Documentation of Project Implementation for IPP 2025/2026
Name and surname: Adam Běhoun
Login: xbehoua00

1 Solution design

The project is implemented in PHP and complies with both the PSR-12 coding standard and PHPStan level 9. It uses three different tables to store the data. Before interpreting the XML tree, it is traversed to collect information about the defined classes, their method names, and method bodies. The built-in classes and their methods are also stored before the interpretation. Once this initial data gathering is complete, the `run` method in the `Main` class is interpreted.

2 Expansion of `ipp-core`

Three new exceptions were implemented:

- `WrongArgumentException` is thrown when the value of the argument is invalid. (exit code 53)
- `DoNotUnderstandException` is thrown when the receiver does not understand the message. (exit code 51)
- `OtherRuntimeException` is thrown when other runtime error occurs. (exit code 52)

3 Data Tables

The separate tables implement the interface called `TableInterface.php`, which defines methods for adding items to the table and printing the table for logging purposes.

Class Table: Contains the `ClassModel` elements, which store information about the name, parent class, and methods. For built-in classes, the methods are represented by their names without the bodies. Using the class table, the search for a method in the hierarchy of classes is performed.

Instance Table: All the instances (`Instance.php`) created during program execution are stored here. Each instance has a unique identifier, instantiate attributes, class name, and a value that can represent a string, number, or boolean. In addition, when creating a block instance, it is stored in the class table with a unique name along with its `value` method generated by its arity.

Symbol Table: Stores the `Symbol.php` elements along with their name and an instance ID, which refers to an element in the instance table. It is used for variable definitions and keeps track of scopes within the XML tree. In the case of "_" variable name, the symbol is not stored in the symbol table, as it is not needed later in the program.

4 XML Interpreting

The XML is interpreted using the **Visitor** design pattern. In `XMLElementVisitor.php`, the `visit` method is implemented. It recursively visits the XML nodes and processes them based on their type. Additionally, XML elements that can be uncategorized are sorted before processing.

5 Context Stack

To keep track of which instance the pseudo-variable `self` refers to, a context stack is used. It is initialized with the identifier of the `Main` class and updated as needed. For example, when the context switches to another class, its identifier is pushed onto the stack and remains there, while the code is interpreted.

6 Built-in Methods

The built-in methods are implemented in the `BuiltInCommands` folder and follow the **Command** design pattern. Each built-in method is in a separate class and implements a command interface that defines an `execute` method. In `BuiltInMethodExecutor.php`, the commands are stored with their names as keys and the corresponding command classes as values.

7 Blocks

The blocks are represented by the `Block.php` model. It stores information about the arity of the block, its body, parameter names, and the identifier of the instance that the pseudo-variable `self` refers to. The `BlockExecutor.php` further defines

how the block is executed. It first pushes the context identifier onto the stack, initializes the parameters of the block, if there are any, and then uses the `visit` method to interpret the block's body.

8 User-defined Methods

The user-defined methods are represented by the `Method.php` model. It stores information about the method's name, arity, body, and whether it is an instantiate attribute setter or getter (both are false by default). The `UserMethodExecutor.php` defines how the user-defined method is interpreted based on its type. First, it pushes the correct context onto the stack and then calls the appropriate method for execution.

9 Execution Facade

The aforementioned executors are encapsulated in the **Facade** design pattern for simplified calling within the program. The `ExecutionFacade.php` contains methods for executing blocks, user-defined methods, and built-in commands.

10 Known Limitations

The program may not correctly handle classes that inherit from the `Block` class.

11 Class Diagram

There is a class diagram below. For better visibility and less chaos, only inheritance (green) arrows are shown. Usage and dependency arrows (blue) are shown only for `Interpret.php`, which serves as the main entry of the program. However, classes that primarly interact with each other are grouped together in the diagram.