Uma Ferramenta para Conversão de Código JavaScript Orientado a Objetos em ECMA 5 para ECMA 6

Daniel V. S. Cruz[1], **Marco Tulio Valente**[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte – MG – Brasil

{danielvsc,mtov}@dcc.ufmg.br

Abstract. *This paper presents a tool to convert source code in JavaScript that follows the object oriented Programming paradigm on ECMAScript 5 to the newest version: ECMAScript 6. The conversion is focused on the new syntax used to define classes instead of using the simulation. We also describe the use of the proposed tool in two popular JavaScript systems, when we were successfully able to convert them to the new ECMAScript 6 syntax.*

Resumo. *Este artigo apresenta uma ferramenta para converter códigos-fonte em JavaScript que adotam o paradigma de Programação Orientada a Objetos em ECMAScript 5 para a nova versão: ECMAScript 6. A conversão é focada na nova sintaxe utilizada para definir classes ao invés de utilizar a simulação. Descrevemos também o uso da ferramenta proposta em dois sistemas JavaScript populares, quando estávamos com sucesso capaz de convertê-los para a nova sintaxe ECMAScript 6.*

1. Introdução

JavaScript é uma linguagem muito conhecida e cada vez mais popular. Ela representa, por exemplo, 15% dos repositórios do GitHub[1]. Inicialmente concebida para ser uma linguagem de script voltada apenas para dar dinamismo à páginas web, a proporção da sua utilização se estendeu e atingiu outros patamares. Além de não possuir concorrentes diretos nesse segmento, através de aplicações como o Node.js[2] , frameworks como Angular.js[3] e conceitos como JavaScript Isomórfico, a linguagem passou a ser utilizada no desenvolvimento de uma ampla gama de sistemas, extrapolando o ecossistema web.

Desde sua criação, programas em JavaScript têm sido implementados seguindo diversos paradigmas, como por exemplo os paradigmas de programação imperativa e funcional. Além destes, o paradigma da programação orientada a objetos é bastante utilizado [L.Silva et al. 2015b], usando princípios de prototipagem [Arnström et al. 2016]. Este conceito de prototipagem define uma abordagem diferente da orientação a objetos tradicionalmente utilizada em linguagens como Java ou C#. Adicionalmente, existe um problema sintático, dado que não existem em JavaScript palavras-chave comumente utilizadas, como por exemplo, **class**. Essas diferenças podem acabar causando dificuldade na implementação do código orientado a objetos.

[1]http://githut.info/
[2]https://nodejs.org/
[3]https://angularjs.org/

Essa divergência sintática foi alvo das atualizações da ECMAScript, que em sua nova versão, ES 6 (ECMAScript 6) [ecm 2015] introduz uma nova sintaxe para classes. A proposta é manter o funcionamento baseado em protótipos [Blaschek 2012] , mas com uma forma de implementação mais conveniente e sintaticamente próxima dos conceitos tradicionais de orientação a objetos. Para que os desenvolvedores possam usufruir dos benefícios desta nova versão, este trabalho apresenta uma ferramenta, intitulada *ECMA Class Parser*, para migração de classes de código JavaScript ES 5 (ECMAScript 5) [ecm 2011] para nova sintaxe proposta de ES 6. Com isso, espera-se ajudar os milhares de desenvolvedores que possuem código que emula classes de acordo com a sintaxe antiga a se beneficiarem de forma automática da sintaxe nativa de classes proposta pela ES 6.

O restante deste artigo está organizado como descrito a seguir. A Seção 2 introduz as diferenças entre a sintaxe de ES 5 e ES 6, focando nos conceitos de orientação a objetos. A Seção 3 descreve as regras propostas no trabalho para conversão de código JavaScript ES 5 para ES 6. A Seção 4 apresenta os resultados das conversões, quando aplicados em 2 sistemas JavaScript populares. A Seção 5 discute trabalhos relacionados, enquanto a Seção 6 conclui o artigo e apresenta sugestões de trabalhos futuros.

2. Background

Esta seção apresenta os conceitos de orientação a objetos suportados por código JavaScript ES 5 e ES 6.

2.1. Definição de Classes

Em ES 6, as palavras-reservadas *class* e *constructor* foram adicionadas permitindo a declaração de classes de uma maneira similar a outras linguagens, como Java. A palavra-reservada *class* substitui o uso da palavra-reservada *function* e os parâmetros desta, que passam a pertencer ao construtor da classe. Por exemplo, o Código 1 apresenta a declaração de uma classe em ES 5, de nome *Foo*, atributo *attr* e método *action*. O código 2 apresenta a mesma classe, porém implementada na versão ES 6. A ferramenta proposta neste trabalho visa converter a classe apresentada no Código 1, para a nova sintaxe de classes, conforme ilustrado no Código 2.

```
1 //Classe Foo declarada como function
2 function Foo(attr){
3     //Atributos
4     this.attr = attr;
5     //Metodos
6     function Action(){}
7 }
```

Código 1. Exemplo de Classe ES 5

```
1 //Classe Foo com o uso da nova palavra-chave
2 class Foo {
3     //Construtor da Classe
4     constructor(attr) {
5         //Atributos
6         this.attr = attr;
7         this.Action();
8     }
```

```
9       //Metodos
10      Action(){}
11  }
```

Código 2. Nova sintaxe para declaração de Classes - ES 6

Adicionalmente, em ES 5, existem variantes de declaração dos métodos, os quais podem ser declarados via protótipo, ao invés de implementados internamente em uma função. O código 3 apresenta um exemplo:

```
1  Foo.prototype.Action = function(){};
```

Código 3. Método adicionado via protótipo

2.2. Herança

Em JavaScript, para herdar de uma classe, é necessário realizar uma cópia do protótipo de um objeto da classe base (conforme mostrado na Linha 9 do Código 4) e realizar sua associação ao protótipo da classe filha (Linha 10).

```
1  //Classe base
2  function Foo(attr){
3      this.attr = attr;
4  }
5  //Classe filha
6  function Bar(){}
7
8  //Copia e Associacao dos prototipos
9  Bar.prototype = Object.create(Foo.prototype);
10 Bar.prototype.constructor = Foo;
```

Código 4. Herança na sintaxe de ES 5

Por outro lado, a nova sintaxe de ES 6 inclui a palavra-reservada *extends* que realiza o mesmo processo de maneira mais intuitiva, conforme mostrado no Código 5. Neste código, a classe *Bar* é declarada como uma subclasse de *Foo* (Linha 8).

```
1  //Classe  base alvo
2  class Foo{
3      constructor(attr) {
4          this.attr = attr;
5      }
6  }
7  //Bar herda de Foo
8  class Bar extends Foo{
9  }
```

Código 5. Herança em ES 6

2.3. Acesso a Classes Base

Em ES 5, o acesso a classe base ocorre por meio de uma chamada ao construtor da classe base em um objeto filho (conforme mostrado na Linha 5 do Código 6). Esse acesso é realizado através de uma invocação direta de função, usando o método *call*.

```
1
2  //Classe Bar
```

```
3 function Bar(attr){
4     //Chamada ao construtor de Foo com o argumento attr
5     Foo.call(this, attr);
6 }
```

Código 6. Acesso a classe Base em ES 5

Em ES 6, a adição da palavra-reservada *super* permite realizar o mesmo acesso de maneira mais simples, pois encapsula a chamada à função e requer que apenas sejam passados parâmetros, conforme mostrado no Código 7.

```
1     //Acesso a classe base com o argumento attr
2     super(attr);
```

Código 7. Acesso à classe Base em ES 6

3. Regras de Conversão

A ferramenta, para o processo de conversão de ES 5 para ES 6, inicialmente realiza uma classificação de todas as funções de um programa ES 5 em três categorias: Classe, Método ou Independente. Essas categorias são definidas nas subseções seguintes.

3.1. Classe

Uma função é identificada como classe se não for anônima, isto é, se possui um identificador, e se não for implementada no escopo de outra função não anônima. Além disso, é necessário que uma função usada para definir classes seja usada como parâmetro do operador *new*, como ilustra o Código 8. Alternativamente, a função deve ter um método adicionado em seu protótipo, como no Código 9.

```
1     var foo = new Foo();
```

Código 8. Criação de um objeto da classe Foo

```
1 Foo.prototype.Action = function(){};
```

Código 9. Adição do método Action à classe Foo

3.2. Método

Uma função é identificada como sendo um método se for associada a uma classe. Essa associação pode ocorrer via protótipo de uma função já identificada como possível classe, como mostrado no Código 10 (Linha 3). Nesse exemplo, *Bar* é classificada como um método da classe *Foo*

```
1     function Bar(){};
2     //Metodo Bar da Classe Foo
3     Foo.prototype.Bar = Bar;
```

Código 10. Método Bar adicionado à classe Foo

Adicionalmente, essa associação pode ocorrer implementando a função a ser classificada como um método internamente em uma função já classificada como possível classe, como mostrado no Código 11.

```
1     function Foo(){
2        function Bar(){};
3     }
```

Código 11. Método bar declarado internamente à classe Foo

3.3. Geração do Código ES 6

Para iniciar a conversão do código para ES 6, inicialmente, obtém-se a AST (*Abstract Syntax Tree*) através do *parser* esprima.js[4] . Em seguida, a AST é percorrida buscando por três tipos de estruturas sintáticas:

1. Declaração de Funções: Criação de funções com nome definido e sem associação, como por exemplo: *function Foo(){}*
2. Declaração de Expressões: Chamadas de funções, criação de objetos, estruturas de repetição e todo o restante de expressões. Um exemplo fundamental é a estrutura da implementação de protótipos: *Foo.prototype.Do = function(action){}*
3. Declaração de Variáveis: Declaração de variavéis que podem estar sendo atríbuidas a uma função, para criação de uma classe, como por exemplo: *var Foo = function(){}*

A seguir, descrevem-se as transformações realizadas sobre cada um dos elementos mencionados.

3.3.1. Declaração de Funções

As estruturas desse tipo são as primeiras a serem verificadas, com o objetivo de criar uma estrutura de dados chamada MIC (Matriz de Identificação de Classes). Essa matriz armazena as possíveis classes que poderão ser alvo do processo de conversão. Cada declaração de função é testada para as três categorias: Classe, Método e Independente, em ordem de verificação, respectivamente. Os testes de classificação de uma função são baseados em duas propriedades: Identificador e Escopo.

Toda função é inicialmente validada quanto a seu identificador. Em sua ausência, a função em questão já é descartada como Classe, conforme a primeira regra de conversão da ferramenta. Na presença de um identificador, o escopo da função é verificado conforme a segunda regra de conversão: se a função for interna a uma função que possui identificador, ela é um método. Esta verificação exclui a opção de classificação dessa função como classe, pois em ES 6 não há suporte para aninhamento de classes. Uma função que satisfaz as duas condições é marcada como Classe em potencial. A confirmação final ocorre após a ferramenta percorrer toda a AST. As funções que não forem confirmadas como Classe são automaticamente classificadas como independentes e descartadas.

3.3.2. Delaração de Variáveis

Para complementar a construção da MIC, as declarações de variáveis são verificadas, pois podem implicar na criação de uma classe através da palavra-reservada *var*. Uma declaração de variável consiste em um identificador e um inicializador (como mostra a Linha 1 do Código 12), sendo este último, facultativo:

```
1    var foo = 'init';
```

Código 12. Declaração de Variável

[4]http://esprima.org/

Variáveis sem inicializador são descartadas, enquanto as demais tem o tipo do seu inicializador verificado. Caso o inicializador possua a estrutura de uma declaração de função, ele é enviado para o processo anterior (descrito na seção 3.3.1), para que a função seja avaliada e classificada como Classe, Método ou independente.

3.3.3. Declaração de Expressões

Após a construção da MIC, realiza-se uma busca por expressões que validem os candidatos a classe. Para isso, consideram-se dois tipos fundamentais de estrutura: *Assignment Expression* e *Call Expression*. A primeira é verificada quando possui composição pela palavra-reservada *prototype*. O termo à esquerda desta palavra-reservada representa o identificador da classe, enquanto o termo à direita, designa um método. A segunda expressão, *Call Expression*, é usada para detectar o uso de herança, pois, quando utilizada na associação do protótipo de uma função (como mostrado na Linha 1 do Código 13), pode estar definindo a cópia do protótipo da Classe Base. Por exemplo, no código 14, ambas as classes, *Bar* e *Foo* são atualizadas na MIC, pois *Foo* é confirmada como Classe Base, enquanto *Bar* é confirmada como classe que herda de *Foo*.

```
1    Bar.prototype = Object.Create(Foo.prototype);
2    Bar.prototype.constructor = Foo;
```

Código 13. Confirmação de Classe - Herança por Prototipagem

```
1    function Bar(){
2        Foo.call(this);
3    }
```

Código 14. Confirmação de Classe - Acesso à classe Base

Uma *Call Expression* também é verificada de maneira independente. Se uma função identificada como Classe em potencial possuir uma *Call Expresion*, onde o termo à direita é o método *call*, ela é confirmada na MIC pois representa um acesso a uma classe Base. O termo a esquerda é o identificador de sua classe base. Por exemplo, no código 14, *Bar* e *Foo* são confirmados como sendo classes, sendo *Bar* uma subclasse de *Foo*.

Ao final do processo, é realizada a formatação do código-fonte convertido. Esta formatação é realizada utilizando a ferramenta js-beautify[5].

4. Resultados

Após testes com sistemas triviais, foram realizadas conversões em dois sistemas reais do GitHub. O primeiro, 2048.js, é um jogo implementado em JavaScript. Com o uso da ferramenta proposta, foram convertidas 10 classes e 56 métodos. Foram realizados testes para verificar se houve alteração no comportamento do jogo, não sendo detectada nenhuma falha. O código convertido está disponível em: `https://github.com/DVSCross/2048`.

No segundo sistema, Isomery.js , uma biblioteca gráfica foram convertidas 7 classes e 47 métodos. Para verificação do comportamento do sistema, foi recriada uma área de

[5]https://www.npmjs.com/package/js-beautify

utilização da biblioteca[6] com o código convertido, que foi testada exaustivamente. Não houve alteração em nenhuma circunstância reproduzida. O código convertido está disponível em: `https://github.com/DVSCross/isomer/`. Uma das classes convertidas, *Canvas*, é mostrada no código 15. Como pode ser veritificado, essa classe possui um construtor (Linhas 2-7) e dois métodos: *clear* (Linhas 8-10) e *path* (Linhas 11-17).

```
1   class Canvas {
2     constructor(elem) {
3       this.elem = elem;
4       this.ctx = this.elem.getContext('2d');
5       this.width = elem.width;
6       this.height = elem.height;
7     }
8     clear() {
9       this.ctx.clearRect(0, 0, this.width, this.height);
10    }
11    path(points, color) {
12      this.ctx.beginPath();
13      this.ctx.moveTo(points[0].x, points[0].y);
14      for (var i = 1; i < points.length; i++) {
15        this.ctx.lineTo(points[i].x, points[i].y);
16
17      }
18      this.ctx.closePath();
19      this.ctx.save();
20      this.ctx.globalAlpha = color.a;
21      this.ctx.fillStyle = this.ctx.strokeStyle = color.toHex();
22      this.ctx.stroke();
23      this.ctx.fill();
24      this.ctx.restore();
25    }
26  }
```

Código 15. Código ES 6 gerado para a classe Canvas do sistema Isomery.js

5. Trabalhos Relacionados

A análise de código JavaScript é alvo de estudos que buscam identificar desde más práticas de codificação [Fard and Mesbah 2013] até a utilização do paradigma de orientação a objetos [L.Silva et al. 2015a]. Existem muitas ferramentas de refatoração para linguagens estaticamente tipadas, porém poucas para linguagens dinâmicas como JavaScript [Feldthaus et al. 2011]. O principal motivo é a complexidade inerente à execução das linguagens dinâmicas [G.Richard et al. 2010]. As refatorações incluem também refatorações sintáticas, visando principalmente a melhoria da legibilidade do código-fonte [Feldthaus and Møller 2013]. A própria ECMA, detentora da sintaxe da ECMAScript, fundamentou grande parte das alterações da nova ES 6 em alterações de caráter sintático. Com o propósito de permitir a utilização da ES 6, mesmo com a falta de compatibilidade encontrada em diversos ambientes, uma ferramenta vem sendo amplamente utilizada: Babel.js[7]. Trata-se de um compilador que realiza a tradução de códigos ES 6 para ES 5, permitindo a utilização da nova sintaxe. Ou seja, essa ferramenta faz o processo inverso da

[6]http://jdan.github.io/isomer/playground/
[7]https://babeljs.io/

ferramenta apresentada neste artigo. O objetivo é exatamente possibilitar a interpretação de código ES 6 em máquinas virtuais que ainda não suportam esse novo padrão.

6. Conclusão e Trabalhos Futuros

Este trabalho propõe uma ferramenta de conversão de códigos JavaScript ES 5 para ES 6. Tal ferramenta se mostra útil, visto que os navegadores vem rapidamente provendo suporte aos novos recursos de ES 6 [Kangax 2016]. E, mesmo sem esse suporte completo, grandes companhias[8] já adotam a sintaxe de ES 6 junto a ferramentas de retrocompatibilidade. A migração para a ES 6 torna-se necessária então, para aqueles que não desejam converter manualmente o código em ES 5 mas desejam aproveitar os novos recursos da linguagem.

Como trabalho futuro, pretende-se aplicar a ferramenta proposta a um número maior de sistemas. Pretende-se também validar as conversões realizadas pela ferramenta proposta com os desenvolvedores de tais sistemas.

Referências

(2011). Ecma-262: Ecmascript language specification, edition 5.1.

(2015). Ecma-262: Ecmascript language specification, 6th edition.

Arnström, M., Christiansen, M., and Sehlberg, D. (2003 (acessado em 18 de Junho, 2016)). Prototype-based programming. `http://www.idt.mdh.se/kurser/cd5130/msl/2003lp4/reports/prototypebased.pdf`.

Blaschek, G. (2012). *Object-Oriented Programming: with Prototypes*. 1th edition.

Fard, A. M. and Mesbah, A. (2013). JSNose: Detecting JavaScript code smells. pages 116 – 125.

Feldthaus, A., Millstein, T., Møller, A., Schäfer, M., and Tip, F. (2011). Tool-supported refactoring for JavaScript. *SIGPLAN Notices*, 46(10):119–138.

Feldthaus, A. and Møller, A. (2013). Semi-automatic rename refactoring for JavaScript. *SIGPLAN Not.*, 48(10):323–338.

G.Richard, Lebresne, S., B.Burg, and J.Vitek (2010). An analysis of the dynamic behavior of JavaScript programs. *SIGPLAN Notices*, pages 1–12.

Kangax (2016). `https://kangax.github.io/compat-table/es6/`.

L.Silva, D.Hovadick, M.T.Valente, A.Bergel, N.Anquetil, and A.Etien (2015a). JSClassFinder: A Tool to Detect Class-like structures in JavaScript, pages 1–8.

L.Silva, M.Ramos, M.T.Valente, A.Bergel, and N.Anquetil (2015b). Does Javascript Software Embrace Classes? *SANER*, pages 73–82.

[8] https://babeljs.io/users/