DSL-2-Browser: Un ejemplo de ejecución de un lenguaje específico del dominio en un navegador web

Álvaro Gutiérrez-Pérez, Luis-María García-Rodríguez, Rober Morales-Chaparro, and Fernando Sánchez-Figueroa

Quercus Software Engineering Group, Universidad de Extremadura
agutierrez|luismaex|robermorales|fernando@unex.es

Resumen. En este trabajo se expone, mediante un ejemplo, la viabilidad de ejecutar un DSL en un navegador web. Para ello se ha usado principalmente Xtext y GWT sobre el caso concreto de un DSL de visualización de datos. Aunque la propuesta se realiza a través de un ejemplo concreto, es posible su generalización para otros DSL.

Palabras clave: Desarrollo de software dirigido por modelos, Ingeniería Web, Lenguajes específicos del dominio, JavaScript

1. Introducción

Los lenguajes específicos del dominio (DSL) [1] surgen para acercar el desarrollo de software a los expertos del dominio. Paralelamente, la web se ha ido imponiendo como plataforma para la visualización de contenido y ejecución de código ante la creciente variedad de dispositivos con acceso a internet. Por todo ello, sería deseable llevar la potencia de los DSL a la omnipresencia de la web, es decir, poder describir mediante un DSL el código que se ejecuta en un navegador web.

En este documento se presenta un proceso para generar código JavaScript ejecutable en un navegador web a partir de un lenguaje específico de dominio. La propuesta se realiza a través de un ejemplo concreto, visualligence [2] un DSL para la visualización de datos. Este DSL permite al usuario crear su propia visualización de los datos de que dispone según la representación gráfica que desee, por tanto es un campo de aplicación que se puede beneficiar de la propuesta.

Partiendo de un determinado modelo en base a un DSL, mediante Xtext y la ayuda de una librería específica del dominio, se obtiene un modelo correspondiente en Java que puede ejecutarse correctamente en la máquina virtual como una aplicación independiente. Tras un proceso de adaptación de la librería, usando GWT se obtiene una descripción equivalente que puede ser ejecutada en un navegador usando únicamente tecnologías estándares. Respecto al ejemplo concreto que se va a detallar, actualmente el DSL se ejecuta convenientemente en la máquina virtual de Java, y la intención de este artículo es poder ejecutarlo en un navegador web mediante JavaScript. En la Fig. 1. se resumen estos pasos, que componen la propuesta presentada.

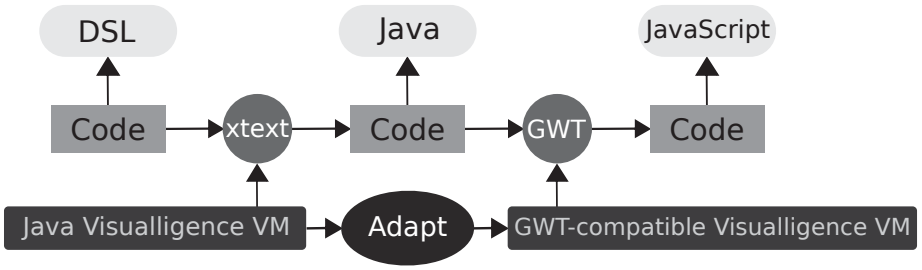

Figura 1. Vista general de la propuesta presentada.

El artículo se estructura de la siguiente forma: la Sección 2 presenta tecnologías existentes relacionadas con el problema a resolver. La Sección 3 presenta el caso de estudio: hilo conductor del desarrollo posterior, en la Sección 4. Por último, la Sección 5 contiene las conclusiones obtenidas junto con trabajos futuros que surgen a partir de nuestra propuesta.

2. Tecnologías y trabajos relacionados

A continuación se describen propuestas que se usan actualmente con un objetivo cercano al nuestro.

Xtext[1] es una plataforma para desarrollar DSLs basada en Eclipse. Usa una sintaxis extendida para la descripción de la gramática y genera automáticamente un modelo de clases de la gramática. Además, se puede lanzar un editor para escribir código basado en ese DSL.

Google Web Toolkit (GWT)[2] proporciona herramientas para el desarrollo de aplicaciones web. Permite desarrollar tanto el código del cliente como el del servidor en la misma tecnología (Java). Posteriormente, el código del cliente se "compila" a JavaScript y es ejecutado en el navegador.

CoffeeScript[3] es un lenguaje de programación que genera código JavaScript. Su principal atractivo es la simpleza comparado con JavaScript, permitiendo ahorrar gran cantidad de texto. **Source Maps**[4] es un estándar que permite relacionar unos ficheros de código con otros. Es usado para hacer corresponder código JavaScript ofuscado o generado con su correspondiente versión fuente. Así, a la hora de depurar, se muestra en el visor del navegador el código original en lugar de la versión que se está ejecutando.

asm.js[5] es un subconjunto de JavaScript muy optimizable, diseñado para ser muy eficiente. Existen compiladores de otros lenguajes a *asm.js* que permiten, por ejemplo, desarrollar un programa en C/C++ y ejecutar el código *asm.js*

[1] `http://www.eclipse.org/Xtext`
[2] `http://www.gwtproject.org/`
[3] `http://coffeescript.org/`
[4] `http://www.html5rocks.com/en/tutorials/developertools/sourcemaps/`
[5] `http://asmjs.org/`

correspondiente en un navegador. **Native Client**[6] es un proyecto para permitir la ejecución de código nativo en un navegador dentro de un entorno controlado e independiente de la plataforma.

El reproductor **Flash** de Adobe permite reproducir contenido dinámico de forma homogénea en distintas plataformas. **WebCL**[7] propone un estándar para llevar la computación paralela al navegador, aprovechando los distintos núcleos y capacidades gráficas del hardware.

Estos proyectos aportan muchas soluciones, cada uno en su ámbito, pero se ven limitados en nuestro propósito de facilitar una comunicación más directa y formal con el cliente mediante un DSL. O bien proponen lenguajes específicos pero ligados a una tecnología concreta, que no permiten su uso directo en la web.

Como se ha visto, hay propuestas tanto para crear lenguajes específicos del dominio como para dotar a la web de nuevos lenguajes de programación. Sin embargo, en nuestro conocimiento, no existe ninguna que permita llevar a la web los lenguajes específicos del dominio, simplificando el desarrollo de código y facilitando la comunicación con el cliente. En la Tabla 1. hay un resumen de las tecnologías mencionadas y la evaluación de algunas características relevantes en nuestro trabajo.

Tabla 1. Características relevantes de las tecnologías similares a nuestra propuesta

	Ejecutable en el navegador	(Sin plugins)	Independiente de la gramática	Código fuente trazable
Xtext			✓	✓
GWT	✓	✓		✓
CoffeeScript	✓	✓		
Source Maps			✓	✓
asm.js	✓	✓		
Native Client	✓	✓	✓	
Flash	✓			✓
WebCL	✓	✓		✓
dsl-2-browser	✓	✓	✓	✓

Existen propuestas académicas [3] [4] relacionadas con DSLs y la web, que abordan problemas similares pero no el mismo.

3. Caso de estudio

La propuesta se presentará a través de un ejemplo, para ayudar a entender más fácilmente la utilidad de lo que se expone. El ejemplo que se usará consiste en un DSL diseñado para crear visualizaciones de datos para la web. Cada vez, más y más datos están disponibles en la Web, de fuentes cada vez más diversas.

[6] `https://developers.google.com/native-client/dev/`
[7] `http://www.khronos.org/webcl/`

En este contexto, visualligence [2] surge como una propuesta dirigida por modelos para desarrollar visualizaciones dirigidas por datos, multi-dispositivo, adaptativas, y elegidas según el perfil del usuario final. El núcleo de visualligence es un DSL pensado para usuarios no técnicos, con el que el dueño de los datos puede modificar, ajustar y desarrollar patrones visuales, así como intercambiarlos entre sí. Actualmente el DSL se ejecuta en la máquina virtual de Java, y la intención de este artículo es ejecutarlo en un navegador web mediante JavaScript.

El Listado 1. muestra un ejemplo de patrón de visualización descrito con el DSL de visualligence en el que se asocia un punto a cada dato.

```
1 loop dots for people { | person, i |
2     output <- Circle {
3         @x <- person.age
4         @y <- person.weight
5         @height <- 0.1
6         @width <- 0.1
7     }
8 }
```

Listado 1. Ejemplo de patrón visual que asocia un punto a cada dato

4. Propuesta

En esta sección se describe el procedimiento para partir de un DSL y obtener código ejecutable en un navegador. Primero se detallan una serie de pruebas que se han realizado para comprobar la viabilidad del proyecto, y posteriormente se aplica el mismo procedimiento a visualligence.

4.1. Experimentación

En este apartado se introduce el funcionamiento de GWT a través de las pruebas realizadas, mostrando los problemas que presentan y cómo se han resuelto.

Un proyecto de GWT consta de varias partes. Por un lado el código del cliente que es posteriormente compilado a JavaScript, y por otro el código del servidor. Además del código, el proyecto también necesita un fichero con extensión `.gwt.xml` que contiene información necesaria para el compilador de JavaScript, entre ella la clase principal o *punto de entrada* de la ejecución en el navegador. Para el propósito de esta propuesta el código del servidor no es relevante, así que nos centraremos en la parte del cliente, que es la que se ejecuta en el navegador.

Para las primeras pruebas se ha usado un DSL muy sencillo basado en un ejemplo de Xtext, con una gramática que contiene saludos. En el Listado 2. se puede ver la descripción en Xtext de esta gramática. La implementación se realiza creando un modelo de la máquina virtual (*jvmmodel*) para las clases

Java y usando el generador de Xtext para los ficheros adicionales de GWT. Se comprueba que este esquema funciona correctamente generando un editor de Xtext para esa gramática, escribiendo un código de ejemplo y observando que la generación es correcta y se visualiza el resultado en el navegador.

```
1 PackageDeclaration:
2   'package' name=QualifiedName model=Model;
3 Model:
4   'Greetings' 'from' name=ID ':'? greetings+=Greeting*;
5 Greeting:
6   'Hello' name=ID surnames+=ID* '!'?;
```

Listado 2. Gramática de ejemplo usada durante las primeras pruebas.

El siguiente paso es poner a prueba los límites de GWT. Primero se hace incluyendo librerías externas precompiladas, en forma de archivo *jar*. Sin embargo, se comprueba que GWT necesita el código fuente de todo el código que se incluye, puesto que la transformación se realiza directamente sobre éste.

A continuación, se incluye en el proyecto el código fuente de una librería externa de cierto tamaño con el fin de determinar el nivel de soporte de GWT de la librería estándar de Java, y se intenta compilar. Dado que hay clases no soportadas por el compilador de GWT[8], al encontrarlas genera un mensaje de error y se detiene el proceso sin llegar a generar código.

Estas importantes limitaciones de GWT se deben tener en cuenta posteriormente a la hora de diseñar la librería del DSL que permita su ejecución en el navegador. Es decir, se debe disponer del código fuente de todas las librerías externas que se vayan a incluir y éstas deben usar únicamente las clases soportadas para poder usarlas sin problemas.

4.2. Visualligence

Una vez determinadas las posibilidades y limitaciones de GWT, habiendo constatado que es posible llevar a cabo la transformación, se procede a trabajar sobre el código de un proyecto real, visualligence. Se debe realizar una adaptación del código actual a uno compatible con GWT que cumpla sus limitaciones.

El principal problema se encuentra en las clases del paquete de entrada y salida `java.io`, dado que la mayor parte de ellas carecen de soporte en GWT. Esto limita enormemente la capacidad para trabajar con flujos de entrada y salida, un pilar fundamental en visualligence. Por tanto, se ha desarrollado una jerarquía de entrada y salida paralela a la proporcionada por la librería estándar de Java, pero sin usar referencias a ellas. También tuvieron que sustituirse o reemplazarse por implementaciones propias otras clases no soportadas por GWT, entre ellas `Scanner`, del paquete `java.util`.

[8] `http://www.gwtproject.org/doc/latest/RefJreEmulation.html`

Una vez resueltos estos problemas, se realiza la compilación a JavaScript sin problemas, y el código generado es ejecutado satisfactoriamente en el navegador.

En la Fig. 2. se puede ver el resultado: en el recuadro de la izquierda están los datos de entrada que recibe la aplicación. En el recuadro central se muestra la salida que produce la ejecución del patrón mostrado en el Listado 1. con los datos de entrada anteriores. Por último, en el recuadro de la derecha se muestra el resultado final tras ser interpretada la salida anterior por el navegador.

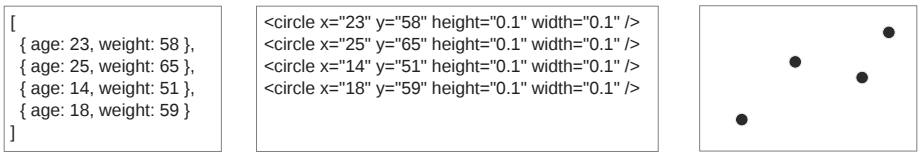

Figura 2. Datos de entrada, salida de visualligence, visualización en el navegador.

5. Conclusiones y trabajos futuros

Esta propuesta se ha descrito a través de un ejemplo realizado con una tecnología concreta. Esto podría dificultar la generalización del proceso para otras tecnologías. Sin embargo, esto no es problema, puesto que para otras tecnologías existen otras herramientas (como se ha visto en los antecedentes) que podrían usarse de forma similar a la descrita aquí.

En el futuro usaremos el mismo procedimiento con otro DSL, comprobando hasta qué punto el mismo mecanismo es reutilizable para distintos lenguajes. También sería necesario realizar una comparativa de rendimiento usando un ejemplo concreto en el DSL del caso de estudio, comparando una ejecución nativa con su correspondiente en el navegador. Por último, automatizar los distintos procedimientos, que ahora se ejecutan manualmente, producirá una transformación más sencilla y generalizable, posibilitando su uso en otros ámbitos.

Referencias

1. Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.
2. Rober Morales-Chaparro, Juan Carlos Preciado, and Fernando Sánchez-Figueroa. Visualizing search results: Engineering visual patterns development for the web. In Stefano Ceri and Marco Brambilla, editors, *Search Computing*, volume 7538 of *Lecture Notes in Computer Science*, pages 127–142. Springer Berlin Heidelberg, 2012.
3. Juan José Cadavid, Juan Bernardo Quintero, David Esteban Lopez, and Jesus Andrés Hincapié. A domain specific language to generate web applications. In *CIbSE*, pages 139–144, 2009.
4. Eelco Visser. Webdsl: A case study in domain-specific language engineering. In *Generative and Transformational Techniques in Software Engineering II*, pages 291–373. Springer, 2008.