Análisis del patrón Visitor dentro del proyecto “Crafting Interpreters”.

Julian David Rios Bravo
Código: 202014750

1. Sobre el proyecto “Crafting Interpreters”.

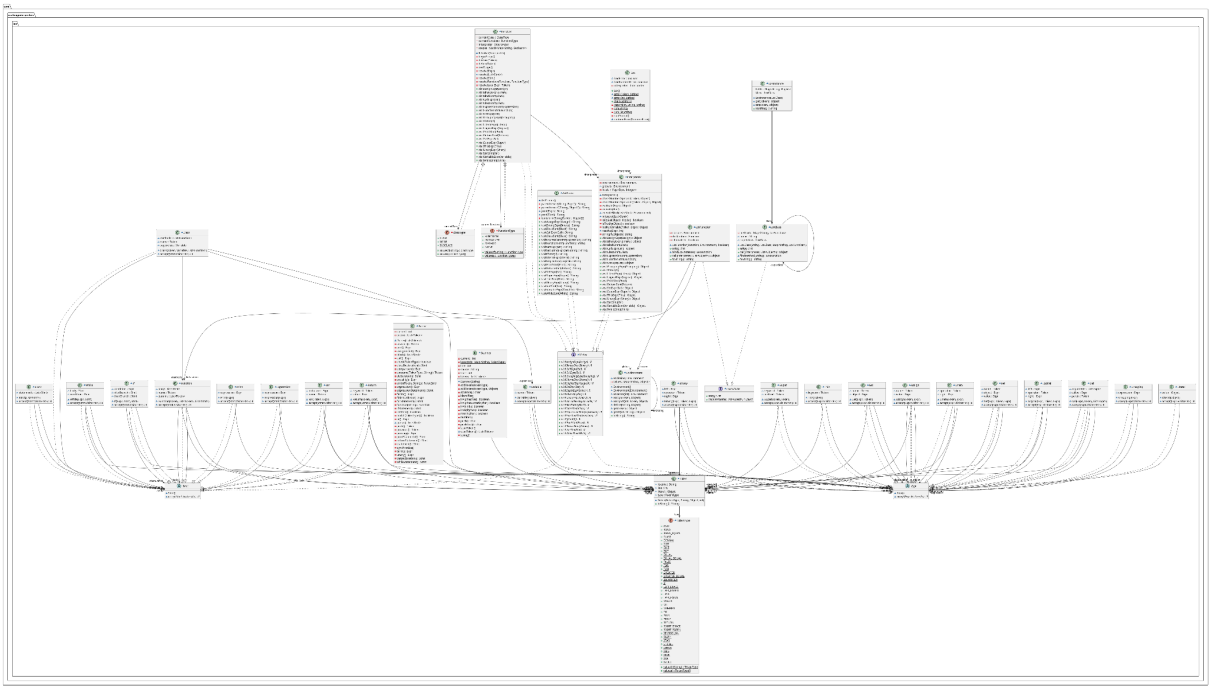
Este proyecto corresponde al resultado final de la implementación realizada en el libro homónimo escrito por Robert Nystron. Dicho libro es una guía para el lector en la construcción y comprensión de intérpretes, haciendo uso del lenguaje denominado “Lox” concebido por Nystron para ser un lenguaje simple y educativo. Un intérprete es una herramienta que ejecuta código fuente directamente, línea por línea, sin la necesidad de compilar previamente el código en un programa ejecutable. Esta herramienta resulta de gran ayuda para lograr entender el funcionamiento de un lenguaje de programación y/o un compilador, por lo que son usualmente utilizados como un recurso educativo para el aprendizaje de los mismos.

La estructura general de un intérprete se compone de:

1. **Scanner:** La clase Scanner analiza el código fuente y genera una secuencia de tokens, que son unidades léxicas como palabras clave, identificadores, números, etc.
2. **Parser:** La clase Parser convierte la secuencia de tokens en una estructura de árbol de sintaxis abstracta (AST). El AST representa la estructura jerárquica del código fuente.
3. **Resolver:** La clase Resolver realiza la resolución de nombres y la vinculación de variables en el ámbito del programa. También se encarga de la resolución de funciones y la gestión de clases.
4. **Interpreter:** La clase Interpreter interpreta y ejecuta el código representado por el AST. Realiza la evaluación de expresiones y la ejecución de declaraciones.

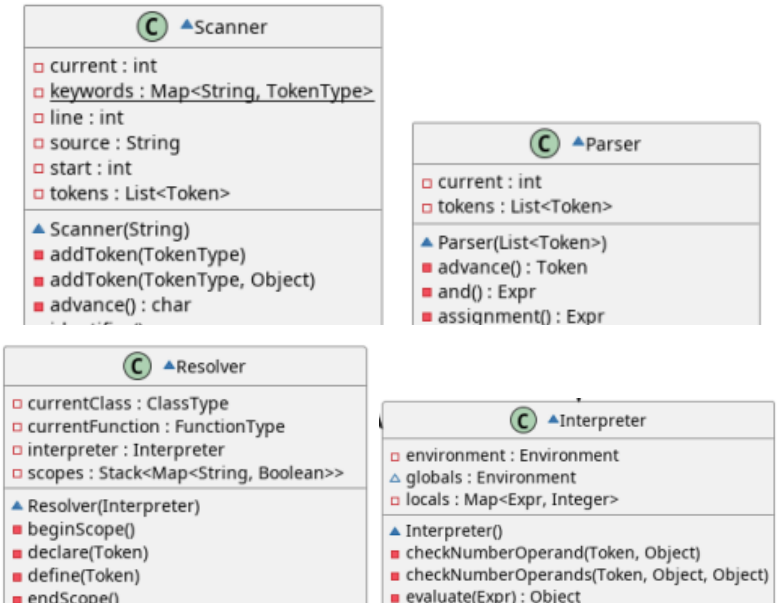
Resulta necesaria la comparación con la estructura del proyecto con el fin de identificar dichos componentes en el mismo, para ello se puede observar su

diagrama UML. Dada la extensión del código se hizo uso de herramientas adicionales para visualización del diagrama UML por lo que algunas relaciones entre las clases pueden llegar a ser poco claras pero en general se tiene una buena visión del mismo.

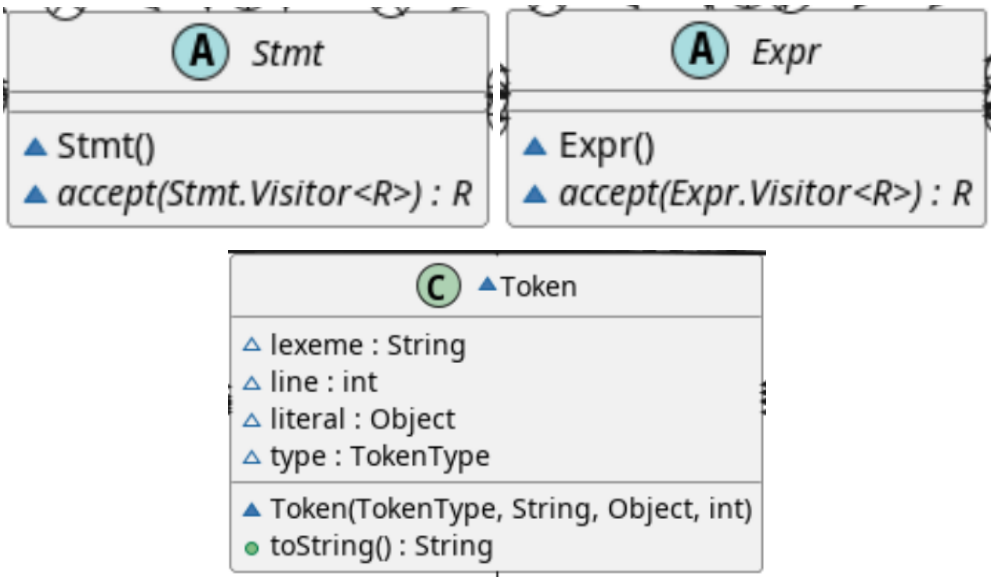


Para una mejor visualización consultar gráfico en el repositorio.

Como se puede apreciar dentro de la estructura se pueden encontrar dichos elementos representados a través de las clases “Scanner”, “Parser”, “Resolver” y “Interpreter”.



Adicionalmente se tienen otros elementos característicos de un intérprete como las definiciones de “Tokens”, “Statements” y/o “Expressions” las cuales construyen la lógica y jerarquía del lenguaje a interpretar, en este proyecto se pueden encontrar a través de clases como “Stmt”, “Expr” o “Token”.



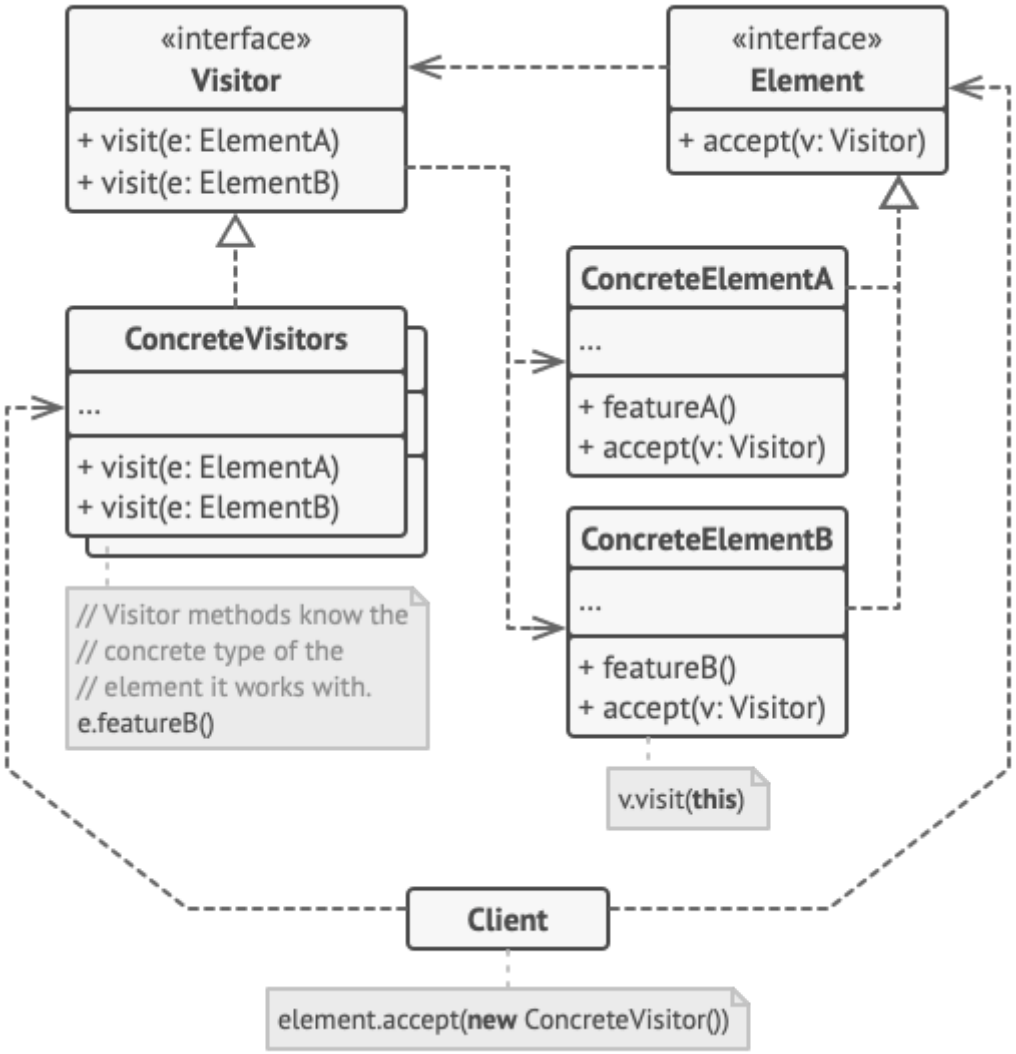
Por lo usual este tipo de proyectos suelen enfrentarse a diversas problemáticas que van desde el mismo diseño del lenguaje hasta el manejo de memoria, pero dentro de este texto se resaltará la implementación de funciones y jerarquía de clases, en el primer caso se tiene que la implementación funciones y cierres (funciones que capturan variables del entorno circundante) puede ser compleja, especialmente si se desea un manejo adecuado de los ámbitos y las variables, y en el segundo de acuerdo a la magnitud del lenguaje la jerarquía de clases puede llegar a ser compleja con múltiples tipos y subtipos.

2. Sobre el patrón “Visitor”.

El patrón visitor es un patrón de comportamiento que se utiliza para definir una función a realizar sobre un grupo de elementos, permitiendo definir dicha función sin cambiar las clases de los elementos sobre los cuales opera.

La estructuctura del Visitor parte de una interfaz la cual permitirá implementar diferentes operaciones a través de Visitors concretos ligados a la misma, los cuales desempeñan una función en específico sobre los elementos, esta interfaz debe definir métodos correspondientes a cada uno de los elementos dentro del

grupo y a su vez cada Visitor concreto debe realizar su definición de cada método de acuerdo a la función a realizar, adicionalmente cada elemento deberá incluir el método “accept()” el cual permitirá ejecutar la operación dada cuando sea requerido.



Extraído de [Visitor (refactoring.guru)]

Nótese que la función implementada sobre el elemento no se define dentro del mismo si no por fuera de él, sobre el visitor, de esta forma permitiendo separar funciones del elemento. Esto resulta enriquecedor dado que permite cumplir y gestionar principios como lo es el principio de “Open/Closed Principle” dado que permite incluir nuevas funcionalidades sin necesidad de cambiar el código base y el principio de Single Responsibility dado que permite separar funcionalidades que a primera mano no son correspondientes de una clase en específico.

3. El patrón “Visitor” dentro del proyecto.

Dentro del proyecto podemos encontrar el patrón visitor en las clases “Resolver” y “Interpreter” los cuales resultan siendo los Visitors concretos dentro del programa y los cuales agregan funcionalidades a las clases “Stmt” y “Expr” que como fue mencionado anteriormente construyen la lógica y jerarquía del lenguaje Lox junto con sus respectivas clases hijas, dentro de la funcionalidad del proyecto los Visitors “Resolver” y “Interpreter” permiten hacer las dos funciones anteriormente descritas.

> **Resolver:** La clase Resolver realiza la resolución de nombres y la vinculación de variables en el ámbito del programa. También se encarga de la resolución de funciones y la gestión de clases.
> **Interpreter:** La clase Interpreter interpreta y ejecuta el código representado por el AST. Realiza la evaluación de expresiones y la ejecución de declaraciones.

Dentro del código la estructura del patrón comienza con la definición de una interfaz para los Visitors la cual se da en las clases “Stmt” y “Expr”, definiendo un visitor para cada una de sus respectivas clases hijas

```
abstract class Stmt {
  interface Visitor<R> {
    R visitBlockStmt(Block stmt);
    R visitClassStmt(Class stmt);
    R visitExpressionStmt(Expression stmt);
    R visitFunctionStmt(Function stmt);
    R visitIfStmt(If stmt);
    R visitPrintStmt(Print stmt);
    R visitReturnStmt(Return stmt);
    R visitVarStmt(Var stmt);
    R visitWhileStmt(While stmt);
  }
```

```
abstract class Expr {
  interface Visitor<R> {
    R visitAssignExpr(Assign expr);
    R visitBinaryExpr(Binary expr);
    R visitCallExpr(Call expr);
    R visitGetExpr(Get expr);
    R visitGroupingExpr(Grouping expr);
    R visitLiteralExpr(Literal expr);
    R visitLogicalExpr(Logical expr);
    R visitSetExpr(Set expr);
    R visitSuperExpr(Super expr);
    R visitThisExpr(This expr);
    R visitUnaryExpr(Unary expr);
    R visitVariableExpr(Variable expr);
  }
```

Es importante tener en cuenta que “Stmt” y “Expr” resultan siendo el grupo de elementos sobre los que se quieren agregar los Visitors, por lo que es necesario definir la función “accept()” para que sus clases hijas puedan implementarlas.

```
abstract <R> R accept(Visitor<R> visitor);
```

```
static class Binary extends Expr {
  Binary(Expr left, Token operator, Expr right) {
    this.left = left;
    this.operator = operator;
    this.right = right;
  }

  @Override
  <R> R accept(Visitor<R> visitor) {
    return visitor.visitBinaryExpr(this);
  }

  final Expr left;
  final Token operator;
  final Expr right;
}
```

```
static class If extends Stmt {
  If(Expr condition, Stmt thenBranch, Stmt elseBranch) {
    this.condition = condition;
    this.thenBranch = thenBranch;
    this.elseBranch = elseBranch;
  }

  @Override
  <R> R accept(Visitor<R> visitor) {
    return visitor.visitIfStmt(this);
  }

  final Expr condition;
  final Stmt thenBranch;
  final Stmt elseBranch;
}
```

Una vez definidas estas bases se procede a analizar la construcción de cada Visitor concreto es decir “Resolver” y “Interpreter” que implementan de la interfaz visitor.

```
class Resolver implements Expr.Visitor<Void>, Stmt.Visitor<Void> {
```

```
class Interpreter implements Expr.Visitor<Object>,
                             Stmt.Visitor<Void> {
```

Dado que estas clases implementan de la interfaz Visitor se ven obligadas a implementar cada una de los métodos referentes a los elementos, en este punto cada Visitor definirá los métodos de acuerdo a la funcionalidad que deseé implementar en ellos, por ejemplo para un mismo método visitLogicalExpr se tiene dos implementaciones distintas, a la izquierda la realizada por el “Resolver” y la derecha la realizada por el “Interpreter”:

```
@Override
public Void visitLogicalExpr(Expr.Logical expr) {
  resolve(expr.left);
  resolve(expr.right);
  return null;
}
```

```
@Override
public Object visitLogicalExpr(Expr.Logical expr) {
  Object left = evaluate(expr.left);

  if (expr.operator.type == TokenType.OR) {
    if (isTruthy(left)) return left;
  } else {
    if (!isTruthy(left)) return left;
  }

  return evaluate(expr.right);
}
```

Toda esta estructura es inicializada en la clase “Lox” la cual a través de su método run() permite inicializar cada visitor e ingresar al mismo un grupo de statements a analizar.

```
Resolver resolver = new Resolver(interpreter);
resolver.resolve(statements);
```

```
//> resolve-statements
  void resolve(List<Stmt> statements) {
    for (Stmt statement : statements) {
      resolve(statement);
    }
  }
//< resolve-statements
```

```
//> Statements and State interpret-statements
    interpreter.interpret(statements);
//< Statements and State interpret-statements
```

```
//> Statements and State interpret
  void interpret(List<Stmt> statements) {
    try {
      for (Stmt statement : statements) {
        execute(statement);
      }
    } catch (RuntimeError error) {
      Lox.runtimeError(error);
    }
  }
//< Statements and State interpret
```

```
//> resolve-stmt
  private void resolve(Stmt stmt) {
    stmt.accept(this);
  }
//< resolve-stmt
//> resolve-expr
  private void resolve(Expr expr) {
    expr.accept(this);
  }
//< resolve-expr
```

Como se puede apreciar esta operación siempre direcciona al método “accept()” el cual permitirá ejecutar la correspondiente función de acuerdo al Visitor y a método concreto del elemento.

4. Ventajas y desventajas del patrón “Visitor” en el proyecto.

Ventajas:

1. Extensibilidad: Permite agregar nuevas operaciones al intérprete sin modificar las clases existentes. Esto facilita la introducción de nuevas funcionalidades o análisis sin afectar el código existente.

2. Separación de Responsabilidades: El patrón Visitor ayuda a separar la lógica de interpretación de la lógica de resolución y enlace. Cada componente tiene su propio Visitor, lo que facilita el mantenimiento y la comprensión del código.
3. Reusabilidad de Visitantes: Los visitantes pueden reutilizarse en diferentes partes del intérprete o en otros proyectos relacionados.

Desventajas:

1. Aumento de la Complejidad: La implementación del patrón Visitor puede aumentar la complejidad del código, ya que se introducen más clases y abstracciones. Esto puede hacer que el código sea más difícil de entender para los desarrolladores que no están familiarizados con el patrón.
2. Mantenimiento: A medida que se agregan nuevas clases de Elementos y nuevos Visitantes, puede haber un aumento en el mantenimiento del código para garantizar que todas las implementaciones estén actualizadas y funcionando correctamente.
3. Mayor Cantidad de Código: La implementación del patrón Visitor generalmente implica escribir más código en comparación con enfoques más directos. Esto puede traducirse en un mayor tiempo de desarrollo y mayor cantidad de código para mantener.

5. Alternativas al patrón.

Una alternativa simple dependiendo de la complejidad del lenguaje podría ser directamente definir las funciones dentro de cada uno de los elementos lo cual para intérpretes de lenguajes simples podría llegar a ser mucho más eficiente que construir un patrón “Visitor”, pero para lenguajes mucho más complejos la organización y flexibilidad que otorga el patrón es poco comparable, dada la cantidad de “Expressions” y/o “Statements” que se podrían tener.

6. Repositorio del proyecto y Bibliografía adicional.

1. Libro “Crafting Interpreters”, Robert Nystron, Rescatado de Crafting Interpreters
2. Repositorio del proyecto, Rescatado de munificent/craftinginterpreters: Repository for the book "Crafting Interpreters" (github.com)

3. Visitor, Refactoring.guru , Rescatado de Visitor (refactoring.guru)