Programación Orientada a Componentes

Metaprogramación Compositiva En JavaScript

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Mayo 2015

Autor

I. ¿Quién Soy?

Licenciado en informática por la UPM desde el año 2001 y doctor en informática por la UNED desde el año 2009, Javier es investigador y su línea de trabajo actual se centra en la innovación y desarrollo de tecnologías de Componentes Web. Además realiza actividades de evangelización y divulgación en diversas comunidades IT, es Polymer Polytechnic Speaker y co-organizador del grupo Polymer Spain que conforma una comunidad de interés de ámbito nacional en relación al framework Polymer y a las tecnologías de Componentes Web.

javier.velez.reyes@gmail.com

@javiervelezreye

linkedin.com/in/javiervelezreyes

gplus.to/javiervelezreyes

jvelez77

javiervelezreyes

youtube.com/user/javiervelezreyes

II. ¿A Qué Me Dedico?

Polymer Polytechnic Speaker

Co-organizador de Polymer Spain

Evangelización Web

Desarrollador JS Full stack

Arquitectura Web

Formación & Consultoría IT

e-learning

Javier Vélez Reyes
@javiervelezreye
Javier.velez.reyes@gmail.com

1 Introducción

- Qué Es La Metaprogramación
- Por Qué La Metaprogramación
- El Proceso De La Metaprogramación
- La Metaprogramación Como Paradigma

Qué Es La Metaprogramación

Definición

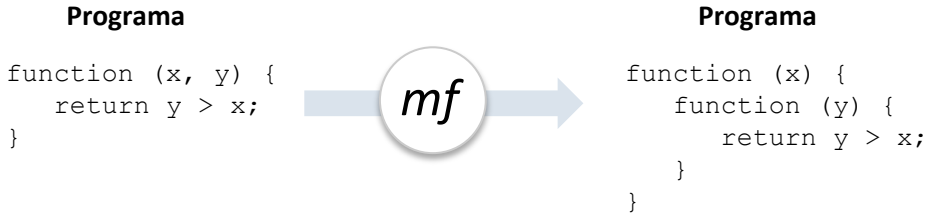
La metaprogramación es una disciplina de programación que consiste en construir programas que generan, manipulan o modifican otros programas, incluidos ellos mismos.

Programación

El programa toma unos datos de entrada y produce unos resultados de salida de acuerdo a un proceso de transformación

Metaprogramación

El metaprograma toma unos datos de entrada que representan un programa y valores de configuración para dirigir el proceso de construcción de otro programa



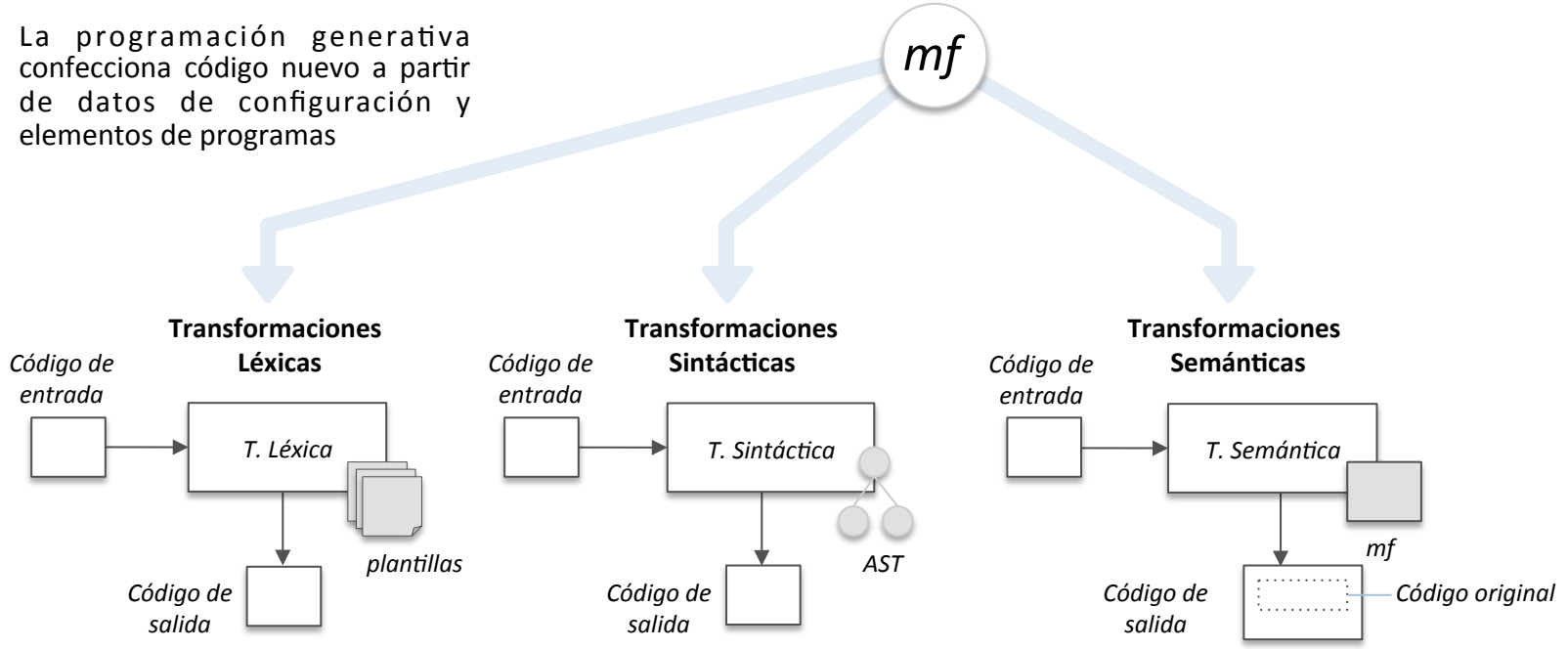
Qué Es La Metaprogramación

Tipos De Metaprogramación

En términos generales se puede pensar en dos grandes familias de técnicas de metaprogramación. La programación generativa y la programación compositiva.

Programación Generativa

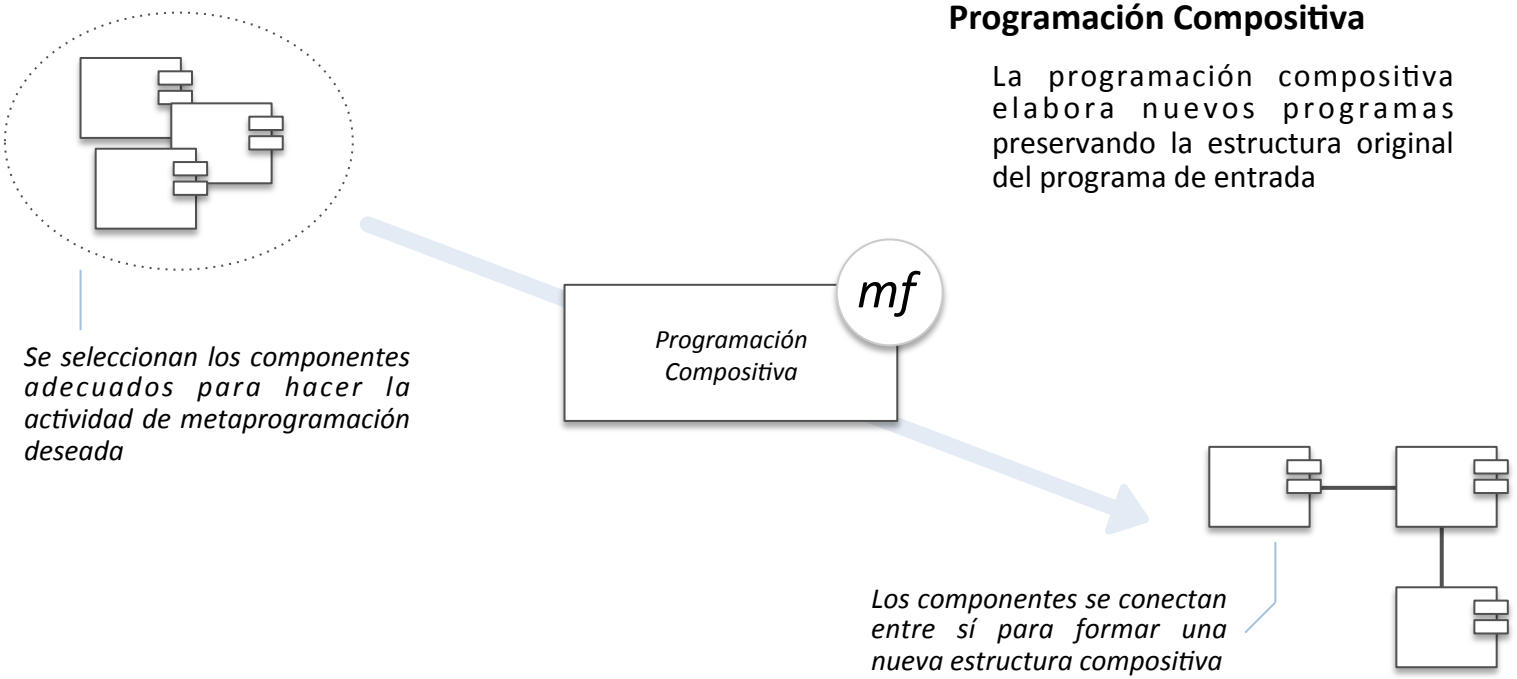
La programación generativa confecciona código nuevo a partir de datos de configuración y elementos de programas



Qué Es La Metaprogramación

Tipos De Metaprogramación

En términos generales se puede pensar en dos grandes familias de técnicas de metaprogramación. La programación generativa y la programación compositiva.



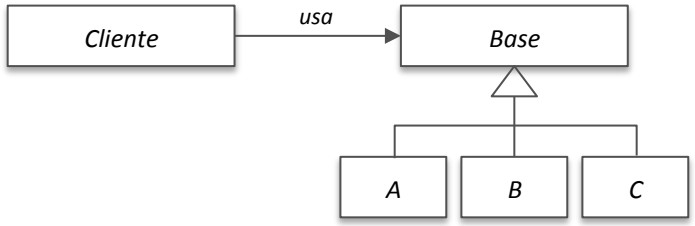
Programación Compositiva

La programación compositiva elabora nuevos programas preservando la estructura original del programa de entrada

Qué Es La Metaprogramación

Metaprogramación vs Programación Orientada a Objetos

La diferencia esencial entre la programación orientada a objetos clásica y la metaprogramación radica en la forma en que tienen ambas aproximaciones de hacer frente a los problemas de alineamiento arquitectónico.

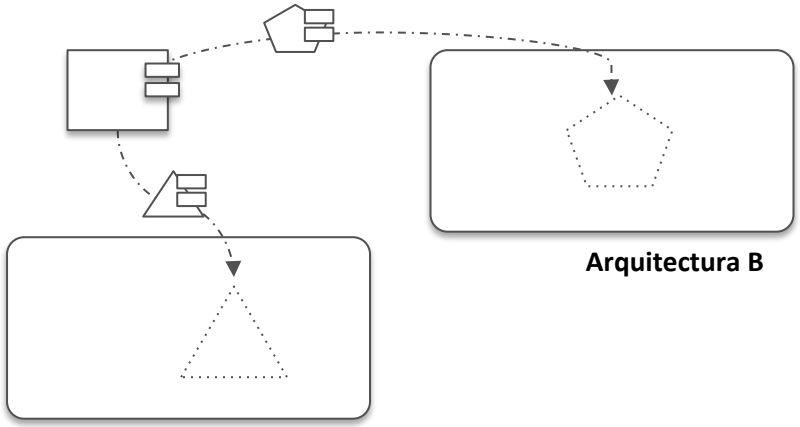


Programación O. Objetos

En la programación orientada a objetos la variabilidad entre artefactos debe ser absorbida por la arquitectura en forma de puntos de extensión polimórfica

Metaprogramación

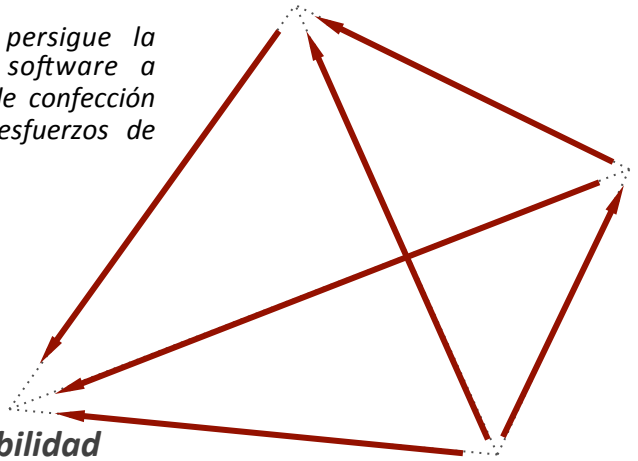
En la metaprogramación los componentes son los responsables de adaptarse a cada contexto arquitectónico de aplicación por medio de una transformación adaptativa



Por Qué La Metaprogramación

Automatización, Reutilización, Mantenibilidad & Extensibilidad

En términos generales, podemos resumir en cuatro los objetivos principales de la metaprogramación. Automatización, reutilización, mantenibilidad y extensibilidad. Todos ellos promueven, en mayor o menor medida, evidentes mejoras en relación a aproximaciones de menor flexibilidad.



Automatización

La automatización persigue la idea de construir software a través de procesos de confección que minimizan los esfuerzos de desarrollo

Reutilización

La plasticidad contractual de los componentes se regenera o adapta para permitir que estos encajen fácilmente en diversos contextos de explotación arquitectónica

Mantenibilidad

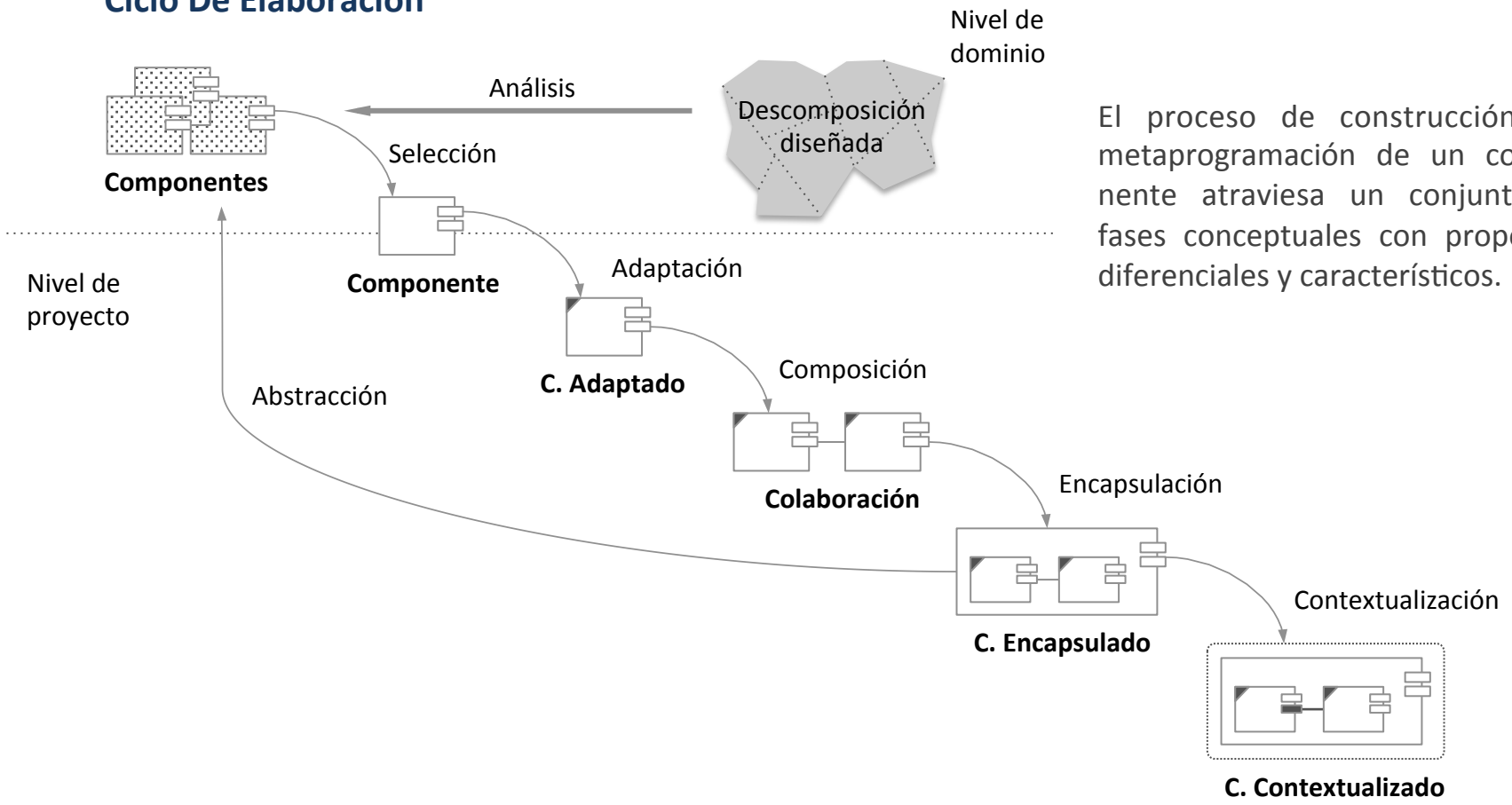
El uso de componentes permite una adecuada especialización de responsabilidades que fomenta la mantenibilidad por sustitución de partes

Extensibilidad

El carácter adaptativo o generativo de los componentes les confiere una mayor capacidad evolutiva lo que facilita la extensibilidad global del sistema

El Proceso De La Metaprogramación

Ciclo De Elaboración



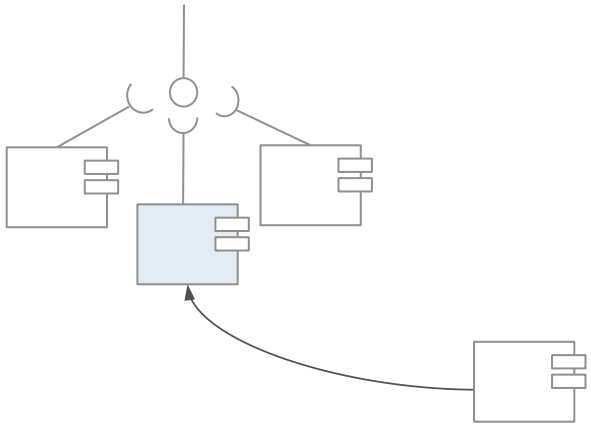
El proceso de construcción por metaprogramación de un componente atraviesa un conjunto de fases conceptuales con propósitos diferenciales y característicos.

El Proceso De La Metaprogramación

Selección

La selección consiste en determinar las variantes específicas de cada tipo de componente que formarán parte del proceso constructivo. La selección estática es aquella que se realiza enteramente antes del tiempo de ejecución. Por el contrario, la selección dinámica implica que el metaprograma inyecta lógica de negocio para articular la selección en tiempo de ejecución pudiendo hacer ésta dinámicamente cambiante.

Selección



Selección Estática

La selección estática se resuelve antes del tiempo de ejecución de manera que las variantes seleccionadas quedan definitivamente fijadas

Selección Dinámica

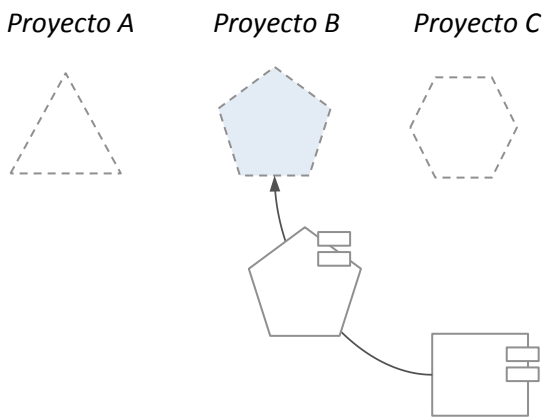
La ventaja de la selección dinámica es que permite dar soporte a escenarios donde las variantes utilizadas pueden cambiar durante el tiempo de ejecución

El Proceso De La Metaprogramación

Adaptación

La adaptación es el proceso mediante el cual un componente se adapta para resolver las impedancias que presenta con el contexto arquitectónico de explotación. La adaptación sintáctica se centra en adaptar el contrato del componente mientras que la adaptación semántica persigue modificar el comportamiento del artefacto a los requerimientos del proyecto. Tanto estandarizar los componentes como trabajar con componentes prototipo evita frecuentemente la adaptación.

Adaptación



Adaptación sintáctica & semántica

Un ejemplo de adaptación sintáctica consiste en cambiar los nombres de los métodos de un componente. La decoración por aspectos del comportamiento de un método es un ejemplo de adaptación semántica

Estandarización & Prototipos

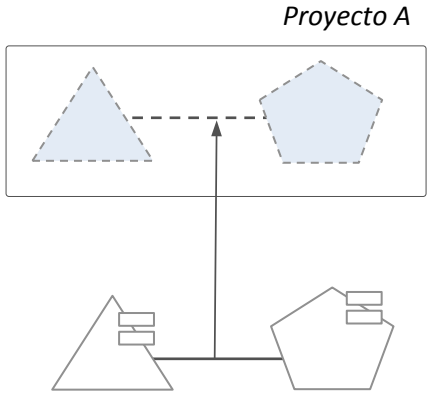
La estandarización de contratos y comportamientos evita la adaptación. Asimismo cuando los componentes son directamente operativos se posibilita la ausencia ocasional de adaptación

El Proceso De La Metaprogramación

Composición

La composición es un ejercicio de transformación en el que un componente al que llamaremos núcleo se enriquece por la adquisición de cierta lógica parcial que reside en otro u otros componentes llamados extensiones. En el marco de esta relación binaria es posible analizar el grado de dependencia que tienen cada una de estas partes entre sí para caracterizar el tipo de composición.

Composición

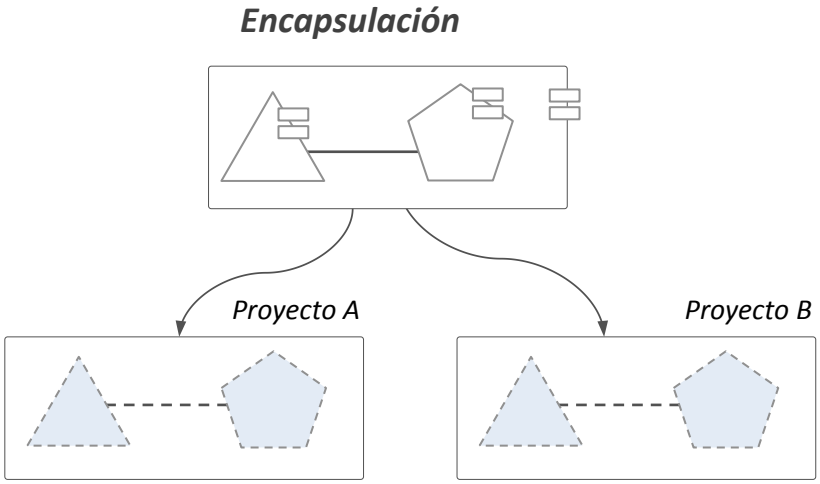


Extensión \ Núcleo	Dependiente	Autónomo
Dependiente	*Composición Cooperativa*	*Composición Subordinada*
Autónomo	*Composición Subordinante*	*Composición Autónoma*

El Proceso De La Metaprogramación

Encapsulación

La encapsulación es un proceso mediante el cual se establecen nuevas fronteras arquitectónicas en el marco de una composición que tienen por objetivo aislar la misma del contexto de explotación. Este aislamiento permite la protección de la información. Por extensión consideraremos que todo mecanismo que garantice esta protección es una forma débil de encapsulación.



Encapsulación Fuerte

La encapsulación fuerte es el proceso mediante el cual se levantan unas fronteras arquitectónicas nuevas en el marco de una relación compositiva

Encapsulación Débil

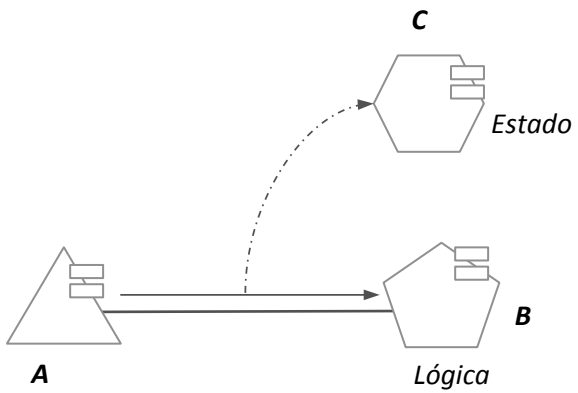
Por extensión, consideraremos que cualquier mecanismo que proporcione protección de información en el marco de la composición es una forma de encapsulación débil

El Proceso De La Metaprogramación

Contextualización

La contextualización es la fase última del proceso compositivo mediante la cual se dota al componente de unas condiciones ambientales de contexto que condicionarán su comportamiento durante el tiempo de ejecución. Cuando estas condiciones ambientales se fijan antes del tiempo de ejecución hablamos de contextualización estática. La variante dinámica permite articular recontextualizaciones en tiempo de ejecución.

Contextualización



Contextualización Estática

La contextualización estática resuelve las condiciones ambientales de contexto antes de que el componente entre en el tiempo de ejecución.

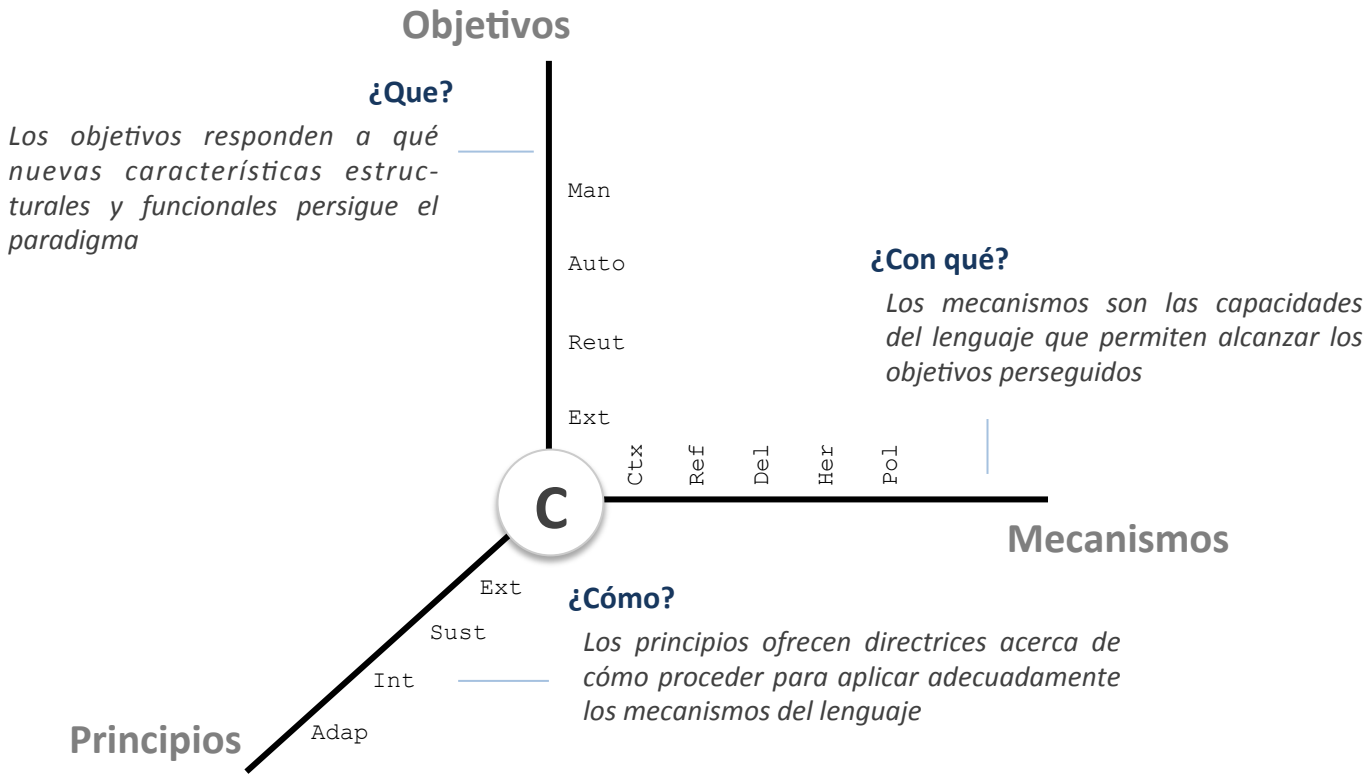
Contextualización Dinámica

La contextualización dinámica permite restablecer recurrentemente el contexto de ejecución lo que ofrece mayores oportunidades de adaptación

La Metaprogramación Como Paradigma

Ejes Dimensionales

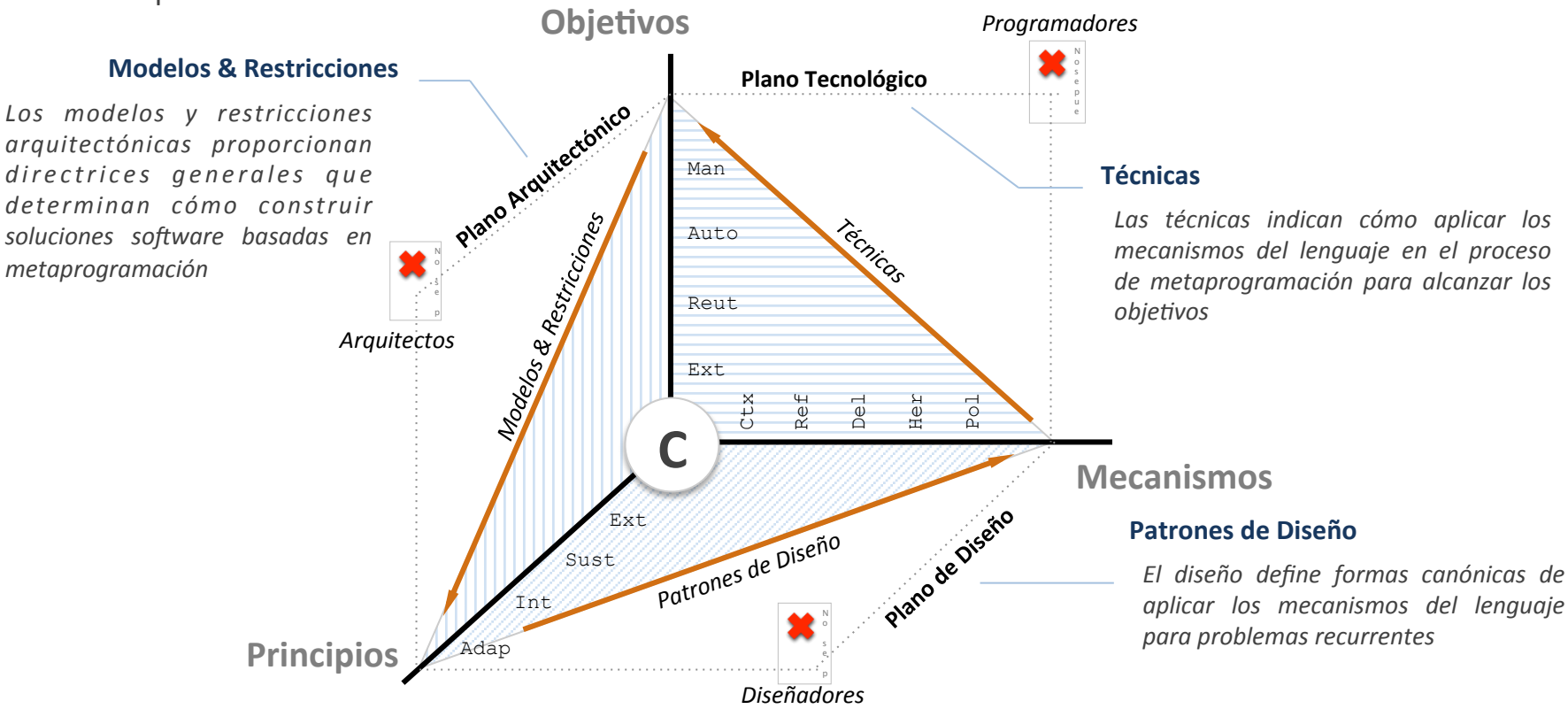
En relación al concepto de componente se pueden describir tres ejes dimensionales. Los objetivos determinan los propósitos perseguidos, los mecanismos caracterizan las capacidades nativas del lenguaje y los principios de diseño ofrecen directrices generales de actuación.



La Metaprogramación Como Paradigma

Planos De Actividad

En torno a los tres ejes dimensionales se pueden establecer sendos planos de actividad. Las técnicas indican cómo aplicar los mecanismos del lenguaje, los patrones presentan soluciones a problemas recurrentes y los modelos arquitectónicos ofrecen restricciones que determinan como proceder.



Javier Vélez Reyes
@javiervelezreye
Javier.velez.reyes@gmail.com

2

JavaScript como Lenguaje De Metaprogramación

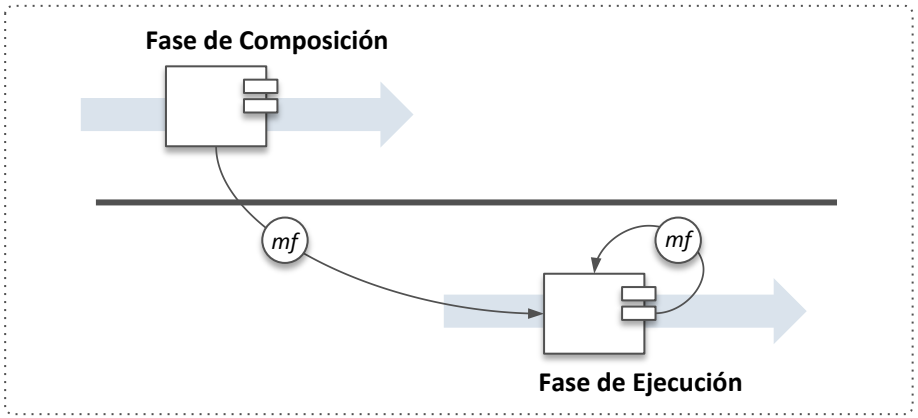
- Introducción
- Capacidades de JavaScript En El Modelo de Objetos
- Capacidades de JavaScript En El Modelo Funcional

Introducción

Composición Dinámica

JavaScript es un lenguaje que permite articular técnicas de composición dinámica donde la fase de composición y ejecución coexisten durante el tiempo de producción de la solución software. Esto permite que los componentes de adapten recurrentemente a las condiciones ambientales cambiantes de ejecución.

Durante la fase de composición se construyen por técnicas compositivas los componentes que formarán parte del programa

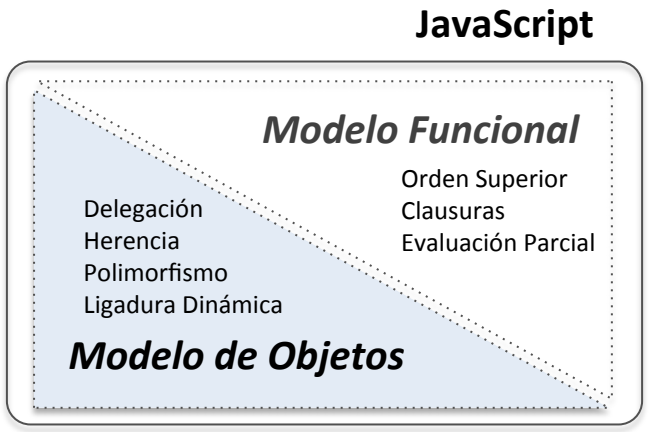


En ejecución, los cambios en las condiciones ambientales provocan transformaciones en los componentes para adaptarse a las mismas

Introducción

Modelo De Componente en JavaScript

El modelo de componentes en JavaScript es dual y combinado. Es posible operar dentro del espacio de objetos o con las álgebras funcionales. De hecho todo objeto da acceso a los métodos miembros que potencialmente incluye y toda función es candidata a explotar o retornar como respuesta una estructura de datos basada en objetos.



El modelo de objetos contiene métodos miembro cuyas funciones manejadoras operan en el espacio funcional

Modelo de objetos

```
var position = {
  x: 1,
  y: 1,
  step: function (i, j){
    return Position (this.x+i, this.y+j)
  }
};
```

Modelo de Funciones

```
var Position = function (x, y) {
   return {
      x: x,
      y: y,
      step: function (i, j){
       return Position (this.x+i, this.y+j)
     }
   };
}
```

El modelo de funciones permite generar nuevos objetos como resultado de su ejecución

Capacidades De JavaScript En El Modelo De Objetos

Los Objetos En La Composición

Los objetos son uno de los dos ciudadanos de primera categoría dentro de este modelo. En JavaScript un objeto es un mero agregador de propiedades organizado como un diccionario cuyos valores pueden ser tipos primitivos o funciones que ejecutan en el contexto del objeto.

o

```
{
  p1: v1,
    ...
  pn: vn,

  m1: function (){},
    ...
  mn: function (){}
}
```

Objetos como agregadores

El objeto en JavaScript es un mero agregador de características funcionales y de estado pero no constituye un clasificador estricto

Objetos Abiertos

```
var o = {

};
```

```
o.x = 1;
o.y = 1;
o.z = 1;
```

```
var o = {
  x : 1,
  y : 1,
  z : 1
};
```

```
delete o.z
```

```
var o = {
  x : 1,
  y : 1
};
```

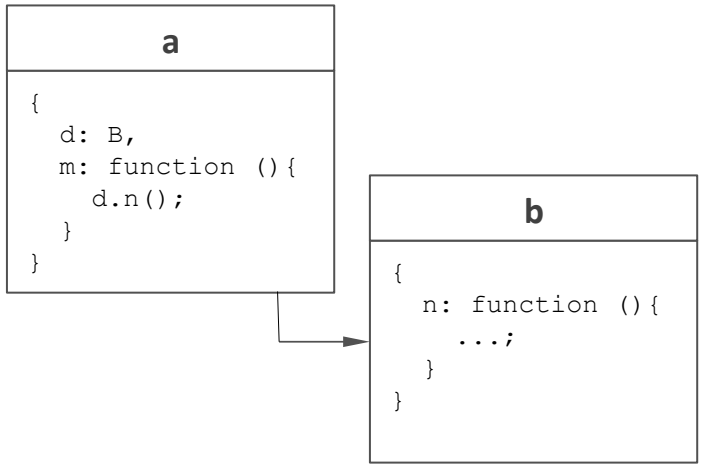
Objetos abiertos

Podemos ampliar y reducir la carga semántica del objeto a través de operaciones de modificación sobre el mismo

Capacidades De JavaScript En El Modelo De Objetos

La Delegación En La Composición

Por medio de la delegación se articulan esquemas de colaboración entre objetos de manera que los algoritmos se distribuyen entre el ecosistema de objetos participantes. Esto permite articular una adecuada división de responsabilidades en las fases preliminares de diseño.

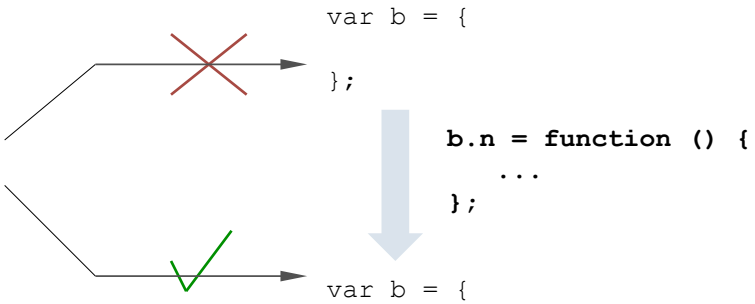


Delegación

El esquema de delegación impone una parte estática, que determina el nombre del método a invocar y otra dinámica que informa del objeto donde reside dicho método

Dynamic Binding

```
var a = {
  d: b,
  m: function (){
    d.n();
  }
};
```



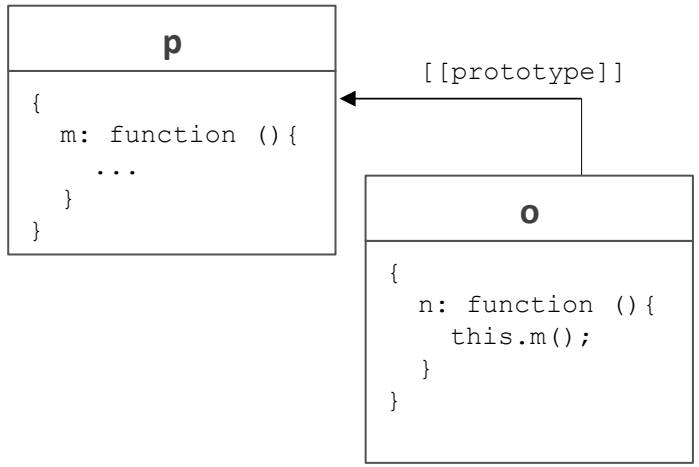
Enlace dinámico

El enlace dinámico permite resolver la localización del método llamado justo antes de la invocación lo que se alinea con el modelo de objetos abiertos

Capacidades De JavaScript En El Modelo De Objetos

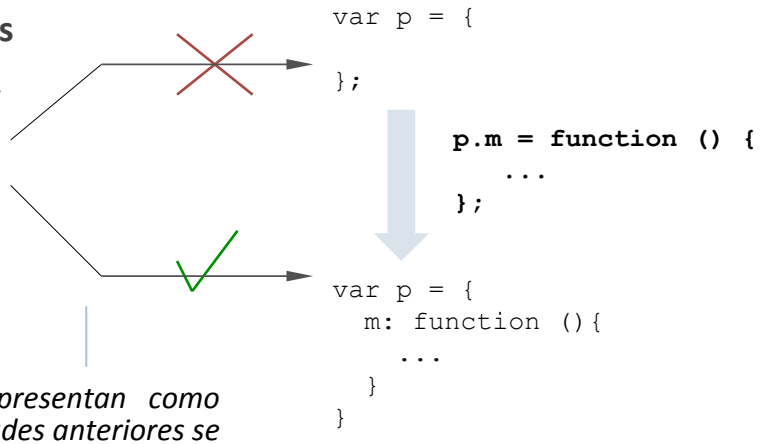
La Herencia En La Composición

Otra relación de delegación implícita entre objetos se marca por medio de la cadena de prototipado. Todo objeto se vincula a un prototipo en el que delega la búsqueda de una propiedad cuando ésta no se encuentra en el propio objeto. Las clases son prototipos de objetos con lo que este concepto se convierte en un rol.



Prototipos como Clases

```
var o = Object.create(p);
o.n = function (){
    this.m();
};
```



Prototipos

Dado que las clases se representan como objetos prototipo, las propiedades anteriores se aplican a la relación de herencia

Herencia

En JavaScript la herencia es un tipo especial de delegación. Dado que las clases se representan como prototipos los conceptos de clase y objeto son meros roles en torno a la relación de herencia

Heredar es Delegar

La herencia es una forma de delegación fuerte a través de la cadena de prototipado. Debido al carácter abierto de los prototipos es la forma más dinámica de delegación

Capacidades De JavaScript En El Modelo De Objetos

El Polimorfismo En La Composición

A diferencia de otros lenguajes donde el polimorfismo se soporta sobre el sistema de tipos, en JavaScript el polimorfismo sólo puede interpretarse en términos de la conformidad a un contrato establecido por convenio. Esta conformidad puede ser total o parcial.

a

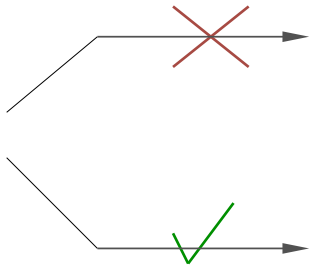
```
{
  m: function () {},
  x: function () {}
}
```

b

```
{
  m: function () {},
  y: function () {}
}
```

Duck Typing

```
var c = {
  m: function (o){
    o.m();
  }
};
```



```
var o = {

};
c.m(o);
```

```
o.m = function () {
    ...
};
```

```
var o = {
  m: function () {}
};
c.m(o);
```

Duck Typing

Para garantizar que o es compatible con el protocolo de c, se debe imponer la existencia de un método m en o. La metaprogramación puede transformar o en este sentido

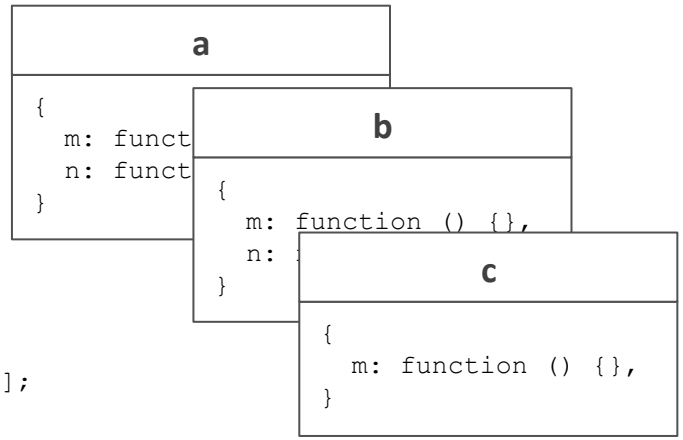
Conformidad Contractual

Los objetos a y b resultan conformes a un contrato definido por la existencia del método m. Esto es una forma de polimorfismo débil que no está articulada a través de herencia ni de definición de interfaces como en la OOP clásica

Capacidades De JavaScript En El Modelo De Objetos

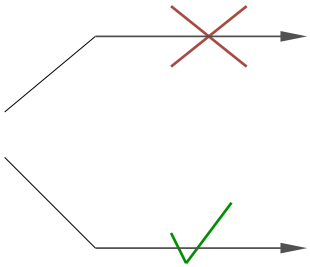
La Abstracción En La Composición

A través del polimorfismo débil que ofrece JavaScript es posible alcanzar cotas de expresividad de mayor abstracción. Esto es una propiedad muy conveniente para la definición de metaprogramas que deben operar en términos de alta generalidad. El siguiente ejemplo muestra un esquema que opera a nivel abstracto.



Abstracción

```
var app = {
  all: function (os){
    for (o in os)
      o.m();
  }
};
```



```
var os = [a, b, c];
app.all (os);
```

```
os[2].m = function () {
   ...
};
```

```
var os = [a, b, c];
app.all (os);
```

Abstracción

Los algoritmos se pueden expresar en términos abstractos asumiendo que cada objeto participante incluirá una versión polimórfica de los métodos implicados

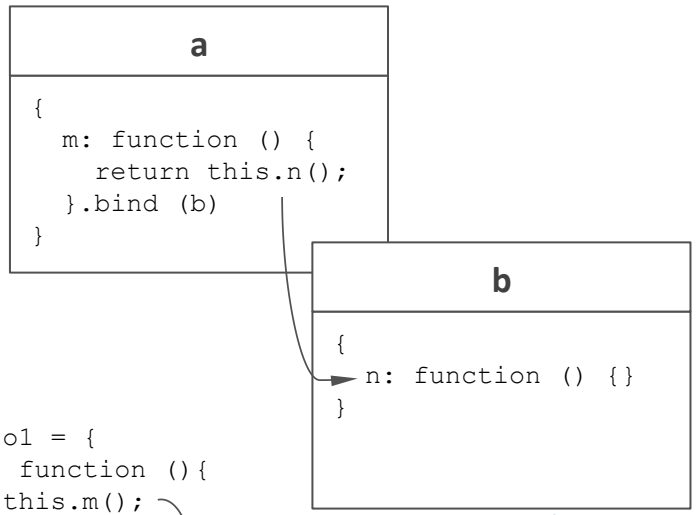
Conformidad Parcial

Los objetos a y b resultan conformes a un contrato que incluye los métodos m y n. El objeto c es un artefacto con conformidad parcial a dicho contrato que sin embargo no puede usarse en aquellos protocolos donde se requiera m

Capacidades De JavaScript En El Modelo De Objetos

La Contextualización En La Composición

Los métodos del modelo de objetos se diferencian de las funciones en el uso de un parámetro implícito – this – que referencia al contexto donde dicho método debe ejecutarse. Dicho contexto puede ser alterado por diversos mecanismos explícitos o implícitos del lenguaje.



Puntero Implícito this

```
var o = {
  m: function () {...},
  n: function (){
    this.m();
  }
};
```

```
var m = function () {...};
var o = {
  n: function (){
    (function () {
      this.m();
    })();
  }
};
```

```
var o1 = {
  n: function (){
    this.m();
  }.bind(o2)
};
var o2 = {
  m: function () {...}
};
```

Recontextualización

Las capacidades de recontextualización que ofrece JavaScript permiten articular diversos esquemas de conexión compositiva que condicionan la variante funcional que se va a invocar en cada caso

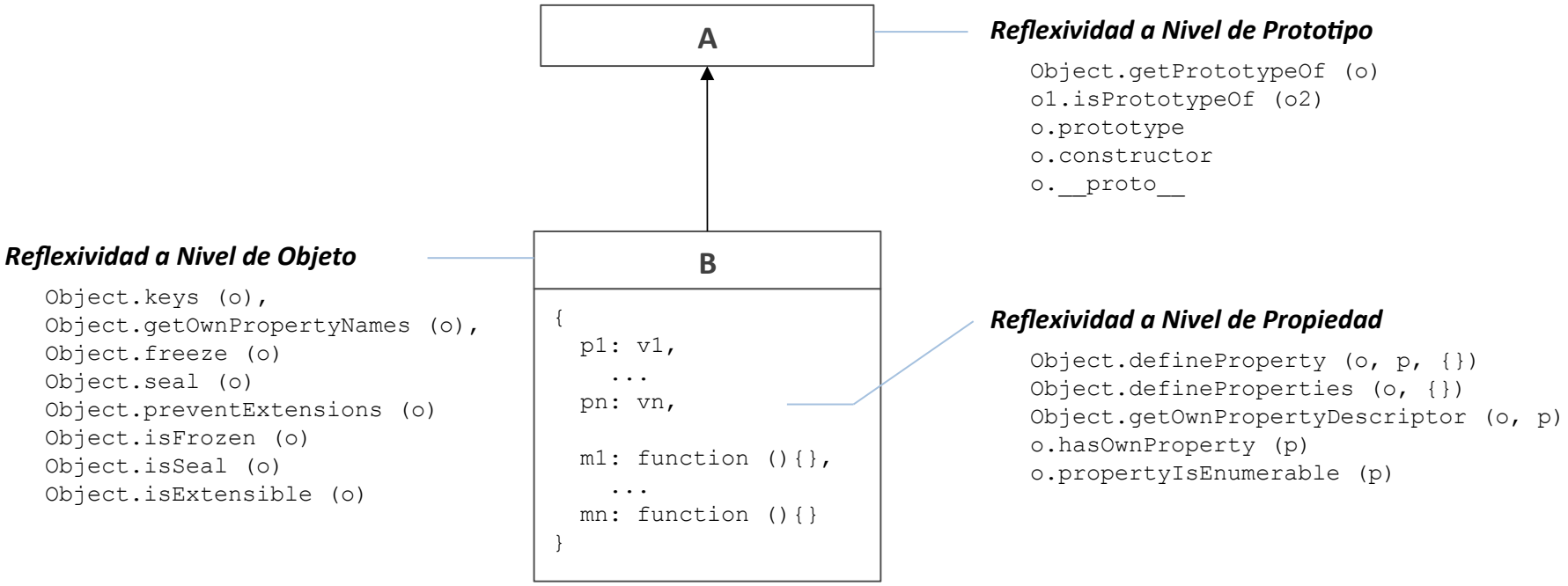
Contextualización

bind permite especificar que el método m debe ejecutarse en el contexto del objeto b. Esto tiene repercusiones acerca de a quien afecta el código dentro de m y permite articular composiciones interesantes

Capacidades De JavaScript En El Modelo De Objetos

La Reflexión En La Composición

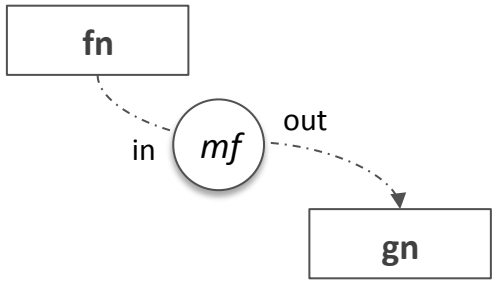
La mayoría de los metaprogramas hacen uno de las capacidades reflexivas del lenguaje. Una arquitectura reflexiva es aquella que ofrece al programador ciertos metadatos para permitirle descubrir las características de los elementos del entorno de ejecución. A continuación resumimos las capacidades esenciales de reflexión en JavaScript sobre el modelo de objetos.



Capacidades De JavaScript En El Modelo Funcional

El Orden Superior En La Composición

Los lenguajes que permiten recibir como parámetros de una función otra función se dice que operan en orden superior. Este es el caso de JavaScript y gracias a ello se pueden definir metaprogramas que transforman a otras funciones genéricas pasadas como parámetro.



Funciones paramétricas

El metaprograma mf recibe como parámetro una función cualquiera fn y la transforma en otra función gn con la misma esencia que la entrada

Orden Superior

```
function maybe (fn) {
  return function () {
    var args = [].slice.call(arguments);
    var callable = arguments.length >= fn.length &&
    args.every(function (p) {
       return (p !== null);
    });
    if (callable) return fn.apply(this, args);
  };
}
```

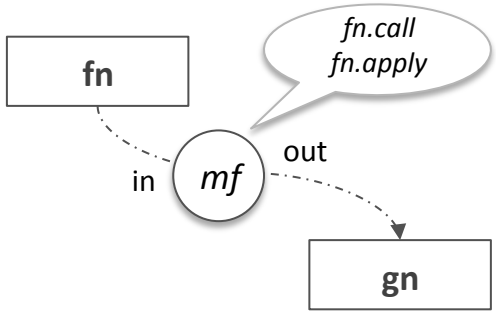
Orden Superior

El orden superior permite definir esquemas algorítmicos agnósticos de la lógica interna de la función genérica que se recibe como parámetro

Capacidades De JavaScript En El Modelo Funcional

La Invocación Dinámica En La Composición

JavaScript ofrece dos métodos equivalentes para invocar dinámicamente a una función como parte de un algoritmo de metaprogramación. Esto es especialmente potente si la función es pasada como parámetro como un ejercicio de orden superior.



Invocación mediante call o apply

La función recibida como parámetro en mf es invocada dinámicamente a través de los métodos call o apply para definir el metaprograma

Invocación Dinámica

```
function provided (pn) {
  return function (fn) {
    return function () {
      if (pn.apply(this, arguments))
       return fn.apply(this, arguments);
    };
  };
}
```

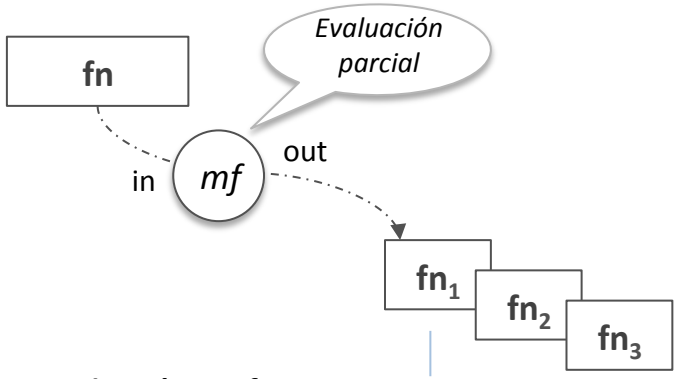
Invocación Dinámica

La invocación dinámica permite inyectar en esquemas de transformación la lógica pasada como parámetro

Capacidades De JavaScript En El Modelo Funcional

La Evaluación Parcial En La Composición

En JavaScript es posible retornar una función como resultado de otra función. Esto permite a los metaprogramas definir esquemas de transformación que evalúen parte de los parámetros formales de una función de entrada y generen una función a la salida con el resto de parámetros pendientes de evaluar.



Evaluación por fases

La función recibida como parámetro en mf transforma la función de entrada fn de 3 parámetros para generar otra función a la salida donde cada parámetro se evalúa en fases consecutivas de invocación. A este proceso se le llama currificación

Evaluación Parcial

Proveedor

```
function first () {
   var fn = arguments [0];
   var params = [].slice.call(arguments, 1);
   return function () {
      var args = [].slice.call(arguments);
      return fn.apply(this, params.concat(args));
   };
}
```

Cliente

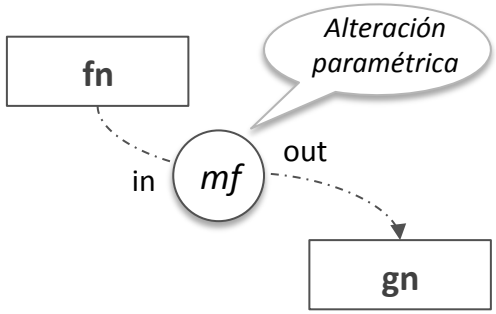
Evaluación Parcial

La evaluación parcial permite definir esquemas donde la evaluación de parámetros se distribuye entre el proveedor y el cliente de manera que este último dispone de menos grados de liberad para operar y lo hace sobre un esquema de configuración establecido por el proveedor

Capacidades De JavaScript En El Modelo Funcional

La Intervención Paramétrica En La Composición

De forma similar, los métodos call y apply también permiten intervenir discrecionalmente en el valor que se proporcionará a las funciones invocadas dinámicamente así como el contexto this en el que dicha invocación debe evaluarse.



Los parámetros en fn se alteran

El metaprograma mf tiene la oportunidad de cambiar o alterar los parámetros actuales que recibirá la función invocada fn

Intervención Paramétrica

```
function arity (n) {
  return function (fn) {
    return function () {
      var args = [].slice.call(arguments, 0, n-1);
      return fn.apply(this, args);
    };
  };
}
```

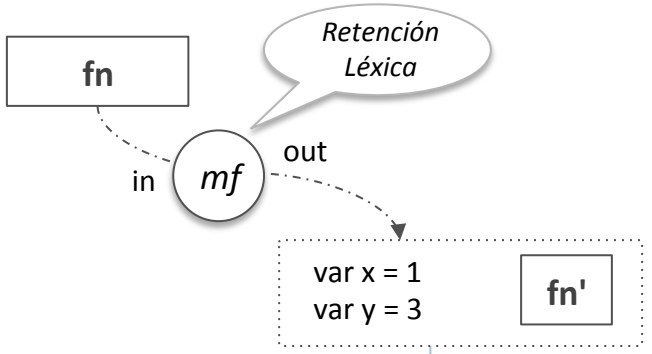
Intervención Paramétrica

La invocación dinámica permite además recontextualizar la evaluación de una función alterando el valor de this, que en otros lenguajes se considera un puntero constante. Este es un ejemplo de conexión entre el modelo de objetos y el funcional

Capacidades De JavaScript En El Modelo Funcional

La Retención Léxica En La Composición

Cada función retornada por un metaprograma mantiene por construcción referencia al ámbito léxico donde se define. Esto permite a los metaprogramas condicionar la evaluación de una función sobre un espacio de variables retenidas en dicho ámbito léxico.



Distintos entornos de retención

El metaprograma mf genera un esquema algorítmico que incluye a fn donde las variables libres, que no son parámetros formales ni variables locales de fn, se evalúan en el contexto léxico definido dentro de mf

Retención Léxica

```
function arity (db) {
  return function () {
     return {
       load: function (id) { return db.find (id); },
       save: function (id, o) { db.save (id, o); }
     };
  };
}
```

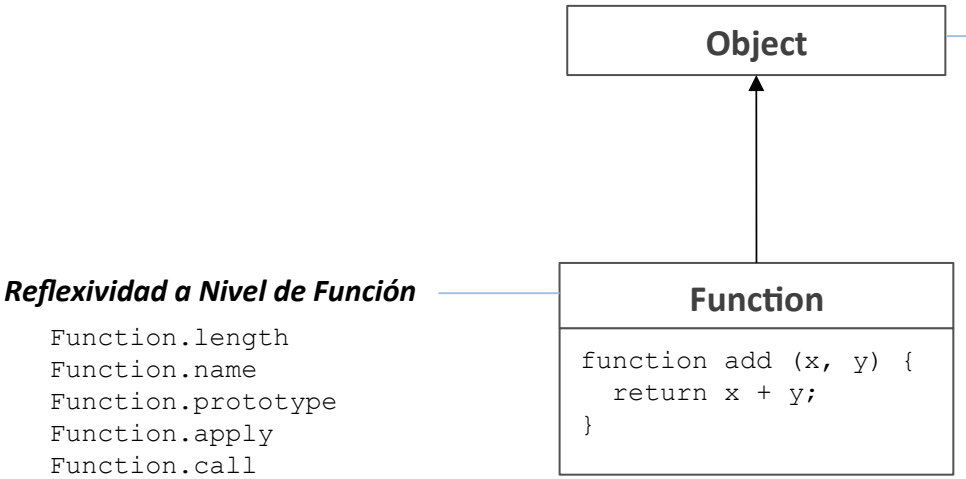
Retención Léxica

La retención léxica ofrece la oportunidad al metaprograma de caracterizar esquemas de comportamiento funcional que dependan de los valores configurados en las variables retenidas

Capacidades De JavaScript En El Modelo Funcional

La Reflexión En La Composición

De manera similar a como estudiábamos en el modelo de objetos, el espacio de funciones de JavaScript también proporciona una considerablemente rica arquitectura reflexiva que permite conocer ciertas propiedades y metadatos de los objetos función. Esto no sólo permite articular reflexividad sino introspección sobre funciones ajenas.



Reflexividad a Nivel de Objeto

Toda función es en realidad un tipo de objeto JavaScript que dispone de atributos y métodos miembro para caracterizar una función. Esto permite operar con las funciones sobre el modelo de reflexión de objetos de JavaScript

Reflexividad a Nivel de Función

```
Function.length
Function.name
Function.prototype
Function.apply
Function.call
Function.bind
Function.toString
```

Javier Vélez Reyes
@javiervelezreye
Javier.velez.reyes@gmail.com

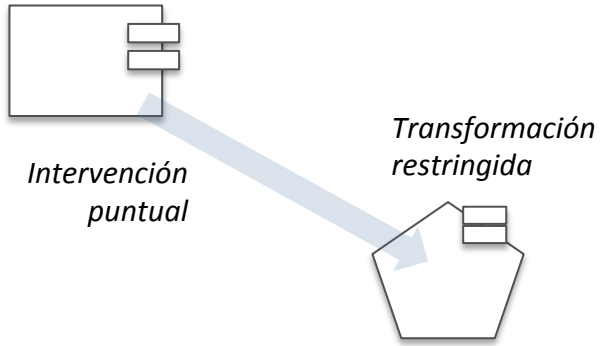
3 Técnicas De Metaprogramación Compositiva

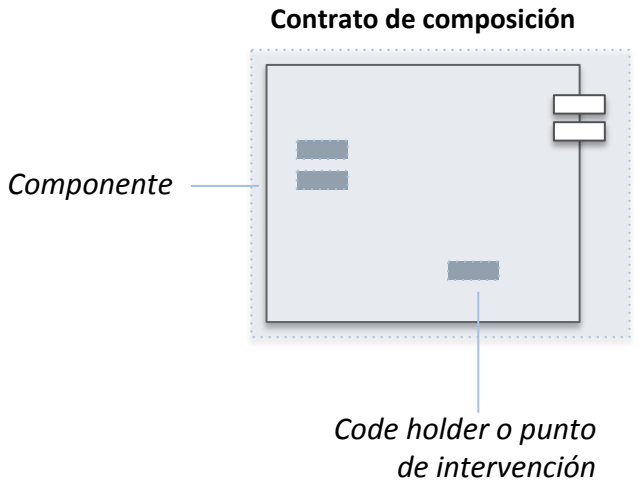
- Técnicas de Adición Compositiva
- Técnicas de Extensión Compositiva
- Técnicas de Intercesión Compositiva
- Técnicas de Delegación Compositiva

Introducción

La Metaprogramación Compositiva Como Intervención

La metaprogramación compositiva se caracteriza por el hecho de que mantiene la estructura esencial de los componentes. Todo ejercicio metaprogramático en este sentido se reduce a un conjunto de actividades de intervención puntual que pretenden adaptar, conectar o contextualizar código de los componentes.





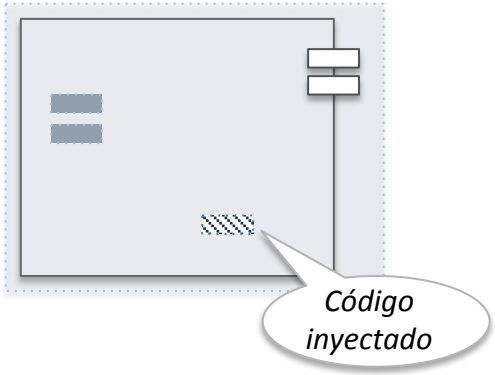
Code Holders & Contratos De Composición

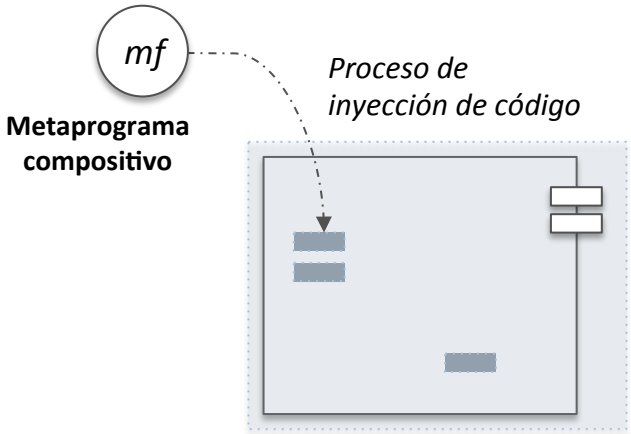
El conjunto potencial de intervenciones sobre un modelo de componentes queda impuesto por las capacidades metaprogramáticas del lenguaje y el modelo seleccionado. A lo largo de este capítulo nos centraremos en objetos aunque es fácil hacer una adaptación de los metaprogramas para que operen sobre las álgebras funcionales. Diremos que cada punto de intervención dentro de un componente es un code holder mientras que el conjunto de todos ellos conforman el contrato de composición sobre modelo el componente.

Introducción

Inyección De Código & Primitivas De Composición

Sobre los puntos de intervención definidos por cada uno de los code holders es posible realizar actividades de metaprogramación relacionadas con la inyección de código. Las posibles variantes de inyección de código se describen a partir de la definición de un conjunto de operadores o primitivas de composición propias de cada tipo de code holder.





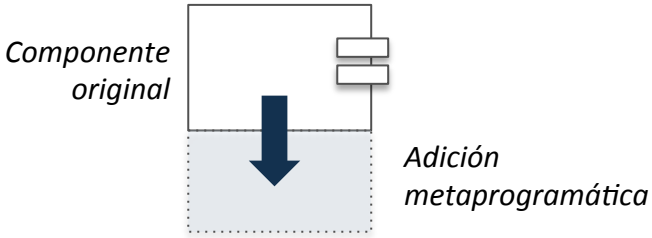
Metaprogramas & Intervención

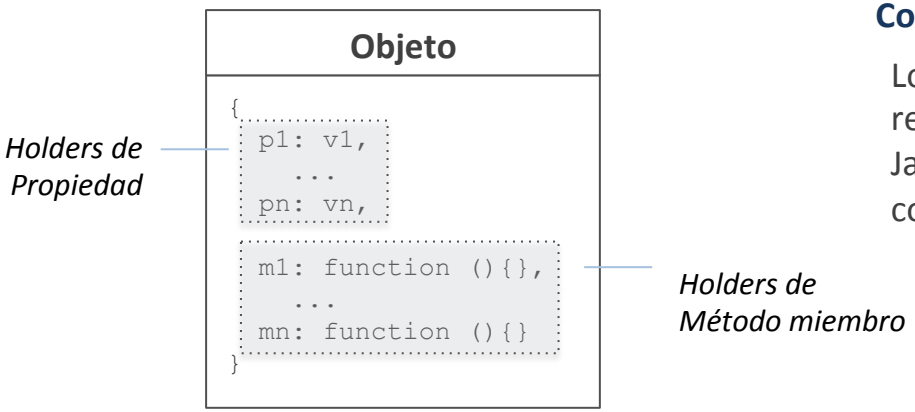
Los programas encargados de llevar a cabo el proceso de inyección de código sobre cada uno de los code holders del modelo de componentes se llaman metaprogramas. Estos algoritmos utilizan los operadores de composición para garantizar que los procesos de intervención son conformes con los diferentes tipos de code holders. A lo largo de este capítulo describiremos los code holders en JavaScript sobre el modelo de objetos y las operación de composición vinculadas a ellos.

Técnicas De Metaprogramación Por Adición

Definición

Las técnicas aditivas persiguen ampliar la lógica de un componente con nueva funcionalidad. De esta manera el componente se adapta a nuevos contextos arquitectónicos donde la lógica adicionada es requerida.





Code Holders

Los puntos de intervención en este caso reflejan el carácter abierto de los objetos en JavaScript con lo que es posible alterar el conjunto de características de los mismos.

Técnicas De Metaprogramación Por Adición

Operadores De Composición Por Adición

Los operadores de composición fundamentales que hacen uso de técnicas aditivas permiten añadir, borrar, actualizar y renombrar las características – métodos y propiedades – de un objeto.

```
mp.add = function add (core, key, value) {
    core[key] = value;
    return core;
};

mp.remove = function remove (core, key) {
    delete core[key];
    return core;
};

mp.update = function update (core, key, value) {
    if (key in core)
        mp.add (core, key, value);
    return core;
};

mp.rename = function rename (core, oKey, nKey) {
    if (oKey in core && oKey !== nKey) {
       mp.add (core, nKey, core[oKey]);
       mp.remove (core, oKey);
    } return core;
};
```

Operadores fundamentales de Adición

Los operadores fundamentales se centran en operar sobre el conjunto de características de un componente que funciona como núcleo.

Operador	o
`mp.add (o, 'x', 1);`	`{}`
`mp.remove (o, 'x')`	`{ x: 1 }`
`mp.update (o, 'x', 1);`	`{}`
`mp.add (o, 'x', 1);`	`{}`
`mp.rename (o, 'x', 'y');`	`{ x: 1 }`
`mp.update (o, 'x', 5);`	`{ y: 1 }`
	`{ y: 5 }`

Técnicas De Metaprogramación Por Adición

Operadores De Composición Por Adición

Como extensión a los operadores de adición se pueden incluir primitivas que permiten copiar y mover propiedades de un objeto a otro o extender un objeto core con las capacidades de otro como extensión.

```
mp.copy = function copy (core, ext, key) {
    if (key in ext)
      mp.add (core, key, ext[key]);
    return core;
};

mp.move = function move (core, ext, key) {
    mp.copy (core, ext, key);
    mp.remove (core, key);
    return core;
};

mp.extend = function extend (core, ext, ctx) {
    var keys = Object.getOwnPropertyNames (ext);
    keys.forEach (function (key) {
        mp.copy (core, ext, key);
        mp.bind (core, key, ctx || core);
    });
    return core;
};
```

Otros operadores de Adición

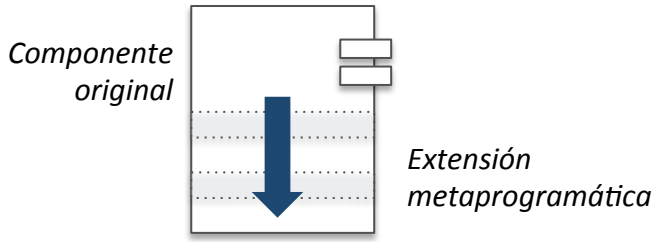
Los operadores fundamentales se complementan con otro conjunto de operadores que implican un núcleo y una extensión

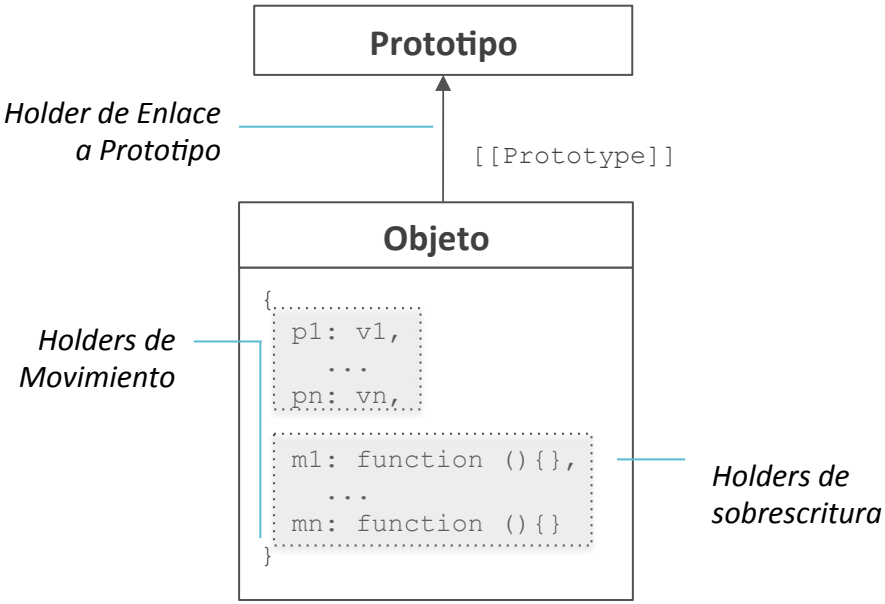
Operador	o1 o2
	{ y: 5 } {}
mp.copy (o2, o1, 'y');	{ y: 5 } { y: 5 }
mp.remove (o2, 'y');	{ y: 5 } {}
mp.move (o2, o1, 'x');	{} { y: 5 }
	{x: 0} {y: 0}
mp.extend (o1, o2);	{x: 0, y: 0} {y: 0}

Técnicas De Metaprogramación Por Extensión

Definición

Las técnicas extensivas tienen por objeto reformular las capacidades funcionales del componente por medio de la especialización semántica. Este tipo de técnicas se encuentra estrechamente vinculado a la herencia por prototipos como mecanismo del lenguaje.





Code Holders

Sobre una jerarquía de herencia es posible alterar la posición relativa de los elementos o reescribir la semántica de los métodos miembros del hijo. Además se puede operar sobre los enlaces de la cadena de proto-tipado.

Técnicas De Metaprogramación Por Extensión

Operadores De Composición Por Extensión

Los operadores fundamentales que hacen uso de técnicas extensivas están relacionados con la alteración de la cadena de prototipado. Es posible añadir, borrar o invertir el sentido de esta relación.

```
mp.getPrototype = function (core) {
    return Object.getPrototypeOf (core);
};

mp.setPrototype = function (core, proto) {
    var ch = Object.create (proto);
    mp.extend (ch, core);
    return ch;
};

mp.removePrototype = function (core) {
    return mp.setPrototype (core, null);
};

mp.reversePrototype = function (core) {
    var proto = mp.getPrototype (core);
    return mp.setPrototype (proto, core);
};
```

Operadores fundamentales de Extensión

Mediante estrategias reconstructivas es posible cambiar dinámicamente el valor de la relación de prototipado entre objetos

Operador	o p
mp.setPrototype(o, p);	o -> p
mp.removePrototype (o);	o
mp.reversePrototype (o);	o <- p

Técnicas De Metaprogramación Por Extensión

Operadores De Composición Por Extensión

Los operadores de copia y movimiento por metaprogramación permiten mover y copiar ascendente y descendentemente características presentes entre un prototipo y cada uno de sus objetos hijos.

```
mp.copyUp = function (core, key) {
    var proto = mp.getPrototype (core);
    mp.copy (proto, core, key);
    return core;
};

mp.copyDown = function (core, key, child) {
    var proto = mp.getPrototype (core);
    mp.copy (core, proto, key);
    return core;
};

mp.moveUp = function (core, key) {
    var proto = mp.getPrototype (core);
    mp.move (proto, core, key);
    return core;
};

mp.moveDown = function (core, key, child) {
    var proto = mp.getPrototype (core);
    mp.move (core, proto, key);
    return core;
};
```

Operadores fundamentales de Extensión

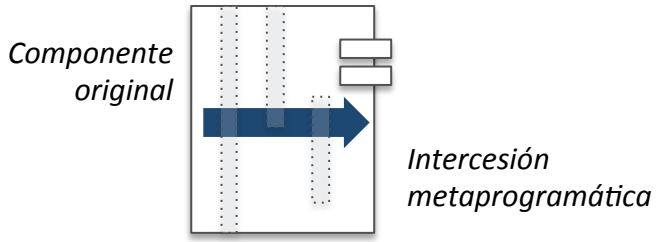
Las operaciones de copia y movimiento de características por la cadena de prototipado permiten alterar la semántica definida por la relación jerárquica entre objetos

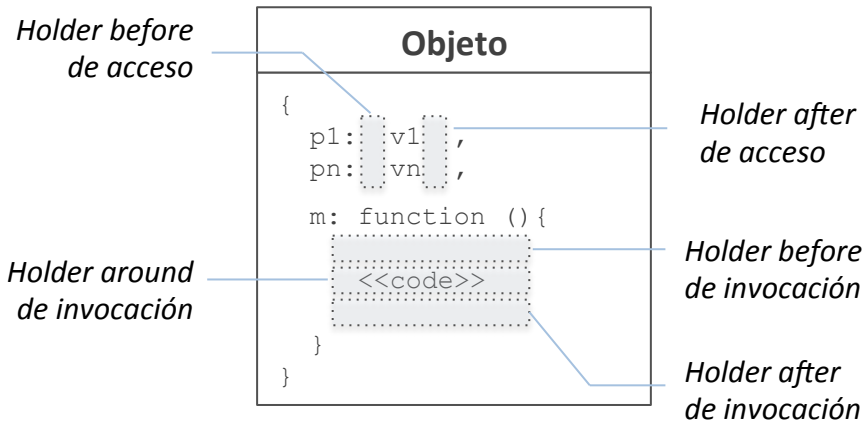
Operador	o p
`mp.copyUp (o, 'x');`	`o{x} -> p, o{x} -> p{x}`
`mp.copyDown (p, 'x', o);`	`o -> p{x}, o{x} -> p{x}`
`mp.moveUp (o, 'x');`	`o{x} -> p, o -> p{x}`
`mp.moveDown (p, 'x', o);`	`o -> p{x}, o{x} -> p`

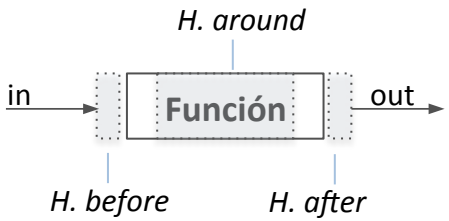
Técnicas De Metaprogramación Por Intercesión

Definición

La intercesión permite inyectar lógica de decoración de manera entrelazada con el código original del componente. El propósito de este tipo de técnicas es ampliar la semántica del mismo sin extender su contrato.







Code Holders

En cuanto a propiedades se puede inyectar código para que se ejecute antes y después de acceder a las mismas tanto para su lectura como para su escritura. En los métodos y funciones, existe similarmente decoración anterior y posterior a la invocación, pero además también es posible inyectar código que se ejecutará cuando el método explícitamente lo demande.

Técnicas De Metaprogramación Por Intercesión

Operadores De Composición Por Intercesión

Los operadores de intercesión fundamentales permiten decorar la invocación de métodos inyectando lógica funcional que se ejecuta antes o después de la activación del método.

```
mp.before = function before (core, key, ext) {
    var fn = core[key];
    core[key] = function () {
        var args = [].slice.call (arguments);
        var out  = ext.apply (this, args);
        return fn.apply (this, args.concat (out));
    };
    return core;
};

mp.after = function after (core, key, ext) {
    var fn = core[key];
    core[key] = function () {
        var args = [].slice.call (arguments);
        var out = fn.apply (this, args);
        ext.apply (this, args.concat (out));
        return out;
    };
    return core;
};
```

Operadores fundamentales de Intercesión

Los operadores fundamentales de intercesión permiten inyectar lógica funcional que se ejecutará antes o después del método

Operador	○

`mp.before (o, 'f', g);`
`mp.after (o, 'f', g);`
	`{f: function() {...} }`
	`f, g`
	`g, f`

Técnicas De Metaprogramación Por Intercesión

Operadores De Composición Por Intercesión

Adicionalmente, y como una especialización de los operadores anteriores, se puede decorar un método en before para condicionar su ejecución en función del resultado emitido por una extensión de guarda.

```
mp.provided = function provided (core, key, ext) {
    var fn = core[key];
    core[key] = function () {
      if (ext.apply (this, arguments))
        return fn.apply (this, arguments);
    };
    return core;
};

mp.except = function except (core, key, ext) {
    var fn = core[key];
    core[key] = function () {
      if (!ext.apply (this, arguments))
         return fn.apply (this, arguments);
      };
    return core;
};
```

Operadores fundamentales de Intercesión

Los operadores de guarda utilizan técnicas de intercesión para condicionar la ejecución de un método en función del resultado emitido por cierto predicado lógico

Operador	○

`mp.provided (o, 'f', p);`
`mp.except (o, 'f', p);`
	`{f: function() {...} }`
	`f sii p es true`
	`f sii p es false`

Técnicas De Metaprogramación Por Intercesión

Operadores De Composición Por Intercesión

Asimismo es posible conceder a un método el control acerca de cuando se ejecutará cierta decoración proporcionada por el cliente en el contexto de su esquema algorítmico que es recibido como parámetro en la función original.

```
mp.around = function around (core, key, ext) {
    var fn = core[key];
    core[key] = function () {
      var args = [].slice.call(arguments);
      args = [ext.bind(this)].concat(args);
      return fn.apply (this, args);
    };
    return core;
};
```

La extensión, g, se antepone como primer parámetro a los argumentos de la función decorada f

Operador explícito de Intercesión

En este caso el código core, f, mantiene referencias explícitas a una función de decoración, r, pasada por convenio como primer argumento

Operador ○

```
{f: function(r) {
    r(); ... r ();
 }
}
```

```
mp.around (o, 'f', g);
```

```
g, ..., g
```

En ejecución, en el esquema de decoración se sustituye r por g

Técnicas De Metaprogramación Por Intercesión

Operadores De Composición Por Intercesión

Los operadores de acceso a propiedades para lectura y escritura en before permiten inyectar código que se ejecutará antes del acceso a las propiedades.

```
mp.beforeGetAtt = function (core, key, ext) {
    Object.defineProperty (core, '_' + key, {
        value: core[key],
        writable: true,
        enumerable: false,
        configurable: false
    });

    Object.defineProperty (core, key, {
        get: function () {
            ext.call (this, core['_' + key]);
            return core['_' + key];
        }
    });
    return core;
};
```

Operador explícito de Intercesión

Se decoran los métodos de acceso y modificación a propiedades get y set y se mantiene una variable oculta para soportar el estado de la propiedad.

Operador	o
`mp.beforeGetAtt (o, 'x', f)`	`{x: 0}` `o.x -> f, 0`

Primero se define una propiedad privada de igual nombre que la clave key pero prefijando un _

Esta propiedad privada permite dar soporte a la implementación de intercesor de lectura get

Técnicas De Metaprogramación Por Intercesión

Operadores De Composición Por Intercesión

Los operadores de acceso a propiedades para lectura y escritura en before permiten inyectar código que se ejecutará antes del acceso a las propiedades.

```
mp.beforeSetAtt = function (core, key, ext) {
    Object.defineProperty (core, '_' + key, {
        value: core[key],
        writable: true,
        enumerable: false,
        configurable: false
    });

    Object.defineProperty (core, key, {
        set: function (v) {
            ext.call (this, core['_' + key], v);
            core['_' + key] = v;
        }
    });
    return core;
};
```

Operador explícito de Intercesión

Se decoran los métodos de acceso y modificación a propiedades get y set y se mantiene una variable oculta para soportar el estado de la propiedad.

Operador	o
`mp.beforeSetAtt (o, 'x', f)`	`{x: 0}` `o.x = 3 -> f, {x: 3}`

Primero se define una propiedad privada de igual nombre que la clave key pero prefijando un _

Esta propiedad privada permite dar soporte a la implementación de intercesor de escritura set

Técnicas De Metaprogramación Por Intercesión

Operadores De Composición Por Intercesión

Similarmente, los operadores de acceso a propiedades para lectura y escritura en after permiten inyectar código que se ejecutará después del acceso a las propiedades.

```
mp.afterGetAtt = function (core, key, ext) {

    Object.defineProperty (core, '_' + key, {
        value: core[key],
        writable: true,
        enumerable: false,
        configurable: false
    });

    Object.defineProperty (core, key, {
        get: function () {
            var out = core['_' + key];
            ext.call (this, core['_' + key]);
            return out;
        }
    });
    return core;
};
```

De manera similar, primero se define una propiedad privada de igual nombre que la clave key pero prefijando un _ para dar soporte a la implementación del get en after

Operador explícito de Intercesión

Se decoran los métodos de acceso y modificación a propiedades get y set y se mantiene una variable oculta para soportar el estado de la propiedad.

Operador	o
`mp.afterGetAtt (o, 'x', f)`	`{x: 0}` `o.x -> 0, f`

Técnicas De Metaprogramación Por Intercesión

Operadores De Composición Por Intercesión

Similarmente, los operadores de acceso a propiedades para lectura y escritura en after permiten inyectar código que se ejecutará después del acceso a las propiedades.

```
mp.afterSetAtt = function (core, key, ext) {

    Object.defineProperty (core, '_' + key, {
        value: core[key],
        writable: true,
        enumerable: false,
        configurable: false
    });

    Object.defineProperty (core, key, {
        set: function (nv) {
            var ov = core['_' + key];
            core['_' + key] = nv;
            ext.call (this, ov, nv);
        }
    });
    return core;
};
```

De manera similar, primero se define una propiedad privada de igual nombre que la clave key pero prefijando un _ para dar soporte a la implementación del set en after

Operador explícito de Intercesión

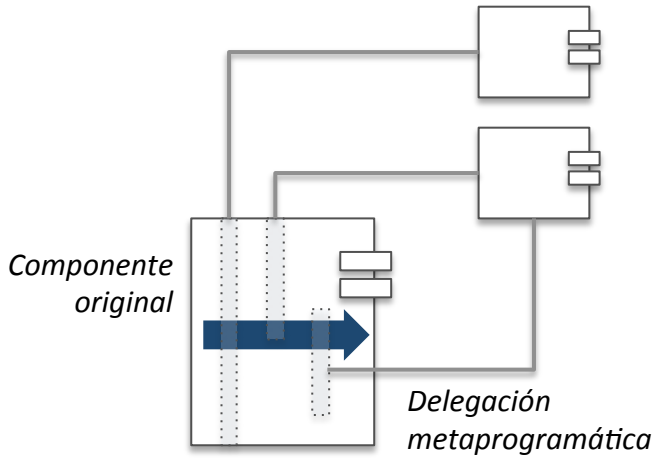
Se decoran los métodos de acceso y modificación a propiedades get y set y se mantiene una variable oculta para soportar el estado de la propiedad.

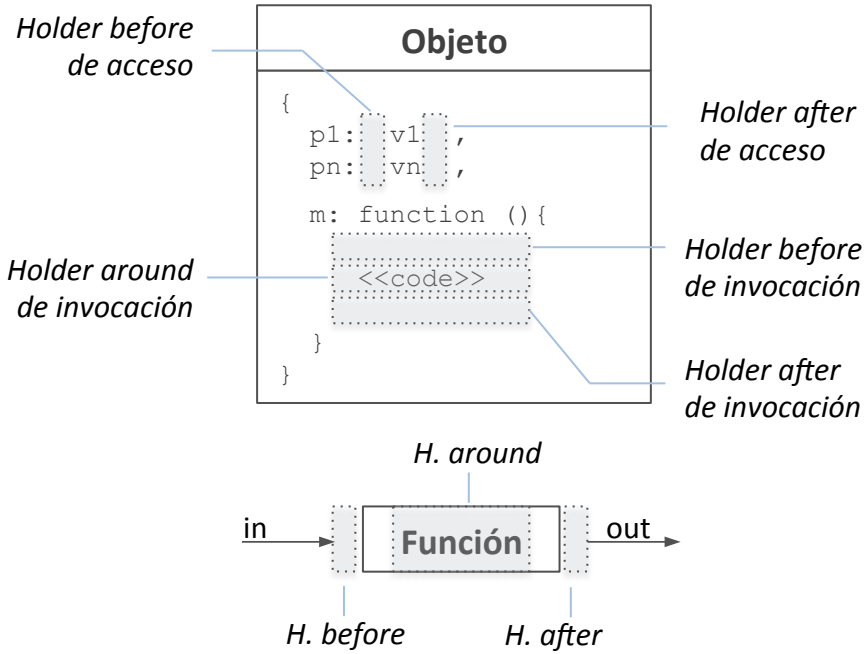
Operador	o
`mp.afterSetAtt (o, 'x', f)`	`{x: 0}` `o.x = 3 -> {x: 3}, f`

Técnicas De Metaprogramación Por Delegación

Definición

Las técnicas basadas en delegación articulan estrategias compositivas que permiten dispersar la lógica de un servicio entre varios componentes con identidad propia dentro de la arquitectura pero estableciendo fachadas de desacoplamiento.





Code Holders

Los puntos de intervención en esta familia de técnicas coinciden con los de la intercesión. La diferencia estriba en este caso en que los metaprogramas inyectan código dirigido a establecer esquemas que delegan en otra lógica.

Técnicas De Metaprogramación Por Delegación

Operadores De Composición Por Delegación

Este tipo de operadores articulan procesos de composición por delegación interna y externa mediante la construcción de métodos proxy añadidos al core.

```
mp.innerDelegate = function (core, oKey, nKey, ctx) {
    var context = ctx || core;
    core[nKey] = function () {
        var args = [].slice.call (arguments);
        var out  = core[oKey].apply (context, args);
        return out === core ? this : out;
    };
    return core;
};

mp.outerDelegate = function (core, ext, key, ctx) {
    var context = ctx || ext;
    core[key] = function () {
        var args = [].slice.call (arguments);
        var out  = ext[key].apply (context, args);
        return out === ext ? this : out;
    };
    return core;
};
```

Operador fundamentales de Delegación

La delegación interna permite generar un método de proxy que delega en otro método del mismo objeto. La delegación externa se produce entre objetos diferentes

Operador	o1 [o2]
`mp.innerDelegate(o1,'f','g')`	`{f}` `{g:function(){o1.f()}` `f}`
`mp.outerDelegate(o1,o2,'f')`	`{}` `{f}` `{f:function(){o2.f()}}` `{f}`

Técnicas De Metaprogramación Por Delegación

Operadores De Composición Por Delegación

A partir de los operadores anteriores se definen los operadores de forwarding y proxy en forwarding que contextualizan la invocación en el delegado.

```
mp.forward = function (core, ext, key) {
    return mp.outerDelegate (core, ext, key, ext);
};

mp.forwardProxy = function (core, ext) {
    if (typeof(ext) === 'string') {
        ext = core[ext];
    }
    if (ext) {
        var keys = Object.keys (ext);
        keys.forEach (function (key) {
            if (typeof (ext[key]) === 'function')
                mp.forward (core, ext, key);
        });
    }
    return core;
};
```

Operador fundamentales de Delegación

El forwarding es una modalidad de delegación débil contextualizada en el delegado. Es lo que se conocer por delegación convencionan en los lenguajes de OOP

Operador	o1 [o2]
mp.forward (o1,o2,'f')	{x:1}{x:3, f(){return this.x}}
o1.f()	3

Técnicas De Metaprogramación Por Delegación

Operadores De Composición Por Delegación

Similarmente, los operadores fundamentales pueden utilizarse para articular técnicas de delegación contextual-lizadas en el core.

```
mp.delegate = function (core, ext, key) {
    mp.outerDelegate (core, ext, key, core);
};

mp.delegateProxy = function (core, ext) {
    if (typeof(ext) === 'string') {
        ext = core[ext];
    }
    if (ext) {
        var keys = Object.keys (ext);
        keys.forEach (function (key) {
            if (typeof (ext[key]) === 'function')
                mp.delegate (core, ext, key);
        });
    }
};
```

Operador fundamentales de Delegación

La delegación es la forma más fuerte y dinámica de vinculación compositiva. En ella el contexto de evaluación permanece en el objeto core

Operador	o1 [o2]
mp.delegate (o1,o2,'f')	{x:1}{x:3, f(){return this.x}}
o1.f()	1

Javier Vélez Reyes
@javiervelezreye
Javier.velez.reyes@gmail.com

4 Patrones De Metaprogramación Compositiva

- Patrones de Selección
- Patrones de Adaptación
- Patrones de Composición
- Patrones de Encapsulación
- Patrones de Contextualización

Introducción

Patrones Basados En Técnicas Compositivas

A continuación presentamos un breve catálogo de patrones de meta programación que hacen uso de las técnicas vistas en este texto. El catálogo distribuye los patrones en relación a la fase del proceso de composición donde se aplican. El objetivo no es hacer una revisión exhaustiva de patrones sino servir de base de ejemplo para la operativa diaria en técnicas de metaprogramación. De hecho, muchos de los patrones aquí presentados son una adaptación metaprogramática de patrones clásicos de la orientación a objetos.

Selección

- *Singleton*
- *Clone*
- *Recruiter*
- *Builder*
- *Locator*

Adaptación

- *Rename*
- *Observe*
- *Dummy*
- *Idempotent*
- *Delay*
- *Throttle*

Composición

- *Broadcast*
- *Balancer*
- *Dispatcher*
- *Filter*
- *Chain*
- *Fluent*

Encapsulación

- *Hide*
- *Self*
- *Hungarian*
- *Prototify*
- *Proxify*

Contextualización

- *Record*
- *Sink*
- *Trace*
- *Cache*
- *Undo*

Patrones de Selección

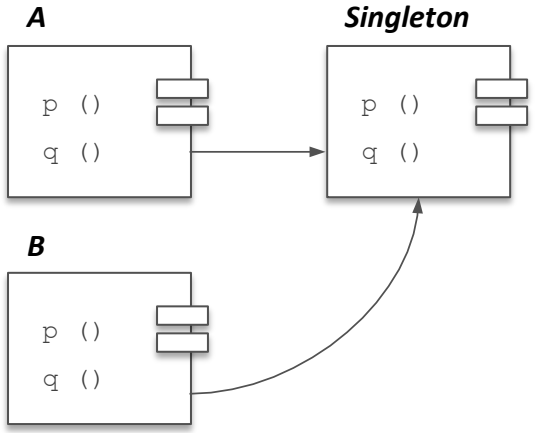
Los patrones de selección persiguen definir estrategias de localización, construcción y recuperaciones de componentes para emplearlos en el contexto de aplicación en curso. La lógica de estos patrones permite encontrar los componentes adecuados en virtud de las condiciones ambientales.

Singleton

Este patrón es una adaptación del clásico patrón singleton desarrollada a partir de técnicas de metaprogramación. Se construye un objeto con todos los métodos del original que delega en éste.

```
function singleton (obj) {
  return function () {
    return mp.forwardProxy ({}, obj);
  };
}
```

El metaprograma utiliza la operación de proxy de forwarding para obtener objetos que hacen forwarding al objeto único



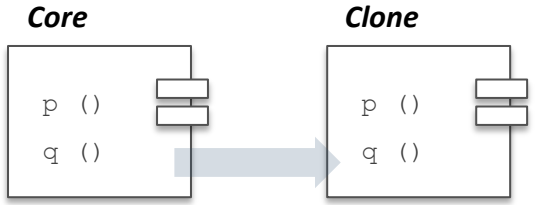
Los objetos A y B se construyen dinámicamente para delegar en un único componente de manera que la lógica corresponde a un patrón singleton

Patrones de Selección

Los patrones de selección persiguen definir estrategias de localización, construcción y recuperaciones de componentes para emplearlos en el contexto de aplicación en curso. La lógica de estos patrones permite encontrar los componentes adecuados en virtud de las condiciones ambientales.

Clone

Se clona un objeto pasado como parámetro para preservar el estado del mismo. Este patrón es útil en componentes que operan entre niveles de una arquitectura de capas.



El patrón crea un clon de un objeto pasado como parámetro y lo devuelve como resultado

```
function clone (obj) {
   return function () {
      return mp.extend ({}, obj);
   };
}
```

El metaprograma consiste esencialmente en invocar el operador de extensión sobre un objeto vacío que se convierte en el resultado una vez recibe los métodos y atributos del parámetro o

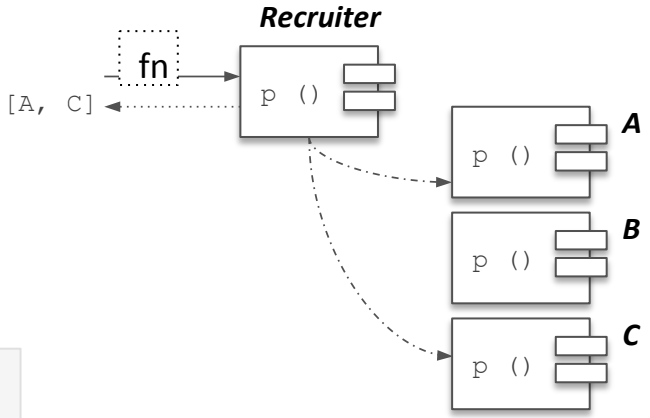
Patrones de Selección

Los patrones de selección persiguen definir estrategias de localización, construcción y recuperaciones de componentes para emplearlos en el contexto de aplicación en curso. La lógica de estos patrones permite encontrar los componentes adecuados en virtud de las condiciones ambientales.

Recruiter

Los componentes reclutadores seleccionan de entre una colección de objetos a aquellos que cumplen las condiciones impuestas por una función de filtro.

```
function recruiter (fn) {
  return function (objs) {
     var out = {};
     Object.keys (objs[0]).forEach (function (k) {
        mp.add (out, k, function () {
           var args = [].slice.call (arguments);
           return objs.filter (function (obj) {
              return fn(obj[k].apply (this, args));
           });
        });
     });
     return out;
  };
}
```



El reclutador evalúa cada función y criba aquellos objetos que verifican una función de filtro pasada como parámetro. Todos los componentes [A, B, C] deben tener un contrato común

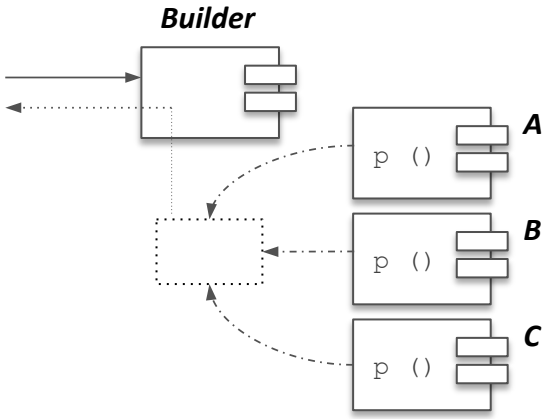
Patrones de Selección

Los patrones de selección persiguen definir estrategias de localización, construcción y recuperaciones de componentes para emplearlos en el contexto de aplicación en curso. La lógica de estos patrones permite encontrar los componentes adecuados en virtud de las condiciones ambientales.

Builder

Para construir objetos por partes podemos adaptar el patrón builder propio de la orientación a objetos clásica pero haciendo uso de la operación de extensión.

```
function builder (proto) {
   var out  = Object.create (proto || null);
   return {
     add : function (obj) {
       mp.extend (out, obj);
       return this;
     },
     create : function () { return out; }
   };
}
```



El patrón permite empezar por la definición opcional de un prototipo para el objeto y luego ir añadiendo nuevos objetos que se agregan como extensiones con lógica de sobrescritura

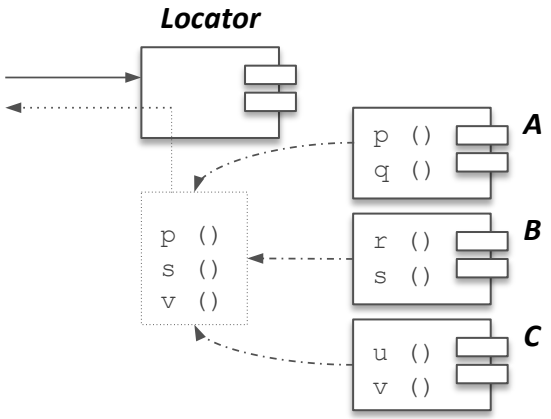
Patrones de Selección

Los patrones de selección persiguen definir estrategias de localización, construcción y recuperaciones de componentes para emplearlos en el contexto de aplicación en curso. La lógica de estos patrones permite encontrar los componentes adecuados en virtud de las condiciones ambientales.

Locator

Asimismo resulta interesante localizar una colección de métodos de una colección de componentes y conformar un nuevo componente incorporando dichos método debidamente contextualizados.

```
function locator (objs) {
  return function (keys) {
    var out = {};
    keys.forEach (function (k) {
      mp.add (out, k, objs.filter (function (obj) {
         return (k in obj);
      })[0][k]);
      mp.bind (out, k, out);
    });
    return out;
  };
}
```



El patrón permite empezar por la definición opcional de un prototipo para el objeto y luego ir añadiendo nuevos objetos que se agregan como extensiones con lógica de sobrescritura

Patrones de Adaptación

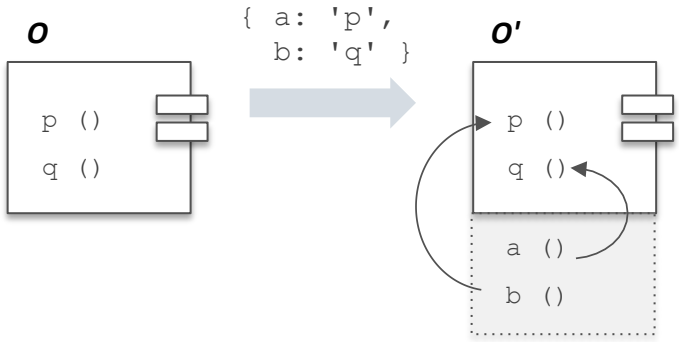
La adaptación es, tal vez, la fase del proceso de composición más relevante diferencialmente con respecto a aproximaciones anteriores. Las arquitecturas no son responsables de absorber la variabilidad sino que son los propios componentes los que se adaptan para alinearse a las restricciones arquitectónicas por medio de la aplicación de metaprogramas.

Alias

El patrones de alias adapta sintácticamente el contrato de un componente para que se ajuste a las necesidades impuestas por el contexto arquitectónico.

```
function alias (obj, trd) {
   Object.keys (trd).forEach (function (k) {
     mp.innerDelegate (obj, k, trd[k]);
   });
   return obj;
}
```

El metaprograma recoge la información de traducción en el objeto de configuración trd y utiliza operaciones de delegación interna para realizar la adaptación



Los nuevos métodos a y b incluidos en O descansan en una delegación interna sobre los métodos anteriores p y q. Esto permite encajar el componente en contextos que requieren a y b como operaciones directrices

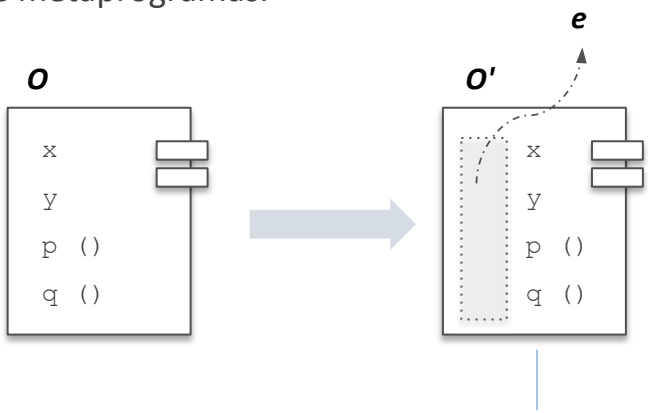
Patrones de Adaptación

La adaptación es, tal vez, la fase del proceso de composición más relevante diferencialmente con respecto a aproximaciones anteriores. Las arquitecturas no son responsables de absorber la variabilidad sino que son los propios componentes los que se adaptan para alinearse a las restricciones arquitectónicas por medio de la aplicación de metaprogramas.

Observe

Se transforman los métodos y propiedades de manera que su ejecución y acceso dispara un evento para escuchadores.

```
function observe (obj, emitter) {
   Object.keys(obj).forEach (function (k) {
     if (typeof (obj[k]) === 'function')
         mp.before (obj, k, function () {
            emitter.emit ('CALL');
         });
     else {
      mp.beforeAtt (obj, k, {
        get: function () { emitter.emit ('GET'); },
        set: function () { emitter.emit ('SET'); }
      });
     }
   });
   return obj;
}
```



Este patrón es una adaptación del clásico patrón observador-observable de la orientación a objetos utilizando técnicas metaprogramáticas

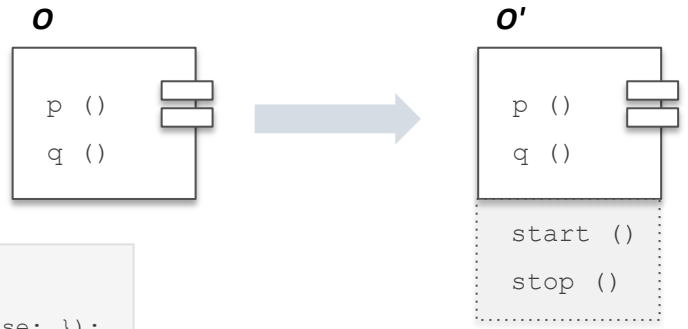
Se utilizan operadores de intercesión para decorar métodos y accesos de propiedad para disparar eventos

Patrones de Adaptación

La adaptación es, tal vez, la fase del proceso de composición más relevante diferencialmente con respecto a aproximaciones anteriores. Las arquitecturas no son responsables de absorber la variabilidad sino que son los propios componentes los que se adaptan para alinearse a las restricciones arquitectónicas por medio de la aplicación de metaprogramas.

Dummy

Un componente incluye un método de desconexión que anula la invocación de métodos. Este patrón resulta útil para entornos de pruebas.



La transformación incluyen métodos públicos para activar o desactivar el funcionamiento de los métodos originales del componente

```
function dummy (obj) {
   mp.add (obj, 'off', false);
   mp.add (obj, 'start', function () { this.off = false; });
   mp.add (obj, 'stop' , function () { this.off = true; });
   Object.keys(obj).forEach (function (k) {
      if (typeof (obj[k]) === 'function')
         mp.provided (obj, k, function () {
            return !this.off;
         });
   });
   return obj;
}
```

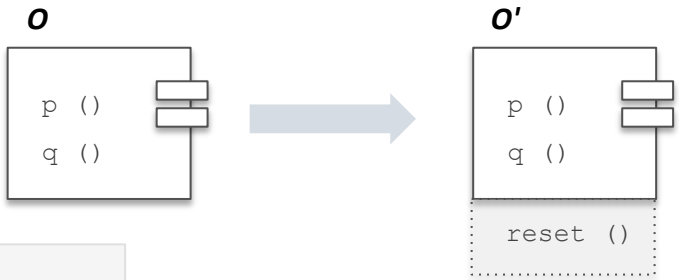
El metaprograma incluye métodos y variables de flag para soportar el estado de activación que se controla desde la intercesión de los métodos

Patrones de Adaptación

La adaptación es, tal vez, la fase del proceso de composición más relevante diferencialmente con respecto a aproximaciones anteriores. Las arquitecturas no son responsables de absorber la variabilidad sino que son los propios componentes los que se adaptan para alinearse a las restricciones arquitectónicas por medio de la aplicación de metaprogramas.

Idempotent

Un componente incluye un método de desconexión que anula la invocación de los demás métodos. Resulta útil en las capas de backend.




```
function idempotent (obj) {
    mp.add (obj, 'fired', {});
    mp.add (obj, 'reset', function () { this.fired = {}; });
    Object.keys(obj).forEach (function (k) {
        if (typeof (obj[k]) === 'function') {
            mp.after (obj, k, function () {
                this.fired[k] = true;
            });
            mp.except (obj, k, function () {
                return this.fired[k];
            });
        }
    });
    return obj;
}
```

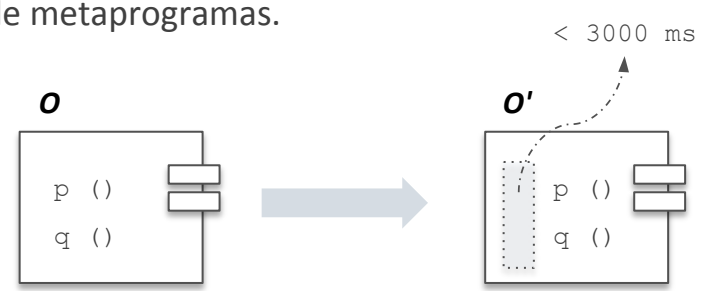
Se incluye estado para garantizar que cada método se ejecute una sola vez. El método reset() permite reconfigurar al componente para volver a su estado inicial

Patrones de Adaptación

La adaptación es, tal vez, la fase del proceso de composición más relevante diferencialmente con respecto a aproximaciones anteriores. Las arquitecturas no son responsables de absorber la variabilidad sino que son los propios componentes los que se adaptan para alinearse a las restricciones arquitectónicas por medio de la aplicación de metaprogramas.

Throttle

El patrón throttle transforma los métodos de un objetos de manera que se descarte toda invocación subsiguiente en un lapso de tiempo. Es especialmente útil para controlar manejadores vinculados a eventos demasiado frecuentes.



Se decora cada uno de los métodos del componente para garantizar que nuevas invocaciones serán descartadas en los subsiguientes instantes a cada invocación legal

```
function throttle (obj, ms) {
  Object.keys(obj).forEach (function (k) {
    mp.add (obj, k, (function (fn) {
      var throttling;
      return function () {
        if (!throttling) {
          throttling = setTimeout (function () {
            throttling = void 0;
          }, ms);
          return fn.call(obj, arguments);
        }
      };
    })(obj[k]));
  }); return obj;
}
```

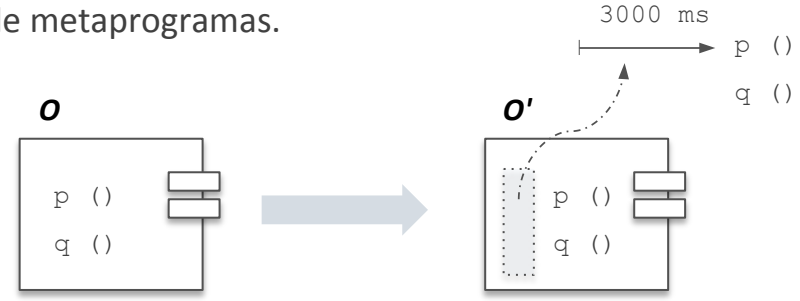
Se utiliza una intercesión mediante adición para vincular la llamada a cada método a la función de timeout de JavaScript y articular el descarte

Patrones de Adaptación

La adaptación es, tal vez, la fase del proceso de composición más relevante diferencialmente con respecto a aproximaciones anteriores. Las arquitecturas no son responsables de absorber la variabilidad sino que son los propios componentes los que se adaptan para alinearse a las restricciones arquitectónicas por medio de la aplicación de metaprogramas.

Delay

El patrón delay transforma los métodos de un componente en asíncronos para provocar un retardo de tiempo constante indicado por parámetro.



Se decora cada uno de los métodos del componente para garantizar que la invocación a cada método se retarda un intervalo de tiempo

```
function delay (obj, ms) {
  Object.keys(obj).forEach (function (k) {
    mp.add (obj, k, (function (fn) {
      return function () {
         var args = [].slice (arguments-1);
         var cb = arguments[arguments.length-1];
         setTimeout (function () {
           cb (fn.call(obj, args));
         }, ms);
      };
    })(obj[k]));
  });
  return obj;
}
```

Al igual que antes, se utiliza una intercesión mediante adición para vincular la llamada a cada método a la función de timeout de JavaScript y articular el retardo

Patrones de Composición

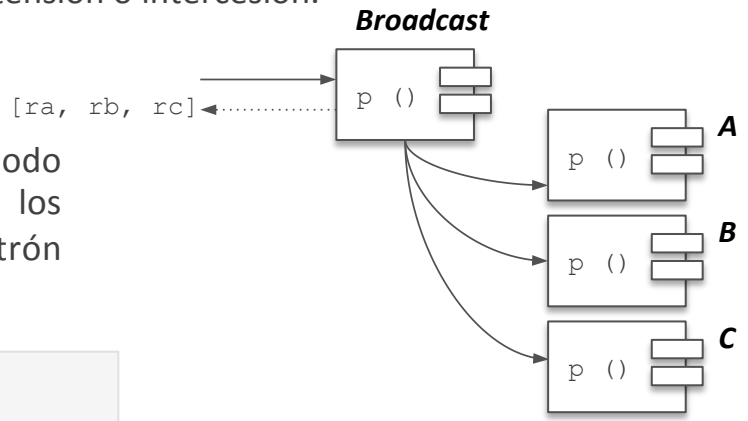
Las técnicas de composición permiten construir nuevos componentes que responden a situaciones específicas de colaboración entre componentes. A diferencia de basar esta composición en delegación como ocurren en orientación a objetos, con metaprogramación también se pueden usar estrategias aditivas de extensión o intercesión.

Broadcast

Genera un componente donde cada método realiza una delegación sobre cada uno de los componentes de una colección. Este patrón permite obtener respuesta de n proveedores.

```
function broadcast (objs) {
    var out = {};
    Object.keys(objs[0]).forEach (function (k) {
        mp.add (out, k, function () {
            return objs.map (function (obj) {
                return obj[k].call(obj, arguments);
            });
        });
    });
    return out;
}
```

Para definir el contrato del componente se analiza el primero de la colección asumiendo que todos tienen el mismo contrato



Se construye un componente de broadcast que delega en cada método de la colección [A, B, C] para obtener un array de resultados

Patrones de Composición

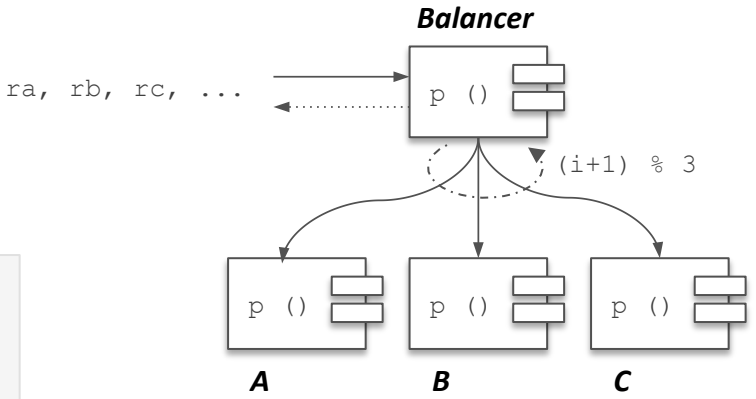
Las técnicas de composición permiten construir nuevos componentes que responden a situaciones específicas de colaboración entre componentes. A diferencia de basar esta composición en delegación como ocurren en orientación a objetos, con metaprogramación también se pueden usar estrategias aditivas de extensión o intercesión.

Balancer

Delega cada petición recibida en uno de los proveedores dispuestos en una colección de acuerdo a una política circular.

```
function balancer (objs) {
  var idx = 0;
  var out = {};
  Object.keys(objs[0]).forEach (function (k) {
    if (typeof (objs[0][k]) === 'function')
      mp.add (out, k, function () {
        var args = [].slice.call (arguments);
        var r = objs[idx][k].call (objs[idx], agrs);
        idx = (idx+1) % objs.length;
        return r;
      });
  });
  return out;
}
```

Cada método del balanceador delega alterativamente en el componente de índice idx



El balanceador se construye para delegar alternativamente entre la colección de proveedores [A, B, C] de acuerdo a una política circular de round robin

Patrones de Composición

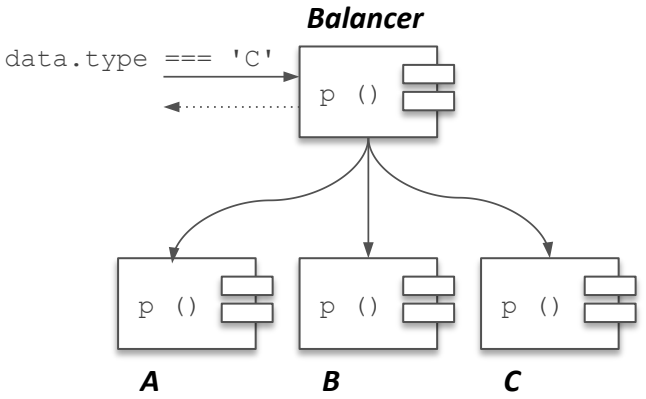
Las técnicas de composición permiten construir nuevos componentes que responden a situaciones específicas de colaboración entre componentes. A diferencia de basar esta composición en delegación como ocurren en orientación a objetos, con metaprogramación también se pueden usar estrategias aditivas de extensión o intercesión.

Dispatcher

El despachador dirige cada solicitud a un delegado distinto en función de la información de enrutamiento proporcionada como parámetro.

```
function router (key, routes) {
  return function (objs) {
     var out = {};
     Object.keys (objs[0]).forEach (function (k) {
        if (typeof (objs[0][k]) === 'function')
            mp.add (out, k, function (data) {
               var target = routes[data[key]];
               return target[k].call (target, data);
            });
     });
     return out;
  };
}
```

Se selecciona el componente objetivo en virtud del valor en data[key] y según la configuración en routes



El dispatcher enruta cada solicitud al proveedor adecuado en función del valor en un campo determinado del parámetro de entrada

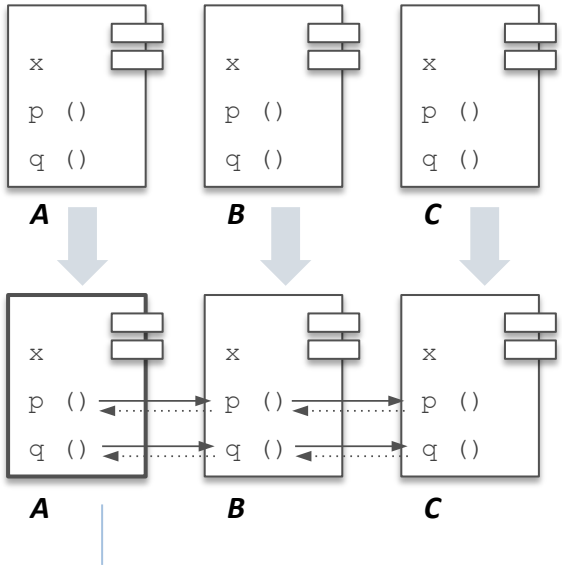
Patrones de Composición

Las técnicas de composición permiten construir nuevos componentes que responden a situaciones específicas de colaboración entre componentes. A diferencia de basar esta composición en delegación como ocurren en orientación a objetos, con metaprogramación también se pueden usar estrategias aditivas de extensión o intercesión.

Filter

El patrón de filtro encadena compositivamente los métodos de una colección de componentes con un contrato común.

```
function filter (objs) {
   objs.reverse ().forEach (function (obj, idx) {
     if (idx > 0) {
       Object.keys (obj).forEach (function (k) {
         if (typeof (obj[k]) === 'function') {
           var fn = obj[k];
           var gn = objs[idx-1][k];
           mp.add (obj, k, function (data) {
             var r = fn.call (this, data);
             return gn.call (this, r);
           });
         }
       }, this);
     }
   }, this);
   return objs.reverse()[0];
}
```



El filtro encadena una colección de componentes por composición para cada método del contrato común. Como consecuencia del encadenamiento se devuelve sólo A

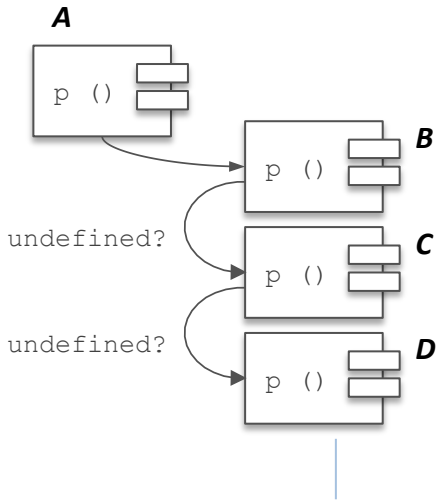
Patrones de Composición

Las técnicas de composición permiten construir nuevos componentes que responden a situaciones específicas de colaboración entre componentes. A diferencia de basar esta composición en delegación como ocurren en orientación a objetos, con metaprogramación también se pueden usar estrategias aditivas de extensión o intercesión.

Chain

La cadena de composición busca un componente que pueda resolver una solicitud. Cuando un componente no puede atender la solicitud devuelve undefined.

```
function filter (objs) {
   objs.reverse ().forEach (function (obj, idx) {
     if (idx > 0) {
       Object.keys (obj).forEach (function (k) {
         if (typeof (obj[k]) === 'function') {
           var fn = obj[k];
           var gn = objs[idx-1][k];
           mp.add (obj, k, function (data) {
             return fn.call (this, data) ||
                    gn.call (this, data);
           });
         }
       }, this);
     }
   }, this);
   return objs.reverse()[0];
}
```



La cadena de composición delega una solicitud a lo largo de una colección de componentes hasta que uno de ellos pueda atenderla (devolviendo un valor distinto de undefined)

Patrones de Composición

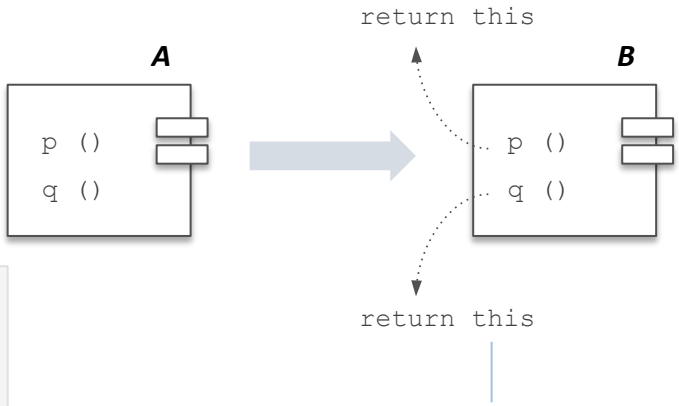
Las técnicas de composición permiten construir nuevos componentes que responden a situaciones específicas de colaboración entre componentes. A diferencia de basar esta composición en delegación como ocurren en orientación a objetos, con metaprogramación también se pueden usar estrategias aditivas de extensión o intercesión.

Fluent

Convierte los métodos de un componente en una API fluida para que puedan invocarse como una cadena de composición.

```
function fluent (obj, hn) {
        Object.keys (obj).forEach (function (k) {
            if (typeof (obj[k]) === 'function') {
                var fn = obj[k];
                mp.add (obj, k, function (data) {
                    hn (fn.call (this, data));
                    return this;
                });
            }
        }, this);
        return obj;
    }
```

Cada método ahora devuelve una referencia a this para que puedan encadenarse invocaciones sobre el propio componente



Cada método se intercede para que envíe los resultados a una función manejadora y devuelva this y convertir el estilo de invocación en API fluida

```
a.p(x);
a.p(y);
```

```
a
  .p(x)
  .p(y);
```

Patrones de Encapsulación

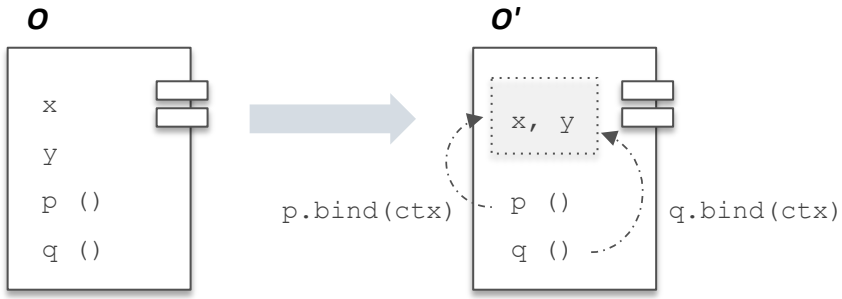
La falta de mecanismos nativos de encapsulación es una de las grandes críticas que se hace a JavaScript. Pese a que en orientación a objetos lo importante es en realidad respetar el principio de protección de la información, aquí presentamos una colección de patrones que ofrecen distintas alternativas para conseguir encapsulación de estado.

Hide

Genera un componente donde las propiedades se introducen en un contexto interno y los métodos se vinculan a dicho contexto.

```
function hide (obj) {
   var ctx = {};
   var out = {};
   Object.keys(obj).forEach (function (k) {
     if (typeof (obj[k]) === 'function') {
          mp.copy (out, obj, k);
          mp.bind (out, k, ctx);
     }
     else mp.copy (ctx, obj, k);
   });
   return out;
}
```

Si se trata de un atributo se copia en el objeto de contexto, sino, se copia como método público y se vincula al objeto de contexto



Los métodos se vinculan dinámicamente al objeto de contexto para que las referencias a this en p y q encuentren los atributos x e y que ahora son atributos privados

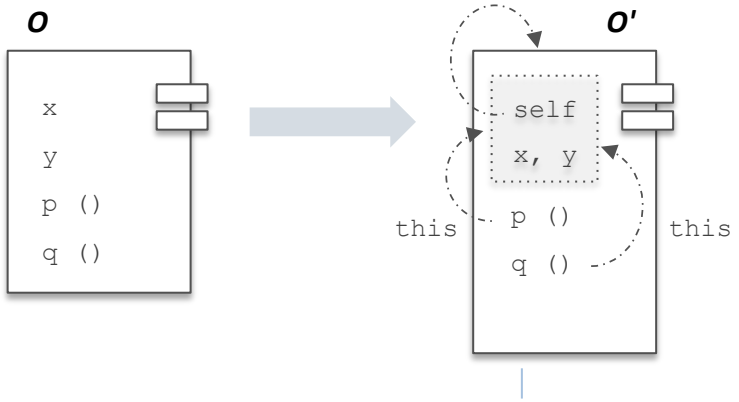
Patrones de Encapsulación

La falta de mecanismos nativos de encapsulación es una de las grandes críticas que se hace a JavaScript. Pese a que en orientación a objetos lo importante es en realidad respetar el principio de protección de la información, aquí presentamos una colección de patrones que ofrecen distintas alternativas para conseguir encapsulación de estado.

Self

Dado que el empleo de hide recontextualiza los métodos del componente en un objeto privado se pierde la posibilidad de acceder a this. Para ello se inserta self dentro del contexto.

```
function self (obj) {
   var out = {};
   var ctx = { self: out };
   Object.keys(obj).forEach (function (k) {
     if (typeof (obj[k]) === 'function') {
          mp.copy (out, obj, k);
          mp.bind (out, k, ctx);
     }
     else mp.copy (ctx, obj, k);
   });
   return out;
}
```



Como antes, debido a la recontextualización, desde el código de p y q se puede acceder al contexto privado donde residen x e y haciendo uso del this. Ahora se introduce self en el contexto para acceder a p y q. Así Desde p y q:

```
this.x      Refiere a x
this.y      Refiere a y
this.self.p Refiere a p
this.self.q Refiere a q
```

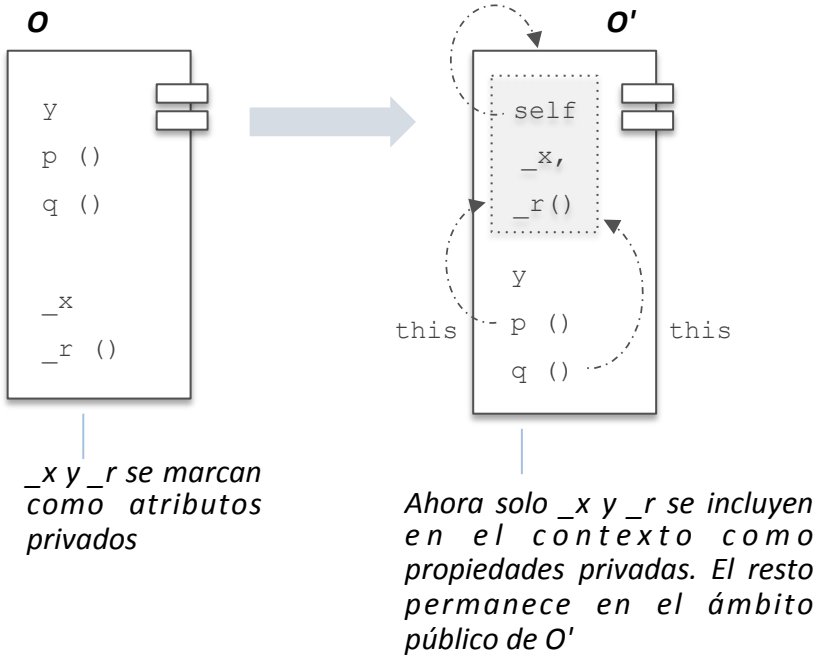
Patrones de Encapsulación

La falta de mecanismos nativos de encapsulación es una de las grandes críticas que se hace a JavaScript. Pese a que en orientación a objetos lo importante es en realidad respetar el principio de protección de la información, aquí presentamos una colección de patrones que ofrecen distintas alternativas para conseguir encapsulación de estado.

Hungarian

A veces en JavaScript se consideran privados sólo los miembros que empiezan por _. Se puede emplear ese u otro criterio sintáctico para modificar el patrón self.

```
function self (obj) {
   var out = {};
   var ctx = { self : out };
   Object.keys (obj).forEach (function (k) {
      if (k[0] !== '_') {
        mp.copy (out, obj, k);
        mp.bind (out, k, ctx);
      }
      else {
        mp.copy (ctx, obj, k);
        mp.bind (ctx, k, ctx)
      }
  });
  return out;
}
```



_x y _r se marcan como atributos privados

Ahora solo _x y _r se incluyen en el contexto como propiedades privadas. El resto permanece en el ámbito público de O'

Patrones de Encapsulación

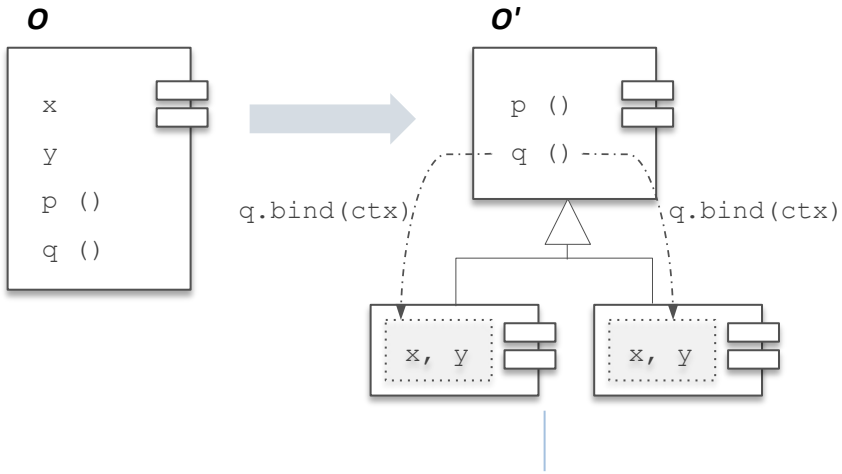
La falta de mecanismos nativos de encapsulación es una de las grandes críticas que se hace a JavaScript. Pese a que en orientación a objetos lo importante es en realidad respetar el principio de protección de la información, aquí presentamos una colección de patrones que ofrecen distintas alternativas para conseguir encapsulación de estado.

Prototify

Genera un componente donde las propiedades se introducen en un contexto interno en cada instancia hija y los métodos se vinculan a dicho contexto.

```
function prototify (obj) {
   var ctx  = mp.context ();
   var out  = {};
   out[ctx] = {};
   Object.keys(obj).forEach (function (k) {
     if (typeof (obj[k]) === 'function')
       mp.outerDelegate (out, obj, k, out[ctx]);
     else mp.copy (out[ctx], obj, k);
   });
   return out;
}
```

Context () define el nombre de una propiedad anónima para que albergue el contexto en cada instancia hija



Cuando los métodos p y q residen en el prototipo deben crearse contextos distintos en los hijos y contextualizar el puntero this en cada contexto

Patrones de Encapsulación

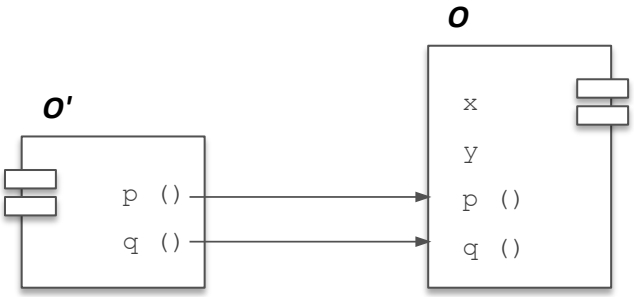
La falta de mecanismos nativos de encapsulación es una de las grandes críticas que se hace a JavaScript. Pese a que en orientación a objetos lo importante es en realidad respetar el principio de protección de la información, aquí presentamos una colección de patrones que ofrecen distintas alternativas para conseguir encapsulación de estado.

Proxify

Genera un componente proxy que contiene sólo los métodos miembro de otro componente en el que delega. De esta manera el proxy no expone los atributos que contiene el delegado.

```
function proxify (obj) {
   var out = {};
   Object.keys(obj).forEach (function (k) {
     if (typeof (obj[k]) === 'function')
         mp.forward (out, obj, k);
   });
   return out;
}
```

La operación de forwarding genera un método de proxy que delega en cada método del objeto original



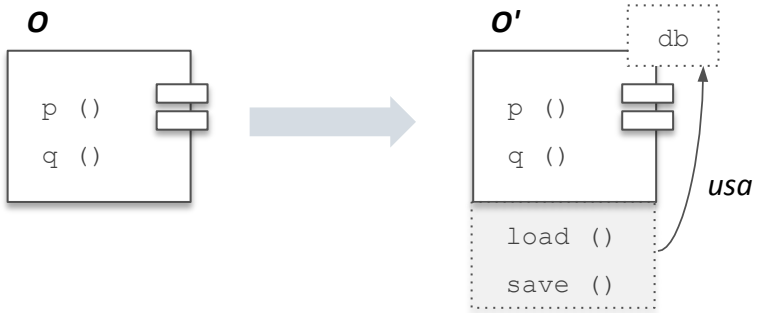
Sólo para los métodos de O se genera un método de proxy en O' que hace forwarding a O. De esta manera el estado x e y no es accesible desde O'

Patrones de Contextualización

Las técnicas de contextualización permiten condicionar el comportamiento de un componente a ciertas condiciones ambientales o parámetros de configuración. Esta sección recoge una familia de patrones que ofrecen ideas de adaptación y composición dinámica a partir de la alteración del contexto de ejecución de los componentes.

Record

Este patrón extiende por adición un componente para que incluya métodos de gestión de estado save & load. La base de datos sobre la que opera es un parámetro subyacente de contexto.



Los métodos save y load se insertan dinámicamente por metaprogramación para articular la gestión de estado del componente

```
function record (db) {
  return function (obj) {
     mp.add (obj, 'load', function () {
        var data = db.find (obj.id);
        mp.extend (obj, data);
     });
     mp.add (obj, 'save', function () {
        db.save (obj);
     });
  };
  return obj;
}
```

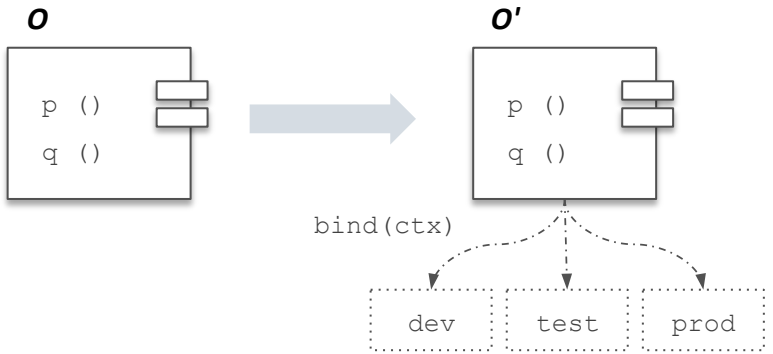
La base de datos es un valor de contexto que permanece oculto tras la construcción por retención de variables

Patrones de Contextualización

Las técnicas de contextualización permiten condicionar el comportamiento de un componente a ciertas condiciones ambientales o parámetros de configuración. Esta sección recoge una familia de patrones que ofrecen ideas de adaptación y composición dinámica a partir de la alteración del contexto de ejecución de los componentes.

Sink

Este patrón contextualiza la evaluación de todos los métodos de un componente sobre un objeto de contexto externo. Resulta útil para manejar diversos entornos de trabajo.



Cada contexto corresponde a un entorno distinto donde se evalúan los métodos de O'

```
function sink (env) {
  return function (obj) {
     var out = {};
     mp.extend (out, obj);
     Object.keys(out).forEach (function (k) {
        if (typeof (out[k]) === 'function') {
           mp.bind (out, k, env);
        }
     });
     return out;
  };
}
```

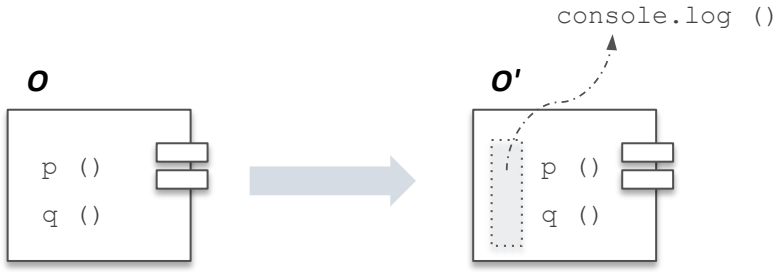
La operativa es sencilla. Consiste en extender el objeto para crear un clon y contextualizar cada método en un objeto de contexto pasado como parámetro

Patrones de Contextualización

Las técnicas de contextualización permiten condicionar el comportamiento de un componente a ciertas condiciones ambientales o parámetros de configuración. Esta sección recoge una familia de patrones que ofrecen ideas de adaptación y composición dinámica a partir de la alteración del contexto de ejecución de los componentes.

Trace

Este patrón decora cada uno de los métodos de un componente para escribir una traza de ejecución. En este ejemplo se utiliza intercesión en before y traza sobre la consola estándar.



Se decora cada uno de los métodos del componente para incorporar cierta lógica que realiza una traza por la consola estándar su ejecución

```
function trace (format, log) {
  return function (obj) {
     Object.keys(obj).forEach (function (k) {
        if (typeof (obj[k]) === 'function')
            mp.before (obj, k, function () {
                log (format, k);
            });
     });
  };
  return obj;
}
```

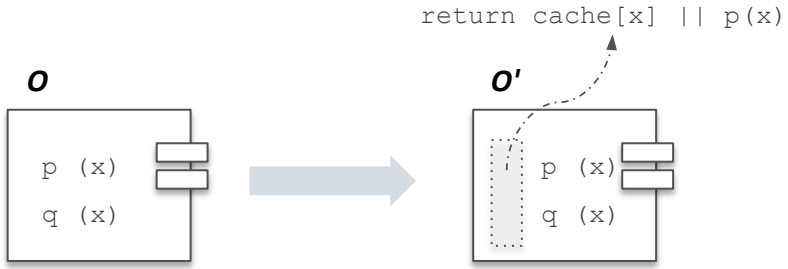
El método en before se aplica para articular la intercesión que requiere este patron

Patrones de Contextualización

Las técnicas de contextualización permiten condicionar el comportamiento de un componente a ciertas condiciones ambientales o parámetros de configuración. Esta sección recoge una familia de patrones que ofrecen ideas de adaptación y composición dinámica a partir de la alteración del contexto de ejecución de los componentes.

Cache

Otro ejemplo de contextualización lo encontramos en el patrón cache que decora cada método para incorporarle una cache local en una variable retenida.



Se decora cada uno de los métodos del componente para incorporar cierta lógica que consulta una cache antes de computar de nuevo una solicitud y se actualiza dicha cache después del computo

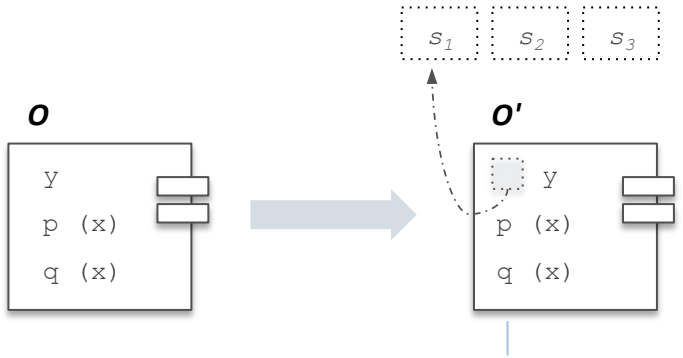
```
function cache (obj) {
   Object.keys(obj).forEach (function (k) {
      if (typeof (obj[k]) === 'function')
        var fn = obj[k];
        mp.add (obj, k, (function () {
          var cache = {};
          return function () {
             var args = [].slice.call (arguments);
             var r = cache[args] || fn.apply (this, args);
             cache[args] = r;
             return r;
          };
        }.bind(this))());
      }
   }, this);
   return obj;
}
```

Patrones de Contextualización

Las técnicas de contextualización permiten condicionar el comportamiento de un componente a ciertas condiciones ambientales o parámetros de configuración. Esta sección recoge una familia de patrones que ofrecen ideas de adaptación y composición dinámica a partir de la alteración del contexto de ejecución de los componentes.

Undo

El patrón Undo decora los métodos de un componente para registrar los cambios en una pila de estado. Se añade un método undo para restaurar el estado.



La decoración en before de los atributos inserta funciones de estado en una pila de estado. Cuando se invoca undo() se desapila el último estado y se invoca para invertir el último cambio

```
function undo (obj) {
    var states = [];
    var off    = false;
    Object.keys (obj).forEach (function (k) {
       if (typeof (obj[k]) !== 'function')
          mp.beforeAtt (obj, k, {
             set: function (ov, nv) {
                if ((ov !== nv) && !off)
                    states.push (function () {
                       off = true;
                       obj[k] = ov;
                       off = false;
                    });
             }
          });
    });
    mp.add (obj, 'undo', function () {
        var state = states.pop ();
        if (state) state ();
    });
    return obj;
}
```

Metaprogramación Compositiva En JavaScript

Preguntas

Técnicas de Composición

Mecanismos de Composición

Patrones de Composición

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Programación Orientada a Componentes

Metaprogramación Compositiva En JavaScript

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Mayo 2015