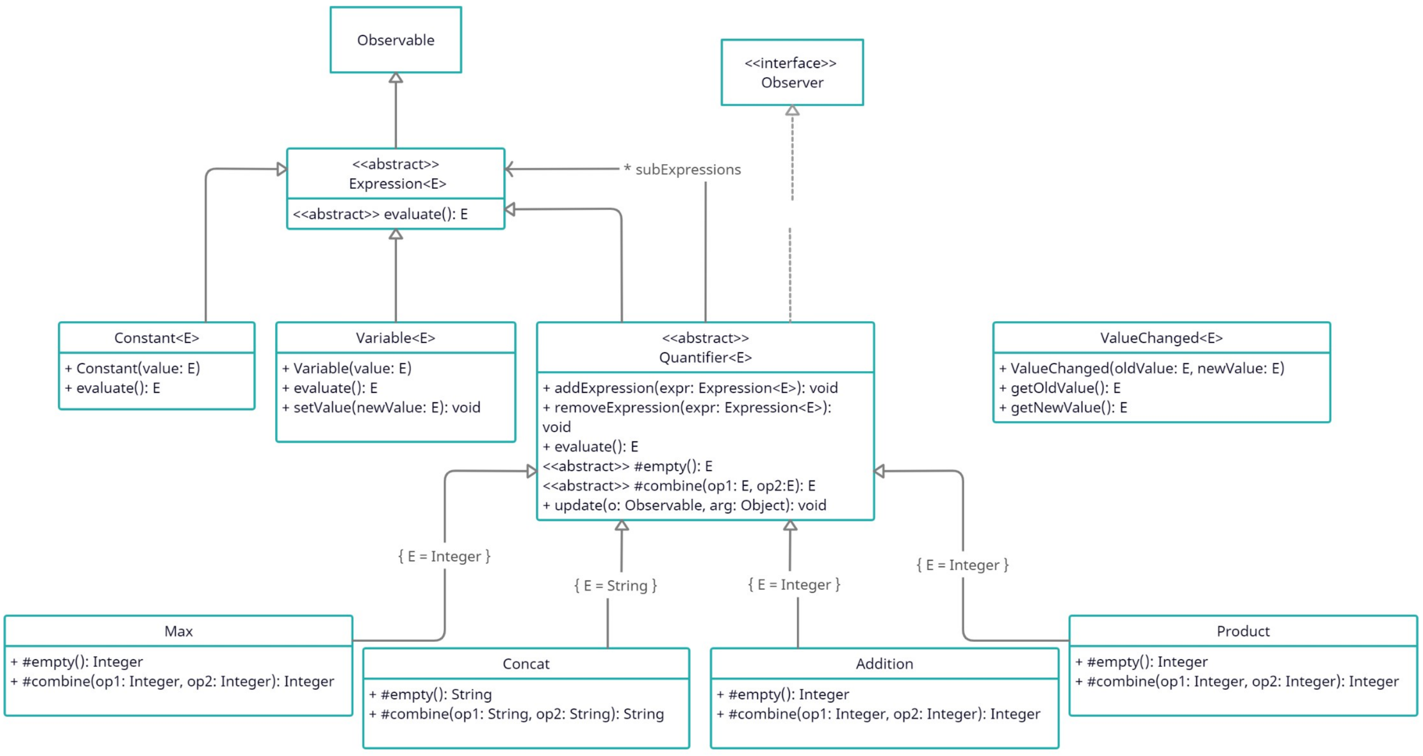
Activitat 4 – Exercici de disseny

Joel Aumedes – 48051307Y

Joel Farré - 78103400T

Apartat A


Observable
<<interface>> Observer
<<abstract>> Expression<E>
<<abstract>> evaluate(): E
* subExpressions
Constant<E>
+ Constant(value: E)
+ evaluate(): E
Variable<E>
+ Variable(value: E)
+ evaluate(): E
+ setValue(newValue: E): void
<<abstract>> Quantifier<E>
+ addExpression(expr: Expression<E>): void
+ removeExpression(expr: Expression<E>): void
+ evaluate(): E
<<abstract>> #empty(): E
<<abstract>> #combine(op1: E, op2:E): E
+ update(o: Observable, arg: Object): void
ValueChanged<E>
+ ValueChanged(oldValue: E, newValue: E)
+ getOldValue(): E
+ getNewValue(): E
{ E = Integer }
{ E = String }
{ E = Integer }
{ E = Integer }
Max
+ #empty(): Integer
+ #combine(op1: Integer, op2: Integer): Integer
Concat
+ #empty(): String
+ #combine(op1: String, op2: String): String
Addition
+ #empty(): Integer
+ #combine(op1: Integer, op2: Integer): Integer
Product
+ #empty(): Integer
+ #combine(op1: Integer, op2: Integer): Integer

Expression

```
import java.util.Observable;

public abstract class Expression <E> extends Observable {
    abstract E evaluate();
}
```

Constant

```
public class Constant<E> extends Expression<E> {
    private final E value;

    public Constant(E value) {
        this.value = value;
    }

    @Override
    public E evaluate() {
        return value;
    }

}
```

Variable

```
public class Variable<E> extends Expression<E> {
    private E value;

    public Variable(E value) {
        this.value = value;
    }

    @Override
    public E evaluate() {
        return value;
    }

    public void setValue(E newValue) {
        ValueChanged<E> VC = new ValueChanged<>(this.value, newValue);
        if (!newValue.equals(this.value)) {
            notifyObservers(VC);
        }
        this.value = newValue;
    }

}
```

Quantifier

```
import java.util.ArrayList;
import java.util.List;
import java.util.Observable;
import java.util.Observer;

public abstract class Quantifier<E> extends Expression<E> implements
Observer {
    private final List<Expression<E>> expressions = new ArrayList<>();

    private E evaluatedValue = this.empty();

    public void addExpression(Expression<E> expr) {
        expressions.add(expr);
        expr.addObserver(this);
        if(!evaluatedValue.equals(this.evaluate())){
            evaluatedValue = this.evaluate();
            notifyObservers();
        }
    }
    public void removeExpression(Expression<E> expr) {
        expressions.remove(expr);
        expr.deleteObserver(this);
        if(!evaluatedValue.equals(this.evaluate())){
            evaluatedValue = this.evaluate();
            notifyObservers();
        }
    }

    @Override
    public E evaluate() {
        if (expressions.isEmpty()) {
            return this.empty();
        }
        if (expressions.size() == 1) {
            return expressions.get(0).evaluate();
        } else {
            E toReturn = combine(expressions.get(0).evaluate(),
expressions.get(1).evaluate());
            for (int i = 2; i < expressions.size(); i++) {
                toReturn = combine(toReturn,
expressions.get(i).evaluate());
            }
            return toReturn;
        }
    }

    protected abstract E empty();

    protected abstract E combine(E op1, E op2);

    public void update(Observable o, Object arg) {
        E newValue = this.evaluate();
        if (!this.evaluatedValue.equals(newValue)) {
            notifyObservers(arg);
        }
        evaluatedValue = newValue;
    }

}
```

ValueChanged

```
public class ValueChanged<E> {
    private final E oldValue;
    private final E newValue;

    ValueChanged(E oldValue, E newValue) {
        this.oldValue = oldValue; this.newValue = newValue;
    }

    public E getOldValue() { return this.oldValue; }

    public E getNewValue() { return this.newValue; }

}
```

Max

```java
public class Max extends Quantifier<Integer> {
    @Override
    protected Integer empty() {
        return Integer.MIN_VALUE;
    }

    @Override
    protected Integer combine(Integer op1, Integer op2) {
        if (op1 > op2) {
            return op1;
        } else {
            return op2;
        }
    }

}
```

Concat

```java
public class Concat extends Quantifier<String> {
    @Override
    protected String empty() {
        return "";
    }

    @Override
    protected String combine(String op1, String op2) {
        return op1 + op2;
    }

}
```

Product

```java
public class Product extends Quantifier<Integer> {
    @Override
    protected Integer empty() {
        return 1;
    }

    @Override
    protected Integer combine(Integer op1, Integer op2) {
        return op1 * op2;
    }

}
```

Addition

```
public class Addition extends Quantifier<Integer> {
    @Override
    protected Integer empty() {
        return 0;
    }

    @Override
    protected Integer combine(Integer op1, Integer op2) {
        return op1 + op2;
    }

}
```

Apartat B

Expression

```
import java.util.Observable;

public abstract class Expression <E> extends Observable {
    abstract E evaluate();
}
```

Constant

```
public class Constant<E> extends Expression<E> {
    private final E value;

    public Constant(E value) {
        this.value = value;
    }

    @Override
    public E evaluate() {
        return value;
    }

}
```

Variable

```
public class Variable<E> extends Expression<E> {
    private E value;

    public Variable(E value) {
        this.value = value;
    }

    @Override
    public E evaluate() {
        return value;
    }

    public void setValue(E newValue) {
        ValueChanged<E> VC = new ValueChanged<>(this.value, newValue);
        if (!newValue.equals(this.value)) {
            notifyObservers(VC);
        }
        this.value = newValue;
    }

}
```

Quantifier

```
import java.util.ArrayList;
import java.util.List;
import java.util.Observable;
import java.util.Observer;

public class Quantifier<E> extends Expression<E> implements Observer {
    private final List<Expression<E>> expressions = new ArrayList<>();
    private Operation<E> operator;
    private E evaluatedValue;

    public Quantifier(Operation<E> operator) {
        this.operator = operator;
        evaluatedValue = operator.empty();
    }

    public void setOperator(Operation<E> operator) {
        this.operator = operator;
    }

    public void addExpression(Expression<E> expr) {
        expressions.add(expr);
        expr.addObserver(this);
        evaluatedValue = this.evaluate();
    }
    public void removeExpression(Expression<E> expr) {
        expressions.remove(expr);
        expr.deleteObserver(this);
        evaluatedValue = this.evaluate();
    }

    @Override
    public E evaluate() {
        if (expressions.isEmpty()) {
            return operator.empty();
        }
        if (expressions.size() == 1) {
            return expressions.get(0).evaluate();
        } else {
            E toReturn = operator.combine(expressions.get(0).evaluate(),
expressions.get(1).evaluate());
            for (int i = 2; i < expressions.size(); i++) {
                toReturn = operator.combine(toReturn,
expressions.get(i).evaluate());
            }
            return toReturn;
        }
    }

    public void update(Observable o, Object arg) {
        E newValue = this.evaluate();
        if (!this.evaluatedValue.equals(newValue)) {
            notifyObservers(arg);
        }
        evaluatedValue = newValue;
    }
}
```

ValueChanged

```
public class ValueChanged<E> {
    private final E oldValue;
    private final E newValue;

    ValueChanged(E oldValue, E newValue) {
        this.oldValue = oldValue; this.newValue = newValue;
    }

    public E getOldValue() { return this.oldValue; }

    public E getNewValue() { return this.newValue; }

}
```

Max

```
public class Max implements Operation<Integer> {

    @Override
    public Integer empty() {
        return Integer.MIN_VALUE;
    }

    @Override
    public Integer combine(Integer op1, Integer op2) {
        if (op1 > op2) {
            return op1;
        } else {
            return op2;
        }
    }

}
```

Concat

```
public class Concat implements Operation<String> {

    @Override
    public String empty() {
        return "";
    }

    @Override
    public String combine(String op1, String op2) {
        return op1 + op2;
    }

}
```

Product

```
public class Product implements Operation<Integer> {
    @Override
    public Integer empty() {
        return 1;
    }

    @Override
    public Integer combine(Integer op1, Integer op2) {
        return op1 * op2;
    }

}
```

Addition

```
public class Addition implements Operation<Integer> {
    @Override
    public Integer empty() {
        return 0;
    }

    @Override
    public Integer combine(Integer op1, Integer op2) {
        return op1 + op2;
    }

}
```