Interpreter Design Pattern

Contents

[Intent of the DP 2](#_Toc516319095)

[Explanation 2](#_Toc516319096)

[Common situations of use 3](#_Toc516319097)

[Related patterns 3](#_Toc516319098)

[Useful Links 4](#_Toc516319099)

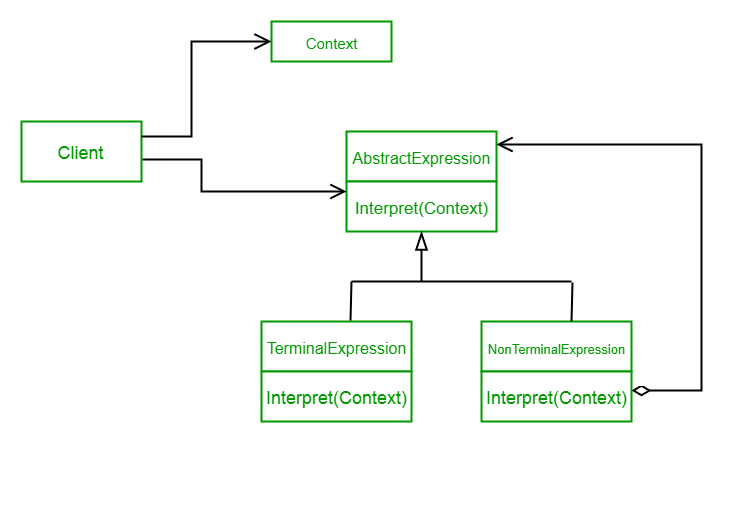
# Intent of the DP

The Interpreter pattern is a behavioral design pattern that specifies how to evaluate sentences in a language. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language. The syntax tree of a sentence in the language is an instance of the composite pattern and is used to evaluate (interpret) the sentence for a client.

# Explanation

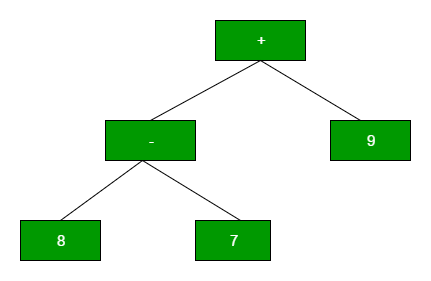
The solution described by the pattern:

* Define a grammar for a simple language by defining an Expression class hierarchy and implementing an interpret() operation.
* Represent a sentence in the language by an abstract syntax tree (AST) made up of Expression instances.
* Interpret a sentence by calling interpret() on the AST.

**UML Diagram Interpreter Design Pattern**:

The expressions form a tree-like hierarchy, resembling the composite design pattern, in which terminal expressions are leaf objects, and nonterminal expressions are composites. The tree contains the expressions to be evaluated and is usually generated by a parser. The parser itself is not a part of the interpreter pattern.

This is how a hierarchy tree for “+ - 9 8 7” looks like:



Here, 9, 8 and 7 are terminal expressions, while + and – are nonterminal.

Regarding the SOLID principles:

**S:** The Expression classes have the single responsibility of interpreting of an expression.

**O:** The abstract Expression class is extended by the terminal Expression class and the nonterminal one. In theory, one could add a third type of expression without needing to modify the rest of the code, if that is necessary. This respects the open/closed principle.

**L:** The Liskov substitution principle stands, as TerminalExpression and NonterminalExpression could be substituted for their parent, the Expression interface.

I: The interface segregation principle doesn’t necessarily apply here as the only needed interface is the one defining Expression, with a single interpret() method.

D: The dependency inversion principle doesn’t apply.

# Common situations of use

* Specialized database query languages such as [SQL](https://en.wikipedia.org/wiki/SQL).
* Specialized computer languages that are often used to describe communication protocols.
* Most general-purpose computer languages actually incorporate several specialized languages.

# Related patterns

As stated before, it uses a tree structure to represent an object hierarchy which is to be treated as a whole. This approach is similar to the one used in the Composite pattern.

The two patterns have a different purpose, but one may use Composite in the implementation of the Interpreter pattern.

# Useful Links

<https://www.tutorialspoint.com/design_pattern/interpreter_pattern.htm>

<https://en.wikipedia.org/wiki/Interpreter_pattern>

https://stackoverflow.com/questions/25720099/design-patterns-what-is-the-difference-between-interpreter-and-composite?utm\_medium=organic&utm\_source=google\_rich\_qa&utm\_campaign=google\_rich\_qa