Trabajo Práctico 3
Programación Orientada a Objetos

Paradigmas de Lenguajes de Programación — 2do cuat. 2015

Fecha de entrega: 17 de noviembre

<HTMLBuilder/>

1. Introducción

En este trabajo se desea crear un generador de código HTML[1] en Smalltalk, que soporte un conjunto de tags básicos de HTML con sus propiedades, pero que deberá ser extensible para soportar nuevas definiciones.

1.1. Subconjunto del lenguaje HTML a utilizar

Al comienzo de cada documento deberá especificarse el tipo de documento. Se seguirá la recomendación HTML5, por lo se define que los documentos generados deberán comenzar con `<!DOCTYPE html>`.

El lenguaje HTML es un lenguaje de marcas (markups) que queda definido por una estructura de tipo árbol con elementos de diferentes clases (tags). En particular, para este trabajo se deberá soportar los siguientes tags HTML:

- `html`: abarca a todo el documento, es el único que puede contener `head` y `body`, y sólo puede tener uno de cada uno.
- `head`: que sólo puede contener a `title`
- `title`: que sólo puede contener texto plano
- `body`: que contiene a `table`, `p`, `b`, `div` y texto plano
- `table`: que sólo puede contener elementos de tipo `tr` y `th`
- `tr` y `th`: que sólo pueden contener elementos de tipo `td`
- `td`: puede tener el mismo contenido que `body`
- `div`: puede tener el mismo contenido que `body`
- `p`: sólo puede tener a `b` o texto plano, a efectos de este trabajo
- `b`: sólo puede tener texto plano dentro, a efectos de este trabajo

[1] HTML5. W3C Recommendation http://www.w3.org/TR/html5/

2. Implementación

La implementación propuesta consta de una clase *constructora* de documentos, que llamaremos `Html`. Esta clase, al recibir el mensaje `build:` con un bloque conveniente, evaluará el bloque e irá acumulando en un *stream* el documento resultante. Para esto, deberán definirse objetos para representar, de alguna manera, los distintos nodos del documento. Un ejemplo completo de comportamiento esperado es el siguiente:

2.1. Primera parte: diseño general y tags estáticos

Dada la definición de un documento a continuación,

```
Html build: [ :html |
  html body: [
    html h1: 'Titulo'.
    html p: 'Lorem ipsum dolor sit amet, consectetur adipisicing elit.'.

    html h2: 'Subtitulo'.

    html p: [
      html append: 'Duis aute irure dolor in reprehenderit in voluptate '.
      html b: 'velit esse cillum dolore'.
      html append: ' eu fugiat nulla pariatur.'
    ]
  ]
]
```

se debe generar el siguiente código html, desestimando saltos de líneas y espaciado:

```
<!DOCTYPE html>
<html>
  <body>
       <h1>Titulo</h1>
       <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit.</p>
       <h2>Subtitulo</h2>
       <p>Duis aute irure dolor in reprehenderit in voluptate
          <b>velit esse cillum dolore</b> eu fugiat nulla pariatur.
       </p>
  </body>
</html>
```

Ejercicio 1 – Nodos y Tags

Cada tipo de nodo posible en nuestro html está determinado por un *tag*. Los nodos deberán poder responder a `openTag:` y `closeTag:` colaborando con un *writeStream* de la manera esperada. Por ejemplo:

```
|ws node1 |
ws := WriteStream on: String new.
node1 := HtmlNode new.
node1 openTag: ws.
node1 closeTag: ws.
ws contents.  ''' => <html></html> '''
```

Observemos que cada nodo conoce un stream sobre el que escribe lo que corresponda. Sólo con `HtmlNode`, ya se puede producir el siguiente código:

```
Html build: [ :html |

  ] ''' => <!DOCTYPE html><html></html> '''
```

Deberán poder representarse todos los nodos correspondientes a los tags definidos en el subconjunto considerado del lenguaje.

En este y en los demás ejemplos a continuación se asume una clase `HtmlNode` con un constructor sin argumentos, pero ni el nombre ni la aridad del constructor son requisitos.

Sí se espera que se respecte la sintaxis relativa a `Html build: [ ... ]`. El resto de los ejercicios orientan el diseño en forma parcial para llegar a una solución abarcativa y completa.

Ejercicio 2 – Inclusión de texto plano

El mensaje `append:` deberá permitir agregar texto plano dentro de un nodo de la siguiente manera:

```
Html build: [ :html |
  html append: 'Lorem ipsum'
  ] ''' => <!DOCTYPE html><html>Lorem ipsum</html> '''
```

Normalmente se debería hacer *escaping* de las cadenas que van al `append:`, de forma de que no se introduzca ningún carácter que pudiera ser interpretado como HTML en el código resultante, pero se puede suponer que para este trabajo no será necesario hacerlo.

Ejercicio 3 – Nodos anidados

Por la propia definición del lenguaje HTML, la representación elegida tiene que soportar anidamiento de nodos.

Un primer enfoque será utilizar una pila, donde `pushChildNode:` agrega un nodo a la pila, de tal manera que todos los nodos que se inserten arriba de él serán sus hijos en el documento. Al desapilar los nodos (`popNode`) se produce el documento, como se ve en los dos ejemplos a continuación:

```
| builder |
builder := Html new.

builder pushChildNode: (HtmlNode new: builder).

builder pushChildNode: (HeadNode new: builder).
builder popNode.

builder pushChildNode: (BodyNode new: builder).
builder popNode.

builder popNode.

builder contents. ''' =>  <!DOCTYPE html><html><head></head><body></body></
    html> '''

| builder |
builder := Html new.

builder pushChildNode: (HtmlNode new: builder).
builder pushChildNode: (HeadNode new: builder).
```

```
builder popNode.

builder pushChildNode: (BodyNode new: builder).
builder pushChildNode: (H1Node new: builder).
builder popNode.
builder pushChildNode: (PNode new: builder).
builder popNode.
builder popNode.

builder popNode.

builder contents. ''' =>  <!DOCTYPE html><html><head></head><body><h1></h1
    ><p></p></body></html> '''
```

Para continuar, se deberán acondicionar los objetos definidos en el ejercicio anterior para que soporten anidamiento de tags de la siguiente manera:

```
Html build: [ :html |
  html body with: [
    html append: 'Lorem ipsum'
    ]
  ]. '''=> <!DOCTYPE html><html><body>Lorem ipsum</body></html>'''
```

Pista: Observemos que en el segundo caso, dado que ya está definido el mensaje `body`, el mensaje `with:` se envía al resultado de éste.
Pista: No nos interesa utilizar los nodos para tener el árbol sintáctico completo del documento que estamos generando. Simplemente nos interesa que se pueda ir generando el resultado en algún stream.

Finalmente, para simplificar la sintaxis, se deberán soportar las siguientes versiones:

```
Html build: [ :html |
  html head title append: 'Lorem ipsum'
]. ''' => <!DOCTYPE html><html><head><title>Lorem ipsum</title></head></
    html> '''

Html build: [ :html |
  html head title: 'Lorem ipsum'
]. ''' => <!DOCTYPE html><html><head><title>Lorem ipsum</title></head></
    html> '''
```

Ejercicio 4 – Validación

La capacidad de anidar nodos requiere detectar si la estructura no respeta las restricciones enunciadas a medida que se va generando el documento. En ese caso, se deberá informar un error y no producir el código html. Por ejemplo el siguiente documento no tiene una estructura válida:

```
<!DOCTYPE html><html><head><body>Lorem ipsum</body></head></html>'''
```

dado que `body` no puede anidarse dentro de `head`. En este sentido, se espera que el siguiente código informe el error y no produzca un documento.

```
Html build: [ :html |
  html head body: 'Lorem ipsum'
]
```

Pista: Utilizar la pila, según los ejemplos a continuación.

El siguiente código es válido y debería ejecutarse sin errores,

```
| builder |
builder := Html new.

builder pushChildNode: (HtmlNode new: builder).

builder pushChildNode: (HeadNode new: builder).
builder popNode.

builder pushChildNode: (BodyNode new: builder).
builder popNode.

builder popNode.

builder contents. ''' =>  <!DOCTYPE html><html><head></head><body></body></
    html> '''
```

mientras que el siguiente debería generar un error.

```
| builder |
builder := Html new.

builder pushChildNode: (HtmlNode new: builder).
builder pushChildNode: (HeadNode new: builder).
builder pushChildNode: (BodyNode new: builder).
```

2.2. Segunda parte: tags dinámicos

Teniendo en cuenta que el lenguaje HTML está en plena actualización y que sólo tuvimos en cuenta un subconjunto de toda la recomendación, se desea que el generador de HTML pueda soportar extensiones sin necesidad de tener que ser reprogramado. Es decir, deben poder agregarse tags nuevos, con sus correspondientes restricciones. Así, inclusive nuestra implementación base del lenguaje HTML debería poder definirse a partir de un builder *vacío*, por medio de la siguiente secuencia de mensajes:

```
|contentNodes|
contentNodes:= #(H1 P B).
Html defNode: #Html canHave: #(Head Body).
Html defNode: #Head canHave: #(Title).
Html defNode: #Title.
Html defNode: #H1.
Html defNode: #B.
Html defNode: #Body canHave: contentNodes.
Html defNode: #P canHave: contentNodes.
```

Observemos la utilización de los mensajes `defNode:canHave:` y `defNode:` para definir nodos con y sin restricciones, respectivamente.

Pautas de entrega

El entregable debe contener:

- un archivo `.st` con todas las clases implementadas
- versión impresa (legible) del código, comentado adecuadamente
- **NO** hace falta entregar un informe sobre el trabajo

Se espera que el diseño presentado tenga en cuenta los siguientes factores:

- definición adecuada de clases y subclases, con responsabilidades bien distribuidas
- uso de polimorfismo para evitar exceso de condicionales
- intento de eliminar código repetido utilizando las abstracciones que correspondan

Consulten todo lo que sea necesario.

Consejos y sugerencias generales

- Lean al menos el primer capítulo de *Pharo by example*, en donde se hace una presentación del entorno de desarrollo.
- Explorar la imagen de Pharo suele ser la mejor forma de encontrar lo que uno quiere hacer. En particular tengan en cuenta el buscador (`shift+enter`) para ubicar tanto métodos como clases.
- No se pueden modificar los test entregados, si los hubiere, aunque los instamos a definir todos los tests propios que crean convenientes.

Importación y exportación de paquetes

En Pharo se puede importar un paquete arrastrando el archivo del paquete hacia el intérprete y seleccionando la opción "`FileIn entire file`". Otra forma de hacerlo es desde el "`File Browser`" (botón derecho en el intérprete > `Tools` > `File Browser`, buscar el directorio, botón derecho en el nombre del archivo y elegir "`FileIn entire file`").

Para exportar un paquete, abrir el "`System Browser`", seleccionar el paquete deseado en el primer panel, hacer click con el botón derecho y elegir la opción "`FileOut`". El paquete exportado se guardará en el directorio `Contents/Resources` de la instalación de Pharo (o en donde esté la imágen actualmente en uso).