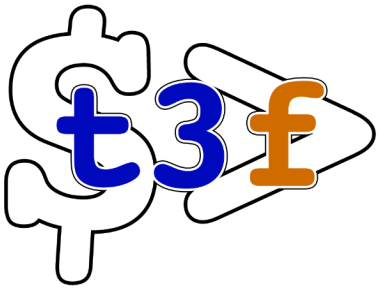
Programación Orientada a Componentes

Metaprogramación En JavaScript

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Febrero 2015



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

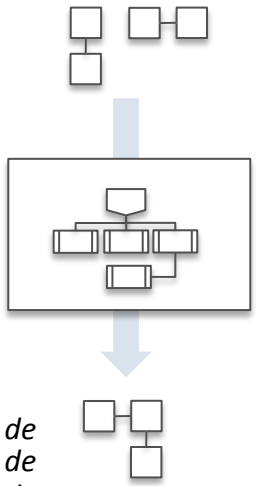
- Qué Es La Metaprogramación
- Por Qué La Metaprogramación
- El Contexto De La Metaprogramación
- El Proceso De La Metaprogramación
- La Metaprogramación Como Paradigma

Qué Es La Metaprogramación

Definición

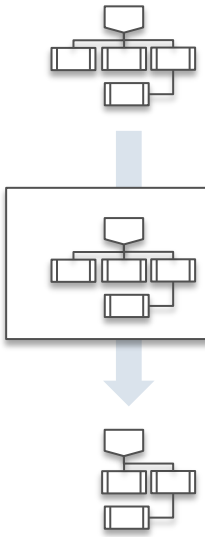
La metaprogramación es una disciplina de programación que consiste en construir programas que generan, manipulan o modifican otros programas, incluidos ellos mismos.

Programa Como Proceso de Transformación De Datos



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

Metaprograma Como Proceso de Transformación De Programas



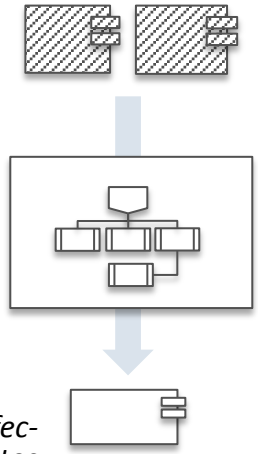
El metaprograma toma unos datos de entrada que representan un programa y valores de configuración para dirigir el proceso de construcción de otro programa

Qué Es La Metaprogramación

Tipos De Metaprogramación

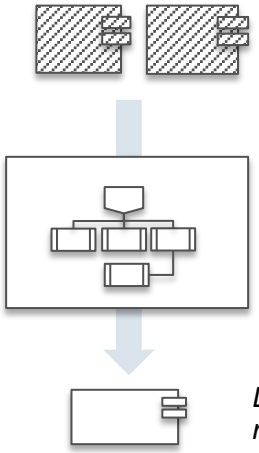
En términos generales se puede pensar en dos grandes familias de técnicas de metaprogramación. La programación generativa y la programación orientada a componentes.

Metaprogramación Generativa
Programación Generativa



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

Metaprogramación Compositiva
P. Orientada A Componentes



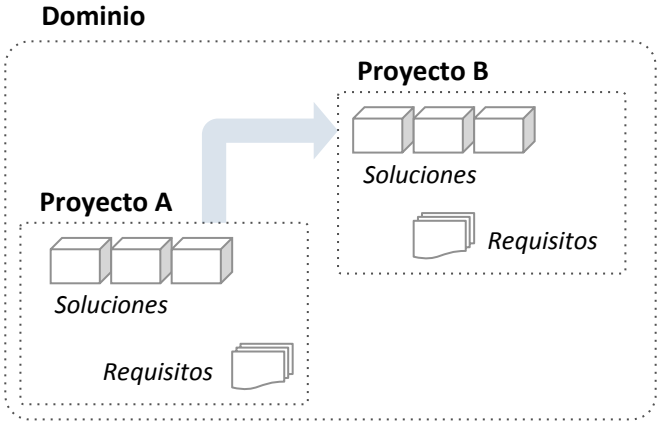
La programación orientada a componentes elabora nuevos componentes a partir de la combinación compositiva de otros componentes

Por Qué La Metaprogramación

Factorías de Software. La Construcción de un Dominio

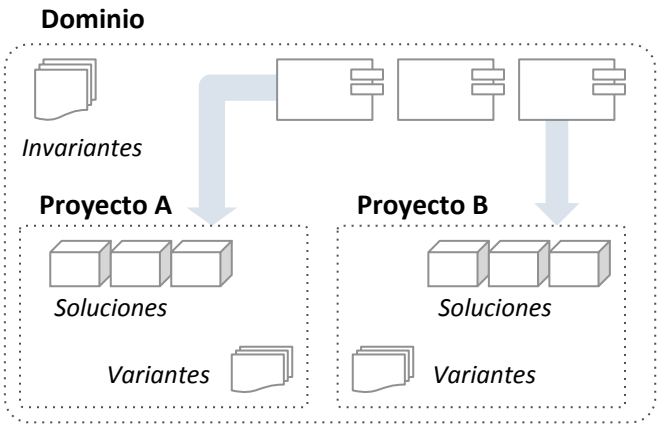
Algunas compañías se especializan en la construcción de un tipo de productos dentro de un dominio específico. Con el ánimo de fomentar la reutilización conviene construir artefactos que contribuyan a dominio en vez de limitar su aplicación al ámbito de un proyecto.

Aproximación Clásica
Contribuciones A Proyecto



La reutilización clásicamente se entiende como un proceso de migración de código entre proyectos

Aproximación Metaprogramática
Contribuciones A Dominio



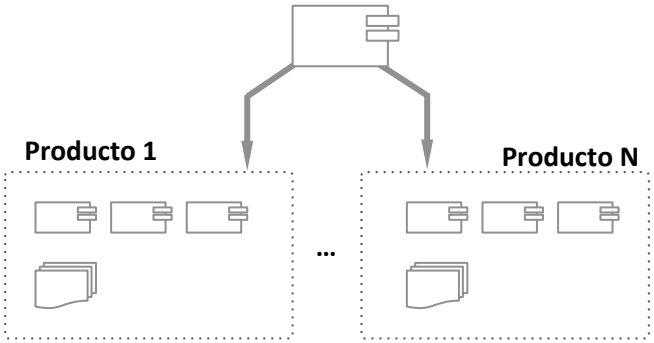
En la aproximación metaprogramática los componentes se expresan como artefactos abstractos a nivel de dominio

Por Qué La Metaprogramación

Líneas de Producto. Economía de Escala & Economía de Alcance

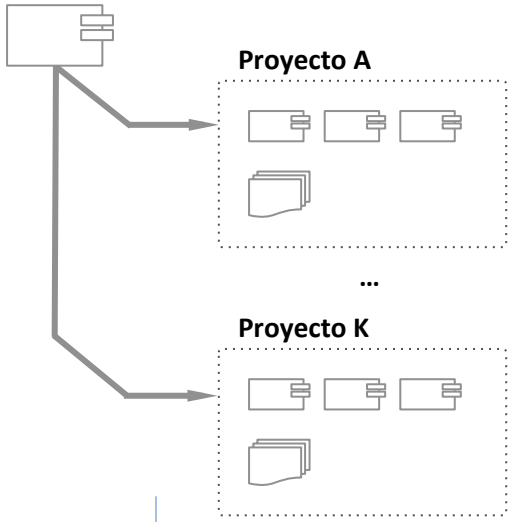
De esta manera se consigue reducir costes de desarrollo y mantenimiento por medio de la economía de escala y la economía de alcance. La primera reduce costes por el aumento de la producción de un solo producto. La segunda lo consigue por el abaratamiento de la adaptación en productos similares.

Economía De Escala



Una línea de producción permite el abaratamiento de costes por el desarrollo en masa de un gran número de productos idénticos

Economía De Alcance



La línea de variación consigue el abaratamiento de costes por la construcción de productos muy similares

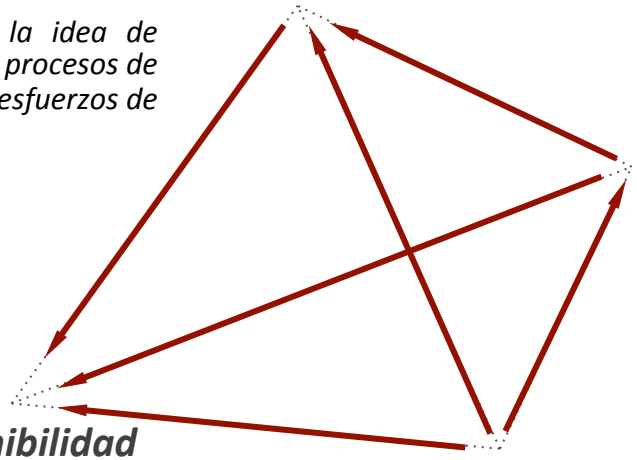
Por Qué La Metaprogramación

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, tanto la economía de alcance como la economía de escala.

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

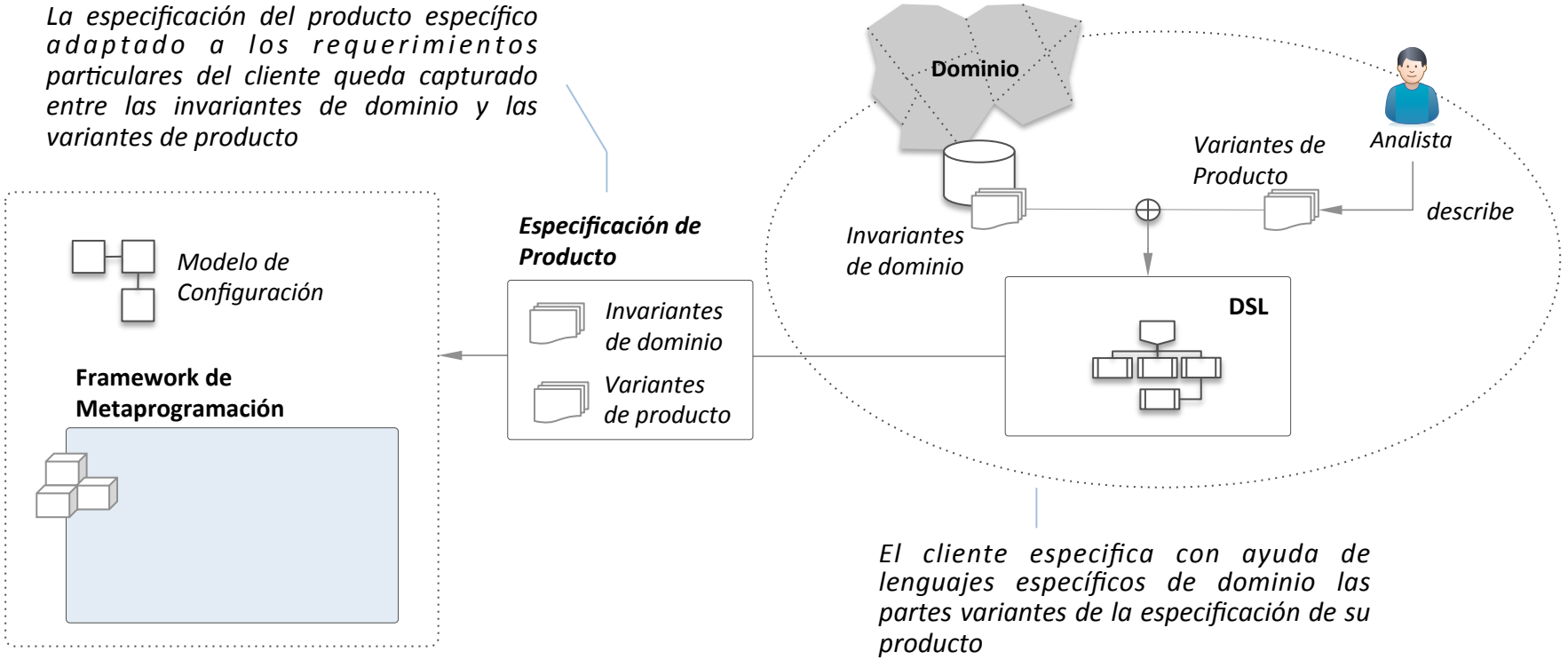
Extensibilidad

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

El Contexto De La Metaprogramación

Perspectiva De Caja Negra

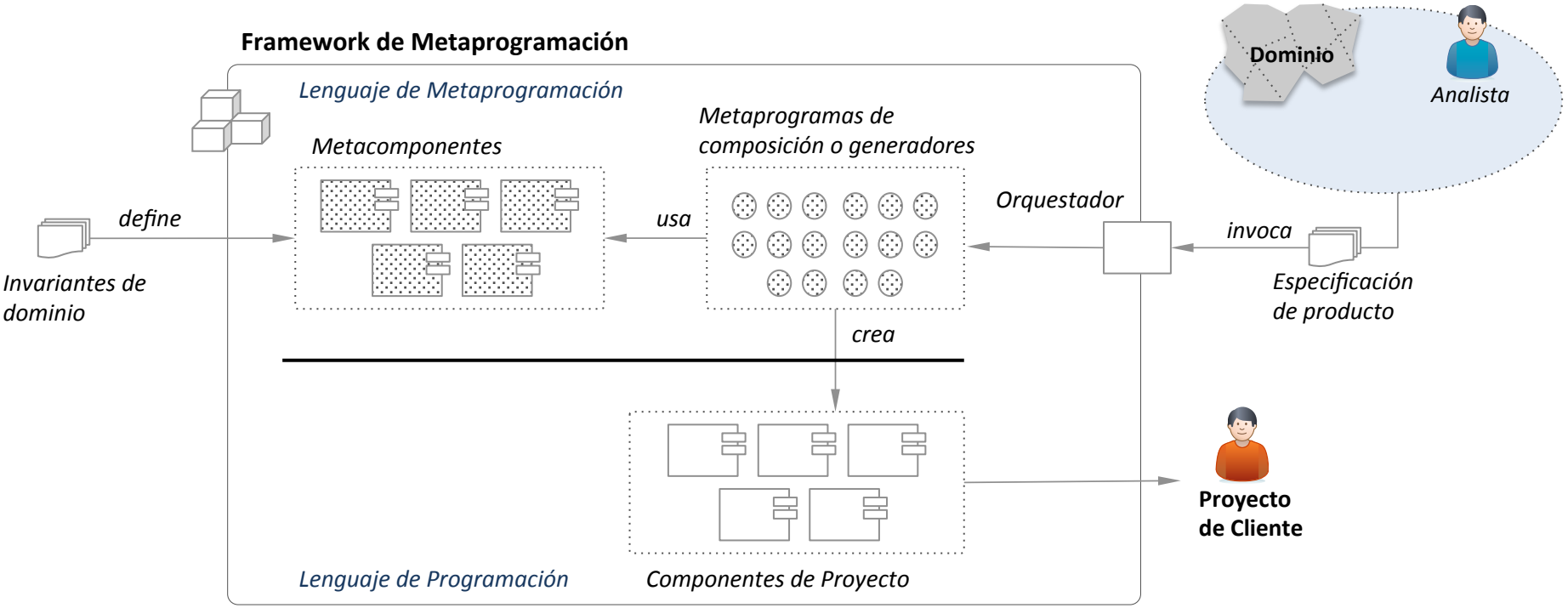
El analista de dominio establece una definición formal y computable del producto mediante el uso de lenguajes específicos de dominio. Esta definición se usa para configurar el framework de metaprogramación que articulará la solución específica requerida.



El Contexto De La Metaprogramación

Perspectiva De Caja Blanca

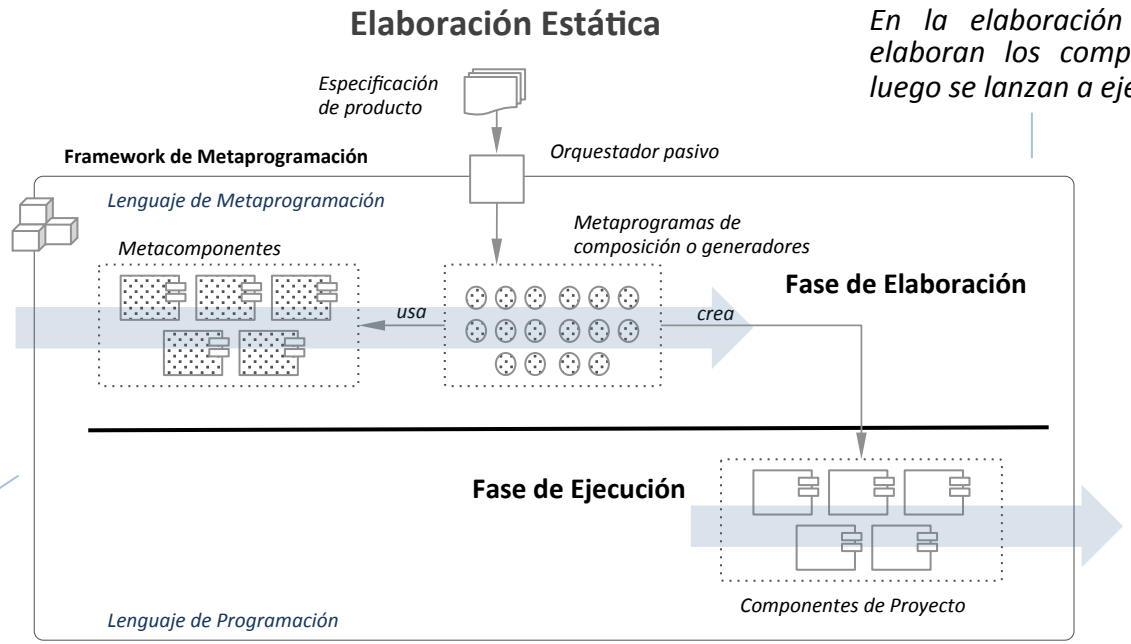
La especificación de las variantes de producto se procesa para determinar los metaprogramas que deben emplearse en el proceso constructivo. Estos metaprogramas pueden utilizar metacomponenentes prediseñados y aplican técnicas de composición o generación para producir componentes.



El Contexto De La Metaprogramación

Metaprogramación Estática & Dinámica

En la metaprogramación estática las fases de elaboración y ejecución se desarrollan secuencialmente mientras que en la elaboración dinámica el desarrollo de ambas fases es paralelo. Esta segunda aproximación ofrece oportunidades de reelaboración que no son posibles con la metaprogramación estática.



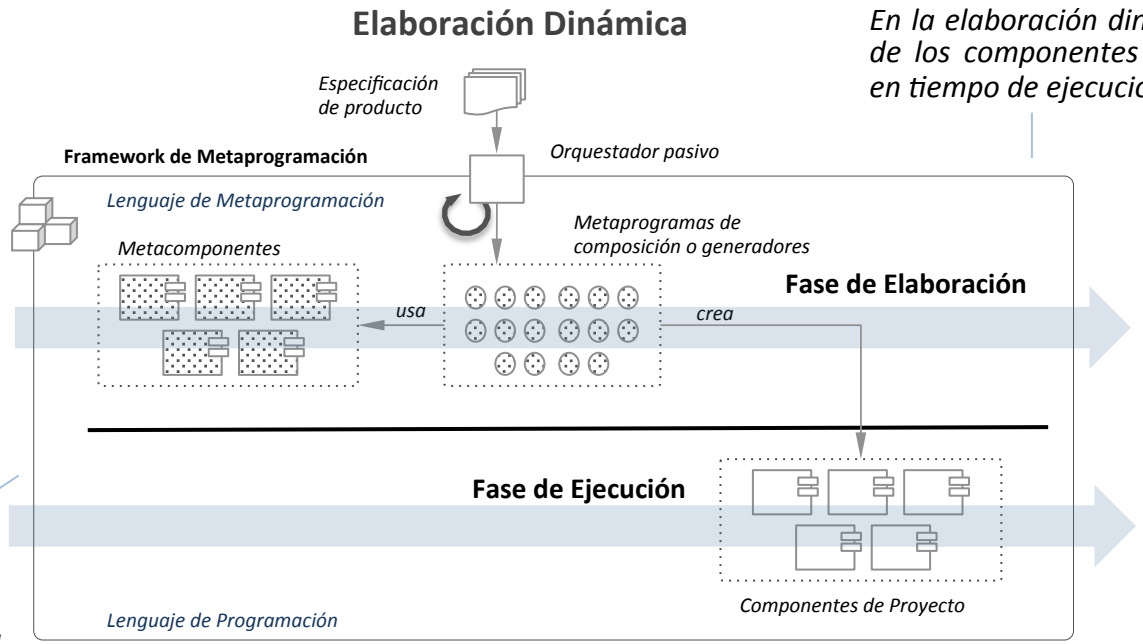
En la elaboración estática primero se elaboran los componentes de cliente y luego se lanzan a ejecución

La desventajas es que una vez creados los componentes, éstos no pueden readaptarse

El Contexto De La Metaprogramación

Metaprogramación Estática & Dinámica

En la metaprogramación estática las fases de elaboración y ejecución se desarrollan secuencialmente mientras que en la elaboración dinámica el desarrollo de ambas fases es paralelo. Esta segunda aproximación ofrece oportunidades de reelaboración que no son posibles con la metaprogramación estática.

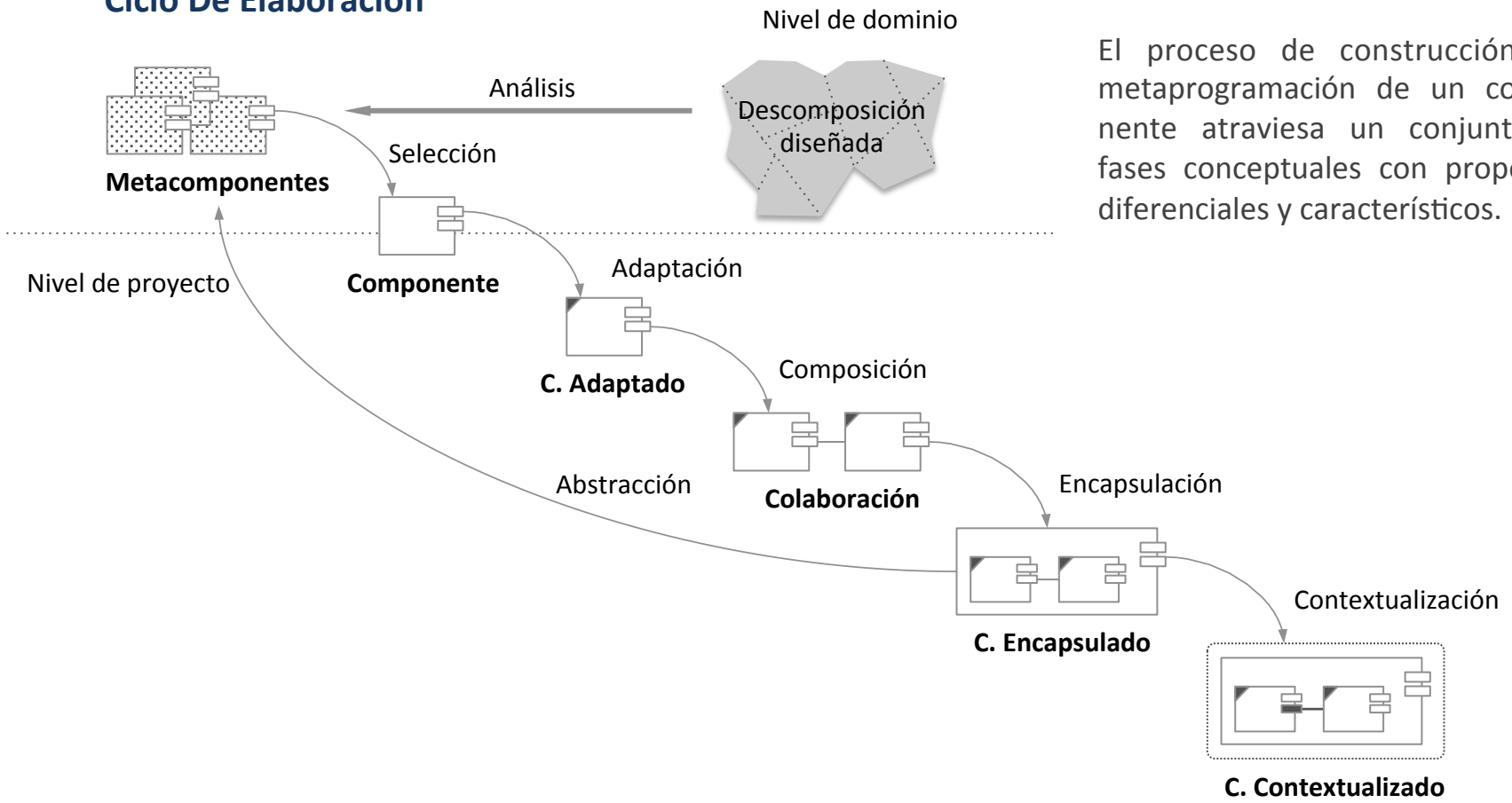


En la elaboración dinámica la elaboración de los componentes de cliente se realiza en tiempo de ejecución

Esto permite que los componentes se readapten dinámicamente al contextos cambiantes de uso

El Proceso De La Metaprogramación

Ciclo De Elaboración

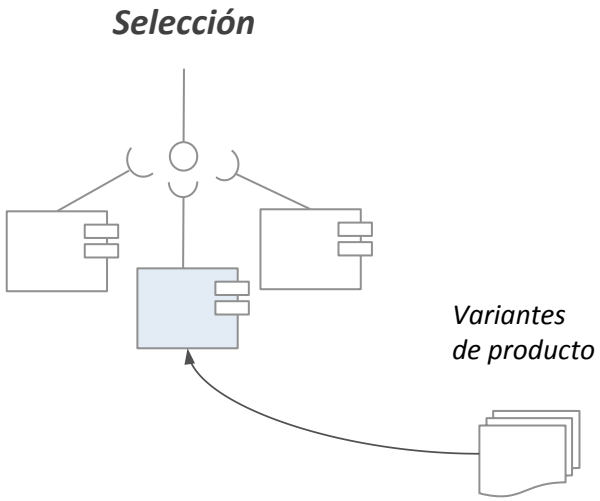


El proceso de construcción por metaprogramación de un componente atraviesa un conjunto de fases conceptuales con propósitos diferenciales y característicos.

El Proceso De La Metaprogramación

Selección

La selección consiste en determinar las variantes específicas de cada tipo de meta-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 Estática

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

Selección Dinámica

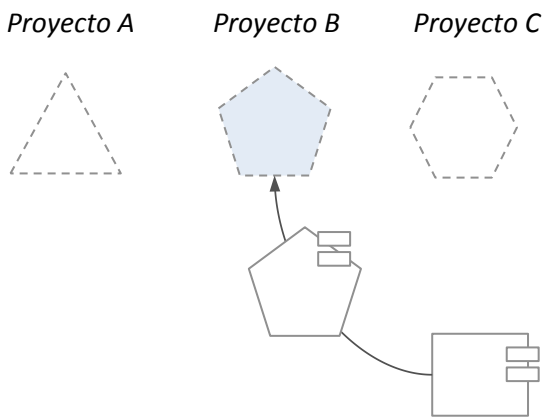
La ventaja de la selección dinámica es que permite dar soporte a escenarios donde las variantes utilizadas pueden cambiar durante el tiempo de ejecución

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 a nivel meta 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

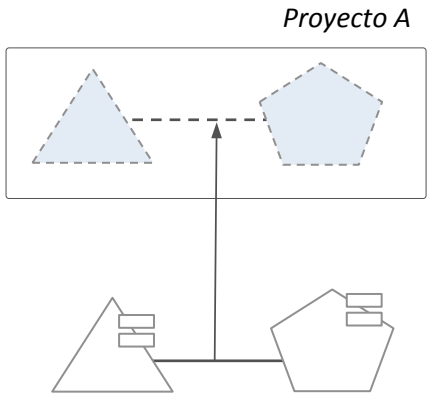
La estandarización de contratos y comportamientos evita la adaptación. Asimismo cuando los metacomponentes son componentes directamente operativos se posibilita la ausencia ocasional de adaptación

El Proceso De La Metaprogramación

Composición

La composición es un ejercicio de transformación en el que un metacomponente al que llamaremos núcleo se enriquece por la adquisición de cierta lógica parcial que reside en otro u otros metacomponentes 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

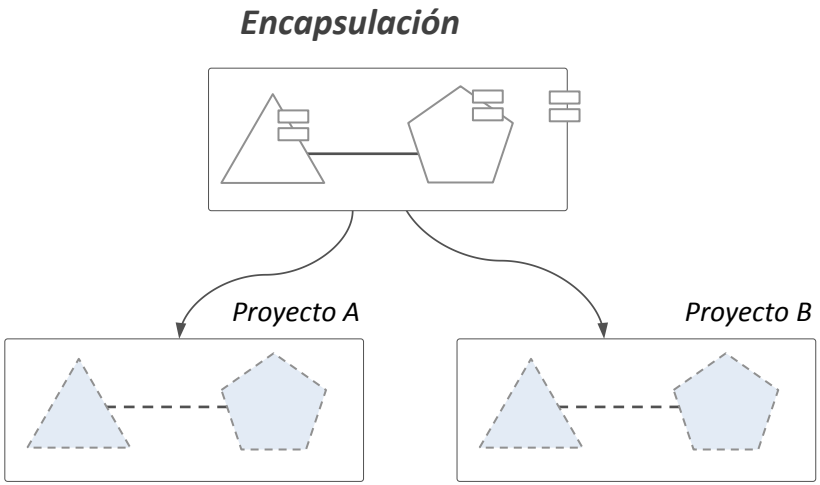


Extensión \ **Núcleo**	Dependiente	Autónomo
Dependiente	*Composición Cooperativa*	*Composición Subordinada*
Autónomo	*Composición Subordinante*	*Composición Autónoma*

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

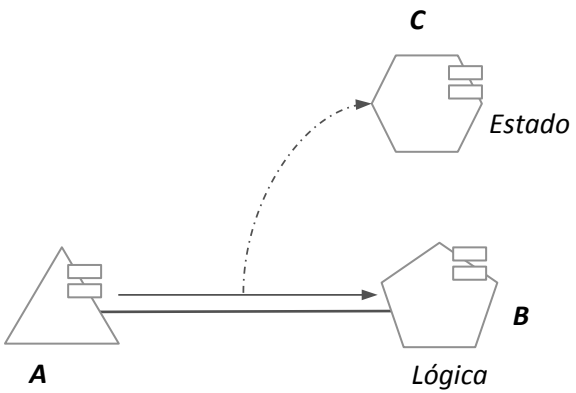
Por extensión, consideraremos que cualquier mecanismo que proporcione protección de información en el marco de la composición es una forma de encapsulación débil

El Proceso De La Metaprogramación

Contextualización

La contextualización es la fase última del proceso compositivo mediante la cual se dota al componente de una 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.

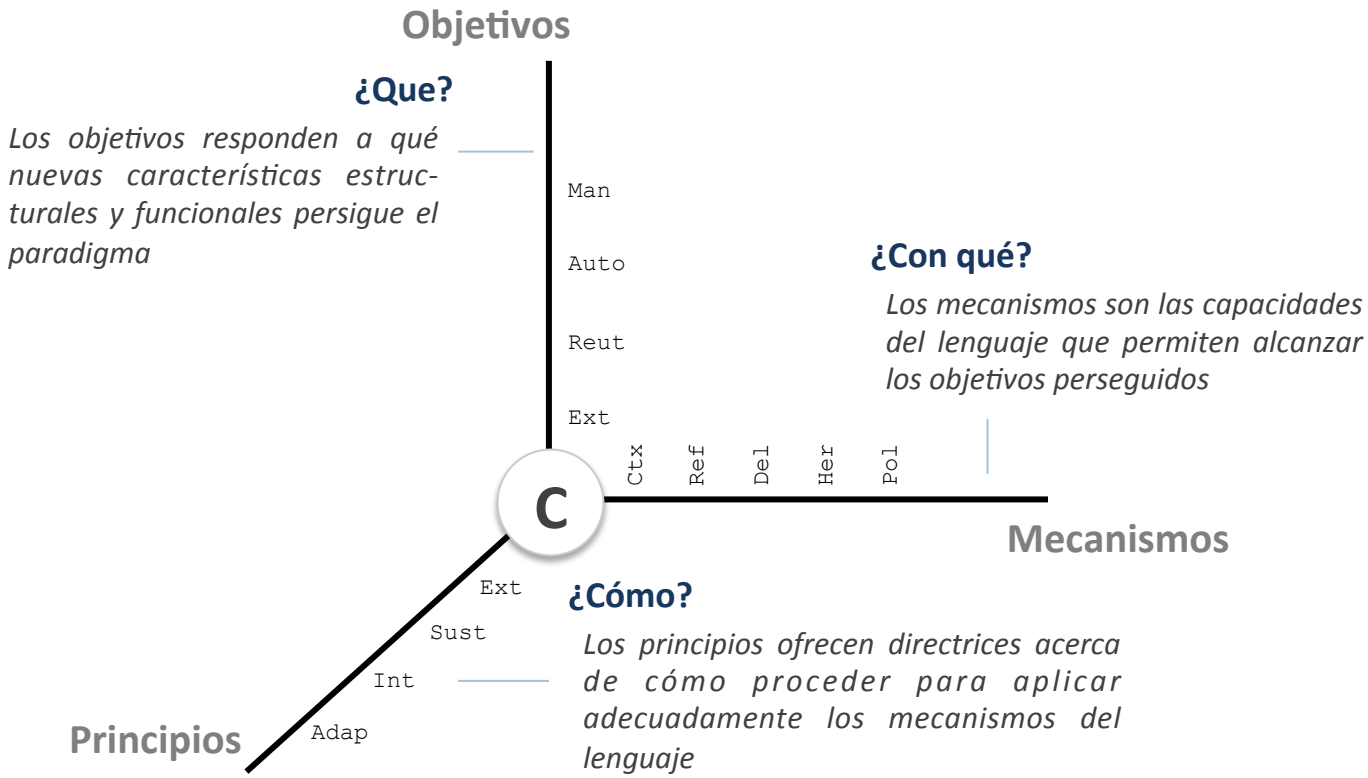
Contextualización Dinámica

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

La Metaprogramación Como Paradigma

Ejes Dimensionales

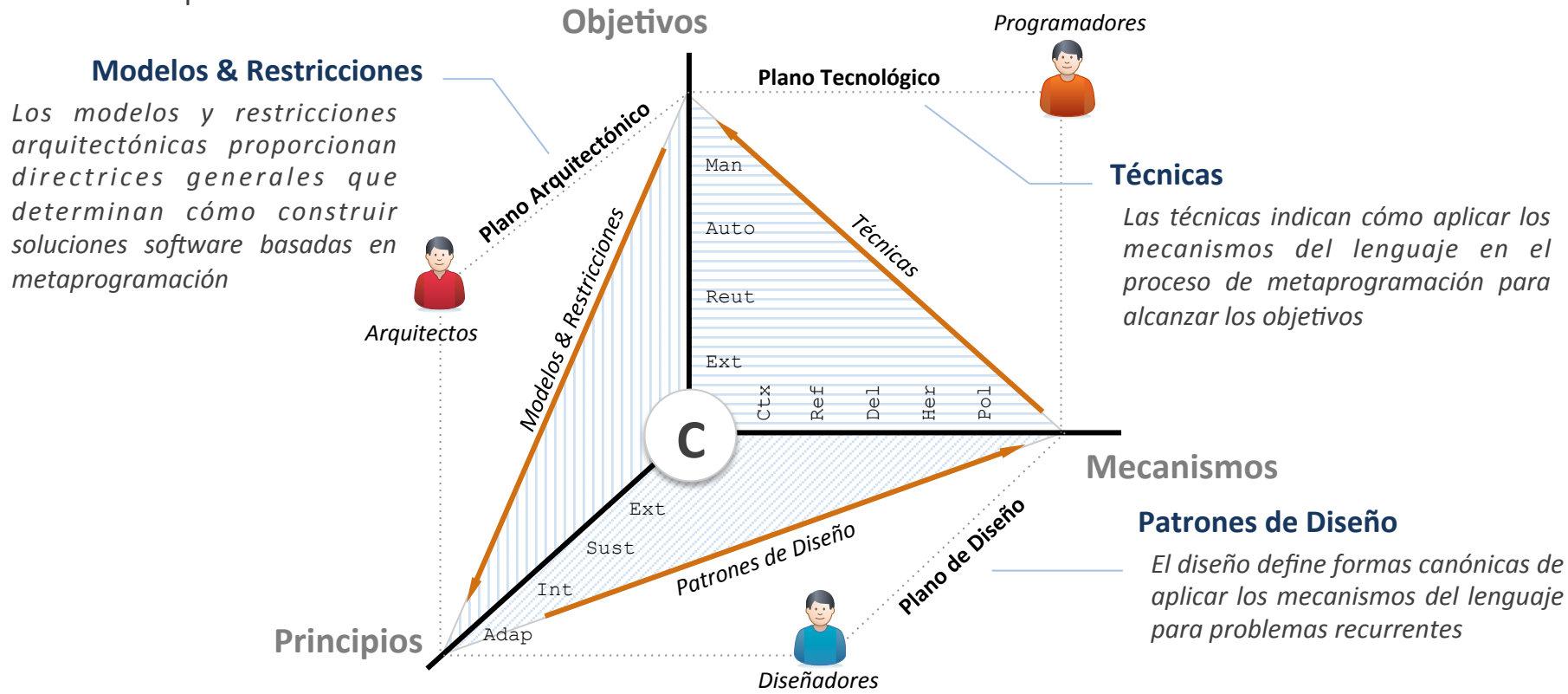
En torno al concepto de componente se pueden describir tres ejes dimensionales. Los objetivos determinan los propósitos perseguidos, los mecanismos caracterizan las capacidades nativas del lenguaje y los principios de diseño ofrecen directrices generales de actuación.



La Metaprogramación Como Paradigma

Planos De Actividad

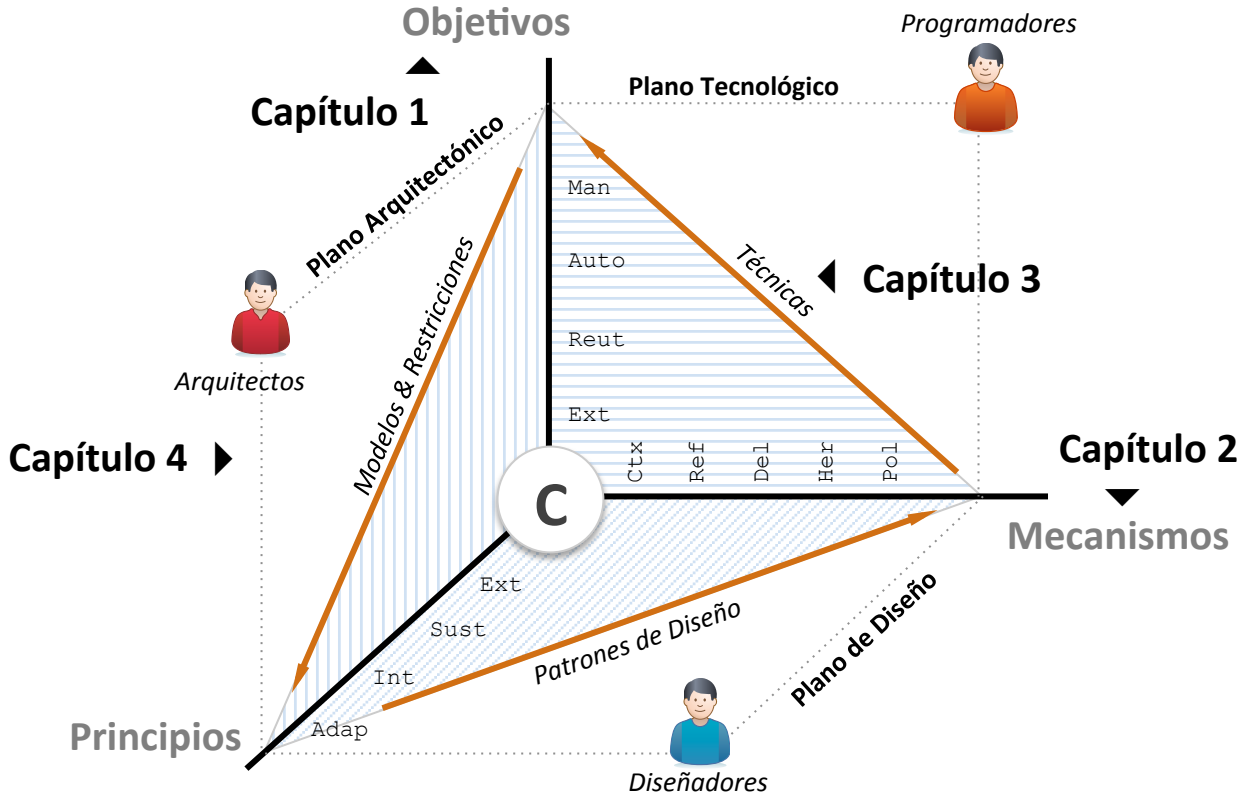
En torno a los tres ejes dimensionales se pueden establecer sendos planos de actividad. Las técnicas indican cómo aplicar los mecanismos del lenguaje, los patrones presentan soluciones a problemas recurrentes y los modelos arquitectónicos ofrecen restricciones que determinan como proceder.



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

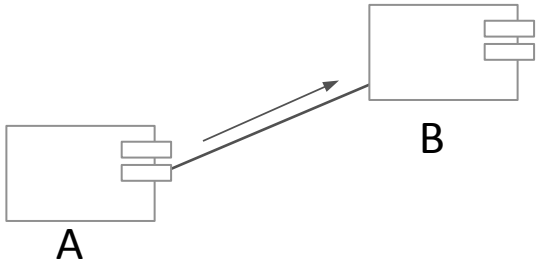
JavaScript como Lenguaje De Metaprogramación

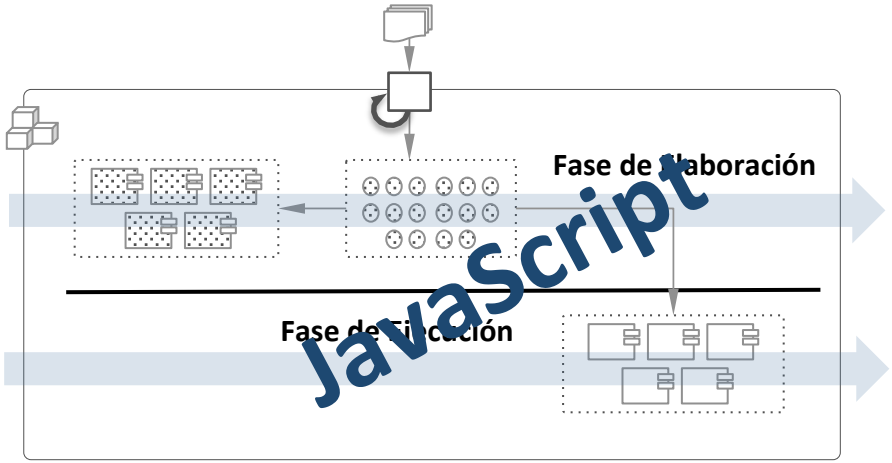
- Introducción
- Delegación & Herencia En La Composición
- Polimorfismo & Abstracción En La Composición
- Contextualización & Reflexión En La Composición

Introducción

Metaprogramación Compositiva

El espacio de posibles opciones cuando abordamos problemas de metaprogramación es muy amplio. Nuestro foco de atención aquí en primer lugar será discutir problemas en el marco de la composición y no de la generación. En este capítulo veremos las principales características del lenguaje que permiten articular procesos de confección compositiva.





Composición Dinámica

Por otro lado nos centraremos en escenarios de composición dinámica donde la fase de elaboración y ejecución coexisten durante el tiempo de producción de la solución software. Como veremos esto requiere que el lenguaje de metaprogramación y programación coincidan y frecuentemente se entre-mezclarán técnicas propias de ambos niveles de desarrollo.

Introducción

Capacidades Del Lenguaje

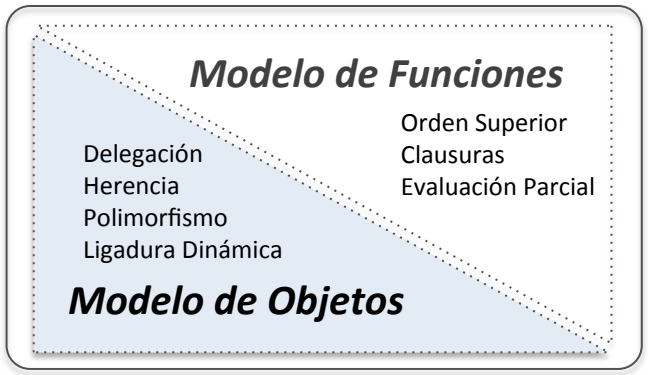
Como discutiremos, JavaScript es un buen candidato como lenguaje para cubrir las necesidades propias de los procesos de metaprogramación. En efecto, su carácter dinámico, su sistema de tipificación débil y su sistema de clases basado en prototipos confieren a los artefactos construidos un alto grado de plasticidad adaptativa que facilita los procesos de metaprogramación.

Dinamicidad

Tipificación Débil

Prototipos

JavaScript



Modelo De Objetos

Sin embargo, aquí restringiremos nuestros estudio a las capacidades del lenguaje en el marco de un modelo de componentes basado en objetos. En este sentido comprobaremos que las álgebras de objetos resultan muy flexibles y potentes para articular procesos de metaprogramación. Quedan fuera las capacidades del lenguaje vinculadas a la programación funcional que fomentan un estilo más generativo que compositivo.

Capacidades De JavaScript En El Modelo De Objetos

Los Objetos En La Composición

Los objetos son el ciudadano 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
};
```

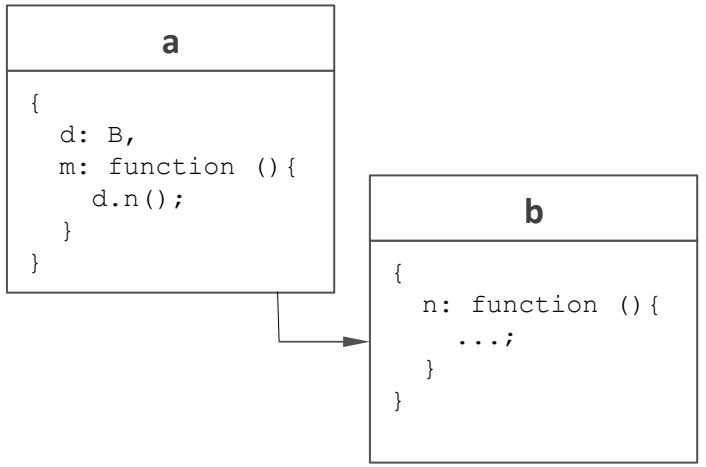
Objetos abiertos

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.

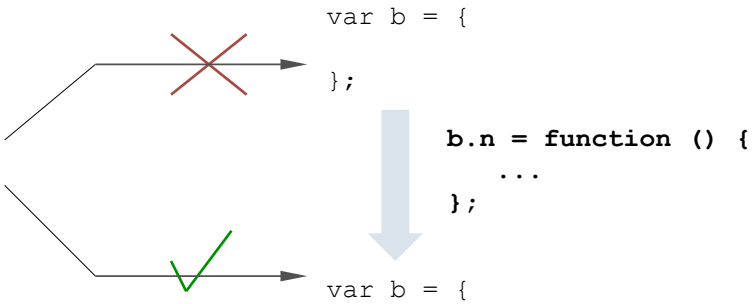


Delegación

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();
  }
};
```



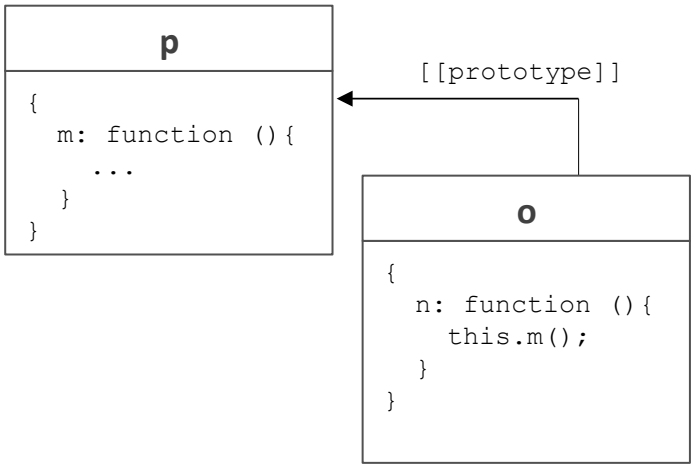
Enlace dinámico

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

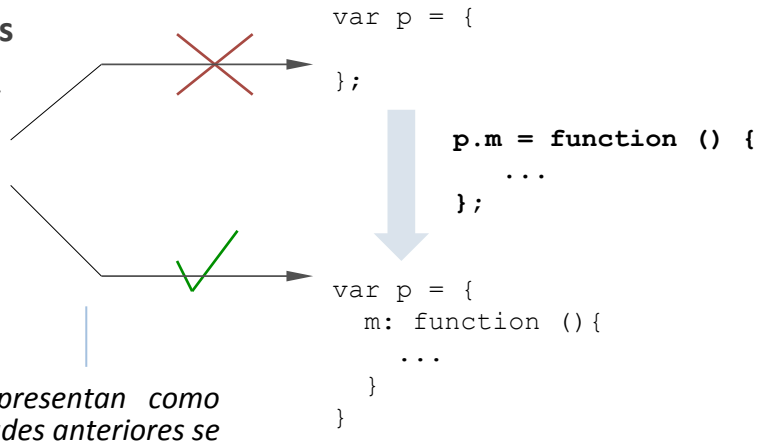
La Herencia En La Composición

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

En JavaScript el polimorfismo sólo puede interpretarse en términos de la conformidad a un contrato establecido por convenio. A diferencia de otros lenguajes, 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

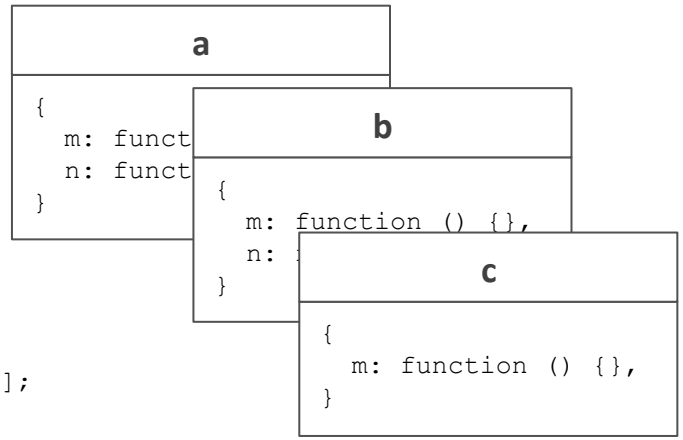
Conformidad Contractual

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

Capacidades De JavaScript En El Modelo De Objetos

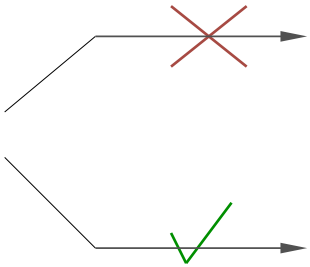
La Abstracción En La Composición

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

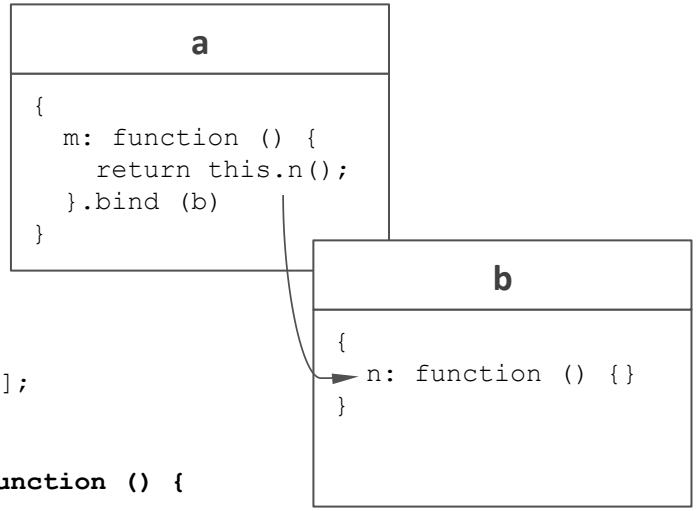
Conformidad Parcial

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

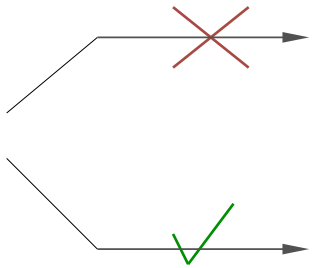
La Contextualización En La Composición

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 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);
```

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

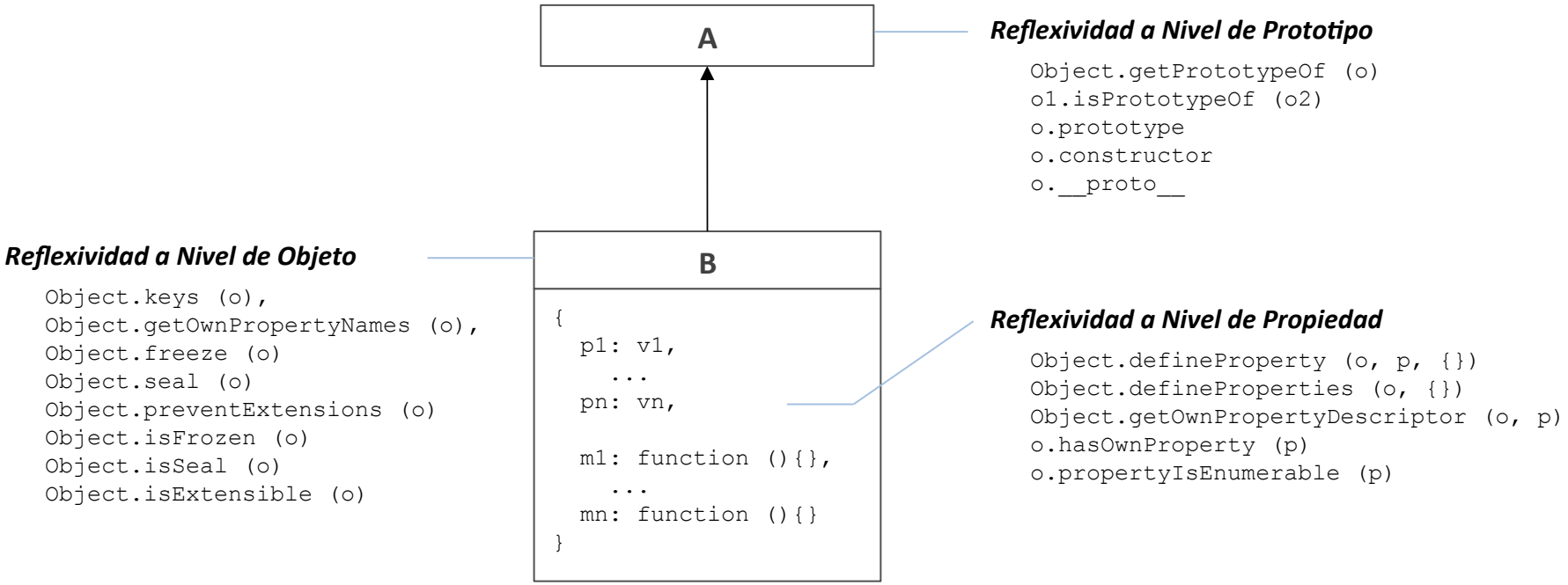
Contextualización

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.



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

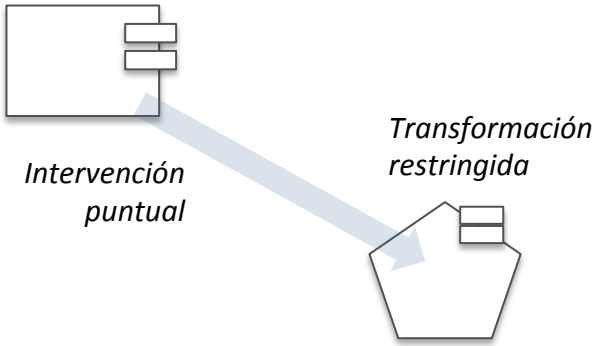
3 Técnicas De Metaprogramación Compositiva

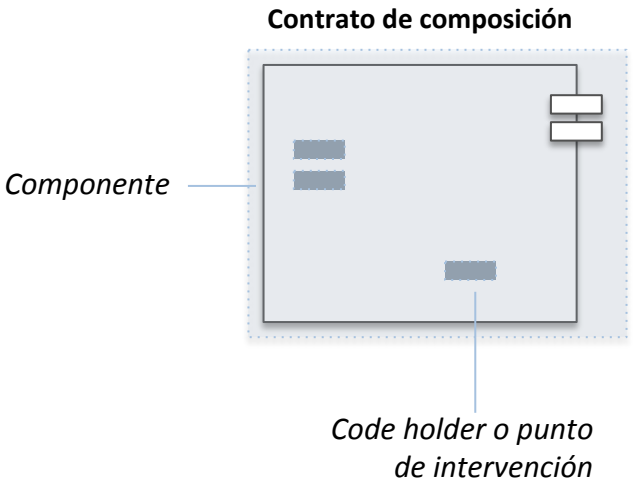
- Técnicas de Adición Compositiva
- Técnicas de Extensión Compositiva
- Técnicas de Intercesión Compositiva
- Técnicas de Delegación Compositiva

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.





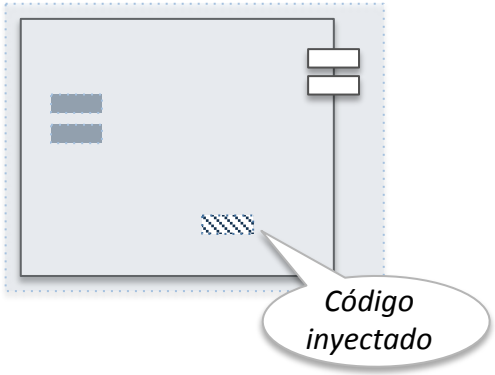
Code Holders & Contratos De Composición

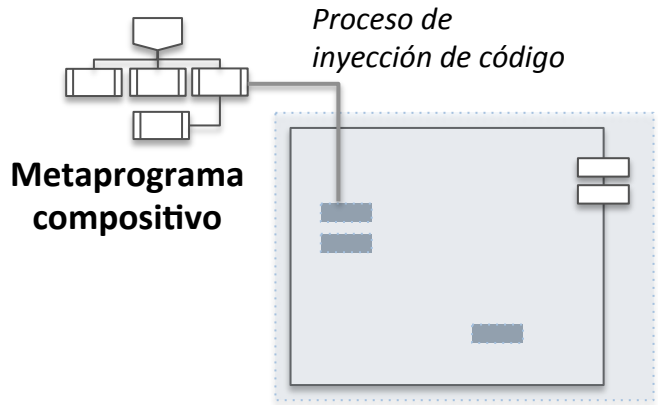
El conjunto potencial de intervenciones sobre un modelo de componentes queda impuesto por las capacidades metaprogramáticas del lenguaje y el modelo seleccionado. A lo largo de este capítulo nos centraremos en objetos aunque es fácil hacer una adaptación de los metaprogramas para que operen sobre las 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.





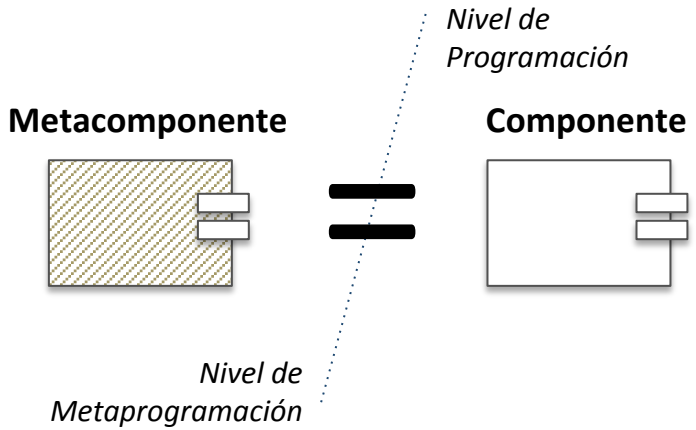
Metaprogramas & Intervención

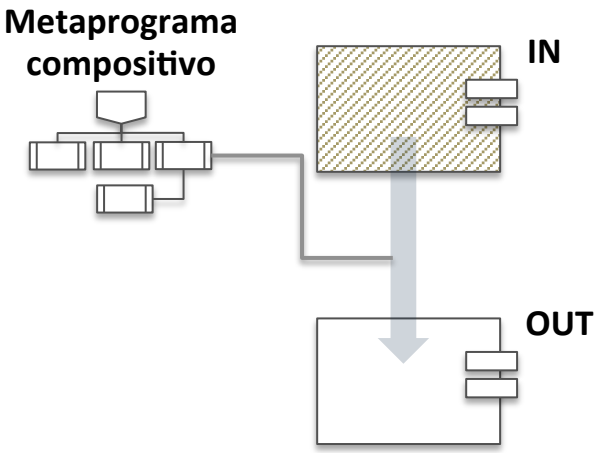
Los programas encargados de llevar a cabo el proceso de inyección de código sobre cada uno de los code holders del modelo de componentes se llaman metaprogramas. Estos algoritmos utilizan los operadores de composición para garantizar que los procesos de intervención son conformes con los diferentes tipos 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.

Introducción

Metacomponentes Como Componentes Prototipo

A diferencia de lo que ocurre en otras aproximaciones de composición, en JavaScript se impone que el modelo de componente y metacomponente coincidan. Es decir, un metacomponente es en realidad un componente convencional que funciona como ejemplar prototípico para confeccionar componentes operativos al nivel de programación.





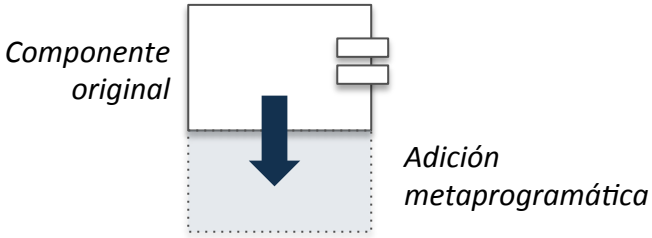
Metaprogramación Exógena

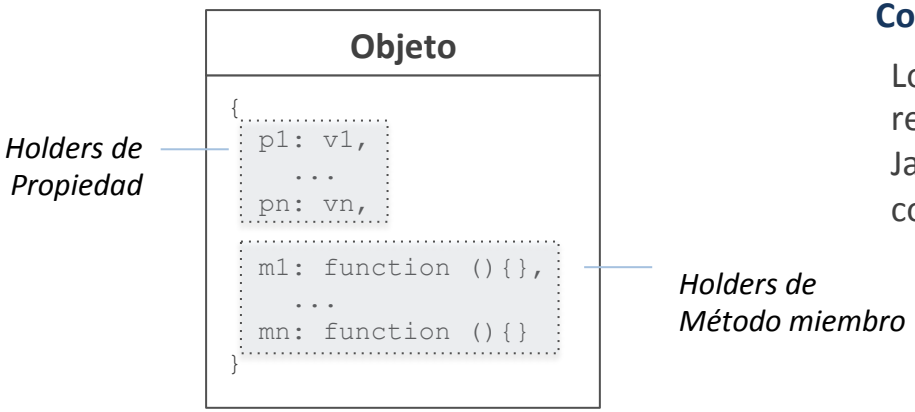
En el campo de la metaprogramación compositiva, los meta-programas se caracterizan por recibir como entrada cada una de las partes participantes en el proceso compositivo – núcleo y extensiones. El código del metaprograma opera sobre estos elementos de manera exógena. Es decir, éste no se incluye dentro de los mismos. En aproximaciones generativas el código del metaprograma coincide con el del componente resultante. Este tipo de meta-programación se conoce como endógena.

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;
};

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

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

mp.rename = function rename (core, oldKey, newKey) {
    mp.add (core, newKey, core[oldKey]);
    mp.remove (core, oldKey);
};
```

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) {
    mp.add (core, key, ext[key]);
};

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

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

Otros operadores de Adición

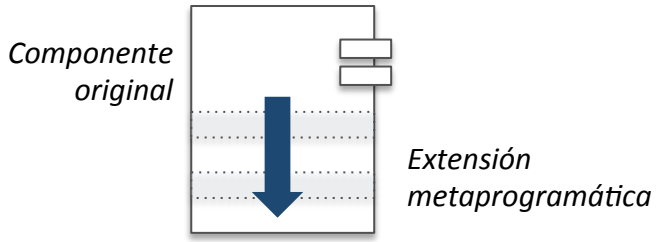
Los operadores fundamentales se complementan con otro conjunto de operadores que implican un núcleo y una extensión

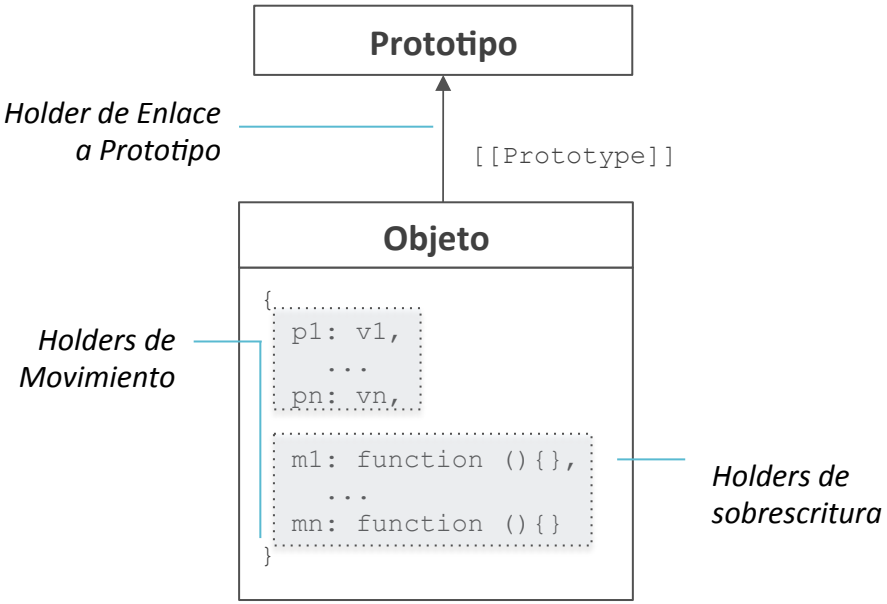
Operador	o1 o2
	{ y: 5 } {}
mp.copy (o2, o1, 'y');	{ y: 5 } { y: 5 }
mp.remove (o2, 'y');	{ y: 5 } {}
mp.move (o2, o1, 'x');	{} { y: 5 }
	{x: 0} {y: 0}
mp.extend (o1, o2);	{x: 0, y: 0} {y: 0}

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 prototipado.

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.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 = Object.getPrototypeOf (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 = Object.getPrototypeOf (core);
    if (proto)
        mp.add (proto, key, core[key]);
};
mp.copyDown = function (core, key, child) {
    mp.add (child, key, core[key]);
};
mp.moveUp = function (core, key) {
    mp.copyUp (core, key);
    mp.remove (core, key);
};
mp.moveDown = function (core, key, child) {
    mp.add (child, key, core[key]);
    mp.remove (core, key);
};
```

Operadores fundamentales de Extensión

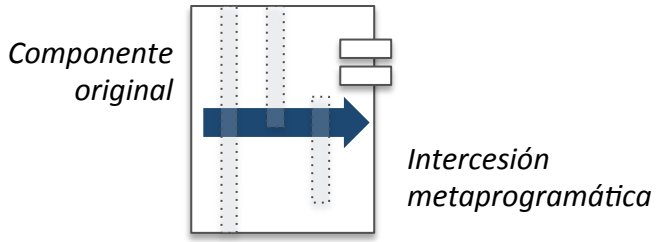
Las operaciones de copia y movimiento de características por la cadena de prototipado permiten alterar la semántica definida por la relación jerárquica entre objetos

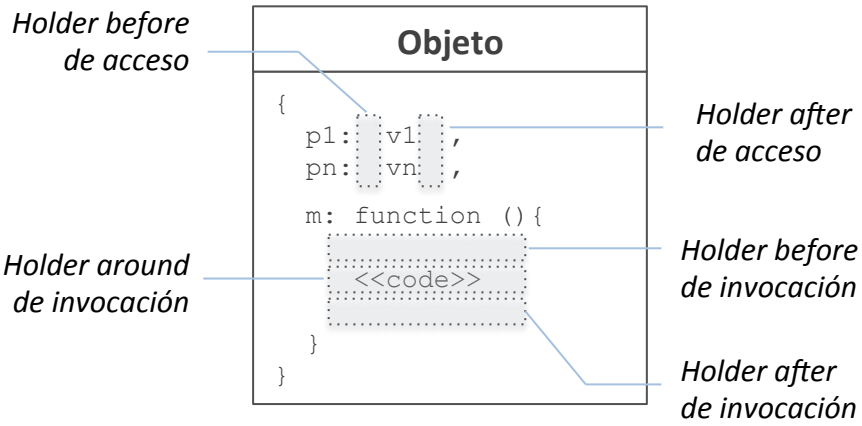
Operador	o p
`mp.copyUp (o, 'x');`	`o{x} -> p, o{x} -> p{x}`
`mp.copyDown (p, 'x', o);`	`o -> p{x}, o{x} -> p{x}`
`mp.moveUp (o, 'x');`	`o{x} -> p, o -> p{x}`
`mp.moveDown (p, 'x', o);`	`o -> p{x}, o{x} -> p`

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, 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 () {
      ext.apply (this, arguments);
      return fn.apply (this, arguments);
    };
};

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

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

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

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.

```
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);
    };

};
```

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

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

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],
        enumerable: false, ... });
    Object.defineProperty (core, key, {
        get: function () {
            var r = core['_' + key];
            ext.call (this, arguments);
            return r; }
    });
};
mp.afterSetAtt = function (core, key, ext) {
    Object.defineProperty (...);
    Object.defineProperty (core, key, {
        set: function (v) {
            core['_' + key] = v;
            ext.call (this, v);
        }
    });
};
```

Operador explícito de Intercesión

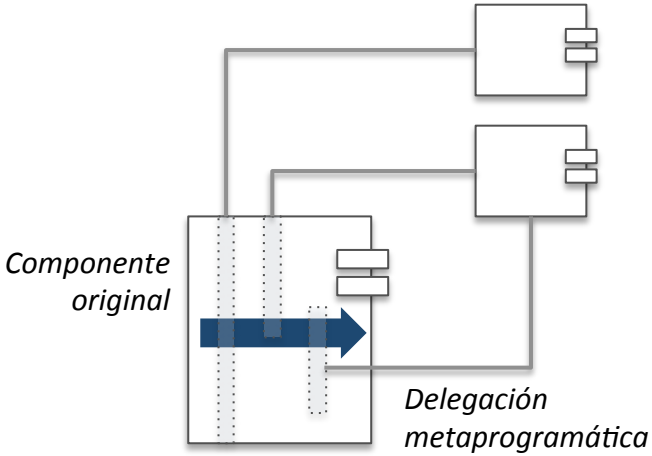
Se decoran los métodos de acceso y modificación a propiedades get y set y se mantiene una variable oculta para soportar el estado de la propiedad.

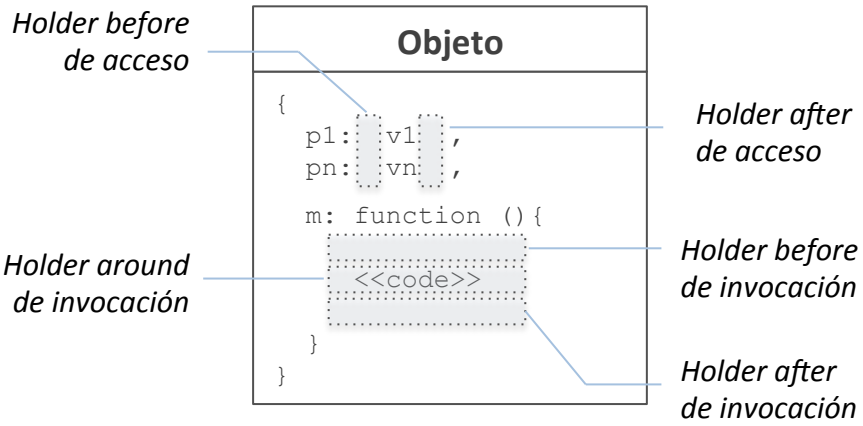
Operador	○
	{x: 0}
mp.afterGetAtt (o, 'x', f)	o.x -> 0, f
mp.afterSetAtt (o, 'x', f)	o.x = 3 -> {x: 3}, f

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 delegativos a 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 () {
      return core[oKey].apply (context, arguments);
   };
};

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

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) {
    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);
        });
    }
};
```

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

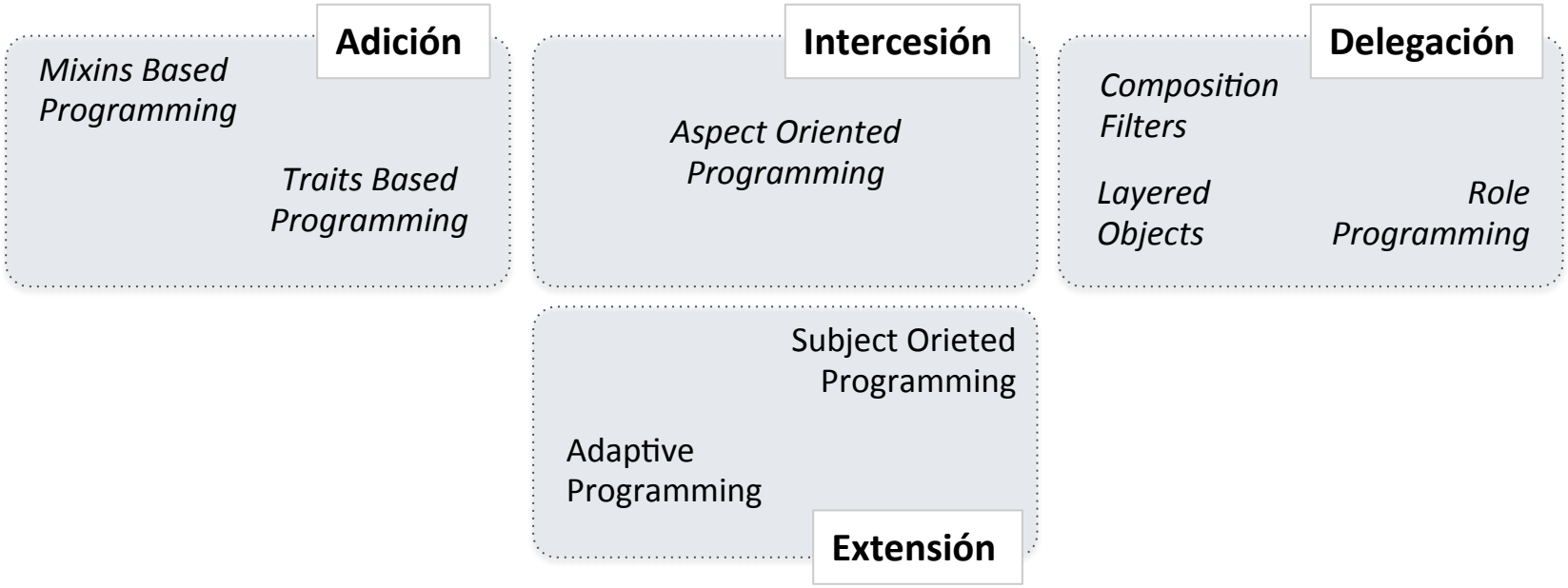
4 Modelos Arquitectónicos De Composición

- Modelos Arquitectónicos Basados en Adición
- Modelos Arquitectónicos Basados en Intercesión
- Modelos Arquitectónicos Basados en Delegación

Introducción

Modelos Arquitectónicos Según Técnicas Compositivas

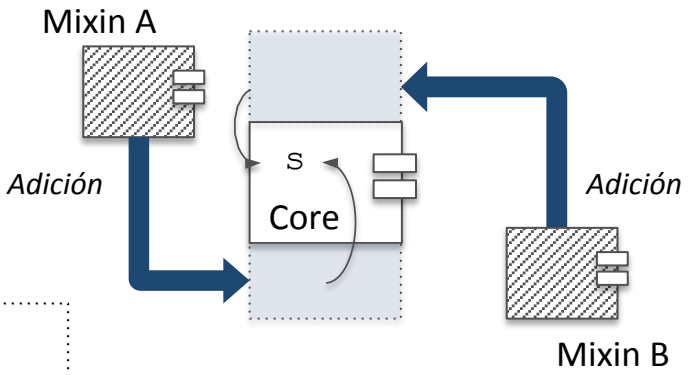
Existen diversos modelos arquitectónicos de metaprogramación compositiva que basan su implementación en cada uno de las cuatro técnicas descritas en el capítulo anterior. En este capítulo describimos dichos modelos.



Modelos Arquitectónicos Basados En Adición

Mixins

Los mixins son extensiones que describen abstracciones de datos parciales para ser añadidos a un core. Ellos agregan el estado al espacio de dicho core.




```
var isAuthor = {
    books: [],
    addBook: function (name) {
      this.books.push(name);
      return this;
    },
    getBooks: function () {
      return this.books;
    }
};
```

```
var isParent = {
    children: [],
    addChild: function (name) {
      this.children.push(name);
      return this;
    },
    getChildren: function () {
      return this.children;
    }
};
```

mp.mixin (john,isAutor)

mp.mixin (john,isParent)

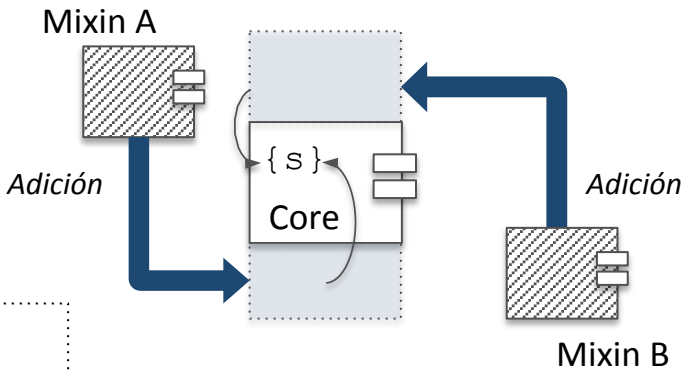
```
var john = {
    first: 'John',
    last : 'Doe'
};
```

```
mp.mixin = function (core, ext) {
  var keys = Object.getOwnPropertyNames(ext);
  keys.forEach (function (key) {
     mp.copy (ext, core, key);
  });
};
```

Modelos Arquitectónicos Basados En Adición

Mixins. Colisión de Estado

Los mixins generan problemas potenciales de colisión de estado ya que varios mixins pueden utilizar los mismos atributos de soporte. Encapsular el estado permite evitarlo.




```
var isAuthor = {
    data: [],
    addBook: function (name) {
      this.data.push(name);
      return this;
    },
    getBooks: function () {
      return this.data;
    }
};
```

```
var isParent = {
    data: [],
    addChild: function (name) {
      this.data.push(name);
      return this;
    },
    getChildren: function () {
      return this.data;
    }
};
```

mp.mixin (john,isAutor)

```
var john = {
    first: 'John',
    last : 'Doe'
};
```

mp.mixin (john,isParent)

```
mp.mixin = function (core, ext) {
  var context = Object.create (null);
  var keys = Object.getOwnPropertyNames(ext);
  keys.forEach (function (key) {
    if (typeof (ext[key]) === 'function') {
      mp.copy (core, ext, key);
      mp.bind (core, key, context);
    }
    else mp.copy (context, ext, key);
  });
};
```

Modelos Arquitectónicos Basados En Adición

Mixins. Self & This

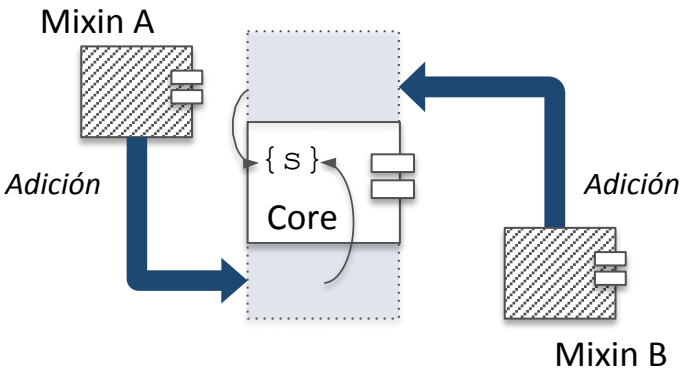
En los mixins con estado protegido existe la necesidad de referir al core desde el cuerpo de sus métodos. Ello requiere meter una referencia al core – self – en el contexto.

```
var isAuthor = {
    data: [],
    addBook: function (name) {
      this.data.push(name);
      return this;
    },
    getBooks: function () {
      return this.data;
    }
    description: function () {
      return 'Books:' + this.getBooks ();
    }
};
```

```
this.self.getBooks ();
```

mp.mixin (john,isAutor)

```
var john = {
    first: 'John',
    last : 'Doe'
};
```

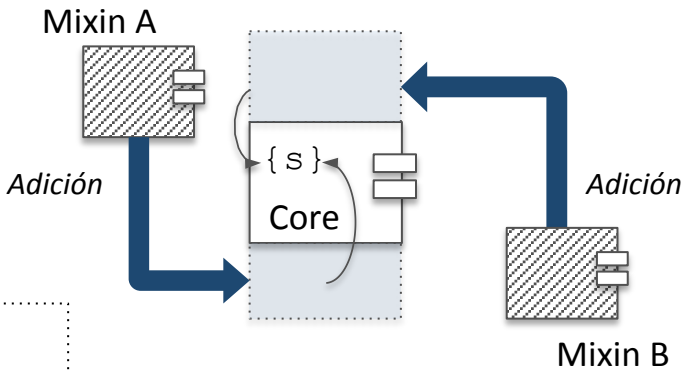



```
mp.mixin = function (core, ext) {
  var context = { self : core };
  var keys = Object.getOwnPropertyNames(ext);
  keys.forEach (function (key) {
    if (typeof (ext[key]) === 'function') {
      mp.copy (core, ext, key);
      mp.bind (core, key, context);
    }
    else mp.copy (context, ext, key);
  });
};
```

Modelos Arquitectónicos Basados En Adición

Mixins. Colisión de Métodos

En los mixins con estado protegido existe la necesidad de referir al core desde el cuerpo de sus métodos. Ello requiere meter una referencia al core – self – en el contexto.




```
var isAuthor = {
    init: function () {
      this.books = [];
    },
    addBook: function (name) {
      this.books.push(name);
      return this;
    },
    getBooks: function () {
      return this.books;
    }
};
```

```
var isParent = {
  init: function () {
    this.children = [];
  },
  addChild: function (name) {
    this.children.push(name);
    return this;
  },
  getChildren: function ()
     return this.children;
  }
};
```

```
var john = {
    first: 'John',
    last : 'Doe'
};
```

mp.mixin (john,isAutor)

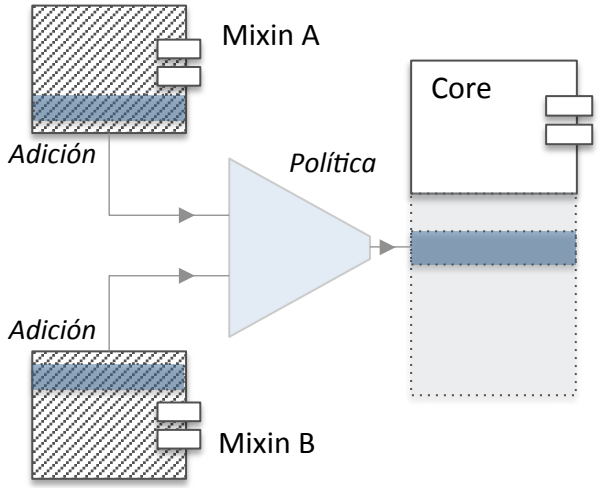
mp.mixin (john,isParent)

```
mp.mixin = function (core, ext, policy) {
  var context = { self : core };
  var decorate = policy || mp.after;
  var keys = Object.getOwnPropertyNames(ext);
  keys.forEach (function (key) {
    if (typeof (ext[key]) === 'function') {
      if (core[key]) {
        decorate (core, ext, key);
        mp.bind (core, key, context);
      } else {
        mp.copy (core, ext, key);
        mp.bind (core, key, context);
      }
    } else mp.copy (context, ext, key);
  });  };
```

Modelos Arquitectónicos Basados En Adición

Mixins. Políticas de Resolución de Conflictos

Para resolver los problemas de colisión de nombres entre métodos de diferentes mixins se puede aplicar una variedad de políticas de gestión de colisiones. Estás políticas aplican en esencia técnicas de entrelazado funcional entre los métodos en conflicto. A continuación mostramos aquellas más prototípicas.



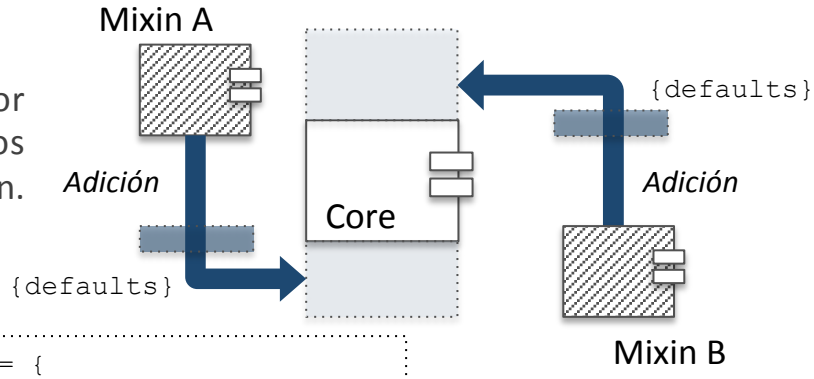

```
mp.override = function (core, key, ext) {
    core[key] = function () {
        return ext.apply (this, arguments);
    };
},
mp.discard = function (core, key, ext) {
    var fn = core[key];
    core[key] = function () {
        return fn.apply (this, arguments);
    };
},
mp.before = function before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

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

Modelos Arquitectónicos Basados En Adición

Mixins. Colisión de Parámetros

Dado que los métodos sinónimos se resuelven por intercesión es necesario discriminar qué parámetros corresponden a cada método en conflicto tras la intercesión. Para ello pasamos a un esquema nominal.




```
var isAuthor = {
    init: function (bks) {
      this.books = bks || [];
    },
    addBook: function (
    init: function (defaults) {
      defaults = defaults || {};
      this.books = defaults.books || [];
    },
    }
};
```

```
var isParent = {
    init: function (chs) {
      this.children = chs || [];
    },
    addChil          on (name) {
    init: function (defaults) {
      defaults = defaults || {};
      this.children = defaults.children || [];
    },
    }
};
```

```
var john = {
    first: 'John',
    last : 'Doe'
};
```

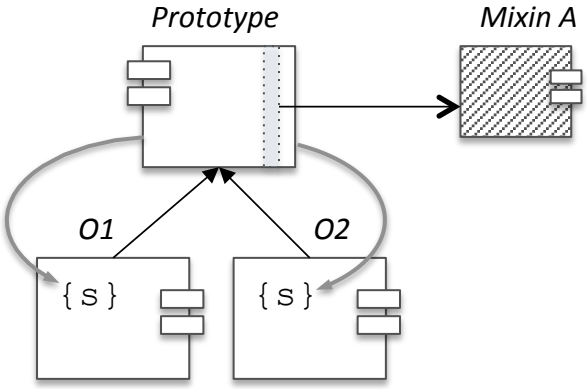
mp.mixin (john,isAutor)

mp.mixin (john,isParent)

Modelos Arquitectónicos Basados En Adición

Mixins. Adición sobre Prototipos

Cuando se crean prototipos por mixtura el estado privado se comparte entre todos los hijos. Para solucionar este problema debemos modificar el metaprograma para crear un estado en cada hijo.



Person

```
var isAuthor = {
    init: function () {
      this.books = [];
    },
    addBook: function (name) {
      this.books.push(name);
      return this;
    },
    getBooks: function () {
      return this. Books;
    }
};
```

```
var isParent = {
    init: function () {
      this.children = [];
    },
    addChild: function (name) {
      this.children.push(name);
      return this;
    },
    getChildren: function () {
      return this.children;
    }
};
```

```
var john = {
    first: 'John',
    last : 'Doe'
};
```

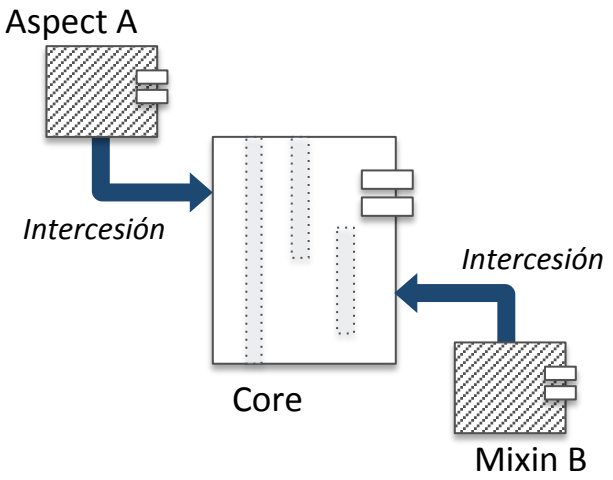
```
var Person = Object.create (null)
mp.mixin (Person, isAutor)
mp.mixin (Person, isParent)
var john = Object.create (Person)
```

```
mp.mixin = function (core, ext, policy) {
  var context = mp.key ();
  core[context] = { self: core };
  var decorate = policy || mp.after;
  var keys = Object.getOwnPropertyNames(ext);
  keys.forEach (function (key) {
    if (typeof (ext[key]) === 'function') {
      if (core[key])
        decorate (core, key, ext[key]);
      else
        mp.outerDelegate (core, ext, key,
                                  core[context]);
    }
    else mp.copy (core[context], ext, key);
  });
};
```

Modelos Arquitectónicos Basados En Intercesión

Aspectos. Intercesión Estática

Los aspectos son abstracciones funcionales encapsuladas que se aplican transversalmente a los métodos miembros de un core. La política de intercesión empleada forma parte de la encapsulación.




```
var checkable = {
  add: {
    when: 'before',
    advice: function (id, v) {
     if (typeof (v) !==
'number')
        throw 'Invalid Type';
    }
  }
};
```

```
var loggable = {
  add: {
    when: 'before',
    advice: function (id, value) {
      console.log ('Adding', id);
    }
  },
  find: {
    when: 'after',
    advice: function (id) {
      console.log ('Find', id);
    }
  }
};
```

```
var manager = {
  data : {},
  find : function (id) {
    return this.data[id];
  },
  add : function (id, value) {
    this.data[id] = value;
  }
};
```

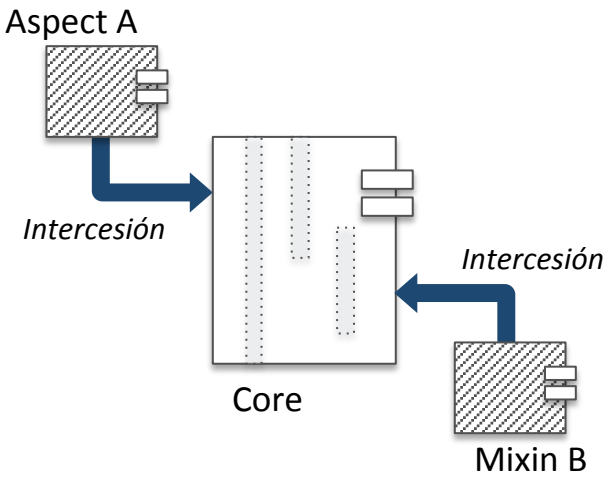
```
mp.sAspect = function (core, ext){
  var ks = Object.getOwnPropertyNames(ext);
  ks.forEach (function (key) {
     var m = ext[key];
     mp[m.when](core, key, m.advice);
  });
};
```

```
mp.aspect (manager, checkable);
mp.aspect (manager, loggable);
```

Modelos Arquitectónicos Basados En Intercesión

Aspectos. Intercesión Dinámica

Los aspectos son abstracciones funcionales encapsuladas que se aplican transversalmente a los métodos miembros de un core. La política de intercesión empleada forma parte de la encapsulación.




```
mp.advisable = function (core, ext, key) {
    var core[key] = function () {
      var args = [].slice.call (arguments);
      method.befores.forEach (function (fn) {
         fn.apply (this, args);
      });
      method.body.apply (this, args);
      method.afters.forEach (function (fn) {
         fn.apply (this, args);
      });
    }
    core[key].isAdvisable = true;
    core[key].befores = [];
    core[key].body = core[key];
    core[key].afters = [];
    core[key].before = function (fn) {
        this.befores.unshift (fn); };
    method.after = function (fn) {
        this.afters.push (fn); };
    core[key] = method;
};
```

```
mp.dAspect = function (core, ext){
 var ks= Object.getOwnPropertyNames (ext);
 ks.forEach (function (key) {
   if (!core[key].isAdvisable)
     mp.advisable (core, ext, key);
   var m = ext[key];
   m[ext[key].when](ext[key].advice);
 });
};
```

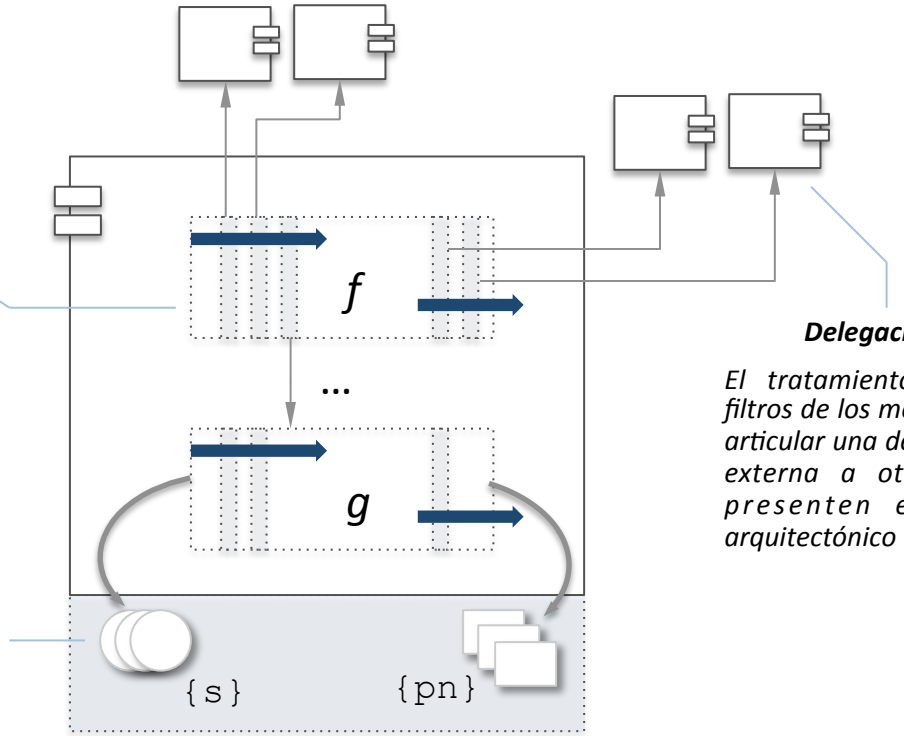
Modelos Arquitectónicos Basados En Delegación

Filtros de Composición

El modelo arquitectónico de filtros compositivos persigue enriquecer cada objeto por intercesión para articular delegaciones que gestionen en tratamiento de mensajes.

Filtros de Entrada y Salida

Cada método se decora con filtros secuenciales de intercesión en before y after que persiguen alterar la semántica del mismo gestionando el tratamiento de los mensajes



Delegación Interna y Externa

El tratamiento que hacen los filtros de los mensajes consiste en articular una delegación interna o externa a otros componentes presenten en el contexto arquitectónico

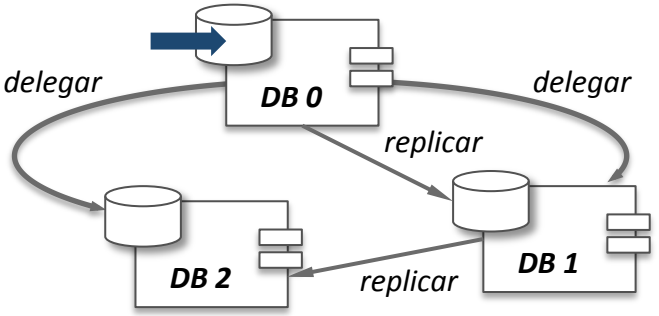
Estado y Predicados

Los filtros determinan las acciones de gestión de mensajes a realizar en función de cierto estado y predicados de control que se añaden al componente

Modelos Arquitectónicos Basados En Delegación

Filtros de Composición

Los aspectos son abstracciones funcionales encapsuladas que se aplican transversalmente a los métodos miembros de un core. La política de intercesión empleada forma parte de la encapsulación.



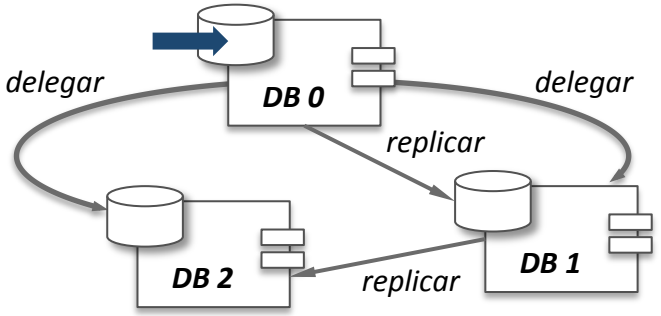

```
var fRound = function (j, dbs){
 return {
   state: {
    active: false,
    dbs: dbs,
    next: (j+1) % dbs.length
   },
   advices: {
    get: {
     when: 'provided',
     advice: function (id) {
      if (this.active) {
        this.active = false;
        this.dbs[this.next].active = true;
        return true;
      } else {
        this.dbs[this.next].get(id);
        return false;
      } ...
};
```

```
var fCopy = function (j){
 return {
   state: { },
   advices: {
     set: {
       when: 'before',
       advice: function (id, obj) {
         if (j < this.dbs.length - 1)
            this.dbs[j+1].set(id, obj);
       }
     }
   }
 };
};
```

Modelos Arquitectónicos Basados En Delegación

Filtros de Composición. Colaboración entre Filtros

Los aspectos son abstracciones funcionales encapsuladas que se aplican transversalmente a los métodos miembros de un core. La política de intercesión empleada forma parte de la encapsulación.




```
var db = {
    get: function (id){...},
    set: function (id, obj){...}
};
var dbs = [Object.create (db),
           Object.create (db),
           Object.create (db)];
dbs.forEach (function (db, i, dbs) {
   mp.filter (db, fRound (i, dbs));
   mp.filter (db, fCopy (i));
});
```

```
mp.filter = function (core, ext) {
  mp.extend (core, ext.state);
  mp.sAspect (core, ext.advices);
};
```

Construcción

Se inicializa el array de bases de datos y se establecen los filtros

```
dbs[0].active = true;
dbs[0].set (1, 1);
dbs[0].set (2, 2);
dbs[0].set (3, 3);
dbs[0].get (1);
dbs[0].get (2);
```

```
[1][1][1]
[1,2][1,2][1,2]
[1,2,3][1,2,3][1,2,3]
Getting 1 from DB0...
Delegating Get to DB1...
Getting 2 from DB1
```

Funcionamiento

Siempre se ataca la base de datos DB0. Cada operación set propaga una replica a la siguiente base de datos. La gestión de queries se resuelve en round robin

Metaprogramación En JavaScript

Preguntas

Técnicas de Composición

Mecanismos de Composición

Modelos Arquitectónicos

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Programación Orientada a Componentes

Metaprogramación En JavaScript

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Febrero 2015