Programación Orientada a Componentes

Arquitecturas Para La Reutilización en JavaScript

Javier Vélez Reyes

@javiervelezreye
www.javiervelezreyes.com

Mayo 2016

Sobre Mí

Licenciado por la UPM desde el año 2001 y doctor en informática por la UNED desde el año 2009, Javier conjuga sus labores como profesor e investigador con la consultoría y la formación técnica para empresa. Su línea de trabajo actual se centra en la innovación y desarrollo de tecnologías para la Web. Además realiza actividades de evangelización y divulgación en diversas comunidades IT y es coordinador de varios grupos de ámbito local como NodeJS Madrid o Madrid JS. Forma parte del programa Polymer Polytechnic Speaker y es mentor del capítulo de Madrid de Node School.

javier.velez.reyes@gmail.com
@javiervelezreye
linkedin.com/in/javiervelezreyes
gplus.to/javiervelezreyes
jvelez77
javiervelezreyes
youtube.com/user/javiervelezreyes

www.javiervelezreyes.com

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

1 Introducción

- Qué es la Reutilización
- El Fracaso de la Reutilización
- Hacia una Verdadera Reutilización
- Arquitecturas para la Reutilización

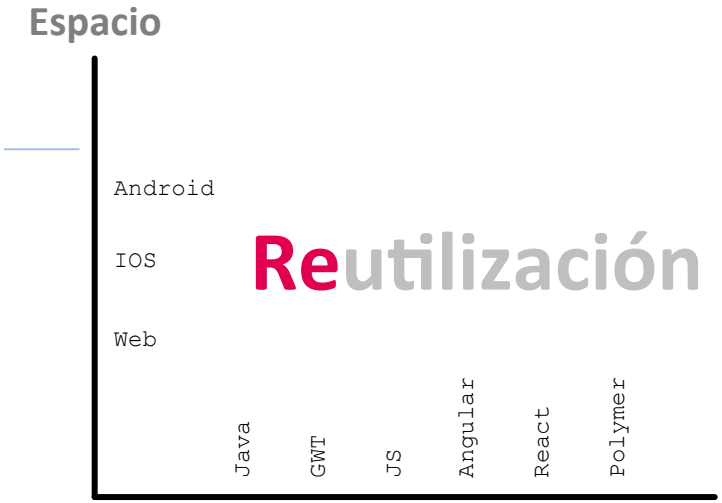
Qué es la Reutilización

La reutilización como ahorro de costes

> La reutilización de código es el proceso por el cual la creación de nuevos sistemas se realiza a partir de artefactos previamente elaborados con el ánimo de reducir costes, tiempos y esfuerzos de desarrollo.

Qué es la Reutilización

Reutilización en el tiempo y en el espacio



Write Once Run Everywhere

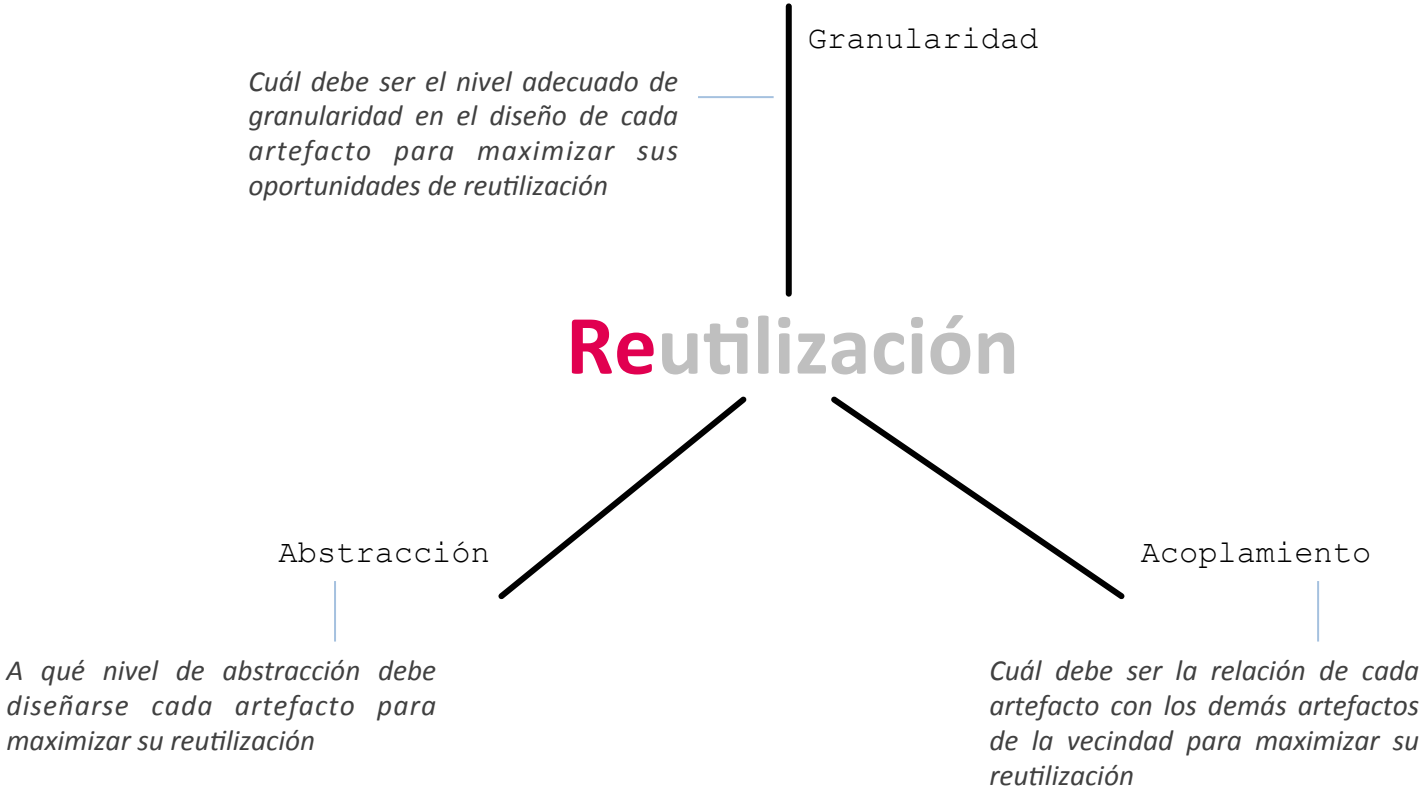
Se requieren mecanismos de estandarización que alcancen la abstracción necesaria para crear esfuerzos de desarrollo agnósticos de tecnología

Write Once Run Whenever

Se requieren criterios de diseño que permitan construir software que supere los dictámenes de la tecnología en uso o en hype

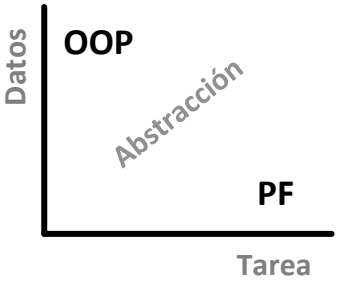
El Fracaso de la Reutilización

Las tensiones de la reutilización

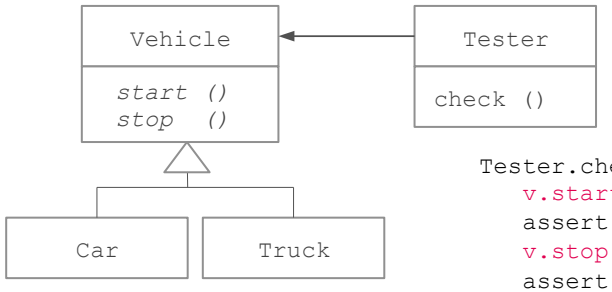


El Fracaso de la Reutilización

Las reutilización desde la abstracción



OOP. Abstracción centrada en datos



Reutilización restringida a variantes

Se consigue reutilizar el esquema del Tester sólo en el marco definido por los objetos de la jerarquía de herencia que participan con éste

```
Tester.check = function (v) {
   v.start ();
   assert (v.started, true);
   v.stop ();
   assert (v.started, false);
};
```

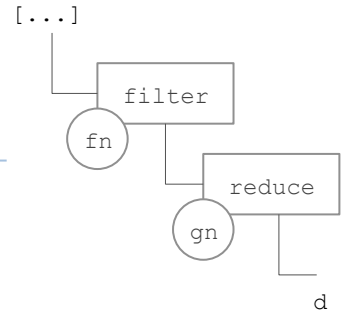
PF. Abstracción centrada en la Tarea

```
var filterReduce = function (data, fn, gn, b) {
  return data
    .filter (fn)
    .reduce (gn, b);
};

var adults = filterReduce (users,
  function (user) { return user.age > 18; },
  function (user) { return 1; },
  0);
```

Reutilización restringida a tareas

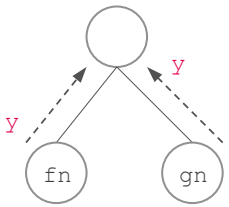
Se consigue reutilizar el esquema algorítmico para distintas tareas pero siempre se aplica sobre los mismos tipos de datos



El Fracaso de la Reutilización

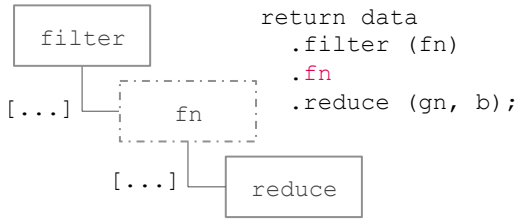
Las reutilización desde el acoplamiento

Acoplamiento a datos



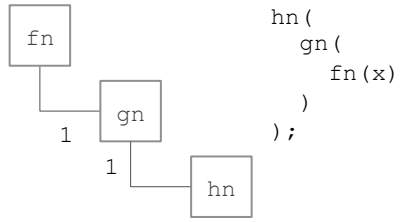

```
var y = fn(x);
return gn(y);
```

Acoplamiento a tipo



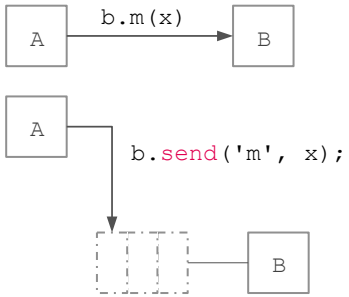

```
return data
  .filter (fn)
  .fn
  .reduce (gn, b);
```

Acoplamiento a aridad

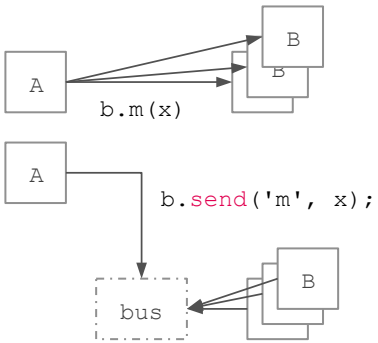



```
hn(
  gn(
    fn(x)
  )
);
```

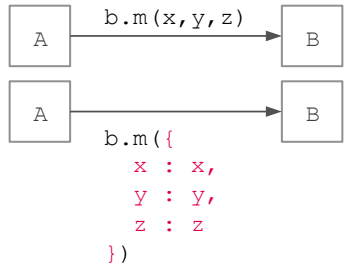
Acoplamiento a tiempo



Acoplamiento a cardinalidad

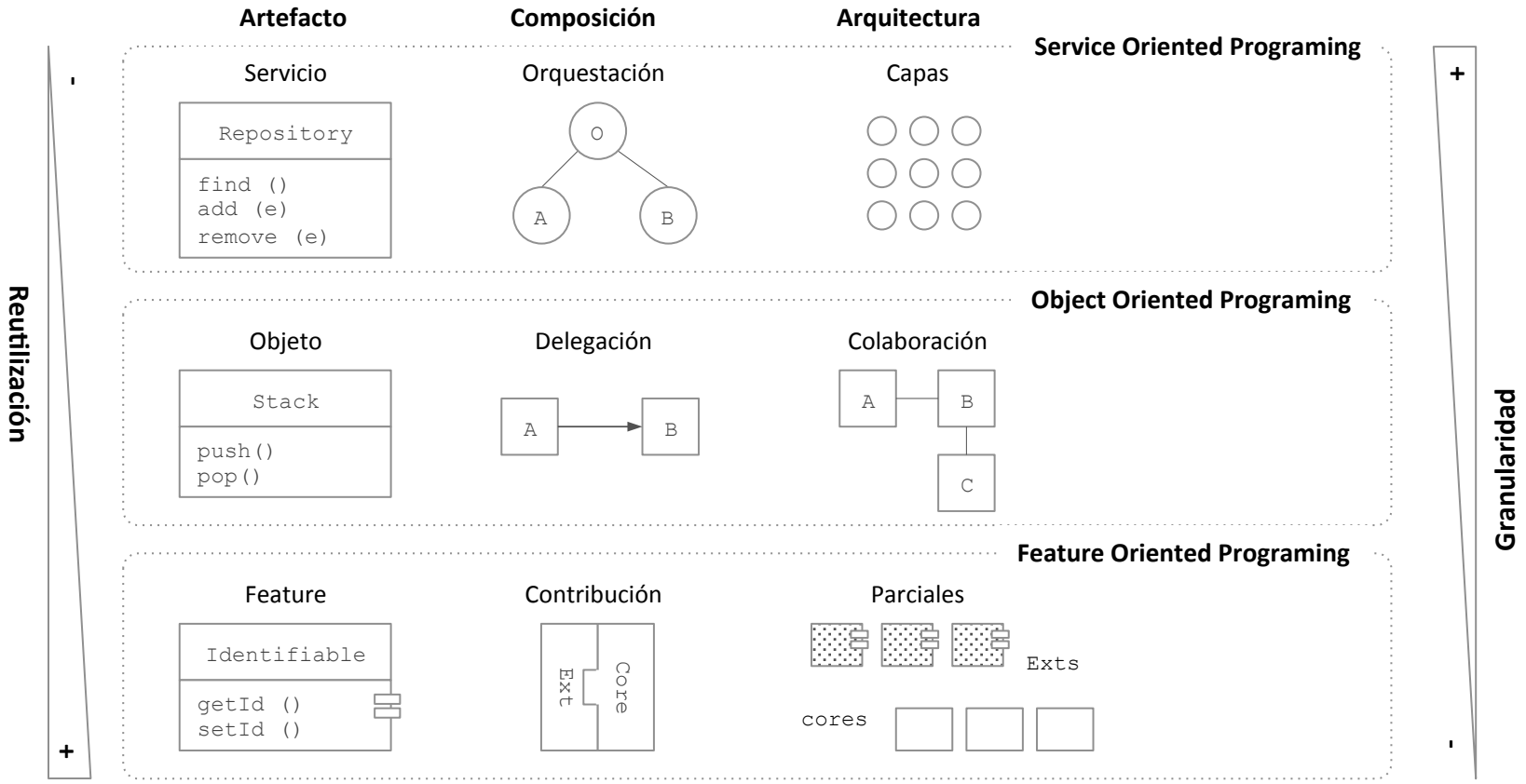


Acoplamiento a volumen



El Fracaso de la Reutilización

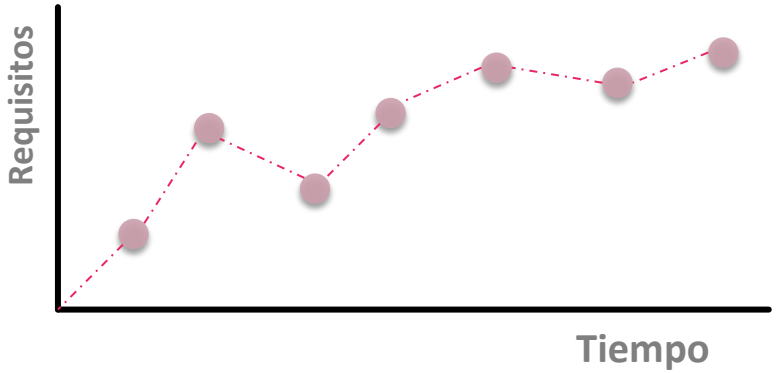
Las reutilización desde la granularidad



Hacia una Verdadera Reutilización

La reutilización por Diseño

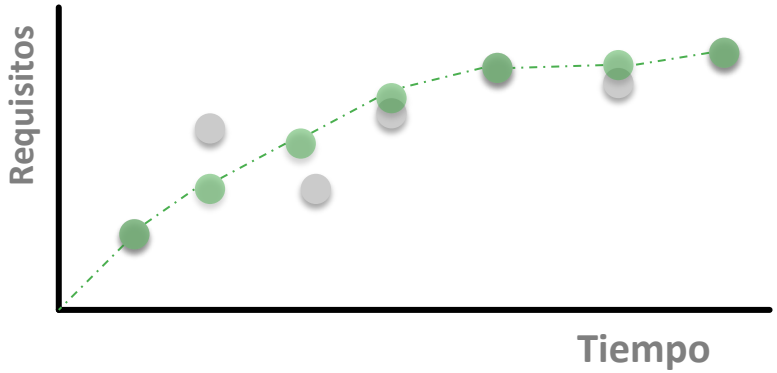
Reutilización por refactorización



El agilismo mal entendido

Se desestima cualquier actividad de análisis y en cada cada sprint se da solución exacta a las demandas del cliente sin atender a los costes del repivotaje

Reutilización por diseño



El agilismo bien entendido

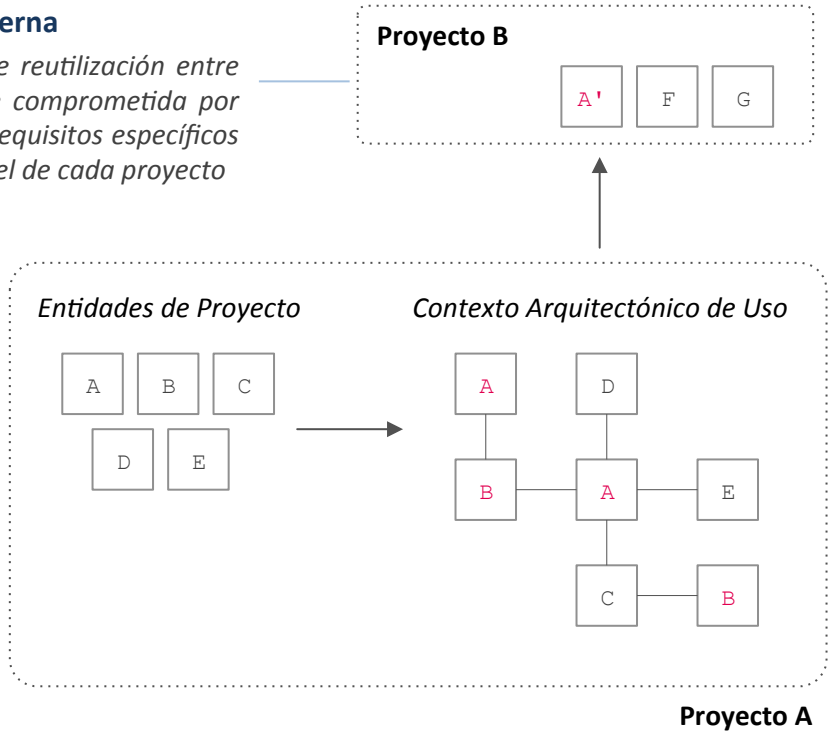
Se dedican esfuerzos de análisis dirigidos a inferir la tendencia evolutiva del software de manera que pueda orientarse el desarrollo para minimizar costes de repivotaje

Hacia una Verdadera Reutilización

Arquitecturas clásicas reutilizables

Reutilización externa

La capacidad de reutilización entre proyectos se ve comprometida por el conjunto de requisitos específicos impuestos a nivel de cada proyecto



Reutilización interna

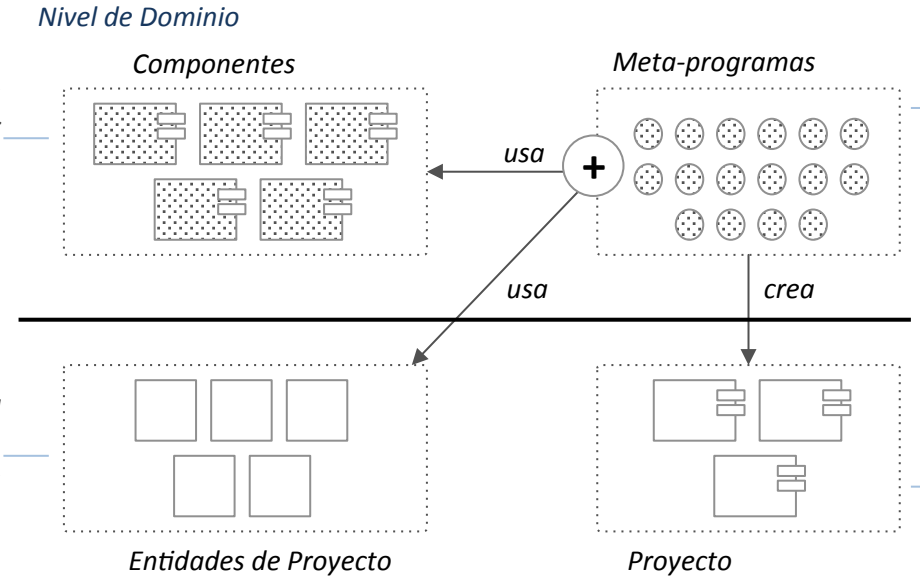
La capacidad de reutilización interna en un proyecto se ve limitada a un pequeño conjunto de contextos arquitectónicos de uso donde cada artefacto se puede usar de forma semánticamente equivalente

Arquitecturas para la Reutilización

Programación Orientada a Componentes

Rasgos reutilizables

Componentes como rasgos parciales que implementar características reutilizables



Esquemas de composición

Meta-programas como esquemas de composición reutilizables de aplicabilidad recurrente

Entidades agnósticas

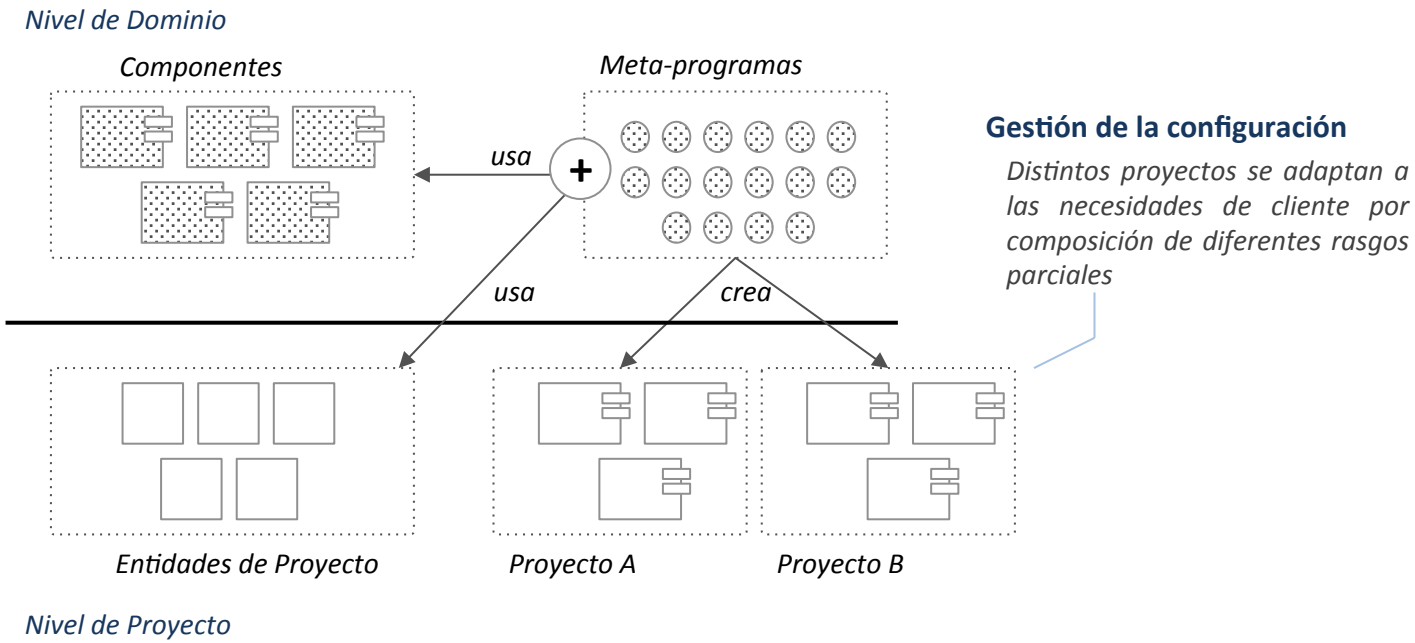
Entidades que capturan el core del negocio de forma agnóstica al uso de rasgos aditivos

Código generado

Proyecto generado a partir de especificaciones de proyecto y dominio reduciendo esfuerzos de desarrollo

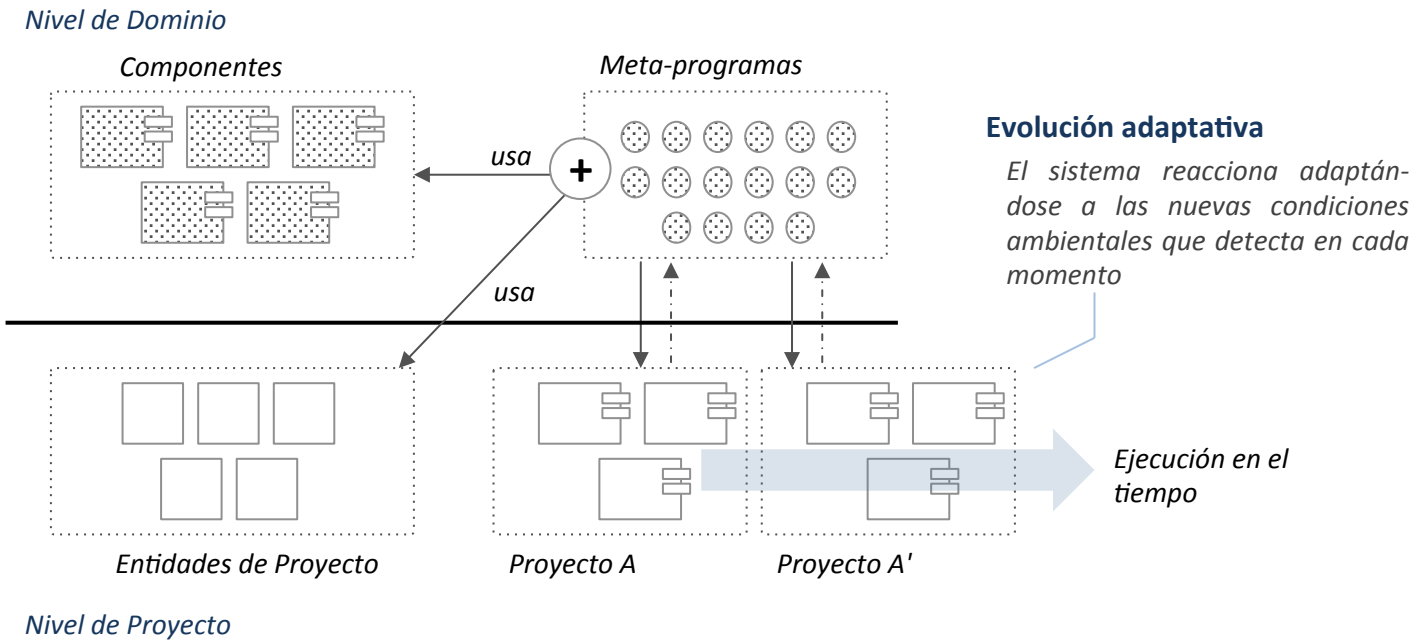
Arquitecturas para la Reutilización

Arquitecturas para la reutilización en el espacio



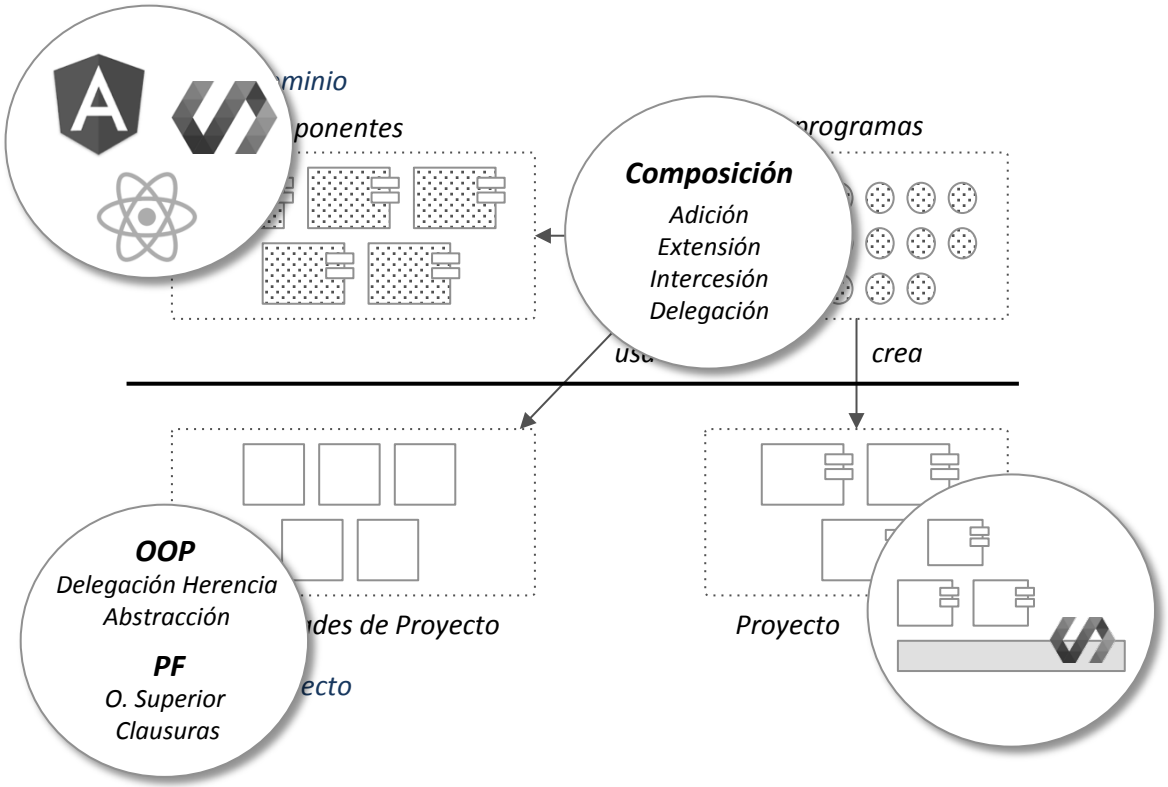
Arquitecturas para la Reutilización

Arquitecturas para la reutilización en el tiempo



Arquitecturas para la Reutilización

En la práctica. Desde Front…

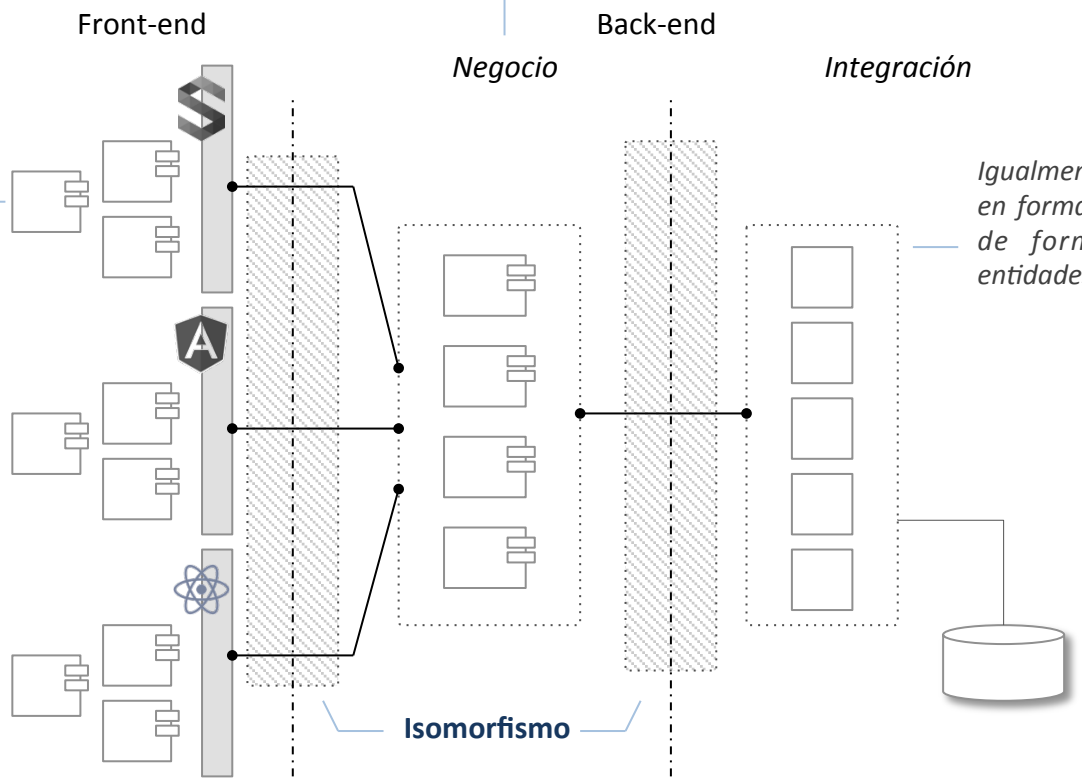


Arquitecturas para la Reutilización

En la práctica. Desde Back…

A. Arquitecturas dirigidas por procesos

El negocio se codifica en forma de componentes que son proxies de los componentes de front y controladores de datos de acuerdo a una aproximación isomorfa



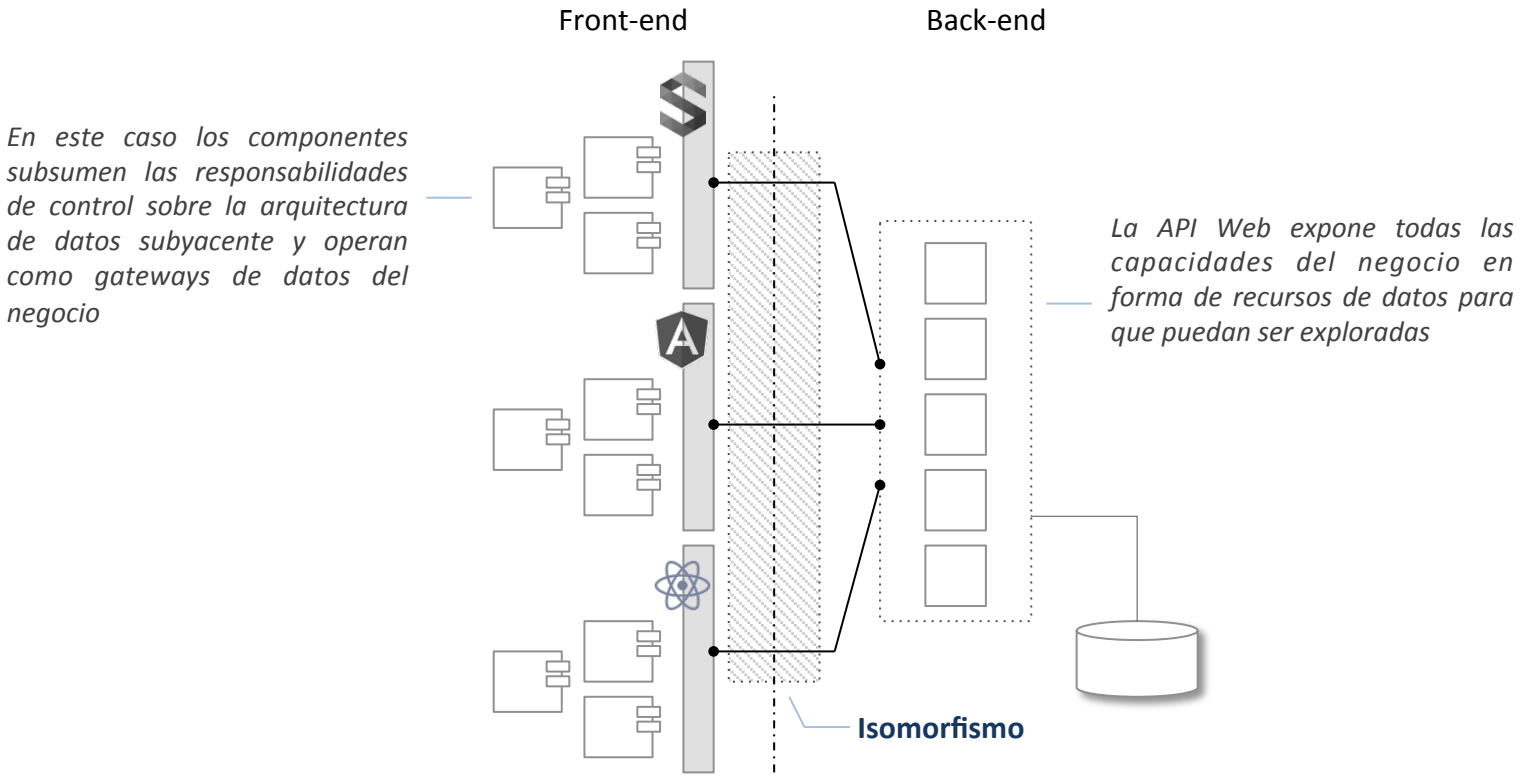
Los componentes de front ofrecen un modelo de interacción para el acceso a las cadenas de valor ofrecidas desde el back

Igualmente los datos expuestos en forma de API Web se alinean de forma isomorfa con las entidades de negocio

Arquitecturas para la Reutilización

En la práctica. Desde Back…

B. Arquitecturas dirigidas por datos



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

2

Arquitecturas Para La Reutilización

- Técnicas de Composición en JavaScript
- Arquitecturas de Mixins & Traits
- Arquitecturas de Subjects & Roles
- Arquitecturas de Aspectos & Filtros

Arquitecturas Para La Reutilización En JavaScript

Arquitecturas Para La Reutilización

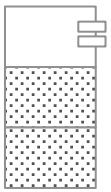
Introducción

Técnicas de Composición

Early Binding

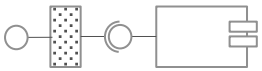
Late Binding

Adición

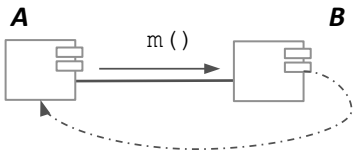


```
var core = {
   x : 1,
   y : 2,
};
core = Object.assign ({}, {
   p : function () { return this.x; },
   q : function () { retu
});
```

Intercesión

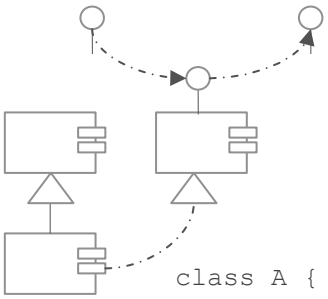


```
var core = { p: function () {...} };
var fn = core.p;
var gn = function () {...};
core.p = function (...args) {
  gn.apply (this, args);
  apply (this, args);
```




```
var core = {
   x : 1,
   p : function () { return this.x; },
})
var ext = { x : 2};
core.p = core.p.bind (ext)
```

Extensión



```
class A { p() {...}, q() {...} }
class B extends A { m() {...}  }
```

Delegación

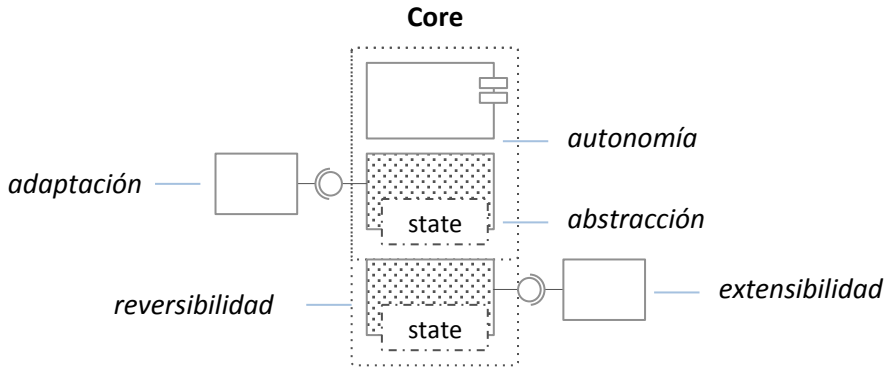
```
   p : function () { return this.x; },
   q : function () { return this.d.m(x); }
};
var ext = { m: function (n) { return 2*n; }};
```

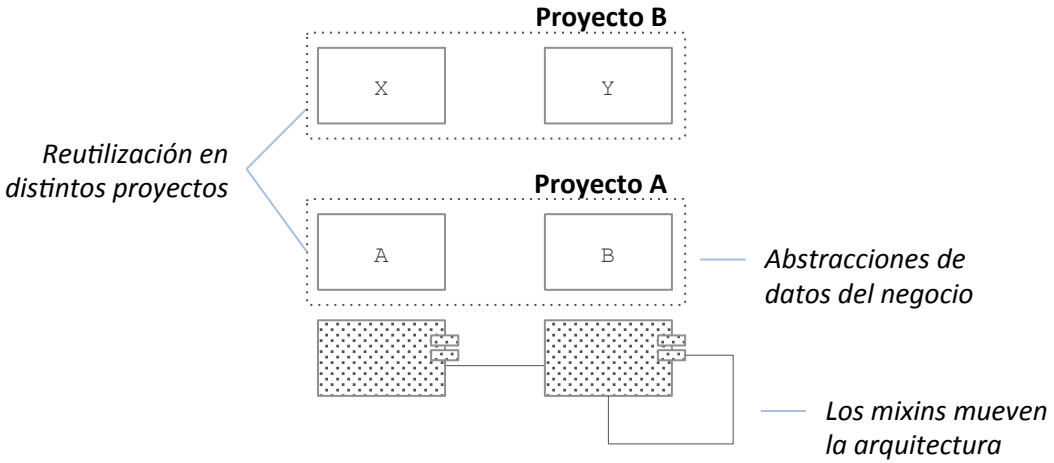
Arquitecturas de Mixins

Descripción General

Componente. Mixin

Un Mixin es una abstracción de datos parcial reutilizable que puede contribuirse a cualquier clase base





Arquitectura

Un modelo de extensiones por mixins mueve una arquitectura de datos con agnosticismo de las entidades de negocio sobre las que opere

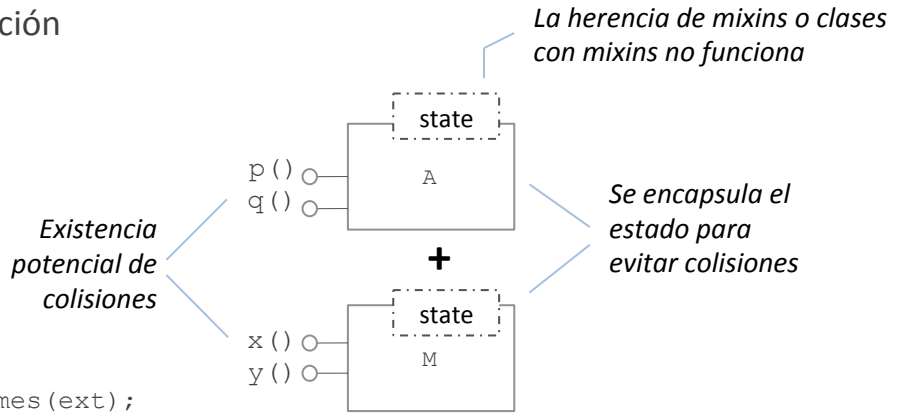
Arquitecturas de Mixins

Implementación I. Composición por Adición

Técnica de Composición

La técnica habitual aplica composición aditiva sobre los mixins. Lamentablemente esta solución no funciona en contextos de herencia

```
mp.Mixin = function (ext) {
   return function (core) {
      var ctx  = Object.create (null);
      var keys = Object.getOwnPropertyNames(ext);
      var mix  = Object.assign({}, core);
      ctx.self = mix;
      keys.forEach (function (key) {
         if (typeof(ext[key]) === 'function')
           mix[key] = ext[key].bind(ctx);
         else ctx[key] = ext[key];
      });
      if (mix.init) mix.init.call(mix);
      return mix;
   };
};
var A  = {...};
var MixinId = mp.Mixin({
    id    : 0,
    getId : function () { return this.id; },
    setId : function (id) { this.id = id; }
});
var B = MixinId(A);
```



Desde el Código

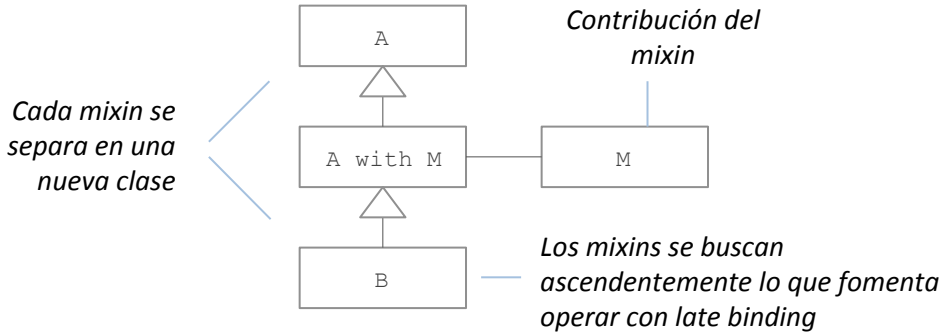
Para garantizar la ausencia de colisiones entre estados de mixins y cores se encapsula el estado de cada mixin. La colisión de métodos debe prevenirse con positivas de exclusión o reescritura

Arquitecturas de Mixins

Implementación II. Composición por Extensión

Técnica de Composición

La técnica de composición por extensión mantiene la independencia de mixins, posibilita la reversibilidad, utiliza late binding y funciona en contextos de herencia




```
let MixinId = function (cls) {
   return class extends cls {
      constructor() {
         super();
         this.id = 0;
      }
      getId() { return this.id; }
      setId(id) { this.id = id; }
};

class A { ... }
Let AM = MixinId(A);
```

Desde el Código

Los mixins ahora son funciones que contribuyen a una clase con nuevas abstracciones de datos. Los problemas de colisión potencial de métodos se mantienen

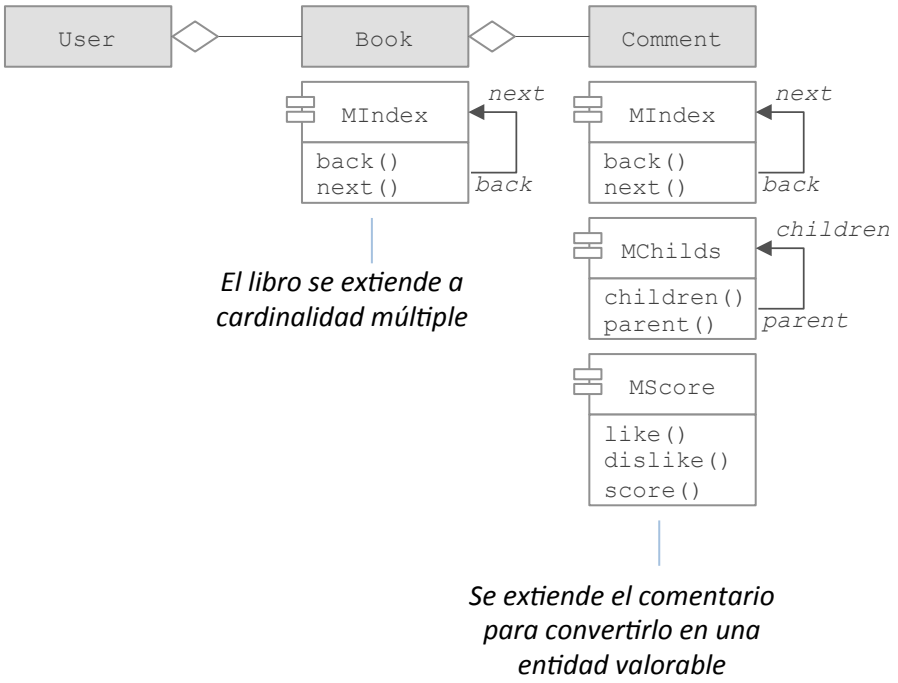
Arquitecturas de Mixins

Ejemplo

Ejemplo

Una solución de mixins mueve una arquitectura para soportar la valoración social de una colección de productos vinculados a un cliente

```
var allScore = function (aProduct, social) {
   var score   = 0;
   var product = aProduct;
   while (product) {
      var social = product[social];
      while (social) {
         score  = score + getScore(social)
         social = social.next();
      }
      product = product.next();
   }
   return score;
}
var getScore = function (social) {
   return !social.children ?
      social.score() :
      social.children.reduce (function (c) {
          return getScore(c) + c.score();
      }, 0);
}
```

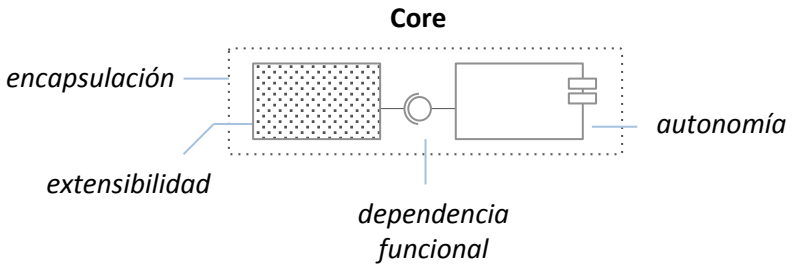


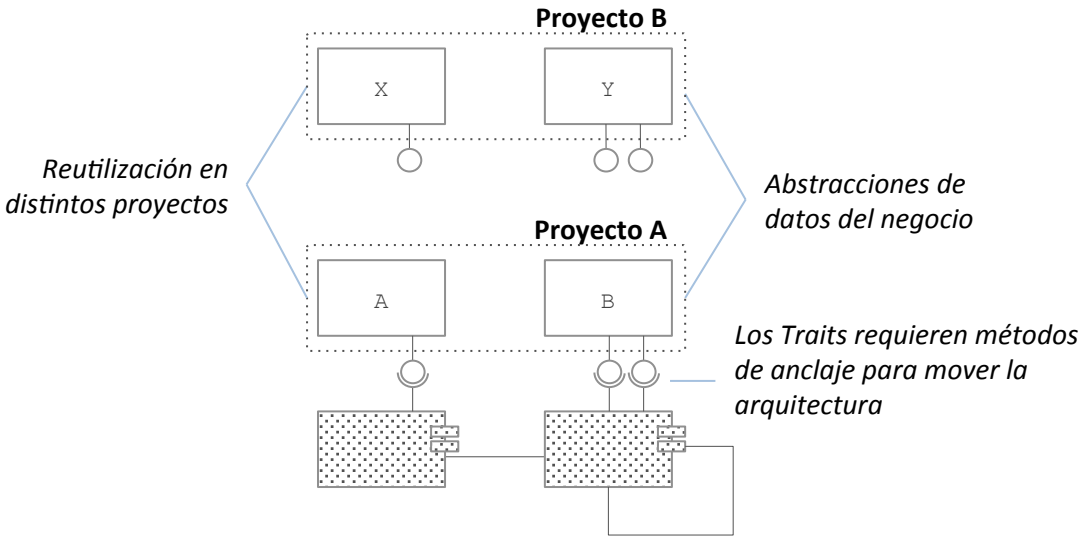
Arquitecturas de Traits

Descripción General

Componente. Trait

Un Trait es una abstracción funcional parcial que contribuye con una colección de capacidades. No tienen estado y se apoya en la existencia de ciertos métodos requeridos en el core





Arquitectura

En este caso la arquitectura de negocio debe proporcionar métodos que permitan a los Traits operar convenientemente.

Arquitecturas de Traits

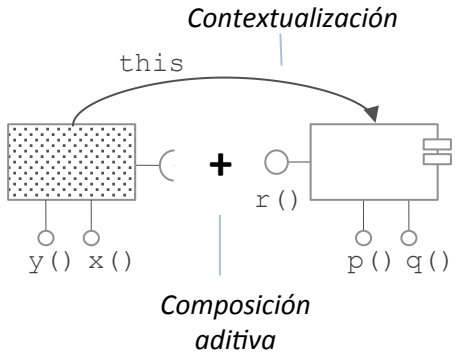
Implementación. Composición por Adición & Recontextualización

Técnica de Composición

La composición por adición es sencilla pero supone operar en early binding. Los Traits se definen como si pertenecieran al core, lo que requiere contextualización

```
mp.Trait = function (ext){
  return function (core) {
      var keys  = Object.getOwnPropertyNames(ext);
      var trait = Object.assign({}, core);
      keys.forEach (function (key) {
         if (typeof(ext[key]) === 'function')
           trait[key] = ext[key].bind(core);
      });
      return trait;
   };
};

var TEnumerable = pm.Trait ({
   map : function (fn) {
      var result = [];
      this.forEach (function (e) { result.push (fn(e)); });
      return result;
   },
   reduce: function (rn, b) { ... },
   filter: function (pn) { ... }
});
```



Desde el Código

Cada método dentro del Trait se contextualiza reasignando el valor del puntero this. Se asume que un rasgo no puede contener otra cosa más que métodos

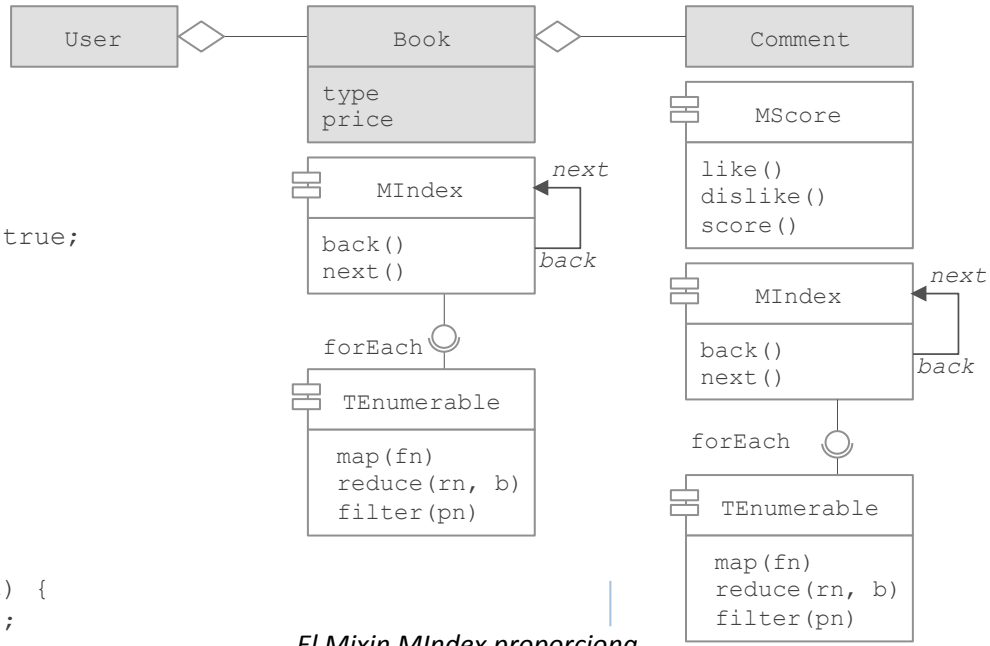
Arquitecturas de Traits
Ejemplo

Ejemplo

Extendiendo el ejemplo anterior queremos saber el coste de la cesta de libros y la valoración social de la misma.

```
var cost = function (user, type) {
   return user.book
     .filter (function (book) {
        return type ? book.type === type : true;
     })
     .reduce (function (price, book) {
        return price + book.price;
     }, 0);
};

var score = function (user) {
   return user.book
     .reduce (function (score, book) {
        return score + book.comment
          .reduce (function (score, comment) {
             return score + comment.score();
          }, 0);
     });
};
```



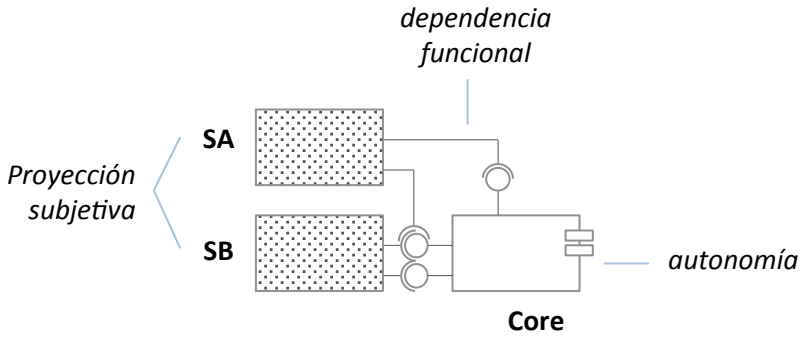
El Mixin MIndex proporciona forEach lo que permite incorporar TEnumerable

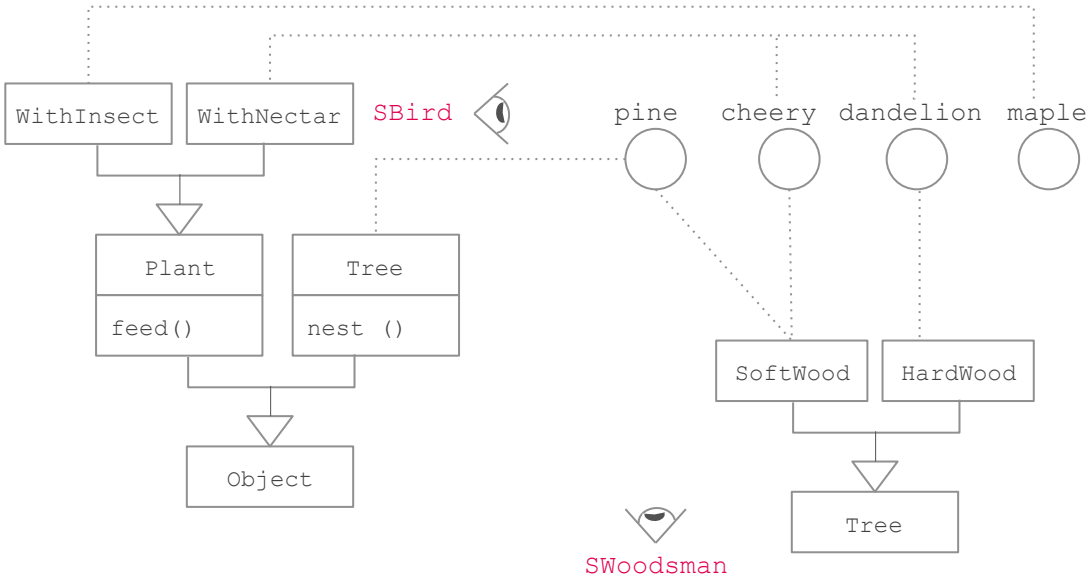
Arquitecturas de Subjects

Descripción General

Componente. Subject

Un Subject es una proyección subjetiva de un modelo de negocio preparada para ser utilizada por un determinado colectivo de desarrolladores dentro del proyecto.





Arquitectura

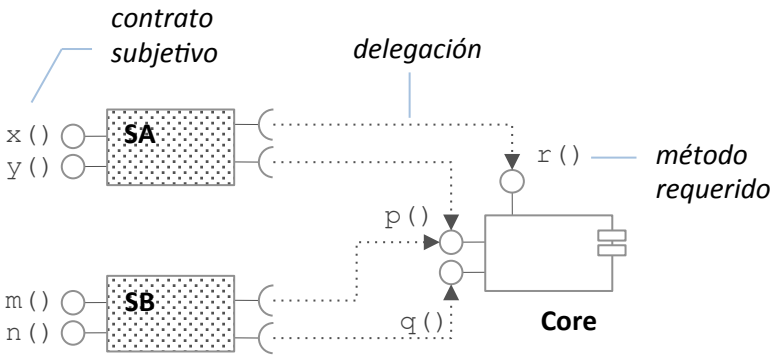
Las arquitecturas subjetivas permiten contemplar un mismo modelo desde perspectivas contrapuestas lo que permite simplificar el desarrollo en proyectos grandes y complejos

Arquitecturas de Subjects

Implementación. Composición por Delegación

Técnica de Composición

Se aplica composición por delegación de manera que cada Subject se mantenga como una proyección simplificada cuyos métodos terminan invocando al modelo real subyacente.




```
mp.Subject = function (cfg) {
   return function (...objs) {
      return Object.keys(cfg)
         .reduce (function (subject, key) {
            var method   = cfg[key];
            var target   = objs[cfg[key].target];
            var feature  = typeof (method)  === 'function' ? method : target[method];
            subject[key] = typeof (feature) === 'function' ?
                feature.bind(target) :
                feature;
            return subject;
         }, {});
};
class A { foo() {...}, bar() {...} }
class B { baz() {...}, qux() {...} }
var S1 = mp.Subject({
  p: {method: 'foo', target: 0}
  q: {method: 'qux', target: 1}
});
var s = S1 (new A(), new B());
```

Desde el Código

Para definir un Subject, se proporciona una descripción de los métodos que contendrá. Esto genera una función que permite activar el Subject sobre un conjunto de objetos

Arquitecturas de Subjects

Ejemplo

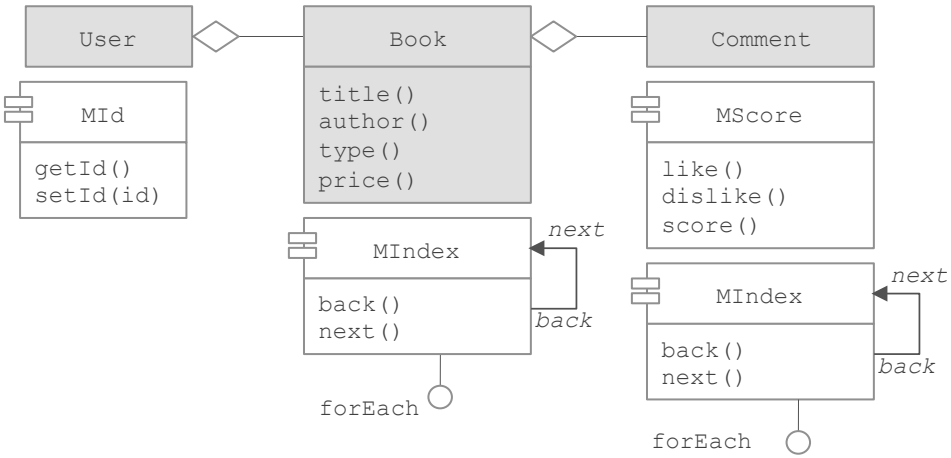
Ejemplo

En este caso realizamos dos proyecciones subjetivas para los departamentos de fidelización y ventas

```
var SLoyalty = mp.Subject({
  title    : {method: 'title',    target: 0},
  author   : {method: 'author',   target: 0},
  comments : {method: 'forEach',  target: 0},
  score    : {method: function () {
      this.reduce(function (score, comment){
         return score + comment.score();
    }, 0);
  }, target: 0}

});
var SSales = mp.Subject({
  price  : {method: 'price',   target: 1},
  owner  : {method: 'getId',   target: 0},
  others : {method: 'forEach', target: 0}
});

var user = ...
var loyalty = SLoyalty (user.book);
var sales   = SSales (user, user.book);
```

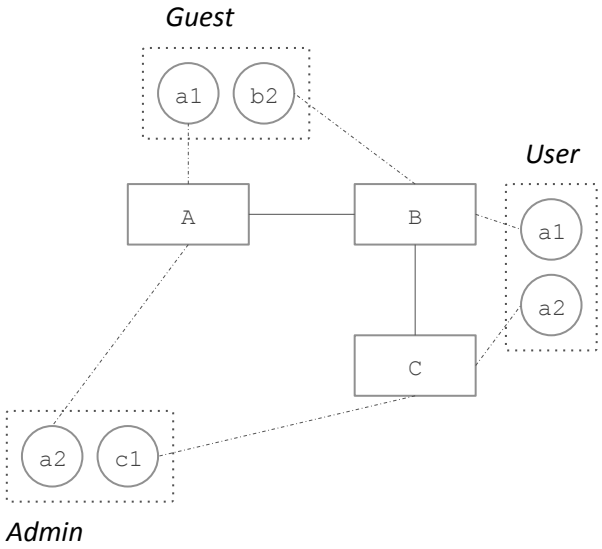


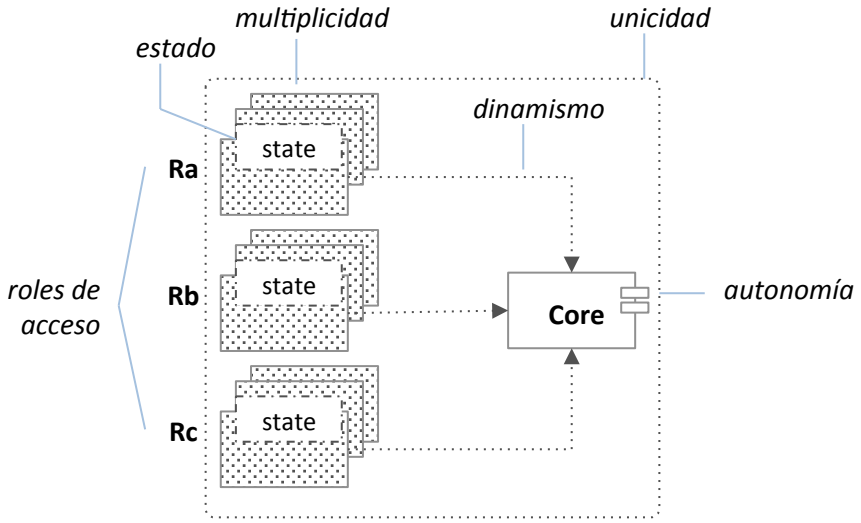
Arquitecturas de Roles

Descripción General

Componente. Role

Un Rol encapsula un perfil de acceso a una entidad que ofrece una forma parcial y restringida de explotar las capacidades funcionales que esta entidad ofrece





Arquitectura

Las arquitecturas de roles permiten modelar mucho más cómodamente los problemas de gestión de recursos con unas políticas de acceso complicadas. A diferencia de los Subjects, en este tipo de arquitectitas cada rol proporciona métodos propios y un estado interno que debe instanciarse

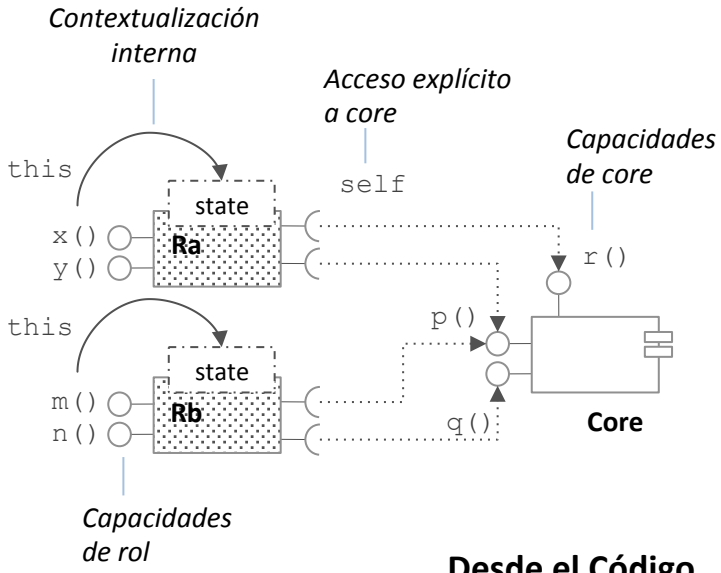
Arquitecturas de Roles

Implementación. Composición por Delegación & Contextualización

Técnica de Composición

Se aplica composición por delegación y contextualización. Cada Role tiene un estado interno con variables primadas y una referencia self al core. De esta manera se soluciona la esquizofrenia de objetos

```
mp.Role = function (core, cfg) {
   return function (...args) {
      var ctx  = Object.create (null);
      ctx.self = core;
      var role = Object.keys (cfg)
        .reduce (function (role, key) {
            role[key] = cfg[key].bind(ctx);
            return role;
        }, {});
      if (role.init) role.init(args);
      return role;
   };
};
var core = { add (x, y) { return x + y; } }
var RA   = mp.Role (core, {
  init : function (x) { this.x = x; },
  inc  : function () { this.self.add (this.x, 1); },
})
var rA1 = RA (1);
var rA2 = RA (2);
```



Desde el Código

La función de definición de un Role recibe el core y los métodos. Ésta contextualiza los métodos sobre un contexto privado ctx para encapsular el estado e incluye una referencia self al core. Se requiere una función init para inicializar el Role en la instanciación

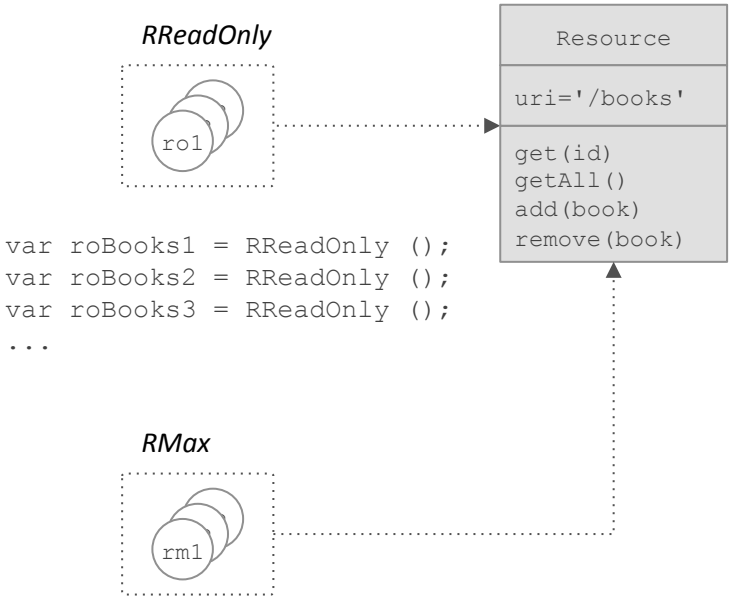
Arquitecturas de Roles

Ejemplo

Ejemplo

Sobre el servicio de acceso a la API Web definimos un ecosistema de roles para operar por cliente con un estado interno

```
var RReadOnly = mp.Role (core, {
  get: function (id) { return this.self.get(id); },
  all: function (k)  { return this.self.getAll(k); }
});
var RMax = mp.Role (core, {
  init: function (max) {
     this.max = max;
     this.n   = 0;
  },
  get: function (id) { return this.self.get(id); },
  add: function (e) {
     if (this.n < this.max) {
        this.self.add(e);
        this.n++;
     } else console.log ('Error - [%s,%j]', k, e);
  },
  reset: function () { this.n = 0; }
});
```

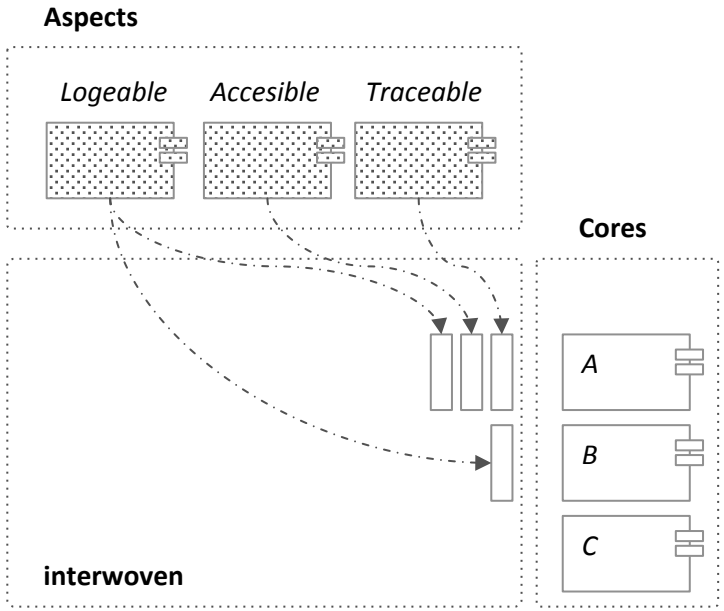


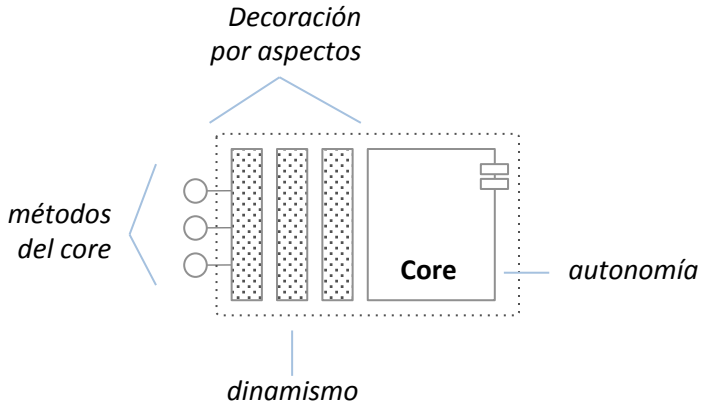
Arquitecturas de Aspectos

Descripción General

Componente. Aspecto

Un aspecto es una proyección parcial de las responsabilidades funcionales que debe cubrir una abstracción para dar respuesta a los requisitos de negocio





Arquitectura

Las arquitecturas de aspectos permiten descomponer los problemas en proyecciones ortogonales que resultan de gran reutilización potencial al poder contribuirse de forma inmediata a distintas entidades de negocio. Son aspectos típicamente el control de acceso, la concurrencia, la distributividad, la trazabilidad, etc. El entrelazado de aspectos conforma el código final del aplicativo

Arquitecturas de Aspectos

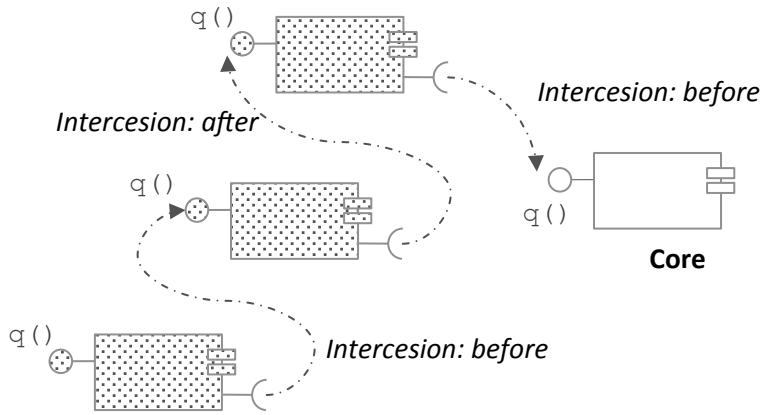
Implementación. Intercesión

Técnica de Composición

La creación de arquitecturas basadas en aspectos entrelazados utiliza técnicas de intercesión sobre funciones y atributos. En JavaScript la intercesión de funciones puede aplicarse repetidamente sobre un mismo core, la de atributos no

```
var aspect = mp.Aspect ({
  p: {
    when   : 'before' | 'after',
    advice : function () {
      ...
  },
  q: {
    when   : 'before' | 'after',
    advice : function () {
      ...
  }
});

aspect (coreA);
aspect (coreB);
aspect (coreC);
```



Desde el Código. Definición

Cada aspecto se define a partir de un advice, función que supone una decoración sobre un método del core, y un punto de croscutting, momento en el cual el decorador debe ser invocado.

Arquitecturas de Aspectos

Implementación I. Intercesión Estática

Desde el Código. Intercesión Estática

La implementación más sencilla consiste en aplicar intercesión directamente sobre los métodos del core. Para ello inicialmente es preciso incluir en la librería los métodos auxiliares en decoración de before y after.

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

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

métodos auxiliares de intercesión

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

La decoración opera en early binding con lo que la decoración no es dinámica ni reversible

Arquitecturas de Aspectos

Implementación II. Intercesión Dinámica

Desde el Código. Intercesión Dinámica

En este caso vamos a aprovechar que en JS cada función es un objeto para incluir propiedades de intercesión en las que delegar

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

Métodos internos de soporte a la intercesión dinámica

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

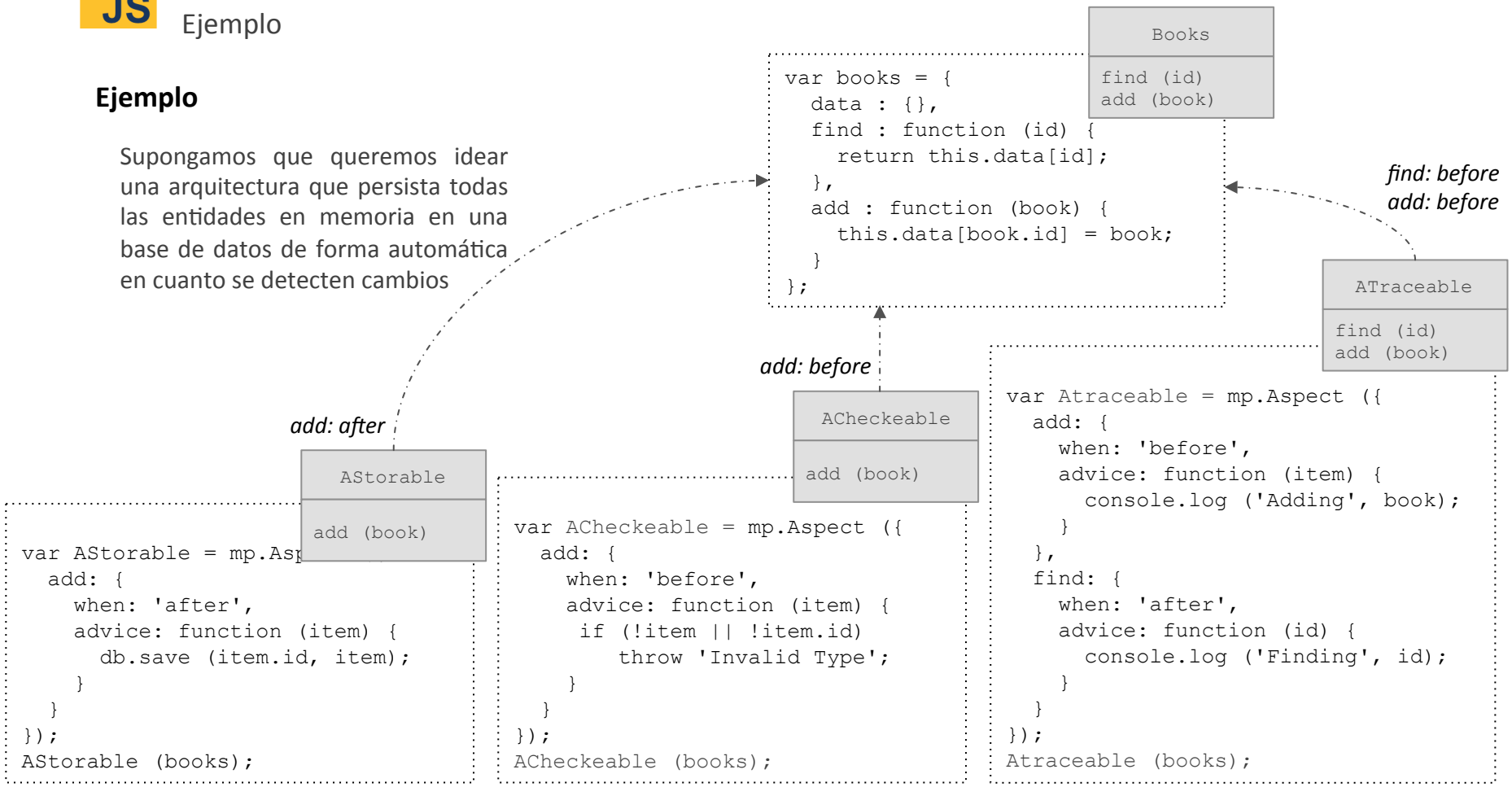
Ahora la decoración opera en late binding con lo que la decoración es dinámica y reversible por modificación de los atributos befores y afters

Arquitecturas de Aspectos

Ejemplo

Ejemplo

Supongamos que queremos idear una arquitectura que persista todas las entidades en memoria en una base de datos de forma automática en cuanto se detecten cambios

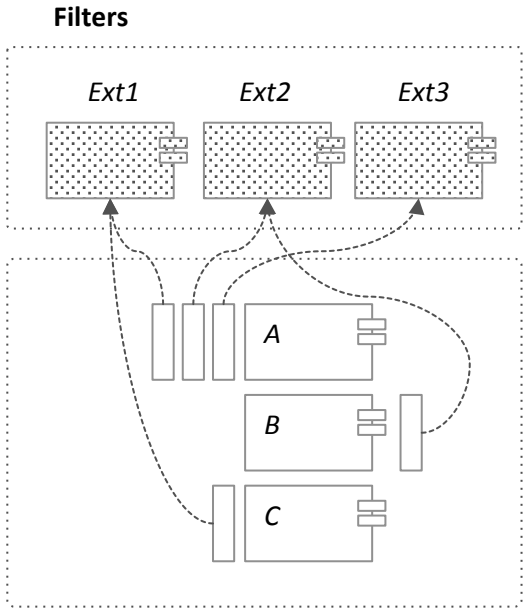


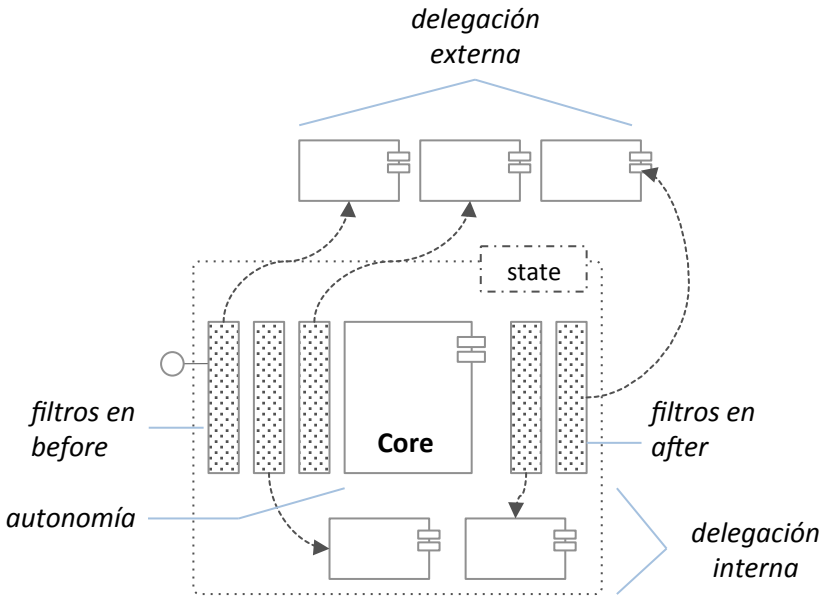
Arquitecturas de Object Filters

Descripción General

Componente. Object Filter

Un filtro es un componente con la capacidad de modificar la gestión natural de invocaciones para un determinado método de manera qué ésta pueda delegarse en otra entidad interna o externa





Arquitectura

Las arquitectura de filtros permiten alterar dinámicamente la gestión de los mensajes de invocación que se reciben sobre cada uno de los métodos de un componente de core. Esto permite aplicar políticas de intervención de forma transparente

Arquitecturas de Object Filters

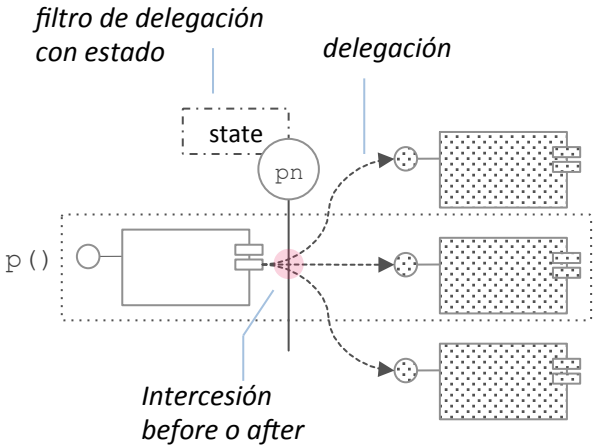
Implementación. Intercesión & Delegación

Técnica de Composición

En este caso se aplica intercesión para poder intervenir la respuesta de cada método con una lógica de dispatching que delegue hacia distintos objetos internos o externos

```
var filter = mp.Filter ({
  init : function () { ... },
  p : {
    when  : 'before' | 'after',
    guard : function () { ... },
    do    : function () { ... }
  },
 q : {
    when  : 'before' | 'after',
    guard : function () { ... },
    do    : function () { ... }
  }
});

filter (coreA);
filter (coreB);
filter (coreC);
```



Desde el Código. Definición

Cada filtro determina qué métodos intervenir y cual es la lógica de intervención que hay que aplicar antes o después del método para delegar en otro componente

Arquitecturas de Aspectos

Implementación I. Intercesión & Delegación

Desde el Código. Intercesión Dinámica

De manera muy similar al caso anterior aplicamos intercesión dinámica. La diferencia aquí es que ejecutamos cada filtro sólo si se cumple la condición de guarda

Se crea un contexto para mantener el estado

Dado que cada filtro tiene acceso al core puede operar con el o comunicarse a través de esquemas de memoria compartida

Se inicializa el estado de cada filtro

Se comprueba la condición de guarda

```
mp.Filter = function (ext){
   return function (core){
      return function (...args) {
         var ctx  = Object.create(null);
         ctx.self = core;
         var keys = Object.getOwnPropertyNames (ext);
         keys.forEach (function (key) {
            if (typeof (ext[key]) === 'object'){
               if (!core[key].isAdvisable)
                  mp.advisable (core, key);
               var m = ext[key];
               core[key][m.when](function (params){
                   if (!m.guard || m.guard && m.guard.apply(ctx, [...params, ...args]))
                      m.do.apply (ctx, [...params, ...args])
               });
            }
            if (ext.init) ext.init.apply(ctx, args);
         });
      };
   };
};
```

Arquitecturas de Aspectos

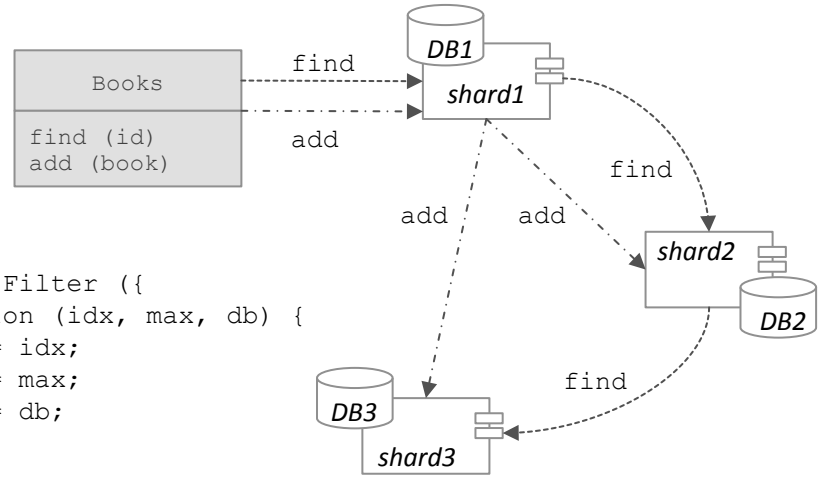
Ejemplo

Ejemplo

Finalmente, supongamos en este último caso que queremos implementar lógica de sharding sobre la base de datos distribuida de libros

```
var Books = {
    find : function (id) {
      return this.data;
    },
    add  : function (book) {}
}

var shardBooks = FShard (books);
shardBooks (0, 3, db0);
shardBooks (1, 3, db1);
shardBooks (2, 3, db2);
```




```
var FShard = mp.Filter ({
    init : function (idx, max, db) {
        this.idx = idx;
        this.max = max;
        this.db  = db;
    },
    find : {
        when  : 'before',
        guard : function (id) {
            return id % this.max === this.idx;
        },
        do : function (e) {
          this.self.data = this.db.get (e.id, e);
        }
    },
    add : {
        when : 'before',
        do   : function (e) {
            this.db.set (e.id, e);
        }
    }
});
```

Arquitecturas Para La Reutilización En JavaScript

Programación Orientada a Componentes

29 y 30 ABRIL 2016
MADRID
JS DayES

www.javiervelezreyes.com

Programación Orientada a Componentes

Arquitecturas Para La Reutilización en JavaScript

Javier Vélez Reyes

@javiervelezreye
www.javiervelezreyes.com

Mayo 2016