INTERPRETER

- Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar

- The pattern uses a class to represent each grammar rule. And since grammars are usually hierarchical in structure, an inheritance hierarchy of rule classes maps nicely.

- Pattern is helpful when we work with domain specific languages (DSL – It has its our grammar that specifies solutions in its own domain)

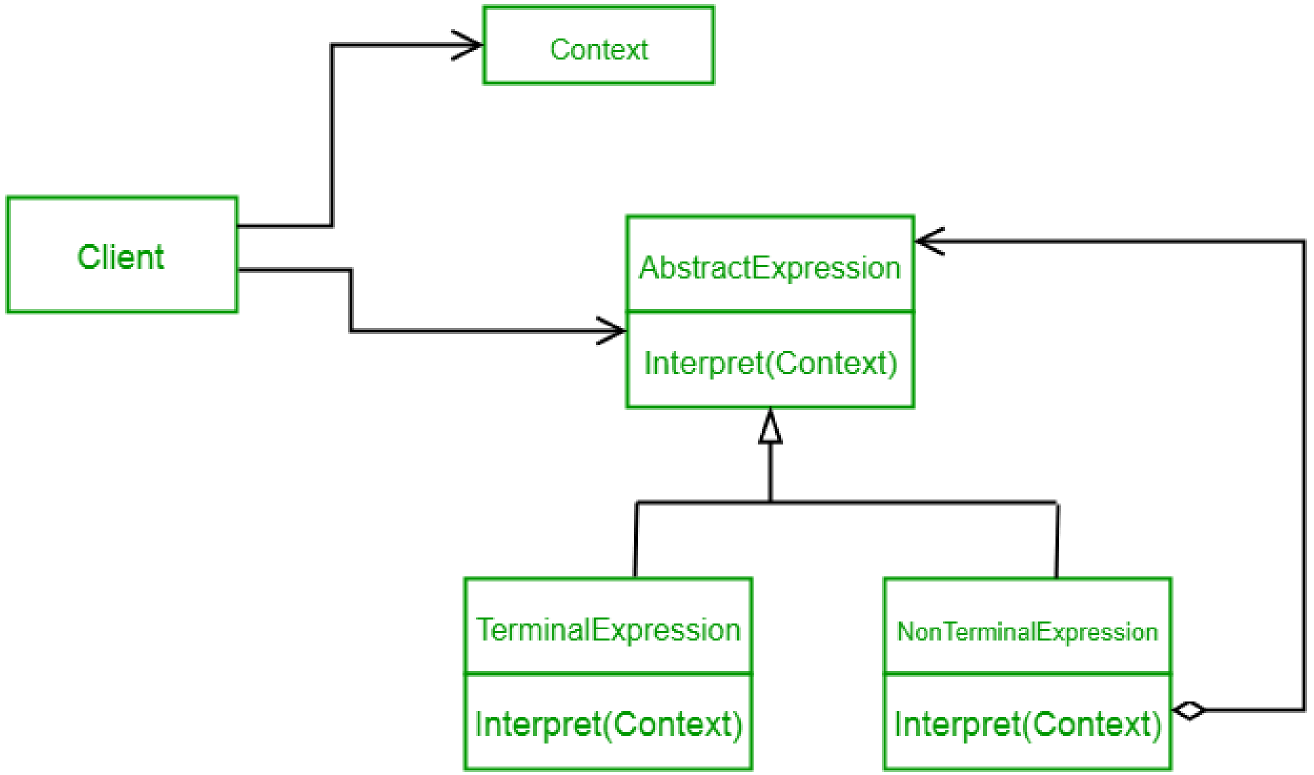
- It hides the complexity of the domain from rest of application structure.

INTENT

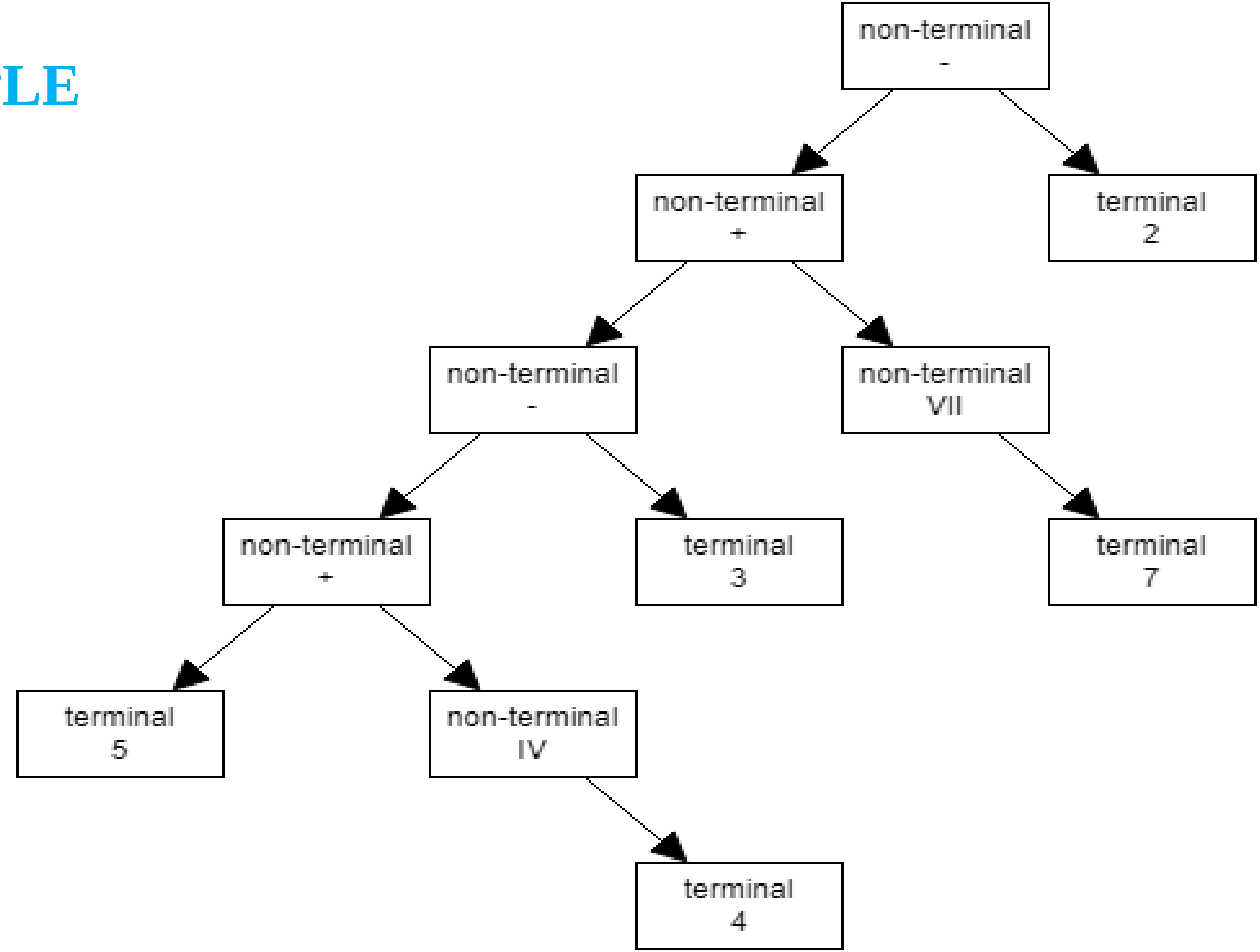
- When you have multiple objectives existing in a well defined and well understood domain.
- Objectives can be expressed with a simple language.

UML CLASS DIAGRAM

- The pattern uses a minimum of 5 classes.
- The 3 expression classes represent a row, keyword, operation or literal.
- The **terminal expression** is independent and does not refer to any other expression
- The **non terminal expression** refer to one or more sub expressions
- The abstract expression is the base class for all other classes.
- The client class holds a reference to the language context and initialises an expression defining grammar and syntax of the domain.



EXAMPLE



non-terminal
-
non-terminal
+
terminal
2
non-terminal
-
non-terminal
VII
non-terminal
+
terminal
3
terminal
7
terminal
5
non-terminal
IV
terminal
4

```python
"The Interpreter Pattern Concept"

class AbstractExpression():
    "All Terminal and Non-Terminal expressions will implement an
`interpret` method"
    @staticmethod
    def interpret():
        """
        The `interpret` method gets called recursively for each
        AbstractExpression
        """

class Number(AbstractExpression):
    "Terminal Expression"

    def __init__(self, value):
        self.value = int(value)

    def interpret(self):
        return self.value

    def __repr__(self):
        return str(self.value)
```

```python
class Add(AbstractExpression):
    "Non-Terminal Expression."

    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() + self.right.interpret()

    def __repr__(self):
        return f"({self.left} Add {self.right})"

class Subtract(AbstractExpression):
    "Non-Terminal Expression"

    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() - self.right.interpret()

    def __repr__(self):
        return f"({self.left} Subtract {self.right})"
```

```python
# The Client
# The sentence complies with a simple grammar of
# Number -> Operator -> Number -> etc,
SENTENCE = "5 + 4 - 3 + 7 - 2"
print(SENTENCE)

# Split the sentence into individual expressions that will be added to
# an Abstract Syntax Tree (AST) as Terminal and Non-Terminal expressions
TOKENS = SENTENCE.split(" ")
print(TOKENS)

# Manually Creating an Abstract Syntax Tree from the tokens
AST: list[AbstractExpression] = []  # Python 3.9
# AST = []  # Python 3.8 or earlier
AST.append(Add(Number(TOKENS[0]), Number(TOKENS[2])))  # 5 + 4
AST.append(Subtract(AST[0], Number(TOKENS[4])))       # ^ - 3
AST.append(Add(AST[1], Number(TOKENS[6])))            # ^ + 7
AST.append(Subtract(AST[2], Number(TOKENS[8])))       # ^ - 2

# Use the final AST row as the root node.
AST_ROOT = AST.pop()

# Interpret recursively through the full AST starting from the root.
print(AST_ROOT.interpret())

# Print out a representation of the AST_ROOT
print(AST_ROOT)
```

OUTPUT

```
5 - 4 + 3 + 7 - 2
['5', '-', '4', '+', '3', '+', '7', '-', '2']
11
((((5 Add 4) Subtract 3) Add 7) Subtract 2)
```