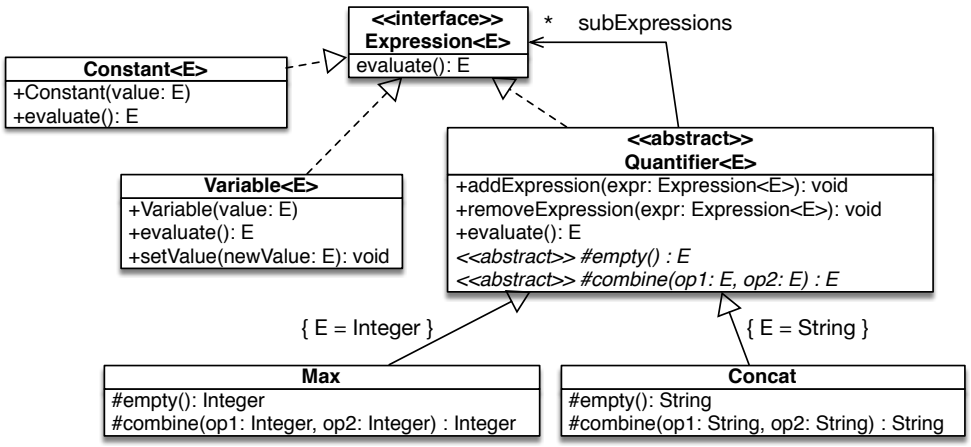

Este diseño permite representar expresiones (que evalúan a un tipo `E` genérico) que pueden ser constantes, variables o una serie de sub-expresiones unidas por un cuantificador.

En todos los apartados podéis suponer que las expresiones forman un árbol.

Se podrán definir diferentes cuantificadores, por ejemplo, para la suma de enteros, para el producto de enteros, para el máximo de enteros, para la concatenación de cadenas, ...

En la evaluación de un cuantificador, usaremos de dos operaciones que definen el comportamiento del mismo:

- La operación binaria (`combine`) que combina dos valores del resultado: la suma, el producto, el máximo, la concatenación, ...
- La operación (`empty`) que define el valor a retornar cuando el rango del cuantificador es vacío: `0` en el caso de la suma, `1` en el del producto, `Integer.MIN_VALUE` para el máximo, **“”** para la concatenación, ...

De hecho podemos ver cada cuantificador como la extensión de la operación binaria combine al caso general de cero o más parámetros.

Por ejemplo, las expresiones que podríamos definir y evaluar, usando **C** para las constantes, **V** para las variables, **Σ** para el cuantificador suma y **Π** para el producto:

$$\Sigma\big(C(4), \Pi(), \Pi(V(2), C(3)), \Sigma(V(3), V(2))\big) = 4 + 1 + (2 * 3) + (3 + 2) = 4 + 1 + 6 + 5 = 16$$

A. **(6 puntos)** Como las variables pueden cambiar de valor, se desea poder **observar** los cambios de valores de las expresiones de manera que:

   - Solamente se notifique en casos en que realmente se varía el resultado de la evaluación. Por ejemplo Max(V(3), C(7)) no ha de notificar nada si el valor de la variable pasa de 3 a 5.
   - Al notificar se envía información adicional, es decir, se desea que apliquéis la **versión push** del patrón observador. El objeto que se enviará será de la clase:

```
public class ValueChanged<E> {
    private final E oldValue;
    private final E newValue;
    public ValueChanged(E oldValue, E newValue) {
        this.oldValue = oldValue; this.newValue = newValue;
    }
    public E getOldValue() { return this.oldValue; }
    public E getNewValue() { return this.newValue; }
}
```

Mostrad el diagrama de clases resultante y la implementación de totas las clases e interfaces involucradas. **Si usáis `Observer` y `Observable` de las clases de Java, no hace falta que las implementéis.**

B. (4 puntos) Finalmente, modificad el diseño original, de manera que la clase `Quantifier<E>` use el **patrón estrategia** para acomodar las diferentes operaciones. Implementad las nuevas clases y/o interfaces que necesitéis para aplicar el patrón, la nueva clase `Quantifier<E>` y la nueva clase `Max`.

Por si os es de utilidad, os recuerdo los métodos de la clase `Observable`:

- `public void addObserver(Observer o)`
- `public void deleteObserver(Observer o)`
- `public void deleteObservers()`
- `public int countObservers()`
- `public void notifyObservers()`
- `public void notifyObservers(Object arg)`
- `protected void setChanged()`
- `protected void clearChanged()`
- `public boolean hasChanged()`

Y de la interfaz `Observer`:

- `public void update(Observable o, Object arg)`

Y algunos métodos (ya sean propios o de alguna superinterfaz) de `List<E>`:

- `public boolean add(E e)`
- `public boolean contains(Object o)`
- `public E get(int index)`
- `public Iterator<E> interator()`
- `public boolean remove(Object o)`
- `public int size()`