

1. Este metodo se aplica cuando existe duplicacion de codigo en metodos que son parecidos en una jerarquia. Es decir, cuando se tiene una jerarquia que posee metodos para una funcionalidad en comun pero esta difiere en algunas partes, por ende, cada subclase debe implementar el metodo nuevamente. En estos casos se puede aplicar el template method para permitir que se use partes en comun y solo se implemente la parte que difiere.

Entonces, se usa cuando se quiere generar un metodo con la parte en comun y dejar que las subclases implementen la parte que difiere.

Tambien se puede usar este patron en frameworks para separar los frozenspots de los hotspots.

1. Este patron puede implementarse cuando se quiere generar una composición de objetos pero que el cliente no distinga cuando se trata de un objeto compuesto o uno simple (atomico), dado que esto es transparente para el.
2. El patron state se aplica cuando se quiere aislar el estado en el que puede estar un objeto del propio objeto. El objeto puede cambiar de estado de forma dinamica e interna, pero el objeto no cambia de clase, si no que cambia de estado.
3. El patron adapter se puede aplicar cuando se quiere usar componentes que no implementan la misma interface que el cliente, entonces se necesita adaptar el mismo dado que la idea no es cambiar el componente que ya funciona o porque por ahi no se tiene acceso para realizar dicho cambio.
4. Un framework es un marco de trabajo que sirve para generar una aplicación de forma mas productiva y con mas calidad dado que provee una plantilla y resuelve problemas que tienen en comun todas las aplicaciones.

Por ejemplo, seaside es un framework de tipo caja blanca que provee metodos que se pueden reescribir (puntos de extension) para generar la aplicación especifica.

Los frameworks son aplicaciones semi-completas dado que tienen gran parte de la misma hecha.

1. Los frozenspots y hotspot son partes que componen una aplicación hecha con un framework. Frozenspot es aquella parte que no puede modificarse y que fue hecha por el desarrollador del framework, son reglas que deben seguirse para usar el mismo.

Por otro lado, el hotspot es aquello que puede manejarse libremente para generar la parte específica de la aplicación.

En el caso de seaside se tienen interfaces que deben estar y no pueden manipularse ya que conforman parte del framework, pero tienen hotspots como el método renderContentOn que puede modificarse como uno lo desee. Sin embargo debe cumplir la interfaz propuesta (frozenspot), es decir, el método debe implementarse con ese nombre para que se vea en la aplicación.

1. Esto ocurre cuando no es el codigo nuestro el que controla al componente que esta usando. Por ejemplo, en las librerias de clases, es el modelo quien usa y controla a los mismos, pero en los framework son ellos los que toman el control del codigo y lo manejan, por esto se dice que se invirtio el control.
2. Hacer refactoring es generar código limpio, con buenas prácticas a partir de código existente sin cambiar su funcionalidad.

Dado que muchas veces las aplicaciones se generan sin tener un buen conocimiento del dominio y se hacen de forma apurada para poder tener el código funcionando, con el paso del tiempo comienzan a generarse BBoM, es decir, grandes cantidades de código inentendible e inmantenible. Por esto es que para que la aplicación no se deteriore o muera es necesario refactorizar para que se pueda seguir manteniendo y ampliando con el tiempo.

1. Los bad smells son aquellos indicadores en un codigo que nos permiten ver que pueden provocar problemas en el futuro y por esto es necesario trabajarlos.

Dos tipos de bad smells pueden ser:

* Envidia de atributos: significa que una clase quiere o utiliza atributos que mantiene otra clase para realizar alguna operacion. Para poder refactorizar esto puede recurrirse a move method (porque quizas hay una mala asignacion de responsabilidad para ese metodo) o move field. Inclusive, dependiendo el caso, puede llegar a darse un extract method seguido de rename method y move method si la envidia se produce en un método muy grande.
* Codigo duplicado: Esto sucede cuando se tiene codigo repetido en varios metodos cuando podrian estar en uno solo y reutilizarse.

Para arreglar esto se podria utilizar el refactoring de template method (aplicar patron) o extract method/pull up method generando una jerarquia si es que no existe.

1. Para testear con SUnit es necesario crear test que puedan probar funcionalidades concretas, por ello es necesario aislar la unidad a testear (SUT), del resto. Para esto se pueden usar mocks que son objetos falsos que imitan los objetos verdaderos que acompañan al sut. Esto se usa dado que no se quiere acarrear problemas que puedan surgir de otros objetos al objeto que se esta testeando.

**1) ¿Qué es un patrón de diseño?**

**2) ¿Cuál es el formato con el que se documentan y qué describe cada sección?**

**3)De acuerdo a lo visto en la clase diría que SUnit es una librería o un Framework. ¿Por qué?**

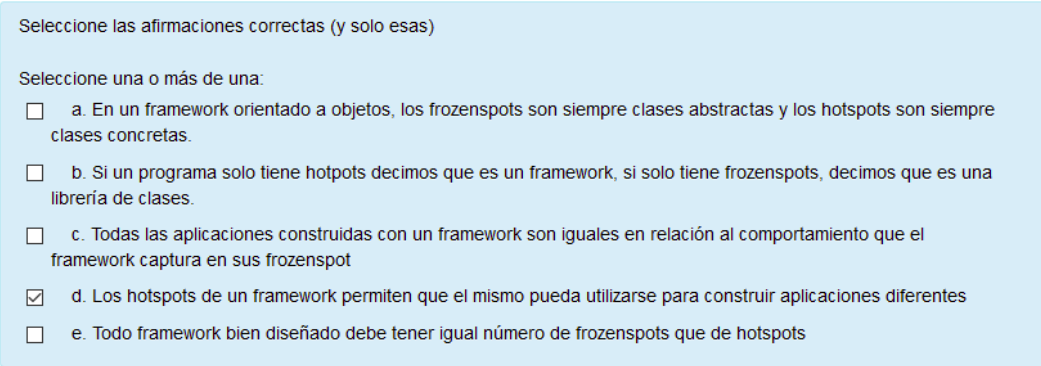
1) Un patrón de diseño es una manera de crear código para resolver una problemática que se ve reflejada en varias situaciones, por esto se crea un patrón que resuelva dicha problemática.

2 ) Un patron de diseño consta de varias partes las cuales se documentan de tal manera:

* Nombre
* Intencion: es el proposito por el cual se genero dicho patron.
* Motivacion:
* Aplicabilidad: son las situaciones en donde se puede aplicar el patron y que ayuda a solucionar las problematicas.
* Estructura: es la estructura general que mantiene el patron.
* Participantes: son los componentes que forman parte de la estructura.
* Colaboracion: son las diferentes colaboraciones que se dan entre los participantes.
* Consequences
* Implementatión
* Code
* Known Uses

3) Diría que SUnit es un framework dado que mantiene una estructura y una interfaz para funcionar y realiza todos los trabajos necesarios para testear, dejando hotspot tipo herencia dado que para hacer un test debe heredarse de las clases de SUnit.

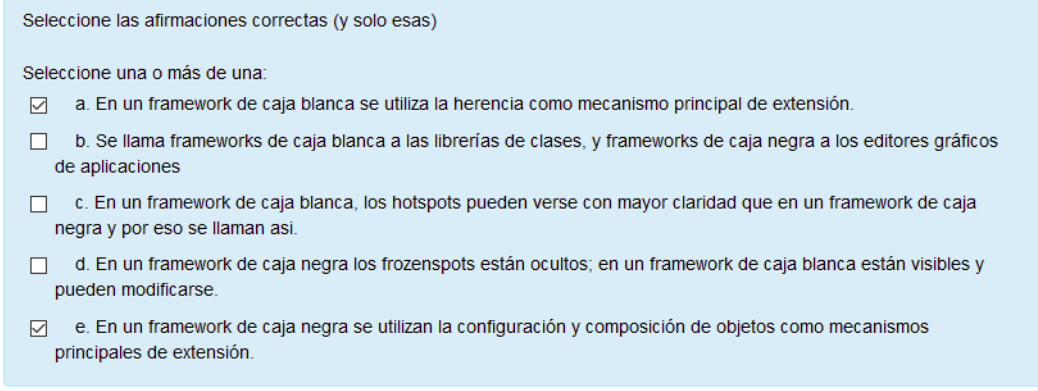
Cuando se utiliza el patron Observer?



Rta.

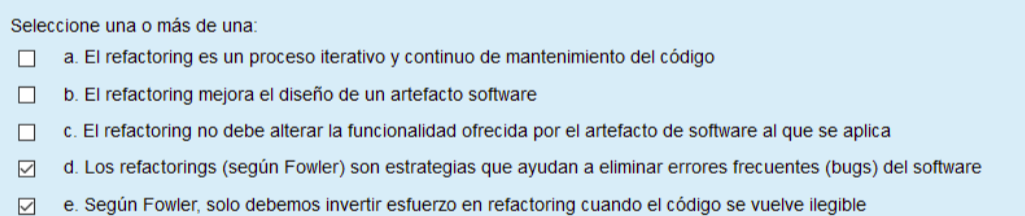
a, no porque existen clases que forman parte del framework que no son abstractas porque no las implementa el programador de la app.

c, porque las apps comparten la parte de frozenspot

d, porque justamente los hotspot son la parte que el desarrollador puede cambiar para hacer apps distintas.

Rta. a y e.

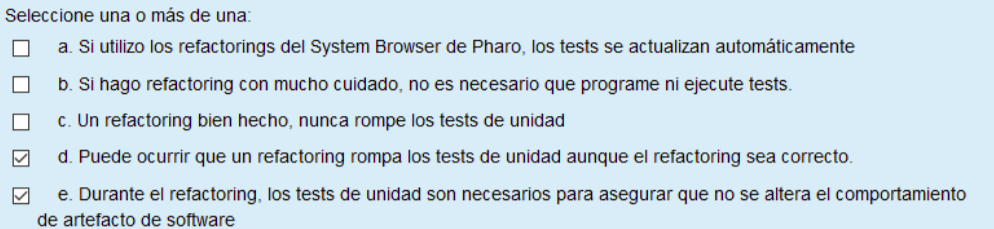
La d me hace dudar porque si bien los frozenspot estan ocultos en caja negra, no se si en caja blanca estan visibles, ademas no pueden modificarse.



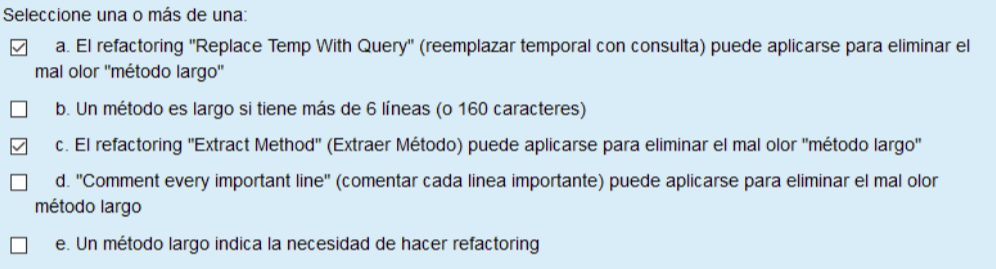
Me suena a que son a,b,c y d o todas esas menos la a.

La e es rara, porque se puede aplicar refactoring cuando el código aún es legible.

El refactoring es un proceso iterativo y continuo de mantenimiento del codigo??



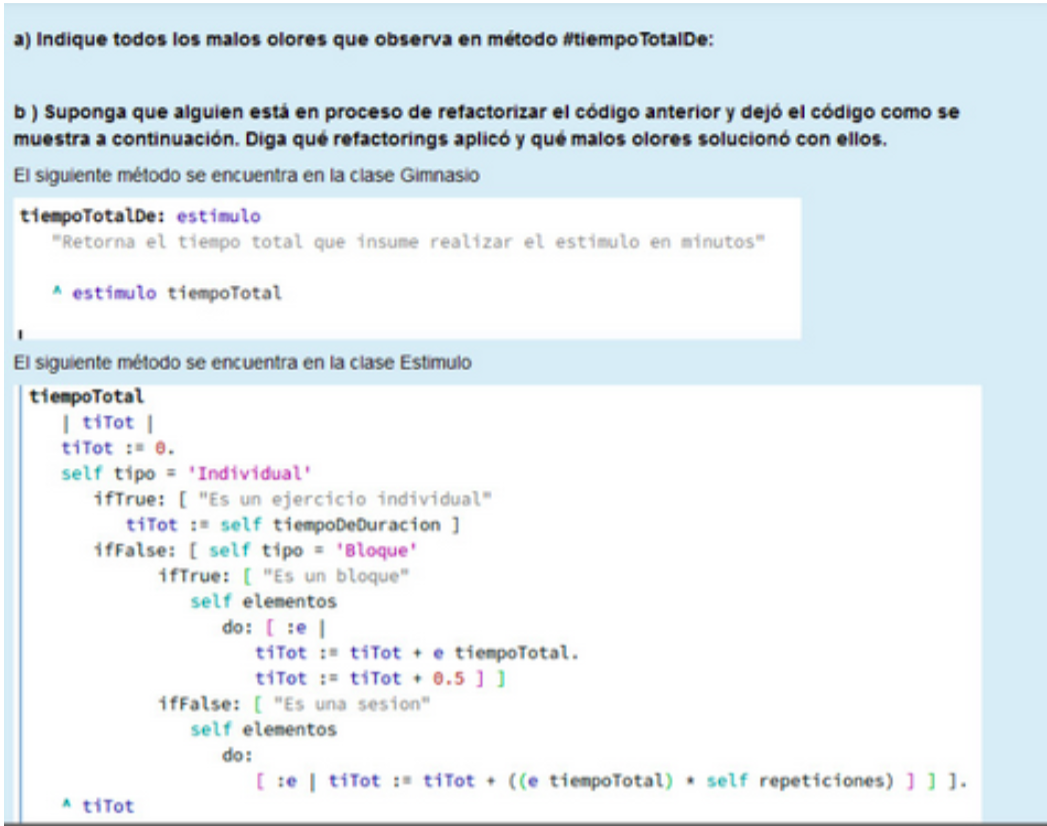
La b es media capciosa, porque técnicamente se podría hacer, aunque no se tiene la seguridad de que no cambió nada.

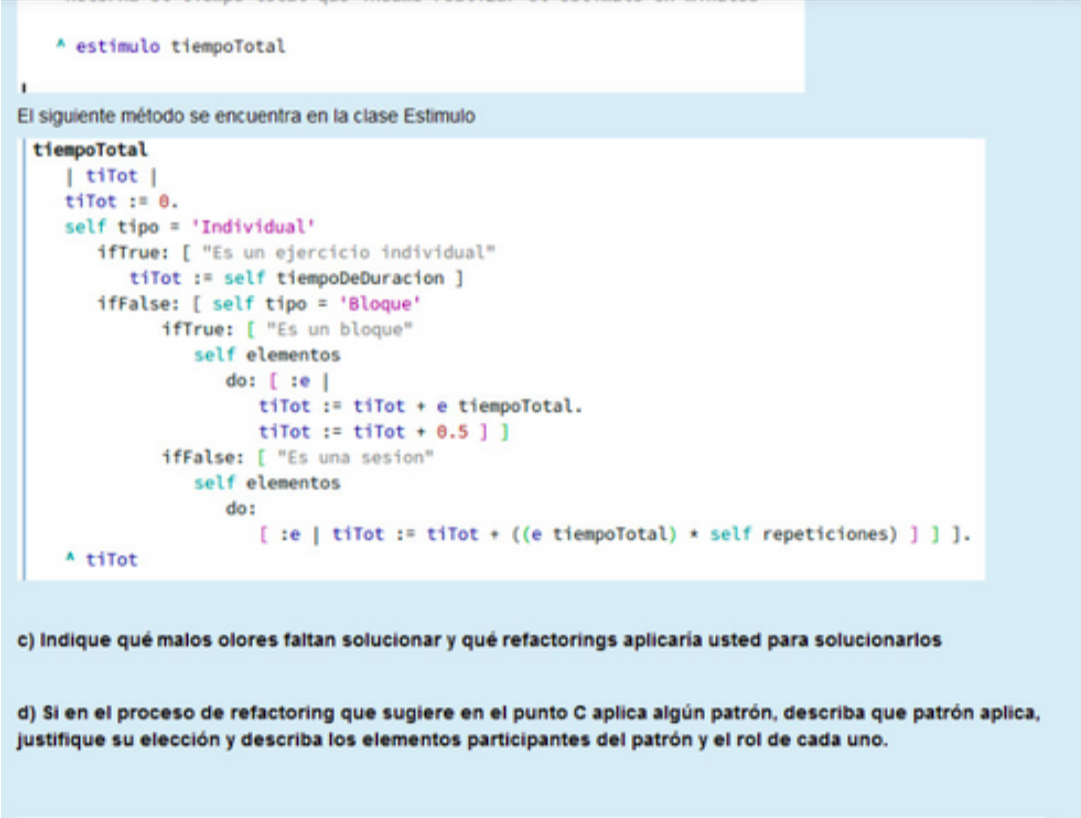


Todas las marcadas están bien pero falta la e.



1. hay envidia de atributos, comentarios para explicar, metodo demasiado largo, mala asignacion de responsabilidad, switch statement



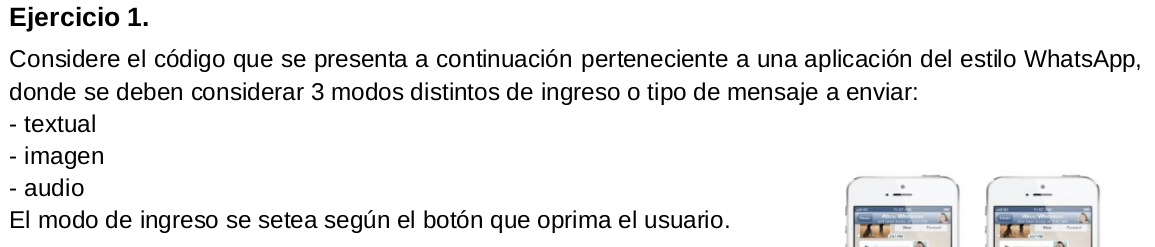
b) Aplico un extract method y move method y soluciono la envidia de atributos y la mala asignacion de responsabilidad.

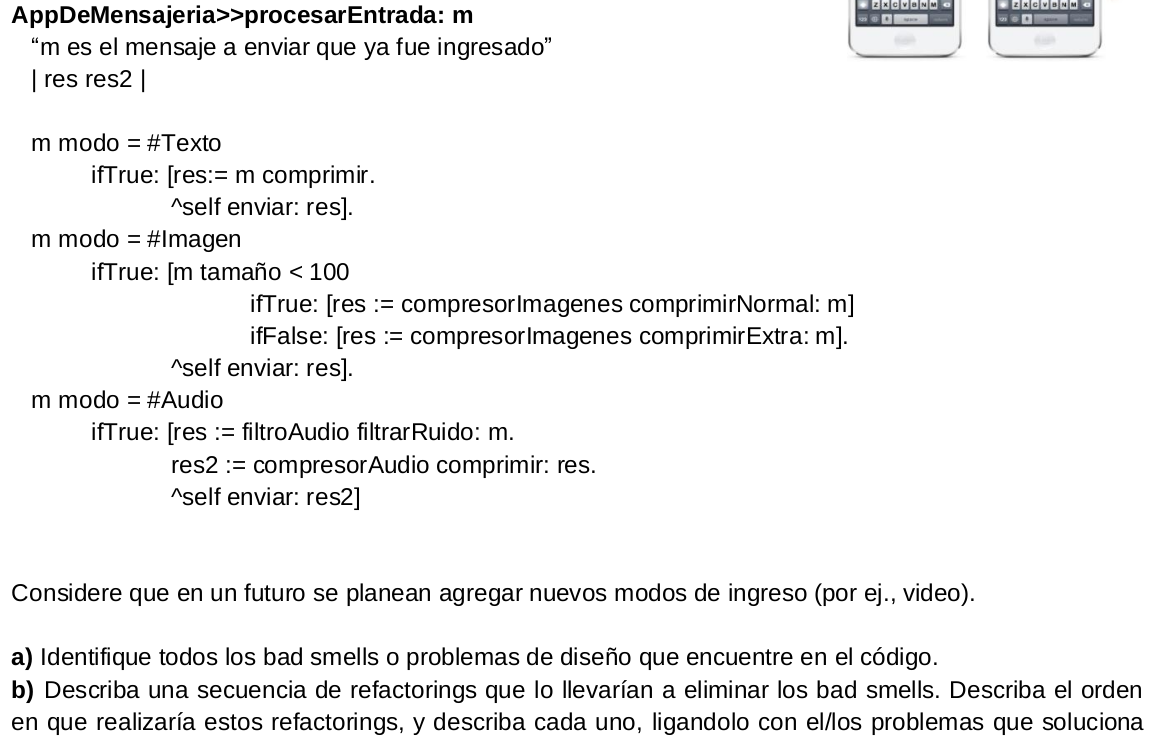
c) Falta solucionar el mal olor de metodo muy largo, los comentarios (no tienen que explicar si el codigo se puede explicar solo) y el switch statement.

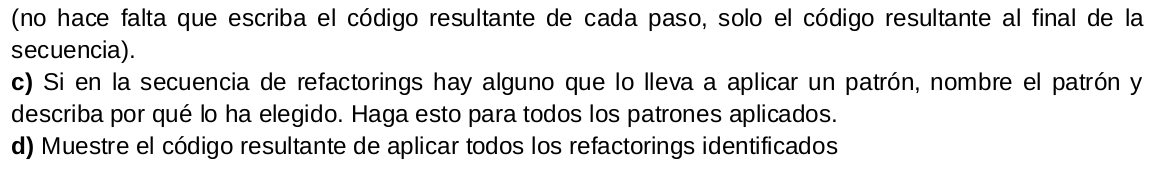
1. Para el metodo largo aplicaria primero extract method para separar los distintos mecanismos de conteo y luego un replace condicional with polymorphism para solucionar el switch statement, quedando estimulo como padre de la jerarquia, y ejercicio, bloque y sesion como subclases con sus respectivos metodos de conteo.
2. Una vez obtenida las clases llevaria el modelo a un composite.

El ejercicio vendria a ser el leaf, el bloque y sesion son composite y el component seria estimulo.

Este patron serviria para mostrar el anidamiento y composicion que existen entre las clases compuestas como bloque y sesion y lo que pueden contener en forma de arbol.







1. 1. Nombre no autodescriptivo: El parametro “m” no es autodescriptivo, no se entiende a que es lo que se refiere solo con el nombre, por ende luego se indica con un comentario.

2. Metodo largo: es un metodo muy largo y dificil de seguir.

3. Switch Statement: se tiene un condicional que pregunta por el tipo de mensaje que es para derivar en una accion, si se agrega otro tipo se puede generar un conflicto si no se propaga el cambio en todos los switch existentes.

4. Envidia de atributos: se pide un campo del mensaje “tamaño” para poder realizar una compresion que en realidad deberia hacer el propio mensaje.

5. Mala asignacion de responsabilidad: la compresion deberia ser algo que realice Mensaje no la la AppDeMensajeria.

7. Codigo repetido: por cada accion se realiza un ^self enviar: res. Esto podria ser realizado una unica vez.

8. Variables temporales redundantes: variables temporales que podrian obviarse.

1. 1. Se cambia el nombre del parametro “m” por “mensaje” y se reemplaza el nombre en todos los lugares que lo mencionen. Por ultimo se elimina el comentario explicativo resolviendo el primer problema.

2. Luego se aplica un extract y move method para mover la parte condicional a otro metodo a la clase Mensaje y si se puede tambien move field para las v.i compresorImagenes, filtroAudio y compresorAudio, si no se puede entonces mandarlo como parametro. Ajustar las partes de envio, si se puede pasar a Mensaje o generar una delegacion (yo supongo que no se puede pasar). En la clase AppMensajeria dejar una delegacion al metodo “procesar” creado en Mensaje. Soluciona el metodo largo dentro de AppMensajeria pero lo genera en Mensaje. Soluciona la envidia de atributos.

3. Una vez dentro de Mensajeria aplicar el refactoring replace conditional with polimorphism. Entonces se extraeria del metodo “procesar” aquellas partes del condicional que correspondan a cada subclase y se redefinen en cada una de ellas: MensajeTexto, MensajeAudio, MensajeImagen. Entonces el metodo de la superclase queda abstracto.

Soluciona el metodo largo y el switch statement.

4. Luego aplico un replace temp with query para las partes que utilicen res o res2.

Todas las partes quedarian unicamente con una query para devolver sin asignar a una variable temporal.

En el caso de MensajeAudio se crearia una cadena de mensajes en vez de asignarla a una temporal y luego a otra.

1. No se utiliza ningun patron en la secuencia de patrones.

AppMensajeria >> procesarEntrada: mensaje

^self enviar: (mensaje procesar).

Mensaje >> procesar

^self subclassResponsability.

MensajeTexto >> procesar

^self comprimir.

MensajeImagen >> procesar

tamaño < 100 ifTrue: [^compresorImagenes comprimirNormal: self ]

ifFalse: [^compresorImagenes comprimirExtra: self ].

MensajeAudio >> procesar

^ (compresorAudio comprimir(filtroAudio filtrarRuido:self)).

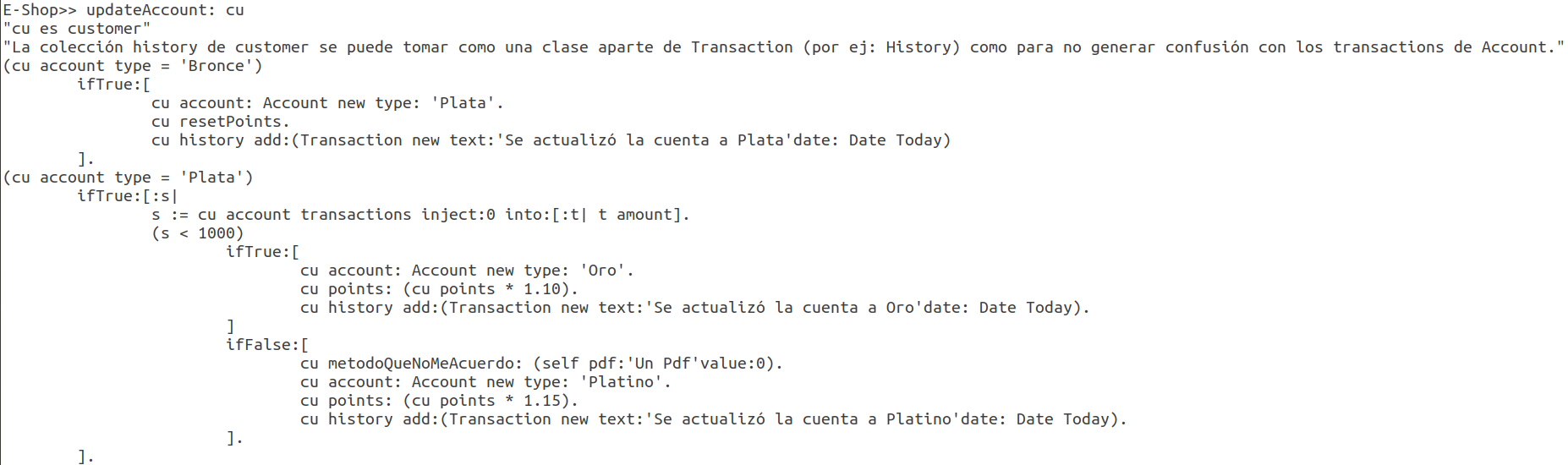
**1) Similitudes y diferencias entre State y Strategy (3 pts)**

Ambos extraen funcionalidad del objeto en cuestión hacia afuera para simplificar este último y evitar condicionales complejos.

La diferencia entre ambos son varias, pero principalmente su objetivo y propósito.

El state pretende extraer algoritmia y que esta pueda ser reemplazable dinámicamente en el cliente, no es transparente a este último. Por otro lado, el state pretende extraer el estado en el que puede estar el objeto, de manera que este no se vea distinto por fuera, es transparente para el cliente el estado en el que está.

Ademas el state conoce a los demas estados mientras que el strategy no conoce las distintas estrategias existentes.

2) Un E-Shop que actualiza la cuenta de un Customer por otra mejor. (7 pts)

a)Listar los malos olores

1. Mal nombre de parametro: se tiene un nombre poco descriptivo para el parametro, por esto necesita un comentario que explique que es.
2. Envidia de atributos: La clase E-Shop esta pidiendo informacion de cu para realizar tareas que deberia hacer esa clase.
3. Metodo largo: es un metodo muy largo y no se entiende, por ende se usan comentarios para indicar como funciona.
4. Switch Statement: se tiene un condicional por cada tipo de customer.
5. Codigo duplicado: se tienen partes del codigo que son parecidas pero difieren por pequeñas cosas.

b) Para cada mal olor decir que utilizarían para arreglarlo. Si usan algún patrón justificar el porqué.

1. Primero renombro el parametro “cu” por “customer” y lo reemplazo en donde se la utilice. Por ultimo elimino el comentario explicativo. Solucion al primer bad smell.
2. Luego aplico el refactoring extract method. Genero un metodo update en Customer y luego paso todo lo referido a la parte condicional pasando el metodo pdf si se puede.

En la clase E-Shop dejo una delegacion de este nuevo metodo a Customer.

Se arregla la envidia de atributos en E-Shop pero se genera en customer cuando pregunta por el tipo de account que es.

Luego se genera una estrategia de actualización (patron strategy) para manejar la misma.

Se crea la clase UpdateStrategy y se pasa el metodo pdf y la parte condicional del metodo del Customer dejando una delegacion en este, y ademas se crea una v.i con la estrategia. En el UpdateStrategy se aplica el refactoring replace conditional with polimorphism.

Se generan subestrategias: UpdateStrategyBronce, UpdateStrategyPlata (si bien se ve que existen cuentas oro y platino, no se tiene un algoritmo de actualización por lo tanto no se lo tratara aqui).

Se pasa la parte condicional a las estrategias correspondientes.

Con esto se arregla la envidia de atributos del customer y el switch statement que hay en este.

1. Para evitar la duplicacion del codigo se puede aplicar un template method, dejando las partes que se repiten arriba de la jerarquia en el strategy y las partes que son diferentes en las subestrategias.

2. Luego se genera un estado para el account (patron state) para manejar la misma.

Se crea la clase CustomerAccountState y se pasa el metodo pdf y la parte condicional del metodo del Customer dejando una delegacion en este, y ademas se crea una v.i con la estrategia. En el CustomerAccountState se aplica el refactoring replace conditional with polimorphism.

Se generan subestrategias: CustomerAccountStateBronce, CustomerAccountStatePlata, CustomerAccountStateOro, CustomerAccountStatePlatino (si bien no se tiene un algoritmo de actualización se los tiene como estados y luego tendran un algoritmo por defecto, si en el futuro se tiene una algoritmia se podra implementar).

Se pasa la parte condicional a los estados correspondientes.

Con esto se arregla la envidia de atributos del customer y el switch statement que hay en este.

Para evitar la duplicacion del codigo se puede aplicar un template method, dejando las partes que se repiten arriba de la jerarquia en el state y las partes que son diferentes en los substates.

c) Realizar diagrama y código resultante del paso b.

E-Shop>> updateAccount: customer

customer update.

Customer >> update

myCustomerAccountState updateAccount.

Customer >> customerAccountState: unCustomerAccountState

myCustomerAccountState := unCustomerAccountState.

CustomerAccountState >> updateAccount

myCustomer account: self newTypeAccount.

self setPoint.

myCustomer addHistorial: self historialMark.

myCustomer myCustomerAccountState: self newAccountState.

Customer >> addHistorial: anHistorialMark

history add: anHistorialMark

CustomerAccountState >> newTypeAccount

^subclassResponsability.

CustomerAccountState >> setPoint

^subclassResponsability.

CustomerAccountState >> historialMark

^subclassResponsability.

CustomerAccountStateBronce >> newTypeAccount

^Account new type: 'Plata'.

CustomerAccountStateBronce >> setPoint

myCustomer resetPoints.

CustomerAccountStateBronce >> newAccountState

^CustomerAccountStatePlata new customer: myCustomer.

CustomerAccountStateBronce >> historialMark

^Transaction new text:'Se actualizó la cuenta a Plata'date: Date Today.

CustomerAccountStatePlata >> updateAccount

myCustomer transactionsLessOf1000

ifTrue: [self updateStrategyAccountToOro ]

ifFalse:[self updateStrategyAccountToPlatino]

CustomerAccountStatePlata >> newTypeAccount

^Account new type: myUpdateStrategyAccount type.

CustomerAccountStatePlata >> setPoint

^myUpdateStrategyAccount points.

CustomerAccountStatePlata >> historialMark

^Transaction new text: myUpdateStrategyAccount message date: Date Today.

CustomerAccountStatePlata >> newAccountState

^myUpdateStrategyAccount newAccountState.

CustomerAccountStatePlata >> updateStrategyAccountToOro

myUpdateStrategyAccount:= UpdateStrategyAccountOro new: myCustomer.

CustomerAccountStatePlata >> updateStrategyAccountToPlatino

myUpdateStrategyAccount:= UpdateStrategyAccountPlatino new: myCustomer.

UpdateStrategyAccountOro >> type

^ ‘Oro’.

UpdateStrategyAccountOro >> points

^myCustomer points \* 1.10.

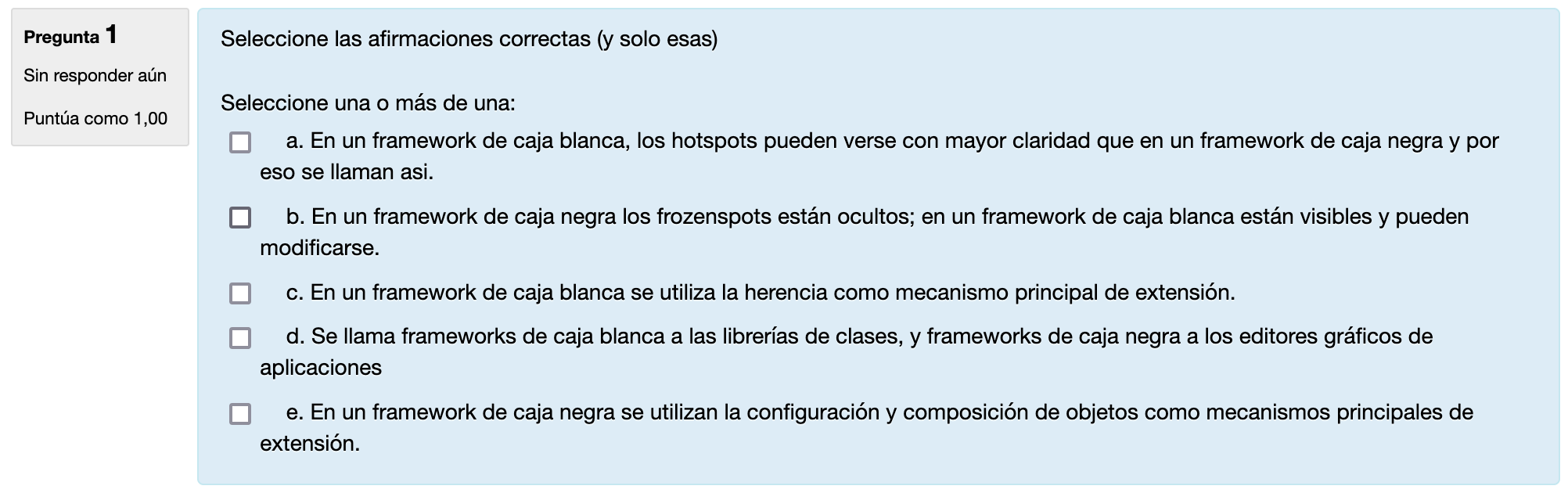
UpdateStrategyAccountOro >> message

^ ‘Se actualizó la cuenta a Oro’.

UpdateStrategyAccountOro >> newAccountState

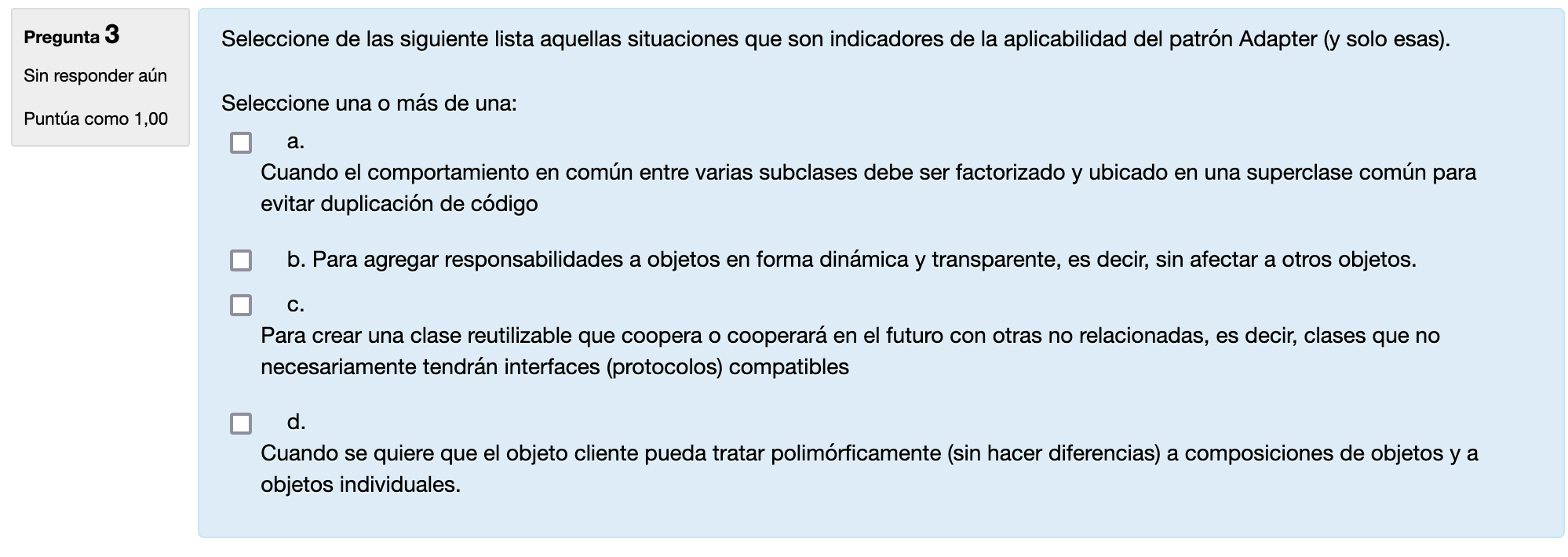
^CustomerAccountStateOro new: myCustomer.

ME PUDRI DE ESTE EJERCICIO DE MIERDA

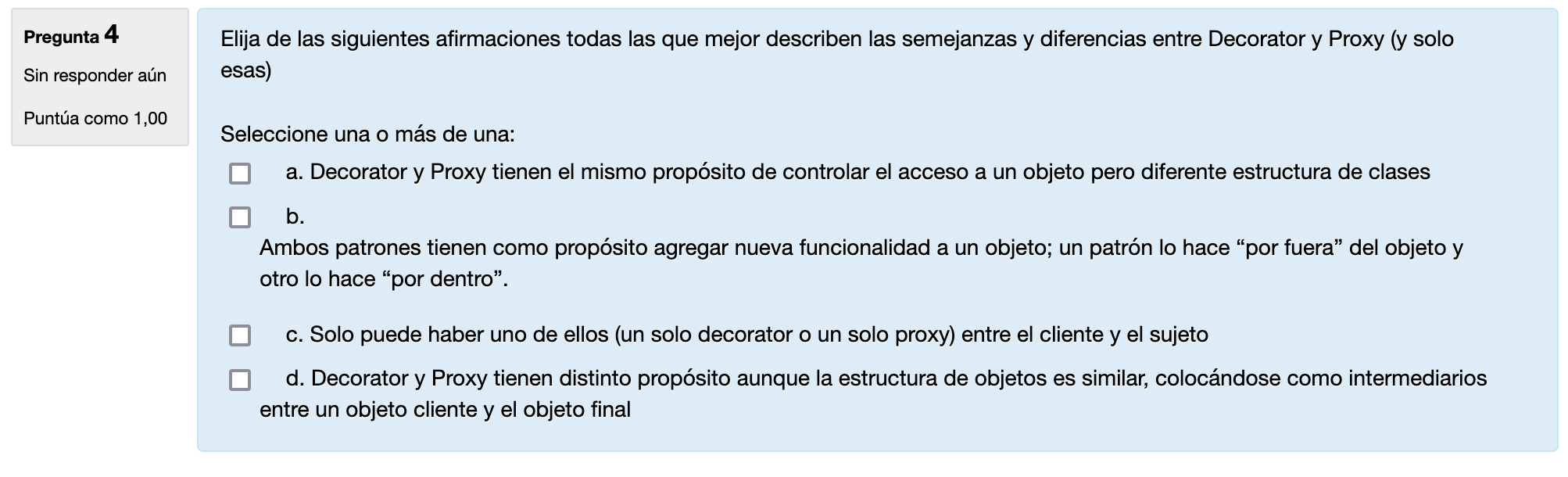


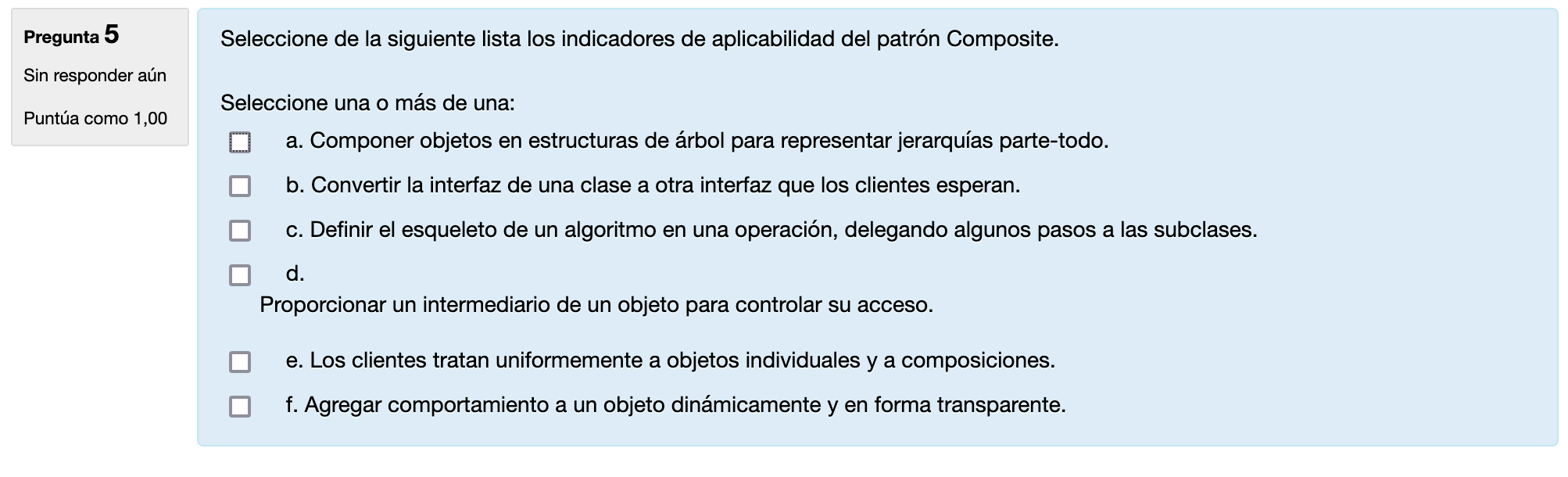
Creo que las correctas son la c y la e. La b me hace dudar porque es falsa, no se si los frozenspots están ocultos o si es porque no se pueden modificar en caja blanca.

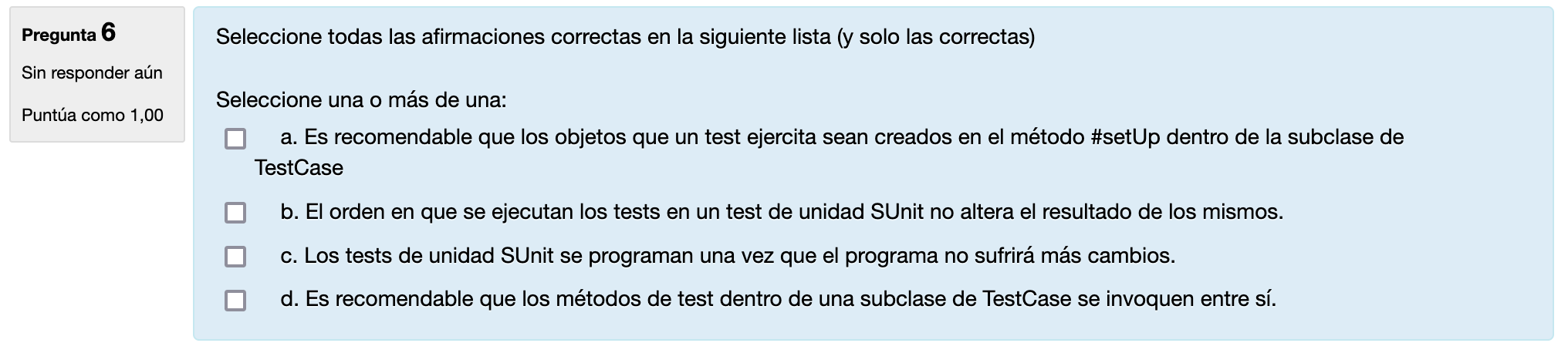


Las correctas son la a y d (en composicion se puede cambiar en ejecucion).

la correcta es la c.

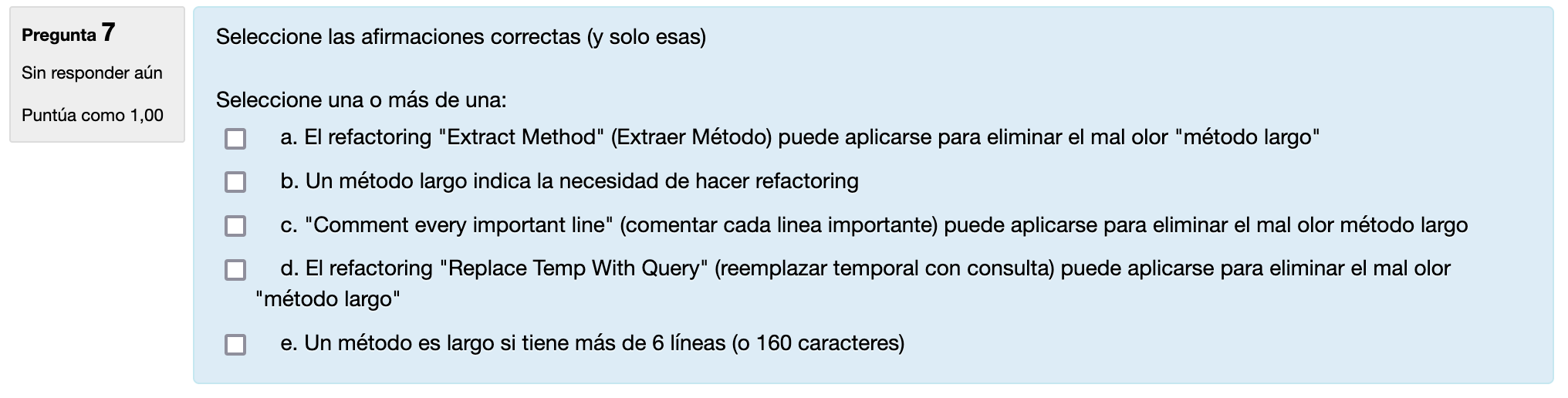
La correcta es la d. La c parece correcta pero podria haber varios decorators entre el cliente el sujeto, mientras que en el proxy no.

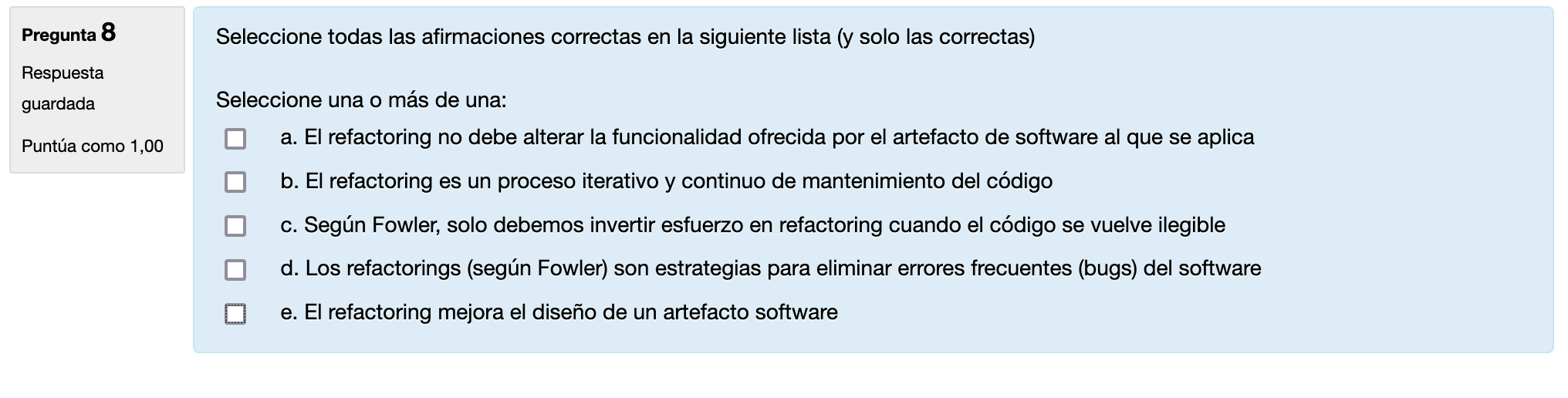
La correcta es la a y la e.

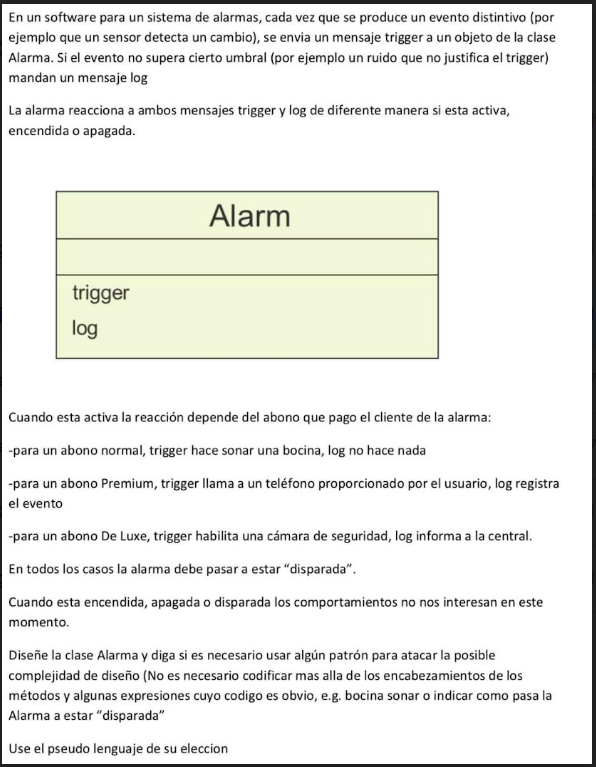


b (no deberia haber acoplamiento entre tests).

me hacen dudar la a y la d.

a, b, d. 

a, d y e. La b no estoy segura.



Para resolver este ejercicio se puede usar el patron State el sirve para extraer el estado de un objeto y que este sea transparente para el cliente.

Alarm >> event

pasoUmbralDeRiesgo ifTrue:[ myAlarmState trigger].

myAlarmState log.

myAlarmState: myAlarmState newState.

myAlarmState typeClientStrategy: myTypeClient.

Alarm >> myAlarmState: anAlarmState.

myAlarmState:=anAlarmState.

AlarmState >> typeClientStrategy: anTypeClient

typeClientStrategy:= anTypeClient.

AlarmState >> trigger

^subClassResponsability.

AlarmState >> log

^subClassResponsability.

AlarmStateActived >> trigger

typeClientStrategy trigger.

AlarmStateTrigger >> trigger

no hace nada

AlarmStateOn >> trigger

no hace nada

AlarmStateOff >> trigger

no hace nada

AlarmStateActived >> newState

^ AlarmStateTrigger new: myAlarm.

ClientAbonoNormal >> trigger

myAlarm sonarBocina.

ClientAbonoNormal >> log

no hace nada.

ClientAbonoPremium >> trigger

myAlarm llamarTelefono.

ClientAbonoPremium >> log

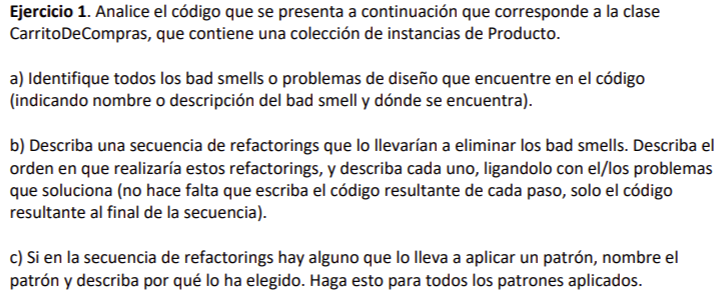
myAlarm registrarLog.

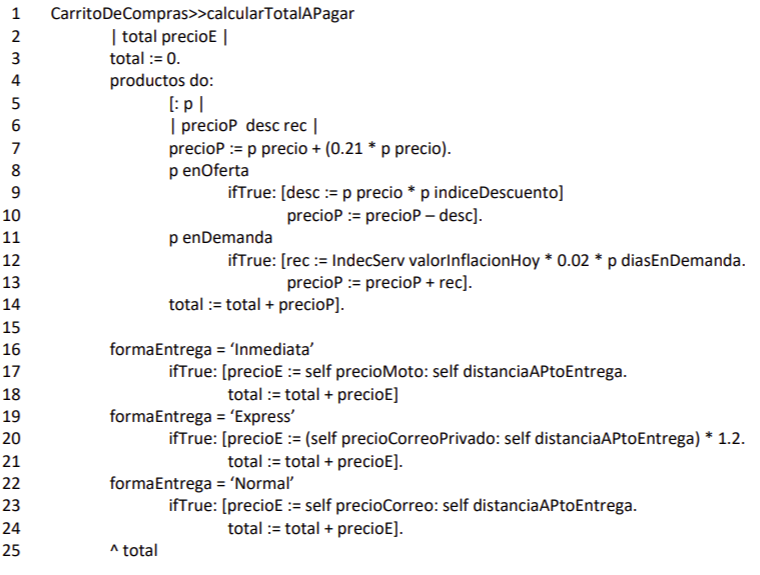
ClienteAbonoDeLuxe >> trigger

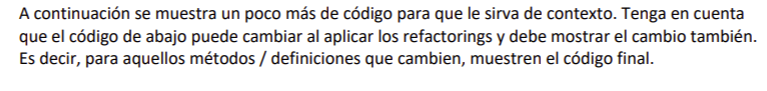
myAlarm habilitarCamara.

ClienteAbonoDeLuxe>> log

myAlarm llamarCentral.







1. Metodo largo: el metodo calcularTotalAPagar es demasiado largo, esto hace que se vuelva ilegible, dificil de mantener y poco reutilizable.

Envidia de atributos: existe envidia de atributos dado que la clase CarritoDeCompras le pide a Producto muchos de sus atributos para realizar cuentas que deberia hacerlo este ultimo.

Variable temporales redundantes: hay muchas variables temporales que podrian no

estar y bastaria con una consulta.

Switch Statement: se tiene una estructura de condicionales que pregunta por los diferentes tipos de entregas que existen, esto puede causar que, al agregar un nuevo tipo, se generen errores dado que se tiene que propagar el cambio en todos los lugares donde se encuentre esta estructura.

Pasa lo mismo con el producto que tiene dos caminos a tomar dependiendo si esta en demanda o en oferta.

Pasaria lo mismo si un producto no pudiese estar bajo demanda y en oferta al mismo tiempo.

Codigo duplicado: existen varias lineas de codigo duplicado, como por ejemplo las lineas 18,21 y 24.

Mal nombre de variable: se tienen nombres de variables que no son autoexplicativos y pueden llevar a confundir o dificultar el entendimiento del codigo. Por ejemplo, la linea 5 donde la variable se declara como “p” en vez de llamarle “producto” que es mas entendible.

1. 1. Como primer refactoring se cambia el nombre de las variables que no se entienden como “p” por “producto”, “precioE” por “precioEntrega”, “precioP” por “precioProducto”. De este modo se arregla el bad smell de mal nombre de variable.

2. Luego para quitar las variables temporales redundantes extraigo la parte condicional de la declaración pasándola a un método con el nombre adecuado y dejando la delegacion de los metodos. Este nuevo metodo no tendra variables sino que devolvera el descuento que corresponda, es como un replace temp with query solo que se encarga de chequear si esta en oferta o demanda segun toque.

“aplicarIVA”, “calcularDescuento” y “calcularRec”(no se que significa rec, si supiera le podria otro nombre).

Por ultimo hago un extract method desde la linea 6 hasta la 13 para dejarlo en un metodo nuevo llamado “calcularPrecioProducto” (usando los nuevos metodos) y en calcularTotalAPagar realizo la delegacion.

4. Como se genera envidia de atributos, realizo move method y paso “calcularPrecioProducto” a la clase “Producto” cambiandole el nombre a “calcularPrecio”.

5. Otra vez aplico un replace temp with query para la suma del total de los productos y asi eliminar la variable temporal total.

6. Para eliminar el codigo duplicado paso el seteo del total mas el precioEntrega afuera del if.

7. Por ultimo para eliminar el switch statement de entregas se puede aplicar el refactoring replace conditional with polimorphism.

Se crea la clase CalculoDePrecioEntregasStrategy. Se realiza un extract method con nombre “calcularPrecio” y un move method de la parte condicional de las entregas (junto a todos sus metodos de precioMoto, precioCorreo, etc).

Y se realiza un replace condicional with polimorphism generando subclases que se encarguen de la parte del condicional que corresponde.

Ademas genero metodos en Producto para setear la estrategia.

En el metodo de calcularTotalAPagar genero una delegacion a la estrategia.

CarritoDeCompras>> calcularTotalAPagar.

^calcularPrecioProductos + entregaPrecio calcular.

CarritoDeCompras>> calcularPrecioProductos

^productos sumNumbers:[ :producto | producto calcularPrecio].

CarritoDeCompras>> entregaPrecio: anCalculoDePrecioEntregasStrategy

entregaPrecio: = anCalculoDePrecioEntregasStrategy.

Producto>> calcularPrecio

^precio + self aplicarIVA - self calcularDescuento + self calcularRec.

Producto>> calcularDescuento

^enOferta ifTrue:[^(precio \* indiceDescuento)].

^0.

Producto>> calcularRec

^enDemanda ifTrue:[^(IndecServ valorInflacionHoy \*0.02 \* diasEnDemanda)].

^0.

Producto>> aplicarIVA

^precio \* 0.21.

CalculoDePrecioEntregasStrategy>> calcularPrecio

^subclassResponsability.

CalculoDePrecioEntregasStrategyInmediata>> calcularPrecio

^self precioMoto: self distanciaAPtoEntrega

CalculoDePrecioEntregasStrategyExpress>> calcularPrecio

^(self precioCorreoPrivado: self distanciaAPtoEntrega) \* 1.2.

CalculoDePrecioEntregasStrategyNormal>> calcularPrecio

^self precioCorreo: self distanciaAPtoEntrega.

Object subclass: #Producto

instanceVariableNames: 'nombre precio enOferta enDemanda indiceDescuento

diasEnDemanda'

Producto>>initializeNombre: aString precio: aNumber

nombre := aString.

precio := aNumber.

enOferta := false.

enDemanda := false.

Producto>>setEnOfertaConIndice: aNumber

enOferta := true.

indiceDescuento := aNumber

TestCaseCarritoDeCompras>>setUp

| prod |

carrito := CarritoDeCompras new.

prod := Producto newNombre: ‘Guitarra Fender Stratocaster American’ precio: 203890.

prod setEnOfertaIndice: 0.15.

carrito addProduct: prod.

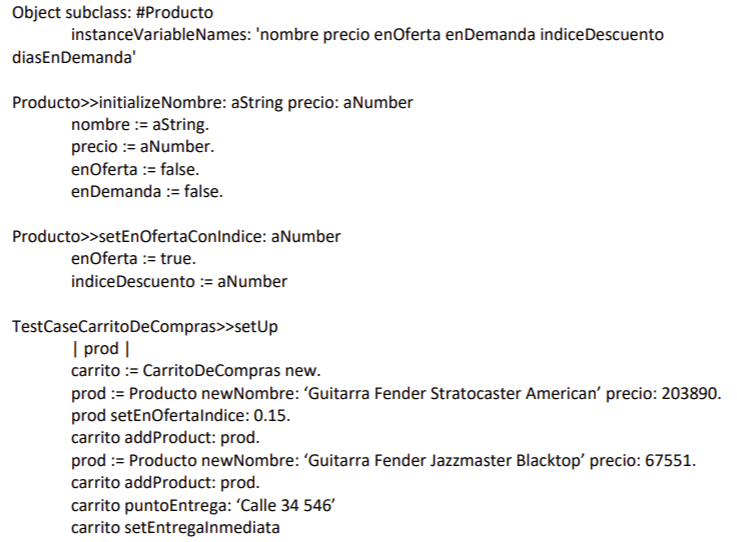
prod := Producto newNombre: ‘Guitarra Fender Jazzmaster Blacktop’ precio: 67551.

carrito addProduct: prod.

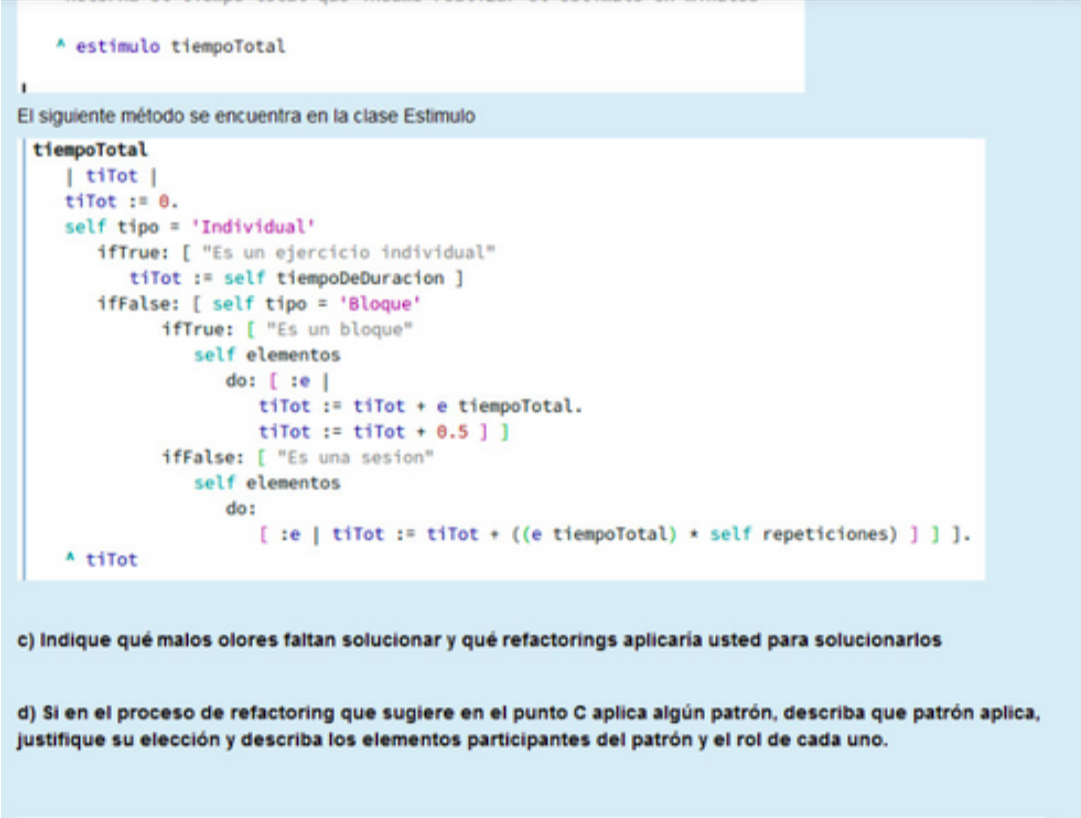
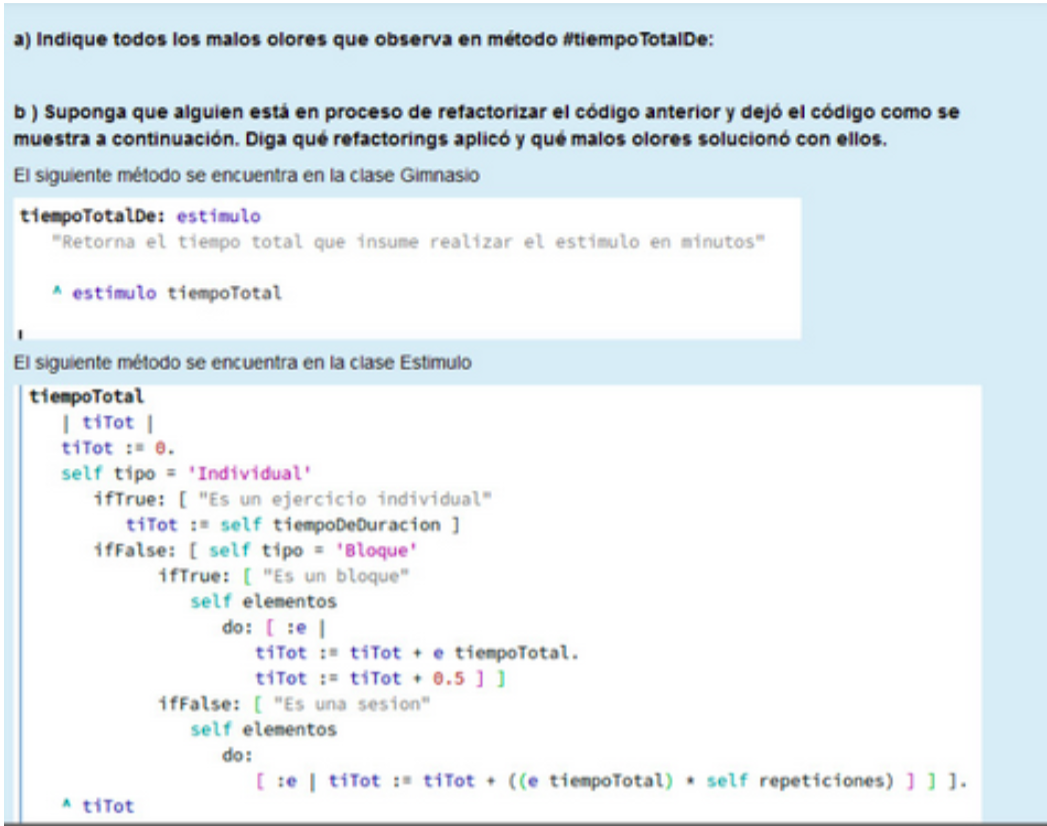
carrito entregaPrecio: CalculoDePrecioEntregasStrategyInmediata puntoEntrega: entregaPrecio puntoEntrega: ‘Calle 34 546’

No se que hace puntoEntrega pero como tiene que ver con las entregas y su calculo, entonces se lo dejo en la estrategia.

1. En la secuencia de refactorings se aplico el patron Strategy. Este lo elegi dado que se tienen diferentes algoritmos de calculo de precios de entrega segun el tipo de entrega que se elija. Por esto, dado que el objetivo del strategy es extraer las distintas algoritmias de un objeto de modo que se pueda cambiar segun lo que la situacion requiera, me parecio adecuado aplicarlo en este refactoring.



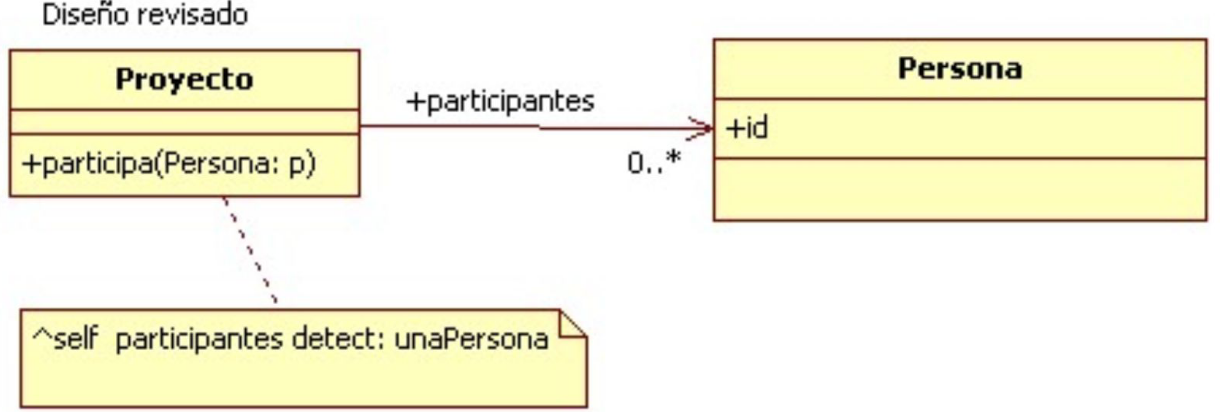




Ejercicios de refactoring

**Indicar qué cambio se realizó entre los dos diseños y si fue apropiado. Justificar**





El cambio que se realizo fue una refactorizacion aplicando move method dado que existia una envidia de atributos por parte de Persona.

Entonces, se movio el metodo a la clase Proyecto para evitar este bad smell.

Considero que fue apropiado en el sentido de que soluciona la envidia de atributos y mala asignacion de responsabilidad, sin embargo Persona queda animica, pero hago la suposicion de que es a favor del ejercicio.

**Analice el siguiente código, indicando que defectos encuentra y como pueden corregirse**

