Trabajo Práctico 1 — Smalltalk

[7507/9502] Algoritmos y Programación III
Curso 1
Primer cuatrimestre de 2020

Alumno:	Grassano, Bruno
Número de padrón:	103855
Email:	bgrassano@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 una aplicación de una agencia de pintores en Pharo utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso.

2. Supuestos

El trabajo se realizo con los siguientes supuestos que fueron surgiendo a lo largo de la realizacion del trabajo, ya que estos casos no están contenidos en las especificaciones entregadas.

* El usuario no puede crear un pintor que trabaje con rodillo y pincel a la vez, este trabajara con una sola herramienta. Si se ingresa con el mismo nombre se considera que es otra persona.
* El usuario sera el encargado de manejar las pinturas que haya creado.
* El pintor con rodillo no ofrece descuento.
* El área para el presupuesto tiene sentido lógico, no se pediría un presupuesto para 0 metros cuadrados.
* Cargar un precio de pintor o pintura como 0 es valido, se considera que se ofrece el servicio o producto gratuitamente.
* El descuento que ofrecen los pintores es cuando el área a pintar es mayor que 40 metros cuadrados estrictos.
* El cliente solamente quiere el presupuesto final, es decir, no le interesa como sean las proporciones de este con respecto a la mano de obra y materiales.

3. Diagramas de clase

Se ocultaron los métodos para dar claridad al diagrama al momento de leerlo.

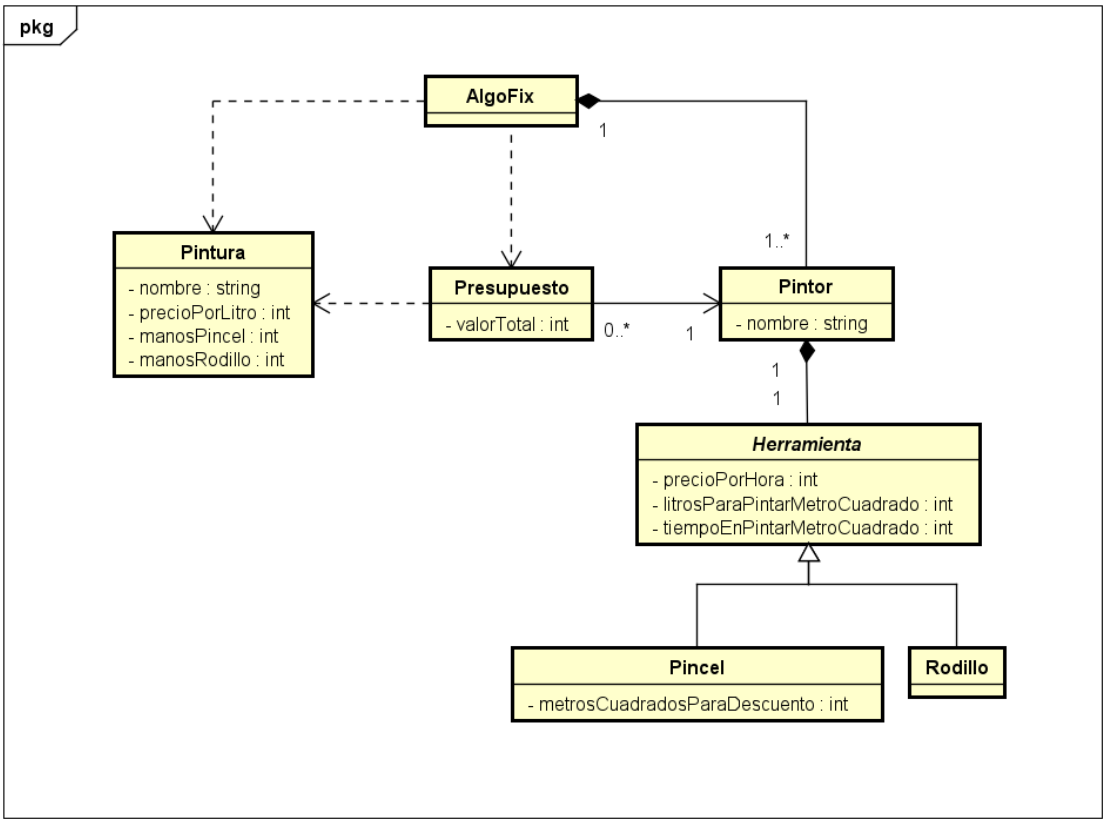

Figura 1: Diagrama general de la implementación.

Este diagrama de clase muestra de forma general la relación que tienen todas las clases del trabajo. Observar que la instancia de AlgoFix va a ser la responsable de realizar las diferentes opciones al ir delegándolas. Notar también que Algofix solamente instancia y utiliza las pinturas, al igual que los presupuestos.

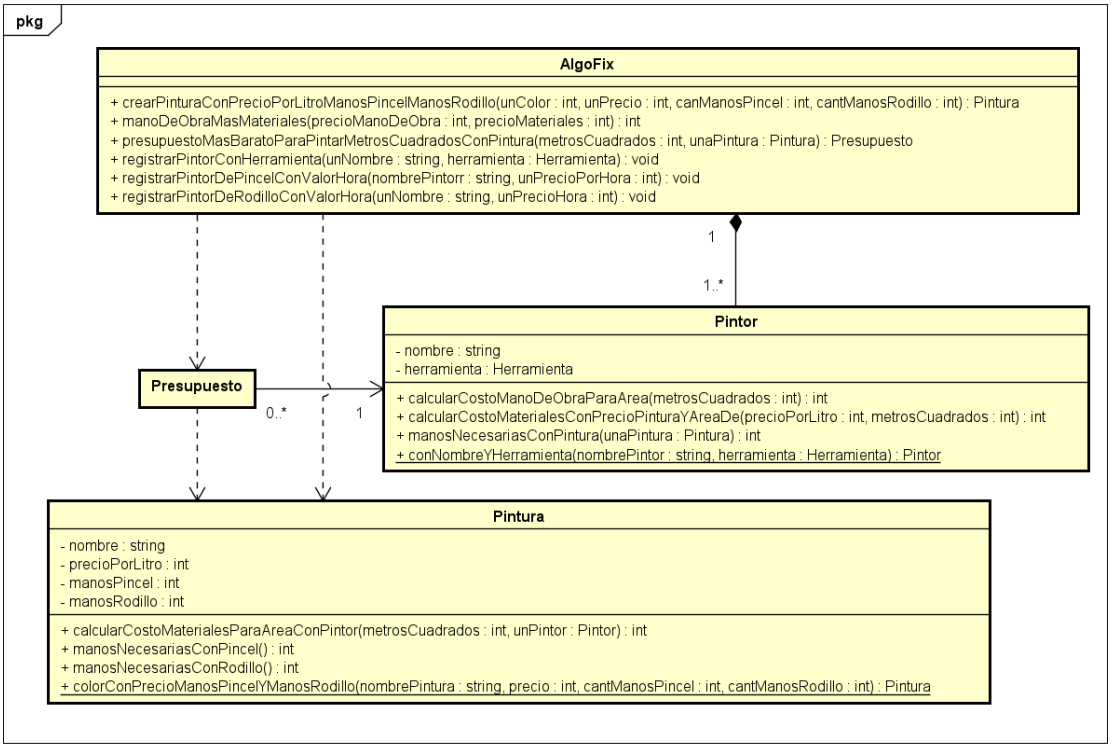

Figura 2: Diagrama mostrando las relaciones de AlgoFix.

En este diagrama se puede observar la relación que tiene AlgoFix con los pintores y las pinturas incluyendo los métodos. Presupuesto solamente se incluyo para indicar la relación que tiene.

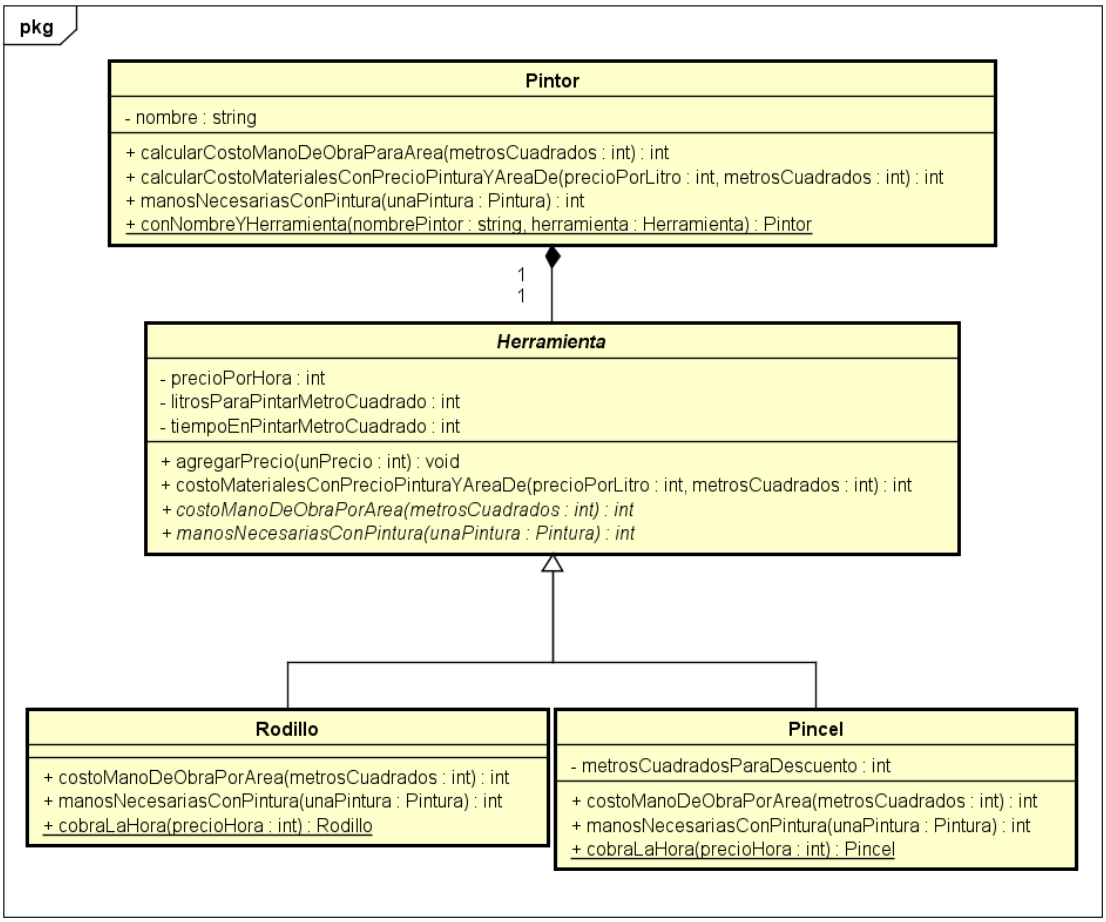

Figura 3: Diagrama mostrando en detalle las relaciones del pintor con la herramienta.

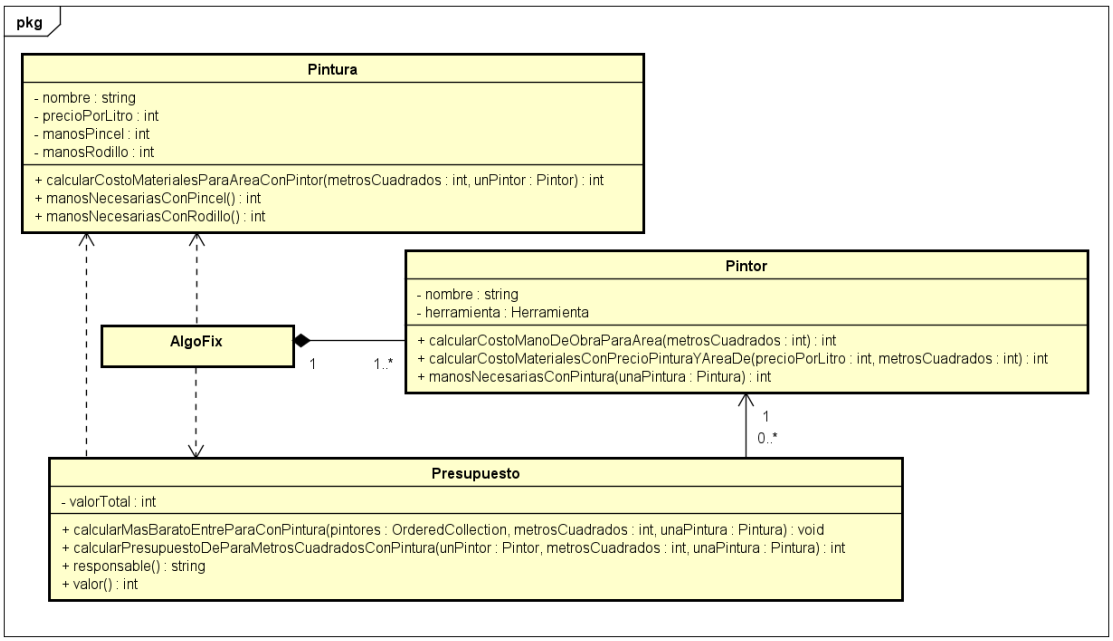

Figura 4: Diagrama mostrando Presupuesto con sus relaciones.

En este diagrama se ocultaron los detalles de AlgoFix junto con algunos métodos de Pintor y Pintura que no son relevantes al momento de realizar un presupuesto.

4. Detalles de implementación

4.1. AlgoFix

Esta clase es la responsable general del trabajo, a través de ella se deben de realizar las diferentes opciones disponibles. Internamente como datos solamente almacena una colección de pintores.

A continuación se detallaran algunos de los métodos mas importantes.

Método crearPintura:conPrecioPorLitro:manosPincel:manosRodillo:

Recibe los datos necesarios para crear una pintura. Este metodo le mandara esa información al constructor de Pintura.

Métodos de registro en AlgoFix

Estos métodos son registrarPintorDePincel:conValorHora:,registrarPintorDeRodillo:conValorHora:, y registrarPintor:conHerramienta:.

Se encargan de recibir los datos de los pintores con los cuales se va a llamar a los correspondientes constructores. Esto lo hacen primero llamando al constructor de una herramienta, que después es agregada al pintor. El método *registrarPintor:conHerramienta:* es el encargado de finalmente crear el pintor llamando al constructor de este y agregar el pintor a la colección de pintores que tiene AlgoFix. Este método es privado y solamente debería ser usado internamente de AlgoFix. Se muestran a continuación dos de los métodos mencionados.

```
registrarPintorDePincel: nombrePintor conValorHora: unPrecioPorHora
    self registrarPintor: nombrePintor conHerramienta: (Pincel cobraLaHora: unPrecioPorHora)

registrarPintor: unNombre conHerramienta: herramienta
    pintores add: (Pintor conNombre: unNombre yHerramienta: herramienta )
```

Metodo presupuestoMasBaratoParaPintarMetrosCuadrados:conPintura:

Este método es el que utiliza el usuario para pedir el presupuesto mas barato con la pintura que el indique.

```
presupuestoMasBaratoParaPintarMetrosCuadrados: metrosCuadrados conPintura: unaPintura
    | presupuesto |
    presupuesto := Presupuesto new.
    presupuesto calcularMasBaratoEntre: pintores para: metrosCuadrados conPintura: unaPintura.
    ^presupuesto
```

4.2. Pintor

Esta clase almacena los datos del pintor y recibe las instrucciones que vienen de AlgoFix o Presupuesto.

Como variables de instancia posee el nombre y una herramienta. La elección de utilizar una herramienta para abstraer parte de su comportamiento se debe a que facilita y vuelve mas claras las operaciones correspondientes a los cálculos. Una ventaja adicional de esto, es que si se quisiera se podrían implementar varias herramientas distintas, o incluso crear un pintor que pueda trabajar con rodillo y pincel a la vez. Lo único necesario para que esto suceda es que esas herramientas entiendan los mismos mensajes. En su apartado se explicara con mas detalle esta decisión de usar herencia con polimorfismo

A continuación se muestran los métodos correspondientes a esta clase.

Métodos obtenerNombre,agregarNombre:, y agregarHerramienta:

Estos métodos son utilizados durante la creación de un pintor, o durante el pedido de información relevante para el cliente, este es el caso del nombre. Estos métodos son privados y no deberían ser usados desde afuera.

Métodos calcularCostoManoDeObraParaArea: y calcularCostoMaterialesConPrecioPintura:yAreaDe:

Estos métodos reciben los metros cuadrados y un precio en el caso de *calcularCostoMaterialesConPrecioPintura:yAreaDe:*. Se encargan de delegar en la herramienta los cálculos necesarios para obtener las partes del presupuesto. Ambos precios que devuelva no contendrán el valor de la cantidad de manos necesarias para el trabajo. Esto es agregado en otro momento, ya que se saco como factor común en la cuenta porque no es necesario en ese momento conocer la cantidad de manos. Haciendo esto se ahorra estar enviándole un parámetro más a los mensajes,evitando que se termine complicando un poco la lectura del código.

```
calcularCostoManoDeObraParaArea: metrosCuadrados
    ^herramienta costoManoDeObraPorArea: metrosCuadrados

calcularCostoMaterialesConPrecioPintura: precioPorLitro yAreaDe: metrosCuadrados
    ^herramienta costoMaterialesConPrecioPintura: precioPorLitro  yAreaDe: metrosCuadrados
```

Método manosNecesariasConPintura:

Este método también delega en la herramienta el saber cuantas manos son necesarias con la pintura que le manden.

Al comienzo se encontró el problema que había que encontrar una forma de poder tener la cantidad de manos necesarias de la pintura, por lo que se implemento temporalmente una estructura de control que terminaba rompiendo el encapsulamiento de las herramientas. Finalmente se decidió arreglar el problema de esta forma ya que así no es necesario saber que herramienta es realmente la que tiene el pintor. La herramienta misma, como veremos mas adelante, le pregunta a la pintura cuantas manos necesita para si misma al utilizar polimorfismo.

```
manosNecesariasConPintura: unaPintura
    ^herramienta manosNecesariasConPintura: unaPintura
```

4.3. Herramienta, Rodillo y Pincel

Al comienzo del trabajo, este estaba siendo implementado con dos clases de pintores, siendo estas *PintorConRodillo y PintorConPincel*. Pero llegado el momento de refactorización se llego a observar que iba a ser mas claro tener una clase *Pintor* que tenga un *Pincel o Rodillo*.

Al estar implementándolas, se vio que compartían bastante código resultando en codigo repetido, por lo que se llego a la conclusión que era mejor abstraer la parte en donde diferían, siendo esta la idea del instrumento del pintor. Por lo tanto, se creo una clase abstracta llamada *Herramienta*, de la cual heredarían *Pincel y Rodillo*, ya que estas clases cumplen la relación "es un" con la herramienta. Esta decisión de diseño termino brindando mas facilidad y claridad, ya que solamente fue necesario implementar los dos métodos en donde diferían *Pincel y Rodillo*.

La clase Herramienta contiene como variables de instancia litrosParaPintarMetroCuadrado y tiempoEnPintarMetroCuadrado, estas variables son inicializas y utilizadas como constantes durante la ejecución.

Método de clase cobraLaHora:

Este método de clase esta implementado en *Pincel y Rodillo*, vendrían a ser los constructores de las herramientas.

Se eligió ese nombre ya que esta pensado para que durante la creación de los pintores se lea todo junto. Un ejemplo de esto se puede ver en el apartado correspondiente a AlgoFix en la parte de los registros de pintores.

Este constructor creara la herramienta pedida y utilizara el setter agregarPrecio:. Este setter es privado y no debería ser usado desde afuera.

Método costoManoDeObraPorArea:

Este mensaje esta implementado y es responsabilidad de las clases hijas *Pincel y Rodillo*. Su responsabilidad es el el calculo de la mano de obra sin aplicar el costo de las manos.

A continuación se muestra primero el código correspondiente a la clase Pincel y después el de Rodillo. Observar que la diferencia esta principalmente en el descuento que se puede aplicar con pincel, resultando en un método ligeramente mas largo. Esto se debe a la estructura de control utilizada para determinar si corresponde un descuento. A este "if" no se le encontró una forma clara de poder evitarlo.

```
costoManoDeObraPorArea: metrosCuadrados
    | descuento |
    (metrosCuadrados <= 0) ifTrue:[AreaDebeSerMayorACeroError new signal.].
    descuento := 1.
    (metrosCuadradosParaDescuento < metrosCuadrados )ifTrue: [
        descuento := 0.5.
    ].
    ^precioPorHora * metrosCuadrados * tiempoEnPintarMetroCuadrado * descuento

costoManoDeObraPorArea: metrosCuadrados
    (metrosCuadrados <= 0) ifTrue:[AreaDebeSerMayorACeroError new signal.].
    ^precioPorHora * metrosCuadrados  * tiempoEnPintarMetroCuadrado
```

Método costoMaterialesConPrecioPintura

Esta implementado en la clase herramienta y es heredado por las clases hijas. El comportamiento es igual en ambas.

```
costoMaterialesConPrecioPintura: precioPorLitro yAreaDe: metrosCuadrados
    (metrosCuadrados <= 0)ifTrue:[AreaDebeSerMayorACeroError new signal].
    ^precioPorLitro * metrosCuadrados * litrosParaPintarMetroCuadrado
```

Método manosNecesariasConPintura:

Le pregunta a la pintura cuantas manos son necesarias para la herramienta que la esta llamando. Se aprovecho enormemente el polimorfismo en este caso, ya que implementar este mensaje fue la forma de evitar romper el encapsulamiento de las herramientas preguntadoles directamente que tipo de herramienta eran.

```
manosNecesariasConPintura: unaPintura
    ^unaPintura manosNecesariasConPincel

manosNecesariasConPintura: unaPintura
    ^unaPintura manosNecesariasConRodillo
```

4.4. Pintura

Las instancias de esta clase son creadas por AlgoFix y son manejadas por los clientes. Contiene toda la información relevante de una pintura y es utilizada para calcular parte de los presupuestos.

Método de clase color:conPrecio:manosPincel:yManosRodillo:

Es el constructor que utilizan las pinturas. Este al crearlas utiliza varios setters que son privados y no deberían ser usados desde afuera. Estos métodos son *agregarNombre:*, *agregarPrecio:*, *agregarManosPincel:*, y *agregarManosRodillo:*.

Métodos manosNecesariasConRodillo y manosNecesariasConPincel

Simplemente devuelven el dato pedido a la pintura.

Método calcularCostoMaterialesParaArea:

Delega en el pintor el calculo correspondiente a los materiales, pasando la informacion del precio de la pintura.

```
calcularCostoMaterialesParaArea: metrosCuadrados conPintor: unPintor
    ^ unPintor calcularCostoMaterialesConPrecioPintura: precioPorLitro
        yAreaDe: metrosCuadrados
```

4.5. Presupuesto

Esta clase esta ya que es una de las pedidas por el enunciado. Se encarga de administrar las operaciones generales de buscar el mejor presupuesto para la cantidad de metros cuadrados indicada. Internamente guarda al pintor que ofrece el mejor presupuesto y al valor total de este.

Métodos valor y responsable

Pedidos en las pruebas provistas por la cátedra, devuelven los datos correspondientes al presupuesto.

Método calcularMasBaratoEntre:para:conPintura:

Este método se encarga de la búsqueda del pintor que ofrezca el menor presupuesto, para eso, se llama a si mismo con el mensaje *calcularPresupuestoDe:paraMetrosCuadrados:conPintura:* que va a ser el responsable de calcular el presupuesto correspondiente a cada pintor.

Luego de finalizar la búsqueda, se decidió realizar el calculo devuelta para el mejor pintor ya que este valor no queda guardado durante el proceso anterior. No se quiso realizar el calculo al momento de pedir el valor ya que en ese caso no se tiene la pintura para realizar las cuentas.

```
calcularMasBaratoEntre: pintores para: metrosCuadrados conPintura: unaPintura
    (pintores isEmpty)ifTrue:[NoHayPintoresRegistradosError new signal].
    pintorConMejorPresupuesto := pintores detectMin: [ :unPintor |
        self calcularPresupuestoDe: unPintor paraMetrosCuadrados: metrosCuadrados
            conPintura: unaPintura.
    ].
    valorTotal := self calcularPresupuestoDe: pintorConMejorPresupuesto
                paraMetrosCuadrados: metrosCuadrados conPintura: unaPintura.
```

Método calcularPresupuestoDe:paraMetrosCuadrados:conPintura:

Este método divide en tres partes el calculo del presupuesto de cada pintor. La primera parte corresponde al calculo de la mano de obra, es decir, el precio que corresponde al pintor por realizar el trabajo. La segunda parte corresponde a los materiales, por lo que se necesita de la pintura y de la herramienta con la que trabaje el pintor. Finalmente , la tercera parte es multiplicar por la cantidad de manos necesarias para pintar. Esto se realiza al final ya que se vio que se simplificaba y evitaba estar pasando un parámetro más a los diferentes métodos del pintor.

```
calcularPresupuestoDe: unPintor paraMetrosCuadrados: metrosCuadrados conPintura: unaPintura
    | valorManoDeObra valorMateriales presupuesto |
    valorManoDeObra := unPintor calcularCostoManoDeObraParaArea: metrosCuadrados.
    valorMateriales := unaPintura calcularCostoMaterialesParaArea: metrosCuadrados
                      conPintor: unPintor.

    presupuesto := (valorManoDeObra + valorMateriales) *
                    (unPintor manosNecesariasConPintura: unaPintura).

    ^presupuesto
```

5. Excepciones

NoHayPintoresRegistradosError Esta excepción fue creada con el objetivo de evitar problemas en caso de que el usuario intente realizar la operación *presupuestoMasBaratoParaPintarMetrosCuadrados:conPintura:* cuando la instancia de AlgoFix llamada no tiene ningún pintor registrado.

PrecioNoPuedeSerNegativoError Esta excepción cumple el objetivo de evitar que el usuario ingrese algún precio negativo, ya sea en pinturas o en el precio del pintor. Si esto estuviese permitido, no tendrían sentido los diferentes costos.

AreaDebeSerMayorACeroError Esta excepción indica que el usuario intento pedir un presupuesto con una cantidad de metros cuadrados nula o negativa. Esto no tendría sentido al momento de realizar los diferentes cálculos.

LaCantidadDeManosDebeSerMayorACeroError Esta excepción se lanza cuando se intenta crear una pintura con cero o menos manos de pintura, ya sea en rodillos o pincel. Ya que por lo menos algo de pintura se le debe de pasar a la superficie pintada.

PresupuestoNoCalculadoError Esta excepción fue creada para que se lance cuando se le pide a un presupuesto el responsable o el valor cuando por algún motivo no se tiene uno.

6. Diagramas de secuencia

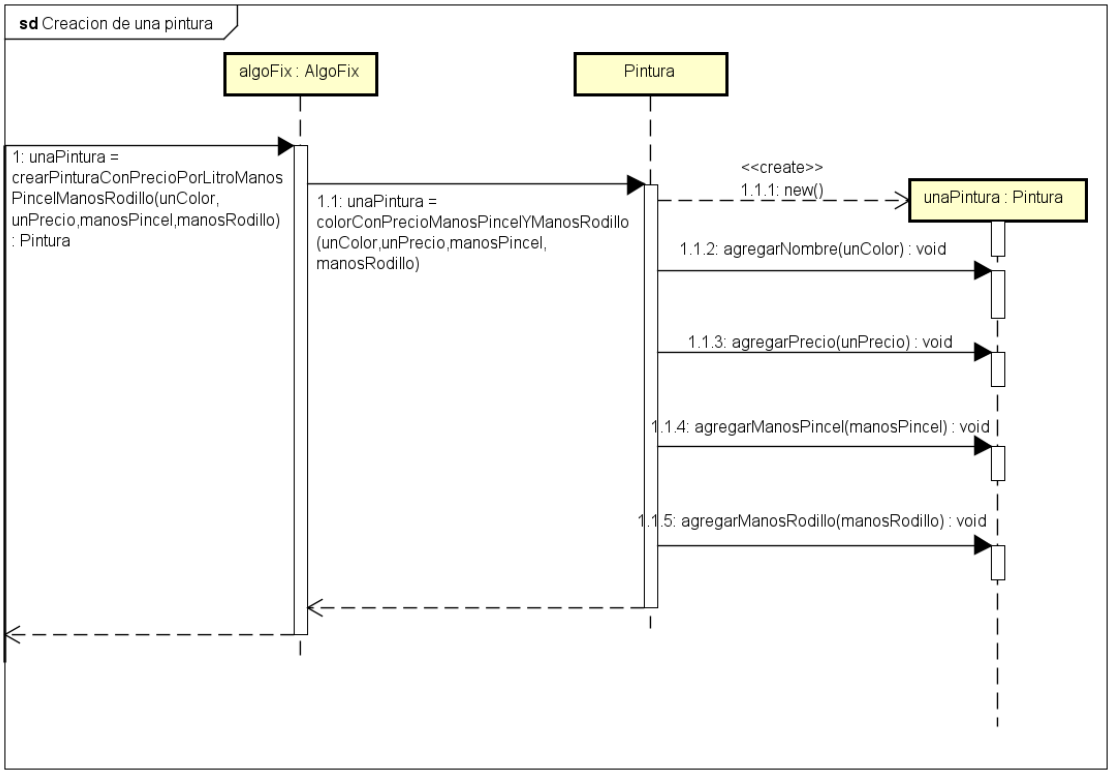

Figura 5: Creación de las pinturas

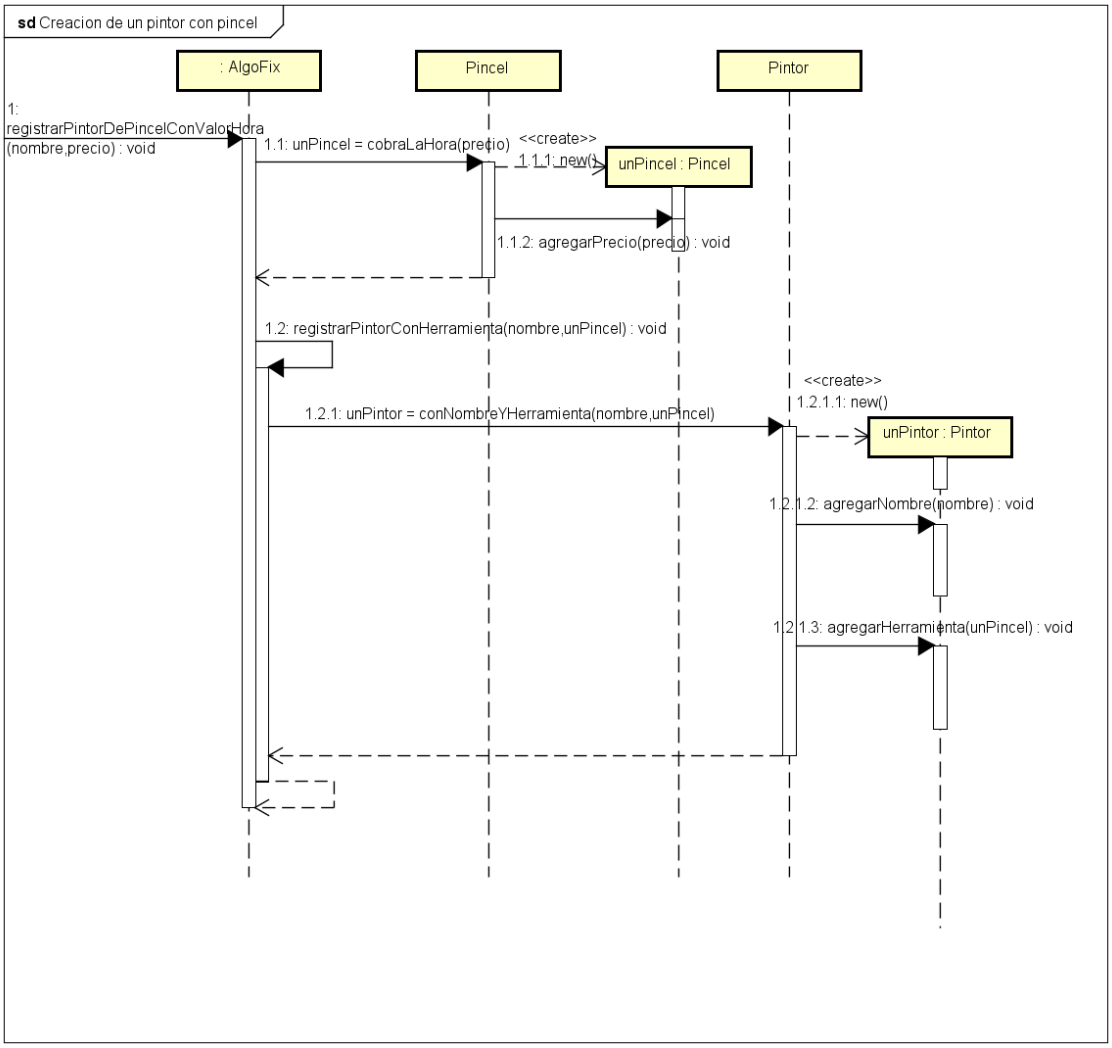

Figura 6: Creación de pintor con pincel

En el caso de querer crear un pintor con rodillo seria muy similar, solamente cambia la clase que recibe el mensaje de creación y la instancia creada.

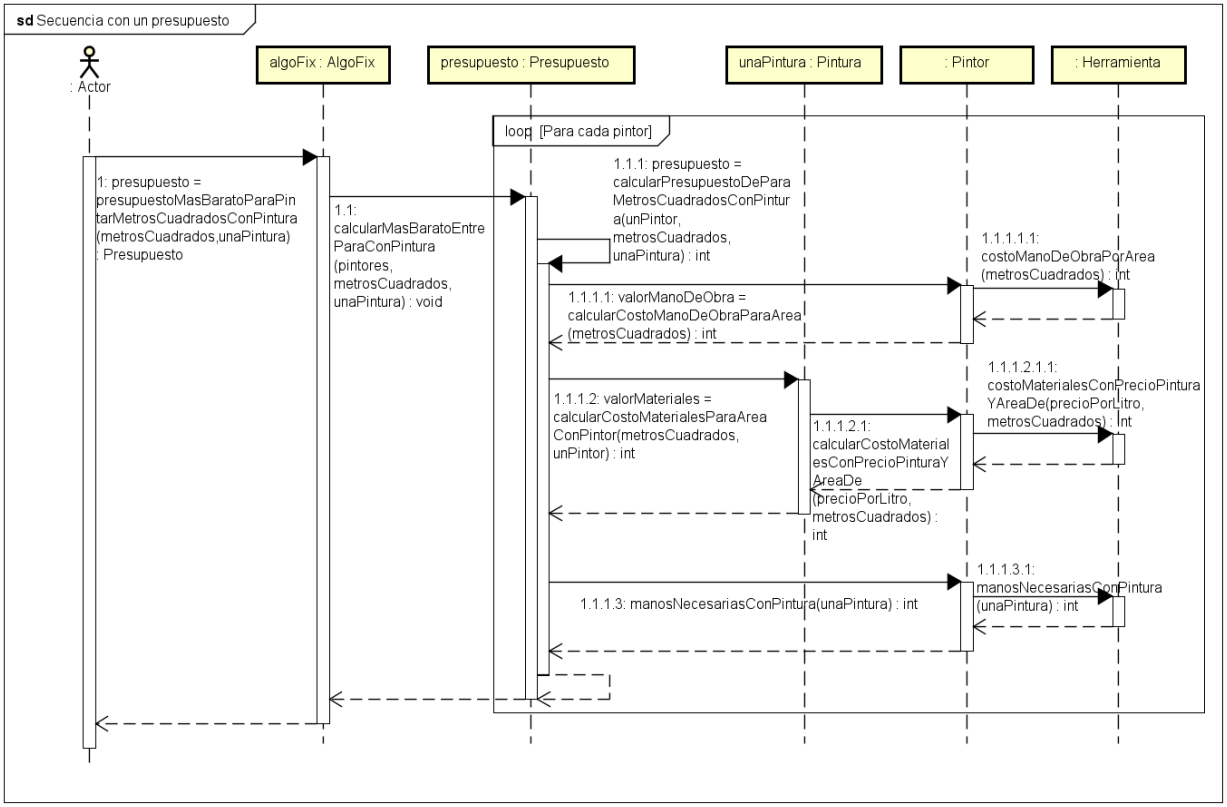

Figura 7: Calculo de un presupuesto.

Se omitió en el diagrama el mensaje «create» que iría de algoFix a la clase de presupuesto para que quede mas prolijo.

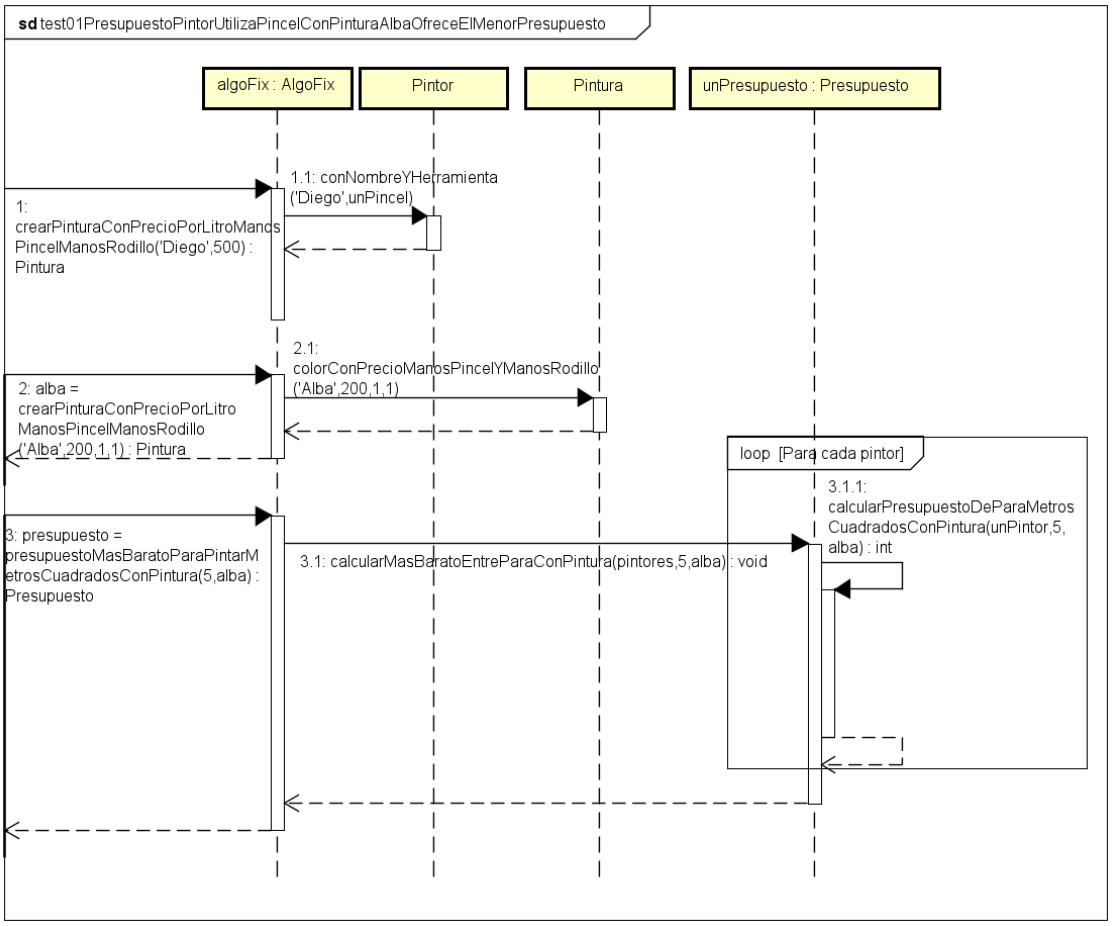

Figura 8: Prueba 01 provista por la cátedra

Este diagrama muestra la primera prueba del trabajo. Como se puede ver, se crea un solo pintor y una pintura. Luego se le pide a algoFix el presupuesto más barato, el cual terminara siendo el del único pintor cargado.

Se ocultaron algunos métodos internos para dar mas claridad al diagrama. Estos detalles internos fueron mostrados en los diagramas anteriores.

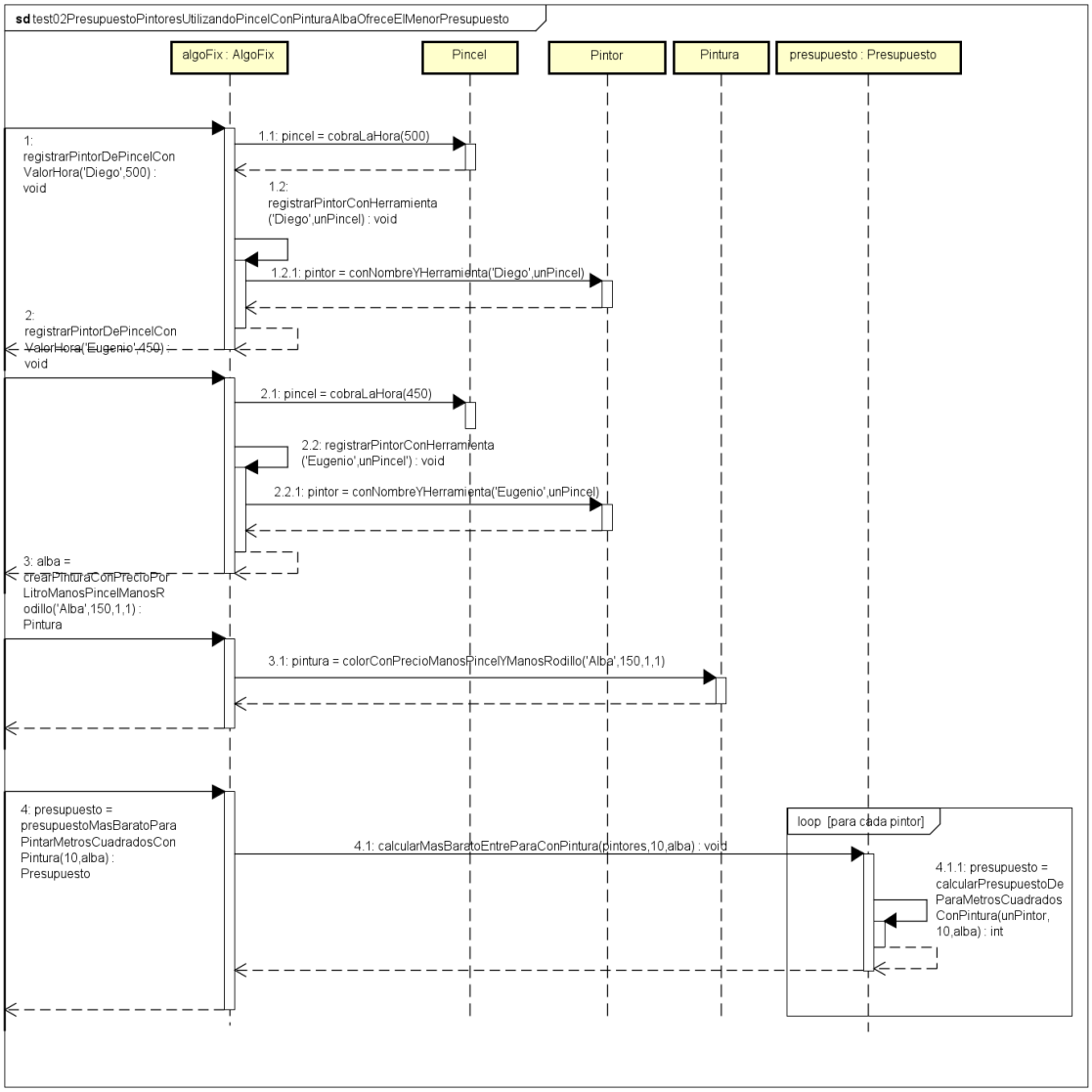

Figura 9: Prueba 02 provista por la cátedra

En este diagrama se puede observar la segunda prueba provista por la cátedra. Los partes que surgen de los dos primeros llamados a algoFix corresponden al agregado de los pintores con pincel a la colección que esta contiene. Estos pintores son 'Diego' y 'Eugenio'.

La tercera parte responde a la creación de la pintura que pide el usuario, en este caso la pintura 'alba'. Por ultimo, la cuarta parte es el pedido de un presupuesto para 10 metros cuadrados. Este presupuesto seria creado en algoFix y se le es mandado el mensaje para que calcule el mas barato. Este luego recorrerá cada pintor en la colección que recibe y se quedara con el que menor presupuesto tenga.

En este diagrama de clases, se ocultaron algunos métodos que no se vieron necesarios para transmitir el objetivo de la prueba. Estos son algunos setters y getters junto con los mensajes internos del presupuesto para el calculo. Esto se puede ver en los anteriores diagramas.