****

**Paradigma**

**Orientado a Objetos**

**Resolución**

**de un Parcial**

**en Objetos**

**Por Fernando Dodino, Valeria Pennella**

**Versión 1.1**

**Mayo 2016**

**Indice**

[1 Enunciado](#_ljavw23ruuu6)

[2 Nuestro objetivo](#_s6p1de75s2n)

[3 ¿Por dónde empezar?](#_u8fudwbl21fc)

[3.1 Tareas de un buen diseñador](#_38t315frjoqv)

[3.2 Encontrando los objetos](#_wvbid3d3gnsu)

[4 Objeto Tarea](#_fbnvkapwvtun)

[5 Alternativas al If](#_ag3y256svbmg)

[6 Programar es humano, delegar es divino](#_1hg6se45gxje)

[7 Responsabilidad y delegación](#_c6iyvgr0vhh3)

[8 Superficie promedio de las tareas](#_l6tbusf8e7kk)

[8.1 Superficie de una tarea](#_sz0pz19itowy)

[9 Saldo de un proyecto](#_579rfx5dtcmw)

[9.1 Si hay que cambiar, se cambia](#_mnzdqzt7du6p)

[9.2 Sobre errores y volantazos](#_py0o5nhdnotv)

[10 Saldo de un proyecto (2° parte)](#_cmqsatu9d6cd)

[11 Sabiendo si se puede hacer una tarea](#_k4uf39kacwlb)

[12 Proyectos y tareas en el ambiente](#_l8453i4ynuyg)

[13 Proyecto coherente](#_bgrj4tndsfnv)

[14 Margen anterior de una tarea](#_q6cyoz1pu4v7)

[15 Qué puede cambiar si estoy haciendo el TP (o en el trabajo)](#_3h3qehd1gaqb)

[15.1 Y nos vamos...](#_ckgy2rj184v3)

[16 Anexo: Diagrama de clases final](#_2bq5vmpgi9iz)

# 1 Enunciado

Desarrollar, de acuerdo al paradigma de objetos y escribiendo el código en lenguaje Wollok, el modelo para una aplicación de manejo de proyectos. La aplicación debe manejar varios proyectos; cada proyecto se compone de un conjunto de tareas. Para cada tarea se indica:

1. fecha en la que se va a hacer; para simplificar el modelo, suponemos que todas las tareas llevan exactamente un día.

b. en dónde se va a hacer:

* algunas tareas se hacen en oficina, se indica la dirección de la oficina,
* algunas tareas se hacen en toda una ciudad, p.ej. acciones de marketing,
* algunas tareas se hacen en una zona rural, p.ej. cosechar en un campo.

c. de qué otras tareas depende, o sea que tienen que hacerse antes.

Es importante tener en cuenta el saldo de un proyecto, que permite que se puedan llevar a cabo sus tareas. Cada proyecto nace con un presupuesto inicial; luego el saldo va variando en función de las tareas que incluye. Hay: tareas de producción, tareas de recaudación, y reuniones.

* Las tareas de producción implican un gasto, que es la suma del costo de los servicios que se usan, que se indica para cada tarea. El costo de cada servicio se indica explícitamente, y es independiente de la tarea: p.ej. si digo que el servicio de desinfección cuesta 45 pesos, ese valor es el mismo para todas las tareas en las que se incluya el servicio de desinfección.
* Las tareas de recaudación implican un ingreso, que se indica explícitamente.
* Las reuniones no implican ni gasto ni ingreso.

Las tareas se cargan, todas, desde un archivo externo, la carga queda fuera de lo que hay que modelar. Lo que hay que hacer es definir qué objetos deben crearse, de forma tal que se pueda cumplir con los requerimientos respecto de los proyectos que ya están cargados con todos sus datos.

Los requerimientos son:

1. Saber en qué provincias va a haber actividad de un proyecto para un rango de fechas. De cada oficina se sabe la ciudad, de cada ciudad y zona rural se sabe la provincia.
2. Saber la superficie promedio en la que se desarrollan las tareas de un proyecto. Para las oficinas se establece una superficie, la misma para todas en m2. Para las ciudades se informa explícitamente la superficie en m2. Las zonas rurales se asumen como rectangulares; se informa ancho y largo en metros.
3. Saber el saldo de un proyecto a una fecha, que debe tener en cuenta todas las tareas hasta esa fecha inclusive.
4. Saber si se puede hacer una tarea; la condición es que todas las tareas de las que depende deben tener una fecha anterior. Para las tareas de producción, además, el saldo del proyecto a la fecha en que se hace la tarea debe ser no negativo (o sea, >= 0).
5. Saber si un proyecto es coherente; es coherente si pueden hacerse todas las tareas en la fecha indicada
6. Saber el margen anterior de una tarea, que es la cantidad de días entre que se hace la última de las tareas de las que depende, y el día indicado para esa tarea.Si la tarea no depende de ninguna, su margen anterior es 0.

# **2 Nuestro objetivo**

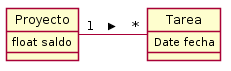
La idea al final del apunte es que se lleven una idea de cómo encarar un ejercicio de Objetos y que estas ideas les sirvan para:

* resolver el Trabajo Práctico
* el examen

Y también… para aplicarlo en el trabajo

# 3 ¿Por dónde empezar?

Leyendo qué nos piden. Entonces se trata de proyectos, tareas, ok. La primera tentación es armar un diagrama de clases y empezar a pensar la estructura interna de los objetos

[](http://www.plantuml.com/plantuml/img/AyxEp2j8B4hCLIWkgSn91P95cSabfRavNCavYSN52W55-PMfoINv2jLS2WffEVc99Gh5YJaf-K1UBLH9I69HQY98sYMn95KXBJKv8v6hbWkkKSbGIK5NGKbB2Q9GmKh13W00)

Entonces tengo que pensar si el proyecto conoce a sus tareas, si la tarea tiene que conocer o no al proyecto, etc. Pero todavía no estoy seguro de muchas cosas, entonces vamos a tomarnos un tiempo para cambiar la óptica del problema que queremos resolver antes de mandarnos a escribir código o a crear diagramas.

**

## 3.1 Tareas de un buen diseñador

Primero que nada… ¿qué vamos a hacer al modelar?

* Encontrar los objetos que pueden resolver los requerimientos que yo quiero
* Darles responsabilidades a cada uno de esos objetos
* Encontrar las relaciones que tienen los objetos

Ok, entonces tan importante como los objetos son sus responsabilidades y sus relaciones. Si un objeto no tiene responsabilidades, no tiene sentido que esté en el sistema. Si un objeto no se comunica con nadie, nadie le envía mensajes, lo que ofrece no le interesa a nadie, tampoco tiene sentido que esté en el sistema.

## 3.2 Encontrando los objetos

Primero, leamos el enunciado una vez más. Detenidamente. Podemos anotar cosas, subrayar, etc. pero lo que más me importa en este momento es entender qué me están pidiendo. Ok, entonces vemos el primer punto que nos piden:

“Saber en qué provincias va a haber actividad de un proyecto para un rango de fechas. De cada oficina se sabe la ciudad, de cada ciudad y zona rural se sabe la provincia.” ¿A qué objeto le podría pedir saber en qué provincias va a haber actividad de un proyecto?

* ¿A la tarea? Mmm… todavía es prematuro pensar en la tarea, de hecho el punto 1) no habla de la tarea.
* ¿A la empresa? ¿Qué empresa? Tampoco aparece la idea de empresa en el punto 1). Claro, seguramente la aplicación es para una empresa, pero recordemos que Objetos me propone un modelo, que **no es la realidad.** En la realidad yo no le pregunto el saldo al cliente, ni la nota al alumno, pero en mi modelo me resulta útil que el alumno sepa si aprobó o no y que el cliente sepa el total de facturación, porque el cliente de mi modelo es distinto al cliente de la realidad.

Pregunta posible del lector: ¿entonces está mal tener una clase Empresa dentro de mi solución? No, no está mal. De hecho en algunos ejercicios necesito que algún objeto conozca a todos mis empleados, bueno, ok, entonces encontrar como abstracción el objeto Empresa no está mal. Lo que no tiene que pasar es que la Empresa tome responsabilidades que no le correspondan (sobre eso vamos a seguir hablando en el apunte).

* Entonces, ¿qué tal un objeto Proyecto?

Sí, suena bien. Al proyecto le pregunto eso. ¿Y cómo se lo pregunto? Con un mensaje. Nótese que hablamos de mensaje y no de método. Esta distinción parece de molestos nomás, pero es importante, indica que primero voy a pensar en cómo voy a usar un objeto y no en cómo lo termina resolviendo. Por supuesto, a los cinco minutos ya estoy pensando en la codificación, pero es importante distinguir que nos paramos como usuarios del proyecto y no como implementadores.

¿Necesitamos pasarle información al proyecto? Sí, el rango de fechas lo vamos a tener que pasar. En un test hacemos:

**const** edificioInteligente = **new** Proyecto()

...

edificioInteligente.provinciasConActividad(unaFecha, otraFecha)

Modelar no admite una única respuesta y tampoco una única visión. Está bueno que a partir de ahora empecemos a usar varias hojas:

* Una que contenga un diagrama de clases. Pero no vamos a encarar todo el diagrama de una, sino que vamos a ir anotando las clases y los métodos a medida que aparecen. De hecho cuando tenemos requerimientos lo suficientemente complejos, el diagrama de clases se separa por requerimiento (a veces llamado *caso de uso*).
* Una hoja tiene que seguir con el REPL, o sea, todavía falta configurar el proyecto (es necesario agregar más líneas todavía)
* Otra hoja va a tener sin dudas el código del método provinciasConActividad()
* Una hoja opcional podría mostrar un diagrama de objetos con un proyecto puntual que creemos con código en un REPL

Cada hoja comunica cosas diferentes de mi solución, y si bien uno tiende a pensar que vale más la codificación del método **todas son útiles y complementarias**.

¿Qué hacemos en el diagrama de clases? Anotamos por ahora sólo un objeto Proyecto con su método correspondiente:

[](http://www.plantuml.com/plantuml/img/Iyv9B2vM22ZAhqnDBibNgEQAZ-SgACeloynBpamiTixFSqmkoIpBJ4bCqKXBJSv8r578BobAT0CnDRdqzFIuQha0)

Fíjense que todavía no definimos qué devuelve el método. Saber diseñar permite dejar algunas puertas abiertas y cerrar otras.

Ahora sí codificamos el método:

>>Proyecto

**method** provinciasConActividad(unaFecha, otraFecha) {

...

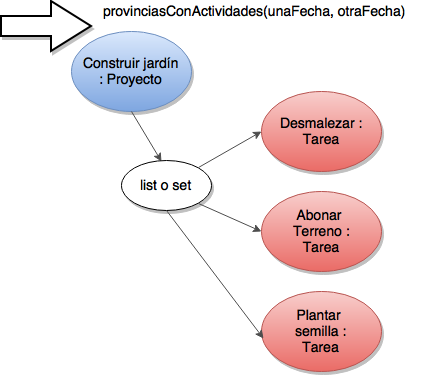
}

¿Un proyecto puede tener muchas provincias en las que se hacen actividades? ¿De qué depende? De cada tarea. Ah, pero además si el tipo de tarea es de oficina lo tengo que calcular...

Ok, ok, es habitual pensar en la implementación, pero si yo estoy codificando la clase Proyecto, sólo me interesan dos cosas:

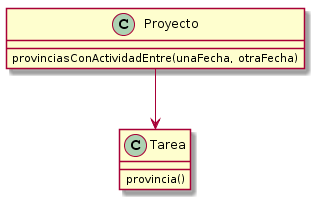
1. Que cada tarea sabe decirme su provincia (no me importa cómo)
2. Cómo me relaciono con mis tareas

Entonces acá surge la necesidad de hacer aparecer al objeto Tarea, y de decir que un Proyecto tiene muchas tareas.

[](https://drive.draw.io/#G0B9t2msYh_ogZSWROWWl5UnR2Nmc)

*Diagrama de Objetos – Consulta de provincias con actividad para un proyecto*

Lo anotamos en el diagrama de clases:

[](http://www.plantuml.com/plantuml/img/Iyv9B2vM22ZAhqnDBibNgEQAZ-SgACeloynBpamiTixFSqmkoIpBJ4bCSSqhAKhLACrBT4jDpaZKKSWlAO8mDRbgkPB1neGa5gKcefcX0PA6cw-hQmTHnGK0)

*Diagrama de clases - versión 2*

Cada tarea ¿conoce al proyecto?

Y... por el momento no lo puedo determinar, porque estoy codificando Proyecto. Lo importante de esto es que

* en este momento no tengo toda la información para responder si cada tarea deba o no conocer al proyecto
* tomar esa decisión ahora me desvía de mi objetivo principal que es resolver el punto 1)

Entonces volvemos a poner el foco en resolver el método. Si cada tarea me dice su provincia, puede que muchas tareas tengan la misma provincia en común. Entonces tengo que definir si quiero devolver una colección de provincias duplicadas o no.

Decido que no. Entonces me conviene devolver una colección que no admita elementos duplicados: un Set va a andar bien. Ahora sólo tenemos que decidir cómo implementar el método. Separemos qué hay que hacer:

* filtrar las tareas entre una fecha y otra
* “recolectar” las provincias de cada tarea y
* eliminar los duplicados de la colección de provincias (transformando la colección en un Set)

Para filtrar las tareas nos damos cuenta de que:

1. La tarea tiene una fecha
2. La tarea me sabe decir si está en un rango de fechas

Claro, uno tiende a pensar en 1) pero está bueno empezar a pensar en 2), porque no sabemos si la implementación considerará la fecha de inicio, la fecha de realización, la fecha real, o que la duración abarque el rango de fechas. Por eso está bueno pensar que desde Proyecto no voy a estar pidiéndole la fecha a la tarea para después comparar si está dentro del rango, sino que voy a delegar esa respuesta a la tarea.

Por otra parte, filtrar las tareas en un rango de fechas parece ser una responsabilidad que estaría bueno tener en un método aparte. Volvemos a las cosas que había que hacer y con la guía de mensajes de Wollok buscamos qué objeto nos puede ayudar a resolver lo que queremos:

|  |  |  |
| --- | --- | --- |
| Tarea a hacer | Qué objeto es responsable | Qué mensaje le mando |
| filtramos las tareas entre una fecha y otra | filter -> una colección | filter(objetoBloque) |
| “recolectar” las provincias de cada tarea | recolectar una colección | map(objetoBloque) |
| eliminamos los duplicados de la colección de provincias | transformar una colección en  Set --> una colección | asSet() |

Ahora sí codificamos el método en cuestión:

**class** Proyecto {

**const** tareas = []

**method** provinciasConActividad(unaFecha, otraFecha) =

**self**.tareasEntre(unaFecha, otraFecha)

.map({ tarea => tarea.provincia() })

**method** tareasEntre(unaFecha, otraFecha) =

tareas.filter({ tarea => tarea.fechaEntre(unaFecha, otraFecha) })

}

# 4 Objeto Tarea

Cada tarea tiene que poder determinar si está en un rango de fechas. Todas las tareas tienen una fecha de comienzo y se asume que duran un día. Veamos la guía de objetos básicos,especialmente las fechas. Ah, hay un mensaje between() que parece calzar justo.

Codificamos entonces el método:

**class** Tarea {

**var** fecha

**method** fechaEntre(unaFecha, otraFecha) =

fecha.between(unaFecha, otraFecha)

}

Por otra parte, tenemos que resolver la provincia. Acá sí hay que discriminar el lugar donde se realizan las tareas:

* Para las tareas que se hacen en una oficina, la dirección de la oficina contiene la ciudad y la ciudad determina la provincia.
* Para las tareas que se hacen en una ciudad, la ciudad determina la provincia.
* Para las tareas que se hacen en zona rural, la zona rural determina la provincia.

# 5 Alternativas al If

Una práctica común cuando no trabajamos en objetos es que la tarea tenga un “tipo de tarea” o “lugar” codificado con algún estilo como el que sigue:

|  |  |
| --- | --- |
| Tipo de Tarea | Qué significa |
| “O” | Se hace en una oficina |
| “C” | Se hace en una ciudad |
| “R” | Se hace en una zona rural |

A veces incluso se eligen códigos menos representativos (15, 21, 28, 74, etc.)

La idea funciona, pero tiene inconvenientes que hay que contemplar cuando estamos pensando esta solución:

* “Tipo de tarea = C”, “Tipo de tarea = 28” implica un código que hay que traducir, tanto para el programador como para el usuario. Esto tiene un costo, el de no darle un nombre representativo para esa abstracción que yo acabo de encontrar (“Tarea que se hace en una ciudad”)
* El código que devuelve la provincia tiene este formato:

>>Tarea

**method** provincia() {

**if** (tipoDeTarea.equals("O")) {

// Oficina

}

**if** (tipoDeTarea.equals("C")) {

//Ciudad

}

**if** (tipoDeTarea.equals("R")) {

//Zona Rural

}

}

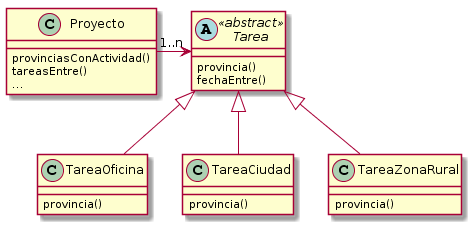
Lo más molesto de este código es si en más de un lugar la respuesta depende del tipo de tarea. Veamos el requerimiento del punto 2):

“Saber la superficie promedio en la que se desarrollan las tareas de un proyecto.

Para las oficinas se establece una superficie, la misma para todas en m2. Para las ciudades se informa explícitamente la superficie en m2. Las zonas rurales se asumen como rectangulares; se informa ancho y largo en metros.”

Nuevamente tengo que armar un if múltiple para resolver el método y son dos lugares donde estoy repitiendo la misma lógica de separación. Si se agrega un nuevo tipo de tarea, o se modifica la codificación (usamos enteros en lugar de Strings, o cambiamos los códigos) son dos lugares donde tengo que acordarme de ir a modificar.

Entonces tenemos que buscar una alternativa. ¿Qué tal definir tres subclases de Tarea? Actualizamos el diagrama de clases:

[](http://www.plantuml.com/plantuml/img/bOun2y9038Nt_8gZamuUk9U256T5d5nAJAyOA3cvIuiY_U-UIgNe88ulvFlUenPZDDlWhuxKcnjK5Po3xrc8CPPULgJSSutrR0wAmM5SYmQNahKM1i1Zr82aXfwc_UF5vFbuBeeFPm8RHoSSBGEyonU5oPRMIlQomCIuQPXOqBYBEgd7cY_pC6LAxjBilv235znr0TkVr1q0)

*Diagrama de clases - Versión 3*

El método fechaEntre() no se modifica, aplica a todas las tareas. Resolvemos el método provincia en cada una de las subclases de Tarea.

En la tarea de oficina necesitamos saber a qué provincia pertenece. ¿Cómo determinamos esto? La tarea de oficina necesita un objeto que represente a la oficina. No puede ser un String, porque a un String no le puedo preguntar lo que necesito.

# 6 Programar es humano, delegar es divino

Entonces tengo varias opciones:

1) Pienso que la oficina tiene una ciudad y que una ciudad tiene una provincia, entonces codifico el método así:

**class** TareaOficina **inherits** Tarea {

**const** oficina

**constructor**(\_oficina) {

oficina = \_oficina

}

**method** provincia() = oficina.ciudad().provincia()

}

2) Pienso que la oficina sabe decirme a qué provincia pertenece, no sé cómo lo resuelve ni me interesa.

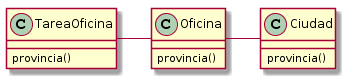
>>TareaOficina

**method** provincia() = oficina.provincia()

Si ya le agarraron la mano al apunte, sí preferimos usar 2) porque la lógica de una oficina para obtener la provincia puede variar y yo no quiero verme afectado.

¿Dónde conozco más? ¿En la opción 1 o en la 2? Y... en la 1). Conocer más implica que cualquier cambio en la implementación de Oficina afecta directamente a TareaOficina[[1]](#footnote-0), y eso es malo en tanto y en cuanto me quedo después de las 18:00 para arreglarlo.

Actualizamos el diagrama de clases:

[](http://www.plantuml.com/plantuml/img/Iyv9B2vMSCuiJKbCKQZcYe_dAYZABylCIyvCrD3agkLA1ilxfsKcP-ObubOGabYKceX75OgyBimyBeG8n0K0)

*Diagrama de clases - versión 4*

# 7 Responsabilidad y delegación

¿Qué es ser responsable? Hacerse cargo de una tarea, asegurarse de que se haga

correctamente. Cada objeto tiene una misión dentro del sistema, y es responsable de una serie de cosas.

Parece una tontería, pero durante todo el ejercicio aplicamos la misma idea:

¿quién sabe las provincias en las que hay actividad? El proyecto

¿quién sabe si una tarea coincide en un rango de fechas? La tarea

¿quién sabe la provincia en la que está una oficina? La oficina.

La **delegación** nos permite:

* Que cada objeto haga sólo lo que le corresponde. Lo que no puede hacer lo debe delegar al objeto que tenga esa responsabilidad.
* Conocer sólo lo que el objeto tiene que conocer (baja así el impacto cuando hay modificaciones)
* No escribir dos veces lo mismo

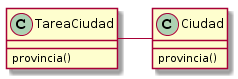
Codificamos los métodos que faltan:

>>TareaCiudad

**method** provincia() = ciudad.provincia()

La provincia: ¿tiene que ser un objeto o puede ser un String? Lo que tengo que preguntarme para estar seguro es: ¿necesito que la provincia tenga comportamiento o solo es el valor lo que me importa? Entonces por el momento lo dejamos como un String.

El diagrama de clases queda:

[](http://www.plantuml.com/plantuml/img/Iyv9B2vM24aiIarqpYnDIKnHgEQAZ-SgACeloynBpapKqEIgvKe6Ay8jZwnT5weE0000)

La tarea que se hace en zona rural vuelve a delegar en la zona la obtención de la provincia:

**class** TareaZonaRural **inherits** Tarea {

**const** zonaRural

**constructor**(\_zonaRural) {

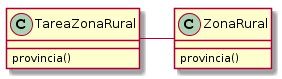
zonaRural = \_zonaRural

}

**method** provincia() = zonaRural.provincia()

}

Y el diagrama de clases queda:

[](http://www.plantuml.com/plantuml/img/Iyv9B2vM24aiIasCoizB32ejIinHgEQAZ-SgACeloynBpapKqEIgvKe6gyEh1CqGNOHg0000)

# 8 Superficie promedio de las tareas

Pasemos al punto 2): necesitamos saber la superficie promedio en la que se desarrollan las tareas de un proyecto. ¿Qué objeto es responsable de esto? Claramente, el proyecto.

Anotamos en la hoja correspondiente al REPL el mensaje que le enviaríamos:

**const** edificioInteligente = **new** Proyecto()

edificioInteligente.provinciasConActividadEntre(unaFecha, otraFecha)

edificioInteligente.superficiePromedio()

Para obtener la superficie promedio, deberíamos saber:

* La superficie total de cada una de las tareas
* La cantidad de tareas

También esas son responsabilidades del proyecto, vamos a ponerle nombre codificando el método:

// Punto 2

**method** superficiePromedio() =

**self**.superficieTotal() / **self**.cantidadDeTareas()

Anotamos lo que hay que resolver y qué mensajes envío a qué objetos en base a la guía de objetos básicos:

|  |  |  |
| --- | --- | --- |
| **Tarea a hacer** | **Qué objeto es responsable** | **Qué mensaje le mando** |
| sumatoria de superficies de cada tarea | totalizar → una colección | fold |
| saber la cantidad de tareas | cantidad de elementos → una colección | size |

Entonces surge como responsabilidad de cada tarea decirme su superficie. Al proyecto no le interesa cómo lo resuelve, sino que esa responsabilidad se la puede preguntar a cada objeto tarea:

>>Proyecto

**method** superficieTotal() =

tareas.fold(0, { acum, tarea => acum + tarea.superficie() })

**method** cantidadDeTareas() = tareas.size()

## 8.1 Superficie de una tarea

¿Cómo sabemos la superficie de una tarea?

* Para las oficinas se establece una superficie, la misma para todas en m2
* Para las ciudades se informa explícitamente la superficie en m2
* Las zonas rurales se asumen como rectangulares; se informa ancho y largo en metros.

Ok, ¿entonces las tareas de oficina en quién van a delegar la obtención de la superficie? En la oficina, claro. Si todas las oficinas comparten la misma superficie, entonces no depende de un objeto Oficina en particular. La información sobre la superficie la devolvemos en un método que devuelve una constante (10, por ejemplo), en la clase Oficina.

>>TareaOficina

// Punto 2

**method** superficie() = oficina.superficie()

>>Oficina

**class** Oficina {

**method** superficie() = 10

}

Seguimos con las tareas que se desarrollan en una ciudad:

>>TareaCiudad

// Punto 2

**method** superficie() = ciudad.superficie()

>>Ciudad

// Punto 2

**method** superficie() = superficie

Si se informan explícitamente, bueno, es eso: la ciudad conoce su superficie.

Y por último las tareas que se desarrollan en zonas rurales:

>>TareaZonaRural

**method** superficie() = zonaRural.superficie()

>>ZonaRural

**method** superficie() = ancho \* largo

# 9 Saldo de un proyecto

Ahora queremos resolver el punto 3: “Saber el saldo de un proyecto a una fecha, que debe tener en cuenta todas las tareas hasta esa fecha inclusive.”

Volvemos a la hoja del REPL y tratamos de darle un nombre a ese requerimiento:

**const** edificioInteligente = **new** Proyecto()

...

edificioInteligente.provinciasConActividadEntre(unaFecha, otraFecha)

edificioInteligente.superficiePromedio()

edificioInteligente.saldoA(unaFecha) ← saldo de un proyecto se lo pregunto... al proyecto

Ok, empezamos a codificar el método saldoA().

>>Proyecto

// Punto 3

**method** saldoA(unaFecha) = presupuestoInicial - ...

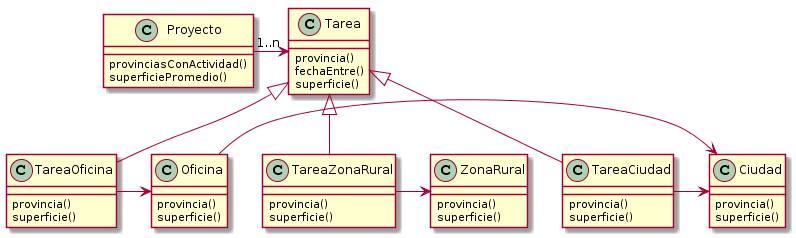
Vuelvo a leer el enunciado:

“Es importante tener en cuenta el saldo de un proyecto, que permite que se puedan llevar a cabo sus tareas. Cada proyecto nace con un presupuesto inicial; luego el saldo va variando en función de las tareas que incluye.

Hay: tareas de producción, tareas de recaudación, y reuniones.”

Mmm… hay algo que no cierra de lo que hicimos antes.

Repasemos el diagrama de clases:



*Diagrama de clases - versión 7*

Ya habíamos subclasificado a las tareas según se hicieran en zona rural, ciudad y oficina y ahora resulta que tenemos tareas de producción, de recaudación y reuniones. Hay dos criterios de subclasificación de la tarea ¡en el mismo enunciado!

## 9.1 Si hay que cambiar, se cambia

Ok, entonces tengo que decidir con cuál [taxonomía](https://es.wikipedia.org/wiki/Taxonom%C3%ADa) me quedo. Por un lado puedo subclasificar por el lugar donde se desarrolla una tarea, por otro lado puedo subclasificar en base a si la tarea representa ingreso/gasto en el presupuesto.

Revisemos el código de TareaOficina, TareaCiudad y TareaZonaRural:

|  |  |  |
| --- | --- | --- |
| >>TareaOficina    // Punto 1  **method** provincia() =  oficina.provincia()    // Punto 2  **method** superficie() =  oficina.superficie() | >>TareaCiudad  // Punto 1  **method** provincia() =  ciudad.provincia()    // Punto 2  **method** superficie() =  ciudad.superficie() | >>TareaZonaRural  // Punto 1  **method** provincia() =  zonaRural.provincia()  // Punto 2  **method** superficie() =  zonaRural.superficie() |
| Delega en oficina | Delega en ciudad | Delega en zona rural |

¿Se ve que tenemos código duplicado?

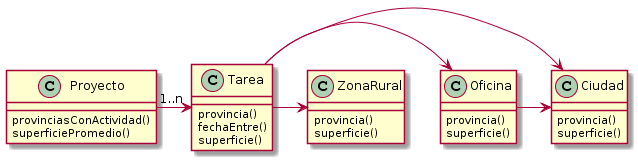
En cada caso estamos siempre delegando los mensajes provincia y superficie a:

* una Oficina
* una Ciudad
* una Zona Rural

Como los tres objetos son polimórficos para la tarea, vamos a hacer un cambio: eliminamos las tres subclases actuales de Tarea y vamos a hacer que la tarea tenga un lugar, que puede ser:

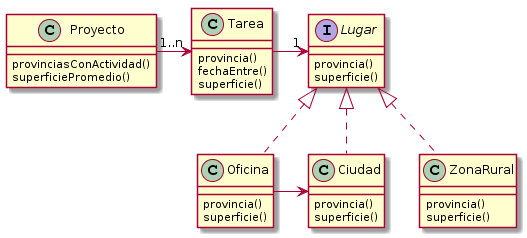
* una Oficina
* una Ciudad
* una Zona Rural

Cambiamos entonces el diagrama de clases:

[](http://www.plantuml.com/plantuml/img/Iyv9B2vM22ZAhqnDBibNgEOgACeloynBpamiTixFSqmkoIpBJ4bCqT3aAYujI2rAoqpEJ0MgparDoSm72jPo9OCD24aiIar4rWsKIqjDpaXqpIifIaNH3TO5rUOF4ib3ruXThNDcAT0XH2cDoizB32ejIimXe1hYQbqxc3iG0X3B47ouYLmm9oCKmGCFAAHagAUNfmG93W00)

La tarea no va a tener 3 referencias, sino un atributo lugar, que apuntará a una zona rural, una oficina o una ciudad, dependiendo del lugar donde se realice la tarea.

El diagrama anterior no respeta fielmente las reglas de los diagramas de clase UML. Si fuéramos más estrictos, definiríamos una interfaz Lugar:

[](http://www.plantuml.com/plantuml/img/dP1D2i9038NtSugXArqeU051YZj14LVkGffgeCsKp4n1nBixTb2qkxhCo_loyx01x-r1tOqmE7itdRgU1HbywMI3WNkke9hDZOyTQSt8bDmjLUoI-30u33Y14dpJgLSJNc4hGUc77YZ34b8DI7ONBw0ZTEpEIyvEu1WL6ijjrr1B4lm4lEIOFlcFtRyKWQbmZcMnjiMgo81vJqfYFiXymXzyowKKESud)

¿Qué representa Lugar? Una interfaz, es decir, **un tipo**.

Las clases Oficina, Ciudad y Zona Rural definen sus propios métodos provincia() y superficie(), pero los tres tienen algo en común: definen un método provincia() y otro superficie().

En Wollok no estamos obligados a definir un tipo a nuestras variables, entonces nos alcanza con escribir

**class** Tarea {

**var** fecha

**var** lugar

Después nosotros sabemos que esa referencia podrá apuntar a una oficina, ciudad o zona rural.

Al desaparecer sus subclases, Tarea debe implementar los métodos provincia() y superficie(), delegando esa responsabilidad al lugar:

// Punto 1

**method** provincia() = oficina.provincia()

// Punto 2

**method** superficie() = oficina.superficie()

Está bueno ver que eliminar las tres subclases no trajo aparejado grandes cambios en la solución. En todo caso lo que hicimos fue preparar el terreno para poder aceptar un nuevo requerimiento (el punto 3) porque de otra manera se nos hubieran complicado las cosas[[2]](#footnote-1).

## 9.2 Sobre errores y volantazos

El lector se preguntará: estoy leyendo este apunte tranquilo y todo parece fluir, ahora bien ¿qué pasa si en un examen noto que la solución parece no funcionar del todo bien? ¿cómo dar el “volantazo” sin sentir miedo?

La respuesta es simple: ¿somos perfectos? ¿sabemos todo? ¿o simplemente vamos construyendo nuestro camino y resolviendo las dificultades cuando se presentan? La realidad es que si asumimos que como seres humanos somos imperfectos, que estamos aprendiendo, y que por lo general lo que hay que resolver es algo nuevo para nosotros, lo más común es cometer errores. Entonces **adoptemos una metodología que tolere el error**, donde el error sea parte esperable de nuestro método de resolución de problemas.

Si ante el primer síntoma negamos la posibilidad de corregir lo que hicimos porque “ya arranqué y no se puede cambiar, no me va a dar el tiempo” estamos atando a la primera solución que se nos ocurrió, que suele ocurrir en el momento más temprano (y por ende cuando menos idea tenemos de cómo resolver el ejercicio). Entonces, el miedo y la resistencia al cambio son naturales, pero el costo de no evaluar si vamos bien es mayor que el beneficio de no quedar empantanados para calzar la solución “a como de lugar”. Para eso, en este apunte hemos ido despacio: no hicimos todo el diagrama de clases al comienzo, solo escribimos el código de los puntos que me fueron pidiendo. De esa manera, podemos ir para atrás en nuestro camino sin tener que volver al punto de partida: si voy a fallar, más vale darme cuenta lo más temprano posible ([fail fast](http://www.martinfowler.com/ieeeSoftware/failFast.pdf)).

# 10 Saldo de un proyecto (2° parte)

Ahora sí nos podemos concentrar en el saldo de un proyecto:

>>Proyecto

// Punto 3

**method** saldoA(unaFecha) =

presupuestoInicial - **self**.montoDeTareasA(unaFecha)

**method** montoDeTareasA(unaFecha) =

...

Tenemos que filtrar las tareas hasta una fecha determinada. ¿Cómo hacemos? Podemos aprovechar el método tareasEntre() del punto 1...

>>Proyecto

**method** montoDeTareasA(unaFecha) =

**self**.tareasEntre(¿?, unaFecha)...

Cómo resuelvo el ¿? . Tenemos aquí dos opciones:

1. El proyecto sabe su fecha de inicio
2. El proyecto busca la fecha de la primera tarea que se hizo

En cualquiera de los dos casos podemos delegar esa responsabilidad a otro método de la misma clase:

>>Proyecto

**method** montoDeTareasA(unaFecha) =

**self**.tareasEntre(**self**.fechaInicio(), unaFecha)...

**method** fechaInicio() =

¿Qué ventaja tiene esto?

Podemos resolverlo de cualquiera de las dos maneras que planteamos arriba y no necesitamos cambiar la lógica del método montoDeTareasA(). En Proyecto...

|  |  |
| --- | --- |
| **var** fechaInicio  ...  **method** fechaInicio() =  fechaInicio | **method** fechaInicio() =  tareas.sortBy({ t1, t2 =>  t1.fecha() < t2.fecha()  }).first() |
| opción 1: getter | opción 2: se calcula en base a las tareas |

También podríamos crear un método que filtra las tareas hasta una determinada fecha:

>>Proyecto

**method** montoDeTareasA(unaFecha) =

**self**.tareasHasta(unaFecha)...

**method** tareasHasta(unaFecha) =

**self**.tareasEntre(**self**.fechaInicio(), unaFecha)

La ventaja de tener este método aparte es la posibilidad de usarlo en otros contextos.

Un proyecto tiene tareas: algunas representan ingresos, otros gastos y otros no son ninguna de las dos cosas. Cada tarea tiene asociado un monto, que puede ser positivo para los ingresos, negativo para los gastos y cero para los que no son ingresos ni gastos. Lo que el proyecto tiene que hacer es totalizar el monto de las tareas (ya vimos cuál es el mensaje que usamos para totalizar):

>>Proyecto

// Punto 3

**method** saldoA(unaFecha) =

presupuestoInicial - **self**.montoDeTareasA(unaFecha)

**method** montoDeTareasA(unaFecha) =

**self**.tareasHasta(unaFecha)

.fold(0, {acum, tarea => acum + tarea.monto()})

¿Cómo se calcula el monto de cada tarea? Veamos el enunciado:

* Las tareas de producción implican un gasto, que es la suma del costo de los servicios que se usan, que se indica para cada tarea. El costo de cada servicio se indica explícitamente, y es independiente de la tarea: p.ej. si digo que el servicio de desinfección cuesta 45 pesos, ese valor es el mismo en el que se incluya el servicio de desinfección.
* Las tareas de recaudación implican un ingreso, que se indica explícitamente.
* Las reuniones no implican ni gasto ni ingreso.

Ok, entonces hay comportamiento diferencial para las tareas de producción/tareas de recaudación/reuniones. ¿Dónde está ese “comportamiento diferencial”? En el método que calcula el monto:

En las tareas de producción, ¿cómo voy a representar cada servicio?

1. Es un objeto
2. Es un String.

Como quiero pedirle su costo, el servicio es un objeto.

Tomemos el ejemplo que da el enunciado para modelar un diagrama de objetos: queremos graficar que si hay dos tareas de producción que contienen el servicio de desinfección, ese objeto es el mismo dentro de mi ambiente:

Tarea Desinfeccion - Diagrama de Objetos.png

Entonces sumarizo el costo de cada servicio, que resta al saldo del proyecto.

**class** TareaProduccion **inherits** Tarea {

**const** servicios = []

**method** agregarServicio(\_servicio) {

servicios.add(\_servicio)

}

**method** monto() = servicios.fold(0, { total, servicio => total + servicio.costo() })

}

**class** Servicio {

**const** costo

**constructor**(\_costo) {

costo = \_costo

}

**method** costo() = costo

}

El lector puede preguntarse: ¿no deberíamos conocer el nombre del servicio? Probablemente, pero todavía no surgió que sea necesario... entonces por el momento lo dejamos porque nos distrae de continuar con lo que sí nos pidieron.

En el caso de las tareas de recaudación necesitamos un atributo ingreso, que suma al saldo del proyecto (entonces tiene que ser un monto negativo):

**class** TareaRecaudacion **inherits** Tarea {

**var** ingreso

**constructor**(\_ingreso) {

ingreso = \_ingreso

}

**method** monto() = ingreso \* (-1)

}

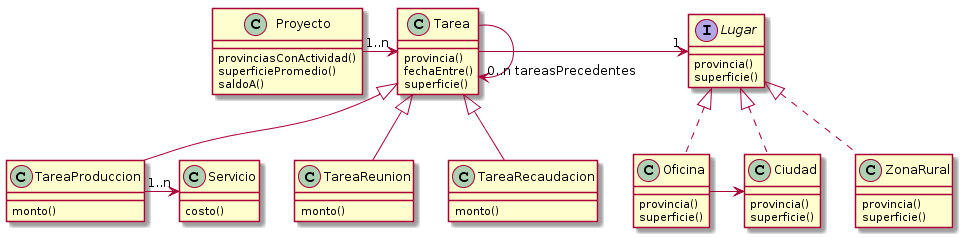
Y por último las reuniones, que no suman ni restan:

**class** TareaReunion **inherits** Tarea {

**method** monto() = 0

}

Actualizamos ahora el diagrama de clases de nuestra solución:

[](http://www.plantuml.com/plantuml/img/dLBBJWCn3BpdAw8zmO6g_01ILN53eYgSk5cE2vPsxLKUAn3Yt_4sb6wNIyij6S-CnfDW2odvJTGFmgp-q_LH1nPaI6kL5MOUE42ukdQfz1HtZ4p6xYYmZY2qGLTs-7AuTtg6I32riTcEy0tk9KSwiPchjbI4LJozPv9mjEfKilub8vG0UBR0GeU2P_6VA0uMS6m3DTMHTopPWWEIVoYl46SxpjRwSNfHWMs9q7hk-fOwafmkaA_PLWp_qpwEYC2bujh9pPrlRfigS0Sd0siWz_jTzhp5GfhQvKIzDDJdyPesaP229Q1qmfkyoD7dqFqt)

*Diagrama de clases - versión 9*

# 11 Sabiendo si se puede hacer una tarea

Bien, para este requerimiento, ¿a qué objeto tengo que enviarle un mensaje?

A una tarea.

En la hoja del REPL tenemos que crear un objeto Tarea:

**const** instalacionElectrica = **new** TareaProduccion()

...

instalacionElectrica.sePuedeHacer()

Aquí vemos que todas las tareas definen si se pueden hacer de una manera predeterminada, pero las tareas de producción van a redefinir ese comportamiento.

“la condición es que todas las tareas de las que depende deben tener una fecha anterior. Para las tareas de producción, además, el saldo del proyecto a la fecha en que se hace la tarea debe ser no negativo (o sea, >= 0)”

Ya sabemos que cada tarea tiene una fecha en la que se ejecuta. Ahora necesitamos agregarle las tareas que la preceden (las tareas de las que depende). Una tarea t se puede hacer si todas las tareas se pueden cumplir antes de la fecha de dicha tarea t.

* **Opción 1:** preguntamos la fecha a cada tarea precedente y verificamos si todas las tareas son anteriores a nuestra fecha, ok. En caso contrario, no se puede hacer.
* **Opción 2:** preguntamos a cada tarea precedente si se puede cumplir antes de mi fecha (mi fecha = fecha de la tarea donde estoy parado).

Sí, sí, opción 2 es la que vamos a preferir.

Buscamos en la guía de lenguajes qué mensaje nos sirve para verificar que todos los elementos de una colección cumplen una determinada condición: all(Closure), entonces codificamos:

**class** Tarea {

...

**var** tareasPrecedentes = []

// Punto 4

**method** sePuedeHacer() =

tareasPrecedentes

.all({ tarea => tarea.sePuedeCumplirAntesDe(fecha) })

}

Nótese aquí que la delegación es sobre la misma clase donde estoy escribiendo el código, pero el efecto es el mismo: yo puedo enviar el mensaje sePuedeCumplirAntesDe(fecha) en cualquier otro contexto (desde el proyecto, por ejemplo) y me sirve.

>>Tarea

**method** sePuedeCumplirAntesDe(unaFecha) = fecha < unaFecha

Según el enunciado todas las tareas tardan un día en completarse, entonces si mi fecha de ejecución es menor que la fecha que recibo como parámetro, puedo asegurar que la tarea se puede cumplir antes de esa fecha.

Nos falta redefinir el comportamiento para las tareas de producción. Oops, el saldo del proyecto tiene que ser positivo. Para esto, necesito conocer al proyecto. ¿Cómo hago?

* La tarea conoce al proyecto
* Para saber si se puede cumplir una tarea necesito que me pasen como parámetro el proyecto
* ¿?

No hay magia: si necesito conocer el saldo de un proyecto y eso se obtiene enviando el mensaje saldo a un objeto proyecto, tengo que conocer a un objeto proyecto sí o sí. Simplemente por una cuestión didáctica abordaremos ambas soluciones posibles: si la tarea conoce al proyecto, entonces la tarea de producción se puede hacer si cumple la restricción por defecto (que está implementada en la superclase) y además el proyecto tiene saldo positivo a la fecha en que se ejecuta la tarea:

>>TareaProduccion

// Punto 4

**override** **method** sePuedeHacer() =

tareasPrecedentes

.all({ tarea => tarea.sePuedeCumplirAntesDe(fecha) })

&& (proyecto.saldoA(fecha) > 0)

Mmm... nos queda código duplicado. Esto se puede mejorar, con super, claro:

**override** **method** sePuedeHacer() =

**super**() && (proyecto.saldoA(fecha) > 0)

La otra opción es agregar como parámetro al proyecto al enviar el mensaje sePuedeHacer.

Modificamos el selector (el nombre del método) en Tarea y en TareaProduccion:

>>Tarea

// Punto 4

**method** sePuedeHacerPara(\_unProyecto) =

tareasPrecedentes

.all({ tarea => tarea.sePuedeCumplirAntesDe(fecha) })

>>TareaProduccion

// Punto 4

**override** **method** sePuedeHacerPara(\_unProyecto) =

**super**(\_unProyecto) && (\_unProyecto.saldoA(fecha) > 0)

Eh... ¿y no está mal que en Tarea reciba un proyecto que después no uso?

Bueno, el sentido que tiene recibir el proyecto en Tarea es conservar el polimorfismo para saber si una tarea se puede hacer. No importa tanto recibir un objeto proyecto y no usarlo como perder el polimorfismo para quienes usan a las tareas.

Lo volvemos a decir:

Si yo quiero hablar con todas las tareas igual, necesito que la interfaz (el nombre del selector y la cantidad de parámetros) se respete en cada una de las tareas. Si las tareas tienen un método sePuedeHacer(), y la TareaProduccion un sePuedeHacerPara(\_unProyecto), entonces antes de llamar tengo que preguntar si es una tarea de producción o de las otras. Claramente eso no es lo que quiero, porque tener dos métodos distintos es un dolor de cabeza. ¿Para quién es un dolor de cabeza? Para el que usa a las tareas...

|  |
| --- |
| El polimorfismo me ayuda a no conocer cosas que no necesito conocer. |

# 12 Proyectos y tareas en el ambiente

Si elegimos que la tarea conozca al proyecto, volvamos al diagrama de objetos:

Proyecto - Tarea.png

*Diagrama de Objetos – Consulta Relación Proyecto/Tarea*

Vemos que **no hay objetos duplicados en el ambiente**, solamente aumenta el número de referencias. Esto es algo totalmente válido en objetos, sólo debemos asegurar la consistencia cada vez que se agregue/elimine una tarea a un proyecto:

Si en el REPL hacemos:

edificioInteligente agregarTarea(instalacionElectrica)

El método agregarTarea: debe encargarse de mantener ambas referencias:

**class** Proyecto {

**const** tareas = [ ]

**method** agregarTarea(\_tarea) {

tareas.add(\_tarea)

\_tarea.proyecto(**self**)

}

...

**class** Tarea {

...

**var** proyecto

**method** proyecto(\_unProyecto) {

proyecto = \_unProyecto

}

Lo mismo al eliminar la tarea.

# 13 Proyecto coherente

Un proyecto es coherente si pueden hacerse todas las tareas en la fecha indicada. ¿A quién le pregunto esto? Por supuesto, al proyecto. En la hoja del REPL escribimos:

edificioInteligente.esCoherente()

Entonces pasemos al código de Proyecto: ¿qué me interesa? Que todas las tareas se puedan hacer en la fecha indicada. Ya sabemos que eso se traduce en un mensaje a cada tarea específica y lo que me importa no es cómo lo hace, sino que retorna un booleano si se puede hacer en la fecha prevista o no. Si todas las tareas cumplen eso, entonces el proyecto es coherente. Ya vimos el mensaje all(Closure), juntamos todo eso y codificamos:

>>Proyecto

**method** esCoherente() =

tareas.all({ tarea => tarea.sePuedeHacerPara() })

Y aquí vemos que el punto 4 nos sirvió en bandeja este punto.

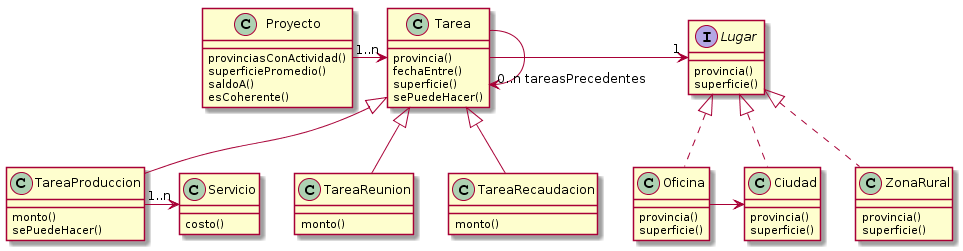
Sólo estaría bueno agregar que si decidimos que el mensaje sea sePuedeHacerPara(\_unProyecto), el mismo proyecto debe pasarse como parámetro:

>>Proyecto

**method** esCoherente() =

tareas.all({ tarea => tarea.sePuedeHacerPara(**self**) })

La delegación facilitó enormemente la resolución de este punto. Actualizamos el diagrama de clases:

[](http://www.plantuml.com/plantuml/img/dPB1IWCn48RlUOgm9prOz0M4KWGFGfVgoTimcRM1tSmoIHP5-kvEkjRkLXJhBJFv_v-PBy4MOhIry1jXOljkUk711_GGLnoMcFpW7RYBInDpJz9uzAJgZfpdqeJMyL8FfF8j2OL4MkqCxdEVG0YceNhN46xXBYIXMMWfgCxaw1wGv3HaGpbu3fPU4mLtJEuu9FukHiWEyCy6tSXb_5N_mt2F98EENyWXnp7P6gyOf56fVSWl82S4Pallp6VICmVOP87M-gvlgLEKyGpxokl6xd_UTUa4EDSy8hgwjTLDDHhC8KcRuq3cwsljTOj5g4Qq4_UrTcqgPQo5K2ahlZZJJHxec7DW_m40)

Claro, si no escribimos todas las variables, el diagrama tiene una cantidad acotada de clases que me permite hacer un paneo rápido de mi solución. Pero ojo:

* Si digo muy poco, el diagrama comunica poco.
* Si digo demasiado, el diagrama no me sirve porque cada vez que lo miro tengo que separar lo importante de lo que no lo es. Por eso cuando tenga demasiados requerimientos, conviene hacer un diagrama de clases por requerimiento. Eso va a permitir comunicar con un foco claro y definido.

# 14 Margen anterior de una tarea

Y nos despedimos con el último requerimiento: “saber el margen anterior de una tarea, que es la cantidad de días entre que se hace la última de las tareas de las que depende, y el día indicado para esa tarea. Si la tarea no depende de ninguna, su margen anterior es 0.”

Nuevamente empezamos pensando quién es el objeto receptor: ya le vamos agarrando la mano al proceso...

Le envío un mensaje a un objeto Tarea:

instalacionElectrica.margenAnterior()

Y sabemos que esperamos un número.

Para calcular el margen anterior de una tarea, necesitamos:

* Encontrar la última tarea de las que depende
* Sacar la diferencia entre la fecha de la tarea y la de la última tarea encontrada

Revisamos la guía de mensajes: ¿cómo hacemos para encontrar la última tarea de la que depende? Bueno, tenemos que ordenar la lista de tareas precedentes por fecha.

Esa responsabilidad puede estar en el método margenAnterior, o mejor, puede estar afuera de ese método:

>>Tarea

// Punto 6

**method** margenAnterior() =

tareasPrecedentes

.sort({ t1, t2 => t1.fecha() > t2.fecha() })...

*Opción 1: Ordeno las tareas precedentes por fecha dentro del método margen anterior, o bien marcamos* tareasPrecedentes.sort({ t1, t2 => t1.fecha() > t2.fecha() }) *y mediante botón derecho Refactor > Extract Method creamos el método tareasPrecedentesPorFecha*

>>Tarea

// Punto 6

**method** margenAnterior() =

**self**.tareasPrecedentesPorFecha()...

**method** tareasPrecedentesPorFecha() =

tareasPrecedentes

.sort({ t1, t2 => t1.fecha() > t2.fecha() })

*Opción 2: Ordeno las tareas precedentes por fecha en un método aparte*

Aquí gané la posibilidad de usar el método tareasPrecedentesPorFecha en otro contexto. Además el método margenAnterior hace menos cosas (su objetivo es bien claro, calcula el margen anterior y no se preocupa por ordenar las tareas precedentes por fecha, porque lo delega en otro método: si el objetivo de un método es claro y coherente decimos que aumenta su cohesión.

Una vez que tengo ordenada las tareas por fecha (de mayor a menor) puedo obtener la tarea más reciente. Pero ¿qué pasa si no tiene tareas precedentes? Entonces tengo que saber si una tarea tiene tareas precedentes.

>>Tarea

**method** margenAnterior() {

**if** (!**self**.tieneTareasPrecedentes()) {

**return** 0

}

**return** **self**.tareasPrecedentesPorFecha().first() // falta

}

**method** tieneTareasPrecedentes() =

!tareasPrecedentes.isEmpty()

**method** tareasPrecedentesPorFecha() =

tareasPrecedentes

.sort({ t1 , t2 => t1.fecha() > t2.fecha() })

Y lo que faltaría es ver cómo calculo la diferencia entre dos fechas. Vemos la guía de mensajes de Date, y con eso tenemos cómo avanzar.

>>Tarea

// Punto 6

**method** margenAnterior() {

**if** (!**self**.tieneTareasPrecedentes()) {

**return** 0

}

**const** ultimaTarea = **self**.tareasPrecedentesPorFecha().first()

**return** fecha - ultimaTarea.fecha()

}

# 5 Qué puede cambiar si estoy haciendo el TP (o en el trabajo)

El proceso de resolución en un TP cambia en algo importante: tengo a la máquina a mi favor. ¿Y en qué me puede ayudar tener la máquina? A pensar con ejemplos concretos cómo funciona mi aplicación.

Y para eso no solo podemos pensar en un REPL, sino que contamos con una herramienta muy poderosa: los tests unitarios.

Para poder tener buenos tests unitarios, tenemos que recordar los consejos del Anexo de Testing: tener un buen juego de datos: la casuística o casos representativos. No tiene sentido crear 10 tareas de producción, pero sí está bueno probar una tarea de producción con 1 servicio y otro con varios servicios. Y un proyecto con poco presupuesto inicial, otro con un gran margen entre las tareas, un proyecto coherente y otro que no lo es...

A la hora de crear el juego de datos:

* está bueno generar constructores piolas para mejorar la expresividad y reducir la cantidad de líneas de código, o dicho de otro modo: **vale diseñar en el juego de datos y al construir los tests**.
* Darle a los objetos nombres representativos también ayuda a entender ese juego de datos (en lugar de “producto” ponerle “tornillo”, en lugar de “cliente1” ponerle “bardaroHnos”, algo que sea representativo para el negocio).

El juego de datos es mi input para generar los casos de prueba: cuando se cumplen

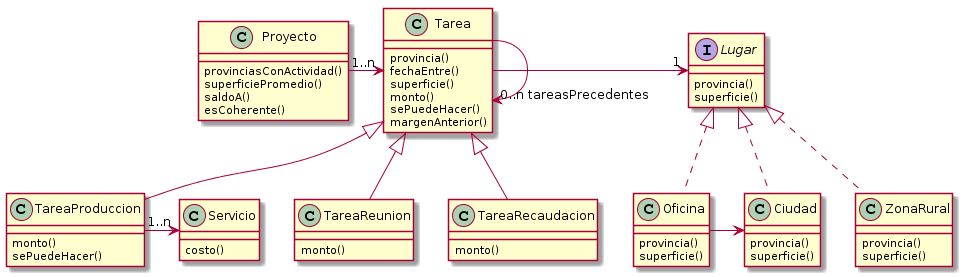
determinadas condiciones el sistema debe funcionar de una determinada manera. Pensar para cada requerimiento cuáles son los casos límite, los casos con error, y cómo esperamos que se comporte el sistema.

Y en el trabajo... esperamos que aprovechen estas herramientas para hacer su estadía laboral más provechosa y satisfactoria: esto que dijimos no se termina en la facultad, es una metodología válida para construir software (al menos es la que utilizamos nosotros).

## 15.1 Y nos vamos...

Eso. Pásenla lindo.

# 16 Anexo: Diagrama de clases final

[](http://www.plantuml.com/plantuml/img/dP91IyGm48Nl-HLXJtfmqJyWb4Nm85XMJzw6oNHte4tA92cAx7ztsbgtxK5SRvtNzxvaNeirn6XB2H-4ATXFqqhes2D3t0LVOEAE7RYhQnDpIr8nCgcx8SUX5w5seT07KlkHX7mYdKu61-uh2C4SgkygmYCy-2IqWEhG19y6A9MP73q2alGoo85yeMJX86lwdhBdu2sz9_BkVEG4MvaHiWFySq1NTHb_zQzkEyLVI3hTgwyKGnp9rd2_HQLM-vGF8Ajg5csSp3VfBNZOPu7QSjFMr6Z7yOBuZdLZzx_iSwzuk3GyLdHpRpTtcp5W9fAAuuNCppyt-BPRlnchdQLlLRMf7sCfXDgorXSNlja7EdEcxiqN)

1. Está teniendo un alto acoplamiento la TareaOficina respecto del objeto Oficina [↑](#footnote-ref-0)
2. Este proceso de hacer cambios para permitir que una solución sea más general se llama *Refactoring*, y lo verán más adelante en la carrera [↑](#footnote-ref-1)