Paradigma Orientado a Objetos

Módulo 16: El proceso de diseño.

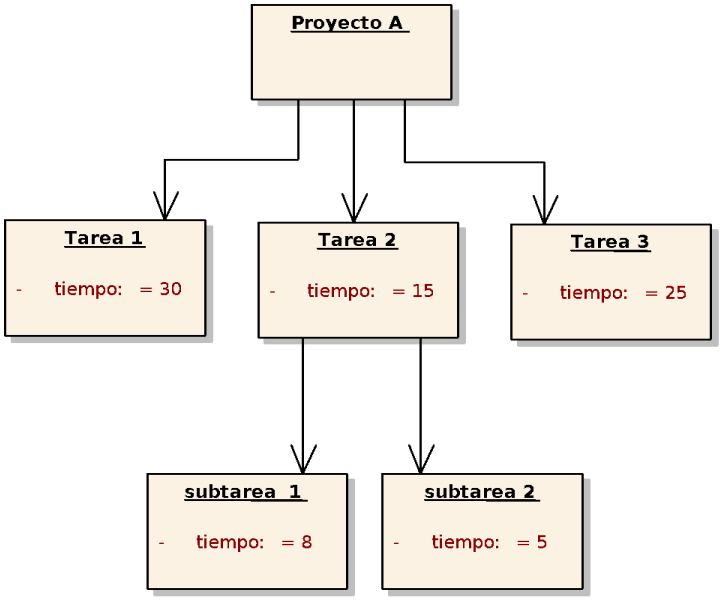
por Fernando Dodino

Versión 3.0
Octubre 2017

Indice

1 Enunciado

La compañía Kendari Corporation INC. desea desarrollar para sus sucursales internacionales una herramienta de manejo de proyectos.



Los proyectos se componen de tareas y las tareas de subtareas (las subtareas pueden tener a su vez subtareas). Cada una de estas tareas y subtareas tiene un costo en tiempo y en dinero según se muestra en la figura siguiente:

El costo de las tareas se calcula de acuerdo a la complejidad de la misma, a saber:

Complejidad mínima: tiempo * $25
Complejidad media: (tiempo * $25) + 5% de lo anterior
Complejidad máxima:

- si tiempo es menor o igual a 10 días --> tiempo * $25 + 7%
- si tiempo es mayor a 10 días --> (tiempo * $25) + 7% + $10 por cada día después del décimo

Nota: Para calcular el costo de una tarea sólo se utiliza el tiempo específico de la misma. Por ejemplo para calcular el costo de la tarea 2, el tiempo es a utilizar es 15.

A su vez las tareas que tengan más de 3 subtareas asociadas tienen un costo extra por overhead del 4%.

Por último hay un costo impositivo extra que dependerá de la reglamentación fiscal del país y algunos otros factores, lo importante es que existe el impuesto A (3% del valor) y el impuesto B (5% del valor). Hay tareas en donde aplican los dos impuestos, algunas que aplica sólo uno de ellos y otras que no se aplica ninguno.

Se debe poder determinar el porcentaje de completitud de cada tarea. Las tareas que no posean subtareas sólo pueden estar completas o no (al 0% o al 100% taxativamente). En cambio, en las tareas que posean subtareas se calculará en base al promedio ponderado de las mismas.

Nota: Los impuestos sólo se aplican sobre la tarea que los tiene, no sobre las subtareas.

Otro punto importante para el sistema es poder calcular el atraso posible en una tarea. Este dato depende exclusivamente de la complejidad y el tiempo de la misma y se calcula de la siguiente forma:

Tareas de complejidad mínima: 5 días.
Tareas de complejidad media: 10% del tiempo de la tarea.
Tareas de complejidad máxima: 20% del tiempo de la tarea + 8 días.

Aplicar el diseño necesario que permita:

- Obtener el costo de una tarea
- Obtener el costo total de un proyecto
- Obtener los días máximos de atraso de un proyecto
- Saber o asignar el porcentaje de completitud de una tarea

2 Objetivo de la clase

Esta clase tiene varios objetivos: la primera y principal es plantear algunas decisiones que no son triviales, **y son decisiones que tienen que ver específicamente con nuestro diseño**. En esta clase no nos importará tanto el resultado final de ese diseño, sino cómo es el proceso que me lleva a elegir alternativas.

De aquí se desprenden otros objetivos periféricos

- encontrar formas objetivas de comparar soluciones posibles y decidir en consecuencia
- repasar cómo la composición es una herramienta más flexible que la herencia
- definir cuándo elijo modelar con clases y cuándo con instancias
- encontrar una metodología iterativa que permita que me equivoque

entre otros que iremos descubriendo a lo largo del apunte.

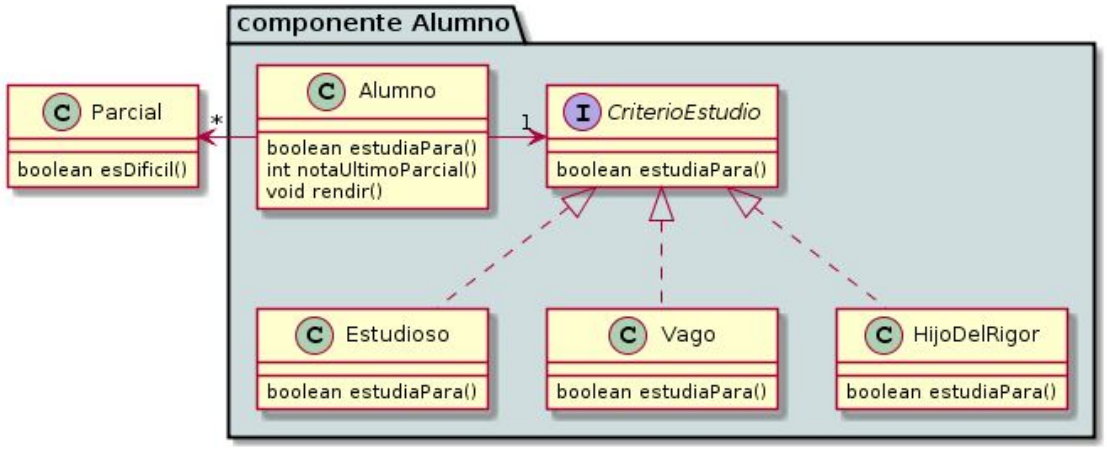
3 Qué es diseñar

Preguntamos a la clase qué es diseñar, salen respuestas que sin duda son valiosas. Nosotros vamos a quedarnos con algunas definiciones que nos sirven para disparar preguntas, antes que para estudiar de memoria para un examen:

Diseñar es

- encontrar los componentes
- qué hacen esos componentes
- cómo se relacionan

En particular, dentro del paradigma orientado a objetos, los componentes pueden ser objetos, o bien un conjunto de ellos que brindan servicios para resolver los requerimientos del usuario. Por ejemplo: cuando en el módulo anterior hablamos de un alumno y el criterio de estudio, podemos pensar en que ambos forman un único componente que nos sabe decir si un alumno va a estudiar para un parcial:



4 Algunas cuestiones sobre el diseño

4.1 El diseño en otros paradigmas

Vale recordar que tanto en lógico como en funcional hemos estado diseñando, solo que los componentes se construyen de diferente manera. Aquellos que deseen profundizar al respecto pueden leer el apunte Introducción al Diseño de Sistemas.

4.2 Cuánto diseñar

Algunas metodologías proponen pensar **todo** el diseño antes de programar, nosotros preferimos concentrarnos en algunas decisiones, y postergar otras. ¿Por qué postergo algunas decisiones?

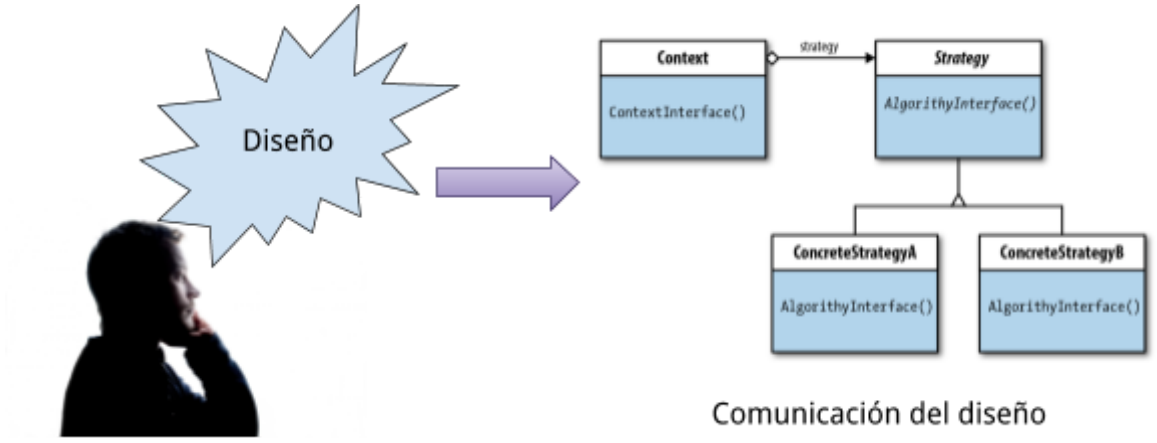
- No siempre tengo toda la información
- El usuario y el desarrollador van aprendiendo sobre el dominio mientras se construye la aplicación: evito así tomar decisiones prematuramente
- Los requerimientos cambian

Entonces cuando tengo alguna idea, la valido con algo de código. De esa manera puedo darme cuenta de si es implementable la idea o no. No tiene que ser todo el código, ni todas las alternativas posibles, pero sí algo que me permita saber si estoy yendo bien o no. Al no diseñar todo, es más fácil aceptar el error y volver a pensar una solución. El error es parte de la naturaleza humana, deberíamos adoptar una metodología que los tolere y no que los esconda.

4.3 Diseño y diagramas

Queremos dejar en claro que el diseño ocurre en la mente de las personas

cuando generan abstracciones, por lo tanto diseñar no es saber UML, en todo caso saber UML me permite comunicar el diseño. Puedo o no tener documentación formal, eso no excluye que el **diseño ya exista**.



Comunicación del diseño

5 Manejo de proyectos

Les damos 10/15 minutos para leer el enunciado.

5.1 Por dónde arrancar

Leer una serie de requerimientos puede ser un poco abrumador, pero a esta altura contamos con muchas herramientas que nos van a ayudar a resolver lo que nos piden.

Primero que nada, no hay que perder de vista los requerimientos (a veces en forma de caso de uso[1], a veces con el formato de *user story*), porque son los que deben guiar el proceso. ¿Qué necesitamos resolver?

- el costo de una tarea
- el costo de un proyecto
- la cantidad máxima de días de atraso de un proyecto

Es importante concentrarse en lo que tenemos que resolver para no hacer cosas de más.

Ahora bien, para poder calcular el costo de una tarea deben intervenir muchos componentes, entonces para poderlos encontrar tenemos a mano varias opciones:

- Diagrama de clases (o estático)
- Diagrama de objetos

[1] Para más información se puede leer el apunte Las entradas del diseño

- Buscar objetos **candidatos** (no todos van a quedar) y responsabilidades que pueden cumplir
- Pensar en cómo lo pruebo: escribir un test primero (técnica que se conoce como TDD)
- O bien podría ser que imagine una clase o un objeto tarea y empiece a tirar código, que es una posibilidad más para iniciar nuestro diseño. Lo importante es saber frenar en algún momento y levantar la cabeza para que no se transforme en una actividad “code & fix”.

Tener varias opciones nos permite comenzar por cualquiera de las variantes que mejor nos parezca, o con la que nos sintamos más cómodos.

Cuando me trabo... cambio la perspectiva del problema:

1. si estoy haciendo un diagrama estático, podría tirar código (en otra situación podría ser tratar de escribir un test o pensar un diagrama de objetos)
2. pregunto al que entiende del negocio, o
3. no tomo este camino y postergo la decisión.

De esa manera, mis herramientas para diseñar y la metodología que adopto permiten que cuando me trabe pueda seguir avanzando con mi solución.

Nosotros vamos a escribir una lista de objetos candidatos.

5.2 Objetos candidatos

Armamos una lista tentativa, no necesariamente todos van a quedar:

- Proyecto
- Tarea
- Complejidad
- Impuesto
- Subtarea

5.3 Diagrama estático: primeros intentos

Para resolver el costo de una tarea vemos que eso depende

- de si la tarea tiene subtareas
- de la complejidad de la tarea (mínima, media o simple)
- de los impuestos que tenga una tarea

A su vez, veamos lo que sucede con el porcentaje de completitud:

- las tareas que no tienen subtareas se pueden marcar como cumplidas,

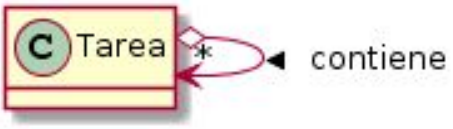
las tareas que tienen subtareas no
- el % de completitud de una tarea se calcula distinto para una tarea que tiene subtareas y para las que no tienen subtareas

Entonces, una decisión importante a resolver es ¿cómo represento a las tareas y las subtareas?

Opción 1

Mmm... no me sirve porque no permite que una subtarea tenga subtareas.

Opción 2: una tarea tiene una colección de objetos tarea (referencia recursiva). Si la colección está vacía, la tarea no tiene subtareas.

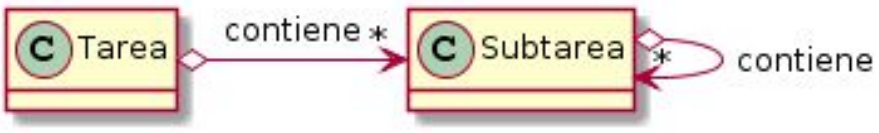


Si bien es una solución flexible, nos obliga a preguntar si la tarea tiene subtareas en tres lugares

1. para determinar el % de completitud,
2. para asignar el % de completitud
3. e incluso el % de costo extra depende de que la tarea tenga subtareas

Entonces la tarea es un objeto que tiene **baja cohesión**: representa a las tareas que tienen y las que no tienen subtareas.

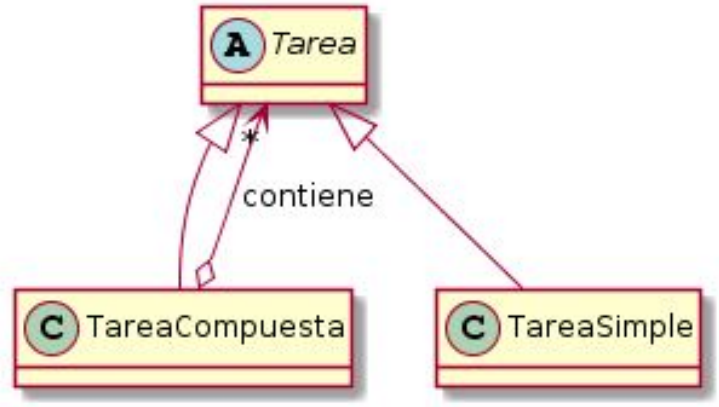
Opción 3



Esta solución tiene dos desventajas: es la más compleja de las tres y además no permite reutilizar conceptos que a priori parecen similares como la tarea y la subtarea. Tampoco discrimina las tareas que tienen y las que no tienen subtareas.

En realidad, lo que pasa es que el enunciado tiene una trampa de redacción: lo importante no es diferenciar la tarea de la subtarea, sino discriminar las tareas que tienen subtareas vs. las tareas que no tienen subtareas (como estuvimos escribiendo en las dos páginas anteriores, vuelvan a releer el apunte si no nos creen)

Opción 4



Al subclasificar la tarea le puse un nombre mejor (encontré una abstracción).

Pero no olvidemos que para calcular el costo y el tiempo máximo de atraso de una tarea hay comportamiento diferencial para las tareas de complejidad mínima, media y máxima. Entonces si pensaba subclasificar la tarea en TareaComplejidadMinima, TareaComplejidadMedia y TareaComplejidadMaxima tengo un conflicto, porque la herencia solo permite armar taxonomías (clasificaciones) por un único punto de vista.

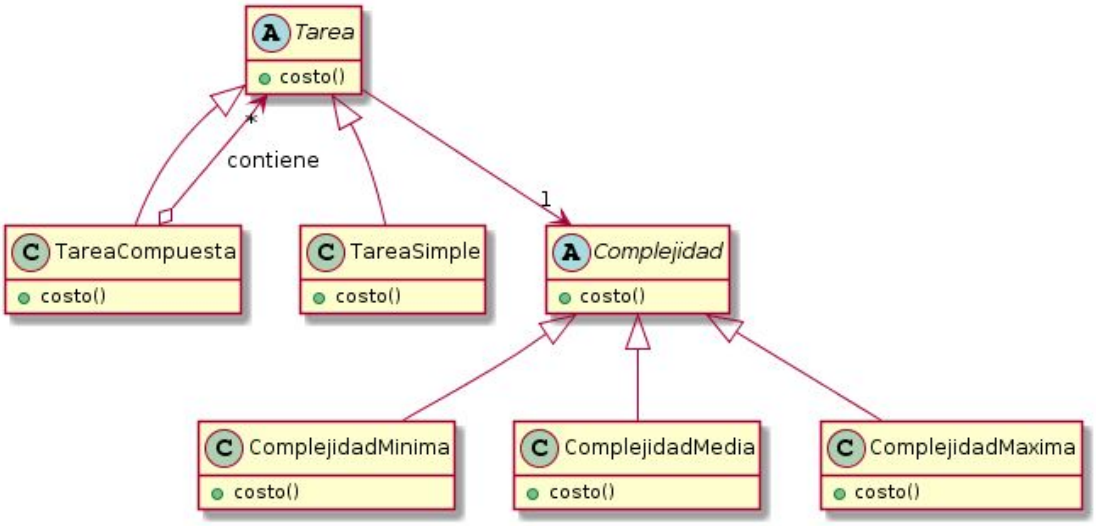
Separamos dos tipos de clasificación posibles para tareas:

- Tarea simple y compuesta
- Tarea mínima, media y compleja

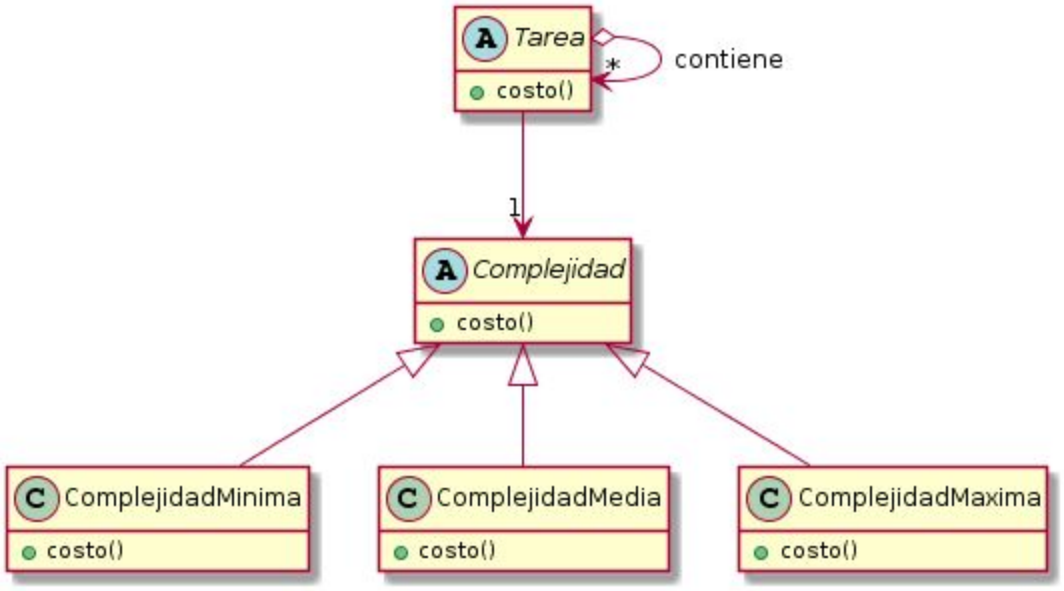
Si ambas jerarquías no pueden convivir juntas, recordemos que una alternativa a la herencia es la **composición**, entonces tenemos diferentes alternativas posibles:

Opción 1

Según esta solución nos interesa separar la tarea de su complejidad. La consecuencia es que es costoso hacer que una tarea simple devenga en compuesta, mientras que cambiar la complejidad de una tarea es simplemente intercambiar el objeto que representa la complejidad.



Opción 2



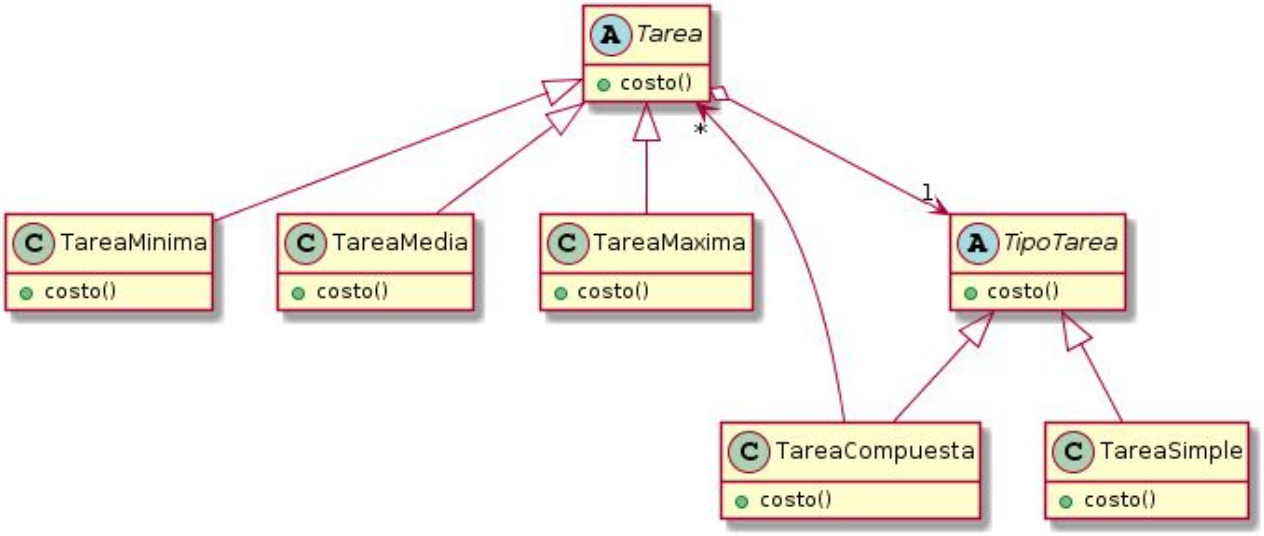
Según esta solución es posible que una tarea de complejidad mínima pase a ser de complejidad media. Por otra parte si en muchos lugares se discrimina el comportamiento entre tareas simples y compuestas, habrá muchos métodos que pregunten con un if si la tarea es simple o compuesta. Podemos reificar la decisión en un método esSimple():

```
if (self.esSimple())
     <-- se abstrae en un método el concepto simple
     ... comportamiento de tarea simple...
} else {
     ... comportamiento de tarea compuesta...
```

}

Aun así la Tarea debe contemplar ambos casos y pierde cohesión (sabe hacer las cosas de la tarea simple y de la compuesta también).

Opción 3



Con esta solución es más fácil pasar una tarea simple a compuesta que cambiar la complejidad de una tarea.

¿Qué hacemos entonces?

No puedo decidir yo, pero **el proceso de diseño me ayuda a hacer una pregunta valiosa al usuario/analista**. El usuario dice: “las tareas podrían cambiar su complejidad, es decir, una tarea fácil podría pasar a ser compleja, en cambio una tarea simple no se transforma en compuesta ni viceversa”.

En base a este último requerimiento, es más fácil subclasificar la tarea en simple y compuesta y delegar en otro objeto la complejidad, nos decidimos por la primera opción. A esta altura desaparece la idea de subtarea (queda como objeto candidato nomás).

¿Qué falta?

1. calcular el costo (de una tarea y de un proyecto), para lo cual debemos
    a. resolver el costo extra por cantidad de subtareas
    b. resolver el costo impositivo
2. atacar la completitud del proyecto

Toda esta lista de tareas pendientes las anotamos en el pizarrón, en un papel,

en un archivo de texto plano, abrimos un issue en Github o una tarjeta en Trello, como quieran. Lo importante es saber qué cosas tenemos que encarar.

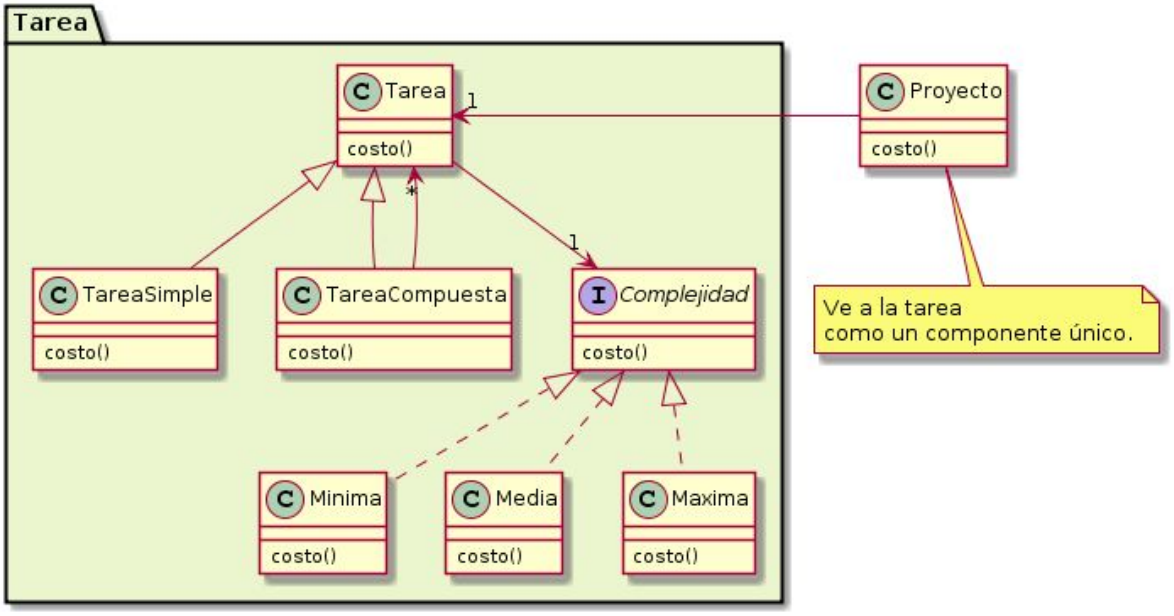
5.4 Identificando los requerimientos a resolver

Cada uno se resuelve enviando un mensaje a un objeto. Ok, a qué objeto le mando qué mensaje

- obtener el costo de una tarea ➪ a una tarea.costo()
- obtener el costo total de un proyecto ➪ a un proyecto.costoTotal()
- asignar % de completitud a tarea simple ➪ a una tareaSimple.porcentajeCompletitud(x)
- obtener % de completitud de un proyecto ➪ a un proyecto.porcentajeCompletitud()
- obtener el tiempo total de un proyecto ➪ a un proyecto.tiempoTotal()
- obtener los días máximos de atraso de un proyecto ➪ a un proyecto.diasMaximoDeAtraso()

En este ejercicio todo parece fluir normalmente... hasta que aparece el caso típico del siguiente requerimiento: “Quiero saber el saldo de un cliente”. ¿A quién le pregunto eso? Podría parecer "poco natural" preguntárselo al cliente, ya que en la realidad yo no le pregunto al cliente. El tema es que objetos trabaja con un modelo que es una simplificación de la realidad. El cliente que yo represento como objeto no es igual al cliente físico, es una abstracción para mi sistema, y está bien que en esa abstracción el saldo sea un comportamiento de mi cliente que vive en un ambiente de objetos, y no tiene nada que ver con el cliente físico[2].

[2] Para más información ver el apunte Guías para comunicar un diseño



Tenemos componentes claramente delimitados por el rectángulo verde. No hablamos de componente a nivel objeto, sino que un componente agrupa a varios objetos que cumplen una funcionalidad (todos los objetos incluidos en el rectángulo me ayudan a saber el costo de una tarea).

¿Para qué sirve pensar en componentes? Si hacemos cambios sin cambiar la interfaz de Tarea, el Proyecto no debería verse afectado. Y acá hablamos de interfaz como la parte visible / los servicios / lo que me puede contestar un objeto.

Ahora sí vamos al código para calcular el costo.

5.5 Costo de una tarea

```
class Tarea {
	method costo() = ...
}
```

Si recordamos que una tarea depende

- de la complejidad
- del costo por overhead (para subtareas)
- de los impuestos

Podemos pedir que el costo tenga esas tres abstracciones:

```
method costo() = self.costoComplejidad() +
      self.costoPorOverhead() +
      self.costoImpositivo()
```

El costoPorOverhead

- se puede definir como abstracto en tarea y redefinirlo para tarea simple o compuesta. Esto tiene como consecuencia que nuevas subclases de tarea están forzados a definir su comportamiento y no toman un valor por defecto, lo cual puede ser una ventaja o una desventaja.
- o bien se puede definir por defecto 0 en tarea y redefinirlo en tarea compuesta. Esto permitiría que Tarea pueda instanciarse, algo que deberíamos preguntar si es lo que queremos.

Decidimos que Tarea sea una clase abstracta y que el método costoPorOverhead() sea abstracto en Tarea:

```
method costoPorOverhead()
```

En TareaSimple se redefine para que devuelva 0

```
class TareaSimple inherits Tarea {
      override method costoPorOverhead() = 0
}
```

y en Compuesta preguntando si las tareas son más de 3. La desventaja de esta solución es que tenemos que calcular dos veces el costo de la complejidad (podemos almacenar ese valor y pasarlo como parámetro).

```
class TareaCompuesta inherits Tarea {
      override method costoPorOverhead() =
            if (self.tieneMuchasSubtareas())
                  self.costoComplejidad() * 0.04
            else
                  0
}
```

Fíjense que elegimos delegar en otro método la decisión de “tener muchas subtareas” lo que permite que lo utilicemos en otro contexto:

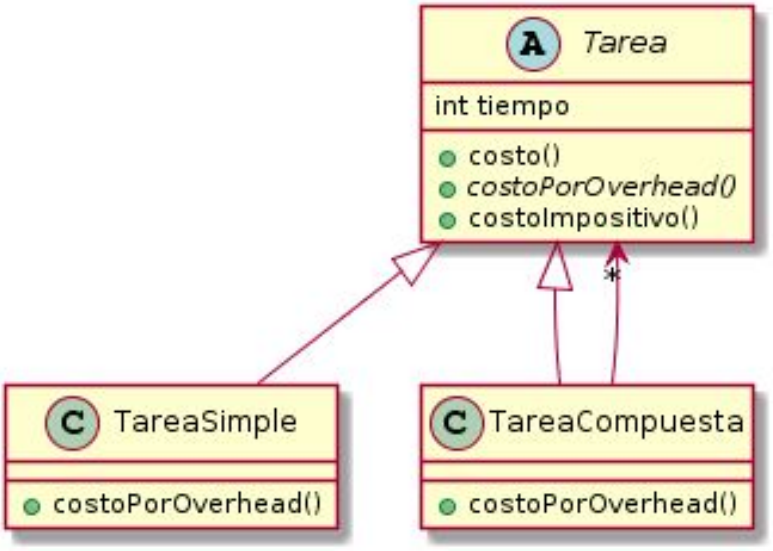
```
class TareaCompuesta inherits Tarea {
      const subtareas = []

      override method costoPorOverhead() =
            if (self.tieneMuchasSubtareas()) ...

      method tieneMuchasSubtareas() = subtareas.size() > 3
```

```
}
```

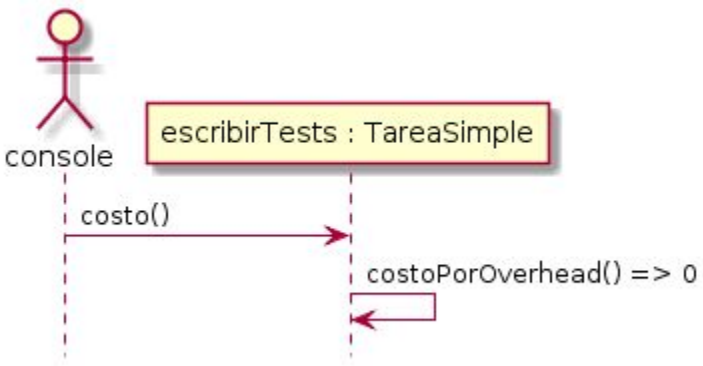
Y aun cuando todavía no tenemos implementado el costo, queremos contar una idea interesante que acabamos de implementar, el patrón **Template Method,** que consiste en definir una responsabilidad (el costo) en una clase abstracta que delega parte de la resolución en cada una de las subclases (costo por overhead).

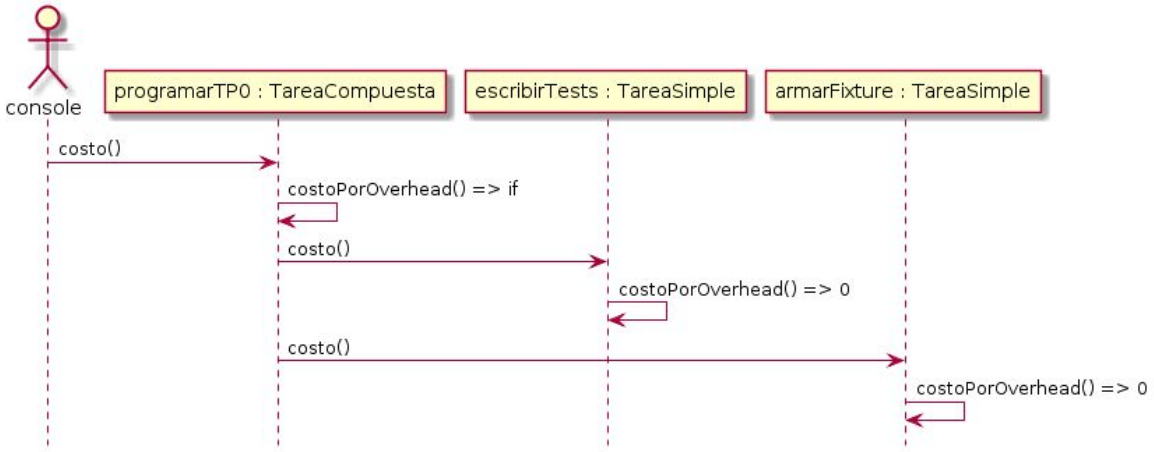


Si tenemos una tarea simple y una compuesta:

```
>>> const escribirTests = new TareaSimple()
>>> escribirTests.costo()
>>> const programarTP0 = new TareaCompuesta()
>>> const armarFixture = new TareaSimple()
>>> programarTP0.agregarTarea(armarFixture)
>>> programarTP0.agregarTarea(escribirTests)
>>> programarTP0.costo()
```

Podemos ver cómo se resuelve el método costoPorOverhedad para *escribirTests* y para *programarTP0*:

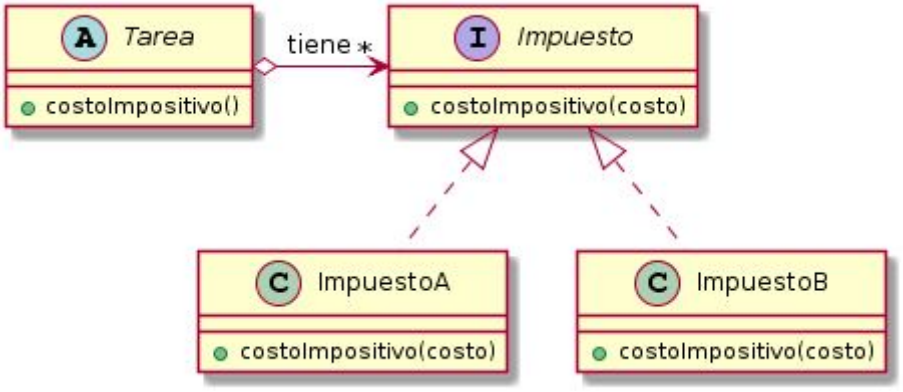




5.6 Agregando los impuestos

Tenemos distintos impuestos que pueden aplicar sobre una tarea, podemos considerar que una solución posible sería:

Opción 1: *los impuestos como clases*



Ahora, ¿en qué se diferencia cada impuesto en particular?
Y… en el impuesto A es un 3% y el impuesto B es un 5%. Es decir, si codificamos el impuesto A y el B, solo se diferencian en el porcentaje a descontar.

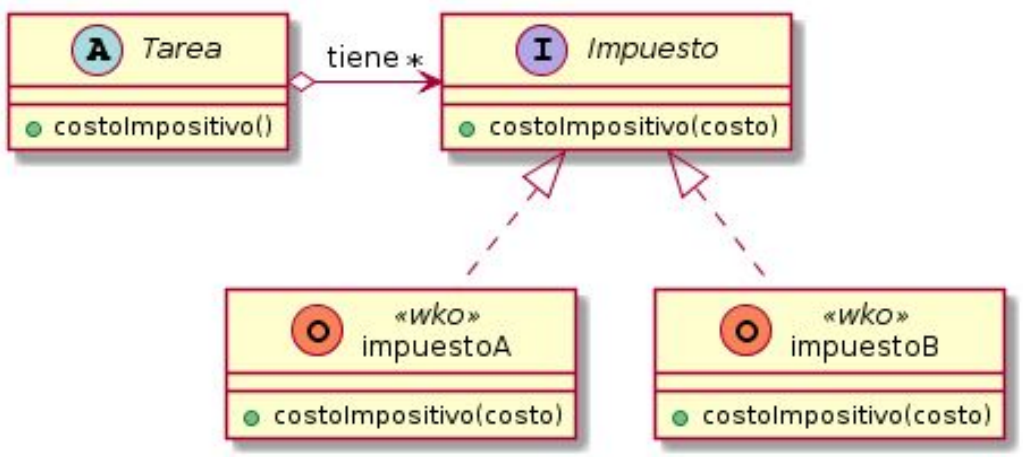
```
class ImpuestoA {
	method costo(costo) = costo * 0.03
}

class ImpuestoB {
	method costo(costo) = costo * 0.05
}
```

En esta solución no existe la necesidad de modelar una superclase Impuesto, por lo que ImpuestoA e ImpuestoB comparten una interfaz común, que en

Wollok no se codifica.

Opción 2: los impuestos como wko



Una segunda opción en Wollok podría ser modelar el impuesto A y B como wko. En cuanto a código, no es mucho el cambio, simplemente los class pasan a objects:

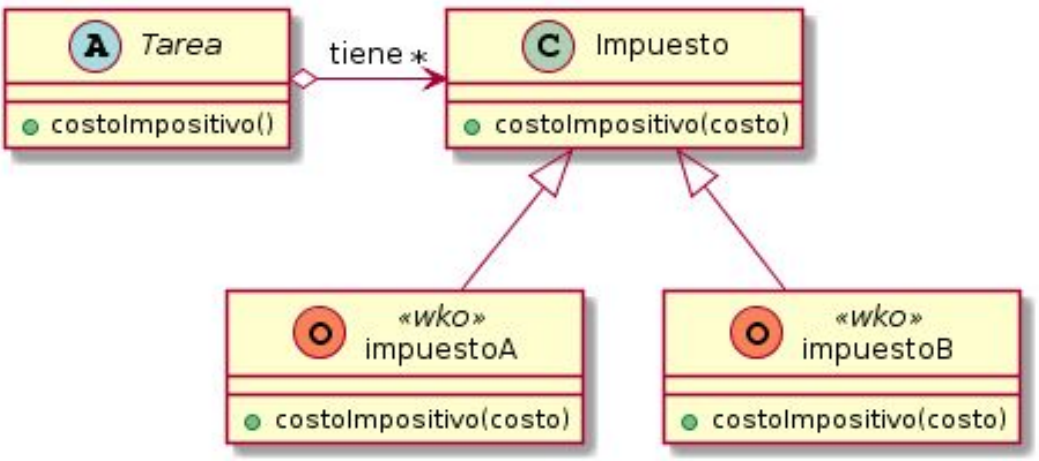
```
object impuestoA {
	method costo(costo) = costo * 0.03
}

object impuestoB {
	method costo(costo) = costo * 0.05
}
```

La diferencia está en el uso:

- el impuesto A y B se pueden acceder globalmente
- también se pueden compartir entre varias tareas, ya que no tienen estado (no tienen referencias específicas de una tarea). Como consecuencia de esto, puedo tener 5490 tareas apuntando a dos impuestos A y B.
- aun así, estoy modelando dos *named objects* que internamente comparten un comportamiento similar.

Opción 3: los impuestos como *named objects (wko)* heredando de una superclase común



Si queremos evitar la repetición de la idea, podríamos decir en un solo lugar cómo es el cálculo, y tener dos wko específicos para el impuesto A y el B.

```
class Impuesto {
	const porcentaje

	method costo(costo) = costo * porcentaje / 100
}

object impuestoA inherits Impuesto (porcentaje = 3) {}
object impuestoB inherits Impuesto (porcentaje = 5) {}
```

Aquí tenemos algunas ventajas:

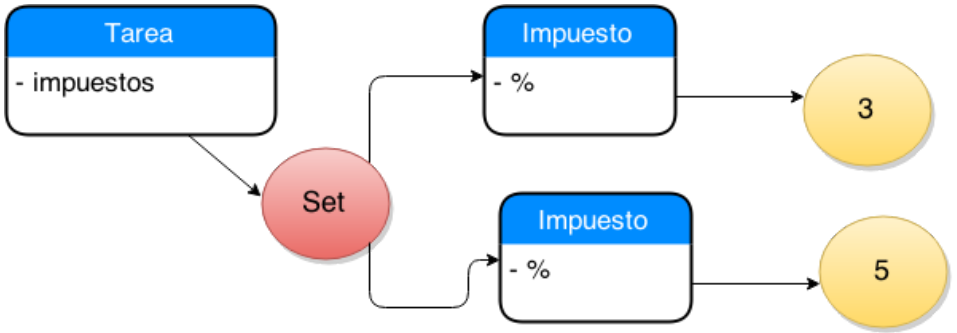
- el comportamiento del impuesto se define una única vez, y es posible redefinirlo en cada impuesto
- los impuestos A y B se acceden globalmente, como en el ejemplo anterior

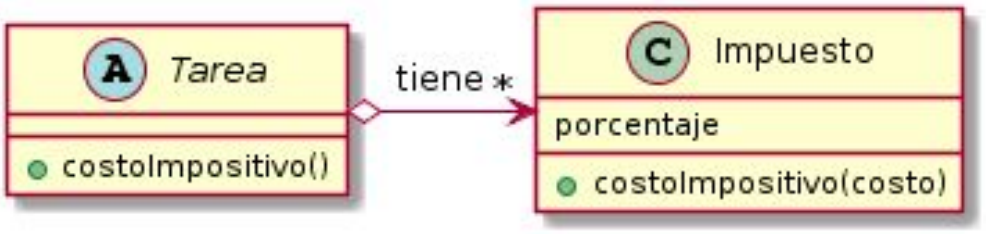
Por otra parte, esta solución no escala cuando la cantidad de impuestos empieza a crecer, si lo único que varía es el % que se aplica sobre el costo de la complejidad.

Opción 4: Tener una clase impuesto y manejarla con diferentes instancias

Si no hay comportamiento diferencial, podemos manejarnos con instancias a secas. Hay dos objetos: un objeto Impuesto que representa al impuesto A y otro objeto Impuesto que representa al impuesto B.

Entonces, cada tarea puede tener 0, 1 ó 2 objetos que van a pertenecer a la misma clase:






```
class Impuesto {
	const porcentaje

	method costo(costo) = costo * porcentaje / 100
}
```

La ventaja de esta solución: puedo tener una gran cantidad de impuestos y solo necesito instanciar cada uno de ellos. La desventaja: podría perder la abstracción “impuesto A” e “impuesto B” que dentro de la solución podía ser importante, aunque de todas maneras podria definirlas como constantes globales:

```
const impuestoA = new Impuesto(porcentaje = 3)
const impuestoB = new Impuesto(porcentaje = 5)
```

Ya sabemos al menos que tenemos una pregunta para disparar al usuario: ¿cuántos impuestos surgieron en los últimos años? ¿hubo impuestos basados en otro cálculo? ¿siempre hubo dos impuestos A y B? Esa tarea de relevamiento permitirá que estemos más seguros de nuestra decisión, mientras tanto nos inclinamos por la cuarta opción, ya que es un modelo simple y no nos interesa por el momento identificar los impuestos A o B (no se pretende en el apunte abordar la carga de los impuestos de una tarea).

Ahora sí, para calcular el costo impositivo de una tarea hacemos:

```
class Tarea {
	const impuestos = #{}
```

```
	method costoImpositivo() = impuestos.sum { impuesto =>
		impuesto.costo(self.costoComplejidad())
	}
```

Claro, eso fuerza a que por cada impuesto estamos calculando el costo de la complejidad. Podemos evitar eso mediante una variable local:

```
method costoImpositivo() {
	const costoBase = self.costoComplejidad()
	return impuestos.sum { impuesto =>
		impuesto.costo(costoBase)
	}
}
```

5.7 Costo total de un proyecto

El costo total de un proyecto es la sumatoria de los costos de todas las tareas asociadas.

```
class Proyecto {
	const tareas = []

	method costoTotal() = tareas.sum {
		tarea => tarea.costoTotal()
	}
}
```

En la tarea simple el costo total es el costo a secas:

```
>>TareaSimple
method costoTotal() = self.costo()
```

Y en las tareas compuestas es el costo de dicha tarea más la sumatoria de los costos de cada subtarea:

```
>>TareaCompuesta
method costoTotal() = self.costo() + subtareas.sum {
	tarea => tarea.costoTotal()
}
```

Vemos que la codificación del método para el proyecto y la tarea compuesta es el mismo. Ojo con cambiar **todo** el diseño porque hay en un lugar código que es "parecido" pero no "igual" (porque representan distintos conceptos).

Podríamos tener una tarea raíz en proyecto y que para calcular el costo total se delegue directamente a la tarea raíz:

```
class Proyecto {
	var tareaRaiz

	method costoTotal() = tareaRaiz.costoTotal()
}
```

Cada vez que planteamos un diseño, existe el riesgo de caer en el **sobrediseño**, hacer cosas de más que luego nunca se utilizan (el problema es que esto tiene un costo; además la aplicación se vuelve más compleja y por lo tanto más difícil de mantener). La idea de trabajar en forma **iterativa**, donde vamos explorando ideas de diseño y tirando código permite comenzar con un problema de baja complejidad e incrementarla a medida que uno gana confianza.

5.8 Complejidad de una tarea

La idea de separar a la tarea de su complejidad en otro objeto resulta interesante para remarcar algunas cosas:

- en la mayoría de los lenguajes OO la herencia ofrece una sola chance para categorizar, y ya decidimos que era más importante para nosotros distinguir a las tareas simples de las compuestas
- si elijo tener un `int` (0 para tareas de mínima complejidad, 1 para tareas de complejidad media y 2 para tareas de complejidad máxima), voy a tener un if cuando calcule el costo de la complejidad de una tarea:

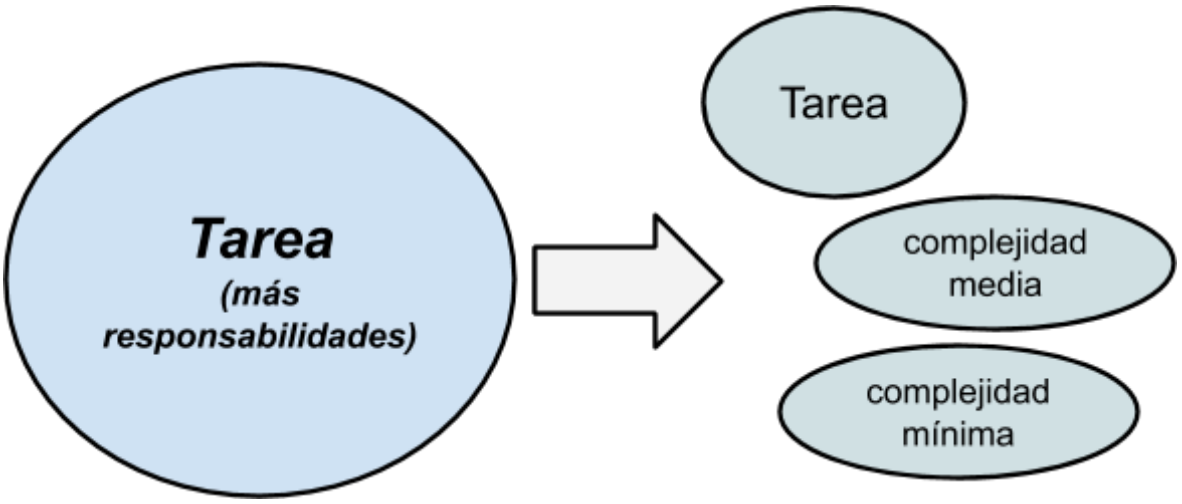
```
method costoComplejidad() {
	if (complejidad == 1) { // minima
		...
	}
	if (complejidad == 2) { // media
		...
	}
	if (complejidad == 3) { // maxima
		...
	}
}
```

Cuando necesite conocer los días máximos de atraso de un proyecto, voy a enviar el mensaje diasMaximosDeAtraso() al proyecto, que a su vez va a sumarizar los días máximos de atraso de una tarea. Una vez más, tengo que

poner un if múltiple:

```
method diasMaximosDeAtraso() {
	if (complejidad == 1) { // minima
		...
	}
	if (complejidad == 2) { // media
		...
	}
	if (complejidad == 3) { // maxima
		...
	}
}
```

Lo más criticable de esta solución es que tengo dos lugares donde estoy tomando la misma decisión. No solo molesta si aparece la complejidad crítica y hay que agregar un if más y recompilar, molesta sobre todo que estoy escribiendo 1, 2 y 3 en dos lugares distintos que tienen que estar sincronizados. Y además tengo que cuidarme de no exponer la complejidad de la tarea hacia afuera, para que esos números mágicos 1, 2 y 3 no estén diseminados por toda la aplicación. Entonces esa es la otra ventaja de separar tarea y complejidad:




```
>>Tarea
method costoComplejidad() = complejidad.costo(self)
```

Discusiones de diseño que nos interesa tener: la complejidad, ¿debería ser clase abstracta o interface? ¿De qué depende?

- Si hay comportamiento en común (como es el caso, se repite la idea tiempo * $ 25), deberíamos utilizar una clase abstracta.
- Cuando no hay comportamiento común, una interfaz que defina los mensajes que deben entender las clases que la implementen alcanza.

El método costo() en cada subclase de Complejidad, le debe pasar ¿una tarea o sólo el tiempo? Hay argumentos a favor de uno y otro:

```
method costoComplejidad() = complejidad.costo(self)
```

al pasar la tarea, la complejidad puede enviar más mensajes que tarea.tiempo(). A priori esto parece generar un acoplamiento mayor. Por otra parte hoy el costo de la complejidad depende solo del tiempo de la tarea: si los requerimientos cambian y necesitamos alguna información extra, eso no implicará un cambio en la interfaz del método costo()

```
method costoComplejidad() = complejidad.costo(tiempo)
```

al pasar la referencia *tiempo* a la complejidad, la Tarea sabe que el costo de la complejidad se determina en base a su tiempo, de manera que hay un grado de acoplamiento o conocimiento de la Tarea respecto a lo que hace la complejidad. Además si los requerimientos cambian y para calcular el costo se necesita más información de la tarea, eso requerirá cambiar la interfaz del método costo(tiempo). Por otra parte, la complejidad no recibe una referencia a la tarea, esto evita mandarle mensajes adicionales (en alguna medida baja el acoplamiento).

5.9 Resolviendo el costo de la complejidad

- El costo de la complejidad mínima se calcula como tiempo * 25.
- El costo de la complejidad media se calcula como tiempo * 25 + 5%.
- El costo de la complejidad máxima se calcula como tiempo * 25 + 7% + $ 10 a partir del décimo día.

En todos los casos se repite la misma idea: tiempo * 25, entonces podemos modelar eso en un único lugar: una superclase Complejidad. Por otra parte si pensamos que en la complejidad mínima

tiempo * 25 es igual a tiempo * 25 + 0, podemos nuevamente tener un **Template method**. La superclase Complejidad define el comportamiento base, y cada subclase implementa el valor adicional que hay que sumarle al costo final, que depende en algunos casos del costo base y en otros del tiempo:

```
class Complejidad {

  method costo(tarea) {
    const costoBase = tarea.tiempo() * 25
    return costoBase +self.costoExtra(tarea.tiempo(), costoBase)
```

```
    }

  method costoExtra(tiempo, costoBase)
}
```

En este caso tenemos:

- *costo(tarea)* como **template method**, el método definido en la superclase
- *costoExtra(tiempo, costoBase)*: el **método hook** o **primitiva** que están obligados a definir las subclases de Complejidad

La complejidad mínima define un método *hook* de compromiso (que nuevamente, podría haber definido la superclase Complejidad):

```
class ComplejidadMinima inherits Complejidad {
      override method costoExtra(tiempo, costoBase) = 0
}
```

La complejidad media considera el 5% del costo base:

```
class ComplejidadMedia inherits Complejidad {
      override method costoExtra(tiempo, costoBase) = 0.05 *
costoBase
}
```

Y el cálculo de la complejidad máxima es el más molesto, se puede resolver considerando 7% del costo base, y 10 $ a partir del décimo día, lo que es lo mismo decir 10 $ * (el máximo entre 0 y la cantidad de días que pasaron a partir del décimo día):

```
class ComplejidadMaxima inherits Complejidad {
      override method costoExtra(tiempo, costoBase) =
            costoBase * 0.07 +
            self.costoAdicionalPorRetraso(tiempo)

      method costoAdicionalPorRetraso(tiempo) =
            10 * 0.max(tiempo - 10)
}
```

Otra forma de resolverlo:

```
class ComplejidadMaxima inherits Complejidad {
      method costoExtra(tiempo, costoBase) =
            costoBase * 0.07 +
            self.costoAdicionalPorRetraso(tiempo)

      method costoAdicionalPorRetraso(tiempo) =
```

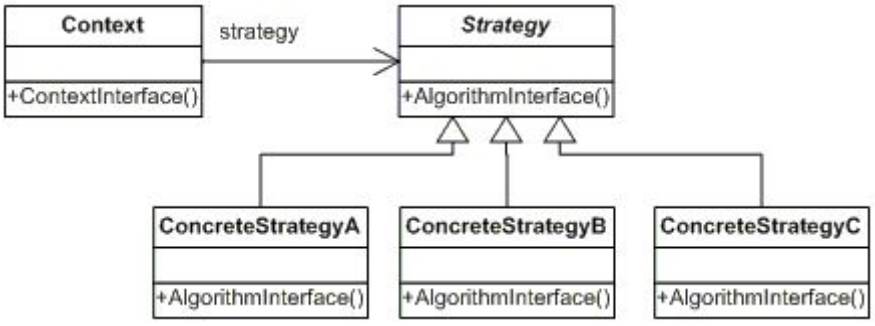
```
            if (tiempo <= 10)
                  0
            else
                  10 * (tiempo - 10)
}
```

6 Design Patterns que aparecieron en la solución

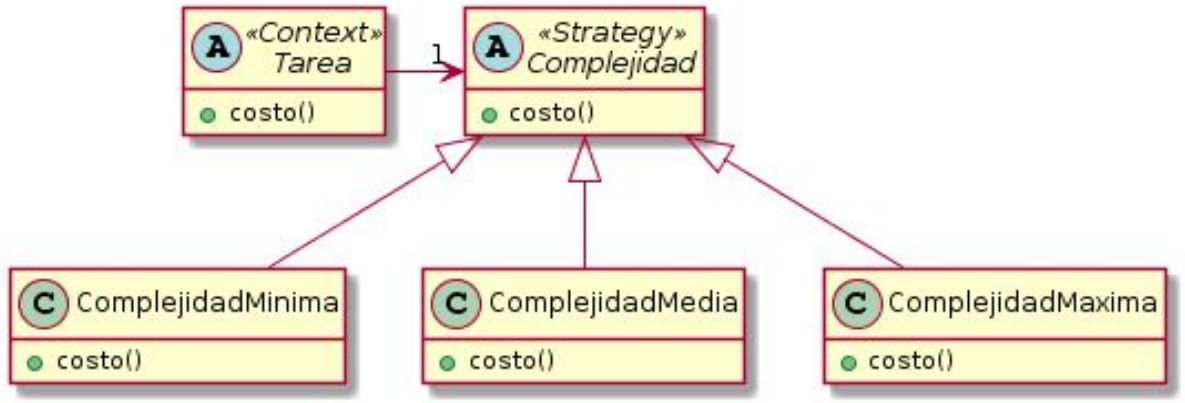
Además del patrón Template Method, explicamos otras ideas de diseño que surgieron en el ejercicio:

6.1 Strategy

Para separar la complejidad de las tareas: cada algoritmo que calcula la complejidad se encapsula en un objeto.



(patrón Strategy del libro *Design Patterns*, de Erich Gamma et al.)



Ventajas

- *Respecto a tener el comportamiento en Tarea y distinguir con un if la complejidad*: encapsulo el algoritmo que calcula el costo y los días

máximos de atraso en varios objetos polimórficos. Así hay mayor división de responsabilidades entre la tarea y la complejidad.

- *Respecto a subclasificar la complejidad de la Tarea:*
  - El Strategy permite encontrar abstracciones nuevas (la complejidad) que antes estaban incorporadas dentro de una abstracción mayor (la tarea).
  - mientras que la herencia es estática (me obliga a cambiar de objeto), el cambio de comportamiento que ofrece el Strategy es dinámico (la tarea sigue siendo la misma, conserva la identidad).

Consecuencias

- Hay más objetos (antes había una tarea, ahora hay tarea + complejidad).
- La tarea debe conocer a la complejidad, la complejidad puede o no conocer a la tarea (el que implementa el Strategy debe definir si la complejidad tendrá una variable tarea, si la recibirá como parámetro cuando lo necesite o si sólo recibirá el tiempo).

Desventajas con respecto a la subclasificación/manejo con ifs de la complejidad

- La construcción de una tarea es más compleja
- Se ve distribuido el comportamiento del objeto en otro objeto y no dentro de él mismo, esto parece a simple vista algo no natural.

Ejemplo asociado: una colección que

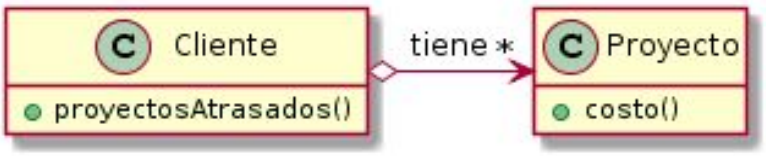
- almacena los elementos de una determinada manera (Set, List, Map)
- y puede ordenarse con diferentes algoritmos de ordenamiento: Burbujeo, QuickSort, MergeSort, etc. Cada algoritmo representa un *strategy*.

6.2 Composite

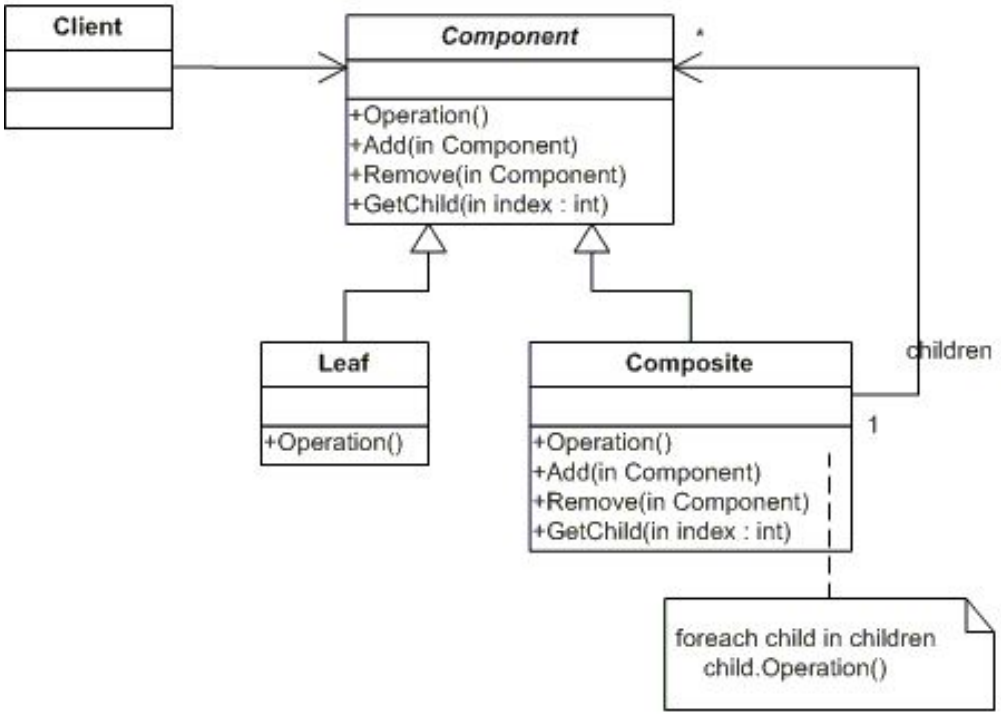
Para trabajar una estructura jerárquica en donde ramas y hojas operan en forma polimórfica. *Ejemplo asociado:* los directorios y los archivos en el Explorador del Sistema operativo:

- yo los quiero trabajar en forma polimórfica (con ambos puedo hacer botón derecho y abrir, copiar o eliminar)
- un archivo no se transforma en directorio y un directorio no se transforma en archivo

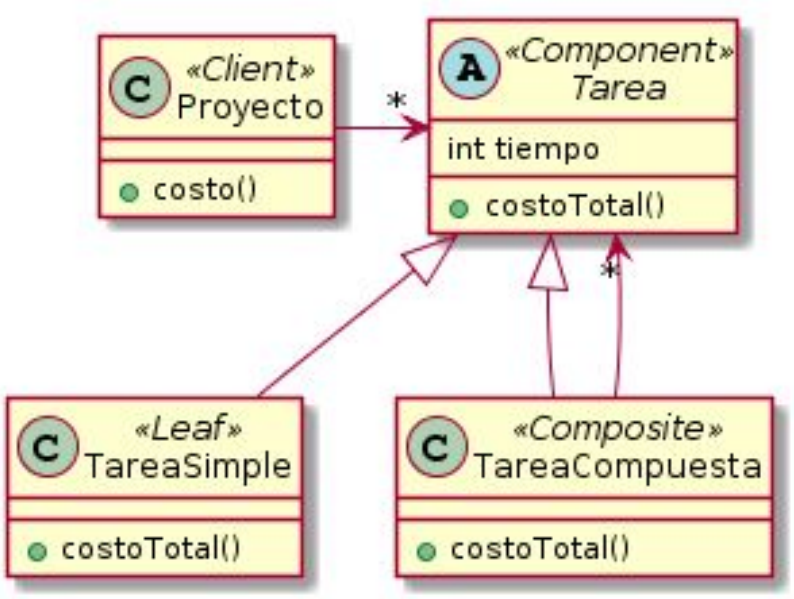
Ojo, composite pattern y composición son dos términos que describen situaciones diferentes. Un cliente tiene proyectos, una tarea tiene recursos, etc. En ese caso estamos hablando de composición de objetos (cuando un objeto tiene una relación de asociación con otro/s):



En el Composite pattern lo que nos interesa remarcar es que hay un cliente opera con componentes polimórficos (al cliente no le interesa distinguir si son simples o compuestos).



Para ser polimórficos, los componentes simples y compuestos comparten una interfaz común. Aplicado al ejercicio:



7 La solución

Se encuentra en el repositorio https://github.com/wollok/manejoProyectos

8 Resumen

A lo largo del apunte hemos estudiado a partir de un ejemplo decisiones no triviales de diseño, en la que la composición surge como una forma de generar jerarquías de objetos polimórficos para implementar comportamientos específicos. La búsqueda de alternativas requiere un análisis objetivo de las ventajas y debilidades de cada solución; también nos sirve para reforzar y retroalimentar el análisis disparando más preguntas al usuario. Por último, este proceso de iteraciones de pensar y bajar a tierra nuestro modelo permite que nuestras decisiones de diseño puedan cambiar y escalar.