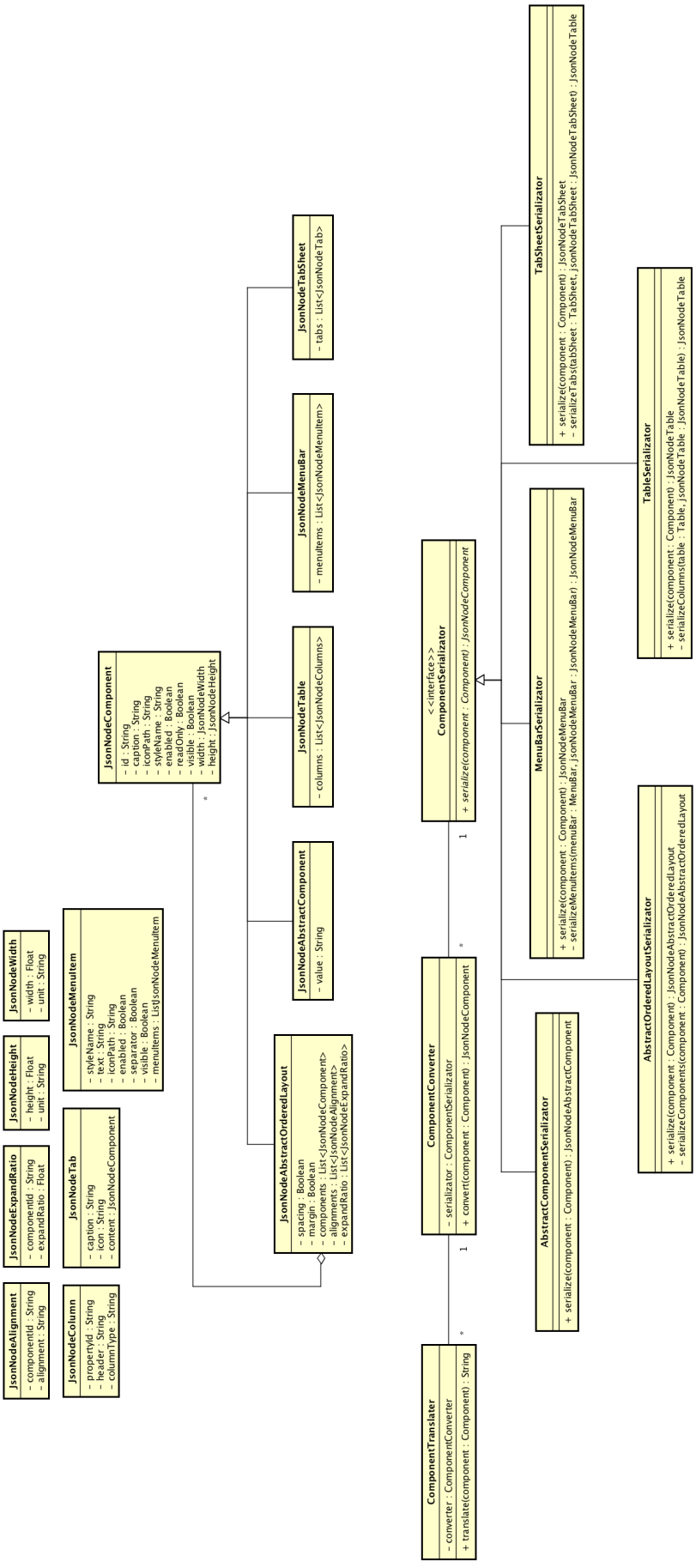

Figura 4-41. Modelo de domino parte 2. Serialización de componentes.

Esta segunda parte es la encarga de serializar un componente de Vaadin a un Json. Todo componente que es interpretable hereda de la clase `JsonNodeComponent`. Esta clase contiene todas las propiedades comunes al resto de componentes, es decir, las propiedades que la clase `Component` de Vaadin posee y se van a poder utilizar en el framework. Las propiedades específicas de cada componente las posee la clase concreta que equivale al componente en cuestión. Por ejemplo, la propiedad `columns`, la posee la clase `JsonNodeTable` ya que es la única que puede tener columnas.

La serialización posee tres clases principales:

- `ComponentSerializator`. Se trata de una interfaz que define los métodos necesarios para serializar un componente de Vaadin a Json. Cada `JsonNodeComponent` debe poseer su propia implementación de `ComponentSerializator` para que ese tipo de componente pueda serializarse.
- `ComponentConverter`. Se encarga de utilizar el `ComponentSerializator` adecuado dependiendo del tipo de componente Vaadin que se desee serializar. Convierte el componente de Vaadin a un `JsonNodeComponent`.
- `ComponentTranslater`. Se encarga de transformar el `JsonNodeComponent` obtenido por el `ComponentConverter` a un Json en formato String.

Al igual que para la traducción de Json a componentes Vaadin, cada componente que se desee poder serializar debe poseer un `JsonNodeComponent` y un `ComponentSerializator`.

Hay que aclarar que si en algún futuro se soporta un nuevo tipo de componente, se deben actualizar las dos partes del modelo de domino para permitir tanto su traducción como su serialización, de esta forma el framework quedaría completo.

Diseño software

5. Diseño software

Una vez realizado el análisis software del framework, el siguiente paso es la realización del diseño software, es decir, un análisis más a fondo del desarrollo teniendo en cuenta la arquitectura y librerías que se van a utilizar para su desarrollo.

5.1. Estudio de patrones de diseño a utilizar

Debido a que el desarrollo del TFG se trata de un framework y no de una aplicación, partes como la descripción de la arquitectura de un típico Modelo-Vista-Controlador no tienen sentido en esta memoria. Por ello, se ha realizado un estudio más detallado de los posibles patrones de diseño que serían de gran utilidad incluirlos en el diseño.

Los patrones de diseño son muy importantes en el desarrollo software. Son técnicas estudiadas y probadas para diferentes problemas comunes en la implementación de productos software. De esta manera, se consigue tener un código bueno, reutilizable y fácilmente mantenible.

Existen distintas categorías de patrones de diseño, adecuadas a distintos problemas categorizados. Las principales categorías de patrones son las siguientes:

- **Patrones creacionales**. Este tipo de patrones proporcionan soluciones a la creación de instancias, ayudando a encapsular y abstraer esta creación. Algunos ejemplos son; *Builder*, *Abstract Factory*, *Singleton*…
- **Patrones estructurales**. Están enfocados en la gestión de la forma en la que las clases y los objetos se combinan para dar lugar a estructuras más complejas. Algunos ejemplos son; *Adapter*, *Composite*, *Decorator*…
- **Patrones de comportamiento**. Ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, como por ejemplo; *Iterator*, *Observer*, *Visitor*…

5.1.1. Singleton

El patrón creacional Singleton se ha decido introducir en la arquitectura del desarrollo del framework. Con el patrón Singleton restringimos que únicamente exista una instancia de una clase. Es útil cuando se requiere que un único objeto exista simultáneamente en la aplicación.

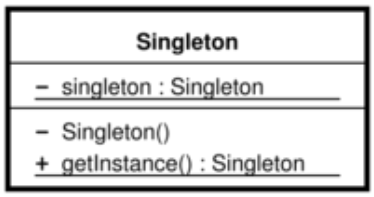

Figura 5-1. Representación patrón Singleton

Como vemos, no es un patrón que requiera de mucho código para su implementación, la propia clase es la que se encarga de asegurarse que únicamente exista una instancia simultáneamente.

Se ha decido utilizar este patrón para la clase `JSONValidator`. Esta decisión ha sido tomada debido a que esta clase tiene cargado en memoria el `JsonSchema` que se encarga de validar que un Json es válido para nuestro framework. De esta manera evitamos que cada `JSONTranslater` posea su propia instancia de `JSONValidator`, y este a su vez una nueva instancia del `JsonSchema`. Como el `JsonSchema` nunca va a cambiar, al menos en tiempo de ejecución, evitamos tener objetos iguales para el mismo desempeño.

Como muestra la representación UML del patrón Singleton, simplemente hay que ocultar el constructor, es decir hacerlo privado, y crear un método estático para obtener la instancia de la clase.

5.1.2. Composite

El patrón Composite es un patrón de tipo estructural. Con este patrón podemos construir objetos complejos a partir de otros más simples, y los cuales son similares entre si. De esta manera se simplifica la forma en la que se interactúa con los objetos creados, ya que al poseer todos una interfaz común, se pueden tratar igual.

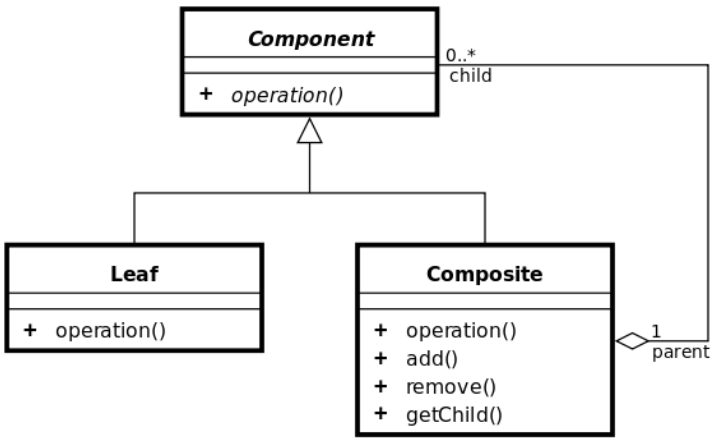

Figura 5-2. Patrón Composite

En Vaadin, hay ciertos componentes que pueden contener a otros, e incluso a componentes del mismo tipo, este es un claro ejemplo de patrón Composite. En este framework, se traducen objetos de tipo `AbstractOrderedLayout` y `TabSheet`, los cuales son componentes del tipo contenedor que acabamos de describir. Por lo tanto, los `JSONComponent` aplican el patrón Composite como se puede ver a continuación.

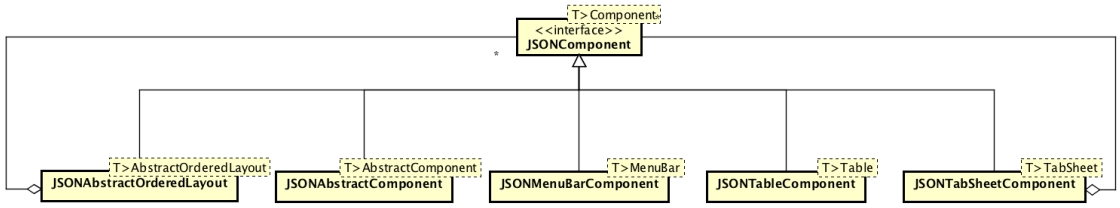

Figura 5-3. Patrón Composite aplicado

Como vemos, los componentes `JSONAbstractOrderedLayout` y los `JSONTabSheetComponent` pueden contener a cualquier componente que implemente la interfaz `JSONComponent`, y por consiguiente a ellos mismo. El día de mañana, si se quieren ampliar las opciones de traducción de componentes Vaadin, únicamente habrá que crear un objeto que implemente `JSONComponent` y un `JSONParser` correspondiente, sin la necesidad de tocar nada más de código.

5.1.3. Interpreter

Se trata de un patrón de comportamiento. El objetivo de este patrón es evaluar sentencias en un lenguaje. La idea es tener una clase por cada componente del lenguaje que sea capaz de interpretar ese componente. Además, en todo lenguaje, una sentencia puede estar compuesta por otras de otros tipos, lo que nosotros conocemos también como el patrón Composite.

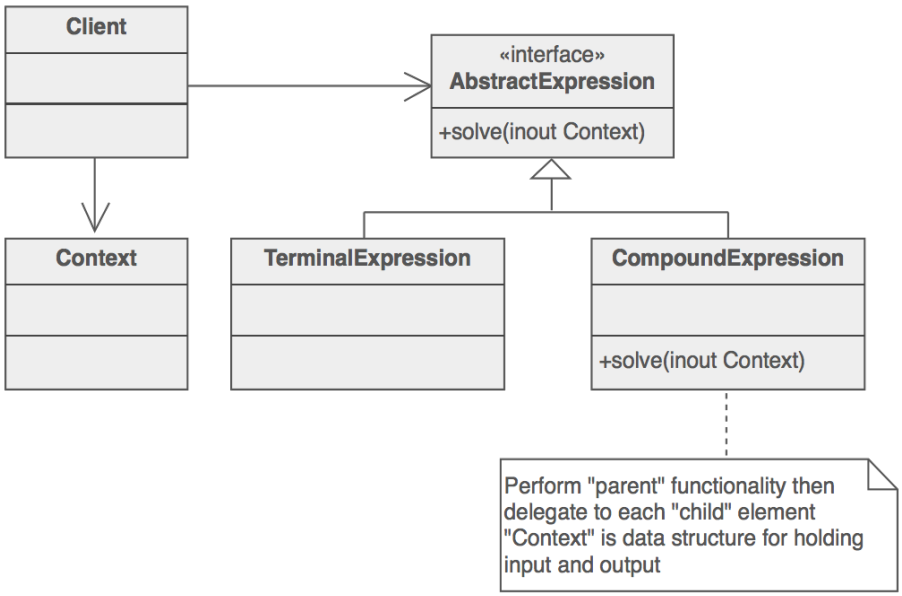

Figura 5-4. Patrón Interpreter

Como vemos, hay una interfaz común que implemetan todos los interpretes distintos, de esta manera el 'client' puede usar cualquiera de las implementaciones de los interpretes únicamente cambiando la implementación por el interprete adecuado que se desee.

Este patrón se ha usado para la implementación de los Parser y los Serializator. De esta forma, cada implementación del `JSONParser`, es en lo que la descripción del patrón se denomina como interprete. El cliente, sería en este caso el `JSONConverter`, que a partir del Json que se le proporciona para convertir, utiliza el parser adecuado, y realiza la conversión del Json a la vista de Vaadin.

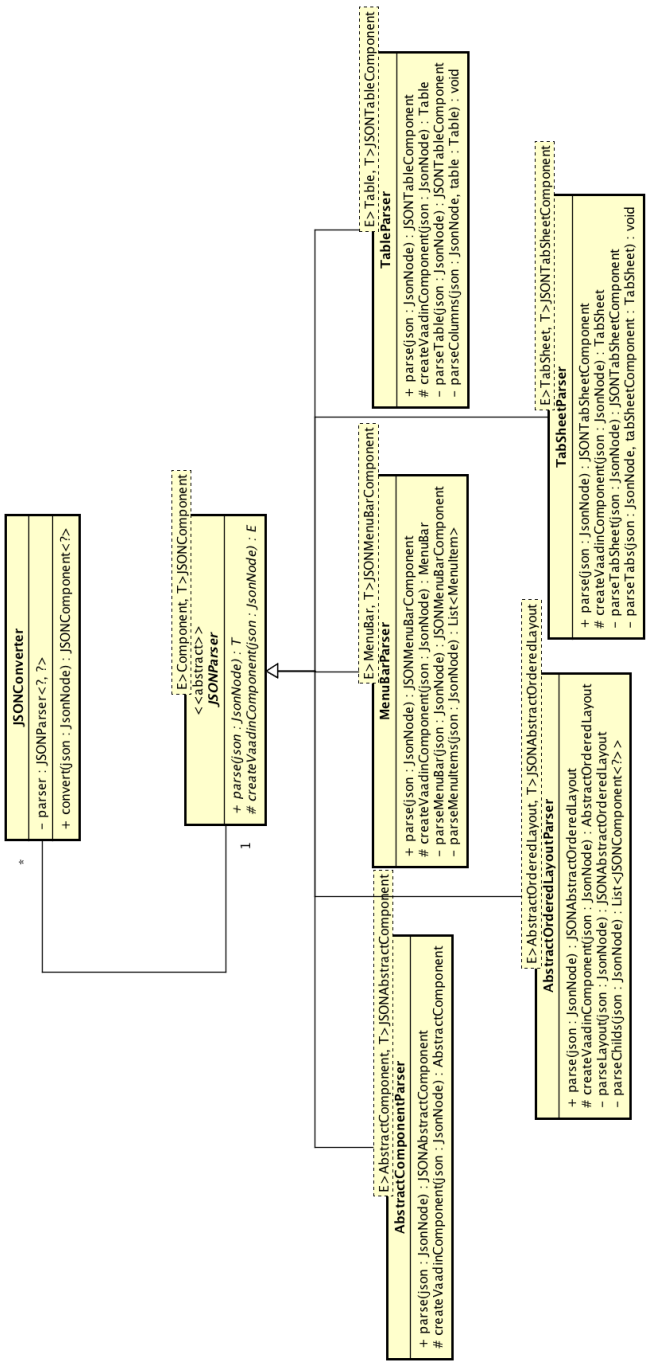

Figura 5-5. Patrón Interpreter aplicado

En el punto de diagramas de secuencia, que se encuentra más adelante, se podrá ver como es el flujo de este patrón de una forma más detallada.

Este patrón tiene muchas similitudes con el patrón denominado Strategy. Como se trata de la interpretación de un 'lenguaje' (un Json con una sintaxis bien definida) se ha decido que este es el patrón adecuado. Además tiene puntos a favor frente al patrón Strategy, la composición. Por ejemplo, los `AbstractOrderedLayout` están formados a su vez por cualquier tipo de componentes, por lo que para interpretar un `AbstractOrderedLayout` hay que interpretar antes todos los componentes que lo forman.

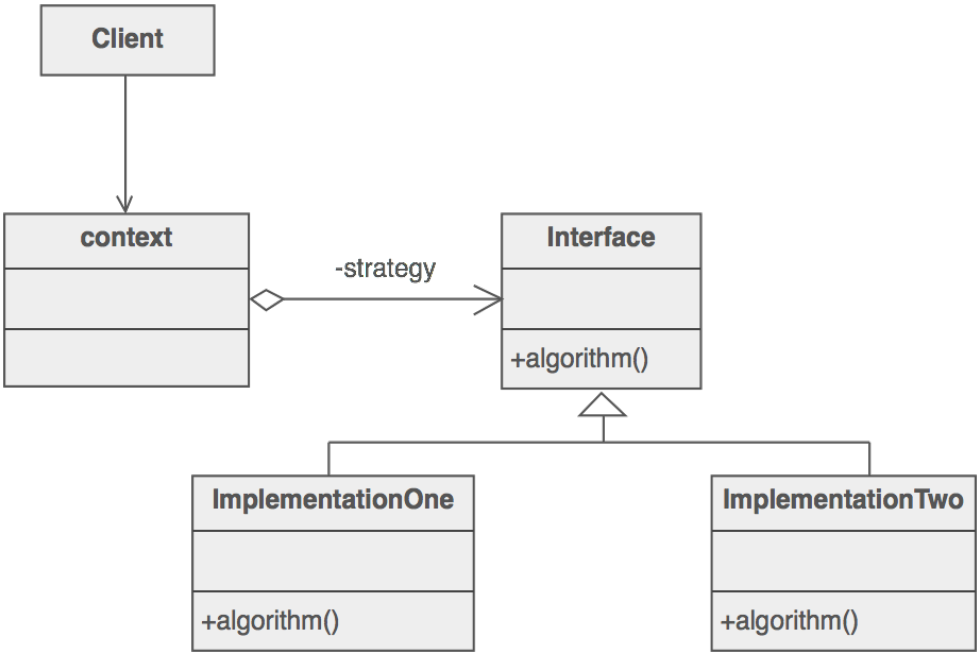

Figura 5-6. Patrón Strategy

Viendo la estructura de ambos, nos damos cuenta de que son patrones muy parecidos, por lo que básicamente se podría haber usado cualquier de ellos.

5.2. Modelo de dominio

A la hora del diseño software aparecen clases que no se habían contemplado en el modelo de domino realizado durante el análisis de la aplicación. A continuación, se muestra el modelo de dominio completo del framework a desarrollar. Como antes, lo dividimos en dos partes, primero la parte que realiza la traducción de Json a vistas de Vaadin y por último la parte de la serialización de vistas de Vaadin a Json.

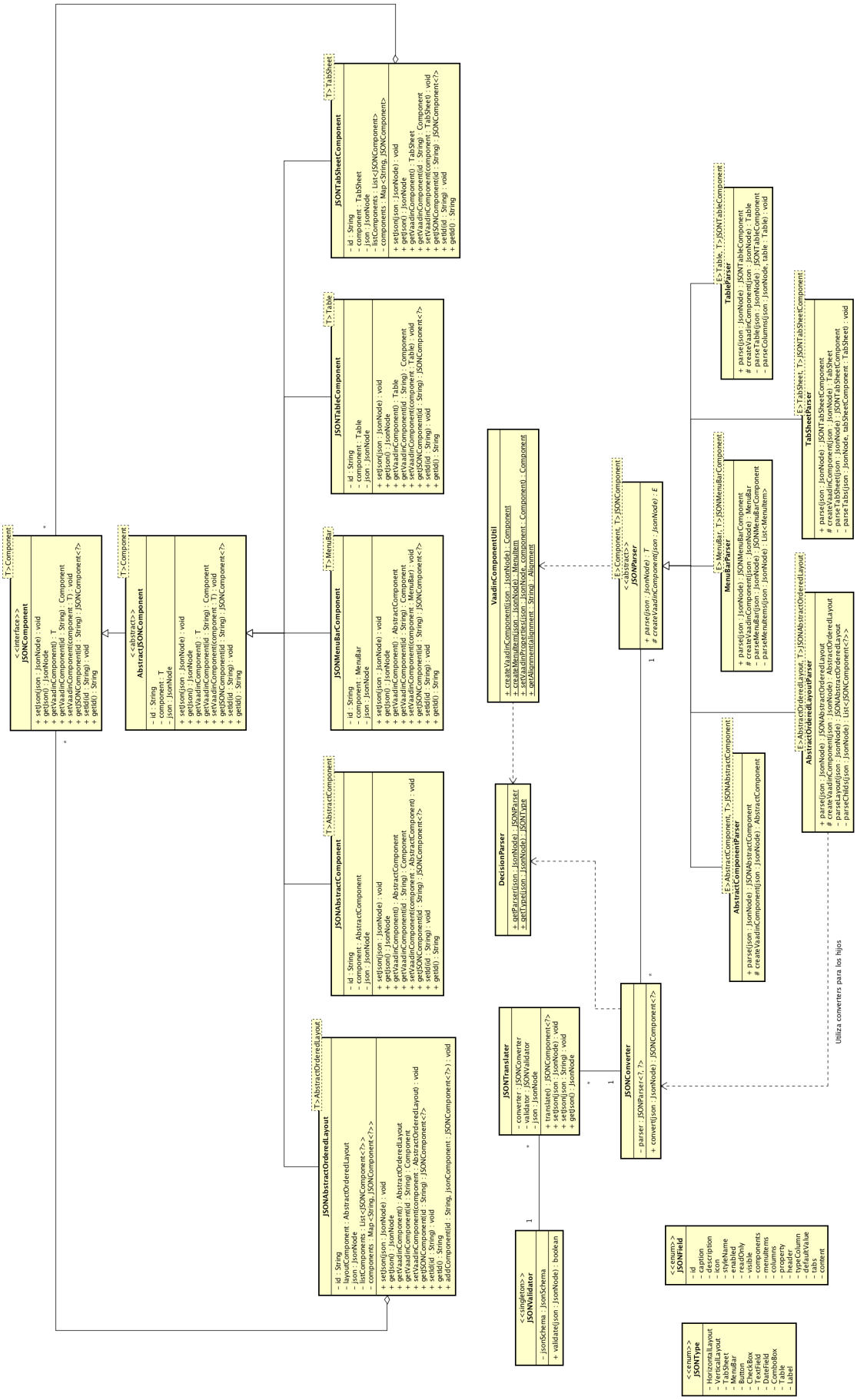

Figura 5-7. Modelo de dominio diseño, parte 1. Generación de vistas a partir de JSON

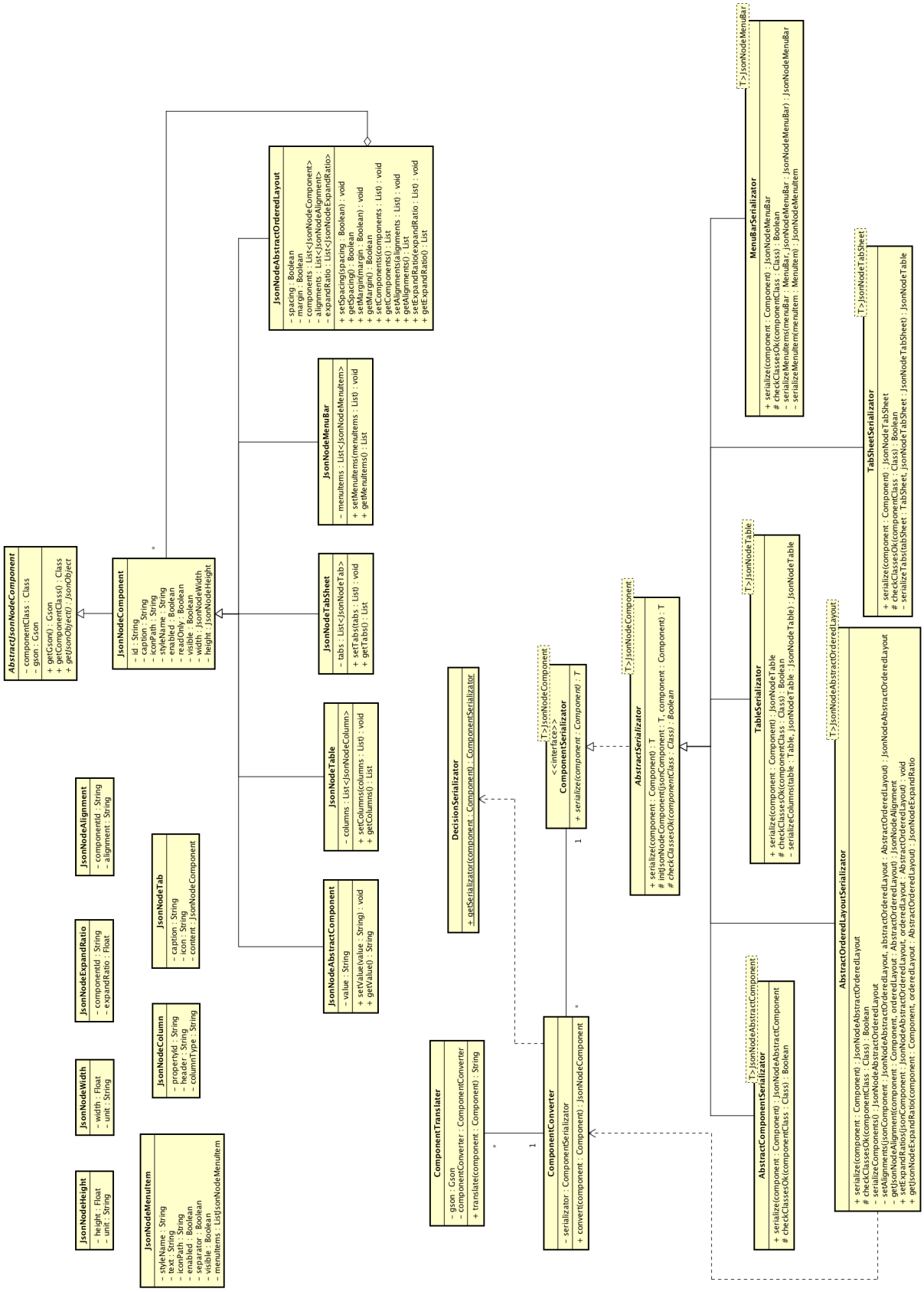

Figura 5-8. Modelo de dominio diseño, parte 2. Serialización de vistas a Json

5.3. Descripción de la estructura de los Json interpretables

En este apartado se van a describir los Json que va a admitir el framework para realizar la conversión a vistas de Vaadin. Para su descripción, se va usar la especificación de JsonSchema[1] utilizada para describir de una forma detallada y clara la documentación de esquemas basados en objetos Json.

Antes de empezar con la descripción detallada de cada uno de ellos, todos los componentes de Vaadin tienen propiedades en común, es por esto que todos heredan de la clase `Component`. Estas propiedades comunes se describen en el siguiente fragmento de JsonSchema.

[1] JsonSchema es una especificación para la descripción de objetos Json, para más información visitar su web principal json-schema.org

```
{
   "definitions":{
      "Component":{
         "type": "object",
         "properties":{
            "id":{ "type": "string" },
            "caption":{ type": "string" },
            "description":{ "type": "string" },
            "iconPath":{ type": "string" },
            "styleName":{ "type": "string" },
            "enabled":{ "type": "boolean" },
            "readOnly":{ "type": "boolean" },
            "visible":{ "type": "boolean" },
            "width":{
               "type": "object",
               "properties":{
                  "width":{
                     "type": "number",
                     "minimum":0
                  },
                  "unit":{
                     "enum":[px", "%" ]
                  }
               },
               "additionalProperties": false
            },
            "height":{
               "type": "object",
               "properties":{
                  "height":{
                     "type": "number",
                     "minimum":0
                  },
                  "unit":{
                     "enum":[ "px", "%" ]
                  }
               },
               "additionalProperties": false
            }
         }
      }
   }
}
```

5.3.1. Json de los objetos AbstractField

Se denominan objetos base a aquellos objetos principales presentes en cualquier framework de desarrollo de interfaces gráficas. En nuestro caso son; Button, CheckBox, TextField, DateField y ComboBox.

```
{
   "type": "object",
   "properties":{
      "Button":{
         "allOf":[ { "$ref":"#/definitions/AbstractFieldSchema" } ]
      },
      "CheckBox":{
         "allOf":[ { "$ref":"#/definitions/AbstractFieldSchema" } ]
      },
      "TextField":{
         "allOf":[ { "$ref":"#/definitions/AbstractFieldSchema" } ]
      },
      "DateField":{
         "allOf":[ { "$ref":"#/definitions/AbstractFieldSchema" } ]
      },
      "ComboBox":{
         "allOf":[ { "$ref":"#/definitions/AbstractFieldSchema" } ]
      }
   },
   "additionalProperties": false,
   "definitions":{
      "AbstractFieldSchema":{
         "allOf":[
            { "$ref":"#/definitions/Component" }
         ],
         "properties":{
            "id":{ },
            "caption":{ },
            "description":{ },
            "iconPath":{ },
            "styleName":{ },
            "enabled":{ },
            "readOnly":{ },
            "visible":{ },
            "width":{ },
            "height":{ },
            "value":{
               "type": "string"
            }
         },
         "additionalProperties" : false
      }
   }
}
```

5.3.2. Json de los objetos AbstractOrderedLayout

Dentro de este tipo de objetos encapsulamos los componentes HorizontalLayout y VerticalLayout.

```
{
   "type" : "object",
   "properties" : {
        "HorizontalLayout" : {
            "allOf" : [ { "$ref": "#/definitions/AbstractOrderedLayout" } ]
        },
        "VerticalLayout" : {
            "allOf" : [ { "$ref": "#/definitions/AbstractOrderedLayout" } ]
        }
   },
   "additionalProperties" : false,
   "definitions":{
      "AbstractOrderedLayout":{
         "allOf":[ { "$ref":"#/definitions/Component" } ],
         "properties":{
            "id":{ },
            "caption":{ },
            "description":{ },
            "iconPath":{ },
            "styleName":{ },
            "enabled":{ },
            "readOnly":{ },
            "visible":{ },
            "width":{ },
            "height":{ },
            "spacing" : { "type" : "boolean" },
            "margin" : { "type" : "boolean" },
            "components" : {
                "type" : "array",
                "items" : {
                    "type" : "object",
                    "oneOf" : [ { "$ref": "#" } ]
                },
                "minItems": 1
            },
            "alignments" : {
                "type" : "array",
                "items" : {
                    "type" : "object",
                    "oneOf" : [
                        { "$ref" : "#/definitions/Alignment" }
                    ]
                },
                "minItems" : 1
            },
   ...
```

```
            "expandRatio" : {
                "type" : "array",
                "items" : {
                    "type" : "object",
                    "oneOf" : [
                        { "$ref" : "#/definitions/ExpandRatio"}
                    ]
                },
                "minItems" : 1
            }
        },
        "additionalProperties" : false
       },
       "Alignment" : {
         "type" : "object",
         "properties" : {
             "componentId" : {
                 "type" : "string"
             },
             "alignment" : {
                 "enum" : [
                     "TOP_LEFT",
                     "TOP_CENTER",
                     "TOP_RIGHT",
                     "MIDDLE_LEFT",
                     "MIDDLE_CENTER",
                     "MIDDLE_RIGHT",
                     "BOTTOM_LEFT",
                     "BOTTOM_CENTER",
                     "BOTTOM_RIGHT"
                 ]
             }
         },
         "additionalProperties" : false
         },
         "ExpandRatio" : {
           "type" : "object",
           "properties" : {
               "componentId" : {
                   "type" : "string"
               },
               "expandRatio" : {
                   "type" : "number",
                   "minimun" : 0
               }
           },
           "additionalProperties" : false
        }
    }
}
```

5.3.3. Json de los objetos MenuBar

```
{
   "type": "object",
   "properties":{
      "MenuBar":{
         "allOf":[ { "$ref":"#/definitions/MenuBar" } ]
      }
   },
   "additionalProperties": false,
   "definitions":{
      "MenuBar":{
         "allOf":[ { "$ref":"#/definitions/Component" } ],
         "properties":{
            "id":{ },
            "caption":{ },
            "description":{ },
            "iconPath":{ },
            "styleName":{ },
            "enabled":{ },
            "readOnly":{ },
            "visible":{ },
            "width":{ },
            "height":{ },
            "menuItems":{
               "type": "array",
               "minItems":1,
               "items":{
                  "type": "object",
                  "oneOf":[
                     { "$ref":"#/definitions/MenuItem" }
                  ]
               }
            }
         },
         "additionalProperties": false
      },
      "MenuItem":{
         "properties":{
            "styleName":{
               "type": "string"
            },
            "text":{
               "type": "string"
            },
            "iconPath":{
               "type": "string"
            },
            "separator":{
               "type": "boolean"
            },
            "enabled":{
               "type": "boolean"
            },
...
```

```
            "visible":{
               "type": "boolean"
            },
            "menuItems":{
               "type": "array",
               "minItems":1,
               "items":{
                  "type": "object",
                  "oneOf":[ { "$ref":"#/definitions/MenuItem" } ]
               }
            }
         },
         "required":[ "text" ]
      }
   }
}
```

5.3.4. Json de los objetos Table

```
{
   "type": "object",
   "properties":{
      "Table":{
         "allOf":[ { "$ref":"#/definitions/Table" } ]
      }
   },
   "additionalProperties": false,
   "definitions":{
      "Table":{
         "allOf":[ { "$ref":"#/definitions/Component" } ],
         "properties":{
            "id":{ },
            "caption":{ },
            "description":{ },
            "iconPath":{ },
            "styleName":{ },
            "enabled":{ },
            "readOnly":{ },
            "visible":{ },
            "width":{ },
            "height":{ },
            "columns":{
               "type": "array",
               "minItems":1,
               "items":{
                  "type": "object",
                  "oneOf":[ { "$ref":"#/definitions/Column" } ]
               }
            }
         },
         "additionalProperties": false
      },
...
```

```
        "Column":{
            "properties":{
                "propertyId":{
                    "type": "string"
                },
                "header":{
                    "type": "string"
                },
                "columnType":{
                    "enum":[
                        "String",
                        "Integer",
                        "Float",
                        "Double",
                        "Boolean"
                    ]
                },
                "defaultValue":{
                    "type": "boolean"
                }
            },
            "required":[ "propertyId", "columnType" ]
        }
    }
}
```

5.3.5. Json de los objetos TabSheet

```
{
    "type": "object",
    "properties":{
        "TabSheet":{
            "allOf":[ { "$ref":"#/definitions/TabSheet" } ]
        }
    },
    "additionalProperties": false,
    "definitions":{
        "TabSheet":{
            "allOf":[ { "$ref":"#/definitions/Component" } ],
            "properties":{
                "id":{ },
                "caption":{ },
                "description":{ },
                "iconPath":{ },
                "styleName":{ },
                "enabled":{ },
                "readOnly":{ },
                "visible":{ },
                "width":{ },
                "height":{ },
...
```

```
            "tabs":{
               "type": "array",
               "minItems":1,
               "items":{
                  "type": "object",
                  "oneOf":[ { "$ref":"#/definitions/Tab" } ]
               }
            }
         },
         "additionalProperties": false
      },
      "Tab":{
         "properties":{
            "caption":{
               "type": "string"
            },
            "icon":{
               "type": "string"
            },
            "content":{
               "type": "object",
               "oneOf":[
                  { "$ref":"#" }
               ]
            }
         },
         "required":[ "content", "caption" ],
         "additionalProperties": false
      }
   }
}
```

5.4. Diagramas de secuencia

En este apartado vamos a describir parte de los diagramas de secuencia, ya que debido a su tamaño, y que no se pueden visualizar correctamente en una hoja de tamaño A4, únicamente se utilizarán los diagramas que tienen más importancia en el desarrollo del framework.

5.4.1. Caso de uso validar Json

Este caso de uso es de los más importantes, ya que se encarga de validar los Json que el framework va a intentar traducir a vistas de Vaadin. De esta forma prevenimos que si el Json no tienen un formato adecuado, no se proceda a su traducción, lo que podría provocar fallos no esperados o controlados.

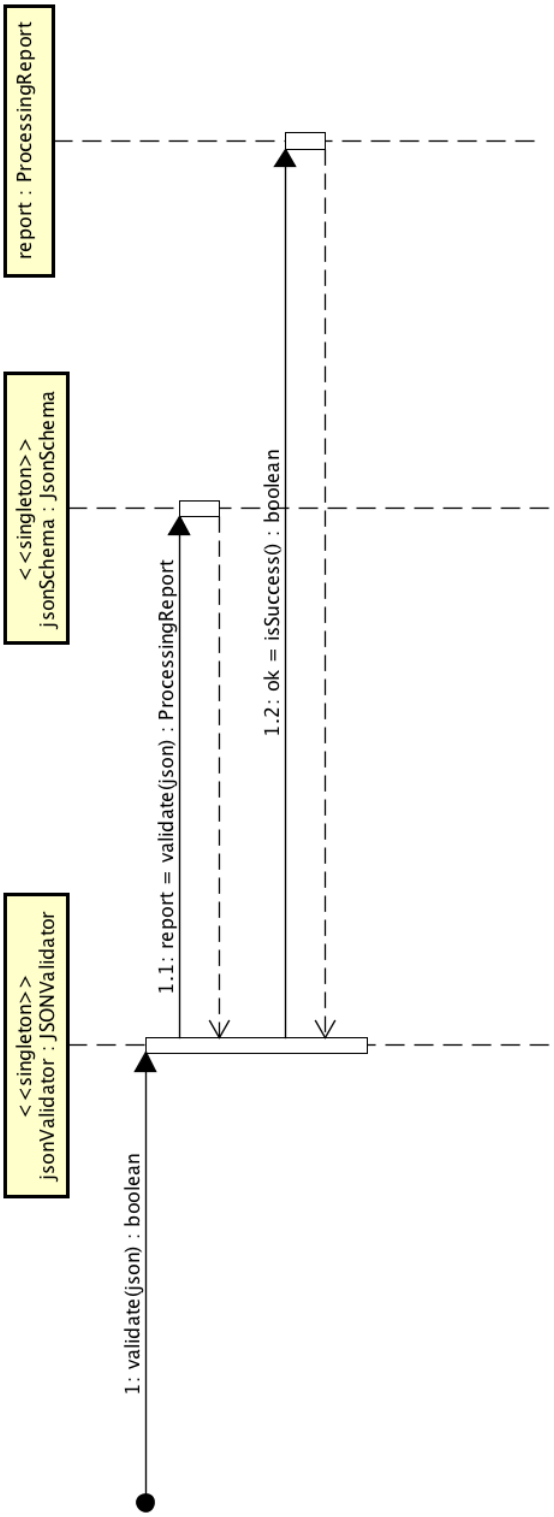

Figura 5-9. Caso de uso 'Validar Json'

Como se ve en la figura, es un caso de uso muy sencillo, y esto es gracias a la especificación del JsonSchema y a la librería utilizada para la validación de los Json a partir del JsonSchema definido.

5.4.2. Caso de uso Generación de componentes base

Este caso de uso nos va a servir para describir como se realiza la transformación de objetos Json a vistas de Vaadin, debido a que es el más sencillo de los cinco casos de uso existentes de traducción.

Para una mejor interpretación de los diagramas de secuencia, debido a que estos son muy extensos, cada uno de ellos está dividido en sub casos de uso más pequeños.

Obtención del Parser

El primero de estos sub casos de uso es el de la obtención del parser adecuado según el objeto Json que se le indica al `JSONTranslater`. Este caso de uso es el que aplica el patrón Interpreter descrito en el punto 5.1.3. Interpreter.

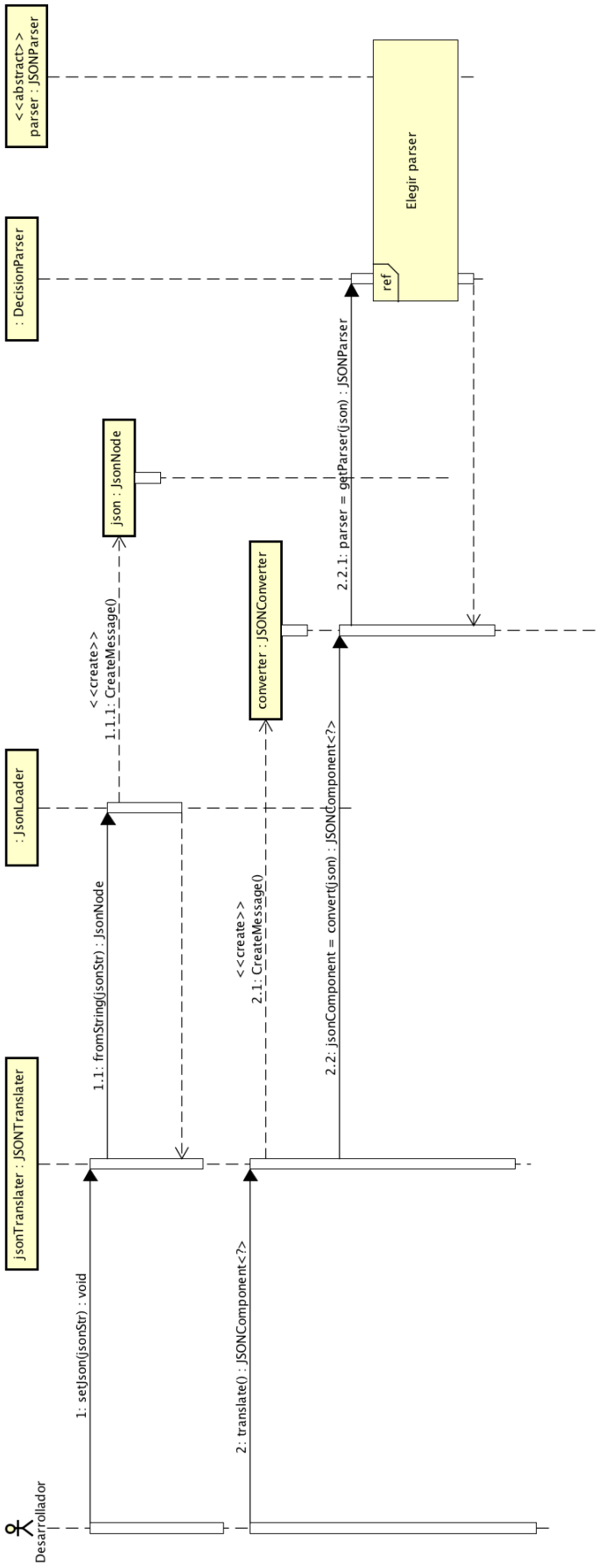

Figura 5-10. Sub caso de uso 'Obtener parser'

A su vez, este caso de uso tiene otro pequeño sub apartado, llamado 'Elegir Parser', el cual se describe en el siguiente diagrama de secuencia.

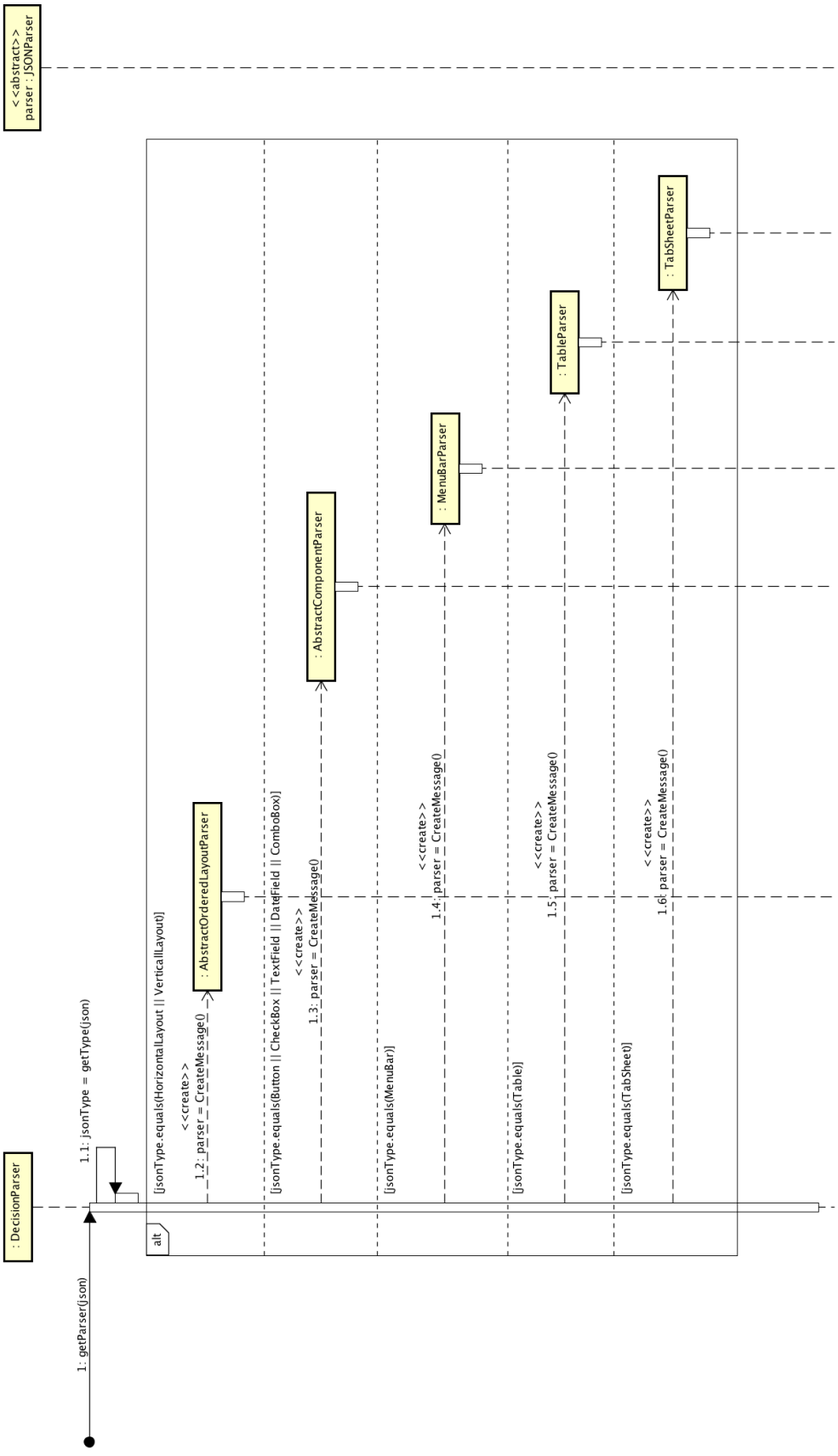

Figura 5-11. Sub caso de uso 'Elegir parser'

Crear componente Vaadin

El segundo sub caso de uso por el cual esta formado el caso de uso de generación de componentes base, es el de creación del componente Vaadin a partir de las propiedades descritas en el Json.

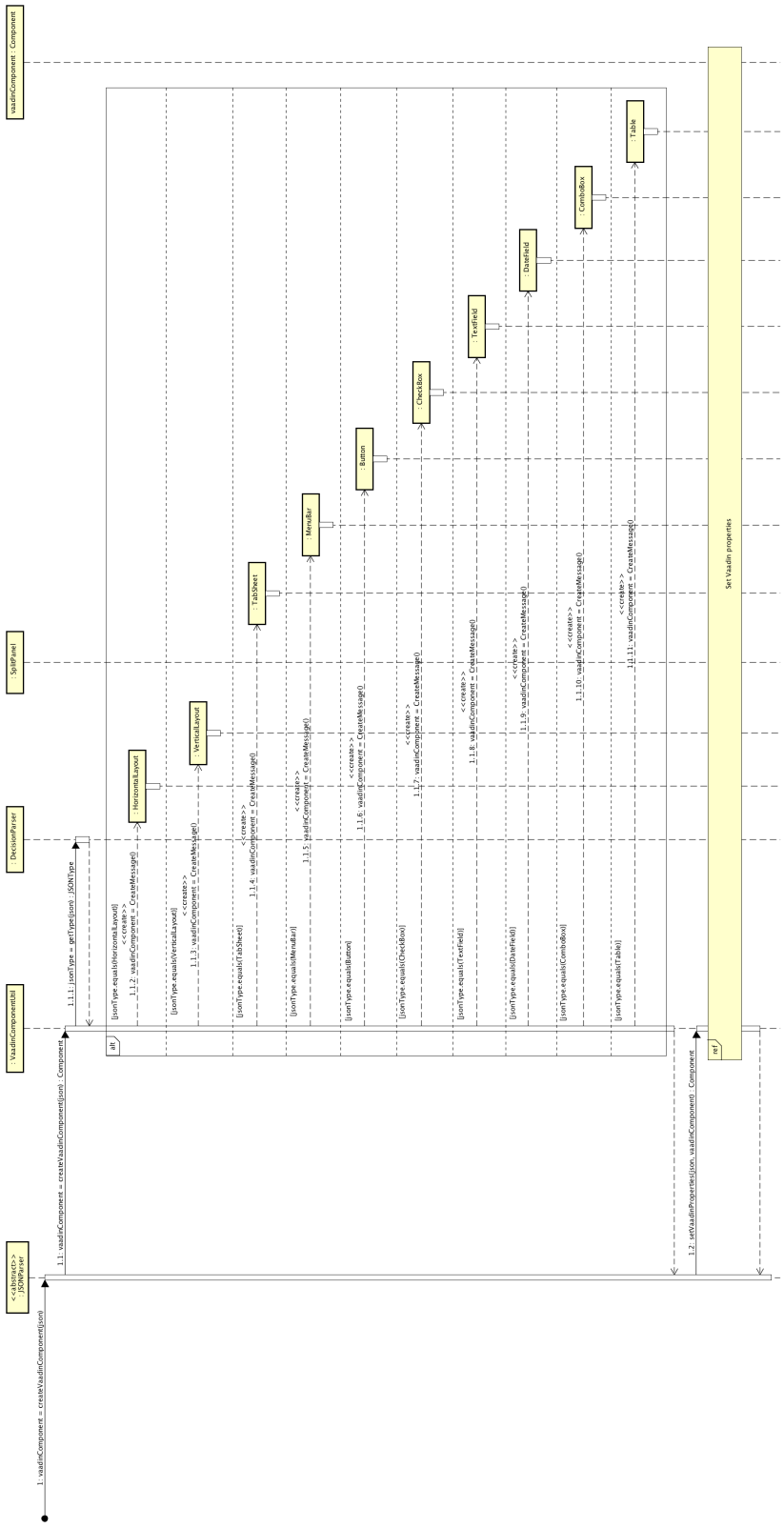

Figura 5-12. Sub caso de uso 'Creación componente Vaadin'

Este diagrama describe como a partir de la propiedad tipo del Json, se crea el objeto de Vaadin adecuado. A continuación, se le asignan las propiedades comunes a todos los componentes, es decir, las que heredan del componente Vaadin `Component`.

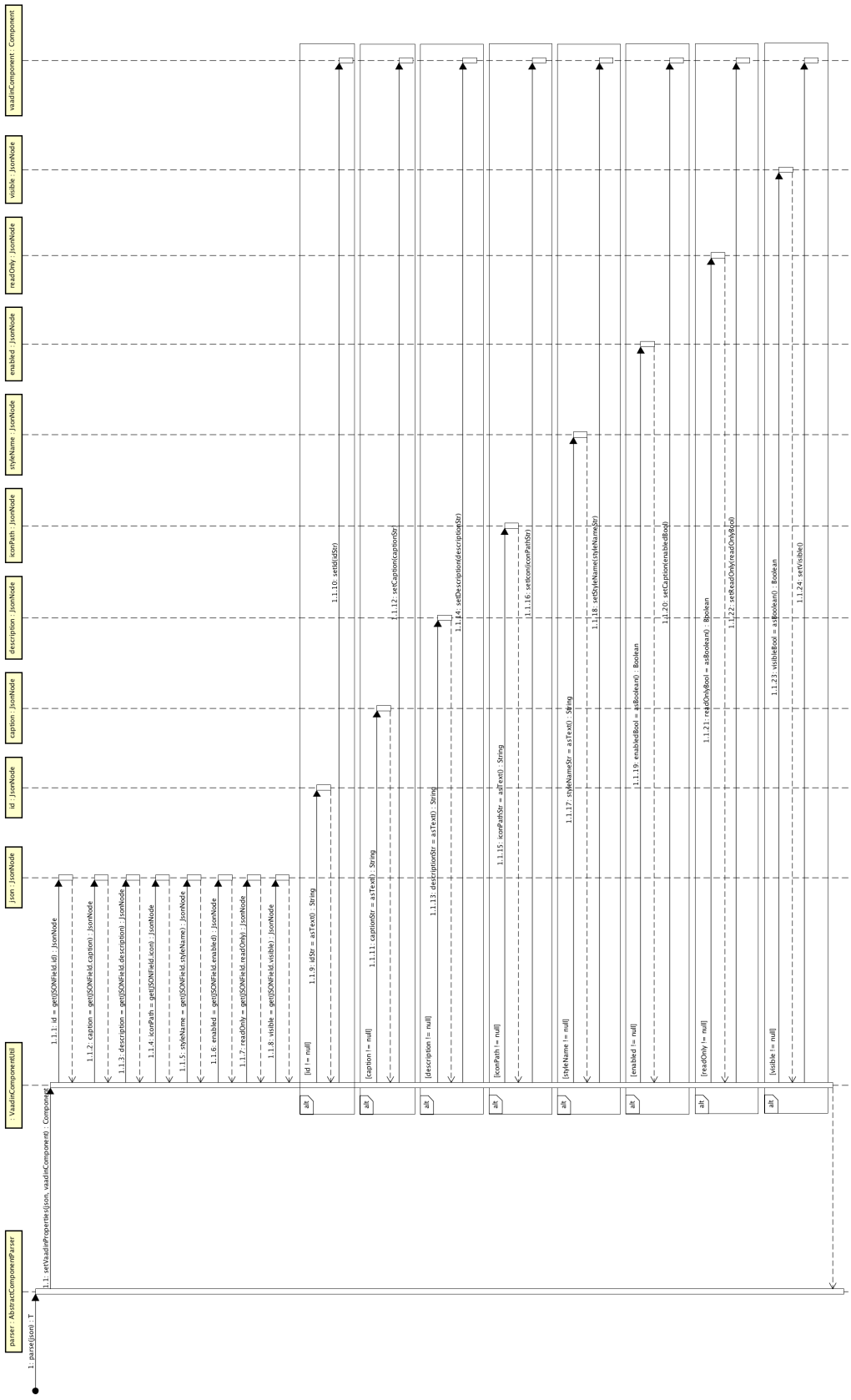

Figura 5-13. Sub caso de uso 'Set Vaadin Properties'

Una vez descritos todos los sub casos de uso por los que está compuesto el caso de uso generador de componentes base, se incluye a continuación el diagrama de secuencia final.

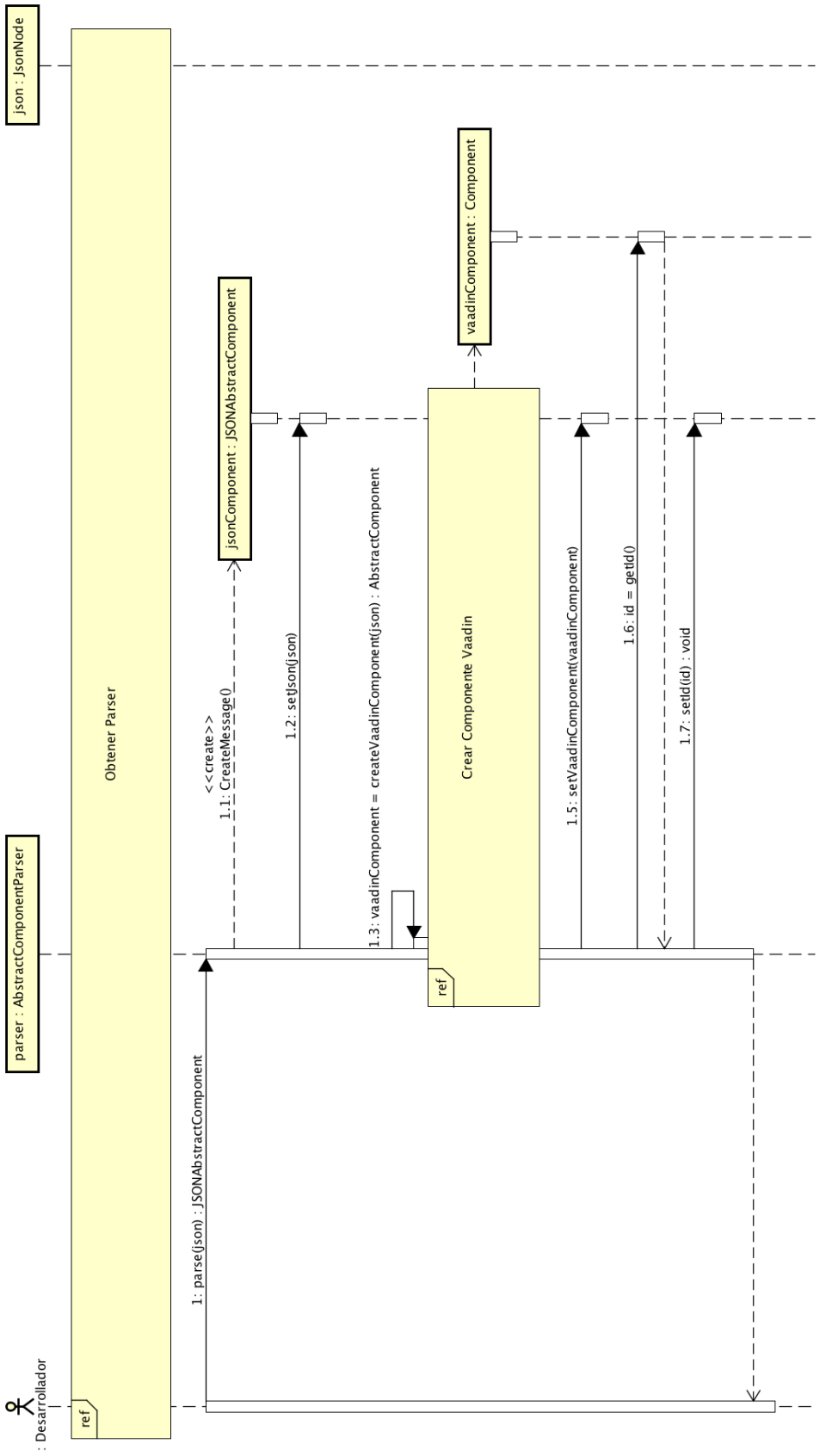

Figura 5-14. Caso de uso 'Generar componentes base'

5.4.3. Caso de uso Serializar componentes base

Para la descripción de un caso de uso de serialización de componentes, se ha elegido el homólogo al descrito en el punto anterior, ya que es el más sencillo de los cinco casos de uso de serialización existentes.

Como en el caso de uso anterior, aunque se trate de un caso de uso sencillo, se ha dividido en sub casos de uso para una mejor interpretación de los diagramas de secuencia.

Obtención del Serializator

Este caso de uso se encarga de escoger el `ComponentSerializator` adecuado para la vista de Vaadin que se quiere serializar a Json.

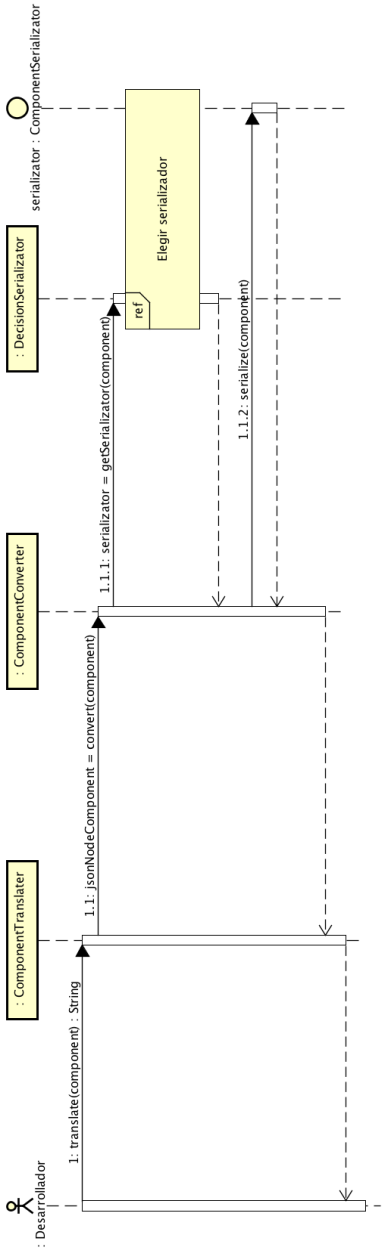

Figura 5-15. Sub caso de uso 'Obtener serializator'

Este a su vez está formado por otro sub caso de uso denominado 'Elegir serializator' el cual se describe a continuación.

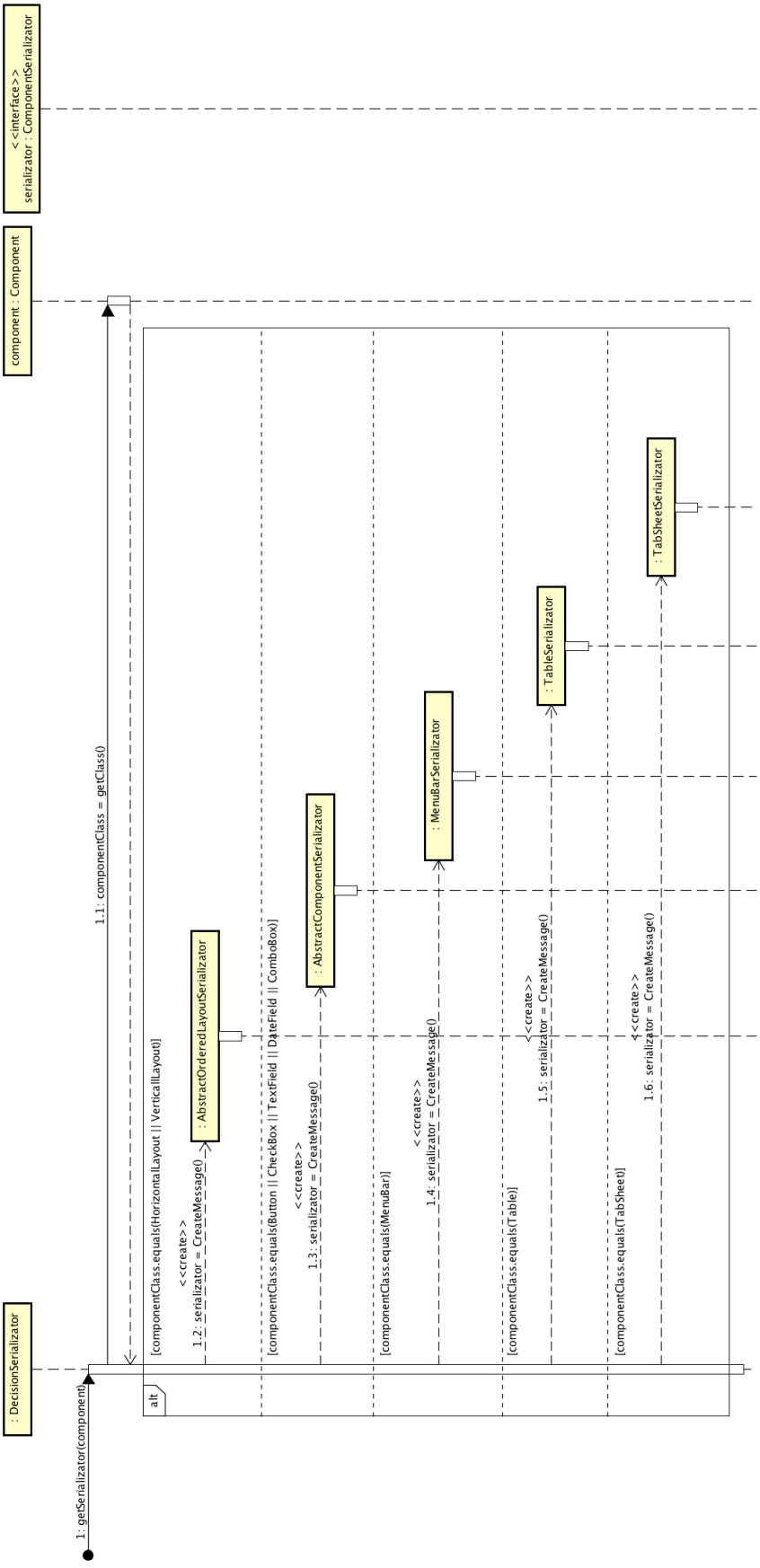

Figura 5-16. Sub caso de uso 'Elegir serializator'

Inicialización JsonNodeComponent

El siguiente caso de uso describe como se obtienen las propiedades comunes de los componentes de Vaadin para formar el Json correspondiente. Estas propiedades son las que todo componente hereda de `Component`.

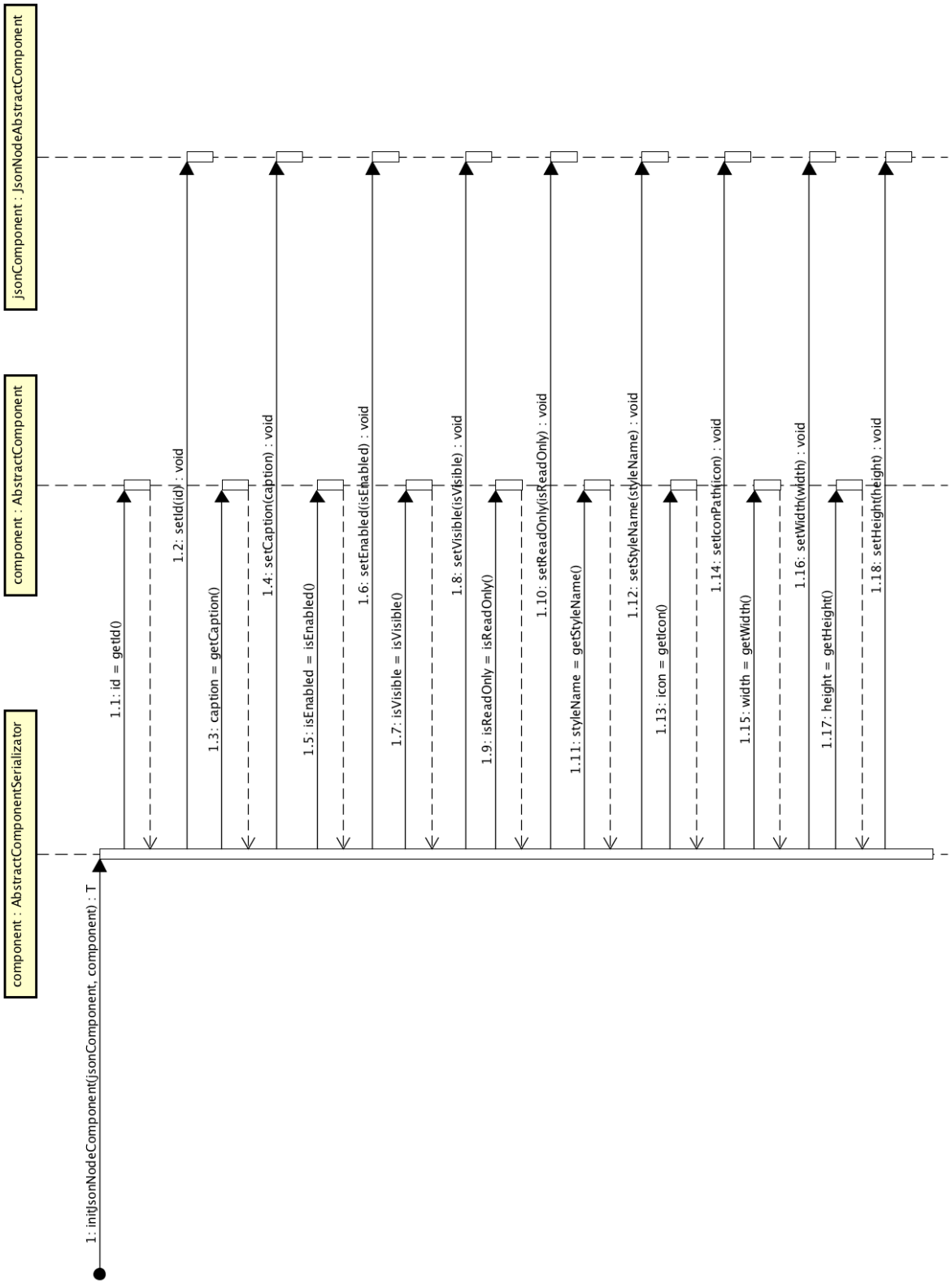

Figura 5-17. Sub caso de uso 'Init JsonNodeComponent'

Una vez descritos los sub casos de uso por los que esta formado el caso de uso ‘Serialización de componentes base’, se muestra a continuación el diagrama de secuencia correspondiente.

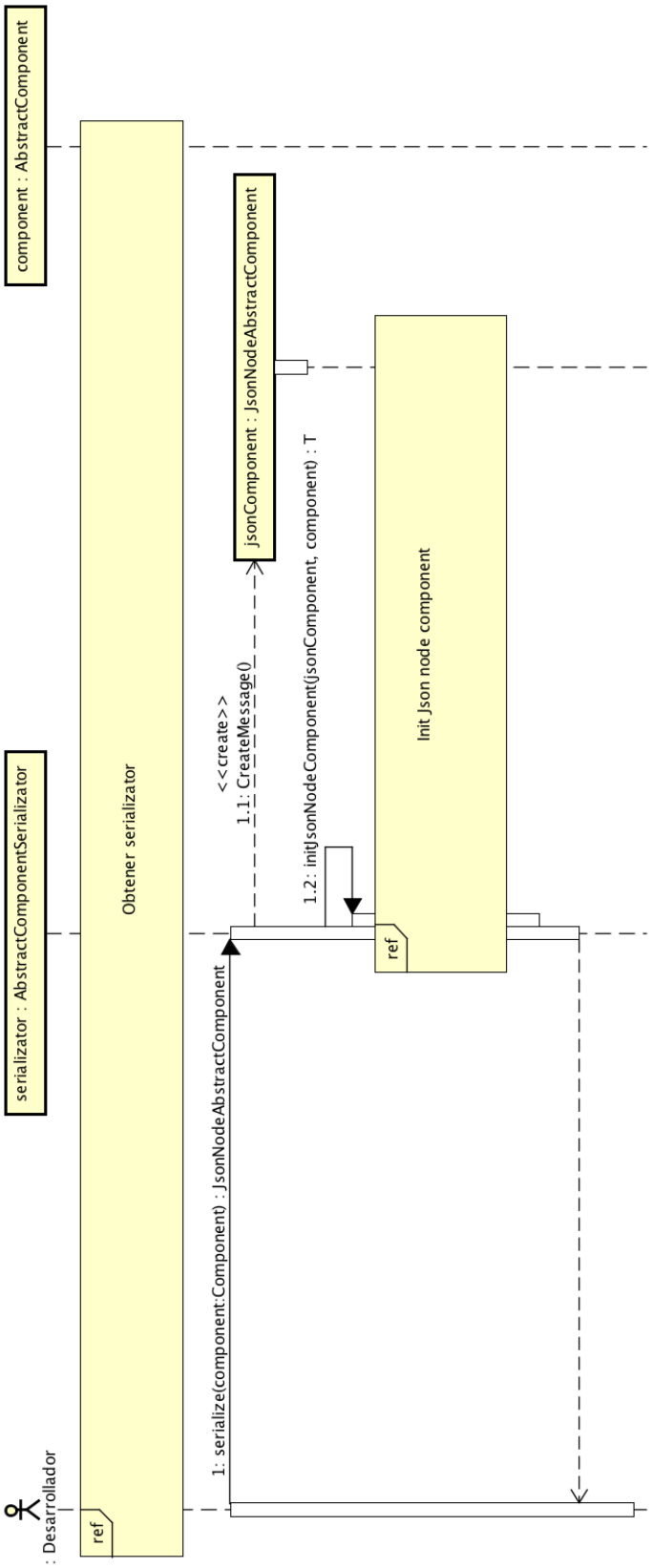

Figura 5-18. Caso de uso 'Serialización de componentes base'

Implementación

6. Implementación

El objetivo de este apartado es describir el software y hardware utilizado a la hora de la realización del proyecto. Se describirá que librerías han sido necesarias para el desarrollo, y que requisitos son necesarios para la utilización del framework.

6.1. Entorno de desarrollo

Durante la implementación del proyecto se han utilizado las siguientes herramientas para su correcta implementación:

- JDK 8. Java Development Kit en su versión 8, es decir, la versión más reciente disponible del lenguaje de programación Java. De esta forma se han podido utilizar las ultimas novedades de este lenguaje como las lambdas.
- Eclipse. El IDE utilizado para el desarrollo ha sido eclipse, en concreto a su ultima versión Mars. Se ha optado por esta elección debido a que es un IDE muy conocido y estable para el desarrollo de Java.
- Maven. Maven es una herramienta de construcción de proyectos Java. Gracias a Maven es muy sencillo incluir librerías externas en el proyecto definiéndolas en un archivo de configuración, el conocido como pom.xml.

Para la utilización del framework, los dos únicos requisitos necesarios son:

- Java 8. El framework únicamente podrá usarse en desarrollos que utilicen el JDK 8 de Java, ya que ha sido desarrollado específicamente para esta versión.
- Vaadin 7. El framework se ha desarrollado para dar compatibilidad a Vaadin 7. No se asegura su correcto funcionamiento en versiones inferiores de Vaadin, por lo que se aconseja usar esta versión de Vaadin.

6.2. Librerías utilizadas

Durante el desarrollo de este proyecto se han utilizado tres librerías que han sido esenciales, sin contar Vaadin, que ya se ha descrito múltiples veces durante el documento.

6.2.1. Json Schema Validator

Como bien se ha dicho antes, se usa la especificación Json Schema para la validación de los Json. En la pagina de la especificación de Json Schema, json-schema.org, se sugieren dos librerías que implementan la especificación del Json Schema para Java.

Ambas librerías son de código abierto y apoyadas por la comunidad, por lo que uno de los motivos para decidir fue el número de forks y de estrellas que disponía cada una de ellas en Github. Por ello, se decidió usar la librería github.com/fge/json-schema-validator la cual obtenía mejores resultados aplicando el criterio previamente descrito.

Esta librería proporciona una completa validación para el último draft de la librería, el draft v4, soportando también el draft anterior v3. La versión de la librería utilizada en el desarrollo es la 2.2.6.

6.2.2. Jackson Databind

El propósito de esta librería, de código abierto también, es la interpretación de objetos Json. Los desarrolladores de esta librería se inspiraron en la cantidad de librerías para interpretar XML que existen para Java, como por ejemplo StAX[2] o JABX[3].

Su código está público en Github, github.com/FasterXML/jackson-databind, y cualquiera puede contribuir al desarrollo de la librería. Es una de las librerías más utilizadas en Java a la hora de tratar objetos Json, y además, la librería usada para el Json Schema espera objetos de esta librería, por lo que ha sido una de las mayores razones por las que se ha decido utilizar.

Gracias a esta librería podemos acceder a las propiedades de los Json de una forma sencilla, y poder así generar los componentes Vaadin que corresponden. La versión utilizada en el desarrollo es la 2.7.0.

6.2.3. Gson

Al igual que la librería de Jackson, esta tiene el mismo objetivo, interpretar objetos Json. Es una librería desarrollada por google, la cual tiene también su código publicado en Github, github.com/google/gson.

Esta librería ha sido necesaria a la hora de la serialización de vistas de Vaadin a objetos Json, ya que tiene alguna funcionalidad que Jackson no posee. La principal ventaja de esta librería es que traduce objetos Java a objetos Json simplemente mediante una función. Esta funcionalidad también la posee la librería Jackson, pero Gson permite muchas más opciones de personalización a la hora de realizar esta serialización de objetos Java. La versión utilizada en el desarrollo es la 2.6.2.

[2] StAX. Librería para leer y escribir documentos XML en Java. en.wikipedia.org/wiki/StAX

[3] JABX. Librería para leer y escribir documentos XML en Java incluida en el propio JDK de Java. oracle.com

6.2.4. Conclusiones

Todas las librerías descritas y utilizadas en el framework son de código abierto, es decir, son librerías de uso gratuito. Hay distintos tipos de licencias de código abierto, y la mayoría permite su uso sin ningún problema, siempre y cuando no sea para beneficio económico.

Es importante antes de desarrollar un proyecto buscar bibliotecas que se puedan utilizar y que te ahorren tiempo y trabajo. Actualmente mucha gente está dispuesta a ayudar y a contribuir en librerías de código abierto, por lo que es mucho más sencillo encontrar librerías acordes a tus necesidades. El uso de estas bibliotecas te ahorrará mucho trabajo, y hay que tener en cuenta que el gran número de desarrolladores aportando a este tipo de proyectos, significa que se detectarán, corregirán fallos y se añadirán nuevas funcionalidades con una frecuencia relativamente alta.

A parte de ser librerías de código abierto, estas se encuentran en los repositorios de Maven central, por lo que es muy sencillo añadirlas a tu proyecto y ponerte a usarlas en un muy corto periodo de tiempo.

Pruebas

7. Pruebas

El objetivo de realizar pruebas en el desarrollo software es el de la detección de errores no controlados durante la implementación. Es muy importante realizar un buen plan de pruebas para evitar errores durante el desarrollo, y para garantizar que el producto a desarrollar ha superado una serie de test, por lo que se podría decir que el producto es correcto y posee un buen funcionamiento.

7.1. Casos de prueba

Para la realización de las pruebas se va a utilizar el framework JUnit, el cual permite definir una serie de casos de prueba junto con las variables de salida esperadas para caso de prueba. De esta forma, si alguna salida no fuese la esperada, el test no pasaría correctamente y por tanto habría algún fallo en los casos de prueba descritos.

Las pruebas a realizar van a ser las denominadas de caja negra. Este tipo de pruebas se caracteriza por que se estudia el componente en función de las entradas que recibe y las salidas producidas, sin llegar a estudiar su funcionamiento interno.

A continuación se detallan los casos de pruebas que se van a realizar durante la batería de pruebas.

7.1.1. Casos de prueba para la validación de los Json

El objetivo de estas pruebas es comprobar que los Json se están validando correctamente, tanto para cuando se trata de un Json válido, como para uno inválido. Se trata de una batería de pruebas reducida, ya que en el caso de comprobar que los Json son inválidos, existiría un caso de prueba por cada propiedad que puede fallar.

Tabla 7-1. CP Validar un Json válido de AbstractField

CP – 001	Validar un Json válido de un componente AbstractField
Descripción	Validar un Json válido correspondiente a un componente Vaadin de tipo AbstractField.
Resultado esperado	El Json se va a validar correctamente.

Tabla 7-2. CP Validar un Json no válido de AbstractField

CP – 002	Validar un Json no válido de un componente AbstractField
Descripción	Validar un Json no válido correspondiente a un componente Vaadin de tipo AbstractField.
Resultado esperado	El Json se va indicar como no válido debido a que no cumple la especificación del Json Schema correspondiente.

Tabla 7-3. CP Validar un Json válido de AbstractOrderedLayout

CP – 003	Validar un Json válido de un componente AbstractOrderedLayout
Descripción	Validar un Json válido correspondiente a un componente Vaadin de tipo AbstractOrderedLayout.
Resultado esperado	El Json se va a validar correctamente.

Tabla 7-4. CP Validar un Json no válido de AbstractOrderedLayout

CP – 004	Validar un Json no válido de un componente AbstractOrderedLayout
Descripción	Validar un Json no válido correspondiente a un componente Vaadin de tipo AbstractOrderedLayout.
Resultado esperado	El Json se va indicar como no válido debido a que no cumple la especificación del Json Schema correspondiente.

Tabla 7-5. CP Validar un Json válido de MenuBar

CP – 005	Validar un Json válido de un componente MenuBar
Descripción	Validar un Json válido correspondiente a un componente Vaadin de tipo MenuBar.
Resultado esperado	El Json se va a validar correctamente.

Tabla 7-6. CP Validar un Json no válido de MenuBar

CP – 006	Validar un Json no válido de un componente MenuBar
Descripción	Validar un Json no válido correspondiente a un componente Vaadin de tipo MenuBar.
Resultado esperado	El Json se va indicar como no válido debido a que no cumple la especificación del Json Schema correspondiente.

Tabla 7-7. CP Validar un Json válido de Table

CP – 007	Validar un Json válido de un componente Table
Descripción	Validar un Json válido correspondiente a un componente Vaadin de tipo Table.
Resultado esperado	El Json se va a validar correctamente.

Tabla 7-8. CP Validar un Json no válido de Table

CP – 008	Validar un Json no válido de un componente Table
Descripción	Validar un Json no válido correspondiente a un componente Vaadin de tipo Table.
Resultado esperado	El Json se va indicar como no válido debido a que no cumple la especificación del Json Schema correspondiente.

Tabla 7-9. CP Validar un Json válido de TabSheet

CP – 009	Validar un Json válido de un componente TabSheet
Descripción	Validar un Json válido correspondiente a un componente Vaadin de tipo TabSheet.
Resultado esperado	El Json se va a validar correctamente.

Tabla 7-10. CP Validar un Json no válido de TabSheet

CP – 010	Validar un Json no válido de un componente TabSheet
Descripción	Validar un Json no válido correspondiente a un componente Vaadin de tipo TabSheet.
Resultado esperado	El Json se va indicar como no válido debido a que no cumple la especificación del Json Schema correspondiente.

7.1.2. Casos de prueba para la traducción de Json a vistas Vaadin

Tabla 7-11. CP Traducir un Json de AbstractComponent

CP – 011	Traducir un Json de un componente de tipo AbstractComponent
Descripción	Traducir un Json de un componente de tipo AbstractComponent, al cual se le asignarán algunas propiedades para su posterior comprobación.
Resultado esperado	El Json se traduce correctamente a un objeto de tipo AbstractComponent y posee las propiedades declaradas en el Json.

Tabla 7-12. CP Traducir un Json de AbstractOrderedLayout

CP – 012	Traducir un Json de un componente de tipo AbstractOrderedLayout
Descripción	Traducir un Json de un componente de tipo AbstractOrderedLayout, al cual se le asignarán algunas propiedades y se le añadirá algún componente para su contendor.
Resultado esperado	El Json se traduce correctamente a un objeto de tipo AbstractOrderedLayout, posee las propiedades declaradas en el Json y además ha traducido correctamente los componentes que contiene.

Tabla 7-13. CP Traducir un Json de MenuBar

CP – 013	Traducir un Json de un componente de tipo MenuBar
Descripción	Traducir un Json de un componente de tipo MenuBar, al cual se le asignarán algunas propiedades y se le añadirán menú items.
Resultado esperado	El Json se traduce correctamente a un objeto de tipo MenuBar, posee las propiedades declaradas en el Json y además contiene los menú items declarados.

Tabla 7-14. CP Traducir un Json de Table

CP – 014	Traducir un Json de un componente de tipo Table
Descripción	Traducir un Json de un componente de tipo Table, al cual se le asignarán algunas propiedades y se le añadirán columnas.
Resultado esperado	El Json se traduce correctamente a un objeto de tipo Table, posee las propiedades declaradas en el Json y además contiene los columnas declaradas.

Tabla 7-15. CP Traducir un Json de TabSheet

CP – 015	Traducir un Json de un componente de tipo TabSheet
Descripción	Traducir un Json de un componente de tipo TabSheet, al cual se le asignarán algunas propiedades y se le añadirán tabs.
Resultado esperado	El Json se traduce correctamente a un objeto de tipo TabSheet, posee las propiedades declaradas en el Json y además contiene las tabs declaradas.

7.1.3. Casos de prueba para la serialización de componentes Vaadin

Tabla 7-16. CP Serialización de un AbstractComponent

CP – 016	Serialización de un AbstractComponent a Json
Descripción	Serializar un componente de Vaadin de tipo AbstractComponent a un Json interpretable por el framework. Se comprobará que el Json obtenido es válido para el framework.
Resultado esperado	El componente se ha serializado correctamente a Json y posee las propiedades que se le asignaron al objeto.

Tabla 7-17. CP Serialización de un AbstractOrderedLayout

CP – 017	Serialización de un AbstractOrderedLayout a Json
Descripción	Serializar un componente de Vaadin de tipo AbstractOrderedLayout a un Json interpretable por el framework. Se comprobará que el Json obtenido es válido para el framework.
Resultado esperado	El componente se ha serializado correctamente a Json y posee las propiedades que se le asignaron al objeto.

Tabla 7-18. CP Serialización de un MenuBar

CP – 018	Serialización de un MenuBar a Json
Descripción	Serializar un componente de Vaadin de tipo MenuBar a un Json interpretable por el framework. Se comprobará que el Json obtenido es válido para el framework.
Resultado esperado	El componente se ha serializado correctamente a Json y posee las propiedades que se le asignaron al objeto.

Tabla 7-19. CP Serialización de un TabSheet

CP – 019	Serialización de un TabSheet a Json
Descripción	Serializar un componente de Vaadin de tipo TabSheet a un Json interpretable por el framework. Se comprobará que el Json obtenido es válido para el framework.
Resultado esperado	El componente se ha serializado correctamente a Json y posee las propiedades que se le asignaron al objeto.

Tabla 7-20. CP Serialización de un Table

CP – 020	Serialización de un Table a Json
Descripción	Serializar un componente de Vaadin de tipo Table a un Json interpretable por el framework. Se comprobará que el Json obtenido es válido para el framework.
Resultado esperado	El componente se ha serializado correctamente a Json y posee las propiedades que se le asignaron al objeto.

7.2. Resultados casos de prueba

Los resultados que se muestran en la siguiente tabla se han obtenido una vez finalizado por completo la implementación del proyecto. Se ha tenido en cuenta si el test ha sido superado (OK), si no se ha superado correctamente (KO) y el tiempo empleado.

En cuanto al tiempo, no se mide el tiempo total empleado en el test, si no únicamente el tiempo que emplea el framework en realizar la operación correspondiente. Es decir, para los test de validación se mide el tiempo que tarda en validar el Json. Para los test de traducción se mide el tiempo que tarda el framework en traducir el Json a vista Vaadin, de este tiempo se están excluyendo las comprobaciones correspondientes para saber si el test se ha ejecutado correctamente. Y por último, para los test de serialización, se mide el tiempo que tarda en serializar el componente de Vaadin a Json, excluyendo el tiempo que tarda el test en realizar las comprobaciones de que se ha ejecutado correctamente. De modo, que con este tiempo, nos podemos hacer una idea del rendimiento del framework.

Tabla 7-21. Resultados casos de prueba

Pruebas	Tiempo	Resultado
CP – 001	0,007s	OK
CP – 002	0,003s	OK
CP – 003	0,041s	OK
CP – 004	0,005s	OK
CP – 005	0,013s	OK
CP – 006	0,005s	OK
CP – 007	0,008s	OK
CP – 008	0,003s	OK

CP – 009	0,015s	OK
CP – 010	0,004s	OK
CP – 011	0,055s	OK
CP – 012	0,062s	OK
CP – 013	0,038s	OK
CP – 014	0,071s	OK
CP – 015	0,047s	OK
CP – 016	0,003s	OK
CP – 017	0,007s	OK
CP – 018	0,013s	OK
CP – 019	0,009s	OK
CP – 020	0,018s	OK

Manual de desarrollador

8. Manual de desarrollador

8.1. Introducción

El objetivo de este manual es que el usuario desarrollador que utilice el framework aprenda a usarlo de manera correcta. A continuación se van a describir los distintos componentes de Vaadin que se van a poder utilizar. Todos estos componentes poseen unas características comunes, las cuales pueden usarse en cualquiera de ellos:

- Id. Identificador del componente.
- Caption. Encabezamiento del componente.
- Icon. El icono que se le puede añadir al componente.
- StyleName. El nombre del estilo CSS que se va a aplicar al componente.
- Enabled. Propiedad que indica si el componente está activo o no.
- ReadOnly. Indica si el componente va a ser de solo lectura o no.
- Visible. Indica si el componente va a ser visible o no.
- Width. Propiedad que indica la anchura del componente, ya sea en píxeles o porcentaje.
- Height. Propiedad que indica la altura del componente, ya sea en píxeles o porcentaje.

La propiedad 'id' posee una doble funcionalidad a parte de la que tiene en Vaadin. A partir del id se podrá obtener cualquier componente para realizar operaciones más específicas, como poder añadir un `ClickListener` a un botón o un `ValueChangeListener` a un campo de texto. Las propiedades comunes a todos los componentes se describen con más detalle a continuación.

Propiedad	Tipo	Ejemplo
id	String	tfNombre
caption	String	Nombre de usuario
description	String	Introduzca el nombre de usuario
iconPath	String	/icons/accept.png
styleName	String	login-textfield
enabled	Boolean	`true`
readOnly	Boolean	`false`
visible	Boolean	`true`
width	Object	`{"width" : { "width" : 150, "unit" : "px" }}`
height	Object	`{"height": { "height" : 30, "unit" : "px" }}`

Los Json que acepta el framework son validados mediante la especificación de Json Schema para comprobar que se pueden interpretar correctamente. La lista de Json Schemas utilizados para validar los componentes se encuentra en el punto 5.3. Descripción de la estructura de los Json interpretables.

8.2. AbstractComponent

Este tipo de componentes son los que denominamos componentes base. En la siguiente lista se detallan los distintos tipos de componentes AbstractComponent interpretables:

- Button
- CheckBox
- DateField
- TextField
- ComboBox

Todos estos componentes, a parte de tener las propiedades comunes a todos, poseen una más, la propiedad `value`, la cual tiene la misma función que la propiedad con el mismo nombre de los componentes Vaadin.

Un ejemplo de Json de este tipo de componentes base, en este caso un TextField, sería el siguiente.

```
{
    "TextField" : {
        "id" : "tfNombre",
        "caption" : "Nombre de usuario",
        "width" : {
            "width" : 150,
            "unit" : "px"
        }
    }
}
```

A este Json se le podrían añadir las características posibles deseadas, y cambiar el tipo del componente simplemente sustituyendo "TextField" por cualquiera de los componentes base.

8.3. AbstractOrderedLayout

En esta categoría se encuentran los componentes contenedores, es decir, que pueden contener a otro tipo de componentes e incluso a componentes del mismo tipo.

- `HorizontalLayout`. Componente en el que todo el contenido se coloca horizontalmente.
- `VerticalLayout`. Componente en el que todo el contenido se coloca verticalmente.

A parte de las características comunes a todos los componentes de Vaadin descritas en el primer punto, existen características específicas para este tipo de componentes:

- `spacing`. Propiedad de tipo Boolean con la cual podemos indicar que se incluya una pequeña separación entre los distintos componentes que contiene la vista.
- `margin`. Propiedad de tipo Boolean con la que podemos indicar que se incluyan márgenes en los laterales del layout.

Para añadir componentes al objeto contenedor se utiliza la propiedad `components`. Es una propiedad de tipo Array en la que se pueden incluir todo tipo de componentes compatibles con el framework. Un ejemplo de Json de este tipo de componentes podría ser el siguiente:

```
{
   "HorizontalLayout": {
      "id": "hlAcciones",
      "spacing": true,
      "margin": true,
      "width": { "width":100, "unit":"%" },
      "height": { "height":100, "unit":"%"},
      "components": [
         {
            "TextField": {
               "id": "tfNombre",
               "caption": "Nombre"
            }
         },
         {
            "TextField":{
               "id": "tfApellido",
               "caption": "Apellido"
            }
         },
         {
            "VerticalLayout": {
               "id": "vlBotones",
               "spacing": true,
               "components": [
                  {
                     "Button": {
                        "id": "btnAceptar",
                        "caption": "Aceptar"
                     }
                  },
                  {
                     "Button": {
                        "id": "btnCancelar",
                        "caption": "Cancelar"
                     }
                  }
               ]
            }
         }
      ]
   }
}
```

Por defecto Vaadin alinea todos los componentes arriba a la derecha, si se desea indicar cualquier otra alineación a los componentes que contiene un VerticalLayout o un HorizontalLayout, se puede utilizar la propiedad `alignment` de la siguiente manera:

```
"alignments" : [
      {
         "componentId" : "tfNombre",
         "alignment" : "MIDDLE_CENTER"
      },
      {
         "componentId" : "tfApellido",
         "alignment" : "MIDDLE_CENTER"
      },
      {
         "componentId" : "vlBotones",
         "alignment" : "MIDDLE_CENTER"
      }
]
```

Los distintos tipos de alineaciones posibles son los siguientes:

- `TOP_LEFT`
- `TOP_CENTER`
- `TOP_RIGHT`
- `MIDDLE_LEFT`
- `MIDDLE_CENTER`
- `MIDDLE_RIGHT`
- `BOTTOM_LEFT`
- `BOTTOM_CENTER`
- `BOTTOM_RIGHT`

Al incluir dos componentes dentro de un `AbstractOrderedLayout`, Vaadin por defecto asigna la mitad del espacio para cada uno de ellos. Si se desea asignar un expand ratio diferente a alguno de los elementos, se puede utilizar la propiedad `expandRatio` de la siguiente manera:

```
"expandRatio" : [
    {
    "componentId" : "vlBotones",
        "expandRatio" : 1
    }
]
```

Las propiedades `expandRatio` y `alignment` usan la propiedad id para hacer referencia a los componentes declarados en el Json, de esta manera se identifica a que componente se le desea asignar un `expandRatio` o un `alignment` que no sea el que Vaadin aplica por defecto.

8.4. MenuBar

Otro de los componentes compatibles es el `MenuBar` de Vaadin. A parte de las propiedades disponibles para todos los componentes de este framework, se añade la propiedad `menuItems`. Se trata de una propiedad de tipo array en la que se declararán los `MenuItem` del MenuBar. Un ejemplo de Json interpretable a `MenuBar` podría ser el siguiente:

```
{
    "MenuBar": {
        "id" : "menuBarPrincipal",
        "width" : {
            "width" : 100,
            "unit" : "%"
        },
        "height" : {
            "height" : 50,
            "unit" : "px"
        },
        "menuItems" : [
            {
                "text" : "Administración"
            },
            {
                "text" : "Usuario"
            }
        ]
    }
}
```

Los menú items a su vez poseen una serie de propiedades propias:

- `text`: Propiedad de tipo String que indica el texto que se quiere mostrar en el menú item.
- `styleName`: Propiedad de tipo String que indica el nombre del estilo CSS que se le quiere aplicar.
- `iconPath`: Propiedad de tipo String que indica la ruta al icono que se quiere añadir junto al texto del menú item.
- `separator`: Propiedad de tipo boolean que indica si se desea añadir un separador justo encima del menú item.
- `enabled`: Propiedad de tipo boolean que indica si el menú item está activo o no.
- `visible`: Propiedad de tipo boolean que indica si el menú item es visible o no.
- `menuItems`: A su vez, un menú item puede tener submenús, por lo que esta propiedad de tipo array permite añadir menús secundarios.

8.5. Table

Otro componente de Vaadin que podemos utilizar son las tablas. Al igual que los Json de tipo `MenuBar`, las tablas añaden una nueva propiedad que se llama `columns`. Con esta propiedad se pueden definir las columnas de la tabla. Un ejemplo de Json que se traduce a una tabla de Vaadin podría ser el siguiente:

```
{
    "Table": {
        "id" : "tblUsuarios",
        "width" : {
            "width" : 100,
            "unit" : "%"
        },
        "height" : {
            "height" : 100,
            "unit" : "%"
        },
        "columns" : [
            {
                "propertyId" : "codUsuario",
                "header" : "Código Usuario",
                "columnType" : "String",
                "defaultValue" : ""
            },
            {
                "propertyId" : "nombre",
                "header" : "Nombre",
                "columnType" : "String",
                "defaultValue" : ""
            },
            {
                "propertyId" : "apellidos",
                "header" : "Apellidos",
                "columnType" : "String",
                "defaultValue" : ""
            }
        ]
    }
}
```

Los objetos columnas poseen las siguientes propiedades:

- `propertyId`: Propiedad de tipo String que indica el nombre de la atributo del objeto que se desea mostrar en la columna.
- `header`: Propiedad de tipo String que indica el nombre que se va a mostrar como título de la columna.
- `columnType`: Propiedad de tipo enum que indica el tipo de la columna. Los tipos posibles son: `String`, `Integer`, `Float`, `Double`, `Boolean`.
- `defaultValue`: Propiedad de tipo String que indica el valor por defecto que se va a mostrar en la tabla si el valor es nulo.

8.6. TabSheet

El último de los componentes compatibles son las `TabSheet`. La única propiedad que añaden las `TabSheet` es la propiedad `tab`. Con esta propiedad podemos añadir tantas tabs como se deseen al `TabSheet`. Un ejemplo de Json interpretable a TabSheet sería el siguiente:

```
{
    "TabSheet": {
        "id" : "tabPrincipal",
        "width" : {
            "width" : 100,
            "unit" : "%"
        },
        "height" : {
            "height" : 100,
            "unit" : "%"
        },
        "tabs" : [
            {
                "caption" : "Primera",
                "content" : {
                    "Button" : {
                        "id" : "btnAceptar",
                        "caption" : "Aceptar"
                    }
                }
            }
        ]
    }
}
```

Los objetos Tab tienen las siguientes propiedades:

- `caption`. El titulo de la tab.
- `icon`. El path al icono que se incluirá al lado del caption.
- `content`. El contenido de la tab. Dentro de este contenido puede ir un objeto de cualquier tipo, siempre que sea compatible con el framework.

8.7. Importación del framework y uso

Para poder importar el framework al proyecto en el que se desea usar necesitamos añadir el jar del framework al classpath del proyecto. Una de las formas más sencillas de hacer esto es utilizando Maven.

Como el jar no está en los repositorios de Maven central tenemos que instalarlo en nuestro repositorio local de Maven. Para ello, Maven proporciona un comando para poder instalar en el repositorio librerías locales, en nuestro caso habría que ejecutar el siguiente comando:

```
mvn install:install-file
      -Dfile=”ruta al .jar del framework”
      -DgroupId=es.uva.tfg.layout.generator
      -DartifactId=vaadin-json-converter
      -Dversion=1.0.0.RELEASE
      -Dpackaging=jar
      -DgeneratePom=true
```

Si todo ha ido correcto debería aparecer en la consola lo siguiente:

```
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
[INFO] Total time: 0.564 s
[INFO] Finished at: 2016-05-21T13:55:26+02:00
[INFO] Final Memory: 6M/77M
[INFO] ------------------------------------------------------------
```

Una vez instalada la dependencia en Maven, vamos a crear un proyecto nuevo en el que vamos a usar Spring boot, Vaadin y el framework desarrollado. Con Spring boot vamos a levantar un servidor en el que se ejecute nuestra aplicación Vaadin. Que no cunda el pánico si nunca antes se ha trabajado con Spring, va a ser un proyecto muy sencillo en el que un inicializador nos cree el esquema del proyecto.

Spring nos proporciona un inicializador de proyectos muy completo con el cual podemos crear la estructura inicial sin ningún esfuerzo, se puede acceder a él en start.spring.io. Una vez dentro nos pide una serie de datos como el group y artifact, pero antes de esto vamos a seleccionar la opción de la versión completa pinchando en el enlace que se encuentra justo debajo del botón de generar proyecto.

Para este ejemplo vamos a utilizar los siguientes datos

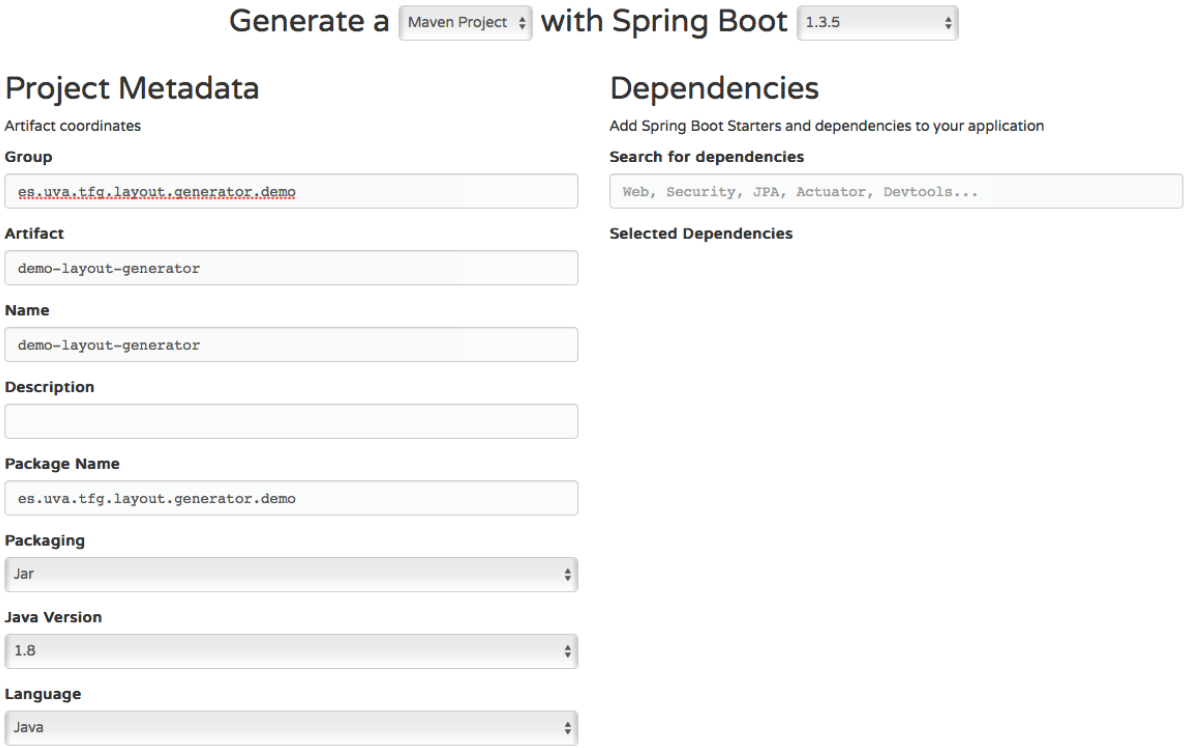

Figura 8-1. Spring Initializr

A parte de la configuración de Maven, vamos a indicarle al inicializador que nos cree un proyecto con Vaadin, para ello solamente tenemos que marcar la opción de Vaadin dentro de las opciones Web disponibles

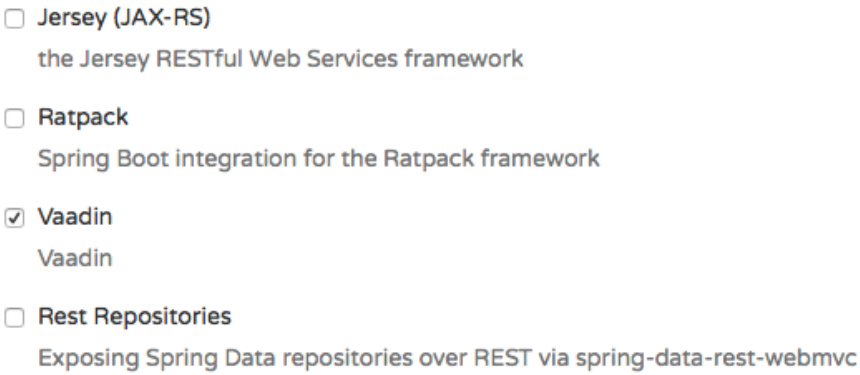

Figura 8-2. Marcar opción Vaadin en el Spring Initializr

Una vez configurado todo lo comentado en el inicializador, pulsamos el botón de generar proyecto, el cual nos va a descargar un zip que contiene el proyecto Maven. Descomprimimos el zip y copiamos el contenido a nuestro workspace de Eclipse. Una vez que el proyecto se encuentra en el workspace de Eclipse, vamos a iniciar Eclipse e importar el proyecto creado.

La importación del proyecto es muy sencilla, seleccionamos en el menú superior "Archivo" o "File" y dentro de este la opción "Importar" o "Import". Nos aparecerá una pantalla como la siguiente.

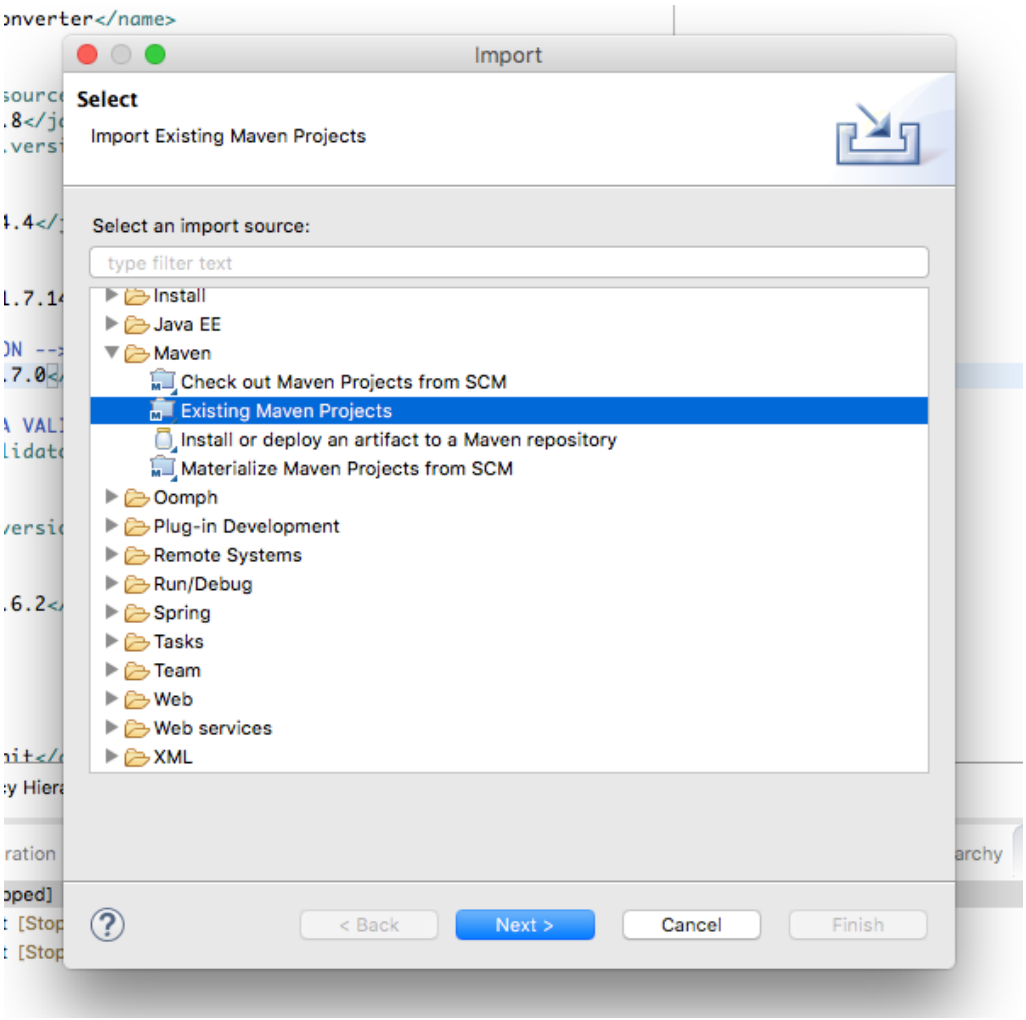

Figura 8-3. Importar proyecto Maven

Seleccionamos la opción de "Existing Maven Projects" y buscamos el proyecto dentro del workspace. Una vez importado veremos nuestro proyecto en la pestaña de "Package Explorer" junto al resto de proyectos que tengamos abiertos en ese momento.

El inicializador nos ha creado la clase principal de nuestra aplicación Spring. Como vemos, es una clase muy sencilla con un simple método main, el cual se encarga de arrancar Spring Boot. Como se comentaba anteriormente, no hace falta conocer nada sobre Spring, ya que no vamos a necesitar saber mucho más para el ejemplo que vamos a crear.

Lo siguiente que vamos a hacer va a ser crear la UI principal de Vaadin en la que vamos a añadir las vistas que queramos convertir. Vamos a crear un paquete nuevo en el que vamos a introducir todo lo relacionado con la vista, en nuestro caso se va a llamar "`es.uva.tfg.layout.generator.demo.view`". Una vez

creado, vamos a crear la clase de la UI principal, la vamos a llamar `MainUI`. En este punto, nuestra estructura del proyecto sería la siguiente.

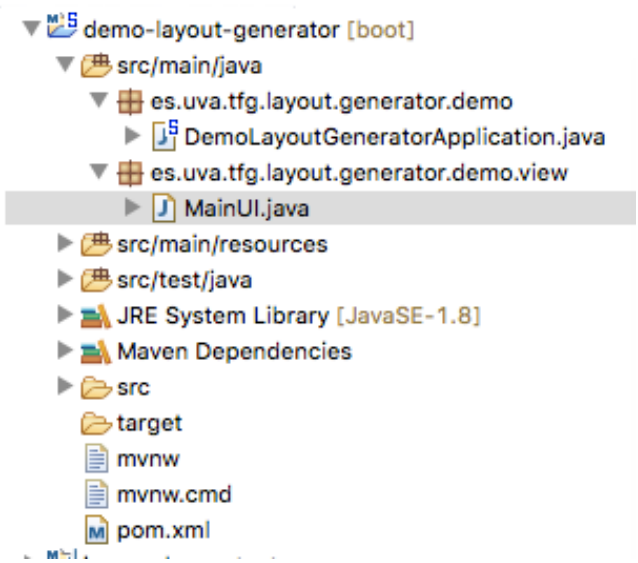

Figura 8-4. Estructura del proyecto demo

Nuestra clase `MainUI` va a extender la clase `UI` de Vaadin, de esta manera estamos indicando que va se va a tratar de una `UI` de Vaadin. Al extender de `UI`, nos va a obligar a que implementemos el método `void init(VaadinRequest request)`. Antes de ponernos a implementar el método, vamos a anotar nuestra clase con las siguientes anotaciones:

- `@SpringUI`. Gracias a esta anotación, Spring boot va a descubrir de forma automática que nuestra clase `MainUI` se trata de una UI de Vaadin.
- `@Theme("valo")`. Esta anotación está indicándole a Vaadin que en nuestra UI vamos a utilizar el tema predefinido de Vaadin con nombre Valo.

Antes de escribir más código, vamos a crear el Json que será nuestra vista para el ejemplo. En este caso vamos a crear una vista que simule el login de una aplicación. Para ayudarnos a diseñar la vista, podemos usar una demo que se ha desarrollado para estos casos, simplemente tenemos que ir a vaadinjsonconverterdemo.eu-west-1.elasticbeanstalk.com y pegar el Json en nuestro editor para ver el resultado.

Un ejemplo de Json para crear una pantalla de login podría ser el siguiente.

```
{
  "VerticalLayout": {
    "width": {
      "width": 100,
      "unit": "%"
    },
    "height": {
      "height": 100,
      "unit": "%"
    },
    "components": [
      {
        "VerticalLayout": {
          "id": "vlContenedor",
          "spacing" : true,
          "components": [
            {
              "VerticalLayout": {
                "id": "vlTextFields",
                "components": [
                  {
                    "TextField": {
                      "id": "tfUsuario",
                      "caption": "Usuario"
                    }
                  },
                  {
                    "TextField": {
                      "id": "tfPassword",
                      "caption": "Contraseña"
                    }
                  }
                ],
                "alignments": [
                  {
                    "componentId": "tfUsuario",
                    "alignment": "MIDDLE_CENTER"
                  },
                  {
                    "componentId": "tfPassword",
                    "alignment": "MIDDLE_CENTER"
                  }
                ]
              }
            },
            {
              "HorizontalLayout": {
                "id": "hlButtons",
                "spacing": true,
                "components": [
                  {
                    "Button": {
                      "id": "btnAceptar",
                      "caption": "Aceptar"
                    }
                  },
```

...

```
                    {
                      "Button": {
                        "id": "btnCancelar",
                        "caption": "Cancelar"
                      }
                    }
                  ]
                }
              }
            ],
            "alignments": [
              {
                "componentId": "vlTextFields",
                "alignment": "MIDDLE_CENTER"
              },
              {
                "componentId": "hlButtons",
                "alignment": "MIDDLE_CENTER"
              }
            ]
          }
        }
      ],
      "alignments": [
        {
          "componentId": "vlContenedor",
          "alignment": "MIDDLE_CENTER"
        }
      ]
    }
}
```

Para ver el resultado de este Json podemos pegarlo en el editor online.

Figura 8-5. Resultado Json login

Se trata de un login sencillo en formato vertical, pero para entender el funcionamiento del framework nos es suficiente.

Ahora que ya tenemos el Json que queremos traducir, lo vamos a guardar a un fichero llamado `login.json` y lo copiamos dentro de la carpeta de recursos 'templates' del proyecto. En nuestra clase `MainUI` vamos a crear una función privada que se va a encargar de leer el fichero Json y traducirlo a una vista de Vaadin.

```
private JSONAbstractOrderedLayout getJsonComponent() {
      JSONAbstractOrderedLayout result = null;

      JSONTranslater translater = new JSONTranslater();

      try {
            JsonNode loginJson = JsonLoader
                  .fromResource("/templates/login.json");

            translater.setJson(loginJson);
            result = (JSONAbstractOrderedLayout)
                  translater.translate();
      } catch (IOException e) {
            LOG.error("Error al leer el fichero json: " +
                  e.getMessage());
      } catch (JSONValidationException e) {
            LOG.error("Json con formato incorrecto: " +
                  e.getMessage());
      }
      return result;
}
```

Lo primero que hacemos es inicializar el `JSONTranslater`, el cual se encarga de traducir el Json. Lo siguiente es generar un objeto Json a partir del fichero, gracias a la clase `JsonLoader` podemos leer un fichero y transformarlo a un `JsonNode` de una manera muy sencilla. Una vez que tenemos el Json como objeto de tipo `JsonNode` se lo indicamos al traductor y le decimos que lo procese para que nos devuelva el `JSONComponent` correspondiente. En este caso como el objeto padre de la jerarquía de componentes es un VerticalLayout, sabemos que nos va a devolver un componente de tipo `JSONAbstractOrderedLayout`, por lo que podemos hacer casting a un objeto de este tipo sin ningún problema.

Una vez que tenemos la función que lee el fichero Json y lo transforma a un componente de Vaadin, vamos a añadir el componente creado a nuestra UI principal. Para ello vamos a implementar la función `void init(VaadinRequest request)` que nos pide implementar la clase `UI` de Vaadin.

```
@Override
protected void init(VaadinRequest request) {
	VerticalLayout content = new VerticalLayout();
	content.setSizeFull();

	JSONAbstractOrderedLayout login = getJsonComponent();
	content.addComponent(login.getVaadinComponent());

	setContent(content);
	setSizeFull();
}
```

Lo único que hacemos es llamar a la función que acabamos de crear y obtener el componente Vaadin padre para añadírselo a un VerticalLayout que le hemos dicho que ocupe toda la pantalla posible.

Una vez implementado la función `init()`, podemos arrancar la aplicación para ver el resultado. Gracias a Spring Boot no necesitamos disponer de un contenedor de aplicaciones, como podría ser Tomcat, para que arrancar la aplicación web. Por defecto arranca un contenedor de aplicaciones embebido en el que despliega nuestra aplicación. Para arrancar la demo, botón derecho sobre nuestra clase principal del proyecto, `DemoLayoutGeneratorApplication.java`, y seleccionar la opción de "Run as -> Java Application". Para ver el resultado, abrimos nuestro navegador y abrimos la pagina localhost:8080.

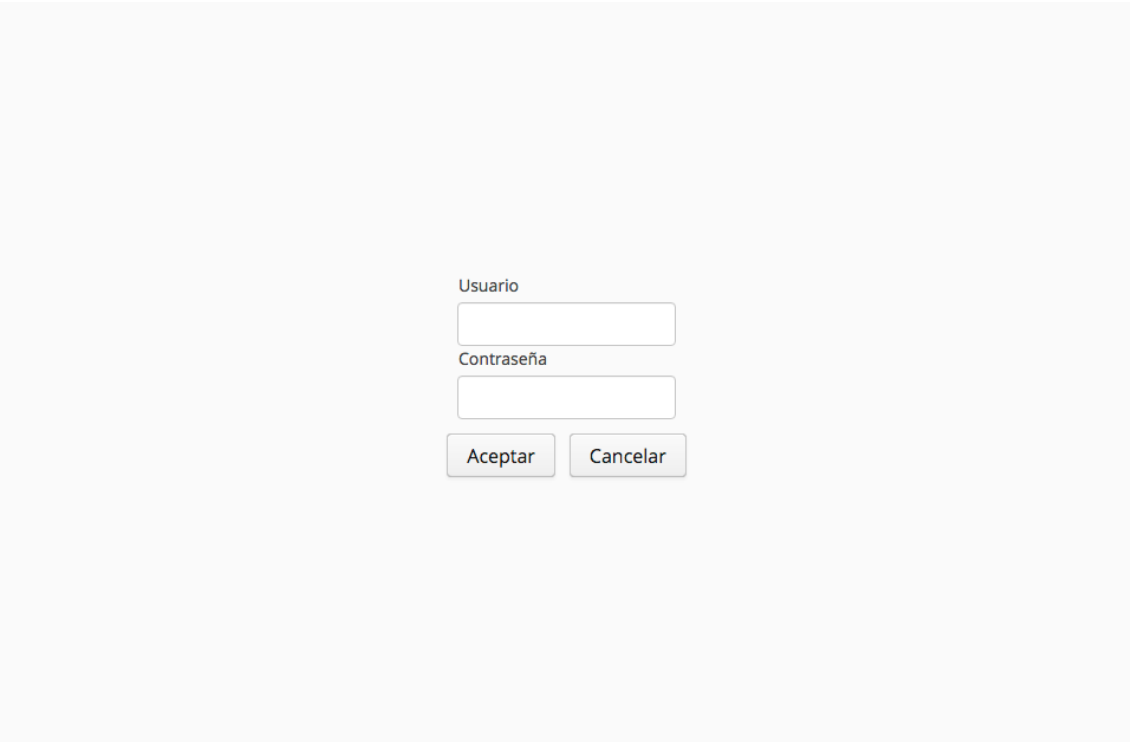

Figura 8-6. Resultado en el navegador del login

Una vez creado el esqueleto de la vista podemos aplicar funcionalidad a los botones, e incluso agregar iconos al encabezamiento de los `TextField` para mejorar un poco su apariencia. Como hemos asignado identificadores a las entradas de texto y a los botones, podemos obtenerlos para realizar cualquier operación disponible en Vaadin sobre los componentes.

```
TextField tfUsuario = (TextField) result
      .getVaadinComponent("tfUsuario");
TextField tfPassword = (TextField) result
      .getVaadinComponent("tfPassword");
Button btnAceptar = (Button) result
      .getVaadinComponent("btnAceptar");
Button btnCancelar = (Button) result
      .getVaadinComponent("btnCancelar");

tfUsuario.setIcon(FontAwesome.USER);
tfPassword.setIcon(FontAwesome.LOCK);

btnAceptar.addClickListener(event -> Notification
      .show("Has seleccionado el botón aceptar"));
btnCancelar.addClickListener(event -> Notification
      .show("Has seleccionado el botón de cancelar"));
```

Añadiendo este fragmento de código a la función `getJsonComponent()` conseguiremos que los encabezamientos de los campos de texto posean un icono y que los botones muestren una notificación cuando son seleccionados.

Para visualizar el código completo del ejemplo desarrollado, descargar el siguiente repositorio bitbucket.org/abemart/demo-layout-generator.

Conclusiones

9. Conclusiones

La realización de este proyecto ha sido todo un reto para poner en práctica los conocimientos adquiridos durante el transcurso del grado.

Hay que destacar que realizar un análisis completo durante la etapa inicial del proyecto es complicado, y que mientras se va avanzando en el desarrollo, van surgiendo nuevas ideas o funcionalidades para la mejora del proyecto. Es una cualidad que distingue los proyectos software de cualquier otro tipo de proyectos, ya que todo desarrollo ha presentado avances y mejores desde su versión inicial, lo cual no podría pasar en un proyecto arquitectónico, en el que todo debe estar planteado desde el principio.

Una de las partes más complicadas cuando no se tiene mucha experiencia en la planificación de proyectos, es la estimación temporal de este. Pueden surgir situaciones no esperadas que retrasen el desarrollo del proyecto, y por lo tanto necesiten de más dedicación si no se quiere retrasar la fecha de entrega.

9.1. Trabajo futuro

Como se comentaba en el punto anterior, durante el desarrollo surgen ideas para realizar nuevas funcionalidades o mejoras. A continuación se indica una lista de los puntos deseados para un posible trabajo futuro:

- Incluir más componentes compatibles para poder ser traducidos.
- Poder dividir una vista compleja en varios archivos para una interpretación más sencilla para el desarrollador.
- Incluir el framework en el repositorio central de Maven.
- Conseguir que el framework forme parte de los addons gratuitos de Vaadin.

Bibliografía

10. Bibliografía

[1] Apache Software Foundataion – Apache Maven, "The central repository", disponible en: http://search.maven.org, (última visita: mayo de 2016)

[2] Erich Gamma, Ralph Johnson, Richard Helm, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Adidison-Wesley 1994.

[3] Github – fge, "Json Schema Validator", disponible en: https://github.com/fge/json-schema-validator, (última visita: marzo de 2016)

[4] Github – Everit.org, "Json Schema", disponible en: https://github.com/everit-org/json-schema, (última visita: enero de 2016)

[5] Github – FasterXML, "Jackson Databind", disponible en: https://github.com/FasterXML/jackson-databind, (última visita: marzo de 2016)

[6] Github – Google, "Gson", disponible en: https://github.com/google/gson, (última visita: marzo de 2016)

[7] Json Schema, "Json Schema: The home of JSON Schema", disponible en: http://json-schema.org, (última visita: enero de 2016)

[8] Spacetelescope, "Structuring a complex schema", disponible en: http://spacetelescope.github.io/understanding-json-schema/structuring.html, (última visita: marzo de 2016)

[9] Source Making, "Design Patterns" disponible en: https://sourcemaking.com/design_patterns/, (última visita: abril de 2016)

[10] Vaadin Ltd., "Vaadin: Vaadin Framework", disponible en: https://vaadin.com/home, (última visita: noviembre de 2015)

[11] Vaadin Ltd., "Vaadin Book: Vaadin Book online Version", disponible en: https://vaadin.com/book/-/page/preface.html, (última visita: enero de 2015)

[12] Wikipedia, "Patrón de diseño", disponible en: https://es.wikipedia.org/wiki/Patrón_de_diseño, (última visita: abril de 2016)

[13] Wikiversidad, "Gestión de riesgos de proyectos software", disponible en: https://es.wikiversity.org/wiki/Gestión_de_riesgos_de_proyectos_software, (última visita: diciembre de 2015)