Polymorphism and the Open/Closed Principle

The Open/Closed Principle

- A design principle
- Main Goal: Make code flexible
- Design the code
  - To be open for extension
    - It should be possible to extend the behavior of the code
  - To be and closed for modification
    - The code should be inviolable

The Open/Closed Principle – How?

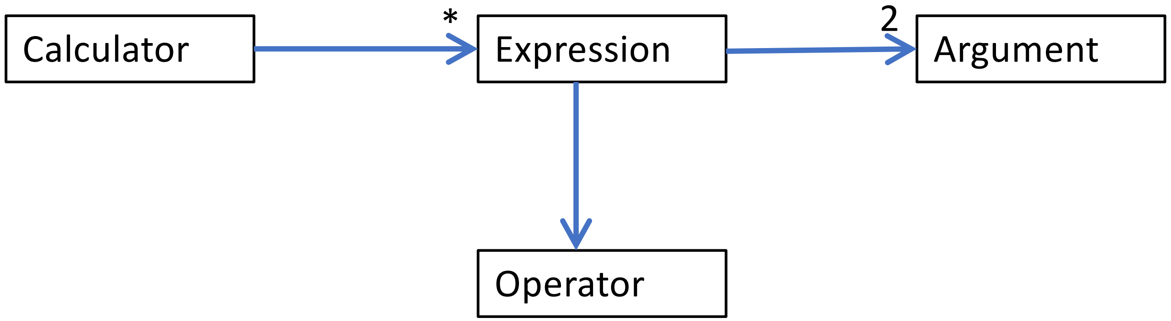
- Abstraction is the key

Example

- Simple calculator machine (consider only integer numbers):
  - 2 + 4
  - 2 / 4
  - 45 % 4
  - …
- Functionalities:
  - Add expression
  - Execute last expression
  - Show all expressions

Without Open/Closed (Java C – version)

- Domain model:



- What are the attributes and methods of these entities?

Without Open/Closed - Code

```
public class Expression {
  private Argument _arg1;
  private Argument _arg2;
  private Operator _operator;

  public Expression(Operator operator, Argument arg1, Argument arg2) {
    _arg1 = arg1;
    _arg2 = arg2;
    _operator = operator;
  }

  public int compute() {
    return _operator.evaluate(_arg1, _arg2);
  }

  public String toString() {
    return _arg1.toString() + " " + _operator + " " + _arg2;
  }
}
```

```
public class Argument {
  private int _value;

  public Argument(int v) {
    _value = v;
  }

  public int getValue() {
    return _value;
  }

  public String toString() {
    return "" + _value;
  }
}
```

Without Open/Closed - Code

```
public class Calculator {
 private List<Expression> _expressions = new ArrayList<>();

 public Calculadora() {
  _expressions = new ArrayList<>(); // or can initialize it here
 }

 public void add(Expression exp) {
  _expressions.add(exp);
 }

 public void computeAll() {
  for(Expression exp : _expressions) {
   int res = exp.compute();
   System.out.println("O valor da expressão \" " + exp + "\" é " + res);
  }
 }
 public void executeLastExpression() {
  System.out.println(exp.toString() + “ = “ +
                   _expressions.get(_expressions.size() - 1)).evaluate());
 }
}
```

```
public class Operator {
 private int _operatorType; // 0 -> +, 1 -> -, 2 -> *, 3 -> /
 private final static String[] OPERATION={“+”, “-”, “*”, “/”};

 public Operator(int type) {  _operatorType = type; }

 public int evaluate(Argument arg1, Argument arg2) {
  switch(_operatorType) {
  case 0:
   return arg1.getValue() + arg2.getValue();
  case 1:
   return arg1.getValue() - arg2.getValue();
  case 2:
   return arg1.getValue() * arg2.getValue();
  case 3:
   return arg1.getValue() / arg2.getValue();
  }
  return 0;
 }
 public String toString() {
  return OPERATION[_operatorType];
 }
}
```

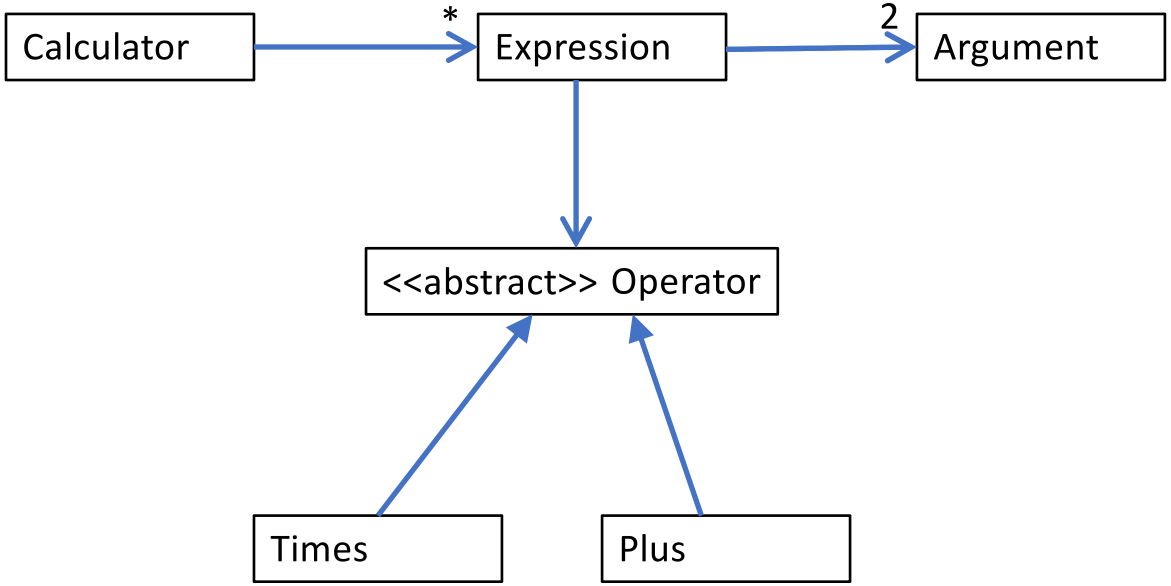
Main Problem with this Solution?

- Does not obey to the Open/Closed Principle

- Operator it is not an abstraction

- Extend the application to support more operation types
  - Implies modifications in the code

Better Solution



Better Solution - Code

```
public abstract class Operator {
  public abstract int evaluate(Argument arg1, Argument arg2);
  public abstract String toString();
}
```

```
public class Plus extends Operator {
  public int evaluate(Argument arg1, Argument arg2) {
    return arg1.getValue() + arg2.getValue();
  }

  public String toString() {
    return "+";
  }
}
```

```
public class Divide extends Operator {
  public int evaluate(Argument arg1, Argument arg2) {
    return arg1.getValue() / arg2.getValue();
  }

  public String toString() {
    return "/";
  }
}
```

Better Solution and Open/Closed Principle

- Now, new operations do not imply modifications to the existing code

- Each operation is represented by a subclass of **Operator**

- Support a new operation -> Implement a new subclass of Operator

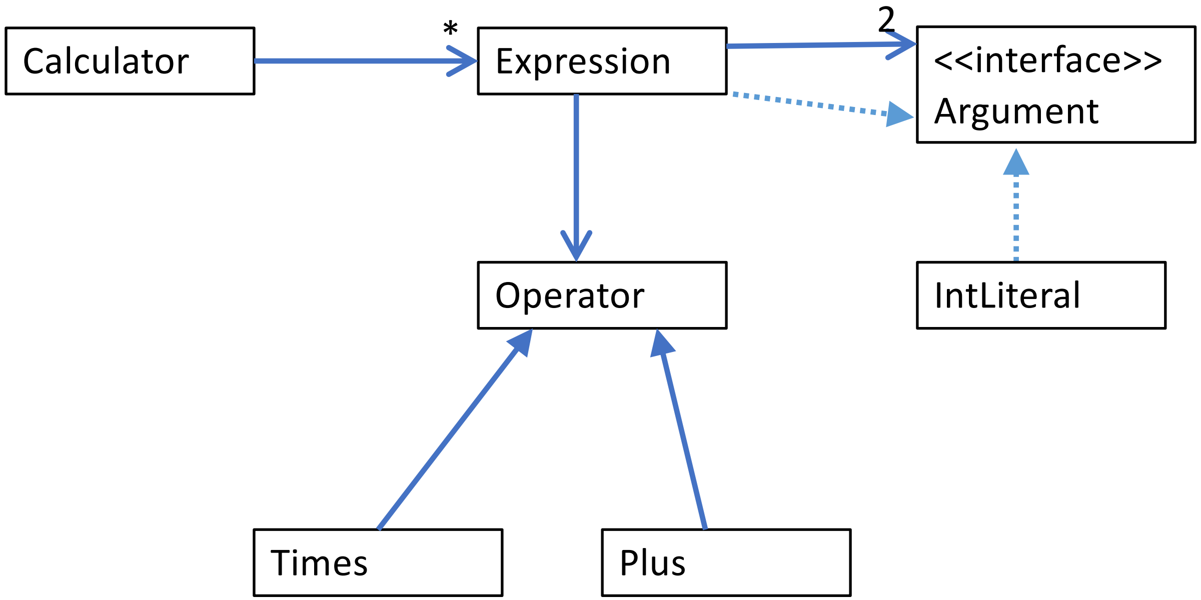
New Requirement

- Support other types of expression
  - ((2 + 3) – (4 + 6))

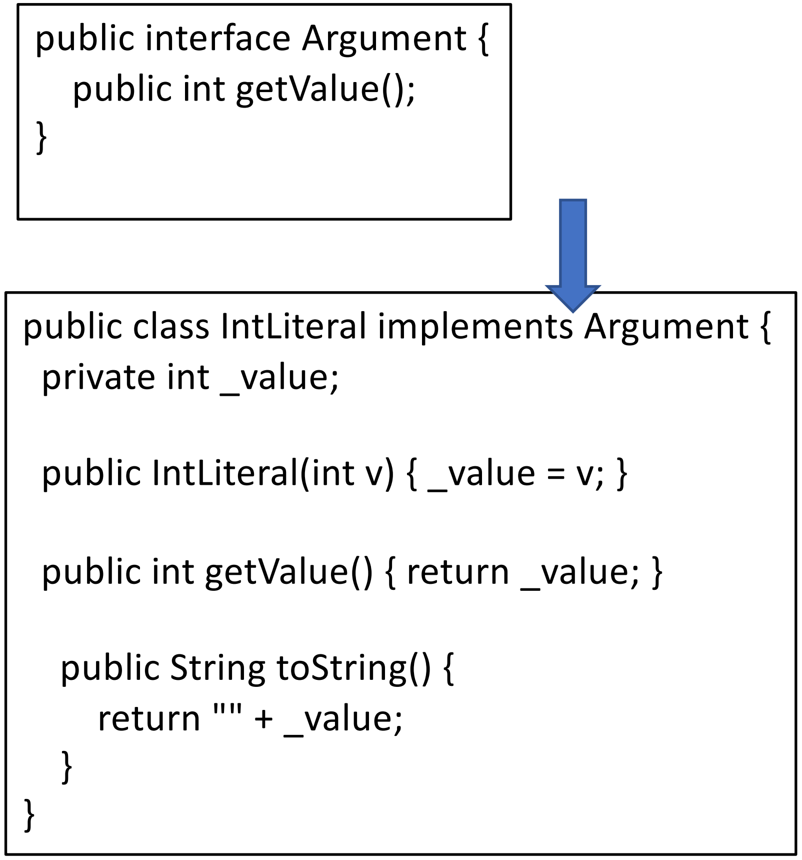
- Can we do it following the Open/Closed Principle?

- Yes, just need to find the right abstraction

Improved Solution



Improved Solution - Code



```
public interface Argument {
   public int getValue();
}
```

```
public class IntLiteral implements Argument {
  private int _value;

  public IntLiteral(int v) { _value = v; }

  public int getValue() { return _value; }

    public String toString() {
       return "" + _value;
    }
}
```

```
public class Expressior  implements Argument {
  private Argument _arg1;
  private Argument _arg2;
  private Operator _operator;

  public Expression(Operator operator, Argument arg1, Argument arg2) {
    // same as before
  }

  public int compute() {
    // same as before
  }

  public String toString() { /* same as before*/  }

  public final int getValue() {
    return compute();
  }
}
```

More information

- Robert C. Martin "The Open-Closed Principle"
  - https://drive.google.com/file/d/0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/view