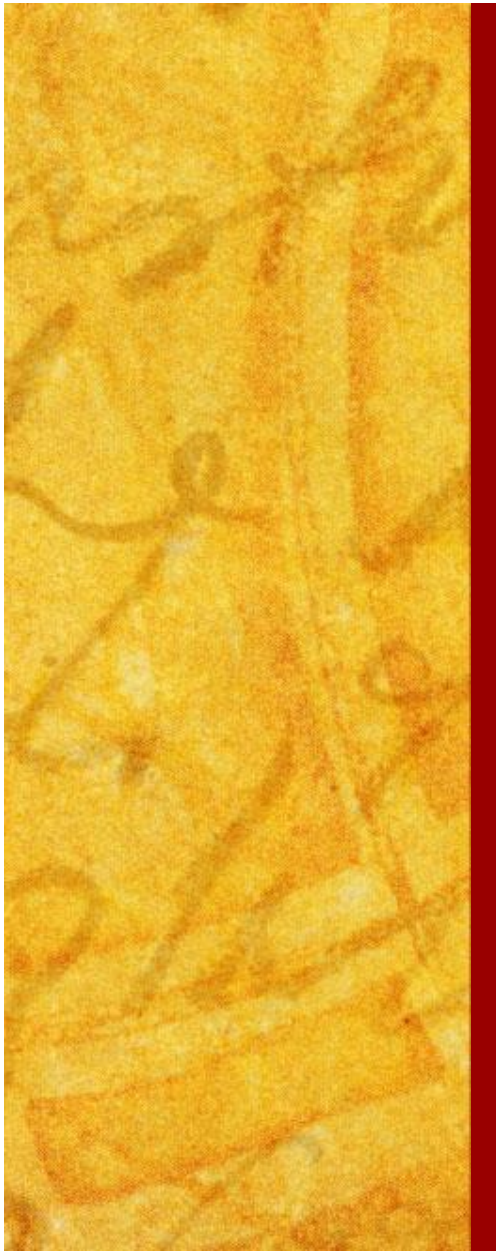
Interpreter

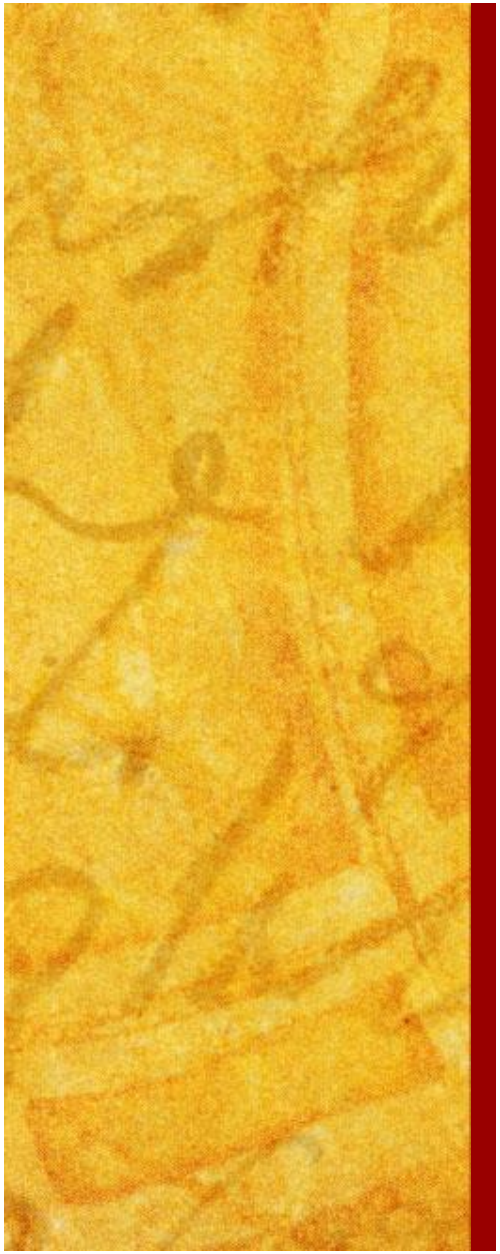
By
Sanket Sharma
Date: January 26, 2011



Intent

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

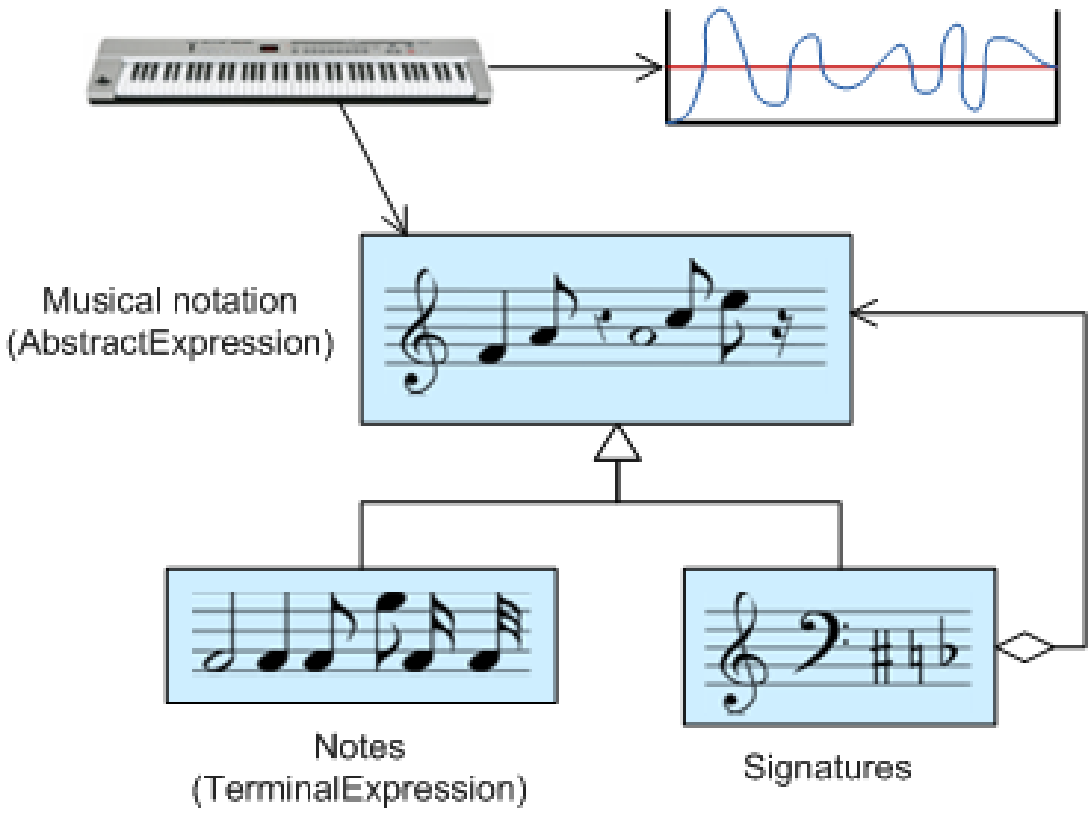
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design.

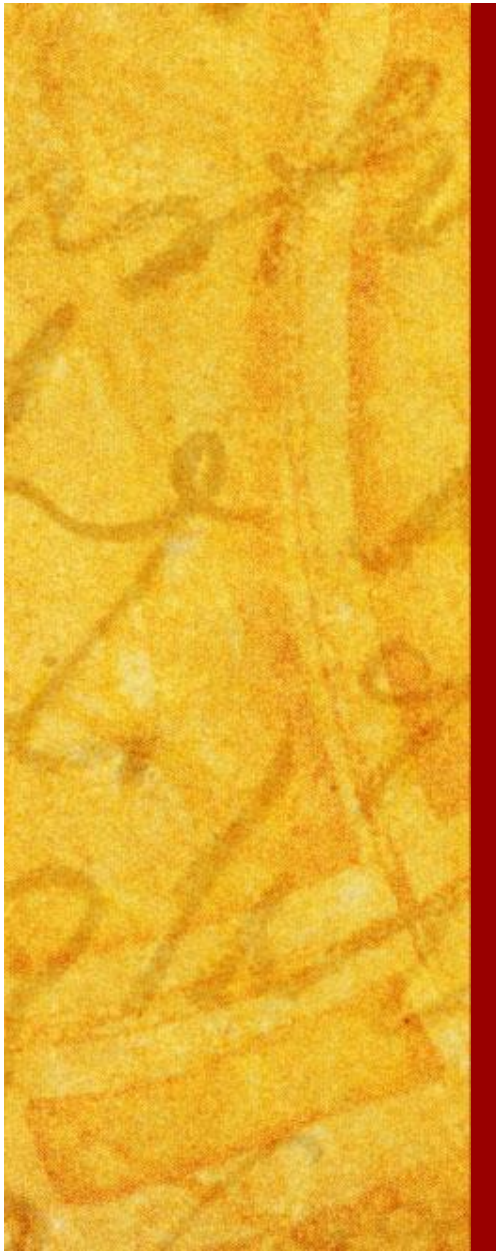


Problem

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a “language”, then problems could be easily solved with an interpretation “engine”.

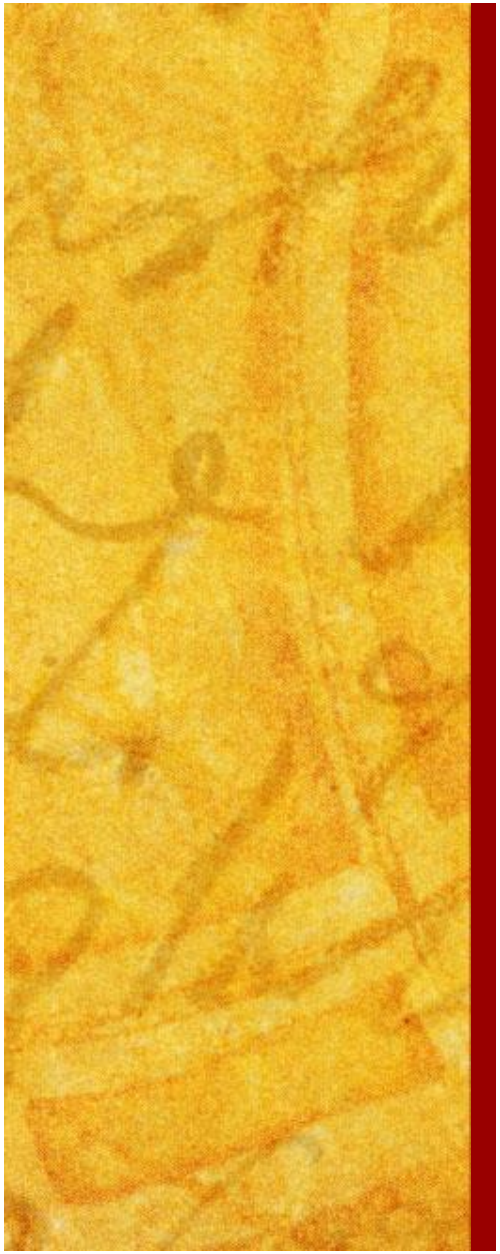
Example


Musical notation
(AbstractExpression)
Notes
(TerminalExpression)
Signatures



Example

The Interpreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.

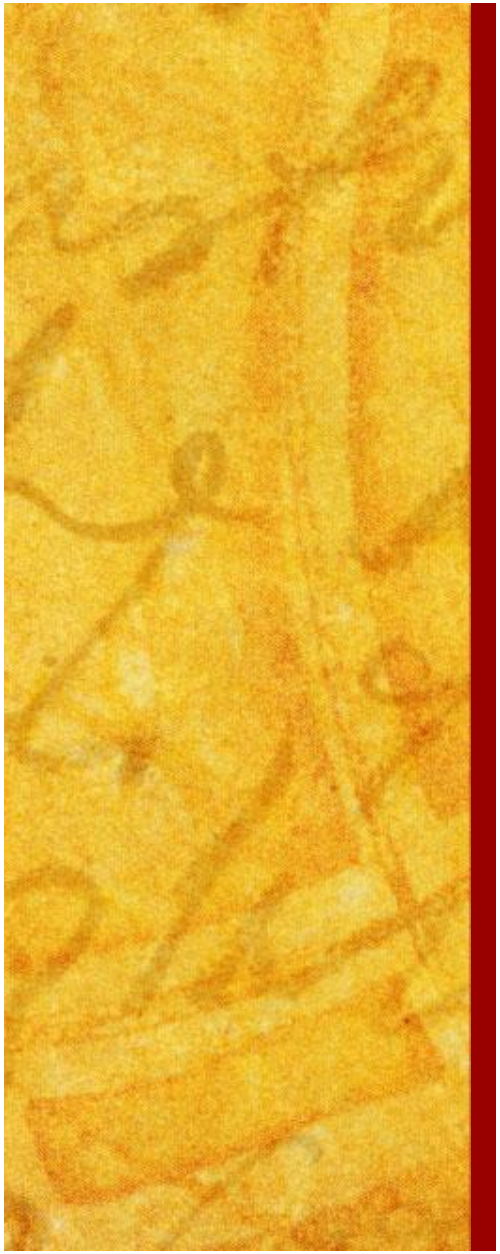


Applicability

Use the Interpreter pattern when there is a language to interpret, and you can represent statements in the language as abstract syntax trees. The Interpreter pattern works best when

- the grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable.

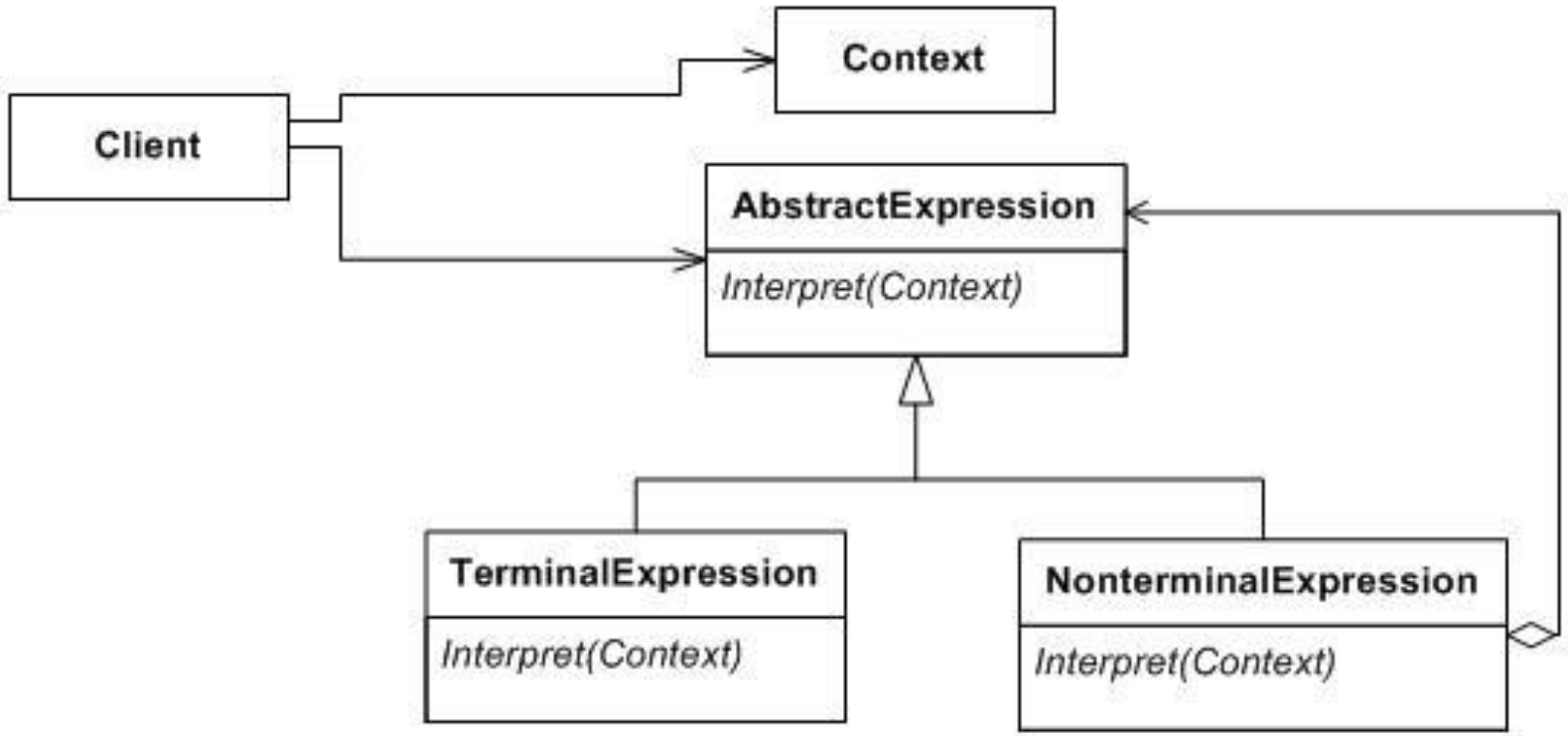
- efficiency is not a critical concern. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form.

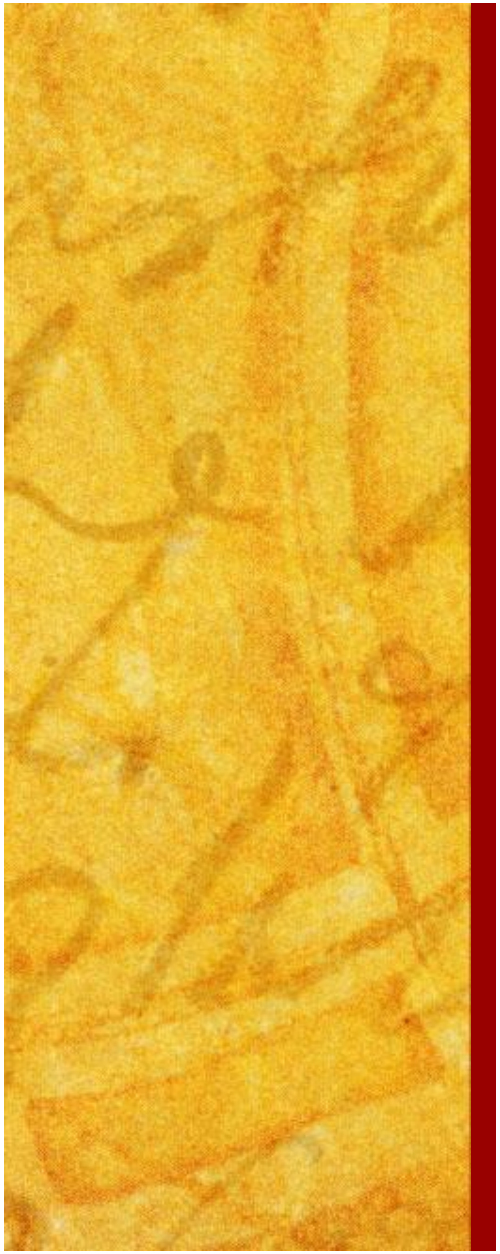


Structure

Interpreter suggests modeling the domain with a recursive grammar. Each rule in the grammar is either a ‘composite’ (a rule that references other rules) or a terminal (a leaf node in a tree structure). Interpreter relies on the recursive traversal of the Composite pattern to interpret the ‘sentences’ it is asked to process.

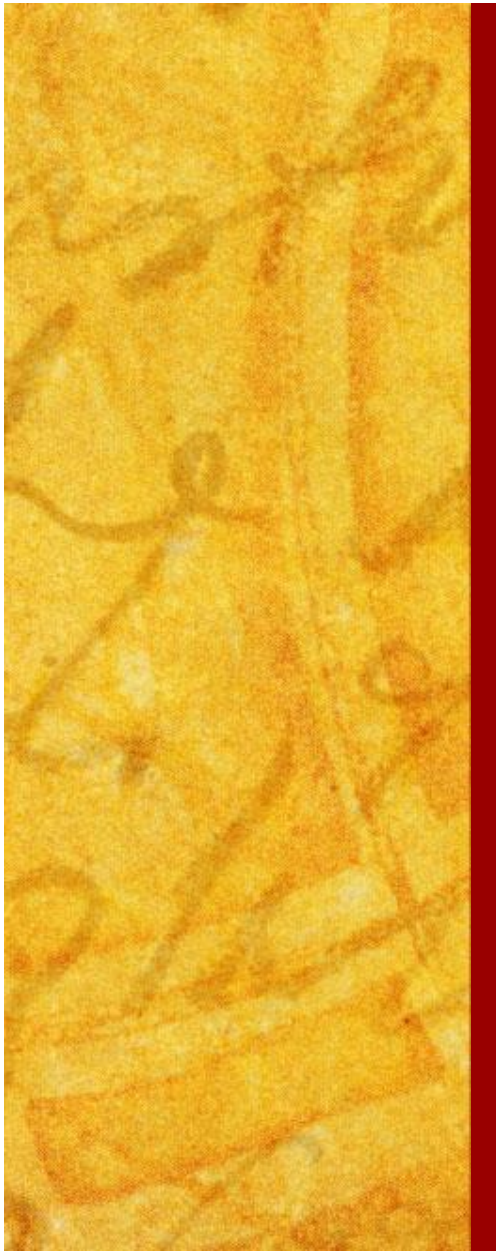
Structure


Context
Client
AbstractExpression
Interpret(Context)
TerminalExpression
Interpret(Context)
NonterminalExpression
Interpret(Context)



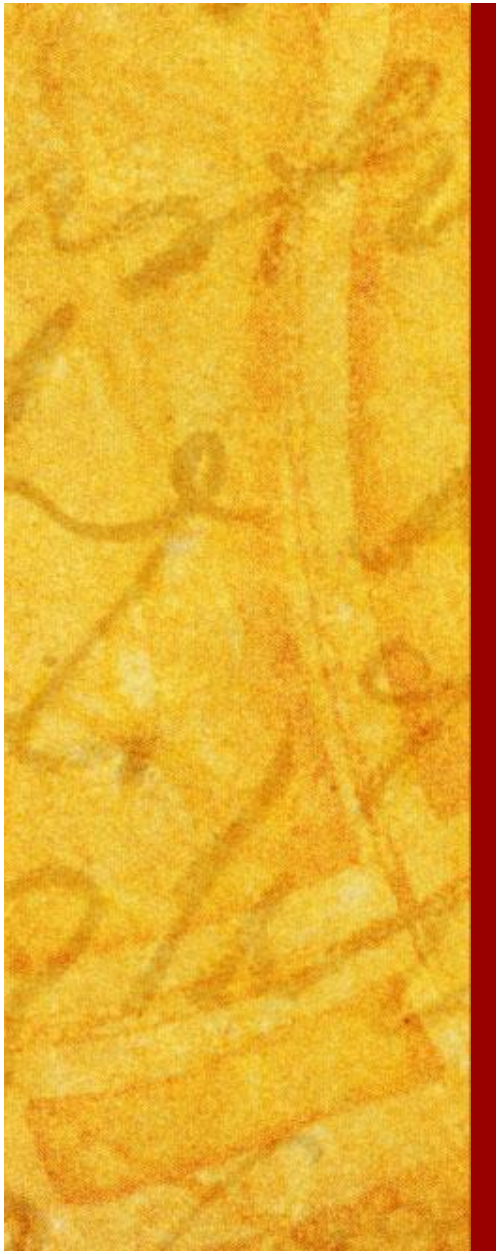
AbstractExpression

Declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree



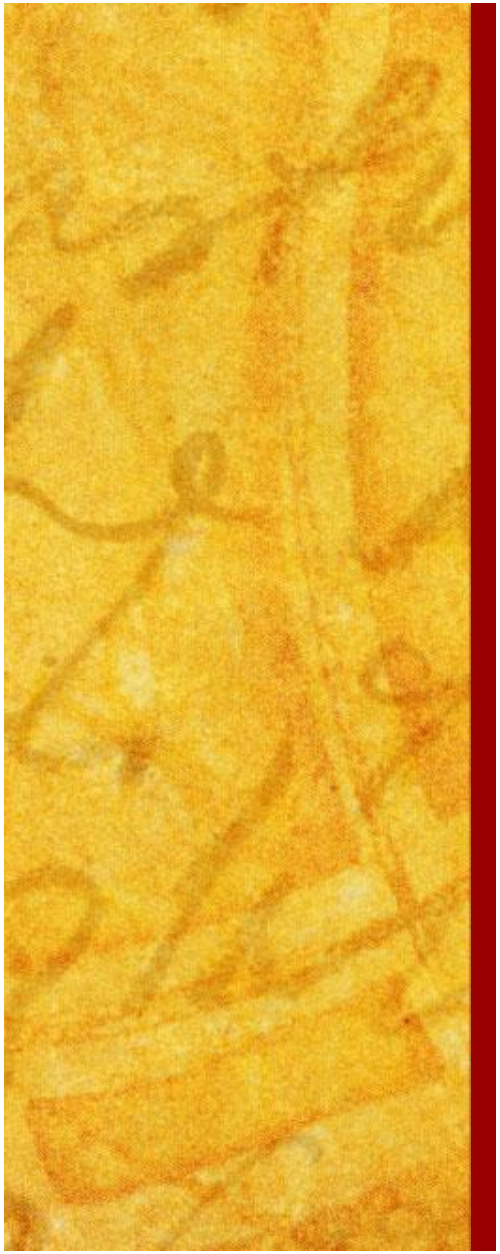
TerminalExpression

- Implements an Interpret operation associated with terminal symbols in the grammar
- an instance is required for every terminal symbol in a sentence.



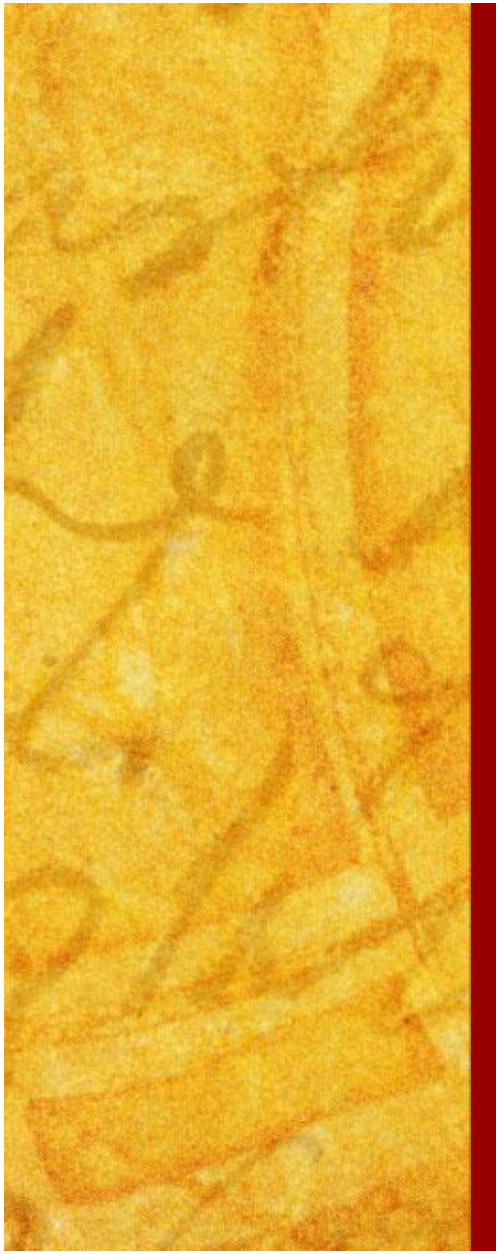
NonterminalExpression

- one such class is required for every rule *R ::= R1 R2 … Rn in the grammar*
- maintains instance variables of type AbstractExpression for each of the symbols *R1 through Rn.*
- implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing *R1 through Rn..*



Context

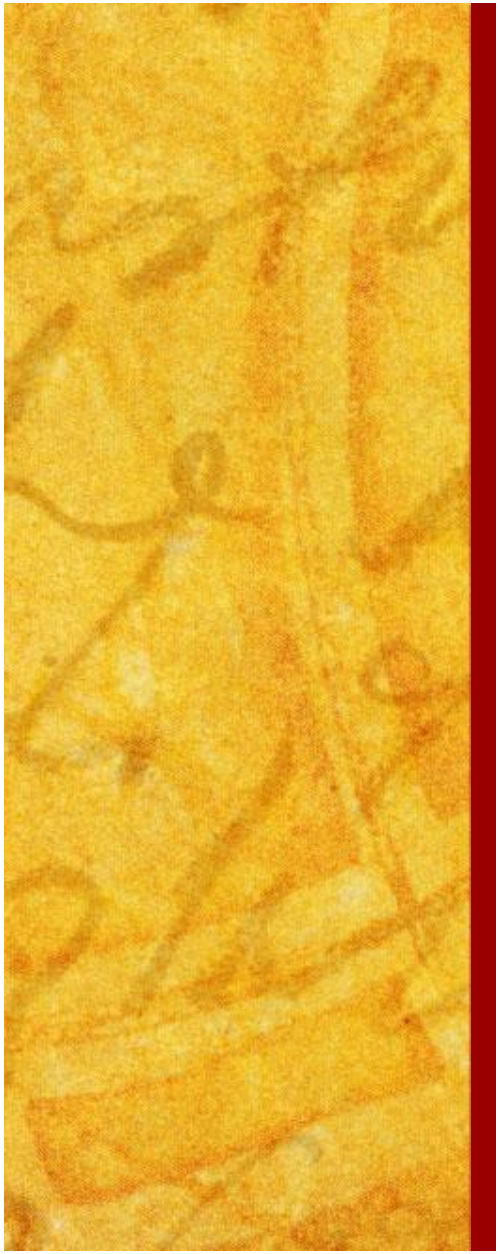
contains information that's global to the interpreter.



Client

- builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the NonterminalExpression and TerminalExpression classes.

- invokes the Interpret operation.



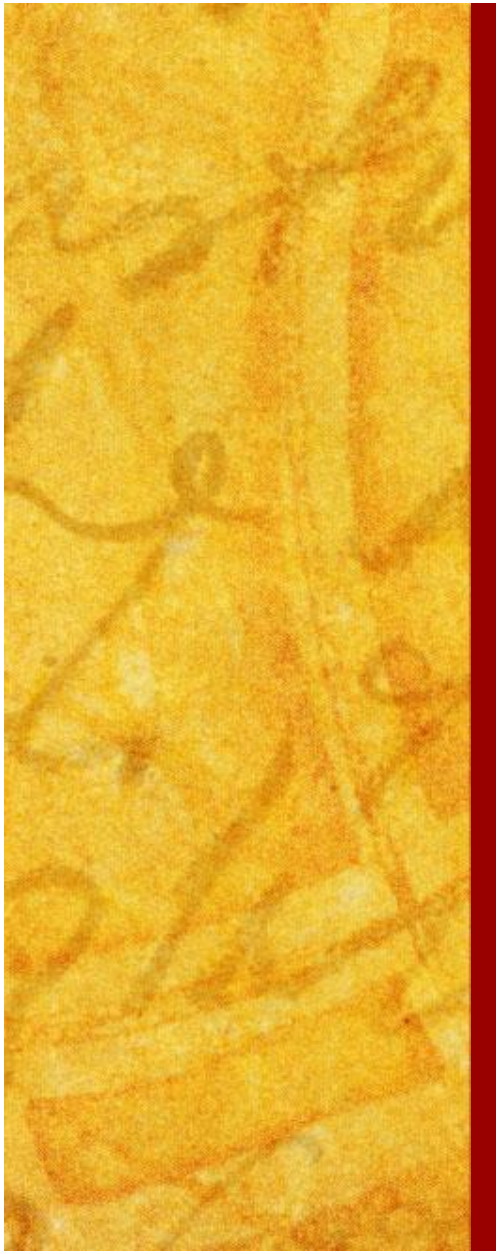
Collaborations

- The client builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances. Then the client initializes the context and invokes the Interpret operation.

- Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression. The Interpret operation of each TerminalExpression defines the base case in the recursion.

- The Interpret operations at each node use the context to store and access the state of the interpreter.

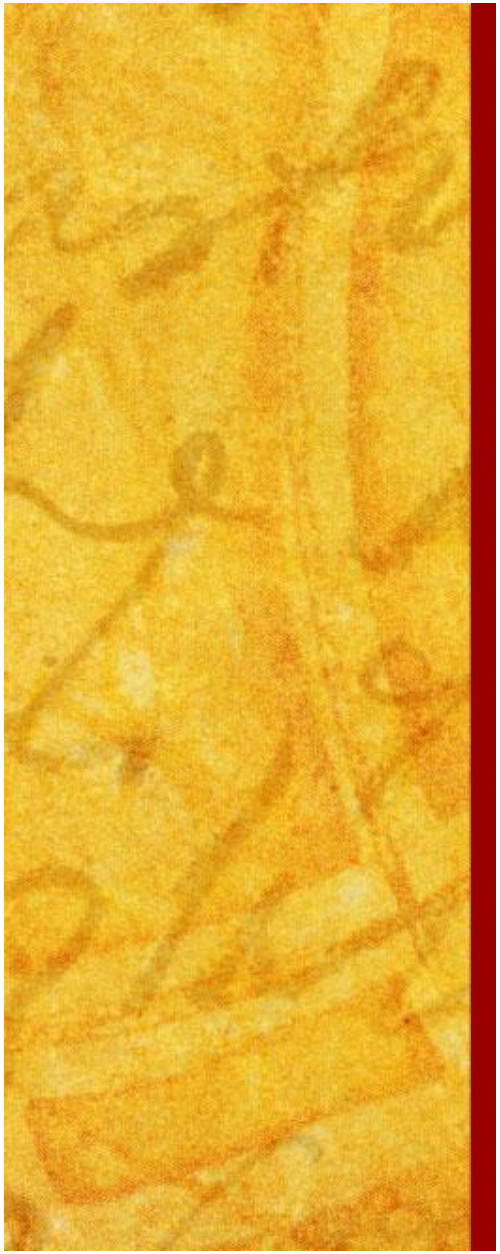
Code Example



Benefits

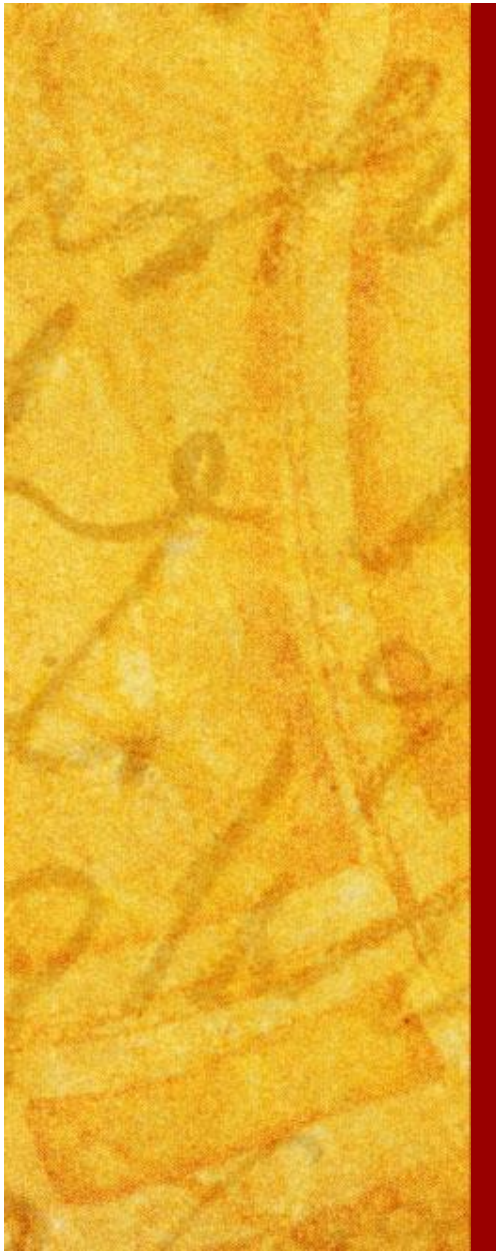
- *It's easy to change and extend the grammar. Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar. Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.*

- *Implementing the grammar is easy. Classes defining nodes in the abstract syntax tree have similar implementations. These classes are easy to write, and often their generation can be automated with a compiler or parser generator.*



Liabilities

Complex grammars are hard to maintain. The Interpreter pattern defines at least one class for every rule in the grammar. Hence grammars containing many rules can be hard to manage and maintain. Other design patterns can be applied to mitigate the problem. But when the grammar is very complex, other techniques such as parser or compiler generators are more appropriate.



References

- Design Patterns - Elements of Reusable Object-Oriented Software
  by Gamma, Helm, Johnson and Vlissides

- http://sourcemaking.com/design_patterns/interpreter

- http://en.wikipedia.org/wiki/Interpreter_pattern

Thanks