Trabajo Práctico I
Smalltalk

[7507/9502] Algoritmos y Programación III
Primer cuatrimestre de 2020

Alumno:	FONTELA, Joaquín
Número de padrón:	103924
Email:	jfontela@fi.uba.ar

Índice

1. Introducción

El presente informe reúne la documentación de la solución del primer trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar un aplicación llamada *AlgoFix*, **cuyo objetivo es administrar a los pintores registrados, para permitirle al usuario obtener aquel que ofrezca el menor presupuesto para el trabajo requerido** (se podrán especificar tipo de pintura y cantidad de metros cuadrados a pintar).

El código de la aplicación estará desarrollado en Pharo, organizado según el paradigma de la Programación Orientada a Objetos (POO).

2. Supuestos

Para el desarrollo de este trabajo, he tomado por supuestos algunos hechos que no estaban especificados en la consigna:

- Ante todo, mi primer supuesto fue que **no realizaria *tests* propios sobre la clase *AlgoFix***, ya que los mismos estan proporcionados por la catedra como tests a pasar para cumplir con la consigna.
- Luego, por un lado, como podemos ver en las pruebas dadas por la cátedra, a los pintores se los identifica por su nombre, sin embargo ya que en ningún lado se indica lo contrario, he decidido permitir que **se puedan registrar dos pintores con el mismo nombre**.
- Además, ya que ningun lado se elimina a ningún pintor de los registros, no he considerado necesario agregar una función que realice dicha acción, por más que aquello pueda sonar más lógico.
- Finalmente, **en los diagramas de clase indicare algunos métodos como protegidos o privados**, por más que en *Smalltalk*, todo ellos sean públicos, esto me sirve para precisar la utilidad de ese método.

3. Modelo de dominio

En cuanto al modelo que he decidido utilizar, se trata de uno con **una clase principal**, aquella que nos ha dado la cátedra: *AlgoFix*. Esta clase, en algún punto interactúa con todas las demás, aunque no siempre de manera directa.

***AlgoFix* se encarga de la interacción con el usuario**, ya que es la única a la cual se tiene acceso y se le puede pedir cosas. Todas las otras clases están implementadas para el funcionamiento interno del programa.

OptimizadorDePresupuesto: esta clase se encarga de **elegir el mejor presupuesto** de entre todos los pintores disponibles, y de entregarle lo elegido a *AlgoFix*.

Pintor: esta clase es la que **representa al pintor** propiamente dicho. Sin embargo, cabe aclarar que **es una clase abstracta**: no se puede instanciar ya

que todo pintor es de rodillo o de pincel. A pesar de ser la clase con mas metodos de todo el diagrama, realmente solo dos de ellos son públicos: el inicializador con parámetros y el obtener presupuesto. Esta última función es la principal de la clase: bajo ciertos parámetros definidos, **le pedimos el presupuesto a un pintor, y este nos lo devuelve**. Los demas metodos de la clase son, o bien abstractos, o bien solo usados por ella y sus clases hijas (protegidos).

PintorDeRodillo y *PintorDePincel*: estas dos clases heredan de la clase *Pintor* y definen todos los métodos abstractos declarados en ella. Se diferencian en los atributos de clase ***horasEnPintarUnMetroCuadrado*** y ***litrosDePinturaParaPintarUnM2***, ademas de que los pintores de pincel ofrecen un **descuento del 50%** en la mano de obra por trabajos de más de 40 metros cuadrados.

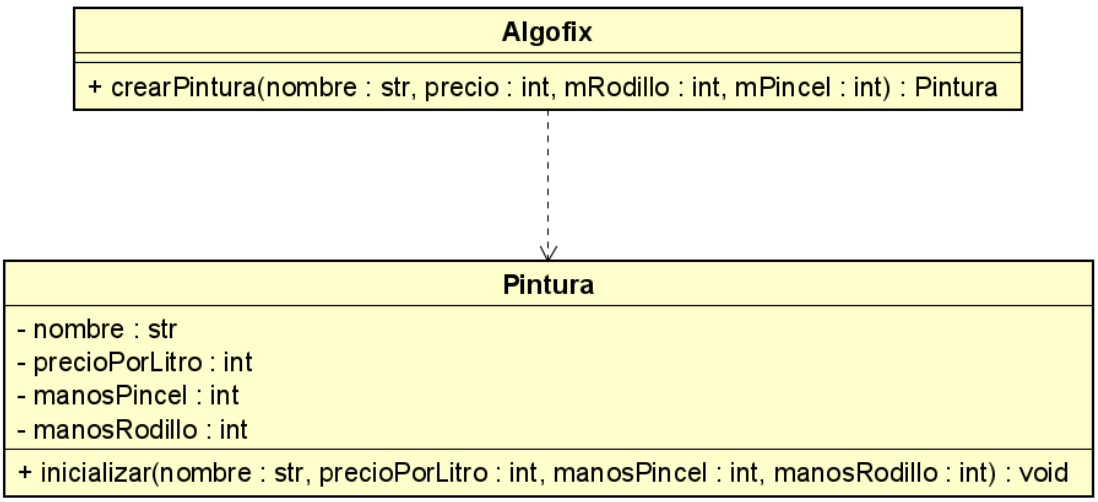
CalculadoraDePresupuesto: **es una calculadora con dos funciones**. *ManoDeObramasMateriales* que recibe estos dos valores, y devuelve su suma, y la segunda función (y principal), *obtenerPresupuesto*, que dados un precio de mano de obra, un precio de materiales y un responsable, **genera y devuelve el presupuesto** en cuestión.

Presupuesto: es una clase con dos atributos. Por un lado, *responsable* contiene el nombre del pintor responsable del trabajo de cuyo presupuesto hablamos. Por otro lado, *valor* es el valor total del presupuesto. Las instancias de esta clase son siempre generadas por *CalculadoraDePresupuesto*. Además, tiene **el método *tieneMenorValorQue* que compara los valores de dos presupuestos**.

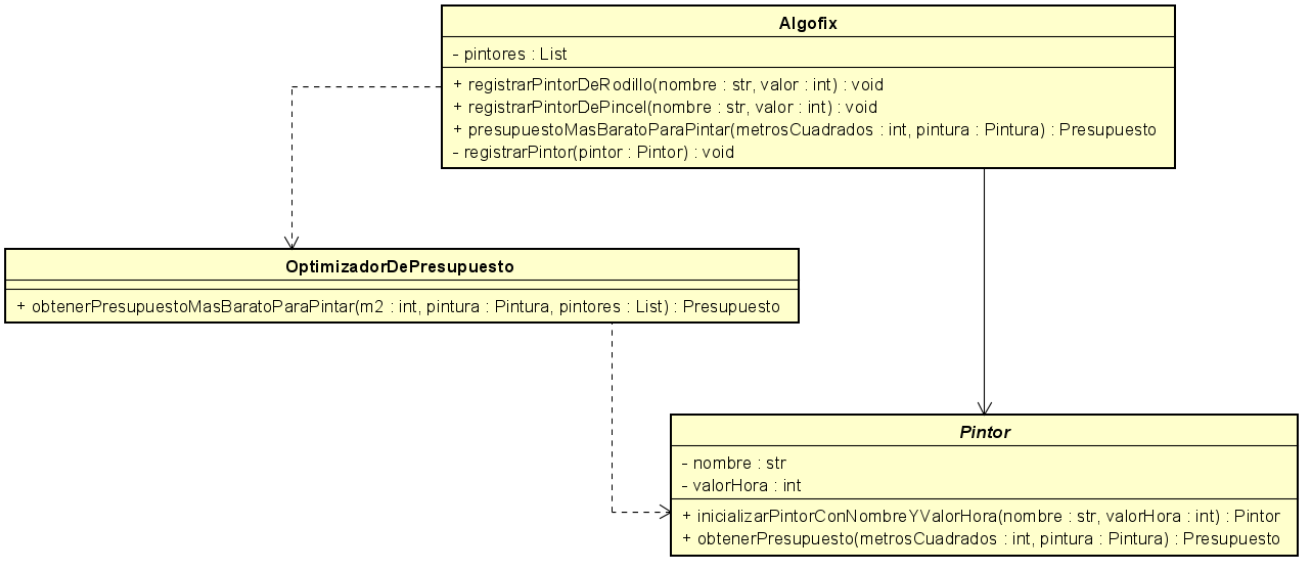
Pintura: la última clase que faltaba mencionar. Tiene como función guardar datos como el nombre, y las manos de pintura necesarias según si el trabajo se hace con pincel o con rodillo. Además, tiene un método que nos permite, **dada una cantidad de litros, obtener su precio**.

4. Diagramas de clase

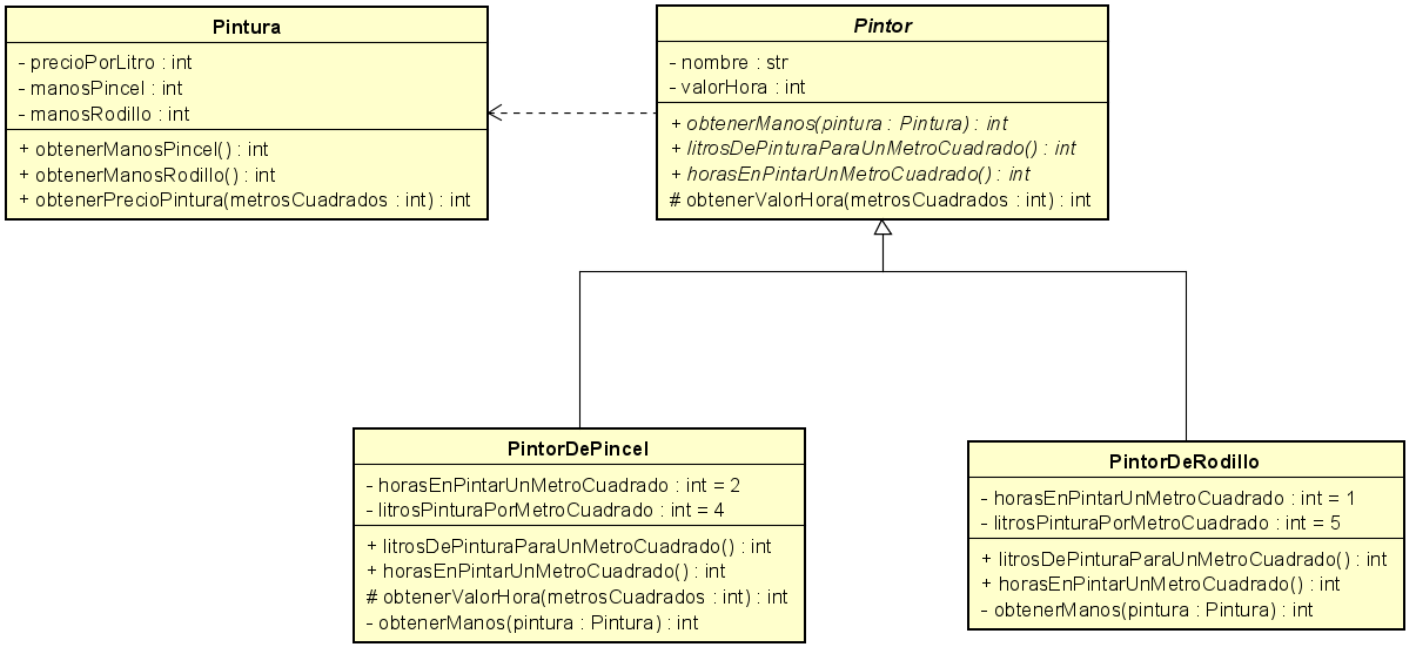
He preparado cuatro diagramas para graficar el diseño de la aplicación.



He expuesto el primero de ellos aquí arriba que muestra la relación entre la clase *AlgoFix* y la clase *Pintura*. Es una **relación de dependencia** ya que *AlgoFix* inicializa las instancias de la clase *Pintura*.

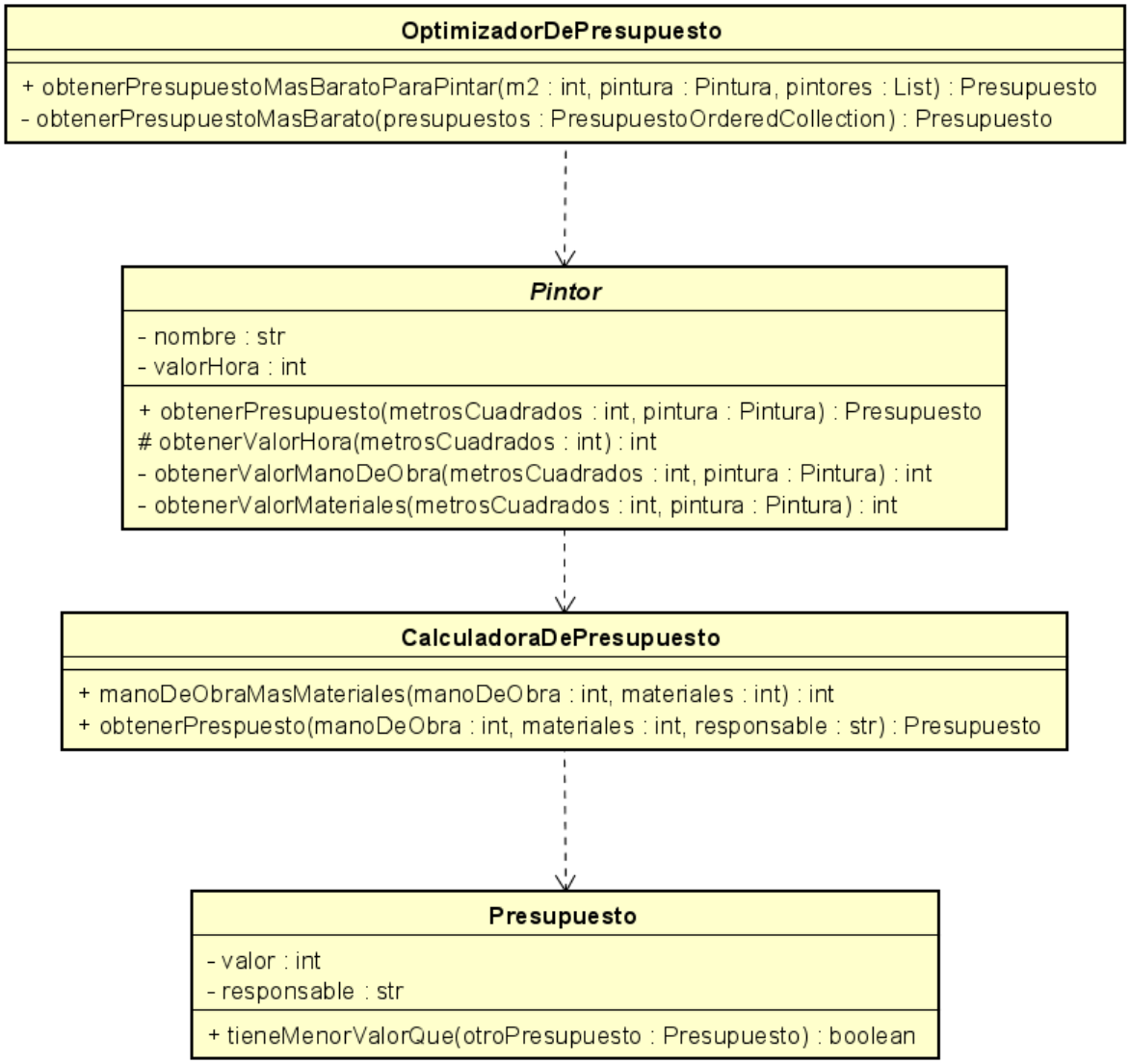


Este segundo diagrama muestra la relación entre las clases *AlgoFix*, *OptimizadorDePresupuesto* y *Pintor*. *AlgoFix está asociada a la clase Pintor* ya que la tiene como atributo, sin embargo aparece también dentro de los atributos de *AlgoFix*, ya que se quiere remarcar que no es un pintor sino una lista. Aquí podemos ver un pantallazo de como funciona internamente el **registro de pintores**, así como también el inicio del recorrido del código, para **obtener el menor presupuesto** entre todos los pintores.



El tercer diagrama expuesto aquí arriba muestra la **relación de herencia** entre las clases *Pintor* con *PintorDePincel y PintorDeRodillo*. Cumplen la **regla de "es un"** en este tipo de relaciones, ya que tanto un pintor de pincel como un pintor de rodillo, son pintores. Podemos ver como los tres métodos abstractos de *Pintor* están definidos en ambas clases hijas, asi como tambien podemos observar los atributos de clase *horasEnPintarUnMetroCuadrado* y *litrosPinturaPorMetroCuadrado*.

Por otro lado, podemos ver la relación de estos con la clase *Pintura*: según el tipo de pintor, al llamar *obtenerManos()*, se pedirá por un distinto atributo de *Pintura* (*manosPincel* o *manosRodillo*) estableciendo de esta manera una **función polimórfica**.



Finalmente, en este último diagrama podemos observar **la relación entre las clases que llevan a cabo el cálculo del mejor presupuesto**. *OptimizadorDePresupuesto* recibe una lista de pintores, y por cada uno obtiene su *Presupuesto* (a su vez cada uno de ellos acude a la *CalculadoraDePresupuesto*) para luego obtener el mejor valor (el más barato).

5. Detalles de Implementación

Cuando se nos entregaron las pruebas, empecé listando aquellos métodos que si o si tendria que implementar: *registrarPintorDePincel, registrarPintorDeRodillo*, *crearPintura…*, *prespuestoMasBarato…*,

manoDeObraMasMateriales, es decir, aquellas que figuraban explícitamente escritas en código.

De todas ellas, **la más compleja fue, por mucho, *presupuestoMasBarato*...**lógicamente, ya que de eso se trataba en gran parte el programa. Por esta razón, me gustaría ahondar aunque sea un poco en la implementación de esta parte de la aplicación.

Ante todo, decidí crear a los pintores con sus inicializadores (haciendo prueba unitaria de por medio antes de cada método, omitir este detalle de ahora en más ya que estas se pueden ver en el código). En cierto punto sabía que tendría que **obtener el presupuesto de los pintores**, entonces considere que lo mas adecuado seria **pedirles a ellos**, ya que, por un lado, es como funciona en la vida real, y además de esta manera respetamos el encapsulamiento.

Aca fue cuando decidí crear una *CalculadoraDePresupuestos*, que más allá de que realiza simples sumas luego de obtener un par de valores, me resulto importante que el pintor no realiza tantas tareas y por lo tanto, **delegarle la creación de las instancias de *Presupuesto* a otra clase.**

Una vez ya creado el método *obtenerPresupuesto…* de la clase *Pintor*, llegó el momento de que el *OptimizadorDePresupuestos* lo utilizara: y acá es donde me ahorro un buen código la función *collect* ya definida en *Smalltalk*. Simplemente *OptimizadorDePresupuestos* haría dos cosas: **obtener el presupuesto de cada uno de los pintores disponibles** y guardarlo en una lista (por medio de *collect*), y **luego devolver aquel presupuesto de la lista cuyo atributo *valor* fuera el menor** de todos.

6. Excepciones

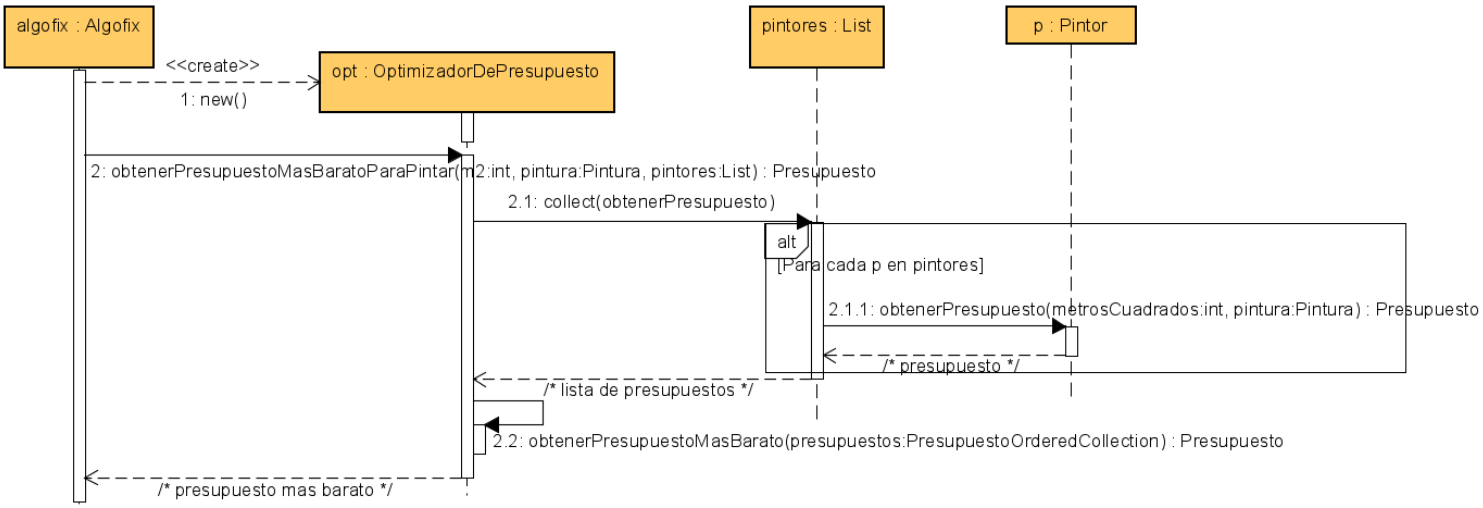
El proyecto consta además de dos tipos de excepciones:

- *ValorNegativoError*: este tipo de error es, por bastante diferencia, el más usado en el código. Básicamente se lanza cada vez que **se intenta utilizar de manera absurda un valor negativo**. Ejemplos de esto son precios, sueldos o superficies negativas. Para ejemplos en código, se pueden observar en la carpeta AlgoFixTestErrores, los test 01 al 07 y el 09.

- *SinPintoresError*: este otro tipo aparece una única vez en todo el código. A pesar de ello, creo que puede llegar a ejecutarse más veces que el arriba mencionado (y de allí lo puedo considerar más importante), ya que se lanza cuando **se intenta obtener el mejor presupuesto sin haber registrado pintores anteriormente**.

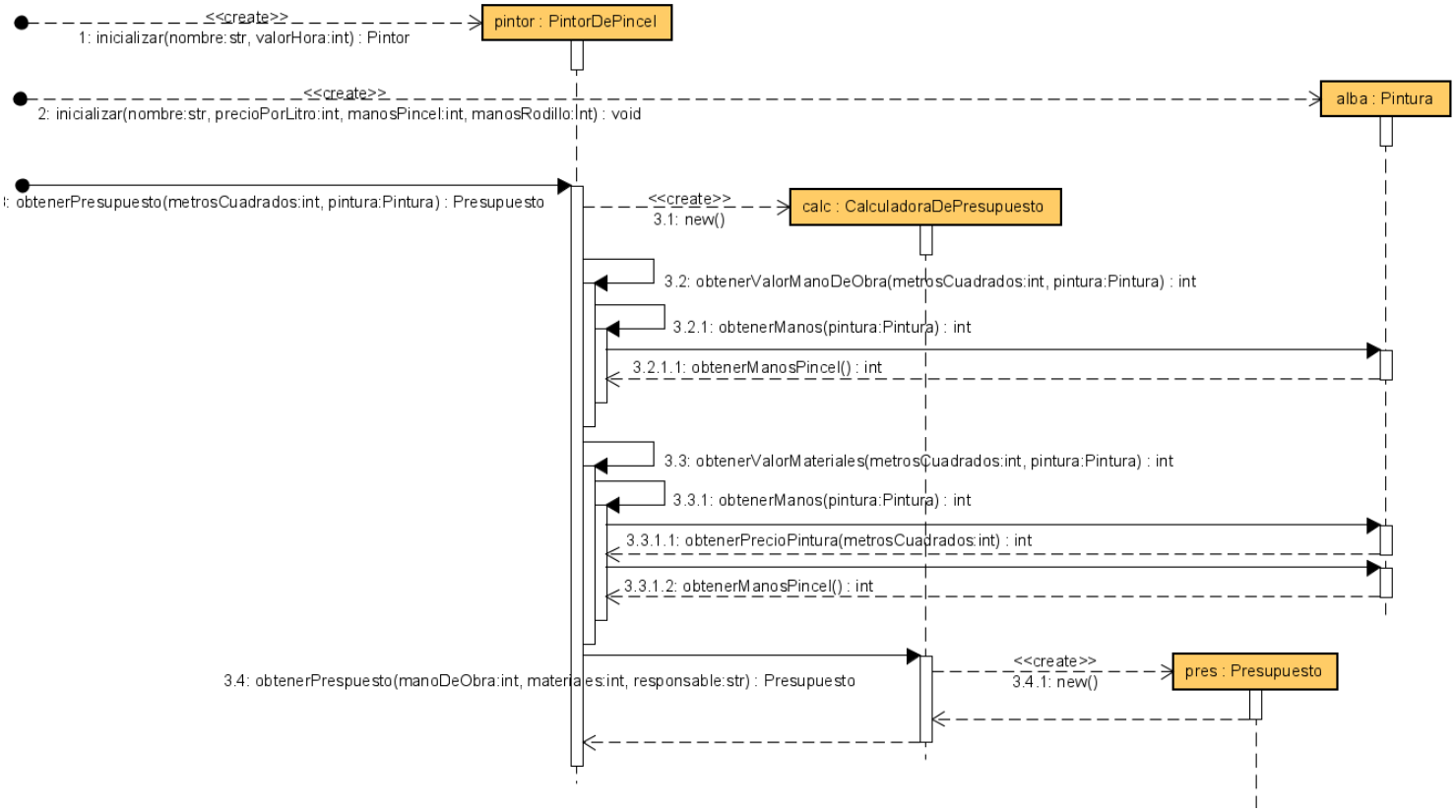
7. Diagramas de Secuencia

Para este proyecto he creado **dos diagramas de secuencia** que muestran el proceso que atraviesa el programa desde que se le pide obtener el presupuesto más barato, hasta que finalmente lo consigue.



En este primer diagrama podemos observar **el comportamiento más "superficial"**, desde que el usuario le pide a *AlgoFix* obtener el presupuesto más barato, **hasta el nivel del *Pintor***, en el cual podemos observar como *OptimizadorDePresupuesto* le pide el presupuesto a cada uno de ellos, y estos lo devuelven, pero sin saber cómo lo hacen.

Algoritmos y Programacion III



En este segundo diagrama nos adentramos **bien "profundo"** en el funcionamiento del programa. Podemos ver **como funciona la función *obtenerPresupuesto***, algo que nos había quedado como incógnita en el anterior diagrama.
Podemos ver que se obtiene por un lado el precio de la mano de obra, y por el otro el de los materiales, para después pedirle *CalculadoraDePresupuesto* el *Presupuesto* con los valores obtenidos.