Documentation of Project Implementation for 2. task IPP 2023/2024
Name and surname: Lilit Movsesian
Login: xmovse00

1 The main logic of the program

The `interpret.php` program retrieves instructions and arguments from the `DOMDocument` of the input source XML and executes the program stored in it. This implementation successfully passes level 9 of the PHPStan static code analysis tool.

2 Functional decomposition

The source code has been organized into seven classes, each responsible for providing specific functionalities. The class structure, which includes the classes already implemented in the `$ipp-core` interface, is illustrated in the Class Diagram.

2.1 Interpreter class

The main logic of the interpreter is implemented in the `Interpreter` class, which contains methods for individual instructions as well as the helper methods. The following class variables are used by methods of the `Interpreter` class.

`$instructions`: array<int, `Instruction`>: Array of instructions with instruction order as a key.
`$instructionPointer` ?int: Current instruction pointer.
`$labels`: array<string, int> Array of labels with label name as a key and order as a value.
`$globalFrame` Frame: Global frame.
`$temporaryFrame` ?Frame: Temporary frame.
`$frameStack` array<Frame>: Stack of the local frames.
`$callStack` array<?int>: Stack of call orders.
`$dataStack` array<VariableTypeData>: Stack of data.

The `execute` method iterates through instructions based on the current instruction pointer and executes the program. Methods for individual instructions validate the number and type of instruction arguments and throw corresponding exceptions for incorrect data.

2.2 InstructionParser class

The `InstructionParser` class is responsible for parsing instructions from an instance variable `$dom`, created in the `Interpreter` class using class methods provided in the `$ipp-core` interface. The `InstructionParser` iterates through `DOMDocument` nodes, validates and extracts opcodes, orders, arguments, and labels. All labels are required before the program execution to perform jumps and other connected instructions, therefore label names and their corresponding orders are stored in the array `$labels` which has the same structure as `$labels` in the `Interpreter` class. The `Interpreter` class first invokes a method `parse` of the `InstructionParser`, which returns an array of instructions and then invokes a method `$getLabels`, which returns a labels array.

2.3 VariableTypeData class

The `VariableTypeData` class represents a data structure for storing the types and data of the variables. `$type` and `$value` are string variables, wherever the integer value is required, the explicit type conversion is performed.

2.4 Frame class

The `Frame` class represents a frame containing variables. It includes methods such as `addVariable`, which adds an empty `VariableTypeData` instance to the array of variables in the frame, `getVariableList`, which gets the list of variables in the frame, `setVariable`, which sets the data of a specific variable and `getVariable`, which gets the data of a specific variable from the frame. Instances of the `Frame` class are created only in the `Interpreter` class.

2.5 Instruction class

The `Instruction` class represents a data structure for storing the operation code string `$opcode`, instruction order int `$order` and arguments array `$arguments` array<int, `Argument`|null>. Instances of the `Instruction` class are created in the `Interpreter` and the `InstructionParser` classes.

2.6 Argument class

The `Argument` class represents an XML argument with a type and a value specified in the `DOMDocument`. `$type` string represents the type of the argument.`$value` string represents its value.

2.7 StudentException class

The `StudentException` class extends the `IPPException` class. It gets the error message based on the error code using its private method `getErrorMessage` and utilizes the parent constructor class to set the error message and code.

3 Extensions

The implementation does not include any extensions.

4 Possible improvements

Refactoring the source code into an object-oriented pattern with better data organization and maintainability could improve the quality of the program. Additionally, implementing separate classes for every literal or solely for integer type could improve code organization and prevent redundant explicit type conversions.

5 UML Class Diagram

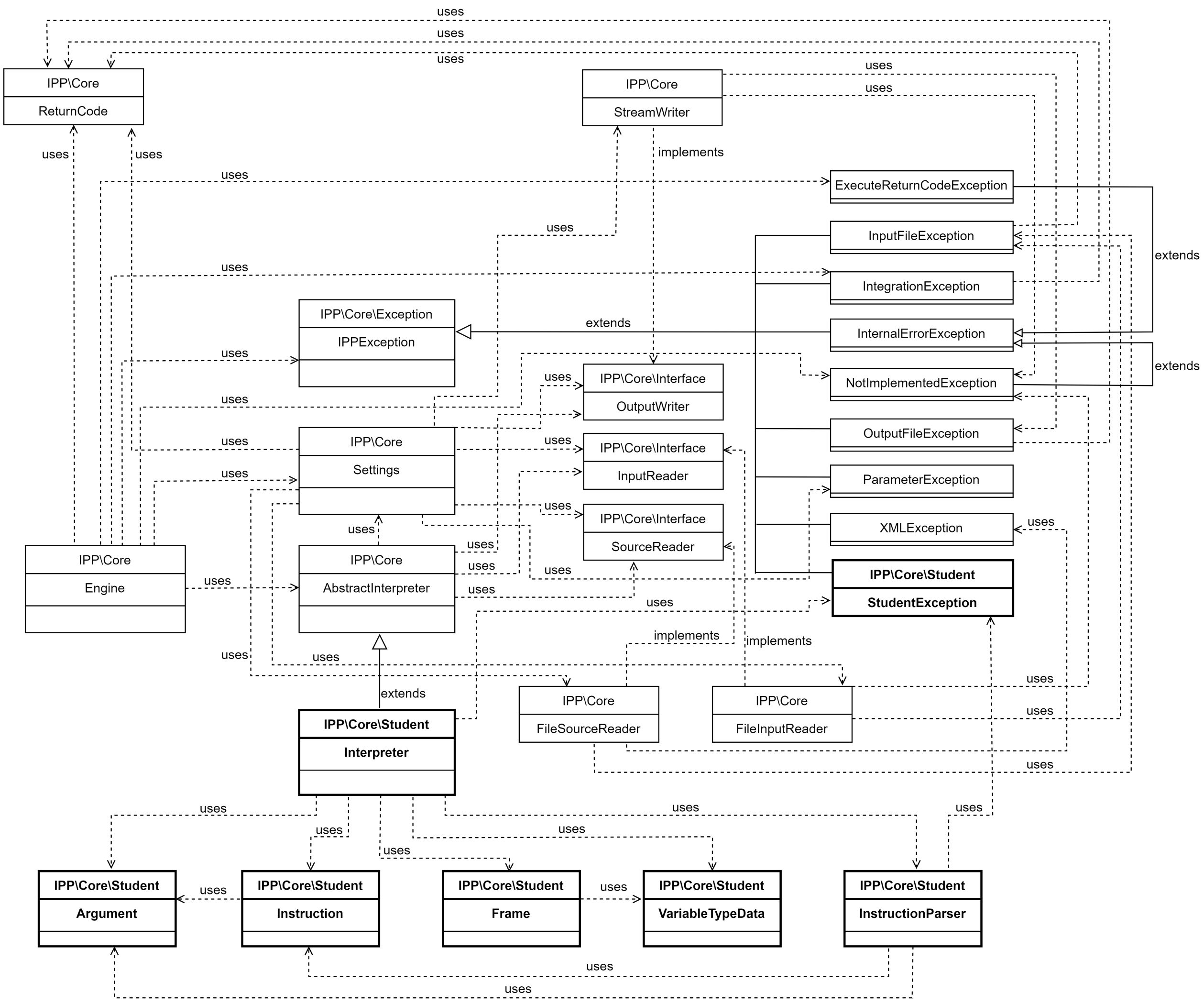

Figure 1: IPP-core Class Diagram