Problemas de Patrones de Diseño (v1)

Problema 1.

Desde la versión 1.0 de Java existe la interfaz java.util.Enumeration cuya funcionalidad fue duplicada por java.util.Iterator, que apareció en la versión 1.2
Una de las clases que implementa esta interfaz (concretarmente Enumeration<Object>) es StringTokenizer. Lo que se pide es:

a) Aplicad el patrón necesario para que StringTokenizer pueda usarse en el contexto de necesitar un Iterator<Object>. Recordad que la operación remove de los iteradores es optativa (leed la documentación de Java). Si se os ocurre más de una forma de hacerlo, mejor.
b) ¿Y si lo hacemos al revés? ¿Cómo adaptaríais Iterator para clientes que precisen de un Enumeration?

Problema 2.

Los patrones también permiten entender mejor algunas de las clases predefinidas de la biblioteca de Java. Por ejemplo, el patrón adaptador puede usarse para analizar los usos de:

a) En el API de colecciones: java.util.AbstractCollection y java.util.AbstractList
b) En el API de swing java.swing.table.AbstractTableModel y java.awt.event.MouseAdapter
c) En el API de proceso de XML, org.xml.sax.helpers.DefaultHandler

¿Qué problema común resuelven estas clases? ¿Puede extraerse una "idea" de cuando utilizar clases similares a las comentadas en nuestros diseños?

Problema 3.

Como entrenamiento previo al patrón Composite, resolveremos un problema sencillo de programación sobre árboles binarios.
Lo que se pide es:

a) Implementad, de forma sencilla, árboles binarios que en sus nodos contengan números enteros.
b) Definid un método recursivo para sumar el valor de todos los números que contiene, con un método sin parámetros que devuelva un entero
c) Definid la clase de iterador asociado e implementad un método que reciba un iterador como parámetro y sume todos los valores

encontrados en el recorrido. Dicho método puede estar fuera de la clase definida para los árboles binarios.

d) Ahora definid el método de suma pero como un método void con un parámetro que simule un entero pasado por referencia. Para este entero deberéis crear una clase auxiliar.
e) En el apartado anterior la única responsabilidad del objeto que se pasa como parámetro es almacenar un entero, mientras que la clase árbol es la que es responsable de actualizarlo convenientemente. ¿Qué pasa si la responsabilidad de actualizar el valor la pasamos al objeto que pasamos como parámetro?

Problema 4.

En un programa de gestión de proyectos, se quiere llevar cuenta del coste y la duración de las tareas que constituyen un proyecto.
Para ello las tareas tendrán definidas las operaciones:

- public Money costInEuros()
- public int durationInDays()

Las tareas más sencillas, pueden realizarse directamente (sin descomponerse en tareas más simples) y de ellas se dispone de su coste en euros y de su duración en días.
Las tareas que no pueden llevarse a cabo directamente han de descomponerse en subtareas, y su coste es la suma de los costes de las subtareas en que se descompone. El método:

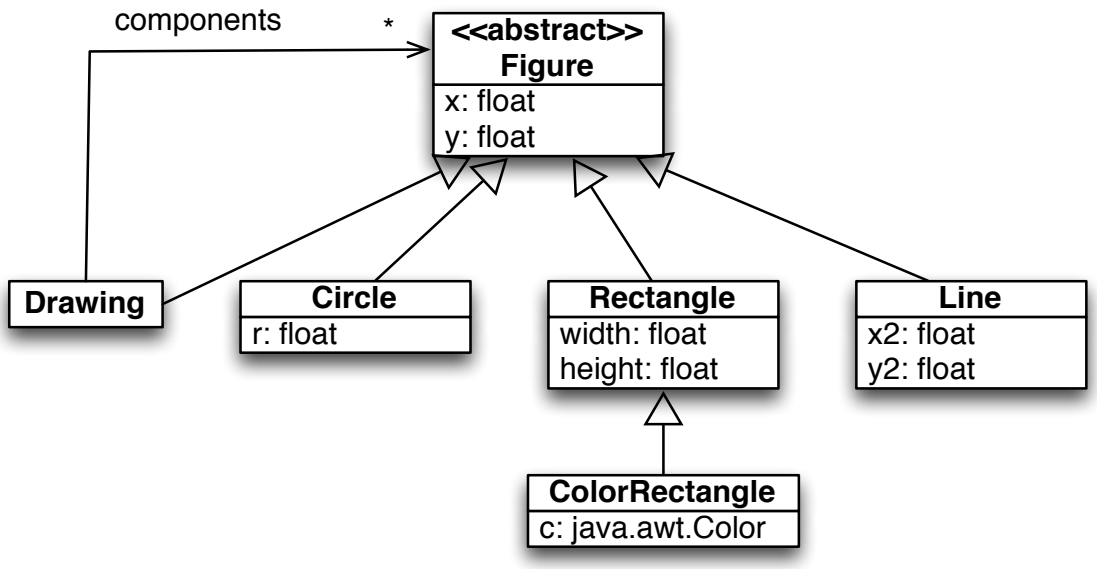
- void addSubTask(task)

sirve para añadir una tarea a otra.
Además existen dos maneras de descomponer un trabajo: secuencial y paralela, lo que afecta a la forma de calcular la duración de una tarea compuesta.
Lo que se pide es:

- Aplicad el patrón composite para resolver el problema
- ¿Podéis definir el método addSubTask para las tareas simples? ¿Qué ventajas/desventajas tendría?

Problema 5.

Dadas las siguientes clases que representan figuras, en las que solamente hemos mostrado los atributos (privados) definidos en cada una de ellas:



Lo que se pide es:

- Declarad todas las clases implicadas
- Implementad sus constructores
- Añadid a la clase Drawing un método para añadir subfiguras a una figura dada
- Añadid una operación abstracta en la clase Figure denominada copy, tal que al aplicarla sobre una figura, devuelve una copia de la misma
    - public void Figura copy()
- Implementad esta operación en todas las clases implicadas. Razonad sobre las condiciones para decidirse entre:
    - copia superficial
    - profunda

PISTA: Constructores copia.

Problema 6.

Una forma de hacer que cada vez que se añade un producto a la venta actual se muestre por pantalla su nombre, precio unitario y el total acumulado, es que la capa presentación pueda registrar observadores en las instancias de la clase Venta. Así, cuando alguien añade una línea al la Venta, el observador es notificado de este hecho.
El código de partida es:

```
1 public class Sale {
2   private List<SLI> lines = new ArrayList<>();
3   public void createSalesLineItem(
4       ProductDescription desc, int quantity) {
5     SLI sli = new SLI(desc, quantity);
```

```
 6     lines.add(sli);
 7   }
 8   //...
 9 }
10
11 public class SLI { // SLI = SalesNewItem
12   private ProductDescription desc;
13   private int quantity;
14   public VLP(ProductDescription desc, int quantity) {
15     this.desc = desc;
16     this.quantity = quantity;
17   }
18   public int subTotal() {
19     return desc.price() * quantity;
20   }
21 }
22
23 public class ProductDescription {
24   private String description;
25   private int price;
26   public String getDescription() {
27     return description;
28   }
29   public int getPrice() {
30     return price;
31   }
32   //...
33 }
34
35 public class Register {
36   private Store tenda;
37   private Sale currentSale;
38   public Register(Store store) {
39     this.store = store;
40   }
41   public void createNewSale() {
42     currentSale = new Sale();
43   }
44   //...
45 }
```

Modificad el código anterior para aplicar el patrón indicado (p.e. añadid, modificad clases, interfaces, métodos, etc.) utilizando:

- Variante pull del patrón, en las que los observadores demandan datos

- Variante push del observador en la que el observador es notificado de los datos necesarios

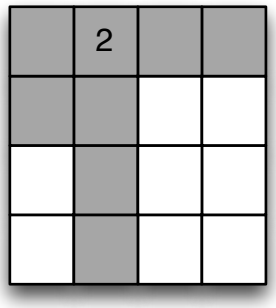
Implementad un observador concreto que escriba en un fichero de texto la información dada (a modo de fichero de log).

Problema 7.

Las "reglas" para resolver un sudoku son bien simples:

- Cada celda puede tener un valor dentro de un dominio finito
- Fijar el valor de una celda, restringe los valores posibles para las celdas de su misma fila, columna y grupo.

Por ejemplo, las celdas que pueden obtener valores del 1 al 4, pueden organizarse en 4 grupos 2x2 de manera que fijar el valor 2 de las celda mostrada en la imagen, imposibilita ese valor para las celdas sombreadas:



Cuando las celdas están en una disposición tabular, los mismos arrays de Java nos sirven para representar la geometría de las restricciones. Para casos más complejos, podemos usar una infraestructura más genérica. La idea del diseño es la siguiente:

- Unas celdas observan a las celdas de las que pueden restringir su valor y, a su vez, están observadas por éstas
- La clase Cell tiene un método void fixValue(int value) que intentará fijar el valor de esa celda a ese valor
  - Si el valor no es uno de los que están permitidos para esa celda, lanzará la excepción IncorrectValue
  - Si todo ha ido bien, además de fijar el valor, la celda notificará a todas las celdas interesadas que ya no pueden usar ese valor. Al recibir esa notificación, la celda:
    - Eliminará ese valor como posible
    - En caso de que el conjunto de valores posibles de la celda quede vacío, lanzará la excepción IncorrectValue
    - Si solamente quedara un valor para esa celda, usaría dicho valor para la celda, lo que provocaría nuevas notificaciones.

Se pide:

a) Diseñar la solución usando el patrón observador (podéis escoger entre push y pull)
b) Implementad en Java el diseño obtenido. Podéis hacer pruebas para comprobar su funcionamiento
c) (Opcional) Cread el código adicional para usar esas clases para resolver sudokus. Pista: necesitaréis poder copiar el estado de todas las celdas al hacer backtracking; para ello podéis usar un Map de celdas a listas de enteros, ya que el estado de la búsqueda puede caracterizarse por los valores que quedan aún como posibles para las celdas.

Problema 8.

En el Problema 4 es posible que la interfaz del sistema, a medida que se van finalizando las tareas, vaya mostrando dicha información.
Para ello, añadiréis a las tareas simples los métodos

- `public void finish()`
- `public boolean hasFinished()`

El método `hasFinished` también se aplica a las tareas compuestas, y una tarea compuesta está finalizada, si todos sus componentes lo están.
Añadid la posibilidad de añadir observadores a las tareas de manera que se les notifique la finalización de las tareas.

¿Qué problema tenéis con las tareas compuestas? ¿Cómo lo resolveríais?

Problema 9.

(Posible idea a explorar) En el Problema 7 tenemos notificaciones en cascada, es decir, que una notificación puede generar una avalancha de notificaciones (cuando al fijar el valor de una celda, una de las celdas dependientes, fija su valor al tener solamente un valor posible).
En casos como éste, a veces es interesante centralizar las notificaciones usando una cola de notificaciones pendientes y un bucle que vaya ejecutándolas una detrás de otra. La clase encargada de gestionar esta cola típicamente es candidata a ser un singleton[1].
En nuestro caso, los datos a guardar para cada petición serán:

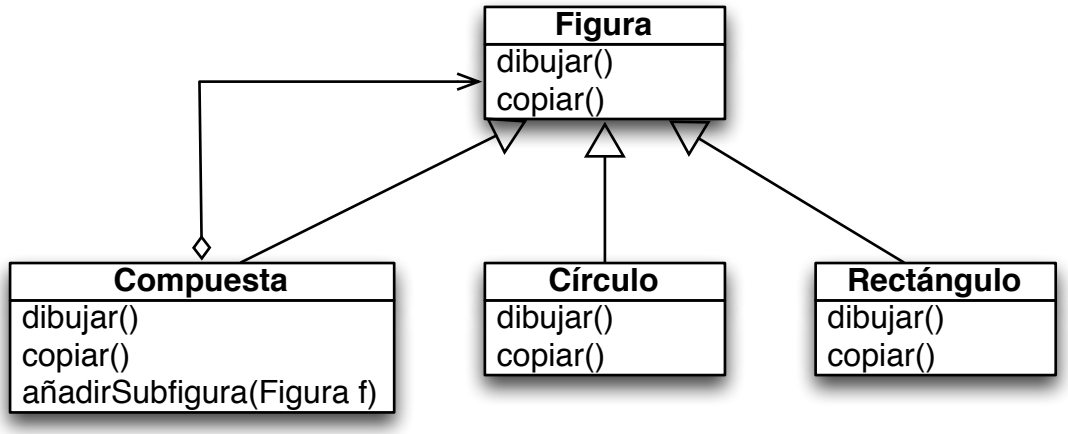
- La celda a notificar
- El valor a eliminar como posible

[1] Patrones como Command permiten, a su vez, usar la misma cola para diferentes tipos de notificaciones.

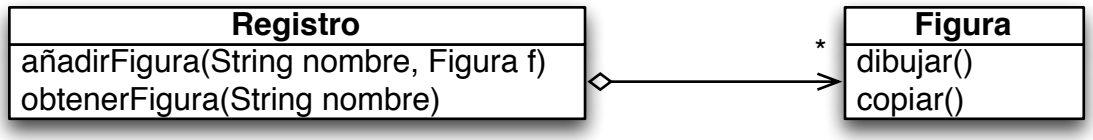
Cuando una celda fija su valor, añade todas las notificaciones a la cola. La notificación inicial es la que, además, provoca el proceso de la cola hasta que no queden notificaciones pendientes. Si alguna de las notificaciones pendientes lanzara una excepción, ésta se propagaría hasta la primera de las celdas que fijó su valor (ya que esta primera llamada sigue esperando a que se complete el vaciado de la cola de peticiones).
Otra posibilidad podría ser agrupar las peticiones pendientes por celda, de manera que si hay pendientes varias peticiones de eliminar valores posibles para una misma celda, se envíen de golpe. Sería interesante usar una tipo de cola por prioridades, en la que se extrajera de la cola la celda con la lista de valores a eliminar más larga.

Problema 10.

En nuestro programa de dibujo tenemos la siguiente jerarquía de figuras



Como el programa permite poner nombres a figuras que disponemos en pantalla, nos interesa que la clase que opera ese registro sea única. Para ello aplicaremos el patrón adecuado ☺



Se pide que implementéis la clase Registro, de manera que éste sea único.

Problema 11.

Una de las aplicaciones más comunes del patrón builder en Java es simular los parámetros optativos con nombre en los constructores y evitar lo que se conoce como constructores telescópicos.

Por ejemplo, en la clase siguiente:

```
 1 public class NutritionFacts {
 2   private final int servingSize; // (mL)    required
 3   private final int servings;    // (units) required
 4   private final int calories;    //         optional
 5   private final int fat;         // (g)     optional
 6   private final int sodium;      // (mg)    optional
 7   private final int carbohydrate;// (g)     optional
 8
 9   public NutritionFacts(int servingSize, int servings) {
10     this(servingSize, servings, 0);
11   }
12   public NutritionFacts(int servingSize, int servings,
13                         int calories) {
14     this(servingSize, servings, calories, 0);
15   }
16   // ....
17   public NutritionFacts(int servingSize, int servings,
18                         int calories, int fat,
19                         int sodium, int carbohydrate) {
20     this.servingSize = servingSize;
21     this.servings    = servings;
22     // ...
23   }
24   // ...
25 }
```

El principal problema es que la invocación de tales constructores genera código tan inteligible como:

```
beverage = new NutritionFacts(240, 8, 100, 0, 35, 27);
```

¿A qué se refiere cada cosa?
Una solución es aplicar el patrón builder y permitir:

```
beverage = new NutritionFacts.Builder(240, 8)
               .calories(100)
               .sodium(35)
               .carbohydrates(27)
               .build();
```

Se pide que apliquéis builder para conseguir esta forma de construir los objetos de la clase NutritionFacts.

Problema 12.

Un compañero vuestro, en una práctica en la que tenía que aplicar el patrón builder, ha hecho el siguiente código:

```
 1 interface HouseBuilder {
 2   House buildHouse();
 3   Floor buildFloor();
 4   Walls buildWalls();
 5   Roof  buildRoof();
 6 }
 7
 8 public class WoodBuilder implements HouseBuilder {
 9   public House buildHouse() {
10     return new WoodHouse();
11   }
12   public Floor buildFloor() {
13     return new WoodFloor();
14   }
15   public Walls buildWalls() {
16     return new WoodWalls();
17   }
18   public Roof buildRoof() {
19     return new WoodRoof();
20   }
21 }
22
23 public class HouseMaker {
24   public House build(HouseBuilder builder) {
25     House house = builder.buildHouse();
26     house.serFloor(builder.buildFloor());
27     house.setWalls(builder.buildWalls());
28     house.setRoof(builder.buildRoof());
29     return house;
30   }
31 }
```

¿Le daríais algún consejo? ¿Qué patrón ha aplicado? Aplicad builder al ejemplo dado y comparad las soluciones.

Problema 13.

Inicialmente el código de nuestra aplicación consistía en las siguientes clases:

```
1 public class Employee {
2   private final String name;
```

```
  private final String adress;
  public Employee(String name, String address) {...}
  //...
}

public class FixedIncome extends Employee {
  private final long fixed;
  public FixedIncome(String name, String address,
                     long fixed) {...}
  //...
}

public class FixedIncomePlusCommissions
             extends Employee {
  private final double percent;
  //...
}
```

La aplicación inicial ha crecido y ahora tenemos otro criterio adicional para clasificar empleados:

- Staff, trabajador en plantilla
- Temporal, trabajador temporal

con sus sublclases de TemporalFixedIncome, etc.

Lo que se quiere es que haya partes del código que sean capaces de crear trabajadores con diferentes tipos de formas de salario, independientemente de si son trabajadores eventuales o en plantilla (por ejemplo el código que lee un fichero con los datos de cobro tanto del fichero de trabajadores en plantilla como de la de los temporales). ¿Qué patrón aplicaríais al problema? Mostrad el diagrama de clases y un esquema de implementación del código relevante.

Problema 14.

Retornemos al famoso ejemplo del Terminal Punto de Venta del Problema 6.
Suponed que tenemos diferentes subclases de SalesLineItem (SLI) que calculan de forma diferente el subtotal, por ejemplo:

- La forma normal, que es la que se usa ahora
- Compre 3, pague 2
- La segunda unidad al 50%

Suponed que, además, el tipo de línea de producto a incorporar a una venta depende del producto (ProductDescription).
¿Qué patrón aplicaríais? Mostrad el diagrama de clases, diagrama de secuencia del funcionamiento y el código relevante en Java.

Problema 15.

Double Dispatch. En programación orientada a objetos, gracias al enlace dinámico, cuando invocamos obj.operation(), el método que ejecutamos depende de implementación de la operación (método) en el tipo dinámico de obj (la clase del objeto referenciado por obj).
Esto va muy bien cuando la decisión sobre qué ejecutar depende solamente del tipo del objeto receptor.
¿Qué pasa cuando la decisión depende de los tipos de dos objetos? Es decir, cuando invocamos obj.operation(other) el método a invocar depende de los tipos dinámicos tanto de obj como de other.
El patrón DoubleDispatch soluciona el asunto delegando sobre una operación polimórfica sobre el objeto que se pasa como parámetro y que identifica el tipo del objeto receptor.
Veamos un ejemplo con dos subclases A y B, en las que la operación depende de los tipos concretos de A y B.

```
1 public interface Common {
2   public void operation(Common other);
3   public void operationWithA(Common other);
4   public void operationWithB(Common other);
5 }
6
7 public class A implements Common {
8   public void operation(Common other) {
9     other.operationWithA(this);
10  }
11  public void operationWithA(Common other) {
12    // Caso self=A, other=A
13  }
14  public void operationWithB(Common other) {
15    // Caso self=A, other=B
16  }
17 }
18 // La clase B sería similar a la A
```

Lo que se pide es:

a) Diagramas de secuencia para la interacción de los casos: anA.operation(otherA) y aB.operation(anA).

b) También las operaciones auxiliares pueden definirse con parámetros correspondientes a los tipos concretos, es decir, operationWithA(A other). ¿Qué implicaciones tiene hacerlo así?
c) Ampliando el punto anterior: usando sobrecarga, puede usarse el mismo nombre de operación y hacer operation(A other). ¿Qué os parece hacerlo de esta manera?

Problema 16.

En una aplicación para cálculos geométricos, se dispone de las siguientes subclases de GeometricElement, para representar elementos geométricos:

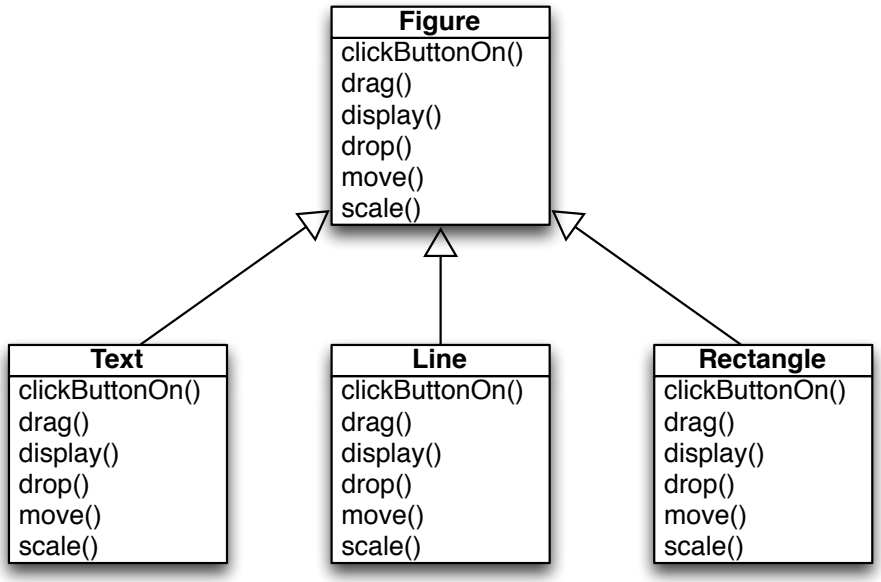
- Point(x, y), que representar el punto con coordenadas x e y
- Line(a, b), que representa la línea de puntos y = a * x + b
- VerticalLine(c), que representa la línea vertical x = c

Aplicad el patrón DoubleDispatch a la operación intersect, tal que elem1.intersect(elem2) devuelve la intersección de los elementos elem1 y elem2.
Consideraciones adicionales:

- Para simplificar la codificación podemos usar otro patrón, denominado NullObject, que permita representar la intersección vacía. Es decir, añadiremos la subclase NullPoint para representar las intersecciones vacías.
- Para hacer comparaciones, como trataremos con números en coma flotante, en vez de usar igualdad, supondremos que la clase GeometricElement tiene un método close(x1, x2) que nos dice si los valores x1 y x2 pueden considerarse suficientemente iguales.
- Tened en cuenta que la operación intersect es conmutativa
- ¿Qué diferencias conlleva el usar GeometricElements que sean inmutables o no?

Problema 17.

Inicialmente nuestro sistema tiene la siguiente jerarquía de figuras:



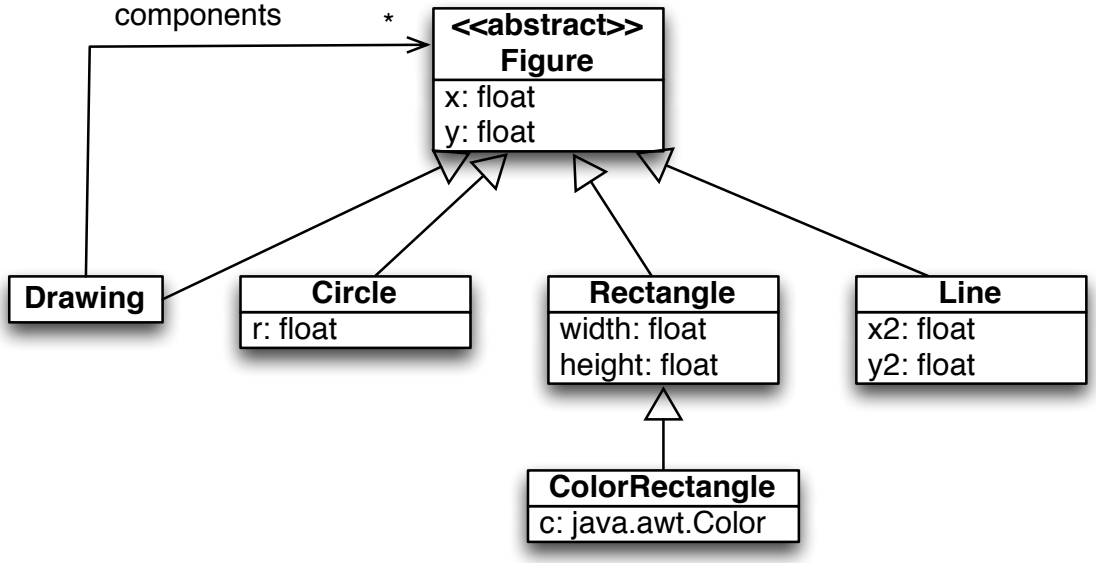
pero en una revisión de código, el consultor alertó de que el diseño de la jerarquía es problemático ya que no se respeta el principio de responsabilidad. Las clases, tiene dos responsabilidades diferentes:

- Una respecto a su visualización
- Otra respecto a cómo interactuar con ellas

Lo que se pide es separar ambas responsabilidades de manera que el impacto sobre los clientes de figuras sea mínimo.

Problema 18.

Dadas las siguientes clases que representan figuras, en las que solamente se muestran los atributos definidos en cada una de ellas:



Hay muchas operaciones que podríamos añadir sobre las clases de esta jerarquía Para poder añadirlas sin necesidad de ir modificando todas las clases de la jerarquía se ha pensado en usar el patrón Visitor.

Para ello:

- deberéis definir la clase FigureVisitor con los métodos visit pertinentes
- y añadir los métodos accept en las clase de la jerarquía
- para ver un caso concreto, se pide también que implementéis el Scaler, tal que, dada una figura, la escalará según el factor de escala dado. Las reglas de escalado son:
    - Escalar un círculo quiere decir multiplicar su radio por el factor
    - Escalar un rectángulo quiere decir multiplicar sus dimensiones por el factor
    - Escalar una línea quiere decir escalar sus diferencias con el origen. Por ejemplo, si la línea tiene x=3 y x2=5, una escala de 4 dejaría x2 a 3 + 4 * (5 – 3) = 11. La modificación de y2 sería similar..
    - Escalar un Drawing implica escalar todas las figuras de las que consta.

Notas:

- Obviamente necesitaremos definir getters y setters para los atributos ya que el Scaler deberá poder modificar la figura
    - Pensad también en una solución en la que las Figuras sean inmutables.
- En este caso el resultado de la visita queda plasmada en la propia instancia por lo que Scaler no hace falta que defina una operación para retornar el resultado.
    - Podéis pensar también en el caso en el que no modificáis la Figure que escaláis (en este caso el Scaler tendrá un método para obtener el resultado).
- Dependiendo de cómo apliquéis el patrón os puede quedar un método cuya implementación está vacía.
    - Esta manera evita tener que acceder a la lista de componentes de un dibujo.
    - Pero restringe las operaciones que pueden implementarse usando la infraestructura creada. ¿Por qué?

Un ejemplo de uso de la clase sería:

```
Figure figure = ...
...
FigureVisitor scale = new Scaler(2);
figure.accept(scale);
```

Problema 19.

Volvamos otra vez a las tareas del Problema 4. Como existen muchas más cosas que el coste y la duración que han de calcularse sobre las tareas, se ha decidido aplicar el patrón Visitor para poder añadir estos cálculos. Para ello:

- Definid la interfaz TaskVisitor
- Modificad la interfaz Task para poder usar los visitantes
- Declarad las clases CostVisitor y DurationVisitor
- Implementad CostVisitor.
- Implementad DurationVisitor. En este caso la situación es más compleja ya que el resultado de la visita de un subárbol depende de si el nodo raíz es un nodo secuencial o uno paralelo. **Pista**: usad una pila y permitid que desde el Visitor se pueda acceder a las subtareas de una tarea compuesta.
- Si se quiere impedir que se pueda acceder a las subtareas de una tarea compuesta, podéis añadir operaciones de preVisit y postVisit en los visitantes. De esta manera el bucle que recorre las subtareas se implementa dentro del accept de las tareas compuestas.