PROYECTO FINAL DE INGENIERÍA

ESPECIFICACIÓN DE FORMATO DEL INFORME ESCRITO DEL PROYECTO FINAL DE INGENIERÍA

Cornazzani, Agustina – LU98170

Ingeniería Informática

Berge, Daniel – LU98126

Ingeniería Informática

Tutor:

Orosco, Ricardo Fabián, UADE

Octubre 15, 2012

UNIVERSIDAD ARGENTINA DE LA EMPRESA

FACULTAD DE INGENIERÍA Y CIENCIAS EXACTAS

Resumen

Balsamiq Mockups es una aplicación utilizada, entre otras cosas, para el diseño de prototipos de pantallas. A través de una interface gráfica, muy sencilla de utilizar, permite el diseño de prototipos de pantallas relacionadas con aplicaciones de escritorio, web o destinadas a dispositivos móviles. Las pantallas diseñadas se almacenan en archivos de extensión *bmml*, un lenguaje propio de la aplicación, basado en *xml*, que puede ser interpretado para obtener la información del diseño creado. Una vez interpretado el diseño, contamos con la información relacionada a cada uno de sus componentes, pudiendo determinar a través del análisis y comparación de los datos, el tamaño y ubicación de los componentes. Del análisis anterior, también se determinará que componentes se comportan como contenedores de otros componentes, de esta manera, construiremos una estructura de objetos que represente a los componentes presentes en el archivo y que aplique el patrón de diseño composite, un patrón utilizado principalmente en la creación de pantallas. A través de la definición de un lenguaje, también basado en *xml*, definiremos los *templates*, es decir, archivos que nos permitirán definir como queremos que se represente cada uno de los componentes presentes en el diseño. Los *templates* también permitirán definir los aspectos relacionados a la ubicación de componentes dentro de los contenedores y la estructura final del archivo generado. De esta manera, el archivo *bmml* representará el *"que"* deseamos generar y el archivo de *template* definirá el *"como"*. El lenguaje de *templates*, permitirá definir la representación de componentes en dos paradigmas de programación completamente distintos como *HTML* y *Java*, de esta manera, un diseño realizado con *Balsamiq Mockups* se podrá representar en cualquiera de estos dos lenguajes y con diferentes estilos. La aplicación de generación de código, a través de su interface gráfica, permitirá seleccionar un diseño *bmml* y un *template*, para dar después lugar al proceso de generación. La aplicación estará formada principalmente por servicios, es decir, clases que implementarán la lógica de interpretación y generación necesaria para todo el proceso. Dichas clases serán implementadas en forma de servicios para que puedan ser utilizadas en un futuro como *web services*. La capacidad de representación de nuevos componentes, no dependerá de la aplicación en sí misma, dado que dicha representación se define en los *templates*, habilitando la posibilidad de modificar los mismos para contemplar nuevos componentes.

Abstract

Balsamiq Mockups is mainly used for application prototyping screens. Through a graphical interface, very easy to use, enables the design of prototypes of different types of screens like desktop applications, web or mobile device. Designed screens are stored in bmml extension, which is a language of the application itself based on *XML* that can be interpreted to obtain information related to created screens. Once interpreted the design, we have the information related to each of its components, such as the size and location of components and containers. From the above analysis, we can determine that components behave as containers for other components. Through this analysis we can build an object structure representing the components present in the file and applying composite design pattern, a pattern mainly used in screen creation. Through the definition of a language, also based on *XML*, we can build template files, which are files that allow us to define how we want to represent each of the components present in the design. The templates also allow defining aspects related to the location of components within the container and the final structure of the generated file. Thus, the file bmml represent the "what" we want to generate while the template file define the "how". The language used to create templates, will define the representation of components in two completely different programming paradigms such as *HTML* and *Java*, thus designs made with *Balsamiq Mockups* may be displayed in either of these languages and different styles. The application used to generate source code, through its graphical interface, allow you to select a design template and a *bmml* archive to begin the process of generation. The application will consist mainly of services, which are classes that implement the logic of interpretation and generation needed for the whole process. These classes will be implemented as services that can be used in the future as web services. The ability to represent new components, not depend on the application itself, since this representation is defined in the templates, enabling the ability to modify them to use new components.

Indice

Introducción

Balsamic Mockups es una aplicación utilizada en el diseño de pantallas de aplicaciones de escritorio, aplicaciones móviles y páginas web, entre otras cosas. Una de las principales característica de esta herramienta, se debe a que los diseños son realizados en un entorno gráfico que simula el dibujo a mano alzada. La herramienta contiene una gran cantidad de elementos que simulan los componentes utilizados en distintos tipos de aplicaciones, permitiendo fácilmente el diseño de pantallas y contemplando distintas alternativas.

Objetivos

El objetivo de este proyecto final, consiste en el desarrollo de una aplicación que permita generar automáticamente el código fuente de las pantallas diseñadas con *Balsamiq Mockups*, contemplando la posibilidad de generar una implementación real de las pantallas en diferentes lenguajes de programación y con diferentes estilos. Se contempla la definición de un lenguaje, basado en *xml*, para la definición de la representación de los componentes en diferentes lenguajes y de diversas maneras. El alcance del proyecto, se limitará a la representación de los componentes en dos lenguajes diferentes, como son *HTML* y *Java*, y que cubren dos paradigmas de programación completamente diferentes.

Estado del Arte

Actualmente existen proyectos que permiten la generación de código a partir de *Balsamiq Mockups*, algunos de ellos, todavía se encuentran en proceso de desarrollo. A continuación describiremos algunas de las aplicaciones existentes detallando sus principales características. Uno de los proyectos, pensados para la generación de código a partir de

Balsamiq Mockups, está integrado a la nueva versión del mismo, limitándose únicamente a la generación de código fuente basado en *Flex*, un lenguaje perteneciente a la plataforma *Flash*. Dicho proyecto, pertenece a la compañía de software *Midnight Coders* (http://www.themidnightcoders.com/develop-ria/balsamiqapp), y en el sitio web de la misma, se describe los pasos a seguir para poder generar código fuente *Flex* a partir de los mockups. Cabe destacar, que el código fuente generado, está diseñado para ser instalado en un servidor de integración *WebORB*, un producto perteneciente a esta misma compañía.

Otro proyecto, es una aplicación llamada *Napkee* (http://www.napkee.com/) completamente desarrollada en *Flash*. Se trata de una aplicación que permite la generación de pantallas con la posibilidad de selección de dos tipos de salidas: *HTML* y *Flex*. El código generado en *HTML* incluye la definición de algunas funciones básicas de *Javascript*, al igual que un estilo visual predefinido por la aplicación y basado en *CSS*. La aplicación permite modificar los estilos CSS que se aplicarán al código fuente de salida y visualizar los mismos antes de generar la pantalla de salida. *Napkee* es una aplicación paga y sin la posibilidad de extensión.

Otro desarrollo destacable, es una aplicación de *Isomorphic Software*, llamada *Riefy* (http://blogs.balsamiq.com/product/2013/01/08/go-from-mockup-to-code-with-reify/). Dicha aplicación permite la generación de código basado en *SmartClient* y *SmartGWT*, dos frameworks de esta misma empresa. Dichos frameworks, proveen componentes predefinidos pensados para integrarse de forma más fácil con aplicaciones del lado del servidor, por esta razón, se debe tener en cuenta, que el código fuente generado está limitado a dichos frameworks y los componentes que son capaces de reproducir.

Teniendo en cuenta los proyectos descriptos anteriormente, se puede concluir que los mismos, no son del tipo *Open Source* y por lo tanto no presentan la posibilidad de ser extendidos por terceros. Este es un punto importante, dado que *Balsamiq Mockups* tiene la capacidad de incorporar nuevos componentes y esto puede ser un inconveniente en aquellos proyectos que generan código fuente a partir de un framework o una arquitectura predeterminada, como puede ser el caso de *Riefy*. Algunas de las aplicaciones son pagas y presentan limitaciones a la hora de seleccionar el tipo de código a generar, dado que no permiten incorporar variantes al código de salida generado, más allá de *HTML* o *Flex*. El propósito de nuestro proyecto se basa en la posibilidad de generar pantallas, a través de

Balsamiq Mockups, contemplando distintos lenguajes de programación, es decir, habilitando la posibilidad de transformar, las pantallas diseñadas, en implementaciones reales codificadas en distintos lenguajes de programación. El principal aporte de este proyecto, es crear una herramienta extensible, que contemple la posibilidad de agregar nuevos componentes, y permita también, representar los existentes de otra forma diferente, es decir, que se pueda redefinir la representación de los mismos. Otro punto importante a tener en cuenta, es la posibilidad de extender la aplicación agregando los componentes necesarios para habilitar la posibilidad de generar código fuente en otros lenguajes.

Descripción de la estructura del informe

Comenzaremos a desarrollar el informe describiendo algunas de las características de *Balsamiq Mockups*, refiriéndonos principalmente a los aspectos relacionados al diseño de pantallas. Posteriormente, se describirán la interpretación y validación de los diseños generados, explicando los problemas encontrados y el motivo de las validaciones. Una vez finalizada la descripción del proceso de interpretación y validación, desarrollaremos el concepto de *templates*, utilizado para definir la representación de los componentes y su ubicación. La arquitectura del generador y la descripción de la interface gráfica de usuario serán los últimos dos puntos a desarrollar, y describirán los detalles de la implementación de los conceptos explicados en los puntos anteriores. Finalmente se presentará un anexo para profundizar el concepto de representación de componentes avanzados a través de otras librerías gráficas.

Diseño de pantallas con Balsamiq Mockups

A continuación se describirán algunos de los detalles más relevantes relacionados a las pantallas diseñadas con *Balsamiq Mockups* y el archivo de salida generado por dicha aplicación.

La aplicación *Balsamiq Mockups* permite diseñar fácilmente diferentes tipos de pantallas, dichas pantallas pueden representar diseños de aplicaciones de software donde se puede observar con facilidad la ubicación de los diferentes componentes que integran la misma. Esta característica, convierte a *Balsamiq Mockups*, en una excelente opción a la hora de diseñar prototipos de pantallas. Las pantallas diseñadas pueden representar distintos tipos de aplicaciones ideadas para funcionar en diferentes tipos de plataformas y esto es posible, gracias a la presencia de distintos tipos de componentes que representan las variantes que pueden presentarse en las plataformas más habituales. La siguiente figura muestra un diseño de pantalla realizado con *Balsamiq Mockups.*

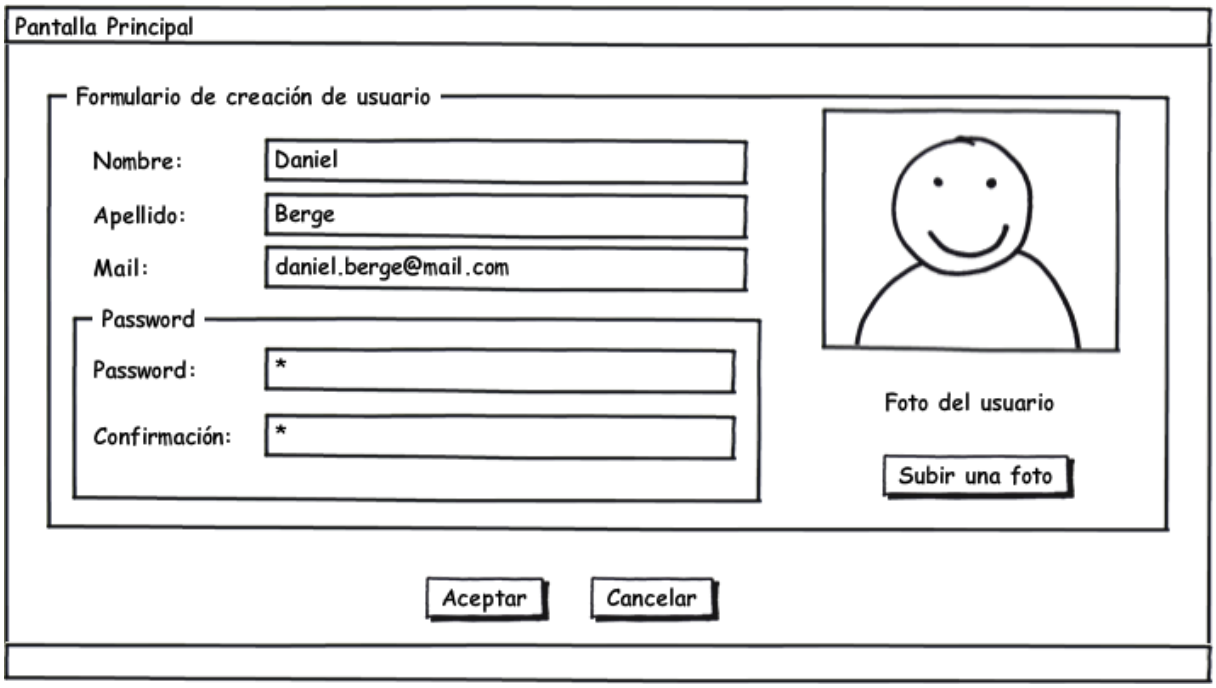

Figura 1: Ejemplo de diseño de pantalla

Como se puede observar en la *Figura 1*, los diseños de pantallas realizados con *Balsamiq Mockups* presentan un aspecto final de prototipo, esto se debe principalmente al aspecto de sus componentes los cuales parecen realizados a mano alzada. Esta característica, junto al hecho de que es muy sencillo de utilizar, lo convierten en una excelente herramienta de diseño y esa es una de las principales razones que originan la necesidad de poder convertir estos diseños en implementaciones reales. Una vez finalizado el diseño de una pantalla, sería de gran utilidad poder interpretar la información presente en el archivo de salida, para poder generar una implementación concreta de dicha pantalla.

Teniendo en cuenta, que el objetivo final es la generación del código fuente que representará un diseño de pantalla, deberemos interpretar como primera instancia, el formato de salida y la información presente en el archivo generado por *Balsamiq Mockups*. Las pantallas diseñadas con *Balsamiq Mockups* se almacenan en archivos de extensión *bmml*, estos archivos contienen internamente una estructura de tipo *xml*, que puede ser analizada e interpretada por cualquier intérprete o parser *xml*. La información contenida en el archivo, permite identificar los componentes que forman parte de la pantalla diseñada, como así también, las características de los mismos. A continuación, se presenta la estructura de uno de estos archivos, el cual representa, un diseño de pantalla muy básico.

```
<mockup version="1.0" skin="sketch" measuredW="553" measuredH="407">
  <controls>
    <control controlID="0" controlTypeID="com.balsamiq.mockups::TitleWindow" x="89" y="93" w="454"
      h="304" measuredW="450" measuredH="400" zOrder="0" locked="false" isInGroup="-1">
      <controlProperties>
        <text>Login</text>
        <topheight>26</topheight>
      </controlProperties>
    </control>
  </controls>
</mockup>
```

Figura 2: Estructura de un archivo *bmml*

Como se puede observar, el elemento raíz del xml se denomina *mockup*, dentro del mismo se puede observar un tag denominado *controls*, el cual será utilizado para contener a todos los controles presentes en la pantalla diseñada. El tag *control* define específicamente un componente, el mismo contiene distintos atributos que identifican y representan las diferentes características de un componente visual. A continuación, se describen algunos de los atributos más importantes con el fin de poder determinar cuáles son indispensables para el proceso de generación de código:

controlTypeID: Define el tipo de componente, este atributo puede ser utilizado para determinar de qué tipo de control se trata (Caja de texto, Etiqueta, Botón, etc…)

x: Determina la posición del componente en el eje de coordenadas x, este atributo puede ser utilizado para determina la posición del componente en la pantalla.

y: Determina la posición del componente en el eje de coordenadas y, este atributo puede ser utilizado para determina la posición del componente en la pantalla.

w: Es el ancho original del componente, es decir, el ancho que tiene el componente cuando se inserta en el diseño.

h: Es la altura original del componente, es decir, la altura que tiene el componente cuando se inserta en el diseño.

measuredW: Determina el ancho del componente y puede ser igual o no al valor del atributo *w*. Esta propiedad varía por cada componente, pero almacena el valor necesario para que el componente se visualice correctamente después de haber sido modificado, por ejemplo, si se trata de una etiqueta, el valor de *measuredW* contendrá el ancho indicado para que el texto de la etiqueta se lea correctamente. Cuando el valor del atributo *w* es igual a -1, se deberá tomar el ancho especificado en el atributo *measuredW*.

measuredH: Es similar a *measuredW*, pero contiene el alto indicado del componente para su correcta visualización.

El tag *controlProperties*, dentro del tag *control*, sirve para definir algunas características adicionales que solo poseen algunos componentes, es decir, no estará siempre presente y dependerá del diseño final de cada uno.

Figura 3: Los componentes del tipo Botón y Cuadro de texto

En la *Figura 3* se pueden observar dos componentes: un botón y un cuadro de texto, el tag *controlProperties* permite definir, entre otras cosas, los textos que aparecerán en cada uno de los componentes. En el caso del botón, el tag *control* quedará definido de la siguiente manera:

```
<control controlID="1" controlTypeID="com.balsamiq.mockups::Button" x="358" y="169" w="108" h="-1" measuredW="57" measuredH="28" zOrder="0" locked="false" isInGroup="-1">
   <controlProperties>
      <text>Botón</text>
   </controlProperties>
</control>
```

Figura 4: El tag *controlProperties*

No todos los componentes presentan un texto o etiqueta como el que presentan el botón y el cuadro de texto, por esa razón, mucho controles no tendrán definido un tag *controlProperties*.

Los atributos descriptos anteriormente permiten identificar la posición del componente, su tamaño, su contenido, y el tipo al cual pertenece el mismo, es decir, de qué tipo de componente se trata, otros atributos no fueron descriptos ya que son utilizados por *Balsamiq Mockups* para presentar los mismos en la pantalla del editor. Teniendo en cuenta la

interpretación de estos atributos, se podrán reproducir las características de los componentes en una pantalla real codificada con un lenguaje de programación.

Es necesario destacar, que existen otros atributos que no serán utilizados en el proceso de generación de código dado que solo son utilizados por *Balsamiq Mockups* para identificar características del componente en tiempo de diseño y no de visualización del mismo.

Debe tenerse en cuenta, que *Balsamiq Mockups* no almacena información sobre la jerarquía de los componentes, es decir, no importa si existen componentes dentro de otros, la aplicación simplemente almacena uno a uno los componentes, guardando la información de cada uno de ellos como los atributos de un tag *control* (atributos previamente explicados).

Cuando nos referimos a jerarquía, hacemos mención a la estructura de contenedores y componentes que generalmente tienen las pantallas de una aplicación. Existen contenedores que contienen y ordenan a un conjunto de componentes y estos contenedores pueden formar parte o estar contenidos en otros contenedores. La siguiente figura muestra el concepto de jerarquía al cual nos referimos.

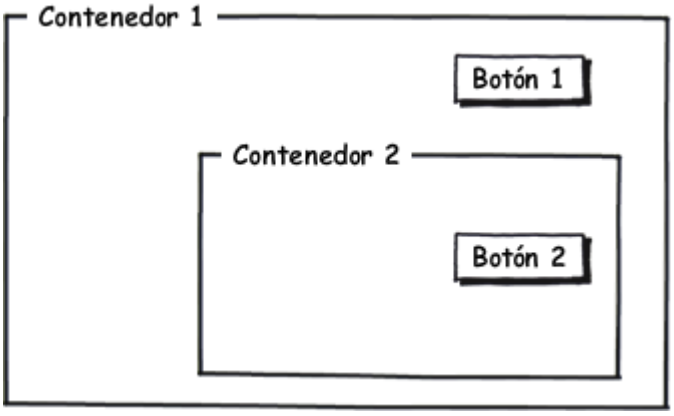

Figura 5: Jerarquía de componentes y contenedores

Como se observa en la *Figura 5*, existe un contendor (contenedor 2) que se encuentra ubicado dentro de otro contenedor (contenedor 1), ambos contenedores contienen a su vez otros componentes. Esto representa una jerarquía, ya que los componentes se ubican dentro de contenedores específicos y los contenedores, a su vez, pueden estar ubicados dentro de otros contenedores.

La falta de información jerarquizada, no representa un problema en la aplicación *Balsamiq Mockups*, dado que se trata de la representación de un dibujo que a su vez representa una pantalla, pero a la hora de generar una pantalla real, los componentes deben respetar una jerarquía o estructura dada. En el siguiente apartado relacionado con la interpretación de la información de un diseño, describiremos como se soluciona la representación de una jerarquía de componentes visuales utilizando algunos de los atributos especificados en el archivo *bmml* (archivo de salida).

Interpretación y validación

En el apartado anterior definimos la estructura de un archivo *bmml*, en este apartado definiremos como se realizará la interpretación de los mismos y su validación.

Cuando nos referimos a la interpretación, tenemos que tener en cuenta que los archivos de salida de *Balsamiq Mockups* tienen un formato *xml*. Esto nos permite utilizar cualquier parser *xml* para interpretar dicha información y transformarla en un modelo de objetos. Dado que nuestra aplicación será desarrollada en *Java*, utilizaremos el framework *XStream* para poder interpretar los archivos *bmml* generados con *Balsamiq Mockups*.

Xstream permite interpretar una estructura *xml* y convertirla en objetos conociendo previamente las clases que se corresponden con dicha estructura. A continuación presentamos el diagrama de clases que representa la estructura *xml* descripta en el apartado anterior.

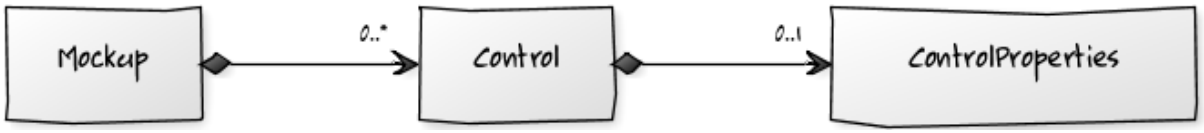

Figura 6: Diseño de clases para la interpretación del *bmml*

Como se puede observar, el diagrama de clases es muy sencillo, la clase *Mockup* se corresponderá con el tag *mockup* del *xml* generado. Una instancia de *Mockup*,

contendrá a su vez una lista de instancias de la clase *Control*. La clase *Control* se corresponde con el tag del mismo nombre y podrá contener o no una instancia de la clase *ControlProperties*, la cual definirá algunas características adicionales del control. Se debe tener en cuenta que las clases *Mockup*, *Control* y *ControlProperties*, contienen como atributos, aquellos que aparecen definidos en cada uno de los tags que representan.

Una vez parseado el *xml* a través de *XStream*, se podrán realizar diferentes validaciones analizando los atributos de cada una de las instancias. Como se mencionó en el apartado anterior, la pantalla diseñada en *Balsamiq Mockups* deberá contar con cierta estructura y deberá contemplar algunos detalles de diseño, de lo contrario, no se podrá generar a partir de la misma, la codificación de una pantalla real.

A continuación, se describen las distintas validaciones que se realizarán sobre la pantalla diseñada y previamente interpretada, para determinar si se puede generar o no la codificación de la misma.

Validación de estructura

Analizando los controles presentes en *Balsamiq Mockups*, podemos dividirlos en dos grupos, aquellos que simplemente representan un componente con alguna función en particular, y aquellos que representan algún tipo de contenedor de componentes. En el apartado anterior, hicimos mención al concepto de jerarquía de componentes, dicha jerarquía contempla la posibilidad, de que algunos componentes estén incluidos en otros que se comportan como contenedores. Los contenedores son utilizados para agrupar componentes, pero a su vez, se comportan también como componentes que pueden estar incluidos en otros contenedores.

La validación de estructura busca determinar si los componentes están incluidos o no en un contenedor. Para el desarrollo del generador, hemos establecido como requisito de diseño, que siempre exista un contenedor principal que agrupe a los demás componentes y que este sea de un determinado tipo, a continuación, se muestran diferentes diseños de pantallas a modo de ejemplo.

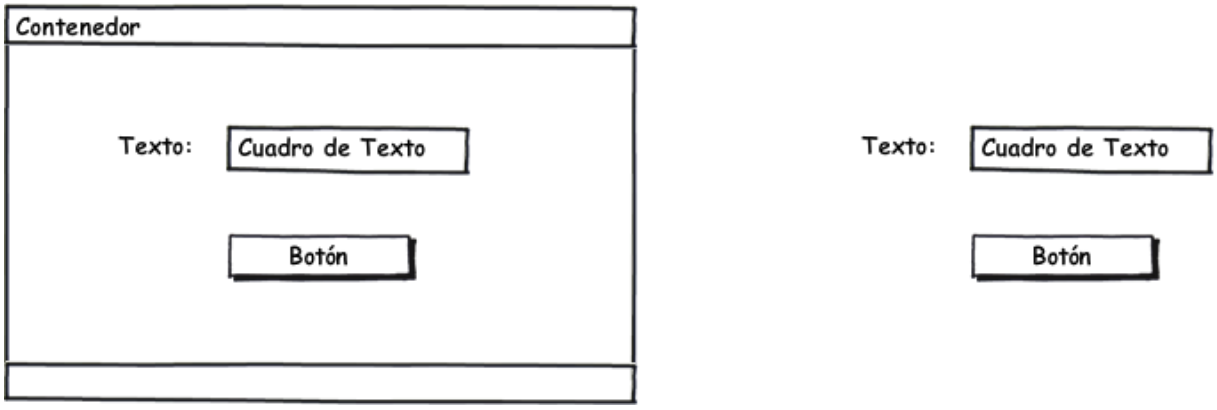

Figura 7: Ejemplos de contenedores

La figura anterior muestra dos pantallas diferentes, la pantalla que se encuentra a la izquierda presenta los componentes dentro de un contenedor y la pantalla que se encuentra a la derecha presenta, los mismos componentes, sin un contenedor. Como se mencionó anteriormente, los componentes siempre se deberán ubicar dentro de un contenedor general, de esta manera y una vez interpretadas las pantallas se determinará, que la pantalla ubicada a la izquierda será válida y la que se ubica a la derecha será inválida.

El contenedor general, esperado por la aplicación de generación de código, podrá pertenecer a cualquiera de los siguientes tipos:

controlTypeID: com.balsamiq.mockups::TitleWindow

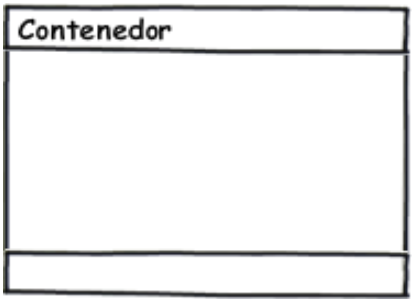

Figura 8: Contenedor de tipo *TitleWindow*

controlTypeID: com.balsamiq.mockups::Canvas

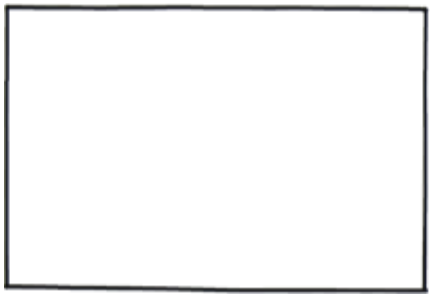

Figura 9: Contenedor de tipo *Canvas*

controlTypeID: com.balsamiq.mockups::FieldSet

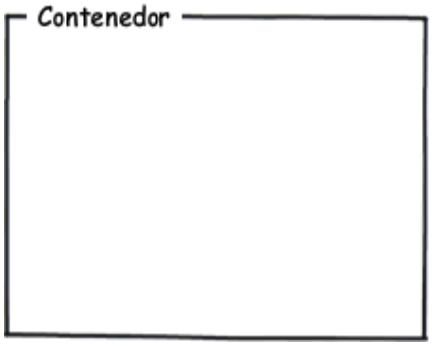

Figura 10: Contenedor de tipo *FielSet*

Como se puede observar, los tres tipos de contenedores son básicos, presentando como diferencia la posibilidad o no de poder añadir algún tipo de título o etiqueta. El título o etiqueta del contenedor, aparecerá en el archivo *bmml* especificado dentro del tag *ControlProperties* descripto anteriormente. Cabe destacar que *Balsamiq Mockups* presenta más componentes del tipo contenedor, pero los mismos, poseen diseños más complejos y no son representativos del concepto de contenedor general.

Básicamente, la validación de estructura, analizará si existe un contenedor general del tipo definido anteriormente.

Validación de jerarquía

Dentro del contenedor, denominado contenedor general, podrán ser ubicados diferentes componentes al igual que otros contenedores, este es el concepto de jerarquía que mencionamos anteriormente y el cual deberá ser analizado si existen diferentes componentes de tipo contenedor.

En la siguiente figura se puede observar un contenedor general y un contenedor secundario utilizado para ubicar y alinear los botones presentes en la pantalla. Se debe tener en cuenta que el contenedor secundario se encuentra correctamente ubicado dentro del contenedor general, es decir, se presenta una jerarquía de componentes.

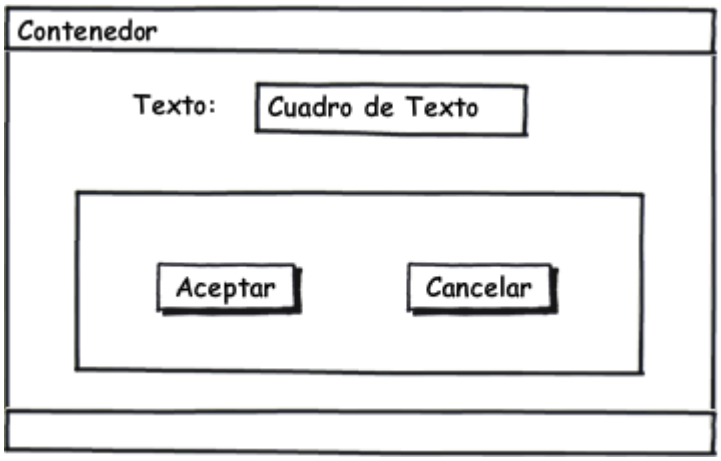

Figura 11: Jerarquía de contenedores

En este caso, se interpretará que el contenedor general es un componente que a su vez contiene otros tres componentes: una etiqueta, un cuadro de texto y un componente de tipo contenedor. El contenedor secundario, se interpretará como un componente que a su vez contiene dos botones.

Anteriormente, al analizar la estructura de un archivo *bmml*, habíamos mencionado la falta de información jerarquizada, es decir, la falta de algún tipo de indicador que nos permita saber que componentes se encuentran ubicados dentro de un contenedor. Debido a esto, la validación de jerarquía deberá tener en cuenta las coordenadas de los componentes al igual que sus tamaños para poder determinar cuales se encuentran ubicados dentro de un contendor.

En el siguiente ejemplo se puede observar una pantalla que cuenta con dos contenedores:

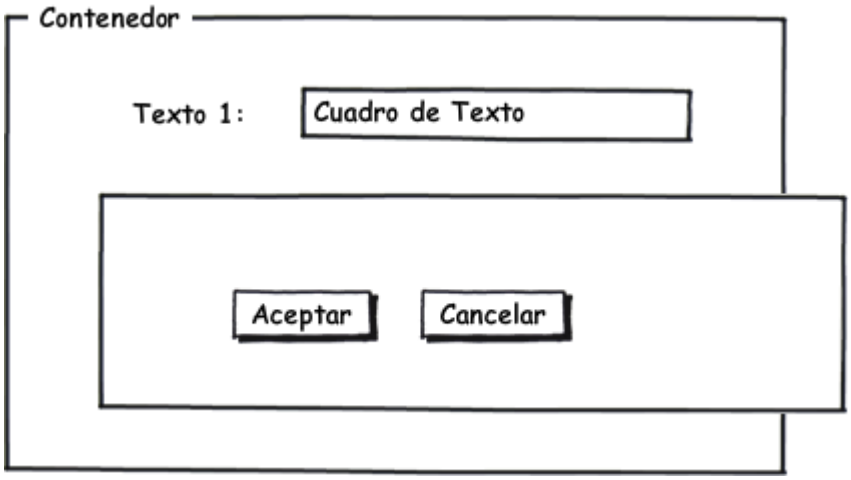

Figura 12: Jerarquía de contenedores mal alineada

En la *Figura 12* se puede observar una jerarquía de contenedores, el contenedor de tipo *Canvas*, que contiene a los botones, no se encuentra correctamente ubicado o alineado dentro del contenedor general. Para realizar la validación de jerarquía, la aplicación determinara si existe más de un contenedor en la pantalla diseñada, de ser así, se compararán uno a uno los contenedores en relación a sus tamaños y coordenadas para poder determinar si alguno de ellos no se encuentra correctamente ubicado en el diseño. Como se analizó anteriormente, los archivos *bmml* contienen, por cada componente, la información relacionada a sus dimensiones y coordenadas. De esta forma se podrá determina que diseños presentan una jerarquía de componentes válida. A continuación se presentan dos ejemplos de jerarquías correctamente alineadas.

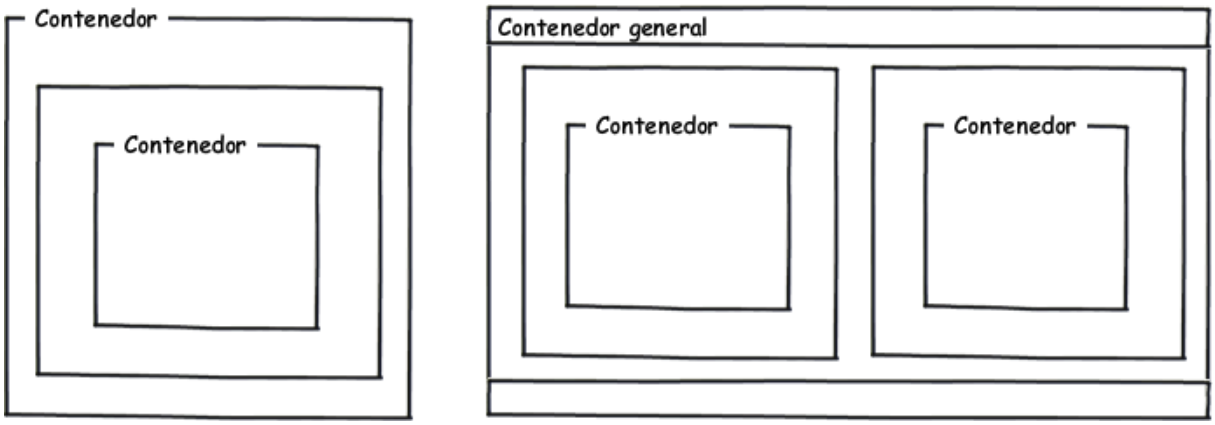

Figura 13: Ejemplos de jerarquías de contenedores

En la pantalla ubicada a la izquierda se pueden observar tres contenedores distribuidos uno adentro del otro y en la pantalla de la derecha se pueden observar contenedores que están ubicados en un mismo nivel de la jerarquía. De esta manera las jerarquías de contenedores se podrán diseñar de diferentes maneras y la validación deberá poder determinar cuándo dichas jerarquías son incorrectas.

Analizando la jerarquía de contenedores podemos describir la misma como una estructura de árbol, en la cual siempre existirá una raíz que simbolizará el contendor general y de la cual se desprenderán más ramas representando a los contendores presentes en el diseño.

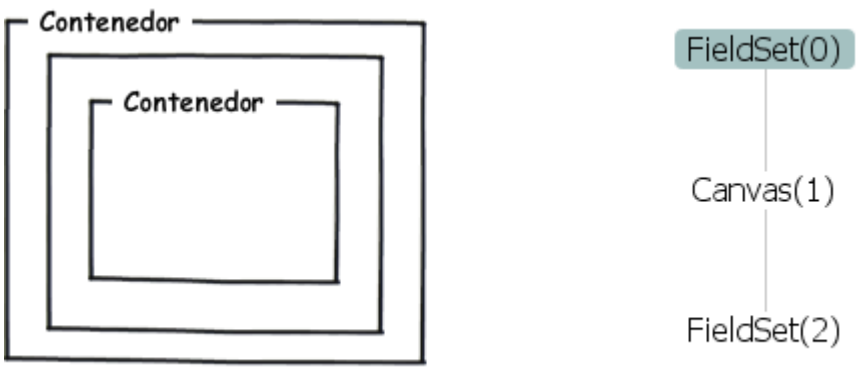

Figura 14: Ejemplo de jerarquía de contenedores y su representación en forma de árbol

La figura anterior, muestra la representación en árbol de la jerarquía de contenedores de la pantalla ubicada a la izquierda. La imagen que representa el árbol fue tomada de la aplicación de generación y se describirá más tarde. Como se puede observar en el árbol, existe una raíz que representa el contenedor general y el cual es del tipo *FieldSet*, dentro del mismo está contenido un control del tipo *Canvas* y dentro de este último otro del tipo *FieldSet*. La aplicación de generación de código, al realizar la validación de jerarquía, irá construyendo una estructura de árbol con los contenedores ubicados en la pantalla. El árbol, es una estructura que permite ver con facilidad la jerarquía presente en las pantallas diseñadas y por esa razón se incorporará, a la aplicación de generación de código, un componente que permita visualizar el árbol de cualquier pantalla interpretada.

Una vez construida la estructura de contenedores y pasado exitosamente la validación de jerarquía, se realizara una nueva validación relacionada con los componentes (no contenedores) presentes en el diseño.

Validación de componentes

En este tipo de validación se determinará si todos los componentes se encuentran correctamente ubicados dentro de un contenedor. Hasta este punto del proceso de validaciones, solo se tiene armada una estructura de contenedores en forma de jerarquía, pero no se ha validado la ubicación del resto de los componentes. Básicamente se analizará, en función a las coordenadas de cada componente, dentro de que contenedor se debe ubicar el mismo. Debido a que ya se encuentra construida una estructura jerárquica de contenedores, el proceso de validación se ubicará en el contenedor interno para determinar si el componente está ubicado en el mismo, de no ser así, se moverá al contenedor del nivel superior y así sucesivamente hasta llegar al contenedor principal, si se determina que el componente no está ubicado en ninguno de los contenedores, se descartará al mismo de la generación, es decir, no se visualizará en la pantalla final.

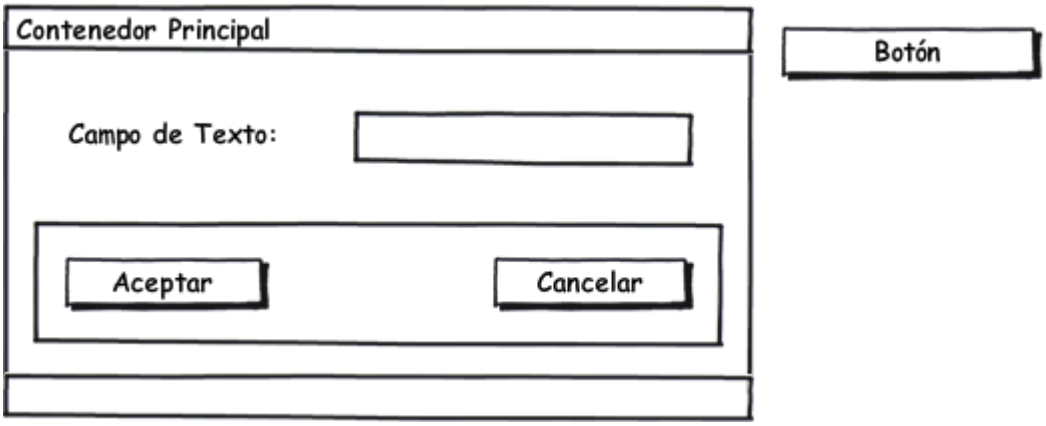

Figura 15: Ejemplo de jerarquía de contenedores y ubicación de componentes

En la figura anterior se puede observar un componente de tipo *Button* fuera de todo tipo de contenedor, al pasar por la validación de componentes, el mismo se descartará y solo se contemplará en la generación el conjunto de componentes ubicados en el contenedor principal. Una vez que se determina que el componente está dentro del contenedor, se deberá determina si el mismo no presenta un solapamiento con otro componente existente dentro del mismo contenedor. La validación tendrá en cuenta el tamaño del componente (propiedades de ancho y largo) al igual que las coordenadas de ubicación del mismo y determinará si se encuentra ubicado por sobre otro componente previamente establecido. En este caso particular, el componente no será descartado y se informará que la estructura de la pantalla diseñada presenta un problema de solapamiento.

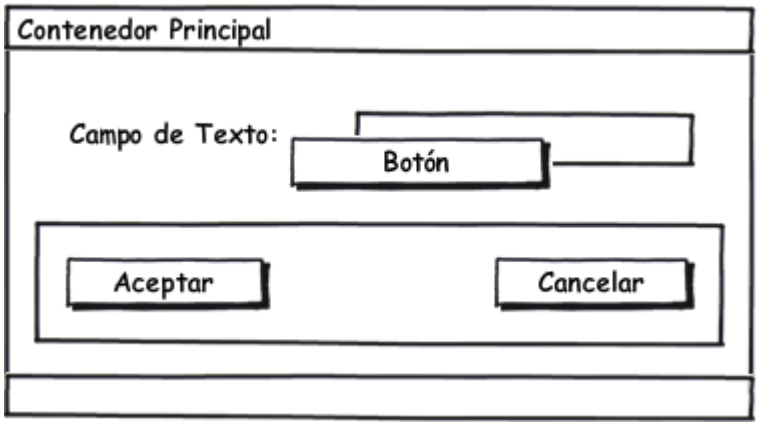

Figura 16: Solapamiento de componentes

Teniendo en cuenta que la aplicación *Balsamiq Mockups* permite arrastrar los componentes a la pantalla, se podrá generar en muchos casos un solapamiento, es decir, componentes que se encuentran ubicados por encima de otros. Básicamente el problema del solapamiento de componentes está relacionado con la generación final de la pantalla, en algunos lenguajes, se podrán presentar componentes arriba de otros, pero otros lenguajes no darán esa opción y por este motivo se determinara, que una pantalla con componentes solapados, es incorrecta.

Cuando un diseño cumpla con todas las validaciones descriptas anteriormente, se determinará que el mismo es válido y se podrá continuar con el proceso de generación. A modo de resumen vamos a repasar nuevamente cada una de las validaciones. La siguiente figura expresa el proceso de validaciones en su totalidad.

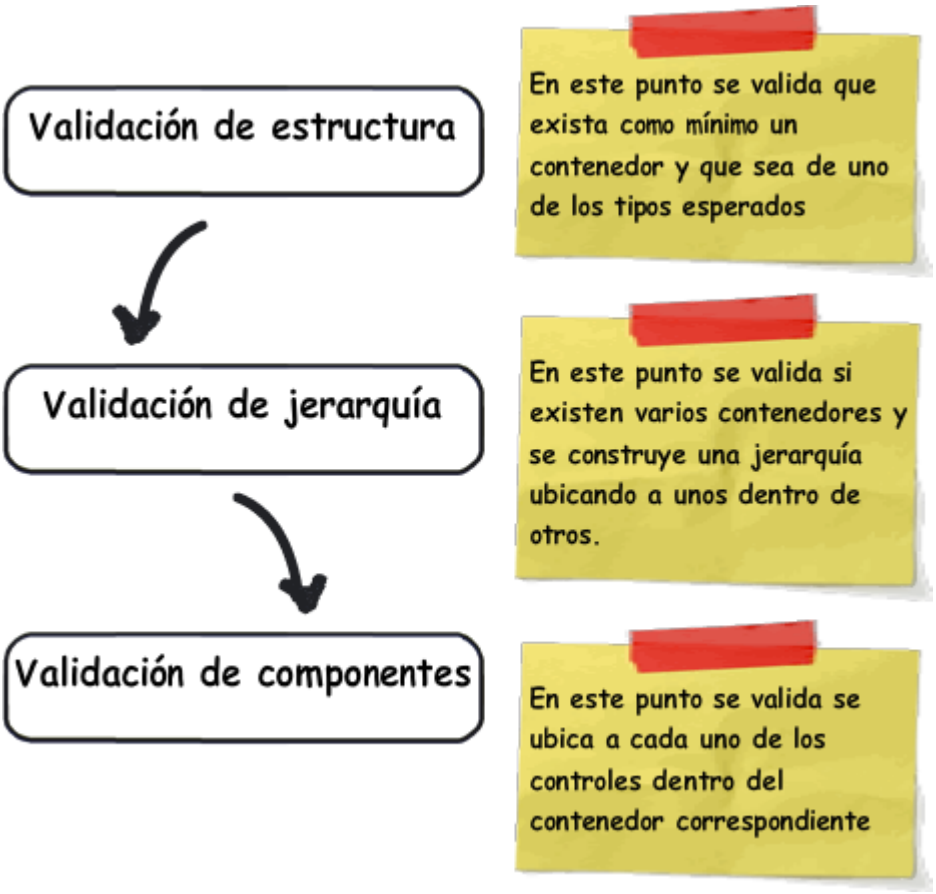

Figura 17: Proceso de validaciones

Una vez finalizada la validación, obtendremos como salida un estructura de contenedores y componentes, instanciada en memoria, y que aplica el patrón de diseño *Composite*. Esto nos permitirá poder recorre la estructura de instancias y visualizarla, entre otras cosas, en una pantalla. El siguiente paso estará vinculado con la representación de cada uno de estos contenedores y componentes en un determinado lenguaje de programación, este será el siguiente paso antes de poder finalizar la generación de código. En el siguiente apartado, detallaremos el concepto de *template* y la forma de especificar cómo se representarán los componentes.

Templates

La interpretación de las pantallas creadas con *Balsamiq Mockups*, tiene como finalidad, la generación de las mismas en un determinado código fuente. En el apartado anterior, describimos como se realizará el proceso de interpretación, que obtendrá como salida, una estructura de componentes con la parte de la información necesaria para la generación final. Está estructura describirá los tipos de componentes presentes, sus características (tamaño, ubicación, etc.), y la jerarquía de los mismos diferenciando controladores y componentes básicos, pero no especificará la forma de representa finalmente estos componentes. La representación de los componentes dependerá del lenguaje de programación seleccionado para escribir el código fuente de la pantalla diseñada, es por este motivo, que es necesaria una estrategia que nos permita seleccionar diferentes lenguajes o formas de representación. En este proyecto, hemos establecido como alcance, la representación de los componentes en dos tipos de lenguajes diferentes: *HTML* y *Java*, pero debe tenerse en cuenta, que la estrategia de representación que utilizaremos nos permitirá, en un futuro, utilizar otros lenguajes. Dado que existirán distintas formas de representar los componentes, deberemos establecer una manera de poder escribir estas representaciones. Si tenemos en cuenta la manera en que *Balsamiq Mockups* guarda la información de las pantallas diseñadas, podremos observar que las mismas, se terminan guardando en un lenguaje definido a partir de *xml*. Los archivos *bmml* descriptos anteriormente, están basados en un lenguaje propio de *Balsamiq Mockups* y definido con *xml*, un metalenguaje utilizado ampliamente en el diseño de software. A partir de *xml*, podemos definir nuestros propios lenguajes basados en

tags, por esta razón, utilizaremos *xml* para definir un lenguaje propio y sencillo que nos permita especificar distintas formas de representación. A las diferentes formas de representación las denominaremos *templates* o *plantillas* y deberán contener toda la información necesaria para poder concretar el proceso de generación del código fuente final.

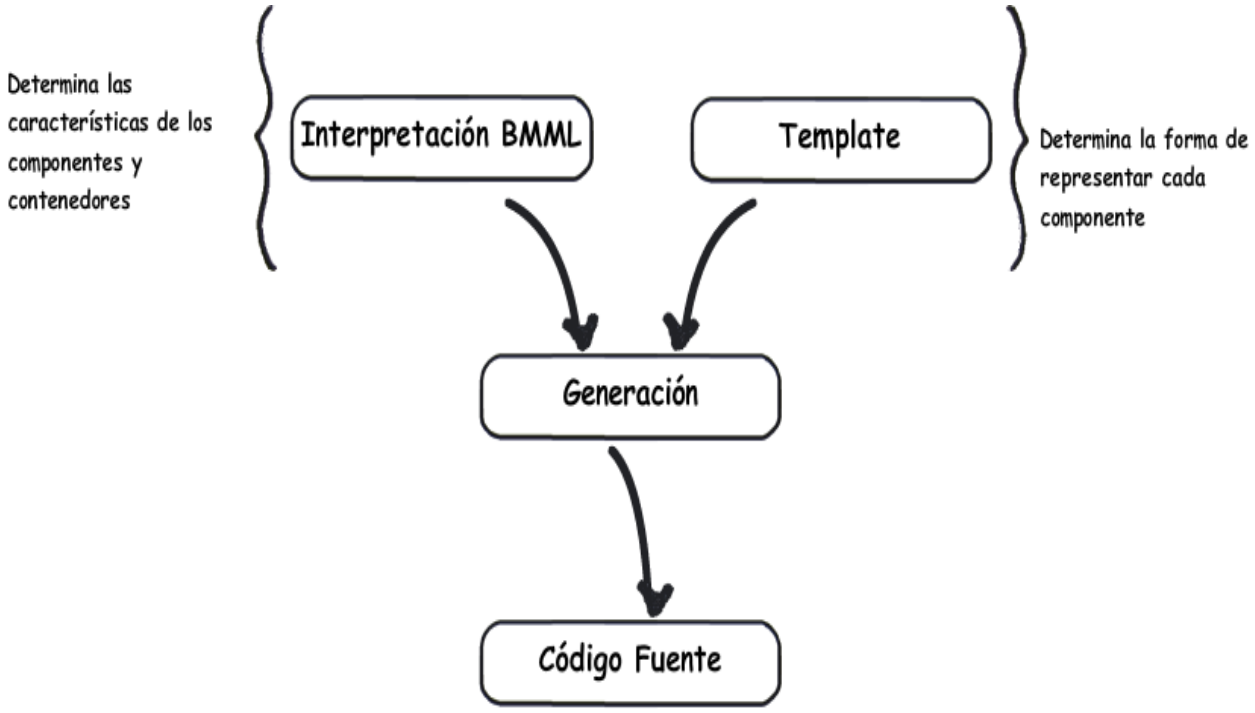

Figura 18: Proceso de generación

Como se puede observar en la *Figura 18*, la interpretación del archivo *bmml* y la selección de un *template* serán la fuente de información utilizada por el proceso de generación. Como se mencionó anteriormente, el lenguaje utilizado para escribir los templates, deberá ser lo más genérico posible para poder especificar la representación de los componentes en diferentes lenguajes de programación. Los dos lenguajes seleccionados, como alcance del proyecto, pertenecen a dos paradigmas de programación diferentes y nos servirán para poder definir un lenguaje de templates, lo suficientemente genérico, que permita contemplar las diferentes características de ambos lenguajes de programación.

Vamos a repasar algunas de las principales características de los lenguajes *HTML* y *Java* a modo de entender sus diferencias.

El lenguaje *HTML* (lenguaje de marcado de hipertexto), es el lenguaje predominante para la elaboración de páginas web y se utiliza para describir la estructura y el contenido en forma de texto, así como para complementar el texto con objetos tales como imágenes. El leguaje *HTML* se escribe en forma de etiquetas denominadas *tags*, rodeadas por corchetes angulares (<,>). *HTML* es un lenguaje de tipo estático y muchas veces es utilizado en conjunto con lenguajes orientados a scripts como *JavaScripts*, este tipo de lenguajes añaden dinamismo a *HTML*.

```
<html lang="es">
   <head>
     <title>Ejemplo</title>
   </head>
   <body>
     <p>ejemplo</p>
   </body>
</html>
```

Figura 19: Ejemplo básico de *HTML*

En la *Figura 19* se puede observar una estructura *HTML* muy básica, los documentos se organizan de manera jerárquica en dos secciones principales: <*head*> y <*body*>. La sección <*head*> se utiliza generalmente para la importación de librerías de script y estilos utilizadas para dar formato a la página y añadir algún tipo de funcionalidad dinámica. La sección <*body*>, como su nombre lo indica, es el cuerpo principal del documento o página web, y dentro del mismo, se declaran todos los componentes visuales que aparecerán en la página. Teniendo en cuenta que nuestro interés es la generación del código fuente de una pantalla vamos a analizar cómo se declaran diferentes componentes en *HTML*, para poder empezar a identificar parte de la lógica final de representación.

```
<html>
    <head>
        <title>Ejemplo</title>
    </head>
    <body>
        <input type='button' value='Botón'/>
    </body>
</html>
```

Figura 20: Representación de un botón en *HTML*

En la *Figura 20* podemos observar la forma de escribir o representar un botón en *HTML*. El tag *input* permite representar otros componentes a partir del atributo *type*, dicho atributo, define el tipo de componente. El atributo *value*, define la etiqueta que se visualizará en el botón, de esta manera el botón se representará visualmente de la siguiente forma:

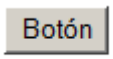

Figura 21: Visualización de un botón en *HTML*

En *HTML* los componentes se podrán definir a partir de tags y los mismos contarán con diferentes atributos para determinar su visualización o comportamiento. De esta manera, queda claro que en términos de *HTML*, se deberán generar tags para la representación de los componentes, y los atributos de los tags, quedarán definidos por las características de los componentes presentes en los diseños realizados con *Balsamiq Mockups*. Anteriormente habíamos hablado de la propiedades que poseen los componentes presentes en un archivo *bmml*, y como se obtendrán las misma en la interpretación de dicho archivo, estas propiedades, deberán ser traducidas a través del template, para convertirse finalmente en los valores de los atributos de un tag. Como conclusión de este análisis de *HTML*, podemos determinar que un template, deberá tener la capacidad de poder definir tags *HTML* y sus respectivos atributos a partir de las propiedades presentes en los componentes especificados en el *bmml*.

Java es un lenguaje de alto nivel, y a diferencia de *HTML*, los componentes se crean a partir de la instanciación de clases en lugar de tags. *Java* es un lenguaje orientado a objetos y los componentes visuales se definen a partir de clases que incorporan todas las características de los mismos. Teniendo en cuenta que hemos presentado la forma de representar un botón en *HTML*, vamos a analizar la representación de un botón en *Java*. La representación de componentes visuales y la creación de aplicaciones de escritorio, se pueden realizar a través de dos frameworks: *Swing* y *SWT*. En este caso vamos a utilizar *Swing*, un framework que permite representar una amplia gama de componentes visuales en *Java*. Los componentes visuales deben estar incluidos dentro de cuadros o frames, estos cuadros representan a los contenedores y los describiremos más adelante, en este caso vamos a ver como se debería instanciar un componente de tipo botón en *Java Swing*.

```
    JButton generatorButton = new JButton(“Botón”);
    generatorButton.setToolTipText("Esto es un botón");
```

Figura 22: Visualización de un botón en *Java*

Como se puede observar en la *Figura 22*, un botón se puede crear a partir de la clase *JButton*. La creación de una instancia de *JButton* no permitirá definir un nuevo botón e invocar métodos del mismo para terminar de definir su comportamiento, como es el caso del método *setTooTipText*, que permite definir el *tooltip* del botón.

Teniendo en cuenta los dos ejemplos anteriores, podemos empezar a entender las diferencias que existen, a la hora de representar componentes, en estos dos lenguajes. En términos de *HTML*, deberemos definir tags y en el caso de *Java*, deberemos escribir clases que instancien a su vez los componentes visuales. Vamos a analizar la confección de templates unificados que permitan representar componentes en ambos lenguajes, teniendo en cuenta, que el lenguaje de templates deberá ser el mismo para ambos casos.

Como mencionamos anteriormente, el lenguaje de templates se definirá a partir de *xml*, por esta razón, empezaremos a describir los diferentes tags que aparecerán en un template y que formarán parte del lenguaje del mismo. En el metalenguaje *xml* se determina que debe existir siempre un tag denominado *root*. El tag *root* es básicamente el tag principal y contendrá el resto de los tags que describirán el template, es por esta razón, que siempre se deberá respetar una jerarquía de tags. En nuestro lenguaje de templates, llamaremos al tag *root* <*template*>, dado que cada uno de los documento *xml*, que respeten este lenguaje, definirán un único template. El tag <*template*> contendrá a su vez un atributo denominado *code*, este atributo será explicado más adelante con mayor grado de detalle, pero su principal función, será determina el tipo de paradigma que aplica el template, es decir, si la representación que define el template se base en un lenguaje de tags (*HTML*) o un lenguaje orientado a objetos (*Java*).

```
<template code="Java">
      <components>
      …
      </components>
      <containers>
      …
      </containers>
      <main>
      …
      </main>
</template>
```

Figura 23: Estructura básica de un *template*

Como se puede observar en la *Figura 23*, el tag principal contendrá otras tres etiquetas que definirán el cuerpo del documento o template. Los tags *components*, *containers* y *main* serán de carácter obligatorio, ya que cada uno de ellos, se utilizará para definir un aspecto específico de la representación final de una pantalla. A continuación, explicaremos el propósito de cada uno de ellos, especificando también, los tags que podrán aparecer dentro de los mismos.

Definición del tag components

El tag *components* será utilizado para la definición de la representación de los componentes. Dentro del mismo, se especificará cada uno de los componentes que podrá ser representado por el template junto con sus características visuales.

```
<components>
     <component type="Button" name="input">
     <attributes>
          <attribute name="type" value="button" />
          <attribute name="id" value="$componentId" />
          <attribute name="name" value="$componentId" />
          <attribute name="value" value="$text" />
          <attribute name="class" value="button" />
          <attribute name="style" value="width: $width px;" />
     </attributes>
     </component>
</components>
```

Figura 24: Definición de un componente *HTML*

Como se puede observar en la *Figura 24*, dentro del tag *components* se define la representación de un componente, en este caso, un botón *HTML*. El tag *component*, se utilizará tantas veces como componentes se quieran definir y dentro del mismo se especificarán las características o propiedades del mismo. Si observamos el tag *component*, presente en la figura, podremos determinar la presencia de dos atributos, el atributo *type* y el atributo *name*. El atributo *type* define el tipo de componente a representar y su valor deberá coincidir con el nombre del componente especificado en el archivo *bmml*, si recordamos los atributos presentes en el archivo de salida, los componentes se identificaban a través del atributo *controlTypeID*, en el caso de un botón, el valor de dicho atributo sería *controlTypeID="com.balsamiq.mockups::Button"*, la parte final del valor del atributo define el componente, y por esta razón se utilizará para identificar su representación dentro del template. Cuando el proceso de generación busque la representación de un componente cuyo *controlTypeID* sea igual a *com.balsamiq.mockups::Button*, se recorrerán todos los componentes, definidos en el *template*, para localizar aquel que tenga como valor, del atributo

type, el identificador *Button*. El otro atributo que aparece definido en el tag *component* se denomina *name*, este atributo define el nombre del tag o clase que representará el componente en el lenguaje de programación seleccionado. Como se puede observar en la *Figura 24*, el valor del componente *name* es *input*, se debe recordar que el tag utilizado en *HTML* para representar un botón es justamente *input* como se puede observar en la *Figura 20*. Teniendo en cuenta los valores de los atributos *type* y *name* se podrá determinar con que tag *HTML* se representará el componente presente en el diseño de pantalla, como ya hemos especificado, los tags *HTML* poseen atributos que permiten definir el comportamiento del tag, en este caso, tratándose del tag *input* vamos a presentar los atributos necesarios para poder representar el botón.

```
<input
type='button'
id='ejemplo'
name='ejemplo'
value='Botón'/>
```

Figura 25: Representación de un botón con el tag *input*

Cuando se utiliza el tag input para la representación de componentes, es necesario especificar algunos atributos, los atributos *id* y *name* identifican el tag y son utilizados generalmente para hacer referencia al mismo desde lenguajes como *javascript*. El atributo *type* del tag *input* define que componente representa el tag, dado que el mismo tag se puede utilizar para la representación de otros componentes, y por último, el atributo value es la etiqueta del botón, es decir aquella que se mostrará en pantalla. Si tenemos en cuenta los atributos descriptos anteriormente y visualizamos la *Figura 24*, podremos determinar que dentro del tag *component* se pueden definir una serie de atributos, estos son los atributos que el generador de código agregará al tag *input* un vez que se inicie el proceso de generación. Todo atributo que se defina en el *template*, para un componente específico y dentro de los tags *atributes*, se escribirá en el cuerpo del tag definido a través del atributo *name* del elemento *component*.

Como podemos observar también en la *Figura 24*, los atributos poseen un nombre y un valor, esto es lo que permite escribir correctamente un atributo dentro del tag, el siguiente código es un ejemplo:

```
<component type="Button" name="input">
     <attributes>
          <attribute name="type" value="button" />
          <attribute name="id" value="id_boton" />
          <attribute name="value" value="Botón" />
     </attributes>
</component>


<input type='button' id='id_boton' value='Botón' >
```

Figura 26: Representación de un botón a partir de un *template HTML*

En la *Figura 26* observamos la definición de un componente de tipo *Button*, presente en el template, y en la parte inferior, como se escribirá el código fuente de dicha definición. Está claro que los valores de los atributos se definen a partir de *value* y sus nombres a partir de *name*, en el ejemplo anterior los valores son especificados directamente en el *template*, como es el caso de value="*id_boton*", pero generalmente estos valores dependerán del contenido del archivo *bmml* o serán valores que se determinarán en tiempo de generación. Si volvemos a observar la *Figura 24*, podremos encontrar, en algunos atributos, palabras reservadas que hacen referencia a valores que no se pueden determinar directamente en el template, estos valores, como mencionamos anteriormente, serán determinados en el proceso de generación. Más adelante, se describirán todas las palabras reservadas que podrán aparecer en la definición de *template*, pero por el momento, vamos a explicar las que aparecen en la *Figura 24*.

$componentId: Es un identificador de componente que se creará en el proceso de generación de código. Dado que no todos los componentes pueden tener el mismo *id* o *name*, el proceso de generación será el encargado de generar un identificador para cada componente.

$text: Como mencionamos en el apartado de interpretación del archivo *bmml*, algunos componentes podrán contener propiedades como una etiqueta o texto, en el proceso de generación se determinará si el componente posee un texto y se expondrá como la variable *$text* para que se pueda utilizar desde el template.

$width: Determina el ancho del componente y puede ser utilizado para que el componente generado tenga el mismo tamaño que el diseñado con *Balsamiq Mockups*. Cabe destacar que la palabra reservada **$height** permitirá definir la altura del componente.

Los valores de las palabras reservadas, que se podrán utilizar en los templates, se determinarán una vez interpretado el archivo *bmml*, de esta manera, las características de los componentes presentes en el diseño se podrán replicar en el código fuente finalmente generado.

Teniendo en cuenta que hemos dado un ejemplo de cómo se debería escribir un template para la generación de un botón *HTML*, vamos a realizar lo mismo para la generación de un botón en *Java*. Como habíamos mencionado anteriormente, el lenguaje de templates será unificado, es decir, servirá para poder especificar una representación en *HTML* o en un lenguaje de programación como *Java*.

```
<component type="Button" name="JButton">
     <attributes>
          <attribute name="value" value="$text" />
     </attributes>
     <methods>
          <method name="setSize">
               <attributes>
                    <attribute name="value1" value="$width" />
                    <attribute name="value2" value="$height" />
               </attributes>
          </method>
     </methods>
</component>
```

Figura 27: Representación de un botón a partir de un *template JAVA*

Como podemos observar en la figura anterior, nuevamente utilizamos los mismos tags para definir la representación de un componente, pero esta vez y a diferencia de la anterior, la representación final será en el lenguaje *Java*. El tag *component* a través de su atributo *type*, define el tipo de componente a representar, y al igual que el *template HTML*, este valor debe coincidir con el id del componente en el archivo *bmml* que se desea representar. El atributo *name*, en este caso, define el nombre de la clase a partir de la cual se representará el botón, recordemos que en el caso anterior, el atributo *name*, referenciaba el nombre de un tag *HTML*. Dentro de la definición de *component* podemos observar la definición de los atributos, en el caso anterior, los atributos se escribían dentro del tag definido en *name*, en este caso, y teniendo en cuenta que estamos hablando de clases, los atributos representarán los parámetros que recibirá el constructor de la clase especificada en *name*. De esta manera, se podrán instanciar clases con todo tipo de parámetros. Como podemos observar en la *Figura 27*, la instanciación de la clase *JButton*, recibirá como único parámetro, el texto definido con la palabra reservada *$text*, es decir, el texto definido en el archivo *bmml* para el componente de tipo botón. Debe tenerse en cuenta, que el orden de los atributos dentro del tag *component* será el mismo utilizado para la instanciación de la clase, es decir, el primer atributo será el primer parámetro, el segundo atributo el segundo parámetro y así sucesivamente.

```
JButton jButton1 = new JButton("Botón");
jButton1.setSize(91, 28);
```

Figura 28: Código fuente de un botón generado a partir de un *template Java*.

En la figura anterior podemos observar el código fuente generado a partir del tag *component* definido en la *Figura27*. A partir de la definición del *template* y del atributo *name* del tag *component*, se determinará que se debe instanciar una clase de tipo *JButton* para la representación del botón. El parámetro que recibe el constructor de la clase *JButton*, se determina a partir del atributo definido dentro del tag *component*, como se observa en la *Figura 27*, solo se definió un único atributo, por lo tanto el constructor de la clase, recibirá un solo parámetro.

La línea de código fuente que se visualiza en la *Figura 28* y debajo de la instanciación de la clase *JButton*, sirve para definir el tamaño del botón creado. La definición del tamaño del botón, así como también, la modificación de sus características visuales o comportamiento, solo se puede realizar a través de la invocación de métodos sobre la instancia creada, como vemos en la figura anterior, se invoca el método *setSize*. Si observamos nuevamente la definición del componente botón, *Figura 27*, podremos notar que dentro del tag *component*, y por debajo del tag *attributes*, se define un nuevo tag denominado *methods*, dentro de este tag, se definirán todos los métodos que se deben invocar sobre la instancia de clase creada. Más adelante definiremos el significado de este mismo tag en un template de tipo *HTML*, dado que Java y HTML son lenguajes distintos. En el caso de un template *Java*, y retomando el ejemplo de la *Figura 27*, en la sección de métodos se definirá un único método a invocar, el nombre del método estará definido a partir del atributo *name* y los parámetro que recibirá el mismo se definirán nuevamente con un tag denominado *attribute*. Nuevamente cabe destacar, que el orden de los atributos, presentes en la definición del tag *method*, será el orden utilizado para la invocación del método, como se observa en el código generado, el primer parámetro es el ancho del botón, y el segundo es el largo. Las variables o palabras reservadas descriptas anteriormente, se podrán utilizar en cualquier punto de la definición del template, como se observa en la figura *Figura 27*, $*width* y $*height* se utilizan para definir el ancho y largo del botón y serán los valores finales que recibirá el método *setSize* de la clase *JButton*.

De esta manera, hemos demostrado hasta ahora, como el lenguaje unificado que hemos creado para la definición de templates, puede ser utilizado para la generación de código en dos lenguajes diferentes *HTML* y *Java*, pero lo descripto hasta el momento, no define la representación de componentes de tipo contenedor ni determina como quedará finalmente la estructura del código fuente generado, para estos dos puntos vamos a describir las secciones *containers* y *main* mencionadas anteriormente y que forman, de manera obligatoria, parte de la definición final de un *template*.

Definición del tag containers

El tag *containers* sirve para definir la representación de aquellos componentes que se consideran contenedores. A diferencia de la representación de los componentes, los contenedores deben especificar también la distribución de componentes dentro de los mismos, es decir, la forma en que los componentes se ubicarán en el contenedor representado. Como especificamos anteriormente, *Balsamiq Mockups* presenta tres tipos básico de contenedores los cuales serán perfectamente soportados por la aplicación de generación, estos contenedores son *TitleWindow*, *Fieldset* y *Canvas*. A continuación describiremos como se deberá especificar la representación de un contenedor dentro de un template, y al igual que con los componentes, lo haremos en primer lugar para HTML y posteriormente para el lenguaje Java.

```
<containers>
      <container type="TitleWindow" name="div" >
      <attributes>
            <attribute name="class" value="container" />
            <attribute name="style" value="position:relative; top: $y px;
            left: $x px; width: $width px; height: $height px;" />
      </attributes>
      <componentLocation name="div">
            <attributes>
                  <attribute name="style" value="position: absolute; top:
                  $y px; left: $x px;" />
            </attributes>
      </componentLocation>
</container>
```

Figura 29: Definición de contenedores en un *template HTML*

Dentro del tag *containers* se definirán todos los componentes de tipo contenedor que podrá representar el template, como se puede observar en la figura anterior, para esto se utilizará el tag *container*. Nuevamente, la relación entre el componente presente en el archivo *bmml* y su representación en el *template*, dependerá del atributo *type*. El valor del atributo *type* deberá coincidir con el valor del nombre del componente que aparece, en el archivo *bmml*, bajo el atributo *controlTypeID*, en el caso del ejemplo anterior, la representación corresponde a un *TitleWindow*. Se debe tener en cuenta, que en el archivo *bmml* generado por *Balsamiq Mockups*, se define el tipo de componente bajo lo que se denomina un nombre cualificado, esto significa que se incluye el nombre del paquete de la

clase que implementa el componente. El proceso de generación, una vez interpretado el archivo *bmml*, descartará los nombres de paquetes presentes en el atributo *controlTypeID* de cada componente, de esta manera y al definir un contenedor, solo se deberá hacer referencia al nombre de la clase que implementa el componente. Como observamos en la *Figura 29*, el atributo *type* del contenedor hacer referencia a *TitleWindow*, este componente aparecerá en el archivo *bmml* de la siguiente manera:

controlTypeID="com.balsamiq.mockups::TitleWindow".

Dentro del tag *container* también se puede observar el atributo *name*, al igual que en la definición de componentes, el atributo *name* determina el tag *HTML* o el nombre de la clase *Java* que representará el componente. En este caso, el tag *HTML div* se utilizará para representar el contenedor, los tags de tipo *div* son utilizados en *HTML* para la definición de capas y son la mejor opción, para un *template HTML*, a la hora de representar contenedores. Dentro de la definición del contenedor, se puede observar la definición de atributos, estos atributos son similares a los definidos en el tag *component*, en la representación de un tag *HTML*, se incorporarán al cuerpo del mismo definiendo las características del tag. Como se puede ver en la *Figura 29*, el contenedor tiene dos atributos, el primero llamado *class*, es el estilo del tag que nos permitirá dibujar el contenedor con un recuadro similar al presentado en el diseño de Balsamiq Mockups, más adelante explicaremos como incorporar estilos *CSS* en un *template*. El segundo atributo presente en el contenedor, también define un estilo, pero relacionado con la posición del *div*, esto nos permitirá ubicar el contenedor en la misma posición que se ubica el contenedor presente en el diseño de pantalla. El valor que recibe el segundo atributo, de nombre *style*, define el posicionamiento del tag *div* y su ancho y largo, como podemos observar, se utilizan las variables *$y* y *$x* para el posicionamiento, y las variables *$width* y *$height* para la definición del tamaño. Recordemos que esta variables contendrán los valores provenientes de la interpretación del archivo *bmml*, por lo tanto, sus valores nos permitirán construir un contenedor con las mismas dimensiones que el diseño original de la pantalla. Para poder entender como queda finalmente el tag definido en el template anterior vamos a observar la siguiente figura.

```
<div class=’container’ style='position:relative; top: 93 px;
left: 89 px; width: 454 px; height: 304 px;' >
…
</div>
```

Figura 30: Representación de un contenedor en *HTML*

La *Figura 30*, muestra el código fuente final de la representación de un contenedor a partir del *template* de la *Figura 29*. Como podemos observar, los valores relacionados al posicionamiento y al tamaño han sido reemplazados por valores concretos. En este caso hemos definido el atributo *style* debido a que deseamos recrear un contenedor con las misma características que el diseñado, pero cabe destacar, que podríamos representar el contenedor de otra manera, sin respetar posiciones y tamaños, básicamente, la representación del contenedor dependerá de los atributos definidos para el mismo. El estilo definido en el atributo *class*, le dará al contenedor un aspecto particular, definiendo el color del recuadro y sus bordes, más adelante cuando describamos la sección *main* del *template* describiremos como se deberán definir estos estilos en un template *HTML*. También debemos notar, que existe un tag *div* de apertura y otro de cierre, dado que el *div* representa un contenedor, dentro del mismo se ubicarán los componentes que pertenecen a dicho contenedor. Si continuamos el análisis de la representación de un contenedor, podremos observar un nuevo tag denominado *componentLocation*, como hemos mencionado, dentro del contenedor se deberán ubicar diferentes componentes, la forma de ubicar dichos componentes será determinada a través del tag *componentLocation*. Si observamos el tag *componentLocation* definido en el ejemplo de *template*, podremos observa el atributo *name*, nuevamente este atributo define un tag, pero en este caso particular, será el tag utilizado para ubicar un componente dentro del contenedor. Los atributos definidos dentro del tag *componentLocation* serán utilizados para definir el comportamiento del tag, como se observa en la *Figura 29*, se define un único atributo con características de posicionamiento, las cuales nos permitirán ubicar a los componentes dentro del contendor, respetando las ubicaciones y distancias presentes en el diseño.

La siguiente figura, muestra finalmente como quedará representado un contenedor, en *HTML*, y con un componente ubicado dentro del mismo.

```
<div class='container' style='position:relative; top:93px; left:89
px; width: 454 px; height: 304 px;' >
      <div style='position:absolute; top:233px; left:134px;' >
            <input type='button' id='boton1' value='Botón'
            class='button' />
      </div>
<div
```

Figura 31: Representación de un contenedor *HTML* con un componente.

Como se puede observar, el contendor se representa por medio del tag *div* ubicado en el tope de la jerarquía, y dentro del mismo, existe otro tag *div* que determina la posición del componente dentro del contenedor. La posición relativa del *div* principal y la posición absoluta del *div* interior, son las que permiten respetar, con exactitud, las posiciones que presenta el diseño de *Balsamiq Mockups*, claro está, que si no quisiéramos respetar el mismo posicionamiento, podríamos omitir la definición del tag *componentLocation* o simplemente declararlo de otra manera. En resumen, el tag *componentLocation* tendrá la responsabilidad de determinar cómo se ubica un componente dentro de un contenedor.

Los anteriores ejemplos muestran como se debe definir la representación de un contenedor en un *template HTML*. A continuación, vamos a realizar el mismo análisis para la representación de un contenedor en un *template Java*. Debemos tener en cuenta, que en el lenguaje *Java*, la representación de componentes generalmente es implementada a través del *framework Swing*, que viene incorporado a la edición estándar de la plataforma. En Swing, existen clases predefinidas, que encapsulan el comportamiento de contenedores y que poseen métodos utilizados para la ubicación de componentes dentro de los mismos. En la siguiente figura podremos observar la representación de un contenedor, definido en un *template Java*.

```
<containers>
<container type="TitleWindow" name="JPanel" >
      <methods>
            <method name="setLayout">
                  <attributes>
                        <attribute name="value1" value="null" />
                  </attributes>
            </method>
            <method name="setBounds">
                  <attributes>
                        <attribute name="value1" value="$x" />
                        <attribute name="value2" value="$y" />
                        <attribute name="value3" value="$width" />
                        <attribute name="value4" value="$height" />
                  </attributes>
            </method>
            <method name="setBorder">
                  <attributes>
                        <attribute name="value1"
                         value='BorderFactory.createTitledBorder("$text")'
                        />
                  </attributes>
            </method>
      </methods>
      <componentLocation name="JPanel">
            <methods>
                  <method name="setBounds">
                        <attributes>
                              <attribute name="value1" value="$x" />
                              <attribute name="value2" value="$y" />
                              <attribute name="value3" value="$width" />
                              <attribute name="value4" value="$height" />
                        </attributes>
                  </method>
            </methods>
      </componentLocation>
</container>
```

Figura 32: Definición de contenedores en un *template Java*

Si comparamos la definición de un contenedor en un template *Java* y la definición de un template *HTML*, podremos notar, que la definición en *Java* es un tanto más larga. Esto se debe a que la instanciación de contenedores y componentes en Java se realiza a través de métodos, a diferencia de la estructura jerárquica *HTML*, que nos permite escribir un tag dentro de otro. En este caso, el atributo *type* del tag *container*, define el tipo de contenedor a representar, como ya se ha explicado en la representación de un template *HTML*. El atributo *name*, en este caso, define el nombre de la clase *Java*, con la cual se representará el contenedor, recordemos que un *template HTML*, dicho atributo tenía como valor el nombre de

un tag. Dentro de la definición del *container*, se podrían escribir atributos, al igual que en la definición de un *component*, estos atributos representarían los parámetros del constructor de la clase que implementa el contenedor, pero en el caso de *JPanel*, no se requieren parámetros de inicialización. Si bien en la figura anterior, el tag *container* no define atributos, si se puede observar, la definición de una serie de métodos. Los métodos presentes en el *container* se utilizarán para inicializar el mismo, al igual que en la definición de un *component*, los métodos tendrán un atributo denominado *name*, que definirá el nombre del método, y dentro del mismo, se definirán los atributos que representarán a los parámetros de invocación del método.

```
JPanel jPanel23 = new JPanel();
jPanel23.setLayout(null);
jPanel23.setBounds(25, 46, 662, 265);
jPanel23.setBorder(
BorderFactory.createTitledBorder(
"Formulario de ingreso"));
```

Figura 33: Representación de un contenedor *Java*

La *Figura 33*, muestra finalmente, como quedará el código fuente generado a partir del *template Java* descripto anteriormente. Como se observa, el contenedor es de tipo *JPanel*, esto se debe a que fue definido con el atributo *name* del tag *container*. Al crear una instancia de la clase *JPanel*, la aplicación de generación define el nombre de la misma. En el caso de *HTML*, los tags se identifican con los atributos *name* o *id*, pero pueden ser omitidos dado que solo se utiliza para hacer referencia al tag, al utilizar programación dinámica (interacción con el tag a través de un lenguaje de scripting). En el caso de una instancia de clase, la definición del nombre claramente es obligatoria y por lo tanto, la aplicación debe tener una metodología para definir dichos nombres. En este caso, la metodología es muy sencilla, se toma el nombre de la clase y se le concatena un número secuencial para habilitar la posibilidad de tener varios componentes, del mismo tipo, en el código fuente resultante. Posterior a la instanciación, podemos observar la invocación de tres métodos diferentes sobre la nueva instancia. Los tres métodos ejecutados, son los observados en la *Figura 32* y

definidos dentro de los tags *methods*, algunos de ellos hacen referencia a las palabras reservadas o variables que mencionamos anteriormente y que pueden ser utilizadas a lo largo del template. Cuando describimos la representación de un contenedor en el *template HTML*, hicimos referencia al tag *componentLocation*, utilizado para determina como los componentes se ubican dentro del contenedor. En el caso de un *template Java*, también se deberá definir dicho tag, en el ejemplo analizado, podremos observar cómo se define el tag *componentLocation* y los métodos necesarios para ubicar el componente dentro del contenedor. En el siguiente ejemplo, podremos observar cómo quedará finalmente el código fuente de un contenedor y un componente ubicado dentro del mismo.

```
JPanel jPanel23 = new JPanel();
jPanel23.setLayout(null);
jPanel23.setBounds(25, 46, 662, 265);
jPanel23.setBorder(
BorderFactory.createTitledBorder(
"Formulario de ingreso"));
JLabel JLabel24 = new JLabel("Nombre:");
JLabel24.setBounds(25, 35, 59, 25);
JPanel23.add(JLabel24);
```

Figura 34: Representación de un contenedor *Java* y un componente

En la figura anterior, podemos observar la creación de una etiqueta *JLabel* que será ubicada dentro del contenedor. El método *setBounds*, definido en el template dentro del tag *componentLocation*, permite determinar la ubicación del componente dentro del contenedor. El método *add*, no se especifica como método del tag *componentLocation*, esto se debe a que es agregado automáticamente, por la clase que genera el código *Java* a partir del *template*, más adelante definiremos el comportamiento de estas clases y su implementación.

Definición del tag main

Los tags *components* y *containers* permiten definir la representación de distintos elementos dentro del diseño, pero la generación de pantallas, requiere además la especificación de otros tipos de detalles para poder implementarse. Si pensamos en términos de *HTML* y tenemos en cuenta la *Figura 20*, el documento final contendrá un cuerpo o estructura definido, en *HTML*, a través de los tags *html*, *body* y *head*. La definición de los componentes o contenedores no especifica cómo se definirán estos tags, es decir, que contendrá cada uno de ellos, así como tampoco, los componentes y contenedores no contienen una especificación de cómo deben ser agregados dentro del tag *body* de un documento *HTML*.

Si ahora pensamos en términos del lenguaje *Java*, la generación dará como resultado una clase, la cual a su vez, implementará la pantalla en sí misma. La clase final deberá tener una estructura predefinida, la cual, deberá ser especificada de alguna manera junto con los estilos y otros detalles relacionados al proceso de generación. Básicamente, tanto en *HTML* como *Java*, se deberá definir una estructura principal, que agrupará a todos los componentes y contenedores definidos, el tag *main*, tiene dicha función y por lo tanto deberá ser definido de manera obligatoria. Al igual que hicimos con los otros dos tags, vamos a empezar analizando un ejemplo del tag *main* para un *template* de tipo *HTML*.

En la *Figura 35*, podemos observar una implementación del tag *main*, dado que dentro del tag se escribirá una porción de código que será tomada como el marco o la estructura del código fuente final, dicho código no requerirá ningún tipo de análisis a la hora de parsear el *template*. Para evitar el parseo o análisis del contenido de un tag *xml*, se debe utilizar la palabra reservada *cdata*, la misma es reconocida por cualquier parser y determina que su contenido no debe estar sujeto a las mismas políticas de análisis que el resto de los tags y sus contenidos. De esta manera, podremos escribir cualquier tipo de código dentro del tag *main* y así definir lo que será el marco principal del código fuente generado finalmente.

```
<main>
      <body>
      <![CDATA[
            <html>
                  <head>
                        <style>

                        .container {
                        color:#4E4628;
                        border:#C3BCA4 1px solid;
                        }

                        input, textarea {
                        background-color:#EFEBDE;
                        color:#0B0B0B;
                        border:#C3BCA4 1px solid;
                        margin:2px 2px 0px 0px;
                        font:normal 14px/20px Arial, Helvetica, sans-serif;
                        }

                        div {
                        color:#0B0B0B;
                        font:normal 14px/20px Arial, Helvetica, sans-serif;
                        }

                        </style>
                  </head>
                  <body>
                        $generatedCode
                  </body>
            <html>
      ]]>
      </body>
</main >
```

Figura 35: Representación del tag *main* para un *template HTML*

Como se observa en la *Figura 35*, simplemente hemos declarado la estructura básica de una página *HTML*, donde los tags *html*, *head* y *body* definen la estructura de lo que será la pantalla final. Dentro del tag *head*, se puede observar la definición del tag *style*, este tag es utilizado en *HTML* para poder darle un estilo particular a los componentes y elementos de la pantalla. La definición de estilos en *HTML* está cubierta por el lenguaje *CSS*, también conocido como *hojas de estilo en cascada*, pero lo que se debe destacar de este punto, es que dentro del tag *main*, podremos definir el *look & feel* de los componentes presentes en la pantalla final. Dentro del tag *body*, se puede observar la palabra reservada *$generatedCode*, esta palabra reservada, es utilizada por el sistema de generación para poder determinar donde se debe insertar el código fuente generado para los componentes y contenedores.

El proceso de generación interpretará, como hemos analizado anteriormente, los componentes y contenedores presentes en el diseño y posteriormente generará un código fuente final que será insertado en el punto donde se define la palabra reservada *$generatedCode*. De esta manera, el código fuente final, será el código presente dentro del tag *main*, con el agregado del código fuente de todos los componentes y contenedores presentes en el diseño. Como mencionamos anteriormente, el ejemplo anterior define una serie de estilos predefinidos que pueden ser utilizados por los componentes, para utilizar los mismos, se deberá definir para cada componente, el atributo que indica la utilización del estilo teniendo en cuenta el nombre con el cual se definió. Para los componentes básicos, el estilo será aplicado automáticamente ya que el mismo se define sobre el tipo de componente, esto se puede observar para el caso de un tag del tipo *input* o *div*. De esta manera y con la definición del tag *main*, cubrimos todos los aspectos referentes a la generación de una pantalla en *HTML*, a continuación, al igual que hicimos con los tags *components* y *containers*, vamos a definir el tag *main* para una pantalla generada con *Java*

Hemos determinado que el tag *main*, definirá el marco de código fuente principal de la pantalla a generar. En *HTML*, como hemos explicado, la pantalla deberá respetar una estructura de tags que definimos en la *Figura 35*, pero en el lenguaje *Java*, deberemos definir una clase que implementará la pantalla en sí misma. La generación de una clase en Java, implica definir el nombre de la misma junto con las importaciones necesarias para su compilación, a diferencia de *HTML*, si la clase *Java* no compila no se podrá implementar finalmente la pantalla. En el lenguaje *HTML*, existen una serie de tags que implementan los componentes visuales que deseamos representar en la pantalla, estos tags están implementados y son reconocidos por los distintos browsers, por lo tanto no se requiere una importación. Si se desean implementar otro tipo de componentes más complejos, se deberán importar las librerías necesarias para la implementación de los mismos, estos aspectos se describirán en el anexo. Dado que los componentes visuales de *Java* están implementados en distintos paquetes, se deberán especificar en el tag *main*, todas las clases de componentes que el *template* pueda representar.

```
<main>
      <body>
      <![CDATA[

                  import java.awt.Container;
                  import java.awt.Insets;

                  import javax.swing.JButton;
                  import javax.swing.JLabel;
                  import javax.swing.JTextField;
                  import javax.swing.JCheckBox;
                  import javax.swing.JComboBox;
                  import javax.swing.JRadioButton;
                  import javax.swing.JFrame;
                  import javax.swing.JPanel;
                  import javax.swing.BorderFactory;
                  import javax.swing.JTextArea;

                  public class SwingGeneratedExample {

                      public static void main (String [] args)
                      {
                              $generatedCode
                      }

                      public void execute(){
                        main(null);
                      }
                  }
      ]]>
      </body>
</main >
```

Figura 36: Representación del tag *main* para un *template Java*

En la *Figura 36*, se puede visualizar la definición de un tag *main* para un *template* de generación *Java*. Como mencionamos anteriormente, lo que deberá definir el tag es una clase *Java*, la cual deberá importar todas las librerías de los componentes visuales que podrá representar el *template*. En el tag *components*, se definían los componentes visuales que podía representar el *template*, y para cada componente, se definía un atributo *name* que definía el nombre de la clase que implementaba dicho componente. Teniendo en cuenta que para la definición de componentes se deberá conocer el nombre de las clases, en el tag *main* de un *template Java*, se deberán importar las clases visuales que darán soporte a los componentes. Como se observa también en la figura, las importaciones hacen referencia a diferentes clases de los frameworks *awt* y *swing*, dichos frameworks son utilizados para la representación estándar de componentes visuales en *Java*, pero no son la única opción.

Existen otras alternativas, como *swt*, que permiten representar los componentes de otra manera, cambiando su implementación y *look & feel*, por esta razón, en el anexo comentaremos como implementar un *template Java* utilizando librerías no estándar o de terceros. Como podemos observar en el ejemplo, nuevamente aparece la palabra reservada *$generatedCode*, la misma será utilizada para definir el punto en el que se debe insertar todo el código generado, es decir, los componentes y contenedores que se representarán en lenguaje *Java*. La clase generada a partir del *template* anterior, tendrá el código visual implementado en el método *main*, método estándar utilizado para la ejecución de un programa *Java*. Esto significa, que se podrá ejecutar la clase mostrando una pantalla implementada en *Java Swing* con todos los componentes presentes en el diseño dibujado con *Balsamiq Mockups*. El método *execute*, simplemente invoca al método principal y se deberá implementar en el *tag* main, como muestra el ejemplo, si se desea que la pantalla generada, se muestre desde la aplicación de generación una vez finalizado el proceso de creación del código fuente. Como podemos observar, la definición del tag *main* para un template *Java* no presenta demasiada complejidad, esto se debe entre otras cosas, a la interpretación que realizará la aplicación de generación de código, la cual a su vez, agregará la funcionalidad necesaria para que la clase *Java*, que definir los componentes visuales, pueda compilar correctamente.

Con la definición del tag *main* hemos completado la descripción de los *templates*, sus características y formas de definición. La lógica de interpretación utilizada por los *templates* será implementada por la aplicación de generación, contemplando los detalles que hemos descriptos anteriormente. En el siguiente apartado definiremos la implementación de la aplicación de generación, describiendo su arquitectura y su interfaz visual.

Arquitectura del generador

En los apartados anteriores, cubrimos dos aspectos fundamentales para la generación final del código fuente de un diseño. Por una parte, describimos los aspectos vinculados a la interpretación de las pantallas diseñadas con *Balsamiq Mockups*, definiendo el tipo de *parser* a utilizar y describiendo los elementos presentes en los archivos de salida. Posteriormente, se definió el concepto de *templates*, describiendo el lenguaje que utilizará la aplicación para escribir e interpretar dichos *templates*. Los dos puntos anteriores, son las entradas necesarias para que el proceso de generación pueda llevarse a cabo, es decir, son los puntos de información necesarios para que el proceso pueda determina que se deberá generar y como. Teniendo en cuenta los aspectos mencionados, empezaremos a describir los componentes que formarán parte de la arquitectura de la aplicación de generación.

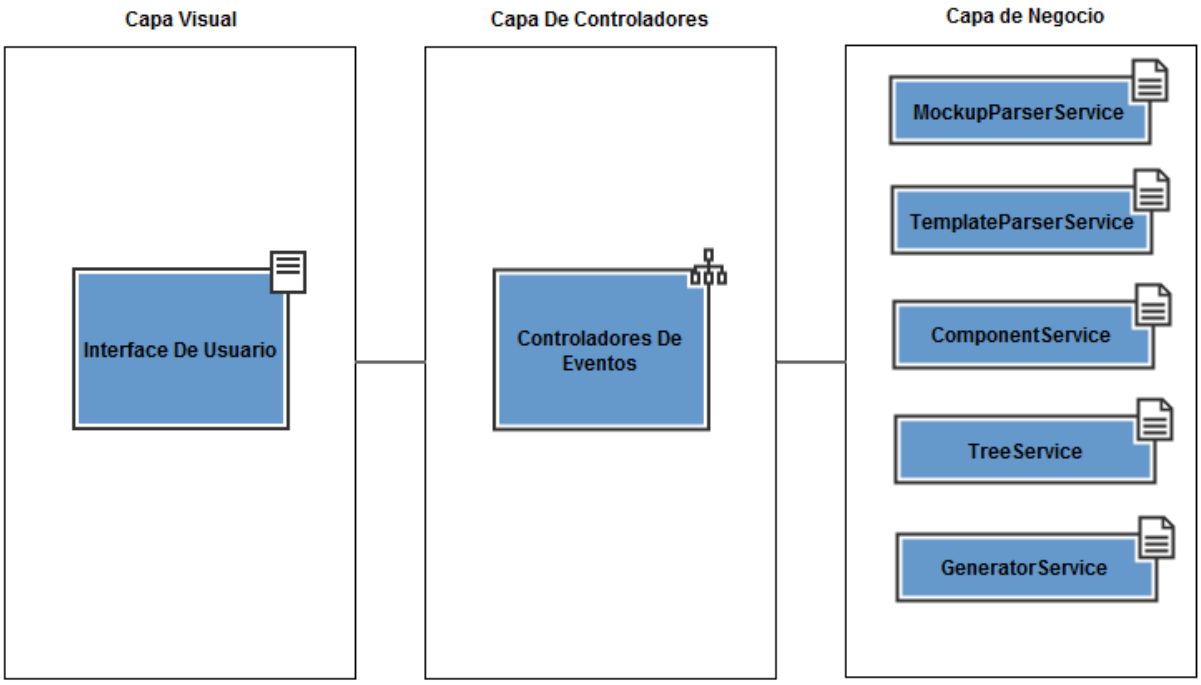

Figura 37: Representación de la arquitectura de la aplicación

La arquitectura de la aplicación, contempla un patrón de diseño muy difundido y conocido como *Modelo-Vista-Controlador*, dicho patrón de diseño, divide las responsabilidades, de las tres capas mencionadas, facilitando la división de las mismas y permitiendo identificar con claridad donde están implementadas determinadas funcionalidades. Como se observa en la *Figura 37*, la arquitectura de la aplicación de generación implementa la división de capas que define el patrón y divide, en cada una de estas capas, los componentes y sus responsabilidades. En la capa visual o representación de la vista, tendremos los componentes que permitirán construir la pantalla de la aplicación de generación. Los componentes visuales se comunicarán, a través de eventos, con los controladores que serán los responsables de interpretar dichos eventos, delegando a su vez, la responsabilidad a los servicios correspondientes ubicados en la capa de negocio o modelo. Como se observa en la figura anterior, no especificamos todos los componentes visuales y de control presentes en las dos primeras capas, esto se debe a que se trata de componentes muy simples que serán detallados posteriormente cuando se describa la interface de usuario de la aplicación de generación. Los servicios presentes en la capa de negocio, han sido detallados en la *Figura 37*, debido a que en ellos radica la implementación total de los puntos descriptos en apartados anteriores y la responsabilidad del proceso de generación. Es por esta razón, que describiremos a cada uno de ellos para poder determinar cómo han sido divididas las responsabilidades en los distintos servicios. Es importante destacar, que en la arquitectura de la aplicación se utiliza el framework *Spring*, un framework muy difundido en la actualidad, que permite definir con mayor facilidad una arquitectura orientada a servicios. Toda la lógica de negocio, presente en lo que sería el modelo de la aplicación, ha sido implementada en forma de servicios por medio de *Spring*, esto nos permitirá facilitar la interacción entre servicios, reduciendo el acoplamiento de los mismos y configurando dicha interacción en lo que será el contexto de la aplicación. En el contexto de la aplicación, se definen los diferentes servicios y las dependencia que existen entre ellos, dicho contexto, no es nada más que un archivo *xml* necesario para que el framerowk de *Spring* pueda facilitarnos el acceso a dichos servicios. Este archivo *xml* permite identificar con claridad las dependencias de cada servicio permitiendo, al mismo tiempo, manejar el acoplamiento de una forma más sencilla. Es necesario destacar, que al implementar la lógica de negocio en forma de servicios, se habilita la posibilidad de presentar en un futuro a los mismos como *web services*, permitiendo el uso de estos a través de la web.

```
<beans>

	<bean id="mockupParser.service"
	class="ar.com.mockup.service.impl.MockupParserServiceImpl" />

	<bean id="component.service"
	class="ar.com.mockup.service.impl.ComponentServiceImpl" />

	<bean id="tree.service"
	class="ar.com.mockup.service.impl.TreeServiceImpl" />

	<bean id="templateParser.service"
	class="ar.com.mockup.service.impl.TemplateParserServiceImpl" />

	<bean id="generator.service"
	class="ar.com.mockup.service.impl.GeneratorServiceImpl">
		<property name="htmlComponentWriter">
			<ref bean="html.writer" />
		</property>
		<property name="swingComponentWriter">
			<ref bean="swing.writer" />
		</property>
	</bean>

	<bean id="html.writer"
	class="ar.com.mockup.service.writer.impl.HTMLComponentWriter" />

	<bean id="swing.writer"
	class="ar.com.mockup.service.writer.impl.SwingComponentWriter" />

</beans>
```

Figura 38: Contexto de la aplicación

En la *Figura 38*, se observa el contexto de la aplicación y la definición de los diferentes servicios. Por medio del framework de *Srping*, podemos obtener referencia a estos servicios invocando el *id* que los identifica, esto genera un desarrollo simple y ordenado ya que todos los servicios se encuentran definidos en dicho archivo de contexto como *beans* de *Spring*. Podemos observar también otras dos definiciones que no aparecen como servicios en la *Figura 37*, estos son *html.writer* y *swing.writer*, estos componentes, también definidos como *beans* de *Spring*, son utilizados para escribir el código *HTML* o *Swing* a partir de los templates y nos referiremos a ellos posteriormente, cuando describamos la funcionalidad del servicio de generación, ya que como podemos observar, dicho servicio los tiene como dependencias.

Servicio de interpretación de mockups

El primero de los servicios a describir, está relacionado con la interpretación de los *mockups*. Como aclaramos anteriormente, sin la interpretación o parseo de los archivos *bmml*, el proceso de generación no podría funcionar ya que no tendría una de sus dos entradas. En la aplicación de generación, la lógica de interpretación de los *mockups* está encapsulada en el servicio *MockupParserService*, identificado en el contexto de *Spring* bajo el id *mockupParser.service*. Como hemos aclarado en el apartado relacionado a la interpretación de los archivos *bmml*, se deberán interpretar los tags presentes en dichos archivos y pasar su contenido a un modelo de clases, a través del cual, se pueda trabajar en *Java*. Mencionamos que dicha tarea se realizaría a través de otro *framework* denominado *XStream*, el cual nos permitía pasar la información de un *xml* a un modelo de clases. En la implementación del servicio *MockupParserService*, se encontrarán implementados los métodos que permitirán realizar la tarea antes mencionada.

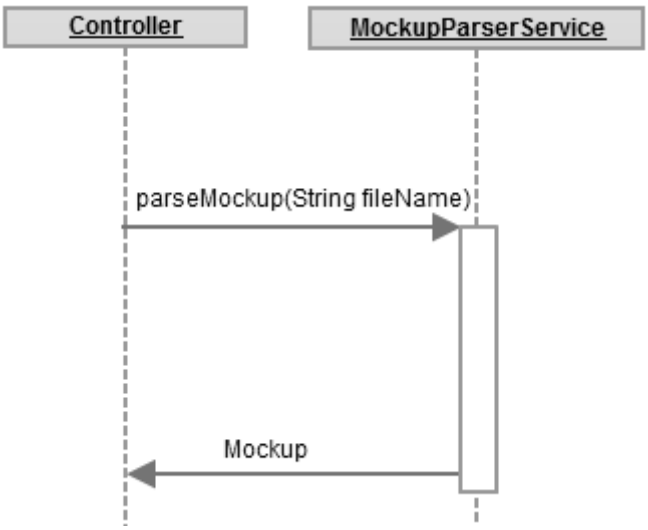

Figura 39: Diagrama de secuencia del servicio de parseo de *mockups*

El diagrama de secuencia de la figura anterior, muestra la invocación del método de parseo presente en el servicio *MockupParserService*. Internamente y a través de *XStream*, el servicio interpretará el archivo *bmml*, referenciado por el *path* pasado como parámetro, y retornará una instancia de la clase *Mockup*. La instancia de la clase *Mockup*

contendrá, de esta manera, toda la información relevante para poder determinar cómo se constituye el diseño de pantalla. La estructura de la clase *Mockup* puede observarse en la *Figura 6*, presente en la explicación de la interpretación a través de *XStream*.

Servicio de interpretación de templates

Una vez que tenemos la información relacionado al diseño y encapsulada en un modelo de clases, requeriremos la información relacionada a la representación de los componentes presentes en dicho diseño. El servicio *TemplateParserService*, nos permitirá interpretar la información, presente en el template seleccionado, para realizar posteriormente la generación de código.

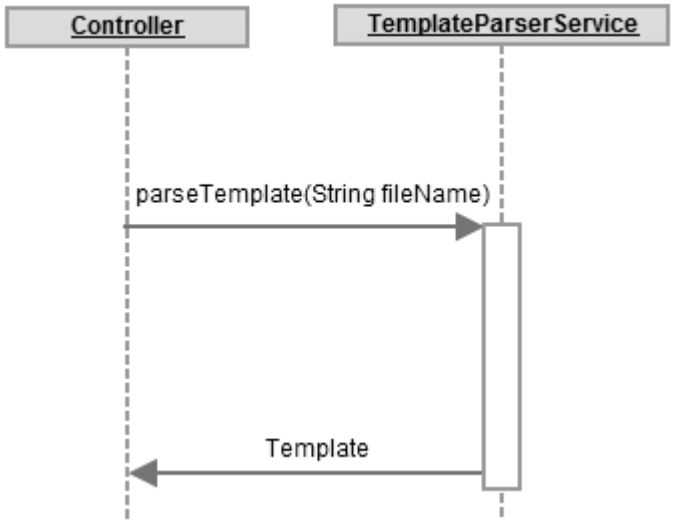

Figura 40: Diagrama de secuencia del servicio de parseo de *templates*

Al igual que el servicio *MockupParserService*, la interpretación del template se realizará por medio de *XStream*, esto se debe a que el template, como ya hemos especificado, es también un archivo *xml* y por lo tanto, se deberá transformar en una estructura de instancias basadas en clases *Java*. El método *parseTemplate*, recibirá como único parámetro el *path* del archivo *template* a interpretar y como retorno obtendremos una instancia de la clase *Template*. Si bien el diagrama de secuencia es sencillo, en el método *parseTemplate* se ubicará toda la

lógica, relacionada al manejo de *XStream*, para obtener finalmente la información del *template*.

Servicio de componentes

El servicio de interpretación de *mockups* nos ofrecerá información relacionada al diseño de la pantalla a generar, pero como hemos mencionado anteriormente, no nos determinará si dicho diseño se encuentra correctamente estructurado. En el apartado relacionado a la interpretación, también mencionamos la necesidad de validar el diseño parseado, es decir, aplicar una serie de validaciones que nos permitan determinar si el diseño es correcto o no. Las diferentes validaciones que describimos, tenían como finalidad, poder determinar si alguno de los componentes o contenedores no cumplía con determinados lineamientos. El servicio *ComponentService*, tendrá como finalidad, poder convertir la información encapsulada en una instancia de *Mockup* y referente a un diseño, en una estructura de componentes y contenedores contemplando las validaciones previamente mencionadas. Como habíamos mencionado anteriormente, el *Mockup* no nos proveerá información relacionada con la jerarquía de componentes y contenedores, por lo tanto, el proceso de validación tendrá como finalidad poder determinar si la estructura es correcta o no. La secuencia de validaciones, mencionada en el apartado de interpretación, contemplaba una validación de estructura, una validación de jerarquía y finalmente, una validación de componentes. En la *Figura 41*, se observa el diagrama de secuencia de uno de los métodos presentes en el servicio de componentes. El método *getContainerHierarchy*, implementará la secuencia de validaciones mencionada para determinar si el diseño contempla los requisitos necesarios para continuar con el proceso de generación. Si observamos el diagrama, notaremos la presencia de varios bucles, esto se debe a la necesidad de comparar los componentes entre sí, a modo de determinar si son o no contenedores y donde están posicionados. El primer bucle determinará si el componente es o no un contenedor, almacenando de esta manera los contenedores en una lista. En este punto se aplicará la primera validación, identificando si en el diseño existe al menos un componente de tipo contenedor. Posteriormente, se recorrerá la lista de contenedores para determinar si alguno de ellos esta superpuesto con otro, pudiendo determinar de esta manera, si el diseño presenta una estructura correcta en relación a los contenedores y aplicando la validación de jerarquía. Una

vez identificados los contenedores, se volverá a recorrer la lista de componentes validando la posición de cada uno de ellos dentro de un contenedor, de esta manera se aplicará la validación de componentes, determinando si todo los componentes está dentro de un contenedor. En última instancia, se recorrerá la lista de contenedores para construir una estructura de objetos que aplique el patrón *composite*, retornando finalmente un objeto de tipo *container* que encapsulará toda la jerarquía de contenedores y componentes en su interior. En el servicio se podrán encontrar otros métodos, algunos de ellos presentes en el siguiente diagrama, pero los mismos no serán explicados dado que solo cubren detalles específicos relacionados con la implementación.

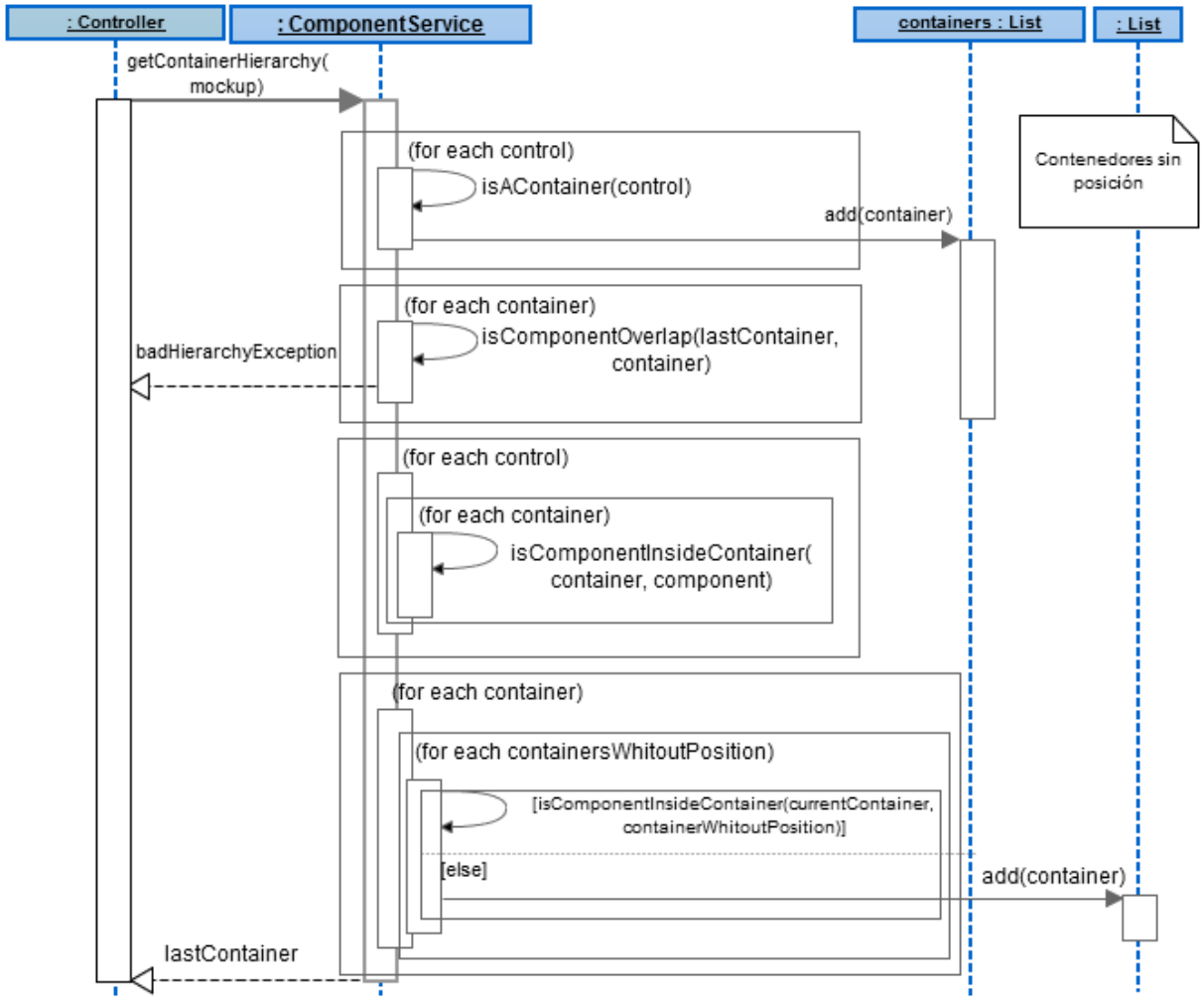

Figura 41: Diagrama de secuencia del servicio de componentes

De esta manera, el servicio *ComponentService* contendrá toda la lógica de validación jerárquica necesaria para identificar, componentes solapados, componentes de tipo contenedor, posicionamiento de componentes dentro de un contenedor y la capacidad de convertir la estructura de un *Mockup* en una estructura de objetos que aplique el patrón *composite*.

Servicio de construcción del árbol de componentes

El servicio de componentes, nos permitirá obtener una estructura jerárquica del diseño de pantalla, a partir del cual, se realizará la generación de código. Dicha estructura, como mencionamos anteriormente, se puede representar como un árbol, pudiendo identificar fácilmente como se interpreto la ubicación de los componentes y contenedores. El servicio *TreeService*, utilizará las librerías de *Prefuse*, para poder transformar la estructura de componentes en un árbol visual. *Prefuse* es un conjunto de herramientas de visualización utilizado para la creación de gráficos interactivos, y en el caso particular de nuestro proyecto, solo nos limitaremos a utilizar las librerías necesarias para la representación de un árbol. En la *Figura 42*, se observa el diagrama de secuencia simplificado del método que nos permitirá construir el árbol de componentes y contenedores. A través del método *buildTree*, el servicio recibirá el contenedor principal, que contiene internamente toda la estructura jerárquica de componentes. Posteriormente, se instanciará un nuevo árbol, y se recorrerá el contenido del contenedor principal, diferenciando entre componentes y contenedores. Los componentes serán agregados al árbol como nuevos nodos, y los contenedores se volverán a recorrer internamente repitiendo el ciclo, dando lugar a la invocación recursiva del método *addChild*. Finalmente, obtendremos una instancia de la clase *Tree*, propia de las librerías de *Prefuse*, la cual será retornada a la vista por medio del controlador que invocó al servicio *TreeService*. En el siguiente apartado, relacionado a la interface de usuario de la aplicación, se mostrará la visualización de un árbol terminado y representado visualmente. Cabe destacar, que el diagrama de secuencia simplemente muestra la estrategia de construcción del árbol, ya que en el servicio *TreeService*, se podrán encontrar otros métodos necesarios para interactuar con *Prefuse*, dichos métodos no serán abordados ya que solo presentan detalles técnicos de la implementación.

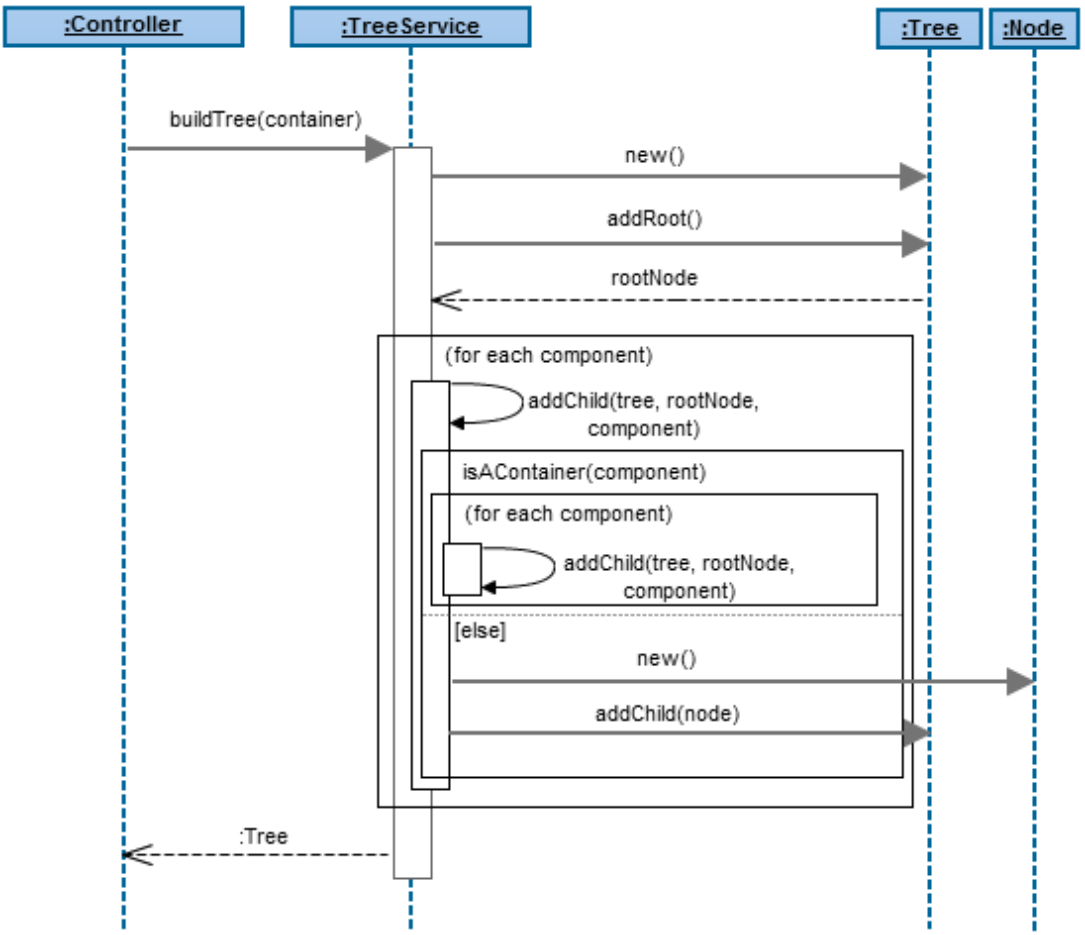

Figura 42: Diagrama de secuencia del servicio de construcción del árbol

Servicio de generación

El último de los servicios que describiremos será el *GeneratorService*, este servicio se ocupará de implementar el comportamiento necesario para poder realizar la generación final del código fuente. Para que dicha generación pueda ser realizada, el servicio de generación deberá recibir la estructura jerárquica (previamente construida por el servicio *ComponentService)* y el *Template* seleccionado para la generación (previamente obtenido a través del servicio *TemplateService).* En el servicio de generación, también se deberá determinar qué tipo de código fuente se creará como salida, es decir, si se trata de *HTML* o *Java*, ya que en función de esa diferencia, se utilizará uno de los dos escritores de código

conocidos por el servicio, este último punto lo describiremos más adelante al referirnos a las dependencias *html.writer* y *swing.writer*.

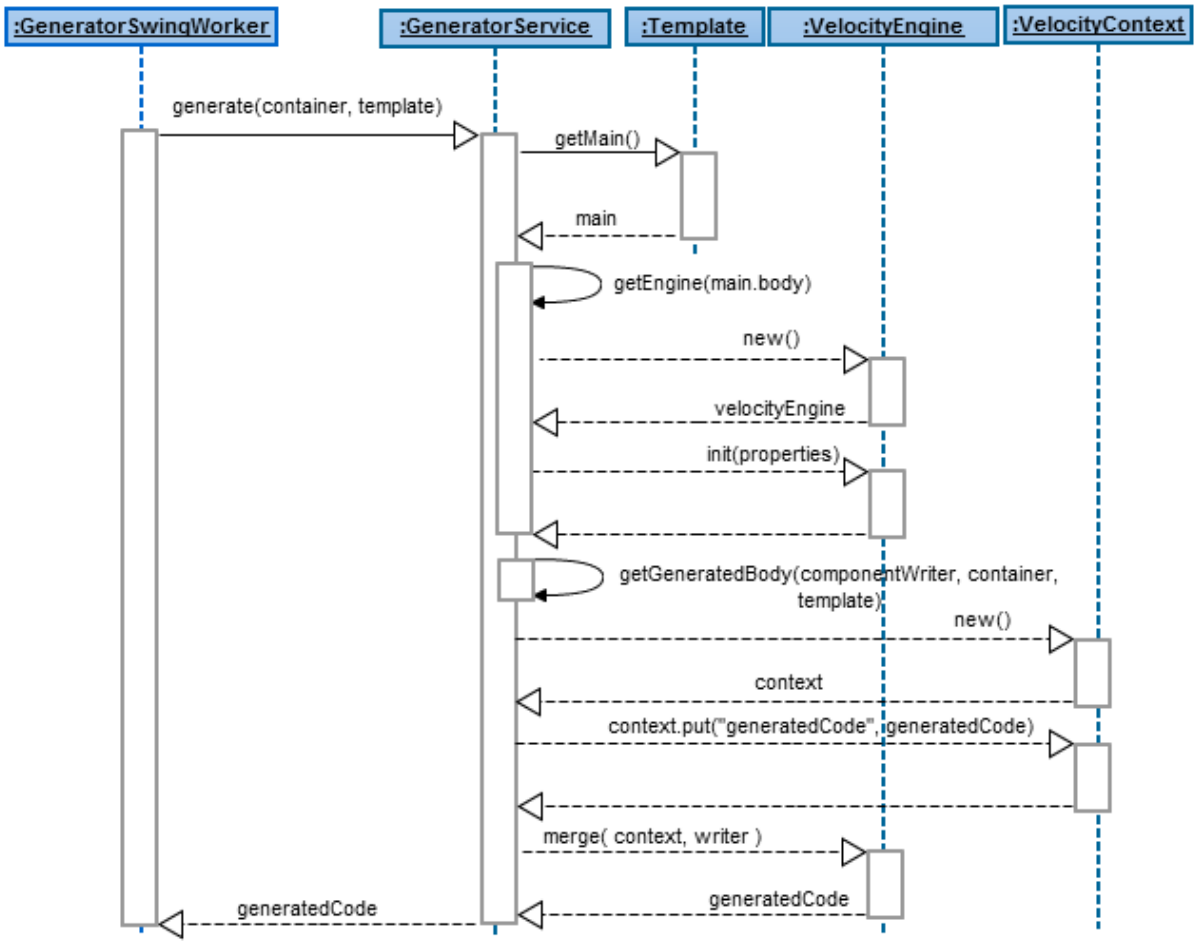

Figura 43: Diagrama de secuencia del servicio de generación

En el diagrama de secuencia de la *Figura 43*, se puede observar parte del funcionamiento de la generación final de código. La clase *GeneratorSwingWorker*, es utilizada para poder manejar la invocación del proceso de generación en un hilo aparte, esto se debe a que la aplicación de generación fue implementada con *Swing* (framework utilizado para la *GUI*), y el hilo principal es utilizado para la renderización de componentes en la pantalla. Como se puede observar, desde una instancia de la clase *GeneratorSwingWorker*, se invocará al servicio de generación pasando como parámetros las dos entradas necesarias para dicho proceso: la estructura de componentes (*container*) y la forma de representar los mismos

(*template*). El servicio *GeneratorService*, obtendrá del *template* la sección *main*, la cual representaba la estructura principal del código y dentro de la cual se ubicarán los componentes y los contenedores. El siguiente paso consistirá en la unificación o merge del código, por un lado tendremos el código generado de todos los componentes presentes en el diseño y representados en función del *template*, y por el otro lado, tendremos la estructura principal del código, dentro de la cual, se deberá introducir el código fuente de los componentes. Para este proceso se utilizará el framework *Velocity*, utilizado comúnmente como motor de generación de código, el cual nos permitirá realizar el merge. Como se había mencionado en el capítulo referente a *Templates*, el código principal contenido en el tag *main*, deberá marcar, a través de la palabra reservada *$generatedCode*, la ubicación de los componentes. Como se observa en el diagrama de secuencia, el servicio instanciará el motor de *Velocity* para realizar la operación de merge y posteriormente, obtendrá el código fuente de todos los componentes presentes en el diseño a través del método *getGeneratedBody*. En la invocación del método *getGeneratedBody*, se pasaran como parámetros, un *componentWriter* junto con el *container* y el *template* recibidos al inicio del proceso. La instancia de *componentWriter*, será utilizada para la escritura del código fuente de cada componente y dependerá del lenguaje especificado en el *template*. El servicio de generación, determinará el tipo de lenguaje (*HTML* o *Java*) del código fuente a generar, y en relación a ello, utilizará una de sus dos dependencias: *html.writer* o *swing.writer*. Ambas dependencias implementarán la misma interface, por lo tanto, si en un futuro se quisiera generar código fuente en otro lenguaje o de otra manera, se podría agregar una nueva dependencia al servicio de generación, siempre y cuando implemente la misma interface. Básicamente, el *componentWriter* encapsulará la lógica y las reglas de sintaxis para escribir código fuente en cada lenguaje, y se basará en la definición del *template* para determinar cómo se representa cada componente. Si a futuro se deseara escribir código fuente en un lenguaje como *.NET*, se deberá implementar el comportamiento de un *componentWriter* y escribir una clase que entienda la sintaxis de dicho lenguaje de programación, es decir, que contenga la lógica para escribir código fuente con la sintaxis de *.NET*. De esta manera, el método *getGeneratedBody* retornará el código fuente de todos los componentes y contenedores, el cual se introducirá en el contexto de *Velocity*, bajo el nombre *generatedCode*. Posteriormente se realizará el merge, invocando el motor de *Velocity* y pasándole como parámetro un contexto, con el valor *generatedCode*, y un *StringWriter* para obtener finalmente un *String* con la representación final del código fuente.

Posteriormente, y con el código ya generado, se escribirá un archivo final que podrá ser utilizado o exportado a otra aplicación, dicho archivo, tendrá una extensión diferente en función del lenguaje especificado en el template. Si se trata de un template basado en Java, el archivo generado será *.java*, y si se trata de *HTML*, se generará un archivo *.html* que podrá ser abierto a través de un *browser*.

Todos los servicios descriptos, forman parte de la capa de lógica de negocio de la aplicación de generación, a través de ellos se podrá realizar la interpretación de los *mockups* y templates, así como también, iniciar el proceso de generación. Los servicios han sido construidos en base al framework de *Spring*, y por lo tanto, implementan el comportamiento definido por interfaces o contratos. Estos servicios serán consumidos por los controladores de eventos de la aplicación, para atender las solicitudes generadas por los distintos componentes gráficos presentes en la interface de usuario. Debe tenerse en cuenta, que el hecho de que la lógica de negocio haya sido implementada en forma de servicios, habilitará en un futuro, la posibilidad de exponer los servicios como *web services*, permitiendo el acceso a los mismos desde otras aplicaciones por medio de internet. En el siguiente apartado, describiremos finalmente la interface gráfica de la aplicación, que a través de sus componentes, terminará invocando a los servicios descriptos anteriormente.

Interface Gráfica del generador

La interface gráfica del generador, permitirá al usuario seleccionar el conjunto de entradas que disparará posteriormente el proceso de generación. Dado que el proceso de generación solo requerirá dos tipos de entradas (el *mockup* y el *template*), la interface será relativamente sencilla. Como mencionamos anteriormente, la interpretación del *mockup* será una de las entradas del proceso de generación, del mismo dependerá la definición de la pantalla, es decir, la estructura y distribución de componentes presentes en la misma. Es por esa razón, que la selección visual del *mockup* aparecerá como primera opción en la distribución de los componentes de la interface.

Figura 44: Selección de Mockups

En la primera sección de la interface gráfica, aparecerá la posibilidad de seleccionar el archivo *mockup* (bmml) con la representación de la pantalla. Una vez seleccionado, será interpretado a través del servicio correspondiente y se visualizará una estructura en forma de árbol en la parte derecha de la pantalla. Como se observa en la figura anterior, el cuadro titulado *Estado de la estructura de Mockup*, permitirá la visualización de errores de diferente tipo, que estarán directamente vinculados a las validaciones previamente

mencionadas. Como se observa en la *Figura 44*, los mensajes de error presentado en la pantalla, permitirán identificar que componentes se encuentra mal ubicados o solapados identificando a los mismo por su *ID*. La siguiente figura muestra el árbol de componentes que será creado al interpretar un *mockup*.

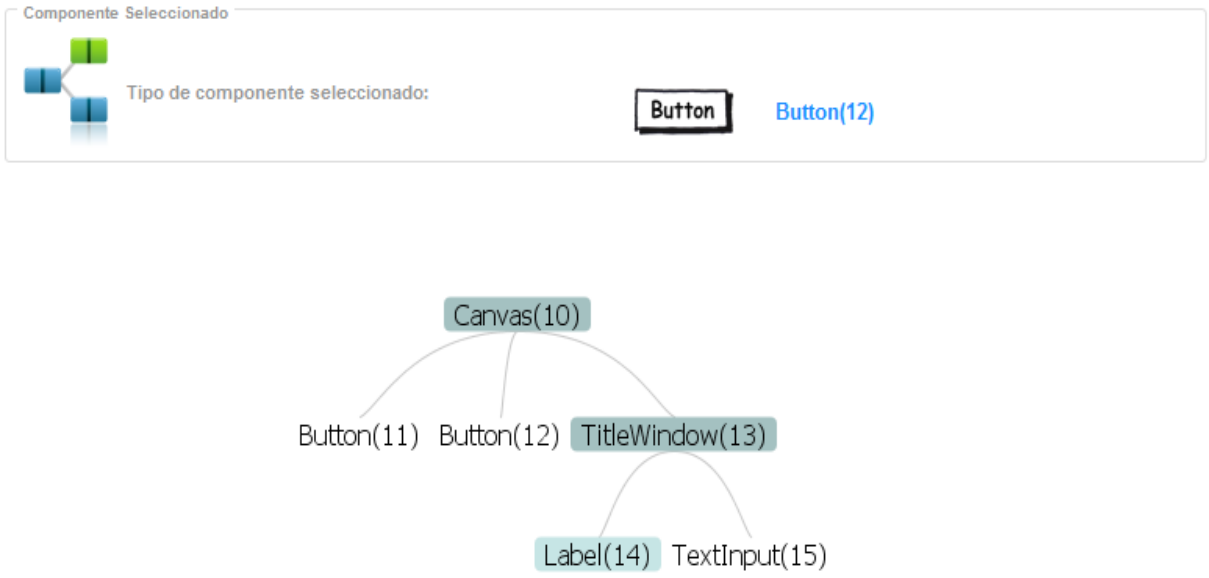

Figura 45: Árbol de componentes

Como se observa en la figura anterior, el árbol permitirá identifica rápidamente la estructura del *mockup* y como ha sido interpretada. La raíz del árbol representará siempre al contenedor principal, y cada componente o contenedor se podrá identificar en la estructura del árbol por su *Tipo* y *ID*, de esta forma, el nodo del árbol denominado *Button (11)* representará al componente de tipo botón que tiene como *ID* el valor 11 en el archivo *bmml*. Posicionando el cursor sobre cualquier componente, se podrá observar en la parte superior de la pantalla, la representación o dibujo del mismo en Balsamiq Mockups, esto permitirá identificar con mayor facilidad la posibilidad de haber interpretado incorrectamente un *mockup*. El árbol de componentes también podrá moverse en diferentes direcciones, esto nos permitirá poder navegar por los diferentes componentes de la pantalla, teniendo en cuenta, que no existirá un límite a la hora de interpretar pantallas con grandes cantidades de componentes y contenedores. Al seleccionar un componente o contenedor, el árbol se posicionará tomando como centro el componente seleccionado y cambiando el color del mismo.

Una vez seleccionada la primera entrada del proceso de generación (el *mockup*), se deberá seleccionar la segunda entrada, es decir, el *template*. El *template* definirá la representación de cada uno de los componentes, es por esta razón, que se incorporarán a la aplicación, diferentes alternativas de *templates.* De esta manera, se podrá demostrar en este trabajo de investigación, que a partir de una definición de pantalla, se pueden obtener diferentes representaciones de la misma, ya sea con diferentes estilos o implementada en diferentes lenguajes.

Figura 46: Selección de *templates*

La *Figura 46* muestra la lista desplegable que nos permitirá seleccionar el *template* a utilizar. En la aplicación se incluirán varios *templates* definidos para *HTML* y un *template* definido para *Java Swing*, a modo de demostrar, las alternativas de generación en los dos lenguajes definidos como alcance del proyecto. Una vez seleccionado el *template*, se podrán definir otras características del proceso de generación, una de ellas será el nombre del archivo de salida, que dependiendo del *template* seleccionado, podrá tener una extensión *.java* o *.html*, cabe destacar, que el usuario solo deberá ingresar el nombre del archivo y la aplicación determinará el tipo de extensión. En la siguiente figura, se puede visualizar el cuadro de texto para ingresar el nombre del archivo junto con la opción de visualizar la generación en un *browser*, esta última permitirá, en caso de seleccionar un template *HTML*, abrir el *browser* automáticamente para mostrar el archivo *.html* generado. Debe tenerse en cuenta que la aplicación ha sido probada en equipos con *Internet Explorer* y *Google Chrome*, siendo estos dos *browser* los más utilizados hoy por hoy en el mercado. Cuando se seleccione un *template*, que defina la representación de los componentes en lenguaje *Java*, la pantalla generada se mostrará automáticamente al finalizar el proceso de generación. El código fuente escrito en *Java* será compilado, y por medio del *classLoader* de la aplicación, se visualizará la pantalla generada.

Figura 47: Selección de otras opciones

Finalmente, en la parte inferior de la pantalla se podrá visualizar el botón de generación, el cuál al ser presionado, iniciará dicho proceso. Si las opciones comentadas anteriormente no han sido seleccionadas correctamente, se mostrarán en la pantalla distintos mensajes de validación para indicar al usuario que opción es incorrecta.

Figura 48: Botón de generación

Cada vez que se seleccione un nuevo *Mockup*, el árbol de componentes se reconstruirá para poder mostrar la nueva estructura de la pantalla y al iniciar el proceso de generación, se tomará la última estructura cargada correctamente.

De esta manera, hemos definido a cada una de las opciones que estarán presentes en la interface gráfica del generador, como se puede observar, la interface es muy sencilla, dado que el proceso de generación solo requiere un conjunto de entradas y opciones para poder ser ejecutado. Todos los servicios descriptos anteriormente, serán utilizados a través de la interface gráfica, por medio de los controladores de eventos relacionados con cada uno de los componentes gráficos comentados.

Conclusión

A lo largo de este informe, hemos demostrado como se puede interpretar la información presente en los diseños realizados por medio de *Balsamiq Mockups*, para posteriormente, ser utilizados como fuente de información para la codificación de una pantalla real. Hemos demostrado que dicha información es suficiente para poder implementar una pantalla gráfica, a través de diferentes lenguajes de programación específicos, como son el caso de *HTML* y *Java*. Los detalles de la interpretación de un diseño, así como también, los problemas que han sido encontrados a la hora de interpretar dicha información, han sido detallados y explicados a modo de entender la solución que se ha encontrado para los mismos. En el informe también se detalla la definición de un lenguaje de templates, utilizado específicamente, para poder definir la representación de componentes y contenedores en un lenguaje dado. A través de la definición de un lenguaje de templates, hemos logrado unificar la forma de definir la representación de un componente gráfico, teniendo en cuenta que la representación de los mismos, es completamente distinta en lenguajes de programación como *HTML* y *Java*. Para demostrar la posibilidad de generar pantallas utilizando los diseños de *Balsamiq Mockups*, se ha desarrollado una aplicación que contempla la interpretación de dichos diseños y de los templates definidos, siendo estos dos, las entradas necesarias para iniciar el proceso de generación, que dará como resultado, el código fuente de la pantalla gráfica. Se ha definido la arquitectura de la aplicación y los detalles principales de su implementación, identificando que modificaciones pueden realizarse para extender la capacidad de la misma, habilitando la posibilidad entre otras cosas, de generar código fuente basado en otros lenguajes de programación. La aplicación de generación de código, cuenta con una interface gráfica y sencilla de utilizar que permitirá fácilmente seleccionar un diseño de *Balsamiq Mockups* y un template, para posteriormente iniciar el proceso de generación. A través de la implementación de la aplicación hemos demostrado que se puede generar código fuente, en diferentes lenguajes de programación, a partir de un diseño de pantalla realizado con la aplicación *Balsamiq Mockups*.

Bibliografía

- **Balsamiq Mockups:** http://balsamiq.com/products/mockups/
- **Midnight Coders:** http://www.themidnightcoders.com/develop-ria/balsamiqapp
- **Napkee:** http://www.napkee.com/
- **Balsamiq Mockups Reify:** http://blogs.balsamiq.com/product/2013/01/08/go-from-mockup-to-code-with-reify/

Anexo A

En el siguiente anexo, se definirán los detalles de la generación de pantallas que contengan componentes avanzados y para los cuales se requerirán librerías especificas. Si tenemos en cuenta un lenguaje como *HTML*, los componentes que podemos representar dependen pura y exclusivamente de la definición del *DOM* (*Document Object Model*), que especifica al *browser* utilizado, que componentes se pueden representar y como. Si en el caso de HTML se quisiera representar un componente especifico, fuera de la definición del *DOM*, no se podría ya que dicha representación no existiría. En este caso, se debería utilizar una librería externa que permita definir el componente, habilitando la posibilidad de incluirlo en la pantalla. A modo de ejemplo, pensemos en un componente de tipo calendario, el cual se encuentra presente entre todos los componentes que permite representa *Balsamiq Mockups*. El componente de tipo calendario no es un componente básico de *HTML*, es decir, nos podemos referir al mismo como un componente de tipo avanzado y para el cual necesitaremos una librería externa. Entre todos los proyectos utilizados hoy por hoy en el diseño de pantallas *HTML*, las librerías de *JQuery* son algunas de las más utilizadas. *JQuery* es una librería desarrollada con *Javascript* que permite, a través de la manipulación del *DOM*, utilizar componentes predefinidos y avanzados como el caso de calendarios, contenedores de tipo acordeón y tabs entre otros. En el caso de querer utilizar estas librerías, simplemente tendríamos que descargarlas e incluirlas en la sección de importación (tag *HEAD*) del documento *HTML* diseñado. En nuestro caso y teniendo en cuenta que el documento *HTML* se generará a partir de un template, deberíamos modificar dicho template para poder definir como se representa un componente de tipo calendario con *JQuery*. A continuación, en la *Figura 49* podemos observar la representación del componente calendario dentro de un template, la misma, no es distinta de la representación de otros componentes salvo por el agregado de un funcionalidad, en *Javascript*, que permite inicializar el componente calendario. Como se observa, se define el tipo y los atributos del mismo al igual que se comentó en la sección de representación de componentes de un template. En este caso particular, en la sección *innerCode* hemos agregado la inicialización del componente, es decir, la invocación de funcionalidades de *JQuery* que nos permitirán representar el calendario.

```
<component type="DateChooser" name="input">
      <attributes>
            <attribute name="id" value="$componentId" />
            <attribute name="name" value="$componentId" />
            <attribute name="type" value="text" />
            <attribute name="style" value="width: $width px; height:
$height px;" />
      </attributes>
      <innerCode>
            <![CDATA[
            <script>
            $(function() {
                  $( "#$componentId" ).datepicker({
                        showOn: "button",
                        buttonImage: "images/calendar.png",
                        buttonImageOnly: true
                  });
            });
            </script>
            ]]>
      </innerCode>
</component>
```

Figura 49: Representación de un calendario

Un calendario en *JQuery* se representará con un tag *input*, pero se invocará sobre el mismo a la función datepicker para inicializar el componente. Como observamos en la definición del tag *component*, el tipo de componente *DateChooser* (es decir un calendario) se escribirá como un tag *input*. En la sección *innerCode* hemos agregado la inicialización del componente invocando a la función previamente mencionada, en este caso, se hace uso de la palabra reservada *CDATA*, utilizada comúnmente en la interpretación de *xml*, para indicar al intérprete que esa sección del código no debe ser analizada. De esta manera podremos agregar la inicialización del componente sin ningún problema ya que la sección de código que invoca a *JQuery* no será analizada. La palabra reservada *$componentId* hará referencia al identificador del componente, en este caso el calendario, y si será reemplazada en la sección *innerCode* ya que dicho reemplazo será posterior a la interpretación del template. De esta manera, podremos tener todos los componentes de tipo calendario que queramos incluir en el diseño de la pantalla, dado que cada uno de ellos, se identificará de manera diferente junto con el resto de los componentes. Los atributos *showOn*, *buttonImage* y *buttonImageOnly*, son parte de la definición del componente calendario de *JQuery*, en el caso particular de *buttonImage*, donde se observa la definición de una imagen, debe tenerse en cuenta que el calendario solo se observará correctamente si se respeta el path definido en dicho atributo. De

esta manera, el código fuente generado deberá localizarse en un archivo ubicado en el mismo directorio donde se encuentra también la carpeta *images* y que contenga en su interior el archivo *calendar.png*. Lo mismo sucederá con las librerías de *JQuery*, las cuales, deberán incluirse en la sección *HEAD* del documento *HTML* generado. Para especificar las librerías en el template se deberá modificar la sección *body* del mismo, donde especificábamos el cuerpo principal del documento *HTML* generado y dentro del cual definíamos el *HEAD* y *BODY* del documento *HTML* final.

```
<link rel="stylesheet" href="css/themes/base/jquery.ui.all.css">
<script src="js/jquery-1.8.0.js"></script>
<script src="js/ui/jquery.ui.core.js"></script>
<script src="js/ui/jquery.ui.widget.js"></script>
<script src="js/ui/jquery.ui.datepicker.js"></script>
<link rel="stylesheet" href="css/demos.css">
```

Figura 50: Librería de *JQuery*

En la *Figura 50* podemos observa la definición de las librerías que se deberán agregar en el template para utilizar otros componentes avanzados, cabe destacar, que el archivo generado finalmente deberá se ubicado en un directorio dentro del cual también se encuentren dichas librerías, respetando siempre los *paths* de la importación. A continuación podemos observar el diseño de *Balsamiq Mockups* que incluye los calendarios.

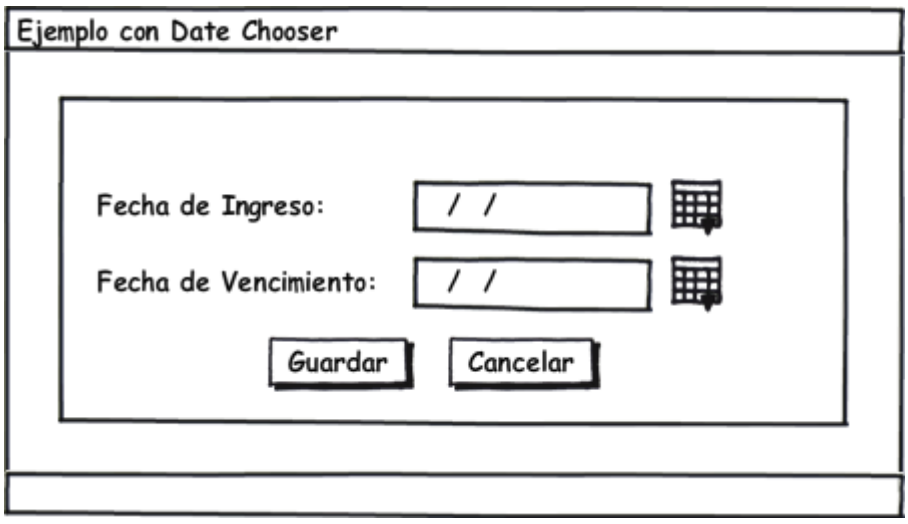

Figura 51: Diseño con calendarios

Como se puede observar en el diseño, se han incluido dos componentes de tipo calendario distribuidos en un contenedor, este diseño se representará de la siguiente manera con las modificaciones realizadas previamente sobre el template.

Figura 52: Pantalla generada con calendarios

La apariencia de la pantalla anterior se debe a los estilos que han sido definidos en el template utilizado, como se puede observar, los componentes de tipo calendario han sido representado por medio del template. Al presionar sobre la imagen, asociada al calendario, se podrá visualizar el mismo ofreciendo la posibilidad de seleccionar una fecha determinada, esto se debe a que la librería utilizada se comporta de dicha manera.

Con el ejemplo anterior hemos demostrado cómo es posible representar componentes avanzados en *HTML* y definiéndolos en un template. La posibilidad de representa otros componentes dependerá pura y exclusivamente de las librerías a utilizar y de la complejidad de las mismas, cabe destacar, que se deberán ubicar la librerías en el directorio utilizado para la generación, así como también, las referencias definidas en el template

deberán respetar la estructura de directorios correcta esperada por las librerías. Teniendo en cuenta que el template incluye una sección dentro de la cual se puede especificar la estructura de un documento *HTML*, se podrán incluir todas las librerías y estilos *CSS* que se deseen, habilitando la posibilidad de representar muchos otros componentes presentes en *Balsamiq Mockups*.

En el caso de los templates *Java*, la utilización de componentes avanzados dependerá, al igual que en el ejemplo anterior, de librerías de terceros. El template que hemos incluido en el desarrollo de la aplicación de generación, tiene la finalidad de generar pantallas en *Java Swing* por medio de los componentes presentes en dicho framework. *Java Swing*, es el framework que deberá utilizarse por defecto en el desarrollo de aplicaciones de escritorio basadas en *Java*, dicho framework, permitirá representar una amplia selección de componentes presentes en *Balsamiq Mockups*, generando en mucho casos, que no se tenga que depender de librerías de terceros. En el caso de que se desee utilizar otro framework o una librería especifica, simplemente se deberá descargar dicha librería e incluir las clases a utilizar de la misma en la sección de importación.

```
<main>
	<body>
		<![CDATA[
			import java.awt.Container;
			import java.awt.Insets;
			import javax.swing.JButton;
			import javax.swing.JLabel;


			public class SwingGeneratedExample {

			    public static void main (String [] args)
			    {
					$generatedCode
			    }
			}
		]]>
	</body>
</main >
```

Figura 53: Sección main de un template *Swing*

Como se puede observar en la figura anterior, en la sección *main* del template se incluirá el código que representa la estructura principal de la clase *Java* a generar. De esta

manera se podrán incluir todas las clases que se desean usar en la generación, dado que en la sección de imports, podremos importar cuantas clases sean necesarias. En el caso anterior, el template solo podrá representar componentes de tipo botón y etiquetas, dado que son solo los dos componentes que se importan. Al ejecutar el proceso de generación, la aplicación intentará compilar la clase, por lo tanto, las dependencias agregadas deberán agregarse al *classpath* de ejecución de la generación. Si las librerías de terceros que representan los componentes que se desean agregar, no están presentes en el *classpath*, la aplicación generará de todas formas el código fuente de la clase, pero el mismo no se compilará dado que no se pueden referenciar las clases necesarias para dicho proceso. Se debe tener en cuenta, que la compilación, tiene como única finalidad, poder mostrar la pantalla generada dentro de la aplicación al momento de ejecutar dicha generación, es por esa razón, que en template *Swing* de ejemplo, se incluye un método *main* para poder ejecutar la clase y mostrar la pantalla generada. Para agregar al template la definición de un componente de terceros, una vez agregada la librería, se deberá escribir la representación del mismo al igual que hemos demostrado en la sección de templates. Debe recordarse, que en un template Swing, los componentes se escriben indicando que clase se debe instanciar, cuales son los parámetros de inicialización (atributos recibidos por el constructor de la clase) y que otros métodos se deben ejecutar sobre la instancia para finalizar su inicialización.

Anexo B

En este anexo, describiremos la posibilidad de extender la aplicación para habilitar la posibilidad de generar código fuente con otro lenguaje de programación. Como se menciono reiteradas veces a lo largo del informe, la aplicación de generación permitirá generar código fuente en dos lenguajes: *HTML* y *Java*, pero uno de los puntos a tener en cuenta, en el diseño y desarrollo del generador, era lo posibilidad de extender el mismo para poder generar código en otros lenguajes. Cuando se describió el componente de generación, en el capítulo relacionado a la arquitectura de la aplicación, se mencionó a dos dependencias del mismo: *html.writer* y *swing.writer*. Como se mencionó, dichas dependencias están relacionadas con la generación y el lenguaje fuente seleccionado. La dependencia *html.writer*, permitirá al generador, poder escribir código fuente en formato *HTML* y la dependencia *swing.writer*, cumplirá la misma función con *Java*, más precisamente con el framework de *Java Swing*. Dichas dependencias, son clases Java que implementan el comportamiento necesario para escribir código fuente en un lenguaje determinado e implementan una interface que define los métodos que deberán estar definidos, de forma obligatoria, para que el generador las pueda utilizar. De esto se deduce, que la generación de código fuente en otro lenguaje, dependerá de la implementación de una clase *writer*. En la siguiente figura se muestra la interface *ComponentWriter* y sus métodos.

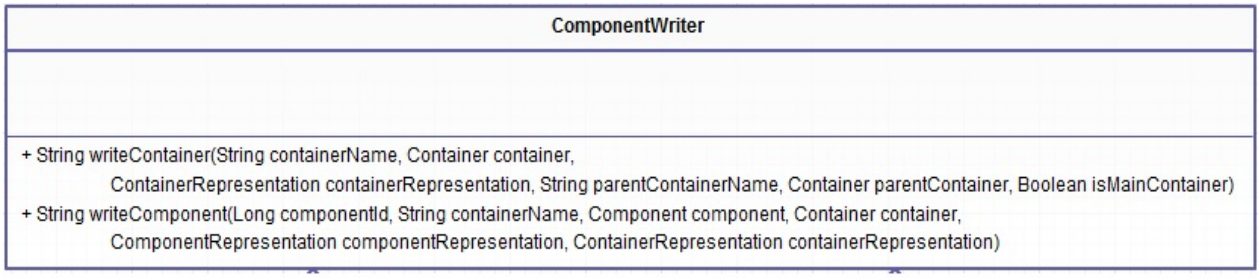

Figura 54: Interface *ComponentWriter*

Como se observa en la *Figura 54*, la interface *ComponentWriter* posee solo dos métodos, y por sus nombres, se puede deducir que relacionan con la codificación de componentes y contenedores. El método *writeContainer* deberá contener la lógica para poder escribir un contenedor en un lenguaje específico y deberá implementar también las distintas variantes posibles. Como se observa en dicho método, se recibe una gran cantidad de parámetros, esto se debe a que la implementación en cada lenguaje puede requerir de mayor o menor información a la hora de escribir el código fuente final. A continuación describiremos cada uno de los parámetros recibidos:

String containerName: Representa el nombre del contenedor, y podrá ser utilizado para dar nombre a una clase o identificar el contenedor.

Container container: Es una instancia de la clase *Container* y es producto de la interpretación del contenedor presente en el *mockup*. Esta instancia encapsulará una lista de componentes, permitiéndonos obtener todos los componentes que se ubican dentro del contenedor (inclusive otros contenedores). A su vez, dado que un *Container* extiende el comportamiento de un *Component*, podremos acceder a las características del mismo, como posicionamiento, tamaño y tipo de componente.

ContainerRepresentation containerRepresentation: En esta instancia tendremos encapsulada la información que define el template en relación al contenedor. Esto nos permitirá acceder a los atributos y métodos que se definieron en el template y que pertenecen a este tipo de contenedor. Se debe recordar que en un lenguaje orientado a objetos, los métodos definidos como tags, serán los métodos que se deberán invocar al inicializar el contenedor, por ejemplo: *setBorder* definirá el tipo de borde que tendrá el contenedor. Básicamente, en la instancia *containerRepresentation*, tendremos todo lo referente y necesario para entender como representar el contenedor.

String parentContainerName: Representa el nombre del contenedor padre y que contiene al contenedor actual. Esta información nos permitirá poder hacer invocaciones al contenedor padre, por ejemplo para agregar el contenedor actual al mismo y poder así construir una jerarquía.

Container parentContainer: Es similar al atributo container, pero es la instancia del contenedor padre. Esta instancia se incluye por cualquier validación o información que se necesite del contenedor padre.

Boolean isMainContainer: Determina si se trata del contenedor de mayor jerarquía. Esto se debe a que puede ser necesario identificar el contenedor principal para poder hacer un manejo diferenciado del mismo, por ejemplo: en el caso de *Java Swing* podremos agregar al contenedor principal los botones de maximización y minimización, ya que dicho comportamiento solo será parte del contenedor de mayor jerarquía, es decir, de la pantalla principal.

Como se puede apreciar, el método encargado de escribir el código fuente de un contenedor, recibirá todo lo necesario para definir como escribir el mismo. Se debe tener en cuenta, que el componente de generación, invocará a dicho método cada vez que tenga que escribir un contenedor, es por esa razón, que la implementación deberá contemplar la escritura de contenedores de alto nivel (el contenedor principal) y contendores de bajo nivel, es decir, distribuidos en el interior de la pantalla.

El método *writeComponent*, es muy similar al mencionado anteriormente, pero su implementación deberá contemplar la escritura de un componente. A continuación describiremos cada uno de los parámetros recibidos:

Integer componentId: Representa el id que se le a asignado al componente y puede ser utilizado para generar el nombre de una clase o el identificador de un tag.

String containerName: Representa el nombre asociado al contenedor que incluye al componente actual.

Component component: Es una instancia de la clase *Component* y es producto de la interpretación del control presente en el *mockup*. Esta instancia encapsulará todas las características asociadas al control en el *mockup*, permitiendo acceder a su tamaño posición u otras características.

Container container: Es la instancia de la clase *Container* que incluye al componente y es producto de la interpretación del contenedor presente en el *mockup*.

ComponentRepresentation componentRepresentation: En esta instancia tendremos encapsulada la información que define el template en relación al componente. Esto nos permitirá acceder a los atributos y métodos que se definieron en el template y que pertenecen a este tipo de control.

ContainerRepresentation containerRepresentation: En esta instancia tendremos encapsulada la información que define el template en relación al contenedor que incluye al componente. Esto nos permitirá acceder a los atributos y métodos que se definieron en el template y que pertenecen a este tipo de contenedor.

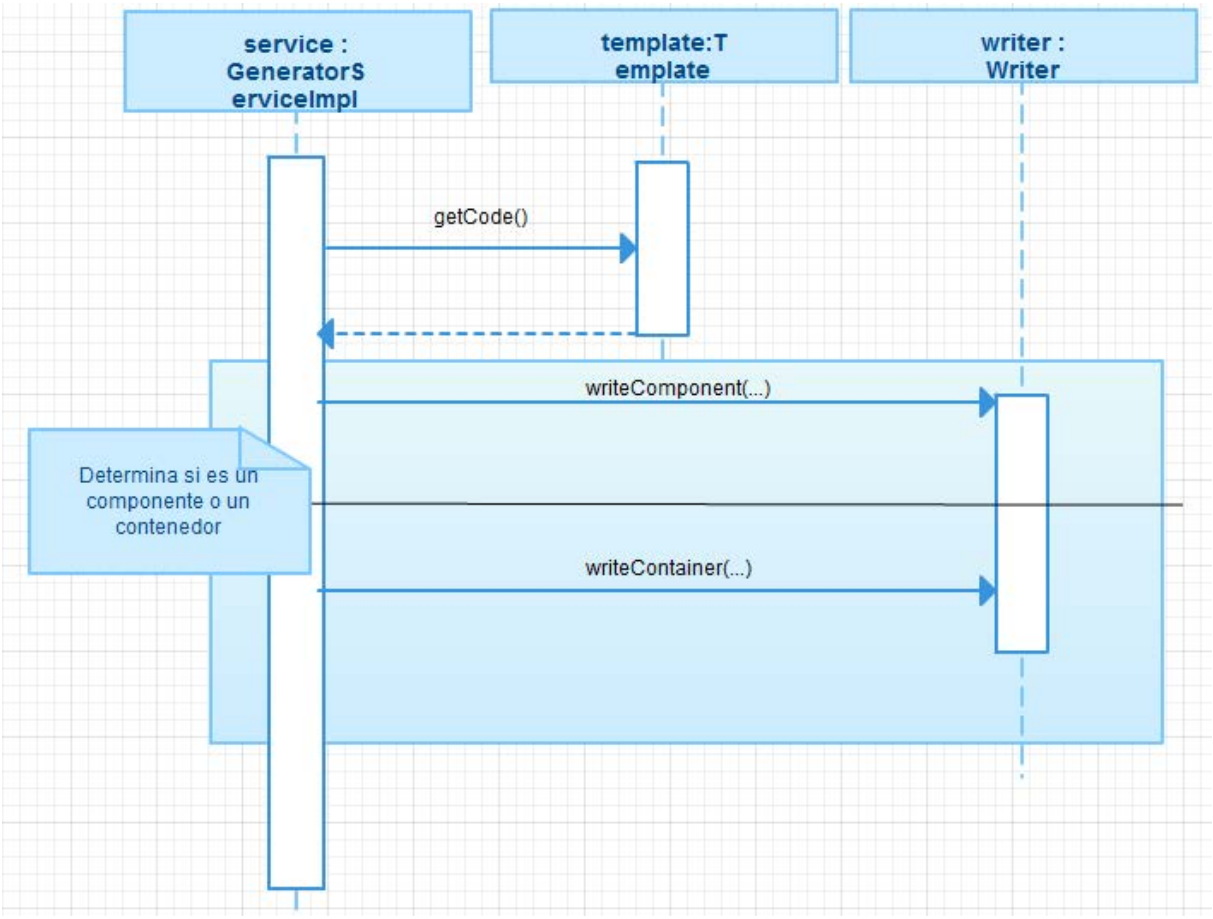

Figura 55: Invocación de una implementación de *ComponentWriter*

La implementación de un contenedor o de un componente tendrán grandes variantes en diferentes lenguajes, particularmente, si estos implementan paradigmas distintos, por esta razón ambos métodos, incluyen una serie de atributos que cubren gran parte de la información, posiblemente requerida, para la codificación de dichos componentes.

En el capítulo referente a la arquitectura del generado, desarrollamos la funcionalidad del servicio de generación. El servicio de generación será el encargado de

invocar los métodos, previamente definidos, a la hora de escribir un componente o un contenedor, es decir, recorrerá la estructura jerárquica de componentes y por cada uno de ellos, invocará al método correspondiente. En la *Figura 55*, se puede observar la secuencia de invocación de una implementación de *ComponentWriter* por parte del servicio de generación. En primer lugar se obtendrá del template el código que identifica el lenguaje y el cual nos permitirá determinar qué tipo de *writer* se debe utilizar. En el capítulo en el que se definió la estructura del *template*, se comentó la existencia del atributo *code*, el mismo será utilizado para identificar el tipo de *writer* que tiene asociado dicho *template*, permitiendo reutilizar el *writer* en distintas definiciones. Una vez que se determina el *writer* a utilizar, el servicio iterará la lista de componentes anidados, que forma la estructura de la pantalla, e invocará a los métodos de escritura correspondientes. A través de la información que contiene la instancia de *Component*, se podrá determinar si es un componente con comportamiento de contenedor y de esa forma saber si se tiene que invocar al método *writeComponent* o *writeContainer*. Se debe tener en cuenta, que aparte de agregar la implementación de *ComponentWriter* al proyecto, se deberá modificar el contexto de la aplicación para poder inyectar el nuevo *writer* al servicio de generación.

```
<bean id="generator.service"
class="ar.com.mockup.service.impl.GeneratorServiceImpl">
      <property name="htmlComponentWriter">
            <ref bean="html.writer" />
      </property>
      <property name="swingComponentWriter">
            <ref bean="swing.writer" />
      </property>
      <property name="newComponentWriter">
            <ref bean="new.writer" />
      </property>
</bean>
```

Figura 56: Inyección de un nuevo *writer* al servicio de generación

Como se observa en la figura anterior, simplemente se deberá agregar en el xml que define los servicios, la definición del nuevo *writer* como una propiedad del servicio de generación, definiendo también su implementación. De esta manera se podrán agregar nuevas implementaciones para la escritura de código en otros formatos o lenguajes de programación.