Paradigma Lógico

Módulo 6:
Elementos de Diseño:
Delegación.
Acoplamiento.
Code smells.

por Fernando Dodino
Franco Bulgarelli
Mariana Matos
Nicolás Passerini
Gastón Prieto
Alfredo Sanzo

Versión 2.0
Diciembre 2016

Contenido

1 Elementos de Diseño

¿Qué es diseñar?

- definir componentes
- qué responsabilidades tienen
- y qué relaciones hay entre esos componentes

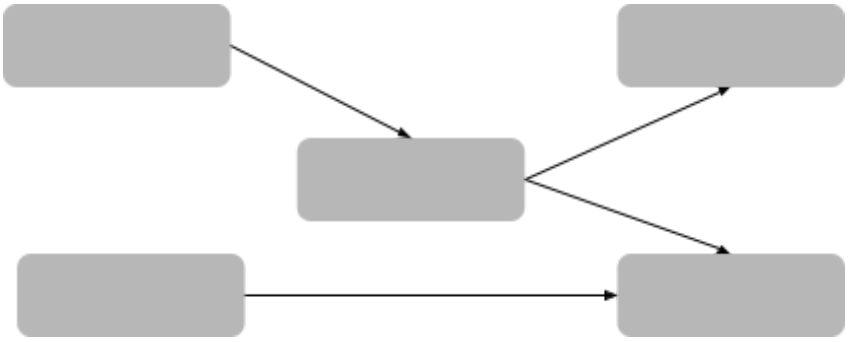


Fig.1: Los componentes en gris y sus relaciones

En particular, dentro del paradigma lógico

- los componentes se implementan mediante predicados e individuos
- ¿qué responsabilidad tiene un componente? para member/2 relacionar un elemento que pertenece a una lista, para forall/2 determinar si todos los individuos que cumplen un predicado satisfacen otra condición

2 Modelando una solución desde cero

2.1 Relevamiento

Por el momento trabajamos con ejemplos en donde los predicados ya estaban definidos. Vamos a completar el siguiente relevamiento implementando una solución dentro del Paradigma Lógico:

Una empresa de remises tiene una flota de 20 autos, y registra todos los viajes que hacen los choferes. Cada chofer puede manejar un auto diferente cada día según la asignación que se le haga. A cada cliente que pide un viaje se le otorga una tarjeta de fidelización donde se le pregunta el nombre completo y se le da un número correlativo.

Necesitamos saber:

- *qué choferes hicieron viajes a Villa del Parque*
- *quiénes manejaron el Volkswagen Gol patente VGT 555 el día 5/2/2016*
- *qué viajes hizo un cliente*

2.2 Modelando entidades

¿Qué entidades queremos modelar?

- Clientes
- Choferes
- Autos
- Viajes

¿y alguno más?

Sí, puede ser qué auto le dieron a qué chofer qué día, un concepto que podemos nombrar Asignación. O bien podemos no tener esa entidad y trabajar esa relación directamente a través del viaje para poder resolver **qué auto le asignaron a un chofer en una fecha**. Es una decisión de diseño...

Sabemos que un viaje relaciona

- un cliente **(así podremos saber qué viajes hizo un cliente)**
- una fecha
- una asignación
    - o bien un auto y un chofer
- la dirección desde donde salimos
- la localidad desde donde salimos
- la dirección de destino
- la localidad de destino **(así podremos saber quiénes fueron a Villa del Parque)**

2.3 Una solución posible

Entonces podemos ver una implementación posible de nuestra solución

```
viaje(cliente(3, sonia), fecha(11, 2, 2017),
      auto(gol, "ZFE447"), matias,
      "Dorrego y Roque Pérez", "Hudson", "Cadorna 283", "Wilde").
```

Y vemos que la forma de representar entidades puede variar:

- el cliente se representa como un functor, al igual que el auto y la fecha
- el chofer es un individuo simple
- los domicilios se representan como strings
- el viaje se está representando como un predicado de aridad 8.

Algunas consideraciones: ¿Podría viaje/8 ser un functor? Sí, pero necesitamos pensar un predicado que lo relacione con otros individuos, por ejemplo la sucursal de la remisería:

```
viajeHecho(viaje(cliente(3, sonia), fecha(11, 2, 2017),
      auto(gol, "ZFE447"), matias,
      "Dorrego y Roque Pérez", "Hudson", "Cadorna 283", "Wilde"),
      sucursal3).
```

En cambio cliente/2 no es un predicado sino un functor que participa de un viaje. Entonces el universo de clientes está dado por el conjunto de clientes que tomaron alguna vez un viaje, no por los potenciales clientes. Volviendo a la definición anterior del predicado viaje/8, si necesito definir un predicado generador para los clientes, esta es una posible definición:

```
clienteReal(Cliente):-viaje(Cliente, _, _, _, _, _, _, _).
```

Lo mismo ocurre con los autos y los choferes.

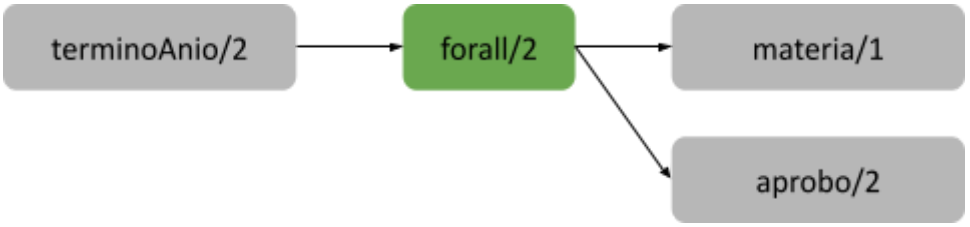
3 Acoplamiento

El acoplamiento es el grado en que los componentes se conocen. Para explicarlo volvamos al predicado que resuelve cuándo un alumno terminó un año:

```
terminoAnio(Alumno, Anio):-
     forall(materia(Materia, Anio), aprobo(Alumno, Materia)).

aprobo(Alumno, Materia):-nota(Alumno, Materia, Nota), Nota >= 6.
```

Si hablamos en términos de componentes:



Hay acoplamiento entre terminoAnio/2 y los predicados materia/2 y aprobo/2. Este acoplamiento es buscado y necesario para poder resolver nuestro requerimiento.

A su vez, el predicado aprobo/2 se basa en el hecho nota/3:

Si la forma de modelar las notas cambia y queremos incluir los aplazos, de

```
nota(raul, pdp, 9).
nota(herminio, pdp, 6).
```

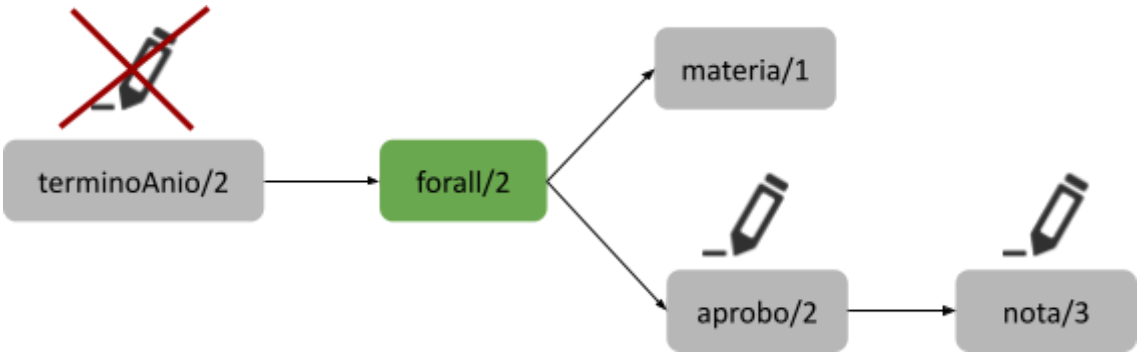
pasamos a

```
nota(raul, pdp, [9]).
nota(herminio, pdp, [2, 2, 6]).
```

¿Qué predicados debemos cambiar en nuestra solución?
aprobo/2, para considerar solamente la última nota:

```
aprobo(Alumno, Materia):-
    nota(Alumno, Materia, Notas),
    last(Notas, Nota),
    Nota >= 6.
```

No es necesario cambiar el predicado terminoAnio/2, ya que aprobó no cambió los individuos con los que trabaja:



Esto muestra que el acoplamiento entre terminoAnio/2 y aprobo/2 es adecuado.

4 Code smells

Es frecuente que al mantener un sistema, lleguemos a una línea de código en la que nos quedamos varios segundos, primero confundidos y luego indignados: ¿quién pudo haber hecho eso? Y a veces descubrimos con gran dolor para nuestro orgullo, que hemos sido nosotros.

Los code smells son señales de que nuestra solución -aun funcionando- debería mejorarse[1]. ¿Cómo? Incorporando abstracciones faltantes, utilizando conceptos más adecuados para construir una solución, haciendo que el código acepte soluciones más generales o sea más expresivo, proceso que se suele llamar **refactorización**.

Para bajar a tierra esto que acabamos de decir, vamos a ver ejemplos concretos.

4.1 Negación vs. Aserción

```
todosSiguenA(Rey) :-
      personaje(Rey),
      not((personaje(Personaje), not(sigueA(Personaje, Rey)))).

sigueA(Alguien, Alguien).
sigueA(lyanna, jon).
sigueA(jorah, daenerys).
%% etc
```

Bueno, el predicado todosSiguenA/1 es inversible y funciona correctamente. Pero ¿qué tan fácil de entender es? Un tanto complejo de decir que “todos siguen a un personaje si no existe otro personaje que no lo siga a él”. En lugar de trabajar con la doble negación, podemos simplificar lógicamente la solución.
∄x / p(x) ∧ ¬q(x)
Se transforma a
∀x / p(x) ⇒ q(x)
“todos siguen a un personaje si ... todos los personajes siguen a ese personaje”

[1] El origen fue acuñado por Kent Beck, quien cuando fue padre recordó el consejo de su abuela: “Cuando el pañal apeste, cambialo” y lo trasladó al código.

```
todosSiguenA(Rey) :-
      personaje(Rey),
      forall((personaje(Personaje), sigueA(Personaje, Rey))).
```

El ejemplo anterior funcionaba y ciertamente no había ningún error conceptual, pero ahora se expone claramente cuándo se satisface el predicado todosSiguenA/1.

4.2 Evitar duplicidades

Una ciudad es interesante si es antigua y tiene más de 10 puntos de interés copados

- un bar es copado si tiene más de 4 variedades de cervezas
- un museo de Ciencias Naturales es copado

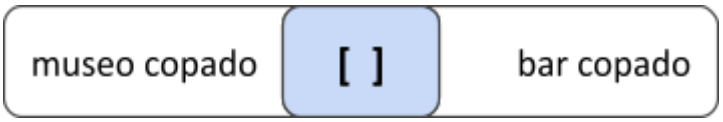
Veamos la siguiente definición:

```
baresCopados(Ciudad, Bares) :-
      findall(bar(CantVarCer), (puntoDeInteres(bar(CantVarCer),
Ciudad), CantVarCer > 4), Bares).

museosCopados(Ciudad, Museos) :-
      findall(museo(cienciasNaturales),
puntoDeInteres(museo(cienciasNaturales), Ciudad), Museos).

ciudadInteresante(Ciudad) :-
      antigua(Ciudad),
      baresCopados(Ciudad, Bares),
      museosCopados(Ciudad, Museos),
      length(Bares, CantidadBares),
      length(Museos, CantidadMuseos),
      CantidadLugaresCopados is CantidadBares + CantidadMuseos,
      CantidadLugaresCopados > 10.
```

Claro en CiudadInteresante estamos sumando en forma individual museos y bares. ¿Qué podemos hacer? Trabajarlos en forma polimórfica



Además CantVarCer tiene un nombre poco representativo, cuando no tengamos el enunciado a mano va a ser difícil recordar qué estaba representando.

```
ciudadInteresante(Ciudad) :-
      antigua(Ciudad),
      cosasCopadas(Ciudad, Cosas),
      length(Cosas, CantidadCosas),
      CantidadCosas > 10.

cosasCopadas(Ciudad, Cosas) :-
     findall(Cosa,   (puntoDeInteres(Cosa,   Ciudad),   copada(Cosa)),
Cosas).

copada(museo(cienciasNaturales)).
copada(bar(VariedadesCerveza)):-VariedadesCerveza > 4.
```

- Quizá en otro lugar estemos necesitando la abstracción del predicado cosasCopadas/2
- Si además de museos y bares hay avenidas, centros culturales y edificios históricos copados, ¿en qué se ve afectado ciudadInteresante/1? En nada. Pero incluso al predicado cosasCopadas/2 tampoco lo afecta. Esta es una métrica real que se relaciona con mantener bajo el acoplamiento.
- Usemos nombres de variables desambiguados.

4.3 Keep it simple

“Resolver el predicado inFraganti/2 que relaciona un delito y un delincuente si un delincuente comete un delito y hay al menos un testigo de ese delito”.

```
inFraganti(Delito, Delincuente) :-
  cometio(Delito, Delincuente),
  findall(Testigo, testigo(Delito, Testigo), Testigos),
  length(Testigos, Cantidad),
  Cantidad > 0.
```

¿Es necesario saber cuántos testigos son? No , solamente saber que hay uno, es un problema de existencia simple.

```
inFraganti(Delito, Delincuente) :-
  cometio(Delito, Delincuente), testigo(Delito, _).
```

Es más sencillo resolverlo de esta manera y funciona exactamente igual (por eso esto que estamos haciendo es un Refactor).

De la misma manera, si necesitamos saber si una persona no tiene hijos, no hace falta que escribamos:

```
noTieneHijos(Persona) :-
   persona(Persona),
   findall(Hijo, padre(Persona, Hijo), Hijos),
   length(Hijos, 0).
```

Porque ese es un truco algorítmico, la regla lógica de una persona que no tiene hijos es $p \wedge \neg q$, donde p = es una persona, q = tiene hijos. Entonces conceptualmente es mejor escribirlo:

```
noTieneHijos(Persona) :-
   persona(Persona),
   not(padre(Persona, _)).
```

Y un hijo único también puede resolverse en forma algorítmica:

```
hijoUnico(Persona) :-
   padre(Padre, Persona),
   findall(Hijo, padre(Padre, Hijo), Hijos),
   length(Hijos, 1).
```

Pero conceptualmente es mejor decir que un hijo único es aquel que no tiene hermanos, o aquel cuyo padre no tiene otro hijo más que él:

```
hijoUnico(Persona) :-
   padre(Padre, Persona),
   not((padre(Padre, Persona2), Persona \= Persona2)).
```

4.4 Keep it simple, again

"Una persona está complicada si tiene algún hijo menor de 18 años problemático"

```
estaComplicado(Persona):-
     findall(Hijo,
            (padre(Persona, Hijo), edad(Hijo, Edad), Edad < 18),
             Hijos),
     member(Hijo, Hijos),
     problematico(Hijo).
```

El error más común para quienes no están acostumbrados a pensar en términos del paradigma es armar listas cuando no son necesarias para la resolución del problema. Esto se pone en evidencia por el uso del findall seguido por un member sobre la lista resultante. El findall arma listas, el member las desarma... ¡son operaciones inversas!

La solución es consultar en forma individual:

```
estaComplicado(Persona):-
   padre(Persona, Hijo),
   edad(Hijo, Edad),
   Edad < 18,
   problematico(Hijo).
```

No solo es más simple la solución, estamos utilizando correctamente las ideas del paradigma.

4.5 Un último ejemplo de Keep it simple

“Un gobernante está feliz si todos los diarios son oficialistas”

Veamos esta solución:

```
feliz(Gobernante):-gobernante(Gobernante),
  findall(UnDiario, diario(UnDiario), Diarios),
  forall(member(Diario, Diarios), oficialista(Diario, Gobernante)).
```

Hay un error conceptual de querer generar la lista de diarios para luego trabajarlos en forma individual. La solución es releer la definición de gobernante feliz e implementarlo según las reglas lógicas: ∀ x / p(x) ⇒ q(x) , donde p = es diario, q = es oficialista.

```
feliz(Gobernante):-gobernante(Gobernante),
  forall(diario(Diario), oficialista(Diario, Gobernante)).
```

En general es una mala práctica el uso de findall seguido de member, ya sea como cláusula individual o dentro de una cláusula forall. Una excepción que hemos visto anteriormente: en la definición de incluido/2

```
incluido(A, B):-forall(member(X, A), member(X, B)).
```

es correcto este uso, porque el dominio **es** una lista.

4.6 Lazy predicate

Esto es frecuente de ver independientemente del paradigma en el que se trabaje, y lógico no es la excepción:

```
estaComplicado(Persona):-personaComplicada(Persona).

personaComplicada(Persona):- ...
```

Saber delegar responsabilidades en un predicado es correcto, pero no está bien que un predicado no agregue valor. Los objetivos de los predicados estaComplicado/1 y personaComplicada/1 son los mismos, entonces hay un predicado que está de más, y hay un problema conceptual de ideas duplicadas.

4.7 Inversibilidad

“Un viejo maestro es aquel pensador en el que todos sus pensamientos llegan a nuestros días”. La definición:

```
viejoMaestro(Pensador) :-
      forall(pensamiento(Pensador, Pensamiento),
      llegaANuestrosDias(Pensamiento)).
```

está ok, solo tiene un detalle: no es inversible. Recordemos que para que un predicado sea inversible tengo que ligar las variables, entonces utilizo un predicado generador:

```
viejoMaestro(Pensador) :-
      pensador(Pensador),
      forall(pensamiento(Pensador, Pensamiento),
      llegaANuestrosDias(Pensamiento)).
```

En todo caso el problema no es conceptual, sino una restricción que impone el motor de inferencia Prolog. Esta solución no calza en sí como un bad smell, o al menos está en una categoría diferente a las anteriores.

4.8 Incógnitas innecesarias

El número de la suerte de una persona es su día de nacimiento y para Joaquín además es 8.

```
numeroDeLaSuerte(Persona, Numero) :-
      diaDelNacimiento(Persona, Numero).
```

```
numeroDeLaSuerte(joaquin, Numero) :-
      Numero is 8.
```

Esto se ve bastante también, el uso de incógnitas innecesarias (lo que en otros paradigmas son variables locales de más). La segunda definición puede acortarse simplemente como:

```
numeroDeLaSuerte(joaquin, 8).
```

4.9 Expresividad

```
obraMaestra(Compositor, Obra) :-
      compositor(Compositor, Obra),
      forall(movimiento(Obra, Movimiento), cumpleCond(Movimiento)).
```

Ah, si el problema es la expresividad, podríamos cambiar el predicado cumpleCond/1 a cumpleCondiciones/1. Pero, ¿qué significa que un movimiento cumple las condiciones? ¿Qué condiciones debe cumplir una obra para ser considerada maestra?

No estamos comunicando adecuadamente las decisiones del *negocio*, necesito ver la implementación del predicado para poder interpretar esa regla, algo que se relaciona con la expresividad de una solución.

4.10 Parametrizar lo que esté fijo

```
puedeComer(analia, Comida) :-
      ingrediente(Comida, _),
      forall(ingrediente(Comida, Ingrediente),
                  (not(contieneCarne(Ingrediente),
                   not(contieneHuevo(Ingrediente),
                   not(contieneLeche(Ingrediente))).

puedeComer(evaristo, asado).
%% etc
```

Una de las cosas que uno podría criticar es la ausencia de un predicado generador **comida/1** que sea más expresivo, pero es un detalle.

Segunda cosa más importante: en lugar de decir

- contieneCarne/1
- contieneHuevo/1
- etc.

podemos incorporar un individuo más a la relación: contiene/2 relacionaría un ingrediente con un elemento (carne, huevo, o cosas que podrían interesarnos, como: azúcar, tacc, sal, etc. para diferentes dietas). Luego, sabiendo qué elementos no son aptos para veganos, ya no necesitamos preguntar explícitamente por un elemento u otro, directamente podemos definir un predicado auxiliar aptoParaVegano/1.

La solución nos quedaría

```
puedeComer(analia, Comida) :-
      comida(Comida),
      forall(ingrediente(Comida, Ingrediente),
             aptoParaVegano(Ingrediente)).

puedeComer(evaristo, asado).
%% etc.

aptoParaVegano(Ingrediente):-
      not((elementoIndeseadoParaVegano(Elemento),
           contiene(Ingrediente, Elemento))).

elementoIndeseadoParaVegano(carne).
elementoIndeseadoParaVegano(huevo).
elementoIndeseadoParaVegano(leche).
```

4.11 Resumen de errores conceptuales

En esta página pueden leerse errores frecuentes que aparecen en el desarrollo dentro del paradigma lógico.

5 Resumen

En el presente apunte no hicimos más que exponer lo que en capítulos anteriores había surgido naturalmente: al resolver un requerimiento hemos tomado decisiones de diseño propias del paradigma lógico como la forma de modelar la información y de relacionar predicados e individuos.

Algunas soluciones se pueden considerar mejores que otras a partir de criterios objetivos, como el grado de acoplamiento que presentan los predicados, la simplicidad de éstos, evitar código duplicado, el uso de afirmaciones por sobre las negaciones, y otras características que se estudian a partir de *antipatterns* llamados *code smells*.