Ejercicio práctico
El proceso de Diseño

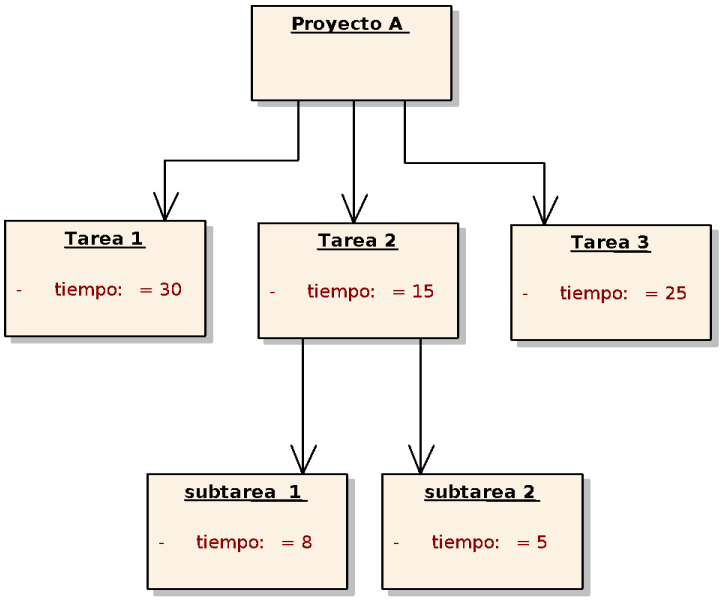
Versión 2.1
Febrero 2012

Indice

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%.

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.

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.

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 tiempo total de una tarea y sus subtareas
- Obtener el costo total de un proyecto
- Obtener los días máximos de atraso de un proyecto

Qué es diseñar y objetivo de la clase

En esta clase no nos importa el resultado del diseño, sino cómo llego a tomar decisiones (el proceso de decidir).

Preguntamos a la clase qué es diseñar, salen respuestas:

- es tomar decisiones/pensar
- "pensar antes una solución". Está bien distinguir dos momentos: pensar vs. hacer. Algunas metodologías proponen pensar **todo** el diseño antes de programar, nosotros (en la materia) muchas veces vamos a validar nuestro diseño con un poco de código y muchas veces vamos a postergar determinadas decisiones hasta el momento oportuno [1]. Una buena metodología "es la que me lleva a tomar las decisiones en el momento en que las tengo que tomar".
- es modelar una solución
- "que sea linda" (lo vamos a ver más adelante en la cursada, en la clase “Cualidades del diseño”)

Qué **no** es diseñar:

- no es hacer dibujos y cajitas
- tirar código solamente (lo que se conoce vulgarmente como “code & fix” o “cowboy coding”)
- saber diseñar no es saber UML, en todo caso saber UML me permite comunicar el diseño

Diseñar orientado a objetos es:

- encontrar los componentes (cuando bajamos bien a detalle tenemos objetos, pero también podemos pensar en un conjunto de objetos como un componente de la aplicación)
- qué hacen esos componentes
- cómo se relacionan

Puedo o no tener documentación formal, eso no excluye **diseñar.**

Trabajar en forma iterativa me da una ventaja: no estoy obligado a tomar todas las decisiones, algunas las tomo y otras las postergo.
¿Por qué las postergo?

- No siempre tengo toda la información
- El usuario y el desarrollador van aprendiendo sobre el dominio mientras se construye la aplicación
- Los requerimientos cambian (las retenciones fijas cambian a móviles, las fórmulas cambian, las leyes se modifican)

Hace tiempo las decisiones eran sencillas y la programación era compleja (10% de diseño y 90% de programación). Hoy no es un problema la programación, lo complejo es el problema en sí.

Manejo de proyectos

Les damos 10/15 minutos para leer el enunciado.

[1] Las metodologías de desarrollo iterativo proponen intercalar momentos de análisis, diseño y programación n veces a lo largo del proyecto

Encontrando los componentes

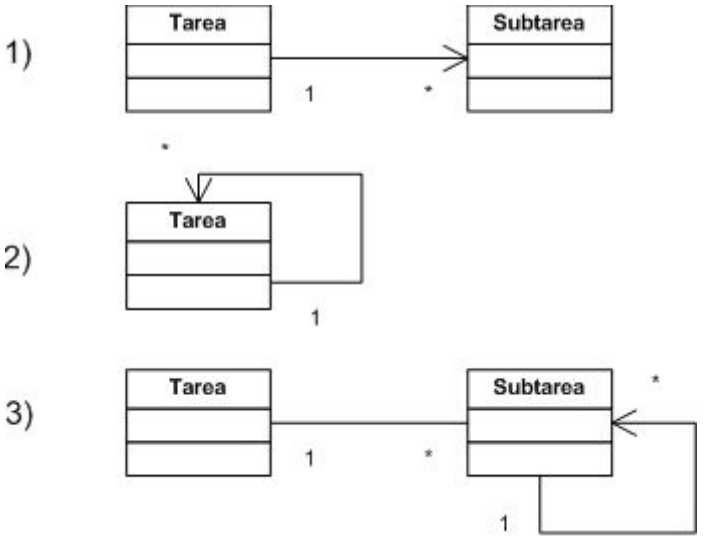
Algunas herramientas que podemos usar:

- Casos de uso / User stories / Encontrar los requerimientos del sistema
- Definir componentes (incluye identificarlos)
    - Diagrama de clases
    - Diagrama de objetos
    - CRC cards (clases, responsabilidades y colaboraciones)
    - Encontrar Objetos **candidatos** (no todos van a quedar) y responsabilidades -- vamos por acá

Tiramos objetos posibles:

- Proyecto
- Tarea
- Complejidad
- Impuesto
- Subtarea

Cómo represento a las tareas y las subtareas:



La opción 1) se descarta porque no permite que una subtarea tenga subtareas.
¿Cuál es más lindo? Depende de las responsabilidades de uno y otro objeto.

Responsabilidades de las tareas

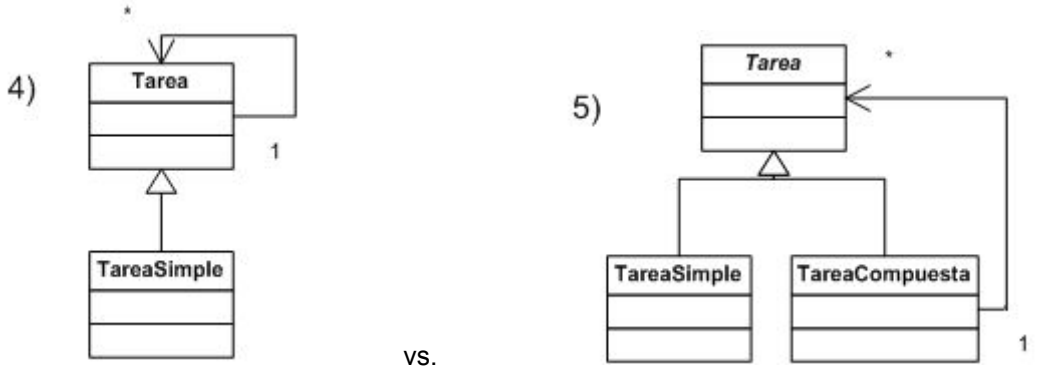
- saben decir su costo
- saben decir su completitud
- las que no tienen subtareas, les puedo setear el % de completitud
- saben su tiempo

Cuando no termino de decidirme por una opción u otra, quizás sirva tirar código.
Cuando me trabo...

cambio la perspectiva del problema:

1) en este caso, tiro código (en otra situación podría ser pensar en el diagrama de clases o el diagrama de objetos)
2) pregunto al que entiende del negocio, o
3) no tomo este camino y postergo la decisión.

El código me ayuda a ver cómo implementar una idea. A veces pareciera que no me sirve de mucho, pero cuando volvemos a pararnos sobre el problema aprovechamos el cambio de perspectiva anterior. Entonces por ahí veo algo que antes se me escapaba, como separar las tareas simples y las compuestas:



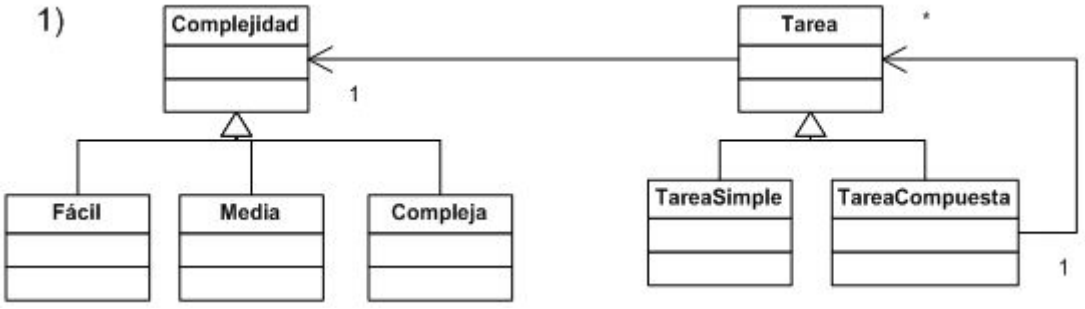
Podríamos elegir cualquiera de las dos opciones, tomamos 5) sólo porque tenemos ganas.

¿Qué pasó acá al subclasificar la tarea? le puse un nombre mejor (encontré una abstracción).

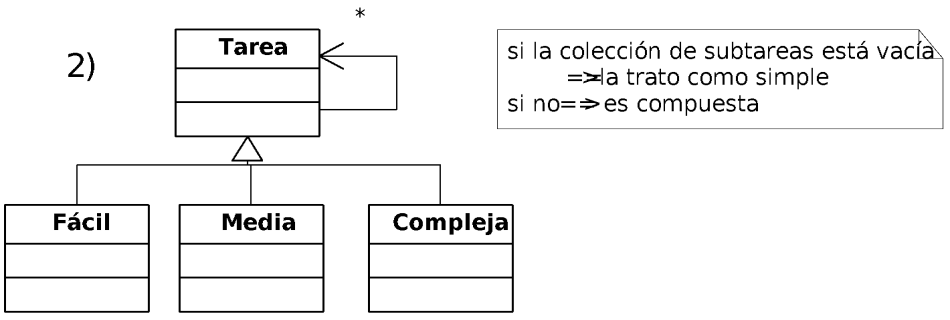
Separamos 3 tipos de clasificación posibles para tareas:

- Tarea y subtarea (si están directamente relacionadas con el proyecto)
- Tarea simple y compuesta
- Tarea fácil, media y compleja

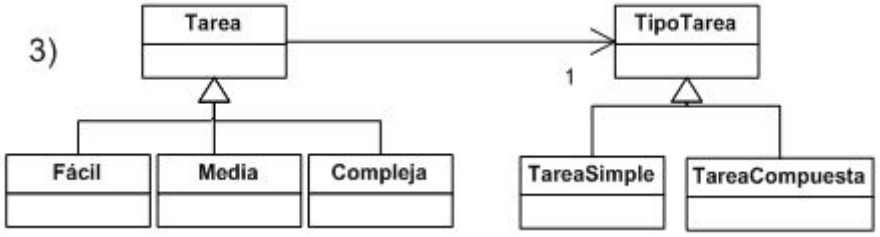
Vemos en qué se diferenciaría cada una.



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.



Según esta solución es costoso hacer que una tarea fácil devenga en compleja. Por otra parte si en muchos lugares se discrimina el comportamiento entre tareas simples y compuestas, habrá muchos métodos con un if. La Tarea debe contemplar ambos casos y pierde cohesión (sabe hacer muchas cosas diferentes).



Con esta solución es más fácil lograr que una tarea simple devenga en compuesta que cambiar la complejidad de una tarea.

¿Qué hacemos entonces?

No puedo decidir yo, pero vemos que 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 simples y compuestas 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- resolver costo extra
2- cálculo del costo
3- completitud del proyecto

Toda esta lista de tareas pendientes se suele llamar **Backlog** (o **TODO-List** o lista de pendientes en criollo).

¿Dónde anoto el backlog de tareas?

- En un archivo de texto plano
- En una planilla de cálculo
- Construyo un sistema propio para llevar la lista de pendientes
- Uso un **Issue Tracker** como Bugzilla (http://www.bugzilla.org/), Assembla (http://www.assembla.com/), GoogleCode, etc. Voy registrando tanto las cosas que me

faltan resolver como los bugs que van apareciendo (en un proyecto importante necesito administrar los bugs, las modificaciones al requerimiento original y las cosas que faltan)

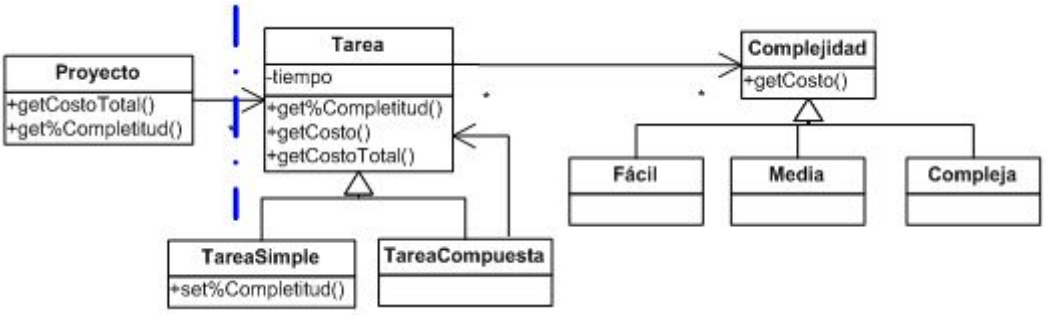
Todo esto depende de la magnitud del equipo de desarrollo.

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.getCosto()
- obtener el costo total de un proyecto ☾ a un proyecto.getCostoTotal()
- asignar % de completitud a tarea simple ☾ a una tarea simple.set%Completitud(x)
- obtener % de completitud de un proyecto ☾ a un proyecto.get%Completitud()
- obtener el tiempo total de un proyecto ☾ a un proyecto.getTiempoTotal()
- obtener los días máximos de atraso de un proyecto ☾ a un proyecto.getDiasMaximoDeAtraso()

Parece trivial, pero para saber el saldo de un cliente 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 no es igual al cliente físico (por eso es discutible cuando uno dice que objetos es el que más naturalmente encaja en la realidad).



Tenemos componentes claramente delimitados por la línea azul.
Acá no hablamos de componente a nivel objeto, sino que un componente agrupa a varios objetos que cumplen una funcionalidad (por ejemplo, del lado derecho de la línea azul, todos los objetos me ayudan a saber el costo de una tarea).

¿Para qué sirven las líneas azules? Para ver que si yo pienso mi solución sin cambiar la interfaz de Tarea, el Proyecto no debería verse afectado.
Y acá no hablamos de interfaz como *interface* de Java, sino como la parte visible/los servicios/lo que me puede contestar un objeto.

Ahora sí vamos al código para calcular el costo:
Una alternativa es que el costo se calcule como costo base + el costo de overhead (a definir en cada subclase). ¿Y el costo impositivo? Vamos a dejarlo stand-by por el momento en un método auxiliar aparte:

Costo de una tarea

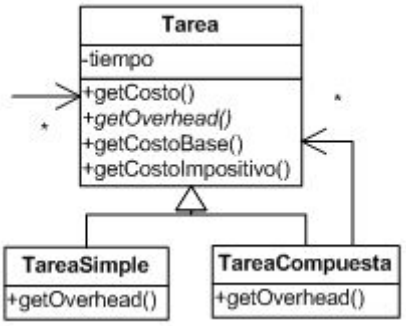
#Tarea

```
public double getCosto() {
   double costo = this.getCostoBase();
   return costo + this.getOverhead(costo) + this.getCostoImpositivo();
}

public double getCostoBase() {
   return this.complejidad.getCosto(this);
}
```

El método getOverhead(double) es abstracto en Tarea. En TareaSimple se redefine para que devuelva 0 y en Compuesta preguntando si las tareas son más de 3.

Esta idea es interesante para mostrar un pattern que se llama **Template Method**: se define un método (getCosto) que delega parte de la resolución en un método abstracto (getOverhead) redefinido para cada una de las subclases.



De todas maneras es sólo una implementación posible. Volvemos para atrás el cambio y dejamos:

#Tarea

```
public double getCosto() {
   double costo = this.complejidad.getCosto(this);
   return costo + this.getCostoImpositivo(costo);
}
```

#TareaCompuesta

```
public double getCosto() {
   double costo = super.costo();
   if (this.tareas.size() > 3) {
      costo = costo * 1.04;
   }
   return costo;
}
```

También se puede sacar afuera el if en un método aparte (ganamos en claridad y cohesión):

#TareaCompuesta

```
public double getCosto() {
   return super.costo() * this.getPorcentajeAjustePorSubtareas();
}

public double getPorcentajeAjustePorSubtareas() {
```

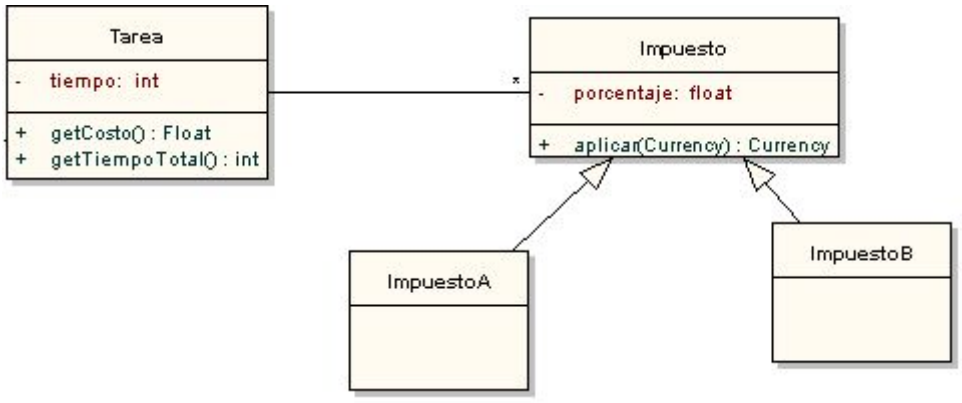
```
    if (this.tieneMuchasSubtareasAsociadas()) {
        return 1.04;
    } else {
        return 1;
    }
}

public boolean tieneMuchasSubtareasAsociadas() {
    return this.tareas.size() > 3;
}
```

La funcionalidad no cambia, pero el código queda más extensible. Tengo más métodos que tienen un objetivo bien definido, algo que está bueno si en más de un lugar quiero saber si una tarea “tiene muchas subtareas asociadas”.

Agregando los impuestos

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



Ahora, ¿en qué se diferencia cada impuesto en particular?
Y… en el impuesto A es un 3% y el impuesto B es un 5%.
Entonces sólo cambia el %, que podemos guardar en un atributo particular de cada impuesto.

¿Clases o 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:

Y para calcular el costo impositivo de una tarea hacemos:

#Tarea

```
public double getCostoImpositivo(double costo) {
	double costoImpositivo = 0;
	for (Impuesto impuesto : this.impuestos) {
		costoImpositivo += impuesto.aplicar(costo);
	}
	return costoImpositivo;
}
```

Costo total de un proyecto

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

#Proyecto

```
public double getCostoTotal() {
	double costo = this.getCosto();
	for (Tarea tarea : this.tareas) {
		costo += tarea.getCostoTotal();
	}
	return costo;
}
```

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

#TareaSimple

```
public double getCostoTotal() {
	return this.getCosto();
}
```

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

#TareaCompuesta

```
public double getCostoTotal() {
	double costo = this.getCosto();
	for (Tarea tarea : this.tareas) {
		costo += tarea.getCostoTotal();
	}
	return costo;
}
```

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:

#Proyecto

```
public double getCostoTotal() {
	return this.tareaRaiz.getCostoTotal();
}
```

Hablamos un poco del 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 va a favor de tener un diseño más simple y me ayuda a no caer en el sobrediseño.

Complejidad de una tarea

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

- 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 base de una tarea:

```
public double getCostoBase() {
	if (this.complejidad == 0) { // minima
		return …;
	}
	if (this.complejidad == 1) { // media
		return …;
	}
	if (this.complejidad == 2) { // máxima
		return …;
	}
}
```

Ahora, cuando necesite conocer los días máximos de atraso de un proyecto, voy a enviar el mensaje getDiasMaximosDeAtraso() 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:

```
public int getDiasMaximosDeAtraso () {
	if (this.complejidad == 0) { // minima
		return …;
	}
	if (this.complejidad == 1) { // media
		return …;
	}
	if (this.complejidad == 2) { // máxima
		return …;
	}
}
```

Lo más criticable de esta solución es que tengo dos lugares donde estoy tomando la misma decisión. No sólo molesta si aparece la complejidad crítica y hay que agregar un if más y recompilar, molesta sobre todo que estoy escribiendo 0, 1 y 2 en dos lugares distintos que tienen que estar sincronizados. Entonces esa es la otra ventaja de separar tarea y complejidad:

Faltaría implementar el

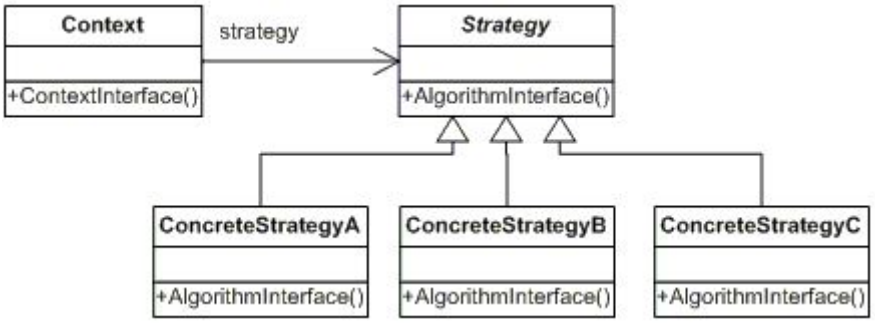
Strategy

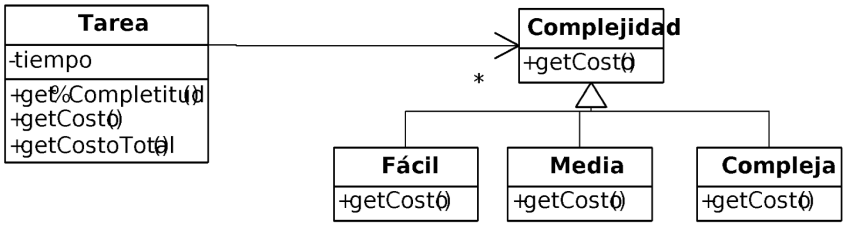
Design Patterns que aparecieron en la solución

Por último, explicamos los patterns que salieron en el ejercicio:

Strategy

Para separar la complejidad de las tareas: cada algoritmo que calcula la complejidad se encapsula en un objeto (aparecen objetos no triviales, que no son los típicos objetos de dominio: Cliente, Factura, etc). *Ejemplo asociado*: una colección que puede ordenarse con diferentes algoritmos de ordenamiento: Burbujeo, QuickSort, etc.





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).
    - Una consecuencia de encontrar nuevas abstracciones es que 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 (es tarea de quien implementa el Strategy definir si la complejidad tendrá una variable tarea o la recibirá como parámetro cuando lo necesite).

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

- La construcción de una tarea es más compleja

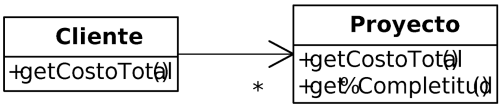
- Se ve distribuido el comportamiento del objeto en otro objeto y no dentro de él mismo, esto parece a simple vista algo no natural. De todas maneras el proyecto sigue delegando en la tarea, que es el que tiene que decir su costo.

Composite

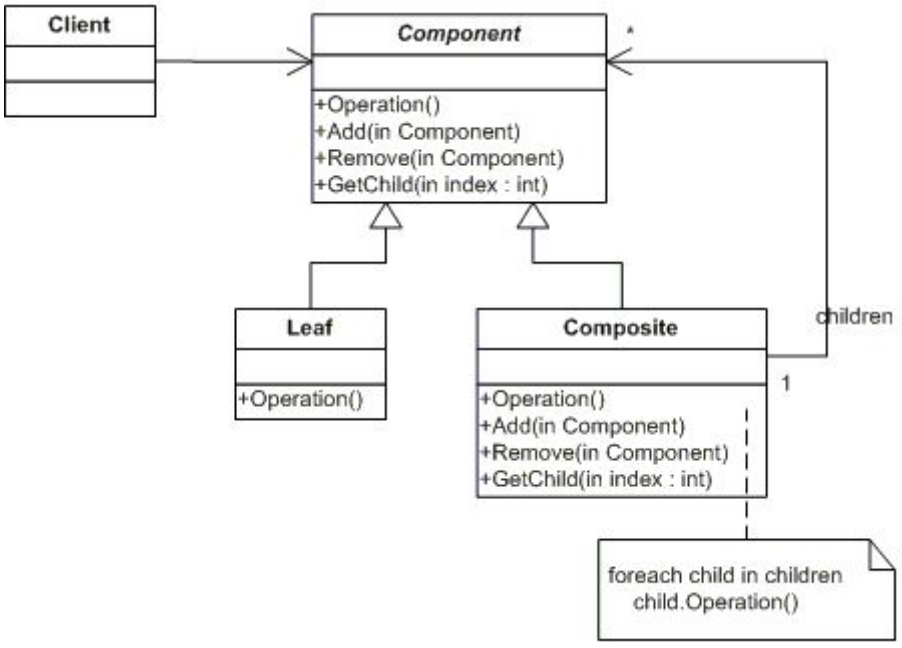
Para trabajar tareas simples y compuestas en forma polimórfica. *Ejemplo asociado*: los directorios y los archivos en el Explorador de Windows:

- yo los quiero trabajar en forma polimórfica (con ambos puedo hacer botón derecho y abrir)
- 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):

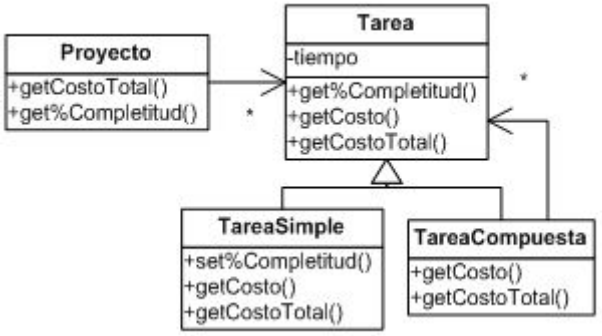


En el Composite pattern lo que nos interesa remarcar es que hay un cliente opera con componentes polimórficos (al cliente no le interesa distinguir si son simples o compuestos).



Para ser polimórficos, los componentes simples y compuestos tienen una interfaz común.

Aplicado al ejercicio:



Tarea

impuestos

…

Impuesto

%

3

5

Impuesto

%

Tarea (gorda)

Tarea

complejidad media

complejidad mínima