

Domain Specific Languages

DSLs

¿DSL?

- Un lenguaje de expresividad limitada enfocado a un dominio en particular
- Expresar claramente
- Expresar cómodamente
- Comprender con facilidad

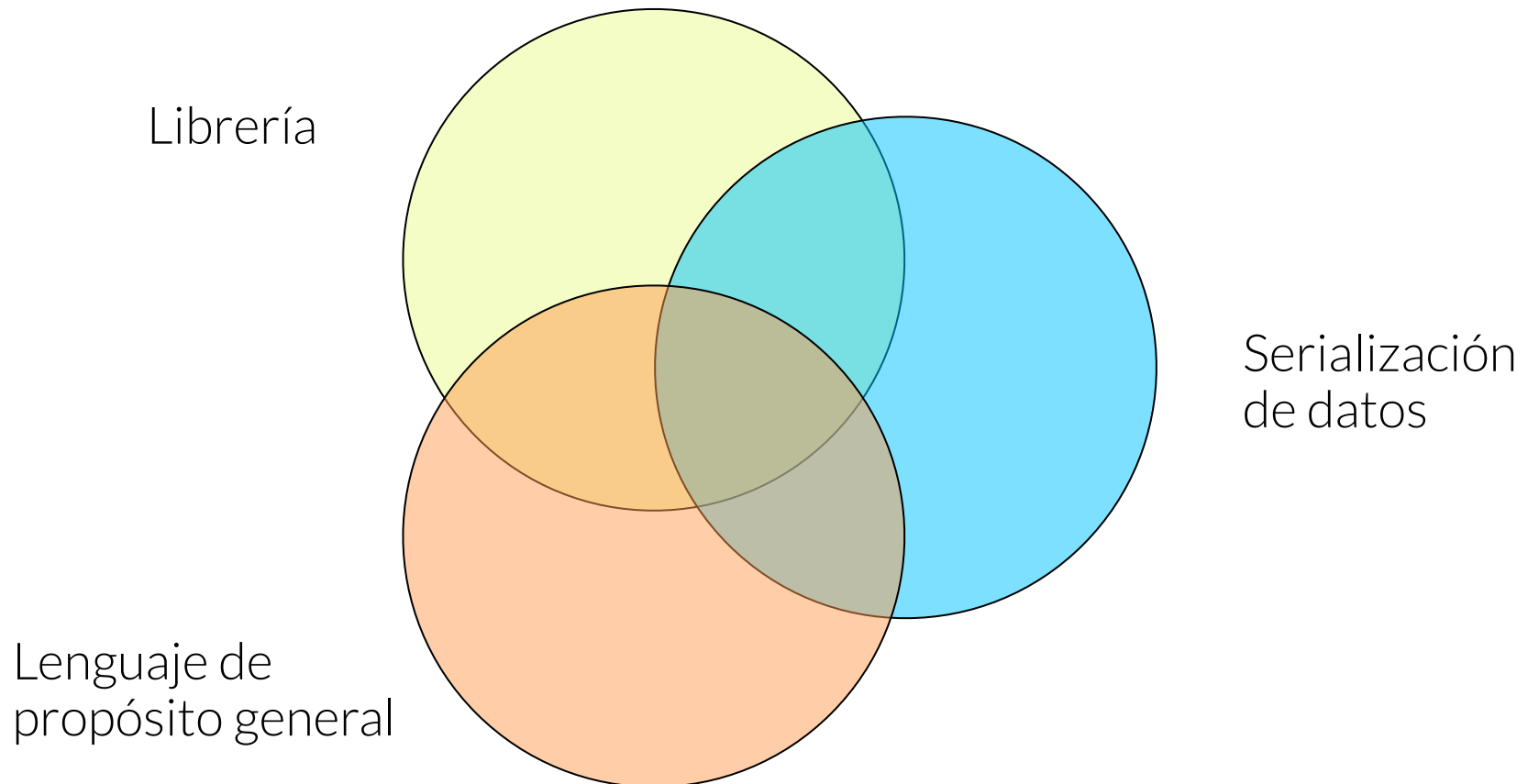
DSLs

Dos tipos:

- Externo
 - Un lenguaje en toda regla
 - CSS
 - SQL
- Interno
 - Un “lenguaje” incrustado
 - jQuery
 - RSpec

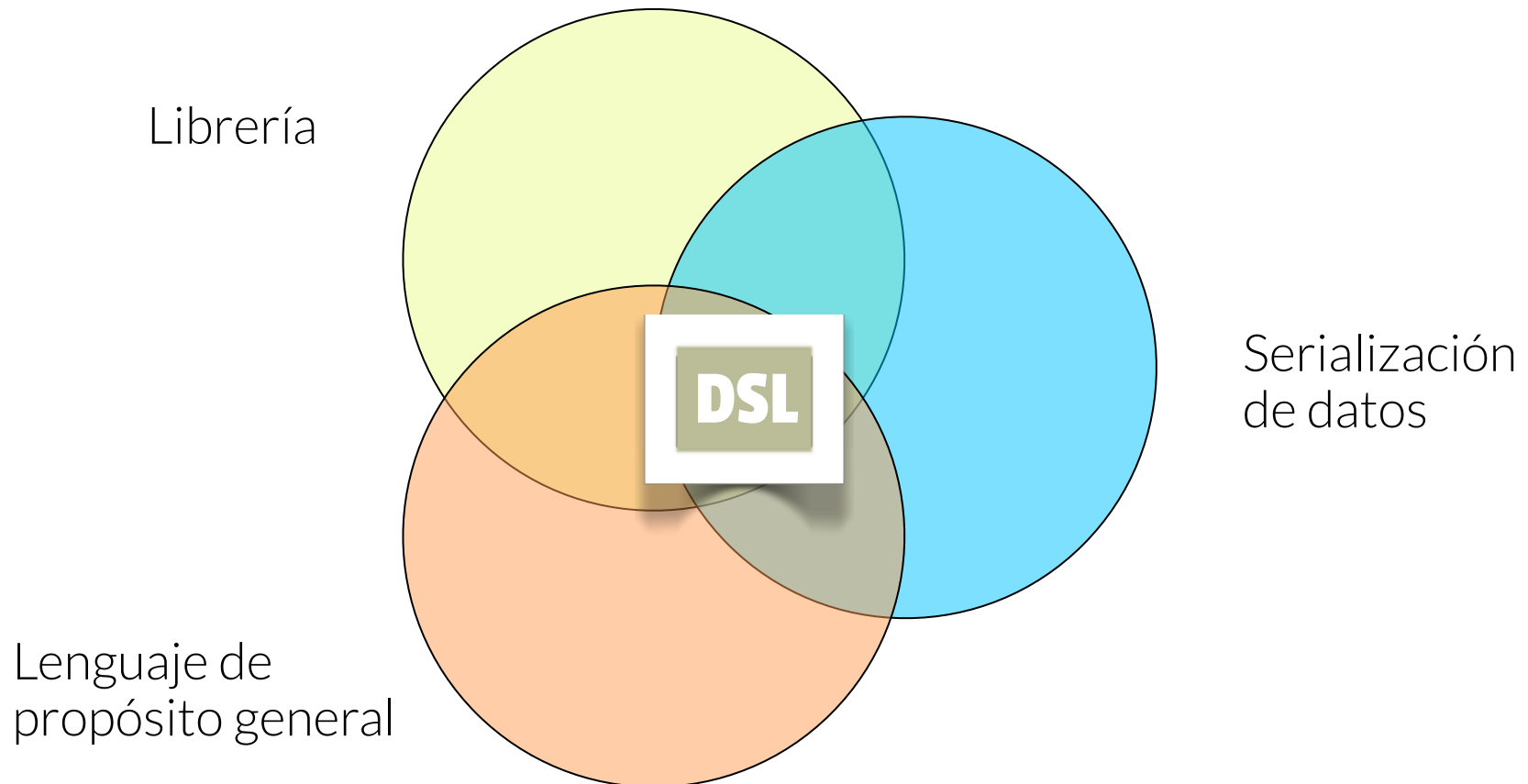
DSLs

Es un título que se otorga subjetivamente



DSLs

Es un título que se otorga subjetivamente



DSLs

¿Es esto JavaScript?

```
position: {  
  my: "center",  
  at: "center",  
  of: window,  
  collision: "fit" }
```

¿O es un lenguaje para definir posiciones?

“Position: my center at [the] center of window, [with a]
collision [of type] fit”

DSLs

¿Es esto JavaScript?

```
should.not.exist(err);  
should.exist(result);  
result.bar.should.equal(foo);  
user.should.be.a('object')  
    .and.have.property('name', 'tj')
```

DSLs

¿Es esto JavaScript?

```
uiDialogContent = this.element
    .show()
    .removeAttr( "title" )
    .addClass( "ui-dialog-content ui-widget-content" )
    .appendTo( uiDialog ),
```


DSLs

¿Dónde está el límite? ¿Es esto JavaScript?

```
{  
  "id": "arrow",  
  "width": 14,  
  "height": 14,  
  "animations": {  
    "idle_down": {  
      "length": 1,  
      "row": 0  
    }  
  }  
}
```

DSLs

¿Y esto?

```
$("#*:not(#divId) a > li.item")
```

DSLs

Pero hay una diferencia entre:

```
var d = document.createElement('div');  
d.className = "container";  
d.id = "main";  
var i = document.createElement('span');  
i.innerHTML = "Hey!";  
d.appendChild(i);  
body.document.appendChild(d);
```

Y:

```
$('<div/>', {id: "main"})  
  .addClass("container")  
  .append($('<span/>', {html: "Hey!"}))  
  .appendTo(document.body);
```

DSLs

¿Cómo se hace? (imperativo)

```
var d = document.createElement('div');  
d.className = "container";  
d.id = "main";  
var i = document.createElement('span');  
i.innerHTML = "Hey!";  
d.appendChild(i);  
body.document.appendChild(d);
```

¿Qué quieres? (declarativo)

```
$('<div/>', {id: "main"})  
  .addClass("container")  
  .append($('<span/>', {html: "Hey!"}))  
  .appendTo(document.body);
```

DSLs

- Externo:
 - Plantillas
- Interno:
 - Eventos
 - Máquinas de estados finitos
 - ▶ Con “bloques”
 - ▶ Interfaz fluida

DSL Externo

Más flexible, pero más costoso

- Manipulación de texto
- Dos enfoques:
 - Transformación (más fácil, más limitado)
 - Parser
- Implementar el lenguaje y el procesamiento del texto
- No se ven mucho en JS...


DSL Externo

Nuestro primer DSL: Plantillas


```
<h1>  
  <%= this.titulo %>  
</h1>  
  
<div>  
  <div> <%= this.cabecera %> </div>  
  <% if (this.body) { %>  
    <p>  
      <%= this.body %>  
    </p>  
  <% } %>  
</div>
```

DSL Externo

Nuestro primer DSL: Plantillas



```
<h1>  
  <%= this.titulo %>  
</h1>
```



```
<div>  
  <div> <%= this.cabecera %> </div>  
  <% if (this.body) { %>  
    <p>  
      <%= this.body %>  
    </p>  
  <% } %>  
</div>
```


DSL Externo

Nuestro primer DSL: Plantillas

```
// Con strings
```

```
var plantilla = "<p> <%= this.body %> </p>",  
    render = Template(plantilla);
```

```
console.log(  
    render({body: "Esto es un párrafo."})  
);
```

```
// O por ID de elemento
```

```
var render2 = Template.byId("id-elemento");
```

```
console.log(  
    render({titulo: "Probando", texto: "Mi sistema de plantillas"});  
);
```

DSL Externo

<http://jsbin.com/projs-dsl-1/5/edit>

DSL Externo

La idea general:

- Convertir el template en JS
- Que al ejecutarse construya el **string** adecuado
- Usando como **this** el parametro de la invocación

DSL Externo

Transformar esto:

```
<p>  
  <%= 2 + 2 %>  
</p>
```

En esto:

```
var __trozos__ = [];  
__trozos__.push('<p>', (2 + 2), '</p>');  
return __trozos__.join('');
```

DSL Externo

Transformar esto:

```
<h1>  
  <%= this.titulo %>  
</h1>
```

En esto:

```
var __trozos__ = []:  
__trozos__.push('<p>', (this.titulo), '</p>');  
return __trozos__.join('');
```

DSL Externo

Transformar esto:

```
<div class="<%= this.class %>">  
  <% if (this.cond) { %>  
    <span> Cierto! </span>  
  <% } %>  
</div>
```

En esto:

```
var __trozos__ = []:  
__trozos__.push('<div class=\"', (this.class), '\">>');  
if (this.cond) {  
  __trozos__.push('<span> Cierto! </span>');  
}  
__trozos__.push('</div>');  
return __trozos__.join('');
```

DSL Externo

1)

```
<div class="<%= this.clase %>">  
  Hola!  
</div>
```

2)

```
<div class=\"<%= this.clase %>\"> Hola! </div>
```

3)

```
'div class\"', (this.clase), '\\\"> Hola! </div>'
```

4)

```
__trozos__.push('div class\"', (this.clase), '\\\"> Hola! </div>');
```

DSL Externo

1)

```
<div class="<%= this.clase %>">  
  Hola!  
</div>
```

2)

```
<div class=\"<%= this.clase %>\"> Hola! </div>
```

3)

```
'div class\"', (this.clase), '\\\"> Hola! </div>'
```

4)

```
__trozos__.push('div class\"', (this.clase), '\\\"> Hola! </div>');
```


DSL Externo

- 1)

```
<div>  
  <% if (this.cond) { %>  
    Si!  
  <% } %>  
</div>
```
- 2)

```
%> <div> <% if (this.cond) { %> Si! <% } %> </div> <%
```
- 3)

```
__trozos__.push(' <div> ');  
if (this.cond) { __trozos__.push(' Si! '); }  
__trozos__.push(' </div> ');
```

DSL Externo

1) `<div>
 <% if (this.cond) { %>
 Si!
 <% } %>
</div>`

2) `%> <div> <% if (this.cond) { %> Si! <% } %> </div> <%`

3) `__trozos__.push(' <div> ');
if (this.cond) { __trozos__.push(' Si! '); }
__trozos__.push(' </div> ');`

DSL Externo

Output

Run with JSAuto-run JS☒

DSL Externo: Plantillas

Hola!

Desde Plantilla

Usuario1

Sencillo, pero útil!

Usuario2

No hay ya librerías que hacen esto?

DSL interno

Una ilusión óptica de fluidez

- Incrustado en JS
- Sintaxis más limitada
- Sensación de “lenguaje”
 - Fluidez
 - Ortogonalidad
- Mucho más común en JS

DSL interno

Cuatro técnicas:

- Parseo de strings
- Parámetros con nombre
- Manipulación de **this**
- Interfaces fluidas (o encadenadas)

DSL: Eventos

Segundo DSL: Declaración de eventos

DSL: Eventos

```
var Component = Class.extend({
  init: function() {
    this.el = $("#root");
    this.el.find(".my-thing").click(bind(this, this.onClickHandler));
    this.el.find(".ok-button").click(bind(this, this.onButtonClicked));
    this.el.find(".cancel-button").click(bind(this, this.onCancelClicked));
    this.el.find(".reset-button").click(bind(this, this.resetForm));
  }
});
```

```
var Component = Widget.extend({
  events: {
    "click .my-thing": "onClickHandler",
    "click .ok-button": "onButtonClicked",
    "click .cancel-button": "onCancelClicked",
    "click .reset-button": "resetForm",
  }
});
```

DSL: Eventos

```
var Component = Widget.extend({  
  events: {  
    "<event> <selector>": "<handler>"  
  }  
});
```


DSL: Eventos

<http://jsbin.com/projs-dsl-2/12/edit>

DSL: FSM

Vamos a por algo más interesante...

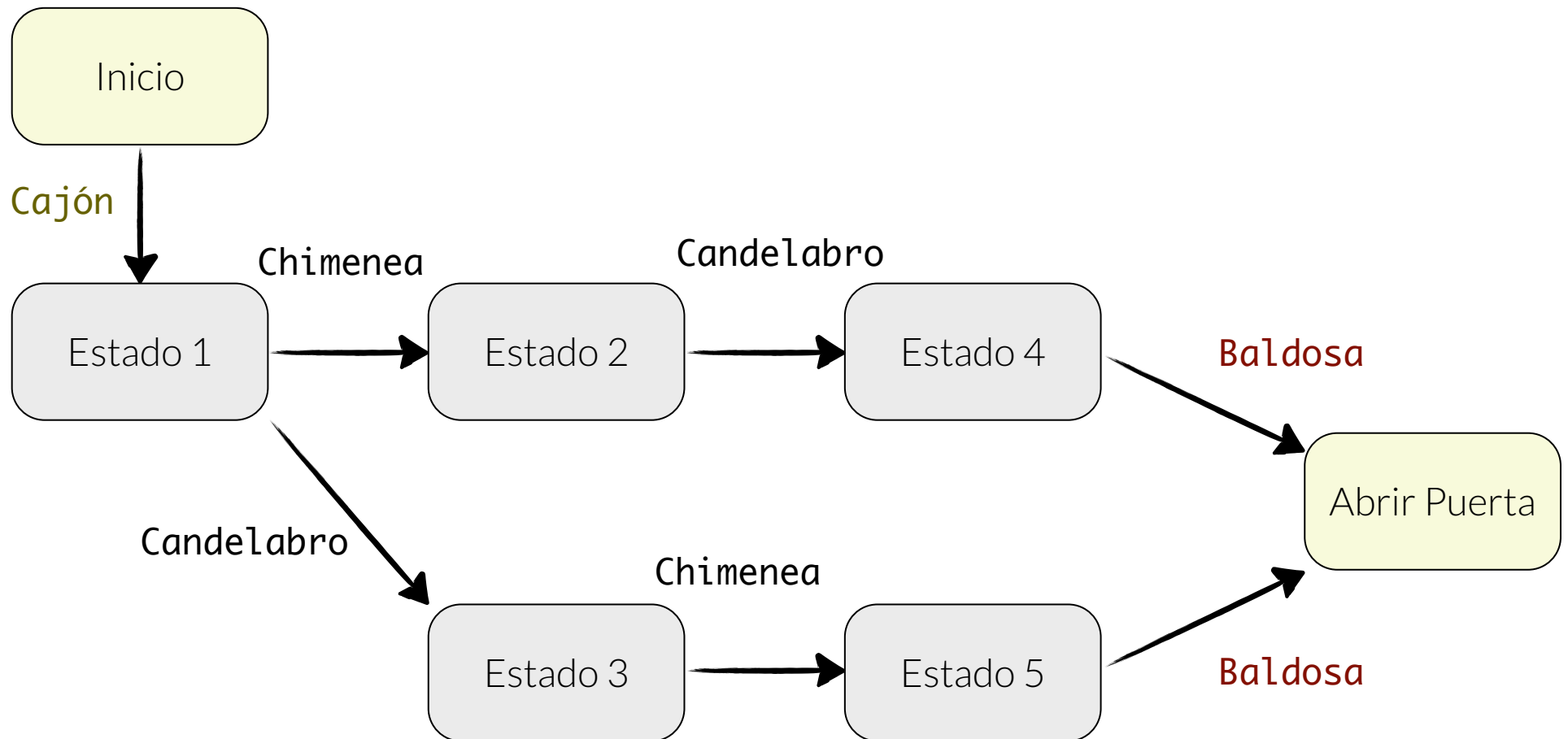
- Máquinas de Estados Finitos
- Muy prácticas para definir comportamientos complejos
- Engorrosas de implementar y describir
- Mucho mejor expresadas con un DSL

DSL: FSM

¡Manos a la obra!

- Una implementación de Máquinas de Estados Finitos
- Vamos a empezar por el modo “tradicional”

DSL: FSM



DSL: FSM

Modelo semántico

- Representar la información
- Modelado tradicional: objetos + operaciones
- Soportar el “significado” del futuro DSL

FSM: DSL

<http://jsbin.com/projs-dsl-3/1/edit>

DSL: FSM

Primer enfoque: “bloques”

- Contexto dinámico
- Mayor expresividad
- Funciones como “bloques”

DSL: FSM (1)

```
var fsm2 = ThisFSM(function() {  
  
  this.describeState('Inicio', function(){  
    this.setAsStart();  
    this.on({input: 'cajon', goesTo: 'Estado 1'});  
  });  
  
  this.describeState('Estado 1', function(){  
    this.on({input: 'chimenea', goesTo: 'Estado 2'});  
    this.on({input: 'candelabro', goesTo: 'Estado 3'});  
  });  
  
  // ....  
  
});
```


DSL: FSM (1)

```
var fsm2 = ThisFSM(function() {
```



```
  this.describeState('Inicio', function(){
```

```
    this.setAsStart();
```

```
    this.on({input: 'cajon', goesTo: 'Estado 1'});
```

```
  });
```

```
  this.describeState('Estado 1', function(){
```

```
    this.on({input: 'chimenea', goesTo: 'Estado 2'});
```

```
    this.on({input: 'candelabro', goesTo: 'Estado 3'});
```

```
  });
```

```
  // ....
```

```
});
```

DSL: FSM (1)

```
var fsm2 = ThisFSM(function() {  
  this.describeState('Inicio', function(){  
    this.setAsStart();  
    this.on({input: 'cajon', goesTo: 'Estado 1'});  
  });  
  
  this.describeState('Estado 1', function(){  
    this.on({input: 'chimenea', goesTo: 'Estado 2'});  
    this.on({input: 'candelabro', goesTo: 'Estado 3'});  
  });  
  
  // ....  
});
```

DSL: FSM (1)

<http://jsbin.com/projs-dsl-3/16/edit>

DSL: FSM (1)

La técnica consiste en:

- Utilizar el modelo semántico
- Ir configurándolo según el DSL
- Mediante builders
 - Builders: Adaptadores DSL-Modelo Semántico

Intermedio: DSL FSM

Ampliar el lenguaje

- Estados del Modelo Semántico tienen callbacks
 - enter
 - leave
- Ampliar el DSL para poder configurar los callbacks

```
this.describeState('Estado 1', function(){  
  this.enter(function() { alert("Hola desde " + this.nombre); });  
  this.leave(function() { alert("Adios desde " + this.nombre); });  
  this.on({input: 'chimenea', goesTo: 'Estado 2'});  
  this.on({input: 'candelabro', goesTo: 'Estado 3'});  
});
```

Intermedio: DSL FSM

<http://jsbin.com/projs-dsl-3/13/edit>

DSL FSM: Bloques

Otra opción similar:

- Usar un parámetro explícito en vez de **this**
- Más explícito, más legible
- <http://jsbin.com/projs-dsl-3/19/edit>

DSL FSM: Bloques

```
var fsm2 = ThisFSM(function(machine) {  
  
  machine.describeState('Inicio', function(state){  
    state.setAsStart();  
    state.on({input: 'cajon', goesTo: 'Estado 1'});  
  });  
  
  machine.describeState('Estado 1', function(state){  
    state.enter(function() { alert("Hola desde " + state.nombre); });  
    state.leave(function() { alert("Adios desde " + state.nombre); });  
    state.on({input: 'chimenea', goesTo: 'Estado 2'});  
    state.on({input: 'candelabro', goesTo: 'Estado 3'});  
  });  
  
  // ...  
  
});
```


DSL: Bloques

¿Cuándo utilizar “bloques”?

- Máxima flexibilidad (¡es JS!)
- Configuraciones complejas
- Configuraciones jerárquicas (contextos anidados)

DSL: Bloques

Ejemplo: Jasmine

```
describe("A suite", function() {  
  it("contains spec with an expectation", function() {  
    expect(true).toBe(true);  
  });  
});
```

DSL: Bloques

Ejemplo: Jasmine

```
describe("A suite", function() {  
  it "contains spec with an expectation", function() {  
    expect(true).toBe(true);  
  });  
});
```

DSL: Bloques

Limitaciones

- JS no resuelve automáticamente nombres de métodos
- Más feo: siempre prefijos
- Se puede apañar con un hack (bastante feo)
 - <http://jsbin.com/projs-dsl-3/22/edit>

DSL: Interfaces fluidas

Segundo enfoque: llamadas encadenadas

- Cada llamada devuelve un objeto
- Sobre el que se puede seguir operando
- ¡Muy sucinto!
- Más característico de JS

DSL: FSM (2)

```
fsm3 = FluentFSM()  
  .state('Inicio')  
    .on({input: 'cajon', goesTo: 'Estado 1'})  
    .leave(function() { alert("Empezamos!"); })  
    .setAsStart()  
  .state('Estado 1')  
    .on({input: 'chimenea', goesTo: 'Estado 2'})  
    .on({input: 'candelabro', goesTo: 'Estado 3'})  
  .end();
```

DSL: FSM (2)

<http://jsbin.com/projs-dsl-4/6/edit>

DSL: FSM (2)

La técnica consiste en:

- Configurar el Modelo Semántico
- Con un objeto que se devuelve a si mismo
- La llamada **.end()** devuelve el Modelo Semántico

DSL: FSM (2)

```
// ...  
state: function(nombre) {  
    var estado = this.estadoActual = new State(nombre);  
    this._fsm.addState(estado);  
    this.estados[nombre] = estado;  
    return this;  
},  
on: function(options) {  
    this.transiciones.push({  
        state: this.estadoActual,  
        input: options.input,  
        target: options.goesTo  
    });  
    return this;  
},  
// ...
```



DSL: Interfaces Fluidas

¿Cuándo utilizar interfaces fluidas?

- El DSL es extremadamente descriptivo
- Encadenar operaciones
- El orden es importante

DSL: Interfaces Fluidas

Ejemplo: jQuery

```
var li = $("- ")  
    .append($('<a/>', {html:name,  
                        href:'#',  
                        class: "button"}))  
    .click(bind(this.queue, action))  
    .prependTo(this.ul);

```

Intermedio: (no) Aumentar Prototipos Primitivos

¿Cómo podríamos hacer esto?

```
"4".times(function(i) {  
  console.log("Hola por " + i + " vez!");  
});
```

Intermedio: (no) Aumentar Prototipos Primitivos

Otro caso útil:

```
"%1 + %1 = %2".format(10, 20); // 10 + 10 = 20
```

Intermedio: (no) Aumentar Prototipos Primitivos

```
String.prototype.format = function() {  
    var args = [].slice.call(arguments),  
        result = this.slice(),  
        regexp;  
    for (var i=args.length; i--;) {  
        regexp = new RegExp("%" + (i+1), "g")  
        result = result.replace(regexp, args[i]);  
    }  
    return result;  
};
```

Intermedio: (no) Aumentar Prototipos Primitivos

Un poco más complicado:

```
[1, 2, 3, 4].map("%1 + 1".to_f());
```

```
"4".times("console.log(%1)".to_f());
```

```
"%1 + %2 + %3".to_f()(10, 100, 1);
```

Intermedio: (no) Aumentar Prototipos Primitivos

```
String.prototype.to_f = function() {  
    var code = this.replace(/\%(\d+)/g,  
                           "(arguments[parseInt($1, 10) - 1])"),  
        statements = code.split(';'),  
        last = statements.pop();  
    if (!/return/.test(last)) last = "return " + last;  
    statements.push(last);  
    return new Function(statements.join(';'));  
};
```


Intermedio: (no) Aumentar Prototipos Primitivos

Otro ejemplo:

```
"5".days().ago();  
"3".months().ago();  
"1".year().ago();
```

Intermedio: (no) Aumentar Prototipos Primitivos

Cuidado con aumentar los prototipos!

- No se considera una buena práctica
- Colisiones
- Pero es útil en casos concretos!

DSL: Conclusión

¿Qué tienen todos los DSL en común?

- “Que” en vez de “cómo”
- Declarativo en vez de imperativo
- Describen una configuración en lugar de un proceso

DSL: Conclusión

Usar DSLs es una apuesta

- ✓ Mayor claridad
- ✓ Más conciso
- ✓ Más cómodo
- ⊙ Un desarrollo extra
- ⊙ Se puede complicar mucho