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?

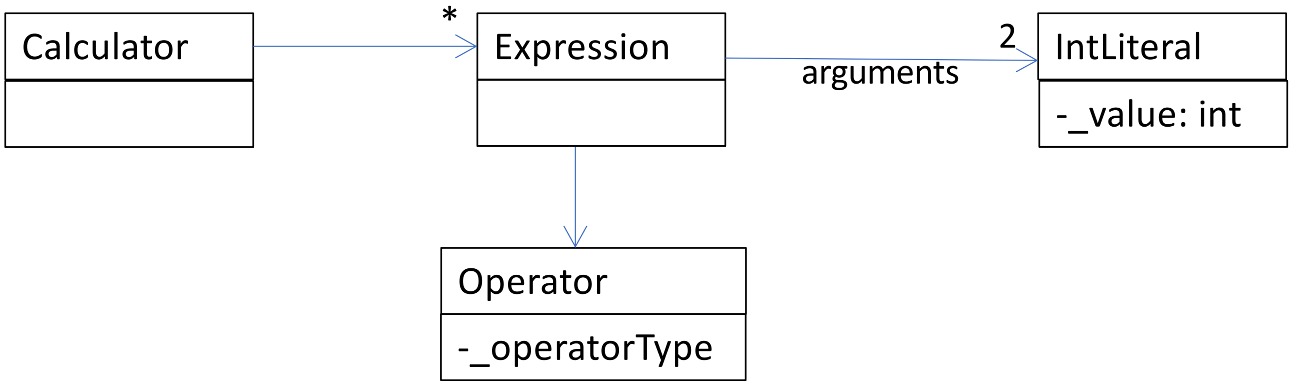
- Abstraction is the key

Example

- Simple calculator machine (consider only integer numbers):
  - 2 + 4
  - 2 / 4
  - 45 % 4
  - …
- Functionalities:
  - Add expression
  - Execute last expression
  - Show all expressions preserving insertion order
  - Check if expression is valid (no arithmetic errors, for instance)
  - Should support operations +,-,*,/
    - There could be more in the future

Without Open/Closed (Java C – version)

- Domain model:



- What are the attributes and methods of these entities?

Without Open/Closed - Implementation

- A calculator knows
  - Several expressions: one-to-many association with Expression
    - Attribute in Calculator for holding this information
    - Type?
    - Expression[] or Collection<Expression> ?
    - Since insertion order should be preserved pick **List<Expression>**

- And has functionality
  - Create a calculator
  - Add an expression
  - Evaluate all expressions
  - Evaluate last expression

Without Open/Closed - Calculator

- A calculator knows
  - Several expressions

- And has functionality
  - Create a calculator
  - Add an expression
  - Evaluate all expressions
  - Evaluate last expression

```
public class Calculator {
  private List<Expression> _expressions;

  public Calculator(int initialSize) {
      _expressions = new ArrayList<>(initialSize);
  }

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

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

Without Open/Closed - Expression

- An expression knows
  - Two arguments
  - And an operator
- And has functionality
  - Evaluate
  - *Convert* to string
  - Is valid

```
public class Expression {
  private IntLiteral _arg1;
  private IntLitral _arg2;
  private Operator _operator;

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

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

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

Without Open/Closed – Argument and Operator

- Argument
  - Has a number
- Operator
  - Has to know operator type
  - Can be an int
    - 0-> +, 1 -> -, …
  - Best solution is to use an enum

```
public class IntLiteral {
  private final int _value;

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

  public int getValue() { return _value; }

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

```
public enum OperatorType {
  PLUS(“+”), MINUS(“-”), TIMES(“*”), DIVIDE(“/”);

  private final String _operation;

  private OperatorType(String t) { _operation = t; }

  public String toString() { return _operation; }
}
```

Without Open/Closed - Operator

```
public class Operator {
  private OperatorType _operator;

  public Operator(OperatorType type) {  _operator = type; }

  public int evaluate(IntLiteral arg1, IntLiteral arg2) {
    switch(_operator) {
    case PLUS:
      return arg1.getValue() + arg2.getValue();
    case MINUS:
      return arg1.getValue() - arg2.getValue();
    case TIMES:
      return arg1.getValue() * arg2.getValue();
    case DIVIDE:
      return arg1.getValue() / arg2.getValue();
    }
    return 0;  // should throw an exception!
  }
  public String toString() {
    return _operator.toString()
  }
```

- And isValid() ?

```
  public boolean isValid (IntLiteral arg1, IntLiteral arg2) {
    switch(_operator) {
    case PLUS:
    case MINUS:
    case TIMES:
      return true;
    case DIVIDE:
      return arg2.getValue() != 0;
    }
    return false;  // should throw an exception!
  }
}
```

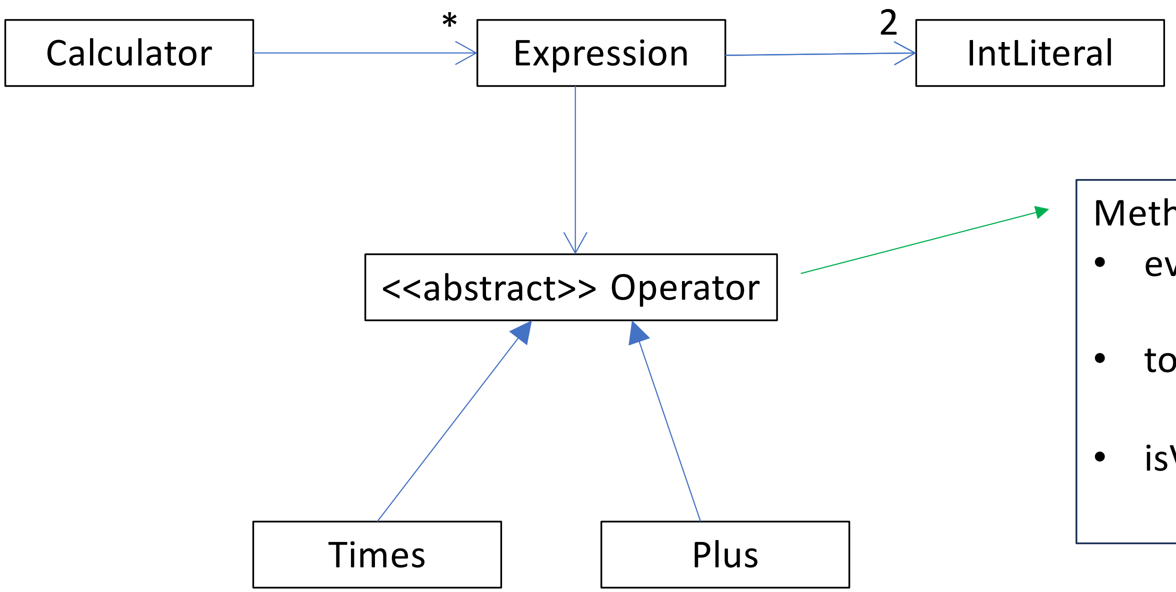
Main Problem with this Solution?

- Does not obey to the Open/Closed Principle

- Extend the application to support more operation types
  - Implies modifications in the code
  - Needs to change two methods in Operator
    - **evaluate()** and **isValid()**

- What is the problem here?
- Operator it is not an abstraction

Better Solution



Methods:

- evaluate()
  - Should be abstract
- toString()
  - Also abstract
- isValid()
  - And this one?

Calculator, **Expression** and **Argument** remain the same

Better Solution - Code

```
public abstract class Operator {
  public abstract int evaluate(Argument arg1, Argument arg2);
  public boolean isValid(IntLiteral arg1, IntLiteral arg2) { return true; }
  public abstract String toString();
}
```

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

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

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

  public String toString() {
    return "/";
  }
  public boolean isValid(IntLiteral arg1, IntLiteral arg2) {
    return arg2.getValue() != 0;
  }
}
```

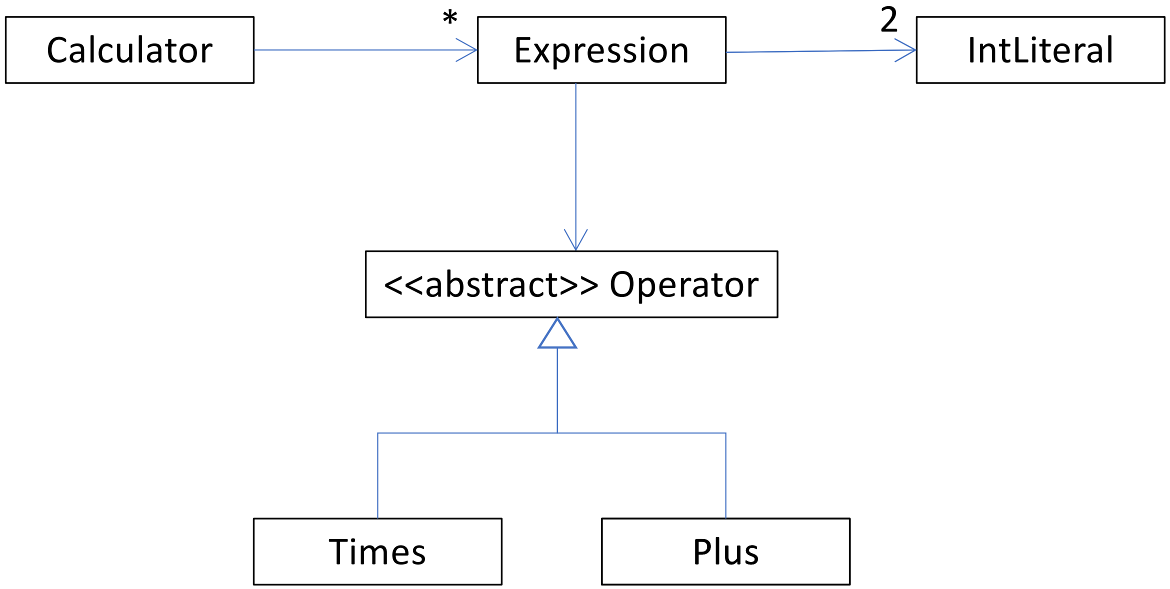
Better Solution and Open/Closed Principle

- 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

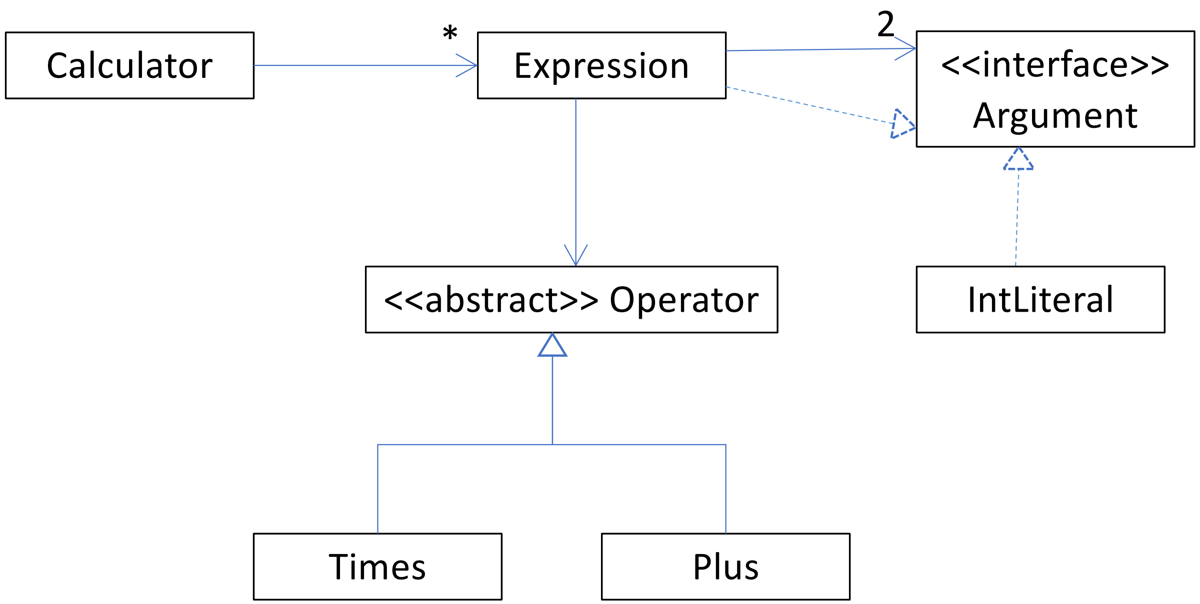
New Requirement

- 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

What do we need to change?



Improved Solution



Improved Solution - Code

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

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

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

  public int getValue() { return _value; }

    public String toString() {
       return Integer.toString(_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 evaluate() {/* same as before*/  }

  public Boolean isValid() {/* same as before*/  }

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

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

New method

More information

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