**Interpreter Design Pattern:** The Interpreter design pattern is a[behavioral design pattern](https://www.geeksforgeeks.org/behavioral-design-patterns/) that defines a way to interpret and evaluate language grammar or expressions. It provides a mechanism to evaluate sentences in a language by representing their grammar as a set of classes. Each class represents a rule or expression in the grammar, and the pattern allows these classes to be composed hierarchically to interpret complex expressions.

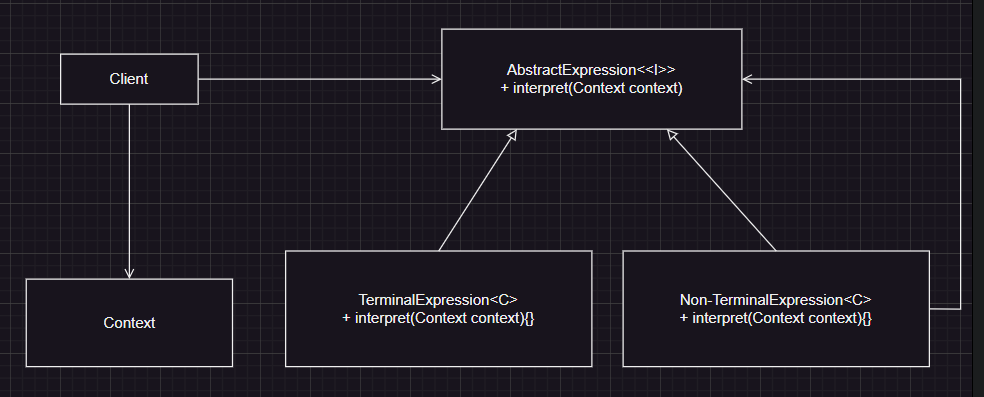
* The pattern involves defining a hierarchy of expression classes, both terminal and nonterminal, to represent the elements of the language’s grammar.
* Terminal expressions represent basic building blocks, while nonterminal expressions represent compositions of these building blocks.
* The tree structure of the Interpreter design pattern is somewhat similar to that defined by the [composite design pattern](https://www.geeksforgeeks.org/composite-design-pattern-in-java/) with terminal expressions being leaf objects and non-terminal expressions being composites.

**When to use Interpreter Design Pattern**

* **When dealing with domain-specific languages:**
  + If you need to interpret and execute expressions or commands in a domain-specific language (DSL), the Interpreter pattern can provide a flexible and extensible way to implement the language’s grammar and semantics.
* **When you have a grammar to interpret:**
  + If you have a well-defined grammar for expressions or commands that need to be interpreted, the Interpreter pattern can help parse and evaluate these structures efficiently.
* **When adding new operations is frequent:**
  + If your application frequently requires the addition of new operations or commands, the Interpreter pattern allows you to add new expression classes easily without modifying existing code, thus promoting maintainability and extensibility.
* **When you want to avoid complex grammar parsers:**
  + If building and maintaining complex grammar parsers seems daunting or unnecessary for your use case, the Interpreter pattern offers a simpler alternative for interpreting expressions directly.

**When not to use Interpreter Design Pattern**

* **For simple computations:**
  + If your task involves only simple computations or operations that can be easily handled by built-in language features or libraries, using the Interpreter pattern may introduce unnecessary complexity.
* **When performance is critical:**
  + Interpreting expressions through the Interpreter pattern might introduce overhead compared to other approaches, especially for complex expressions or large input sets. In performance-critical applications, a more optimized solution, such as compilation to native code, may be preferable.
* **When the grammar is too complex:**
  + If your grammar is highly complex, with numerous rules and exceptions, implementing it using the Interpreter pattern may lead to a proliferation of expression classes and increased code complexity.
  + In such cases, a dedicated parser generator or compiler may be more suitable.
* **When there’s no need for extensibility:**
  + If the requirements of your application are fixed and well-defined, and there’s no anticipation of adding new operations, commands, or language constructs in the future, then implementing the Interpreter pattern may introduce unnecessary complexity.



**Components:**

* **Abstract Expression**: Declares an interface for interpreting the context.
* **Terminal Expression**: Implements the interpretation method for the simplest components of the grammar.
* **Non-Terminal Expression**: Represents more complex grammar rules, usually composed of terminal and non-terminal expressions.
* **Context**: Contains information that’s global to the interpreter and used during interpretation.
* **Client**: The user of the interpreter, which builds the abstract syntax tree representing a specific sentence in the language.

