Documentation of Project Implementation for IPP 2023/2024

Name and surname: Aurel Strigáč
Login: xstrig00

Introduction

This program is a PHP implementation of a interpreter program for the IPPcode24 language that has been transformed into an XML format by parser developed in the first phase of the project. This program interprets the XML-formatted source code, enabling the execution of the described instructions and logic. It effectively simulates a runtime environment where XML-based scripts are executed. The program supports a broad variety of operations ranging from basic arithmetic to ones that control the flow of the program.

Overview of classes and their implementation

Interpreter – This class is the core component of this program. It is designed to parse, evaluate and execute a sequence of instructions defined in XML format. When the Interpreter instance is created, it initializes several internal structures such as several frames (Global, Temporary and a Local), instruction list (empty at start, is filled by the `getInstuctions()` method provided by the `XMLParser` class), and many more control structures. Then, the XML document describing the program instructions is parsed using and `XMLParser`. Before execution of instructions begins, the interpreter scans through the instructions to identify and store values of all labels (LABEL instruction). This facilitates jumps during execution. The interpreter then enters an execution loop, starting from the first instruction (if the ordering is linear) or determined by a control directive. Throughout execution, various exceptions can be raised (e.g., `OperandTypeException`, `FrameAccessException`). These are caught and handled to provide meaningful error messages and to halt the execution. Execution continues until there are no more instructions to execute, or an exit-like instruction (`EXIT` instruction) explicitly stops the execution process.

InterpreterUtils - This class serves as a utility hub, providing tools and configurations for the operation of the XML code interpreter. It features `stringEscape` method to accurately interpret and convert escaped sequences in strings into their respective characters, making the string handling during execution easier. Additionally, the class defines a comprehensive list of opcodes and their expected argument types, organizing the structure and validation of command execution.

InputSetup – This class is designed to validate command-line inputs. It parses command-line arguments to extract file names for source XML and input files, utilizing regular expressions to ensure correct formatting and presence of these arguments. The class throws `ParameterException` for any misconfiguration or misuse of command-line parameters, ensuring correct setup before proceeded execution of instructions.

Instruction – Each instruction has an `opcode`, `args` and `interpreter` attributes, as well as the `execute()` method. Args is an array of arguments, consisting of name, type, value and, if the argument is a variable, a frame. The interpreter attribute is the instance of *Interpreter* class currently executing the code. For every opcode, there is a sub-class which has it's own definition of the `execute()` method, which varies by the instruction. The ability to access arguments is ensured by the `get_argN()` method. The 'N' in `get_argN()` can stand for 1, 2 or 3.

XMLParser – This class is designed to interpret and validate XML documents that define the programming instructions. It utilizes `DOMDocument` and `DOMXPath` classes to parse the XML structure, ensuring that each node conforms to expected formats and contains valid attributes for instructions and their arguments. The class checks for correctness in instruction sequences, opcode validity and argument consistency against predefined rules, throwing `InvalidSourceStructureException` when discrepancies are detected. Additionally, it supports extracting and converting typed arguments (like integers, booleans and strings) according to their XML representation, ensuring that each instruction is properly configured before it's own execution. These parsed instructions are stored in an array, sorted by their specified order and ready for the interpreter to execute. This functionality is provided by the `getInstructions()` method, which return aforementioned array of instructions.

Arg – This class, and it' many subclasses, is a representation of data. They contain the `data` attribute, which can be the numerical of a textual value, depending on type.

Design pattern

For the implementation of the interpreter, while using OOP, the Prototype design pattern was chosen. In our case, this design pattern was used in the `Instruction` class, where the Instruction class is the prototype for all the sub-classes for each instruction. This allows for easy addition of new instruction types by cloning the existing structure, as well unifying the execution method of each instruction.