Interpreter Design Pattern

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".
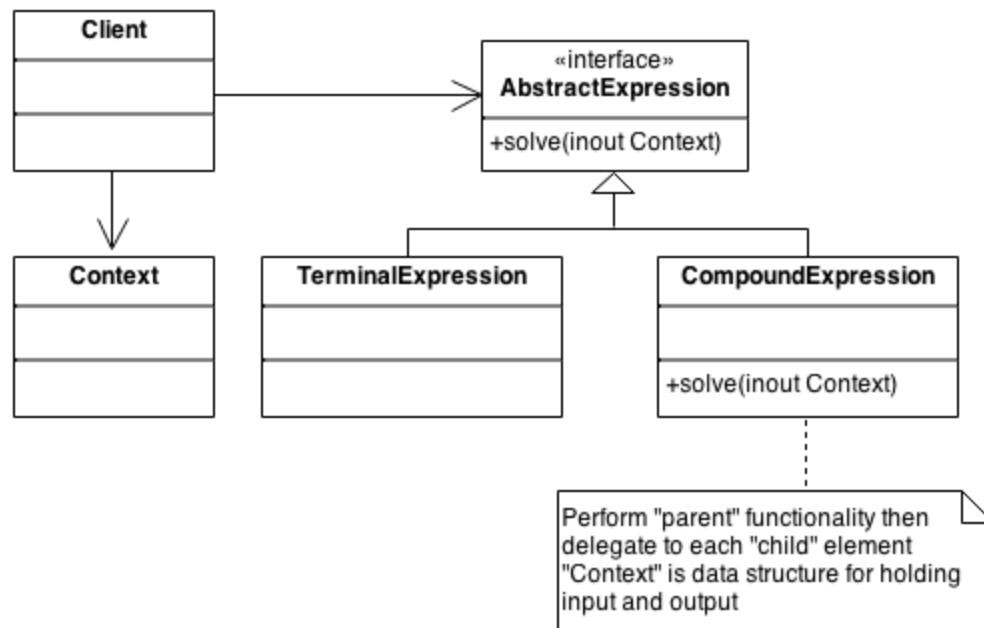
Discussion

The Interpreter pattern discusses: defining a domain language (i.e. problem characterization) as a simple language grammar, representing domain rules as language sentences, and interpreting these sentences to solve the problem. 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.

An abstract base class specifies the method `interpret()`. Each concrete subclass implements `interpret()` by accepting (as an argument) the current state of the language stream, and adding its contribution to the problem solving process.
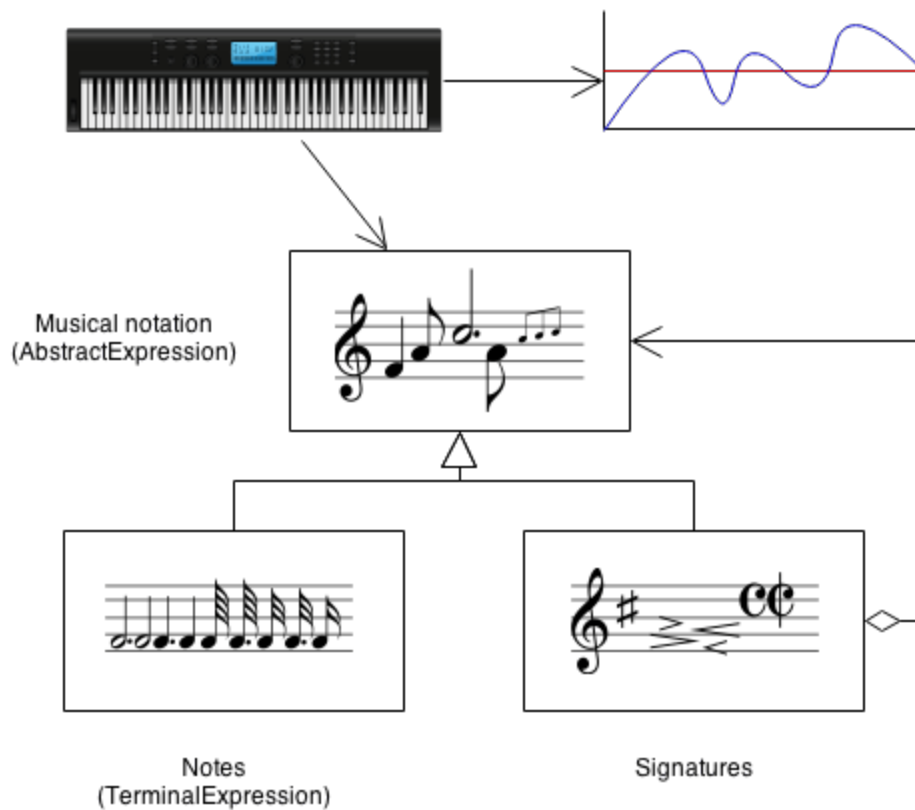
Structure

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.

```
┌─────────────┐          ┌──────────────────────┐
│   Client    │          │     «interface»      │
├─────────────┤─────────▷│  AbstractExpression  │
│             │          ├──────────────────────┤
├─────────────┤          │ +solve(inout Context)│
│             │          └──────────────────────┘
└─────────────┘                     △
       │              ┌─────────────┴─────────────┐
       ▼              │                           │
┌─────────────┐  ┌──────────────────────┐  ┌──────────────────────┐
│   Context   │  │  TerminalExpression  │  │  CompoundExpression  │
├─────────────┤  ├──────────────────────┤  ├──────────────────────┤
│             │  │                      │  │                      │
├─────────────┤  ├──────────────────────┤  ├──────────────────────┤
│             │  │                      │  │ +solve(inout Context)│
└─────────────┘  └──────────────────────┘  └──────────────────────┘
```

Perform "parent" functionality then delegate to each "child" element "Context" is data structure for holding input and output

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.

Musical notation (AbstractExpression)

Notes (TerminalExpression)

Signatures

Check list

1. Decide if a "little language" offers a justifiable return on investment.

2. Define a grammar for the language.

3. Map each production in the grammar to a class.

4. Organize the suite of classes into the structure of the Composite pattern.

5. Define an `interpret(Context)` method in the Composite hierarchy.

6. The `Context` object encapsulates the current state of the input and output as the former is parsed and the latter is accumulated. It is manipulated by each grammar class as the "interpreting" process transforms the input into the output.

Rules of thumb

- Considered in its most general form (i.e. an operation distributed over a class hierarchy based on the Composite pattern), nearly every use of the Composite pattern will also contain the Interpreter pattern. But the Interpreter pattern should be reserved for those cases in which you want to think of this class hierarchy as defining a language.

- Interpreter can use State to define parsing contexts.

- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).

- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.

- The pattern doesn't address parsing. When the grammar is very complex, other techniques (such as a parser) are more appropriate.

Support our free website and own the eBook!

- 22 design patterns and 8 principles explained in depth
- 406 well-structured, easy to read, jargon-free pages
- 228 clear and helpful illustrations and diagrams
- An archive with code examples in 4 languages
- All devices supported: EPUB/MOBI/PDF formats

▤ Learn more...
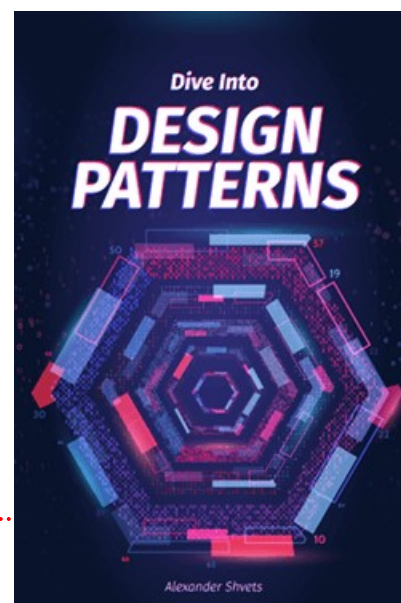
Code examples

Java	Interpreter in Java: Before and after	Interpreter in Java

C++	**Interpreter in C++**
PHP	**Interpreter in PHP**
Delphi	**Interpreter in Delphi**
Python	**Interpreter in Python**

⭐ More info, diagrams and examples of the Interpreter design pattern you can find on our new partner resource Refactoring.Guru.

RETURN

← Command

Design Patterns

AntiPatterns

Refactoring

UML

My account

Forum

Contact us

About us

Terms / Privacy policy