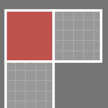
Versión 1.0

Paradigmas de Programación

Resolviendo un ejercicio en Objetos



Contenido

ENUNCIADO 3
NUESTRO OBJETIVO 4
¡TENGO MIEDO NENE! 4
TAREAS DE UN BUEN DISEÑADOR 4
ENCONTRANDO LOS OBJETOS 4
OBJETO TAREA 8
ALTERNATIVAS AL IF 8
PROGRAMAR ES HUMANO, DELEGAR ES DIVINO 9
RESPONSABILIDAD Y DELEGACIÓN 10
SUPERFICIE PROMEDIO DE LAS TAREAS 11
SUPERFICIE DE UNA TAREA 12
SALDO DE UN PROYECTO 13
SI HAY QUE CAMBIAR, SE CAMBIA 14
SALDO DE UN PROYECTO (2° PARTE) 16
SABIENDO SI SE PUEDE HACER UNA TAREA 19
PROYECTO Y TAREAS EN EL AMBIENTE 21
PROYECTO COHERENTE 22
MARGEN ANTERIOR DE UNA TAREA 24
QUÉ PUEDE CAMBIAR SI ESTOY HACIENDO EL TP 26
Y NOS VAMOS… 27
ANEXO: DIAGRAMA DE CLASES FINAL 27

ENUNCIADO

Desarrollar, de acuerdo al paradigma de objetos y escribiendo el código en lenguaje Smalltalk, 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:

a. 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 m^2. Para las ciudades se informa explícitamente la superficie en m^2. 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.

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

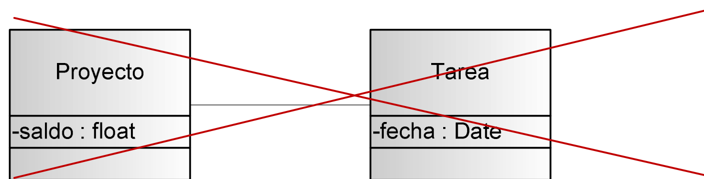
Y también… para aplicarlo en el trabajo

¡TENGO MIEDO NENE[1]!

¿Por dónde empezamos?

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.

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.



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.

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:

[1] Para más información consúltese a Alejandro Apo y la falsa frase famosa que nunca dijo.

"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 Workspace hacemos:

```
edificioInteligente := Proyecto new.
…
edificioInteligente provinciasConActividadEntre: unaFecha y: 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
- Una hoja tiene que seguir con el Workspace, o sea, todavía falta configurar el proyecto (es necesario agregar más líneas todavía)
- Otra hoja va a tener sin dudas el código del método provinciasConActividadEntre: y:[2]
- Una hoja opcional podría mostrar un diagrama de objetos con un proyecto puntual que creemos con el código generado en el Workspace

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**.

[2] En algunos casos puede resultarles útil tener una hoja para cada clase, en otros tener una hoja por cada punto del ejercicio (estarán todos los métodos de cada objeto relacionado con ese requerimiento).

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

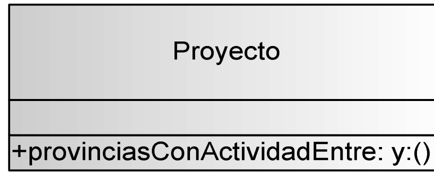


Diagrama de clases - versión 1

Ahora codificamos el método:

```
#Proyecto
provinciasConActividadEntre: unaFecha y: 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:

- Que cada tarea sabe decirme su provincia (no me importa cómo)
- 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.

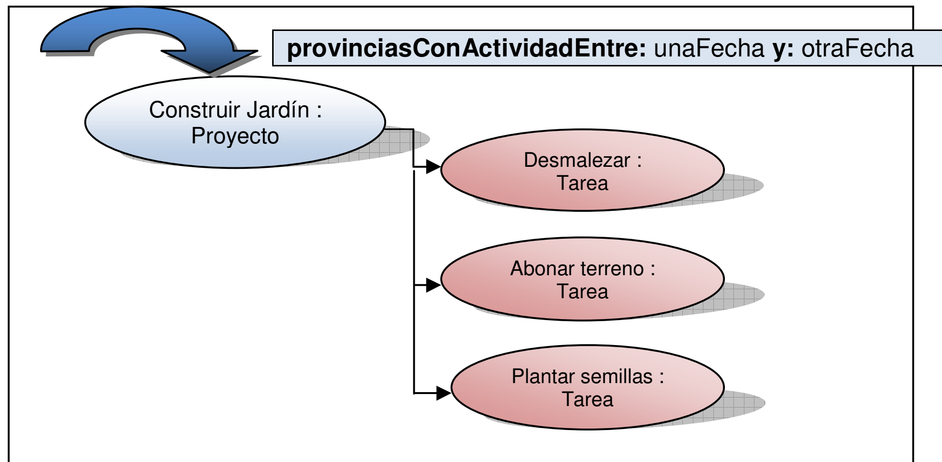


Diagrama de Objetos – Consulta de provincias con actividad para un proyecto

Lo anotamos en el diagrama de clases:

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 Smalltalk 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	filtrar → una colección	select:
"recolectar" las provincias de cada tarea	recolectar → una colección	collect:
eliminamos los duplicados de la colección de provincias	transformar una colección en Set → una colección	asSet

Ahora sí codificamos el método en cuestión:

```
#Proyecto
provinciasConActividadEntre: unaFecha y: otraFecha
    ^((self tareasEntre: unaFecha y: otraFecha) collect: [ :tarea | tarea provincia ]) asSet
tareasEntre: unaFecha y: otraFecha
    ^tareas select: [ :tarea | tarea fechaEntre: unaFecha y: otraFecha ]
```

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: and: que parece calzar justo. Codificamos entonces el método:

```
#Tarea
fechaEntre: unaFecha y: otraFecha
    ^fecha between: unaFecha and: 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.

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
provincia
    (tipoDeTarea = ‘O’)  “ Oficina “
        ifTrue: [ … ].
    (tipoDeTarea = ‘C’)  “ Ciudad ”
        ifTrue: [ … ].
    (tipoDeTarea = ‘R’)  “ Zona Rural “
        ifTrue: [ … ].
```

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:

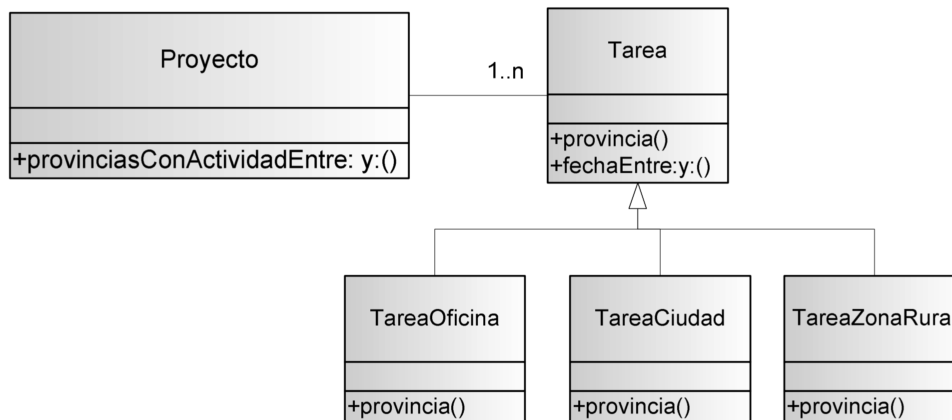


Diagrama de clases - versión 3

El método fechaEntre: y: 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.

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:

```
#TareaOficina
provincia
    ^oficina ciudad provincia
```

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

```
#TareaOficina
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, y eso es malo en tanto y en cuanto me quedo después de las 18:00 para arreglarlo.

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
provincia
    ^ciudad provincia
```

Actualizamos el diagrama de clases:

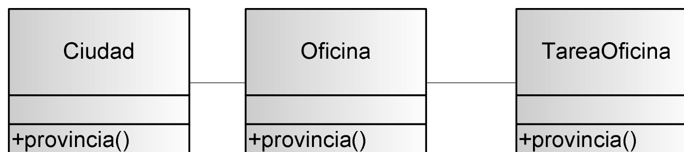


Diagrama de clases - versión 4

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 sólo es el valor lo que me importa? Entonces por el momento lo dejamos como un String.

Seguimos con los métodos de TareaCiudad que necesita un objeto ciudad al que le pueda pedir la provincia:

```
#TareaCiudad
provincia
    ^ ciudad provincia
```

El diagrama de clases queda:

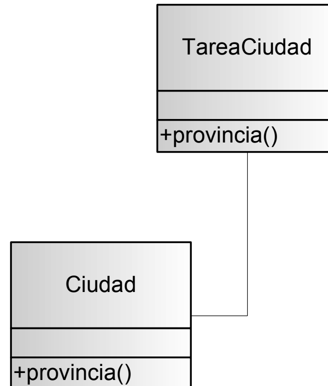


Diagrama de clases - versión 5

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

```
#TareaZonaRural
provincia
    ^ zonaRural provincia
```

Y el diagrama de clases queda:

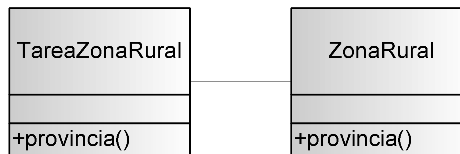


Diagrama de clases - versión 6

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 el Workspace el mensaje que le enviaríamos:

```
edificioInteligente := Proyecto new.
…
edificioInteligente provinciasConActividadEntre: unaFecha y: 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:

```
#Proyecto
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	inject: into:
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
superficieTotal
    ^tareas inject: 0 into: [ :acum :tarea | acum + tarea superficie ]

cantidadDeTareas
    ^tareas size
```

SUPERFICIE DE UNA TAREA

¿Cómo sabemos la superficie de una tarea?

- Para las oficinas se establece una superficie, la misma para todas en m^2.
- Para las ciudades se informa explícitamente la superficie en m^2.
- 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 almacenamos en una variable de clase, en la clase Oficina. Acá tenemos dos opciones:

```
#TareaOficina
superficie
    ^oficina superficie

#Oficina
superficie
    ^SuperficiePromedio
```

En esta implementación la oficina tiene un método de instancia superficie, que devuelve el valor de la variable de clase SuperficiePromedio.

```
#TareaOficina
superficie
    ^Oficina superficiePromedio
```

```
#Oficina
superficiePromedio (MC)
    ^SuperficiePromedio
```

Aquí la TareaOficina envía un mensaje a la clase Oficina (no al objeto oficina que él conoce). La desventaja de esta solución es que la TareaOficina no es tan inocente: sabe que todas las oficinas tienen una superficie promedio (no es lo mismo que delegar la responsabilidad en la oficina, por eso la primera alternativa resulta más deseable).
Seguimos con las tareas que se desarrollan en una ciudad:

```
#TareaCiudad
superficie
    ^ciudad superficie

#Ciudad
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
superficie
    ^zonaRural superficie

#ZonaRural
superficie
    ^ancho * largo
```

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 al Workspace y tratamos de darle un nombre a ese requerimiento:

```
edificioInteligente := Proyecto new.
…
edificioInteligente provinciasConActividadEntre: unaFecha y: otraFecha.
edificioInteligente superficiePromedio.
edificioInteligente saldoA: unaFecha.    ← saldo de un proyecto se lo pregunto… al proyecto
```

Ok, empezamos a codificar el método saldoA:

```
#Proyecto
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:

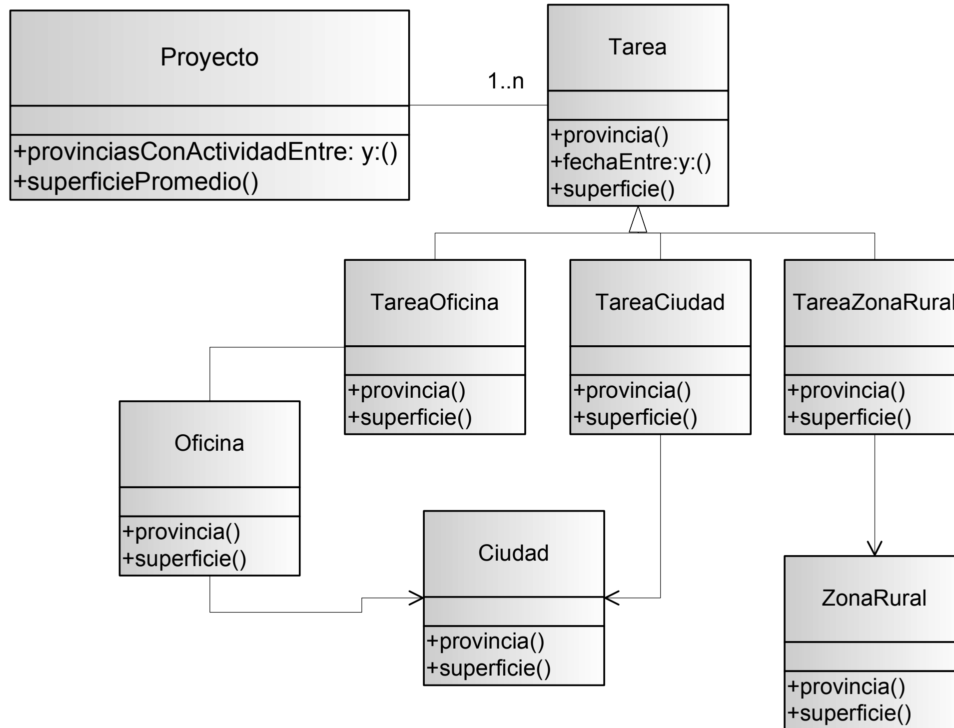


Diagrama de clases - versión 7

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!

SI HAY QUE CAMBIAR, SE CAMBIA

Ok, entonces tengo que decidir con cuál criterio de subclasificación 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
provincia
    ^oficina provincia
 	
superficie
    ^oficina superficie
	#TareaCiudad
	provincia
	    ^ ciudad provincia
	
	superficie
	    ^ciudad superficie
		#TareaZonaRural
		provincia
		    ^ zonaRural provincia
		
		superficie
		    ^ zonaRural superficie

Delego en oficina	Delego en ciudad	Delego 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:

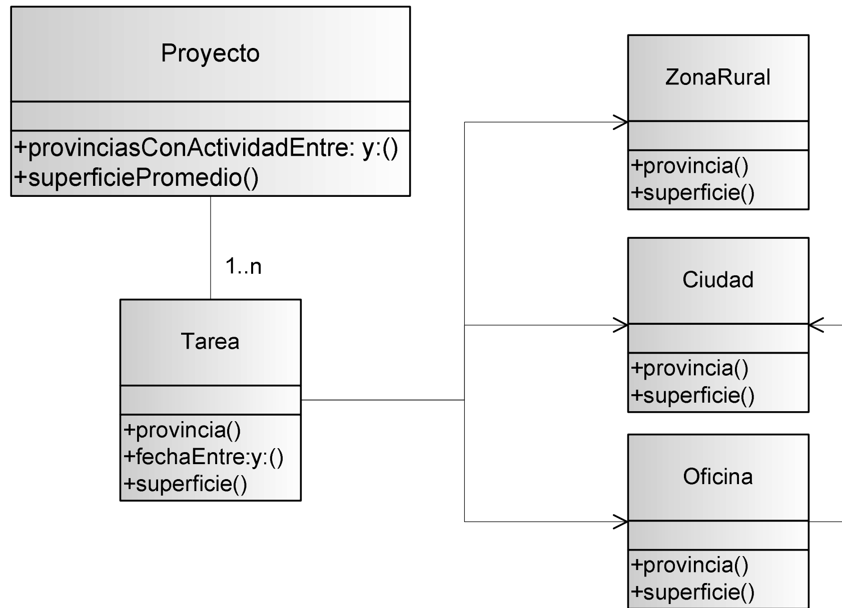


Diagrama de clases - versión 8

La relación entre la tarea y su lugar no es 100% UML, pero por el momento nos va a servir para entender que el lugar puede ser alguno de esos tres objetos, sin que necesariamente la Oficina, la Ciudad y la Zona Rural tengan una superclase en común (es lo que llamamos polimorfismo sin redefinición).

Revisamos el código de Tarea (que es el único que se modifica):

```
#Tarea (VI: lugar)
provincia
    ^lugar provincia

superficie
    ^lugar 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.

SALDO DE UN PROYECTO (2° PARTE)

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

```
#Proyecto
saldoA: unaFecha
    ^presupuestoInicial – (self montoDeTareasA: unaFecha)

montoDeTareasA: unaFecha
    …
```

Tenemos que filtrar las tareas hasta una fecha determinada. ¿Cómo hacemos? Podemos aprovechar el método tareasEntre: y:

```
#Proyecto
saldoA: unaFecha
    ^presupuestoInicial – (self montoDeTareasA: unaFecha)

montoDeTareasA: unaFecha
    ^(self tareasEntre: ¿? y: 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:

```
#Proyecto
saldoA: unaFecha
    ^presupuestoInicial – (self montoDeTareasA: unaFecha)

montoDeTareasA: unaFecha
    ^(self tareasEntre: self fechaInicio y: unaFecha) …
```

¿Qué ventaja tiene esto?
Podemos resolverlo de cualquiera de las dos maneras que planteamos arriba y no necesitamos cambiar la lógica del método montoDeTareasA: unaFecha

#Proyecto **fechaInicio** ^fechaInicio	#Proyecto **fechaInicio** ^self tareasOrdenadasPorFecha first fecha
Opción 1: getter	Opción 2: el proyecto busca la primera tarea que se hizo

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

```
#Proyecto
montoDeTareasA: unaFecha
    ^(self tareasHasta: unaFecha) …

tareasHasta: unaFecha
    ^self tareasEntre: self fechaInicio y: 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
saldoA: unaFecha
    ^presupuestoInicial – (self montoDeTareasA: unaFecha)

montoDeTareasA: unaFecha
    ^(self tareasEntre: self fechaInicio y: unaFecha)
            inject: 0 into: [ :total :tarea | total + 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.

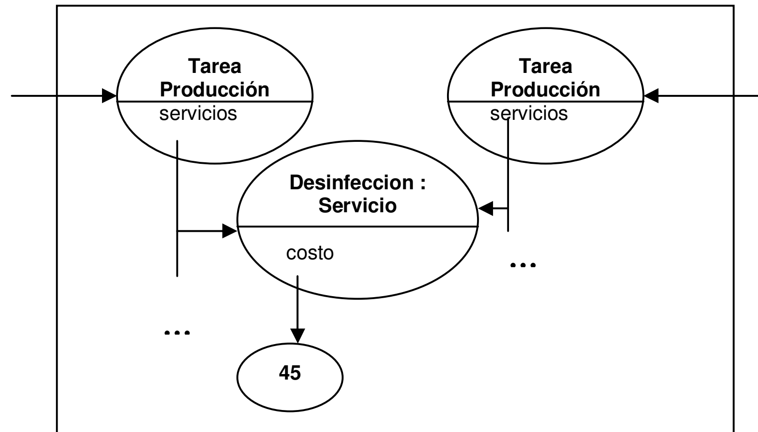
Ok, entonces hay comportamiento diferencial para las tareas de producción/tareas de recaudación/reuniones. ¿Dónde está ese "comportamiento diferencial"? En el método que calcula el monto:

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:



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

```
#TareaProduccion
monto
    ^tareas inject: 0 into: [ :total :servicio | total + servicio costo ]

#Servicio
costo
    ^costo
```

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):

```
#TareaRecaudacion
monto
    ^ingreso negated
```

El mensaje negated equivale a multiplicar por -1 (aquí pueden optar por lo que prefieran).

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

```
#TareaRecaudacion
monto
    ^0
```

Actualizamos ahora el diagrama de clases de nuestra solución:

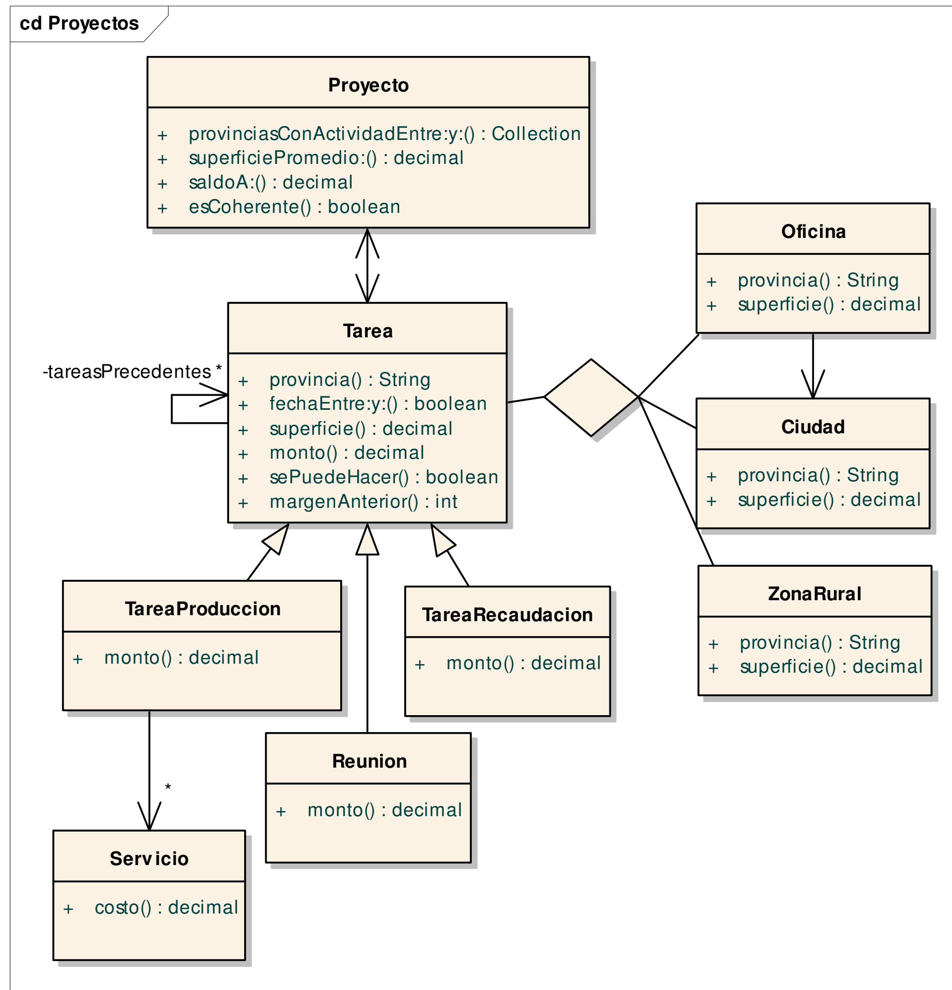


Diagrama de clases - versión 9

SABIENDO SI SE PUEDE HACER UNA TAREA

Bien, para este requerimiento, ¿a qué objeto tengo que enviarle un mensaje?
A una tarea.

En el Workspace tenemos que crear un objeto Tarea:

```
edificioInteligente := Proyecto new.
…
edificioInteligente provinciasConActividadEntre: unaFecha y: otraFecha.
edificioInteligente superficiePromedio.
edificioInteligente saldoA: unaFecha.

instalacionElectrica := TareaProduccion new.
…
instalacionElectrica sePuedeHacer.
```

La codificación del método, ¿es particular para cada tipo de tarea? Solamente la va a redefinir la tarea de producción, según lo que dice el enunciado:

"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 a la tarea 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 hacemos la comparación con nuestra fecha. Si todas las tareas lo cumplen, 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 Collection qué mensaje nos sirve para verificar que todos los elementos de una colección cumplen una determinada condición: allSatisfy:, entonces codificamos:

```
#Tarea
sePuedeHacer
    ^tareasPrecedentes allSatisfy: [ :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:** unaFecha en cualquier otro contexto (desde el proyecto, por ejemplo) y me sirve.

```
#Tarea
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
sePuedeHacer
    ^(tareasPrecedentes allSatisfy: [ :tarea | tarea sePuedeCumplirAntesDe: fecha ]) &&
     ((proyecto saldoA: fecha) >= 0)
```

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

```
#TareaProduccion
sePuedeHacer
    ^super sePuedeHacer && ((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
sePuedeHacerPara: unProyecto
    ^tareasPrecedentes allSatisfy: [ :tarea | tarea sePuedeCumplirAntesDe: fecha ]

#TareaProduccion
sePuedeHacerPara: unProyecto
    ^(super sePuedeHacerPara: 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:, 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.

PROYECTO Y TAREAS EN EL AMBIENTE

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

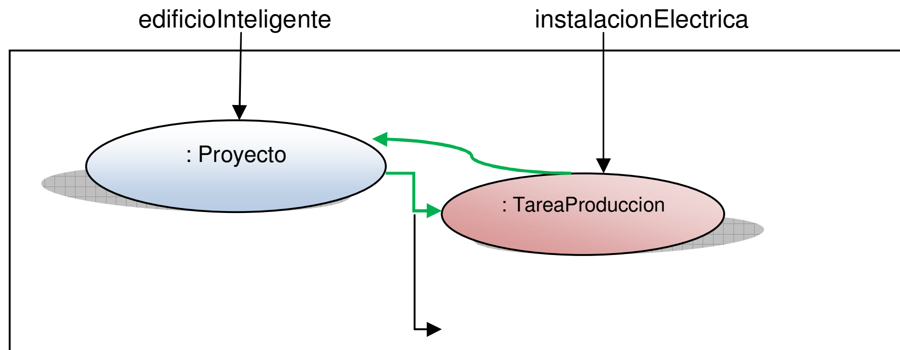


Diagrama de Objetos – Consulta Relación Proyecto/Tarea

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 Workspace hacemos:

```
edificioInteligente agregarTarea: instalacionElectrica.
```

El método agregarTarea: debe encargarse de mantener ambas referencias:

```
#Proyecto
agregarTarea: unaTarea
    tareas add: unaTarea.
    unaTarea proyecto: self.

#Tarea
proyecto: unProyecto
     proyecto := unProyecto.
```

Lo mismo para el eliminarTarea:.

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 un Workspace hacemos:

```
edificioInteligente := Proyecto new.
…
edificioInteligente provinciasConActividadEntre: unaFecha y: otraFecha.
edificioInteligente superficiePromedio.
edificioInteligente saldoA: unaFecha.

instalacionElectrica := TareaProduccion new.
…
instalacionElectrica sePuedeHacer.

edificioInteligente esCoherente.
```

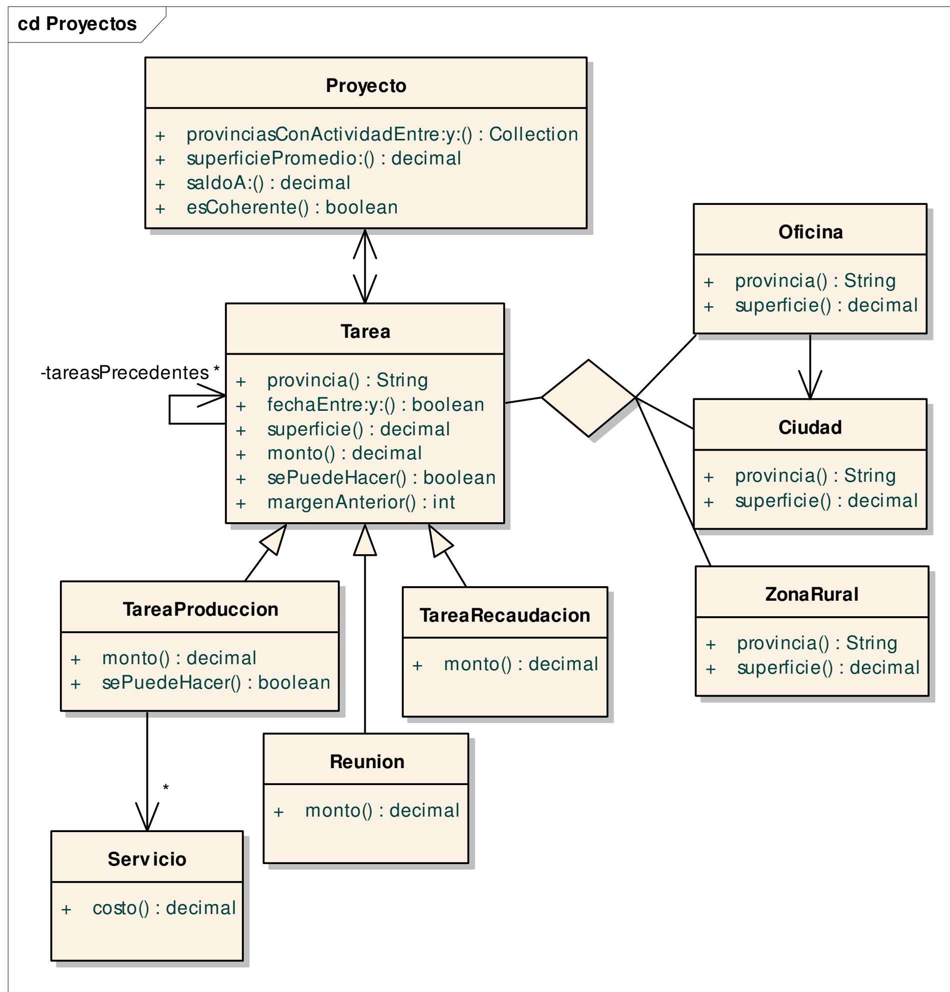
Entonces pasemos al código de Proyecto: ¿qué me interesa? Que todas las tareas se puedan hacer en la fecha indicada. Ya sabemos que eso se traduce en un mensaje a cada tarea específica y lo que me importa no es cómo lo hace, sino que me retorne un booleano si se puede hacer en la fecha prevista o no.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

```
#Proyecto
esCoherente
    ^tareas allSatisfy: [ :tarea | tarea sePuedeHacer ]
```

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
esCoherente
    ^tareas allSatisfy: [ :tarea | tarea sePuedeHacerPara: self ]
```

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



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.

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:

```
edificioInteligente := Proyecto new.
…
edificioInteligente provinciasConActividadEntre: unaFecha y: otraFecha.
edificioInteligente superficiePromedio.
edificioInteligente saldoA: unaFecha.

instalacionElectrica := TareaProduccion new.
…
instalacionElectrica sePuedeHacer.
instalacionElectrica margenAnterior.

edificioInteligente esCoherente.
```

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.

Pregunta: ¿está mal si considero que la lista de tareas precedentes ya está ordenada?

Eso implica que tareasPrecedentes tiene que ser una:

- ¿OrderedCollection? No, porque la ordered collection no me garantiza el orden por fecha sino por el orden en que la cargo. Entonces no es un buen criterio asumir que ya está ordenada.
- SortedCollection: ok, el tema es que uno no suele usar una SortedCollection ya que es costoso insertar y eliminar elementos. Además tampoco estoy seguro de que el criterio por el cual está ordenada la SortedCollection sea por fecha (eso depende del sortBlock: que dice cómo se ordenan los elementos).

Entonces, es buena decisión que yo ordene las tareasPrecedentes por fecha. Esa responsabilidad puede estar en el método margenAnterior, o mejor, puede estar afuera de ese método:

```
#Tarea
margenAnterior
    ^(tareasPrecedentes asSortedCollection: [ :tarea1 :tarea2 | tarea1 fecha > tarea2 fecha ]) …
```

Opción 1: Ordeno las tareas precedentes por fecha dentro del método margen anterior

```
#Tarea
margenAnterior
    ^self tareasPrecedentesPorFecha …

tareasPrecedentesPorFecha
    ^tareasPrecedentes asSortedCollection: [ :tarea1 :tarea2 | tarea1 fecha > tarea2 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[3]).

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
margenAnterior
    self tieneTareasPrecedentes ifFalse: [ ^0 ].
    self tareasPrecedentesPorFecha first … (falta)

tieneTareasPrecedentes
    ^tareasPrecedentes notEmpty

tareasPrecedentesPorFecha
    ^tareasPrecedentes asSortedCollection: [ :tarea1 :tarea2 | tarea1 fecha > tarea2 fecha ]
```

Y lo que faltaría es ver cómo calculo la diferencia entre dos fechas. Vemos la guía de mensajes de Date:

subtractDate: aDate
    "Answer the difference in days between the receiver and aDate, as an Integer"

Ok, entonces ya lo tenemos:

```
#Tarea
margenAnterior
    | ultimaTarea |
    self tieneTareasPrecedentes ifFalse: [ ^0 ].
    ultimaTarea := self tareasPrecedentesPorFecha first.
    ^fecha subtractDate: ultimaTarea fecha
```

[3] Si bien la cohesión es un concepto que van a trabajar más adelante en Diseño de Sistemas, aquí hay un ejemplo práctico que valía la pena comentar.

```
tieneTareasPrecedentes
    ^tareasPrecedentes notEmpty

tareasPrecedentesPorFecha
    ^tareasPrecedentes asSortedCollection: [ :tarea1 :tarea2 | tarea1 fecha > tarea2 fecha ]
```

QUÉ PUEDE CAMBIAR SI ESTOY HACIENDO EL TP

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 como funciona mi aplicación.

Y para eso es muy importante mi Workspace para armar esos ejemplos. Entonces volvemos sobre lo dicho en la página 5:

- "Una hoja tiene que seguir con el Workspace, o sea, todavía falta configurar el proyecto (es necesario agregar más líneas todavía)"

Y bueno, lleva un tiempo pero es tiempo que se aprovecha mucho. Entonces creo un proyecto puntual con tareas puntuales. Ya no hablamos en forma genérica sino con instancias concretas. Eso es lo que llamamos un juego de datos.

Un ejemplo posible:

```
proyecto := Proyecto new.
proyecto presupuestoInicial: 500.
tarea := TareaProduccion new.
proyecto agregarTarea: tarea.
```

Dos recomendaciones:

- A veces está bueno generar constructores piolas para mejorar la expresividad y reducir la cantidad de líneas de código de mi Workspace
- 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).

```
servicioDesinfeccion := Servicio new costo: 45.
piso1 := TareaProduccion para: Date today.
piso1 agregarServicio: servicioDesinfeccion.
edificioEstomba5121 := Proyecto nuevoConPresupuestoInicial: 500.
edificioEstomba agregarTarea: piso1.
```

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.

Ejemplos:

Condición	Resultado esperado
El proyecto edificioEstomba5121 tiene configuradas tareas con precedencias incorrectas. Le pregunto si es coherente.	False
Configuro la tarea de producción piso1 con dos servicios: desinfección ($ 45) y pulido ($ 21). Le pregunto el costo a piso1	$ 66
… etc …	

Armar los casos de prueba puede no resultar una tarea apasionante, pero:

- Por un lado ayuda a entender el dominio que estoy resolviendo
- Por otra parte ayuda a no escribir código “a la ligera”
- Si me tomo el trabajo de codificarlo en el Workspace, me puede servir para probar mi código n veces.

Ayuda que los casos de prueba sean representativos: probar 7 casos similares con tareas de recaudación puede no tener mucha onda. En cambio sí tiene sentido hacer pruebas con cada tipo de tarea, e incluso dentro de las tareas de producción contemplar los valores “límite”: una tarea de producción que no tenga servicios, que tenga 1 servicio o que tenga n.

En definitiva los casos de prueba son una herramienta más que nos dan un grado mayor de certeza que la prueba al azar y manual, sobre todo cuando modificamos nuestro código, porque tenemos la posibilidad de volver a evaluar los mensajes del Workspace y ver si el sistema cambió su comportamiento.

Y NOS VAMOS…

Eso. Pásenla lindo.

ANEXO: DIAGRAMA DE CLASES FINAL