Enabling the Analysis of PHP Metadata

Parsing and analyzing Annotations in PHP Code

Bachelor's thesis

Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch/

handed in by
Michael Rüfenacht

August, 2013

supervised by
Prof. Dr. Oscar Nierstrasz
Fabrizio Perin

Acknowledgements

I especially thank Fabrizio Perin for his efforts and patience during the development of this thesis. Thank you Prof. Dr. Nierstrasz for giving me the opportunity to work at the SCG and the given inputs which helped me to finally finish my work. Further, I would like to thank Jan Kurš, Marc Wiedmer and Mircea Lungu for their reviews and questioning feedbacks. Thank you guys for taking your time.

Abstract

Quality assurance tools and metaprogramming are trending topics in PHP application development. Whilst not supported natively by the language itself, PHP applications and frameworks increasingly make use of annotations. The lack of native annotation support requires programmers to embed metadata in multi-line comments - so called doc blocks - and extract them in an additional parsing step. The separation of the actual program code and meta data into different domains and the absence of specifications in terms of syntax complicate the reasoning about annotations and corresponding behavior.

To enable the analysis of PHP source code and annotation meta data in particular, this thesis presents a modular extension to the *Moose* suite. The implemented components enable the *Moose* suite to parse and import PHP sources directly from the file system. The generated unified meta model includes types, their properties and annotations (extracted from the doc comments) and can be processed and analyzed by the *Moose* panel. Therefore *Moose* is enhanced by the ability to integrate previously separated meta information with the PHP source code and exploit their analysis.

Title: Enabling the Analysis of PHP Metadata

Author: Michael Rüfenacht

University: University of Bern, Switzerland

Department: Software Compositing Group

Supervisor: Prof. Dr. Oscar Nierstrasz

Contents

1

Introduction

1.1 Preface

The permanently evolving nature of programming languages and their ecosystems require corresponding tools to be flexible and adaptive. With a growing focus on quality assurance and testability, software analysis tasks gain importance in software development and maintenance.

Originating as a set of of Perl and C scripts, Rasmus Lerdorf in 1994 [Sev12] invented PHP[1]. PHP is a dynamic server-side scripting language designed to fit the characteristics of the internet and especially the HTTP protocol for server-side application development. The language is the most used server-side programming language for websites[2] and has a growing and active community according to the *TIOBE programming community index*[3]. Despite its popularity, especially the design of the language is controversially discussed e.g. in Alex Munroe's "PHP: a fractal of bad design" [Mun12]. Taking a closer look at the PHP ecosystem nevertheless unveils a pretty mature and advanced understanding and treatment of software engineering. Projects like Facebook's *HipHop for PHP* [ZPY+12] environment (which has static analysis capabilities) or the fact that Google recently added PHP support to their AppEngine[4] indicate the relevance of the language for bigger applications.

The increasing interest in PHP application development unavoidably leads to increasing requirements concerning the quality of the code based on a variety of metrics and hence tools capable to perform adequate analyses. Testing, debugging, profiling and static code analysis became indispensable tools of quality assurance.

1.2 Context

Popular frameworks written in PHP such as Symfony[5], the Zend Framework[6], the Doctrine Project[7] or PHPUnit[8] to name a few, enable the programmer to exploit the benefits of metaprogramming using annotations. The application framework Flow[9] even models aspect-oriented behavior [EFB01] using annotations. Since PHP does not provide native support for annotations, frameworks and tools started nesting meta data in comments called doc blocks. The application areas of annotations are similar to other languages e.g. Routing, Object-Relational-Mapping (ORM) or Dependency Injection (DI). Listing 1 presents how annotations in a doc block which belongs to an instance variable mark the property for dependency injection in the aforementioned Flow framework.

[1] http://www.php.net
[2] http://w3techs.com/
[3] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
[4] https://developers.google.com/appengine/
[5] http://symfony.com/
[6] http://framework.zend.com/
[7] http://www.doctrine-project.org/
[8] https://github.com/sebastianbergmann/phpunit/
[9] http://flow.typo3.org/

```
1    /**
2     * @Flow\Inject
3     * @var \Examples\Forum\Logger\ApplicationLoggerInterface
4     */
5    protected $applicationLogger;
```

Listing 1: Annotations in a comment for dependency injection in the Flow framework

1.3 Problem

The PHP implementation evolved significantly in the last years. While the releases of the language mostly preserve backward compatibility[10], faster release cycles and the ongoing change in terms of syntax require potential analysis tools to be as flexible and extensible as possible.

Moreover, the missing capability of the language core to expose meta data, led to some sort of a subsystem embedded in the doc blocks. While proponents emphasize the enhanced context of meta information stored directly in the source code (i.e. compared to configuration files), the distribution over PHP files makes it hard to track correlations between annotations and the corresponding logic. The absence of validation as well as normalization capabilities and the lack of specification can make systems prone to errors and side effects. Even though documentation tools allow to extract and expose annotation data to the user, they are restricted to documentation without any analysis or reasoning capabilities. The work of this thesis aims at a seamless integration of meta data analysis into a reverse engineering environment able to process PHP source code.

1.4 Solution

Enabling the analysis of PHP source code and the exposure of annotation meta data in a extensible and sustainable way requires a flexible and modular system. As there is no official annotation syntax, our solution focuses on the grammar implemented by the annotation parser of the Doctrine annotation engine[11] which is well documented and and widely used. Further, this grammar applies to all systems e.g. relying on the Doctrine ORM (such as the aforementioned Symfony and Flow frameworks and all applications based on them).

To build a parser which provides the desired characteristics in terms of flexibility and reasoning capabilities, we resorted to the *PetitParser*[12] framework which is also recommended in the *Moose* documentation. To process the generated syntax trees we made use of *intermediate representations* (IR) and node transformations in combination with visitors to build a unified meta model processable by the *Moose*[13] suite. The *Moose* suite is able to analyze, query and visualize the resulting IR depending on predefined or custom metrics. *Moose* is able to apply full featured reverse- and re-engineering and analysis

[10] code of older versions is valid for newer versions of the interpreter
[11] https://github.com/doctrine/common
[12] http://scg.unibe.ch/research/helvetia/petitparser
[13] http://www.moosetechnology.org

techniques to any suitable source code meta representation.

The remainder of this thesis is organized as follows: Chapter 2 briefly introduces the the tools used and describes important characteristics of PHP and the state of annotations. Chapter 3 explains the different components of the implementations and illustrates their interplay. Chapter 4 provides insights to the achievements and proposed solution's analysis capabilities and Chapter 5 concludes.

2

Background

2.1 The Tools

To achieve seamless integration of parsing, transformation and analysis, we propose the combination of *PetitParser* and *Moose*, both written in Pharo Smalltalk. More generic information on how to create and integrate analysis using *PetitParser* and *Moose* can be found in work done supplementary to this thesis [Rü13].

2.1.1 PetitParser

The *PetitParser* [RDGN10] framework is a framework for programmatic parser and grammar composition. *PetitParser* creates top down recursive parsers and exploits the benefits of four parsing techniques presented in Table 2.1.

Scannerless Parsing [Vis97] avoids lexical analysis step and simplifies the syntax in a single formalism	**Parser Combinators** [Hut92] creates a graph of primitive parsers to establish combinatory parsing
Parsing Expression Grammars (PEG) [For04] provides ordered choice and additional predicates to enable the recognition of non-context free languages	**Packrat Parsing** [For02] guarantees linear parse time through memoization and infinite lookahead

Table 2.1: Parsing techniques adopted by *PetitParser*

PetitParser provides a concise internal domain specific langauge [Fow10] (DSL). The creation of parsers with *PetitParser* is modular, reusable and flexible which enables extension, reification and reasoning on the modelled languages. In contrast to parser generators which produce static systems, parsers built with *PetitParser* can be composed, changed and inspected dynamically which enables them to parse heterogeneous systems. *PetitParser* allows flexible composition of parsers and node transformations to prepare the parsing result for further processing which makes it a good choice for the integration into *Moose*.

2.1.2 Moose

Moose [DLT00, DLT01] is a software and data analysis platform. *Moose* provides a language-independent environment for reverse- and re-engineering purposes, offering a variety of analysis capabilities on dedicated unified meta models of structured data. Besides the ability to establish custom reasoning on structural representations, the platform provides a number of generic metrics, queries, mining algorithms, browsers and visualizations. In contrast to similar tools, *Moose* is not restricted to a specific functionality such as specific metrics or visualizations and enables flexible creation of views and reports. Also, analysis can be done using a convenient user interface called `MoosePanel`. The possibility to attach custom importers

and the integration of a unified meta model called *FAMIX* [Tic01] make *Moose* inherently extensible and open to any kind of source code analysis or analysis of structured data in general.

2.2 The Language

PHP[1] lacks a formal specification and is therefore defined through its (open source) reference implementation at `http://php.net`. To sum it up, PHP is an imperative, dynamic scripting language designed for server-side application development, by default interpreted by the *ZendEngine*[2] (consisting of an interpreter and a virtual machine). The syntax of PHP consists of numerous elements adopted from C or Perl syntax. Next to a number of interfaces for servers accessing PHP (Server-API, SAPI), it is also used for command line scripting via the command line SAPI.

Programs written in PHP are usually organized in files with the *.php* extension which can be dynamically included and executed at run-time. PHP source code needs to be enclosed in dedicated delimiters `<?php` and `?>` and can directly be embedded in HTML markup generating valid XML syntax. Embedded HTML is ignored by the interpreter but makes PHP a heterogeneous language by design. In parallel to the initial procedural design, version 3 introduced an object model for full object-oriented programming support. At the time of writing, PHP supports - besides the procedural elements - class based constructs such as interfaces, classes and traits with a single inheritance hierarchy. The type system is dynamically and weakly typed relying on *duck typing*[3] and name equivalence. As pointed out by Zhao et al. [ZPY+12] or Hills et al. [HKV13], the dynamic nature and flexibility of the language complicates static analysis. Even the internal *Reflection API*, which provides reflective behavior enabling introspection or the object model's basic intercession capabilities, have reduced access to the dynamic language features such as variadic[4] functions and methods or dynamic invocations.

Listing 2 presents a contrived method which resolves a controller i.e. in a model view controller (MVC) based application. The snippet demonstrates the difference between a default instantiation and invocation in and the usage of a variadic method and dynamic invocation. This may look tedious, but allows dynamic method invocation without reflection or restrictions of the signature. Static analysis is not able to track the method parameters or the dynamic invocation without further effort. Especially if types and procedures are resolved at run-time, based on information that resides outside the domain of the analysis e.g. user input or additional configuration.

2.2.1 Grammar

The official PHP documentation is a collection of examples on how to use the language, is partially incomplete and does not give any grammar specification. PHP provides an internal tokenizer function[5]

[1] initially standing for *Personal Home Page*, renamed to the recursive acronym *PHP: Hypertext Preprocessor*
[2] Zend `http://www.zend.com` *January, 2013*
[3] Valid semantics depend on the actual set of methods and properties of an object.
[4] Methods and functions can take an arbitrary number of parameters without defining them in the declaration.
[5] PHP: token_get_all - Manual `http://www.php.net/manual/en/function.token-get-all.php` *June, 2013*

```
1   public function executeController($id, $title){
2      $controller = new BaseController();
3      return $controller->execute($id, $title);
4   }
5
6   public function executeController(){
7      $arguments        = func_get_args();
8      $controllerClass  = 'BaseController';
9      $controller       = new $controllerClass;
10     $executorMethod   = 'execute';
11     return \call_user_func_array(array($controller, $executorMethod), $arguments);
12  }
```

Listing 2: Two equal methods instantiating a controller and invoking a method (methods are marked with the function keyword) to demonstrate dynamic features of PHP

`token_get_all(string)` — which performs lexical analysis on any arbitrary string — to tools written in the language itself.

The official Github repository[6] of PHP hosts the source code of the *ZendEngine* containing two particular files of interest: *zend_language_parser.y* and *zend_language_scanner.l*. To be more precisely, these files contain a syntax specification and lexer definition for parser generators such as Yacc [Joh75] which generate LALR [Der69] parsers. Solutions such as *PHP-Parser*[7] use implementations similar to YACC or derivates, like PHP itself which internally uses GNUBison[8].

While these contents provide the necessary specifications to derive a reliable grammar and build a robust parser upon, they imply some characteristics of the grammar such as (indirect) left-recursiveness and ambiguity. Both are hard to handle in recursive top-down [RS70] PEG parsers as generated by the *PetitParser* framework and usually require extensive rewriting of the grammar (in contrast to bottom-up parsers produced by the mentioned generators). As a consequence rewritten or adapted grammars are almost impossible to validate in terms of the accepted language.

2.2.2 Namespaces

Introduced in PHP 5.3, namespaces were quickly adopted by projects following an object-oriented approach. Replacing tedious class naming conventions, namespaces enabled cleaner organization of source code, simplified the avoidance of name collisions and improved dynamic class-loading mechanisms. In absence of a dedicated annotation-type, annotations in PHP are defined as classes. To enable the desired analysis capabilities and extraction of the corresponding logic, namespaces have to be properly resolved. The corresponding chapter of the manual defines the terms listed in Table 2.2 used in the following subsections.

[6] zend-src `https://github.com/php/php-src/tree/master/` *January, 2013*
[7] `https://github.com/nikic/PHP-Parser`
[8] `http://www.gnu.org/software/bison/`

Term	Description	PEG
segment/identifier	*a part of the namespace (subspace)*	[a-zA-Z‘_’][a-zA-Z0-9‘_’]*
separator	\ *separating the segments of a namespace*	‘\’
unqualified name	*the class name (a single identifier)*	identifier
qualified name	*identifiers divided by at least one separator ending with the unqualified name*	(segment separator)+ identifier
fully qualified name	*qualified or unqualified name prepended with a separator (absolute namespace)*	separator (segment separator)* identifier

Table 2.2: Namespace term definitions and derived PEG equivalents

2.2.2.1 Namespace Declaration & Aliasing

Namespaces affect four structural elements of the code: classes, interfaces, functions and constants but do not constrain the file or folder structure of a PHP application. Namespaces are declared in two ways:

Initial namespace statements The initial namespace must be declared in the initial top level statement of a file and applies to all code up to the next namespace statement or the end of the file. Listing 3 gives a short overview over the syntax.

```
1  namespace Framework\Collections;
2  class Iterator {
3  }
```

Listing 3: Definition of a class in a namespace

Namespaced blocks Declared namespaces apply to a statement block as shown in Listing 4. Files structured in that manner are not allowed to have any statements outside containing namespaced blocks.

```
1  namespace Framework\Collections {
2     class Iterator {
3     }
4  }
```

Listing 4: Definition of a class in a namespace block

To avoid redundant and tedious writing of fully qualified names for the affected language constructs, PHP allows namespace aliasing. It is possible to alias parts of the namespacing using the `use nspath as alias` construct, usually referred to as *import-rules*. Listing 5 shows how the previously (Listing 3 and 4) declared classes can be aliased and referenced.

```
1  namespace Framework\Application;
2  use Framework\Collections as Collections;
3  use Framework\Sets\Set; //as Set;
4
5  $it = new Collections\Iterator();
```

```
6    $st = new Set();
```

Listing 5: Instantiation of namespaced types

2.2.2.2 Namespace Resolution

The *ZendEngine* resolves namespaced identifiers at run or compile-time, depending on the existence of import-rules concerning the accessed type. Table 2.3 lists the rules that apply during the resolution of namespaced types.

Resolution of named types with or without (marked with −) affecting import rules.

Name	Import-Rule	Example	Comment
fully-qualified	−	\A\B →\A\B	*access without resolution (absolute), independent of the current namespace*
un-/qualified	use A\B\C as C	C\D →A\B\C\D C →A\B\C	*resolved at compile-time, independent of the current namespace*
un-/qualified	−	namespace A\B C\D →A\B\C\D	*the current namespace is prepended at run-time*

Table 2.3: Overview over namespace resolution rules

2.3 Annotations & Doctrine

The application domain of meta data residing in PHP doc block comments is not only documentation anymore. They provide a way to expose program context without any influence on the semantics of the code. A plethora of frameworks and libraries started to exploit the benefits of an additional meta layer at run-time. Initialized by the capability of PHP's Reflection API (since version 5.1) to access doc block comments, a number of libraries came up targeting the ability to extract annotations comparable to JAVA before JSR-175[9]. Listing 6 shows the difference between ordinary comments and a doc block which is accessible using reflection. The PHP interpreter therefore creates different lexical tokens and while the comments are ignored by potential opcode caches, the doc blocks are preserved.

```
1    // single line comment
2    /* multi line comment */
3    /**
4     * a doc block
5     */
```

Listing 6: PHP comments (T_COMMENT token) compared to a doc block starting with /** (T_DOC_COMMENT token)

[9] http://jcp.org/en/jsr/detail?id=175

Despite the growing usage, requests for built-in annotation support were declined[10] or are still in discussion[11]. The demand for metaprogramming capabilities led to several different syntactical approaches on exposing and handling data. One particular approach grew to a de-facto standard: the Doctrine 2[12] annotation engine. Figure 2.1 briefly depicts the impacts on PHP annotation support of the last decade.

Figure 2.1: Impacts on annotation support

The Doctrine Project is a set of libraries mainly designated for persistence purposes. Especially its object relational mapper is integrated in numerous frameworks, also resorting to the included annotation engine. PHP's annotations are comparable to the JAVA version (in contrast to attributes in C#, or pragmas in Smalltalk) as exposed by the grammar of Doctrine's internal parser (Appendix B). The grammar indicates two types of annotations, deconstructed in Figure 2.2, namely *marker annotations* and *parameterized annotations*. The annotation name is a qualified name and gets resolved to a class name (as described in 2.2.2.2). All defined annotations are themselves annotated as `@Annotation`. Their constructor takes an associative array of arguments which means that the corresponding constructor does not explicitly specify which parameters are valid, hence possible values are derived from their instances and may be incomplete.

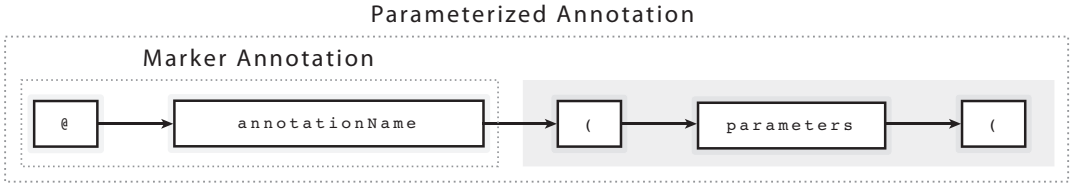

Figure 2.2: Types of annotations deconstructed

2.4 Related Work

A list of static analysis tools for PHP can be found on `http://phpqatools.org/`. Most of these tools provide analysis concerning a limited set of metrics or a dedicated ruleset. As already mentioned, *HipHop for PHP*[13] also has static analysis capabilities. According to their documentation *HipHop for*

[10] Request for Comments: Class Metadata `https://wiki.php.net/rfc/annotations`, *January, 2013*
[11] Request for Comments: Annotations in DocBlock `https://wiki.php.net/rfc/annotations-in-docblock` *June, 2013*
[12] Doctrine, the doctrine project `http://www.doctrine-project.org/`, *January, 2013*
[13] https://github.com/facebook/hiphop-php/

PHP is restricted to PHP version 5.2, which is significantly different in terms of features compared to 5.3 (which e.g. introduced namespaces, closures, late static binding). In general, these tools neither provide interactive analysis capabilities like *Moose* does, nor do they support the analysis of annotations.

3

Architecture

3.1 Overview

The chosen approach to enable the analysis of source code written in PHP mainly depends on 3 stages: The parsing, the generation of an AST and transformation into the unified *FAMIX* model followed by the import into *Moose*. Figure 3.1 depicts an overview of correlations between the different stages, their result and the responsible actor.

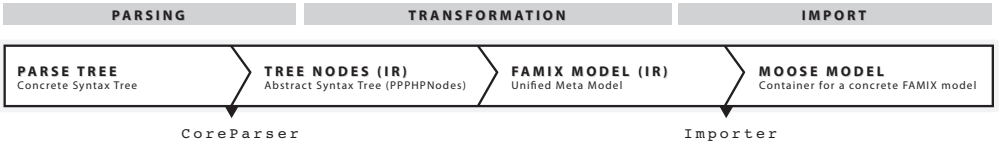

Figure 3.1: Overview over the stages the application runs through, the generated entities and the respective main actors.

As a consequence to the previous elaborations in Chapter 2, the example implementation is subject to some restrictions.

1. The implemented parser adopts the features and syntax of PHP 5.3.X which is the minimum requirement for the Doctrine annotation engine.

2. Due to the left-recursiveness, ambiguity and complexity of the grammar sources discovered in Section 2.2.1, the concrete implementation does not directly correspond to the reference implementation.

3. The implemented solution focuses on an abstract system view, sufficient to enable annotation analysis. Hence, it extracts structural elements such as types and does not explicitly analyze statements and expressions after the parsing. Nevertheless the core parser is able to parse them and can be extended to support analysis of procedural elements e.g. to compute metrics like cyclomatic complexity.

4. Dynamic class aliases not done using the mentioned *import-rules* cannot be resolved and have to be inspected manually in the browser.

3.2 Parsing

Parsers for large grammars based on the *PetitParser* framework can be established following a simple inheritance pattern. Figure 3.2 illustrates the general way to derive a parser and the concrete implementation containing an additional layer. The `PPPHPCoreShared` parser was introduced to have a lightweight superclass for the annotation parsers further explained in Section 3.2.1.

Every parser along the hierarchy performs further transformations and refinements on the structural representations generated by its superclass. The grammar parser specifies a classical set of symbols and productions, while the core parser reifies the results by converting the concrete syntax tree into an initial IR which consists of dedicated entities. The *PetitParser* framework provides a base class for the

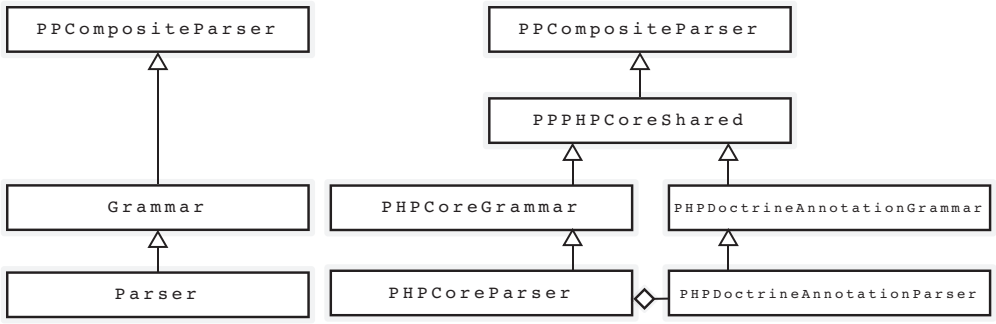

Figure 3.2: Pattern to create a parser and the approach token in the concrete implementation.

implementation of grammars in the form of a PEG parser, the `PPCompositeParser`. A `PPCompositeParser` hosts primitive parser combinations which are composed to productions stored in an accordingly named instance variable. Every instance variable represents a parsing rule and holds the corresponding parser (instantiated on initialization by equally named accessors). Early initialization allows to exploit caching, enables mutual recursive rule definitions and makes the code expressive and readable.

3.2.1 Shared

The `PPPHPCoreShared` parser was introduced to have a lightweight superclass for the annotation parser. Besides a number of helper methods for syntactic sugaring, it hosts parsing rules for scalar data types and basic namespace parsing mechanisms to ensure consistent behavior across the core parsers and modules hooked in the parsing process. We preferred inheritance over horizontal because the *PetitParser* framework and especially its testing environment provides limited support for dependencies between composite parsers.

3.2.2 Grammar

The `PPPHPCoreGrammar` is the basic parser of the application and implements the fundamental literals and productions of the grammar not covered by the shared parser. The construction of such a parser benefits from a reliable grammar specification. Unfortunately, PHP and the corresponding documentation do not provide a suitable specification, hence the written grammar of the example implementation is a derivation from real-world examples. Transformations towards PHP's internal parser generator specifications would require further investigations. Especially the expression syntax is utterly complex in its standard implementation, modelling specific rules for different contexts. The corresponding documentation[1] states:

> *"PHP takes expressions much further, in the same way many other languages do. PHP is an expression-oriented language, in the sense that almost everything is an expression."*

[1] `http://www.php.net/manual/en/language.expressions.php`

The grammar defines numerous special cases for expressions, using different expression terms depending on the specific operator and the position of the term i.e. if it stands on the right or left side of the operator.

We created a subclass of the `PPCompositeParser` - the `PPPHPCoreGrammar` - which models the base grammar and creates an initial AST consisting of collections of literal parsing results. To handle the complex expressions we made use of the `PPExpressionParser`, able to overcome difficulties occurring in expressions e.g. operator precedence. In contrast to PHP's expression rules the `PPExpressionParser` only supports the definition of a single expression term which makes it hard to eliminate possible ambiguities in expressions.

Listing 7 presents how a class declaration is modelled. Instance variables such as `comment` or the `classKeyword` are literal parsers or parser combinations created using the `asParser` method and returned by the corresponding accessor e.g. the `classKeyword` method. Comments before the class are directly taken into account in the `classDeclaration` production, to preserve the context for further processing and annotation parsing.

```
1  PPPHPCoreGrammar >> classKeyword
2     ↑ 'class' asParser trim
3
4  PPPHPCoreGrammar >> classDeclaration
5     ↑  comment optional,
6        classModifiers optional,
7        classKeyword,
8        className,
9        (extends, fullyQualifiedClassName) optional,
10       (implements, (fullyQualifiedClassName separatedBy: comma) withoutSeparators)
     optional,
11       classBody
```

Listing 7: The classDeclaration production of the grammar

3.2.3 Core Parser

The `PPPHPCoreParser` enhances the parsing rules (i.e. composed primitive parsers) created by the grammar to `PPActionParsers`. Action parsers apply a block closure on any successful parse result (i.e. collections of literals). The executed blocks transform the parsing results to according tree node objects. The resulting IR is a tree structure consisting of dedicated nodes which serve two core purposes: convenient access to the properties of the code's structural elements and the exposure of entry points for visitors (elaborated further in Section 3.3). In our case the `PPPHPCoreParser` has additional responsibilities (besides its core functionality of AST transformation and IR generation).

1. **Resolution of namespaces**
   The parser implements the *ZendEngine*'s namespace resolution policy introduced in Table 2.3 by hooking into the parsing of the corresponding productions. The parser builds up a dictionary of *import-rules* i.e. an alias table and resolves affected type names during the parsing to its fully

qualified namespace name. This makes the parser stateful. To preserve its idempotence[2] we flush the stored aliases at the according productions e.g. the end of a file.

2. **Application of the annotation parser**
   To make the annotation resolution as pluggable as possible, the core parser different annotation parsers to be hooked in by extending the annotation rule and add additional choices. At the moment, the implementation only supports the elaborated doctrine annotation syntax using an `PPPHPDoctrineAnnotation` parser (Section 3.2.4) as `#doctrineAnnotation` production.

Listing 8 presents how the parser generates an element of the IR i.e. a `PPPHPClassNode`. The parser transforms the initial parser created by its superclass to an action parser using `==> aBlock`. The applied block first resolves the namespaces and creates a class node. Then properties of the class are unwired: the modifiers (i.e. 'abstract' or 'final'), annotations in the doc block, super types and members, namely methods variables and constants, are added using a double dispatch. Partial rules such as the class members are already resolved due to the recursive top down parsing (depth first algorithms). After processing all elements the block returns the generated class node.

```
1  PPPHPCoreParser >> classDeclaration
2     ↑ super classDeclaration
3        ==> [ :token |
4           | classNode localNamespace |
5           localNamespace := self currentNamespaceResolve: token fourth.
6           classNode := PPPHPClassNode createFromNamespaceStack: localNamespace.
7           classNode addModifier: token second.
8           self setupDocBlockAndAnnotationsOn: classNode fromToken: token first.
9           self setupSuperTypesOn: classNode fromToken: token fifth.
10          self setupInterfacesOn: classNode fromToken: token sixth.
11          token seventh do: [ :member | member addToParent: classNode ].
12          classNode ]
```

Listing 8: Transformation of a classDeclaration to its intermediate representation i.e. a PPPHPClassNode

3.2.4 Annotation Parser

The `PPPHPDoctrineAnnotationGrammar` and the derived `PPPHPDoctrineAnnotationParser` are subclasses of the `PPPHPCoreShared` and embrace the same strategy as the core grammar and parser do. The annotation parser implements the grammar listed in Appendix B and generates dedicated `PPHPAnnotationNodes`. To prevent the annotation grammar from consuming documentation tags, a blacklist is maintained for standard documentation tags[3]. To ignore more than these tags could cause the parsing to exclude expected results.

[2] freedom of side effects if executed repeatedly [DKSJ12]

[3] `http://www.phpdoc.org/docs/latest/for-users/phpdoc-reference.html`

3.3 Visiting Intermediate Representations

The core and the annotation parser both wrap the literal parsing results into entities suitable for further processing. The generated AST nodes are value objects arranged into a traversable tree and provide an `accept: aVisitor` method (according to the visitor pattern). The usage of visitors in combination with double dispatches allows us to process arbitrarily complex data structures without type checks. Due to the dynamic typing of Pharo, the visitor needs to implement concrete methods for each tree node type. The visitor connects the parsing step to the importing process by delegating transformations of the tree nodes to an importer which generates the final IR, an unified *FAMIX* model. Listing 9 presents the usage of the visitor pattern in a `PPPHPClassNode` and how the `PPHPConcreteVisitor` delegates the node transformation to its dedicated importer. A more in depth description of the workflow is depicted by Figure 3.3.

```
1  PPPHPClassNode >> accept: aVisitor
2      ↑ aVisitor visitAPPPHPClassNode: self
3
4  PPPHPConcreteVisitor >> visitAPPPHPClassNode: node
5      ↑ self importer ensureAClass: node
6
```

Listing 9: Usage of the visitor pattern to traverse the IR. The double dispatch performed between the visitor and the node enables the necessary context in terms of type for the importer.

3.4 The Importer and Famix

The extension of *Moose* is based on importers. Importers are handling the generation and transformation of data into a suitable representation. For the particular case of source code, the *Moose* suite provides the *FAMIX* meta model, a language agnostic structural representation of object-oriented programs. Besides the creation of structural representations, the importer maintains types defined in a system (comparable to a symbol table) and creates appropriate relations amongst them.

Listing 10 presents the process of ensuring a class, called by the visitor in Listing 9. The importer creates a type for the passed class node i.e. a `FAMIXClass`. While creating the class node the importer consults a dictionary which contains all types already explored to prevent duplications and ensure consistency amongst all types in the system. The `ensureAType` selector collects methods and fields of the type by reassigning the visitor to child nodes, analogous to the handling of the interfaces at the end of the listing before returning the `FAMIXClass` object.

As explained in Section 3.3 the importer and the visitor are members of each other. The importer designates the traversal strategy, maintains the context of *ensured*[4] nodes and reassigns the visitor to branches of the tree. The visitor guarantees a type independent treatment of the nodes by performing a double dispatch. Figure 3.3 illustrates the process of importing, especially the interplay of visitor and importer.

[4]the process of transforming the nodes into their respective *FAMIX* representation and collecting information about the existence of structural elements e.g. defined classes

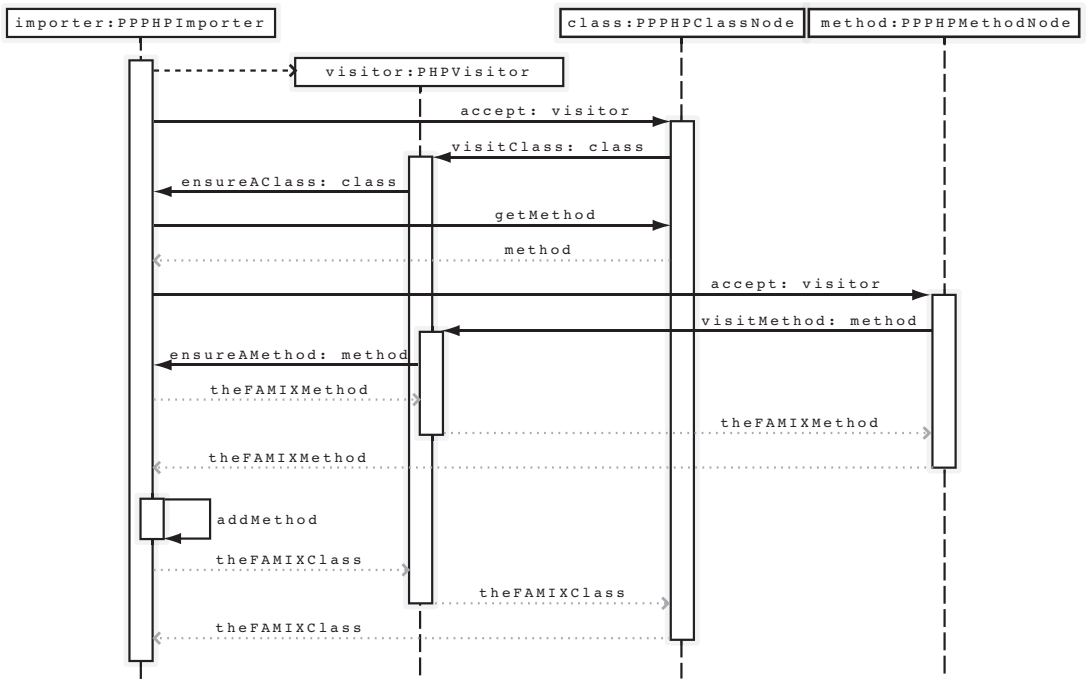

Figure 3.3: Excerpt of the importer workflow.

To achieve a convenient access, the importer provides an import method taking a folder reference as an argument. After recursively indexing the folder and filtering *.php* and *.inc* files, the importer uses the core parser to parse the code and attaches the visitor to the result (which refers back to the importer as already elaborated). After finishing the importing process, the importer populates a `MoosePHPModel` with all tracked entities and returns it.

```
1  MJPHPImporter >> ensureAClass: aPPPHPClassNode
2      | classNode inheritance node |
3      classNode := self createType: aPPPHPClassNode.
4      self ensureAType: classNode from: aPPPHPClassNode.
5      aPPPHPClassNode interfaces
6          do: [ :interface |
7              inheritance := FAMIXInheritance new.
8              inheritance superclass: (interface accept: self phpVisitor).
9              inheritance subclass: classNode ].
10     ↑ classNode
```

Listing 10: The process of ensuring a class, consisting of type creation and population with its members.

3.5 The MoosePHPModel

The `MooseModel` is a *Moose* internal container or grouping entity that allows *Moose* to manipulate and manage entire models. After adding all *FAMIX* entities to the moose model, it provides an installing mechanism[5] which makes the meta representation available in the `MoosePanel`. As soon as the model is installed it is ready to be queried, visualized and measured. We derived a MoosePHPModel which is able to integrate PHP specific entities into the `MoosePanel` by exposing accessors e.g. for all classes or methods in the system. In our case the MoosePHPModel mainly handles the integration of annotations and classes without loosing the default analysis capabilities of *Moose* since we use slightly changed *FAMIX* nodes. Listing 11 presents an example of one of these accessors which filters interfaces and makes them available in the top level navigation of the `MoosePanel`. The enabled analysis capabilities are elaborated further in Section 4.2.

```
1 allInterfaces
2    <navigation:'All interfaces'>
3    ↑ self allClasses select: [:item | item isInterface ]
```

Listing 11: Example of an accessor in the MoosePHPModel which selects interfaces and is annotated to be included in the MoosePanel's navigation

3.6 The Import Command - Wrap up

Our implementation provides an import command which is integrated in the UI of the `MoosePanel`. The import command allows a convenient access to the established functionalities and aggregates the mentioned components. The import command mainly triggers the file picker of the environment and passes the selected directory reference to the dedicated importer method before it installs the returned model.

3.7 Discussion

The most important characteristics of the chosen approach is its modularity and extensibility. All stages of the processing can be extended and provide entry points for enhancements. If implemented properly, the usage of visitors allows the importing mechanism to work independent from the complexity of the traversed IR. Since PHP is backwards compatible, parsers for newer versions can be derived from the core parser only implementing new features.
Whilst the solution follows an architectural approach similar to the implementations for Smalltalk or Java it still has its weaknesses in terms of coupling or separation of concerns. A good example to consider is the missing separation between the core parsing and the namespace resolution which makes the parser stateful and could therefore be prone to side effects. Also, the concrete population of the model happens iteratively instead of relying on a visitor.

[5] Which adds the model to the *Moose* cache and makes it available in the interface

4

Achievements

4.1 Parsing

As stated, the core parser is not based on a suitable specification and therefore almost impossible to be validated. To get an overall impression of the quality of the parser, we did evaluate a number of software systems written in PHP and analyzed possible errors and their sources. Table 4.1 presents an overview of a number of parsed systems and error ratio of the parser. The failures column lists the number of files the parser was unable to process in comparison to the number of files with the dedicated *.php* and *.inc* extensions. The table also lists the source lines of code (SLOC[1], not including empty lines and comments) to illustrate the size of the projects. There are three main reasons the parser failed listed below and depicted in Figure 4.1.

1. Files that do not contain PHP code result in a parse error. Whilst perfectly fine for the *ZendEngine*, the parser expects the files to contain any PHP statement, to avoid false positives when parsing PHP nested in HTML.

2. Even though all system specifications state a minimum requirement of PHP version 5.3.* or earlier, some files contain code written in PHP 5.4.*. These components mostly make use of the new traits construct and are included at run-time depending on the system's PHP version.

3. The grammar suffers from not being fully accurate in comparison to the reference implementation of the *ZendEngine*. Unexpected placement of comments (they can be literally everywhere) and collision of reserved names and identifiers can lead to errors during the parsing of expressions (e.g. static method calls with a literal identifier `self::NULL()`). Up to now we were not able to rewrite the expression engine such as it is fully able to parse these cases without causing others to fail.

4.2 Analysis Capabilities

The integration into *Moose* allows us to browse and visualize the structural representation of the source code and apply metrics. We are able to load and parse PHP source directly from the file system and create a structural representation including annotations for doctrine based systems. The following figures illustrate some of the core functionalities of *Moose* analyzing PHP code. We are able to analyze PHP systems in terms of system complexity (Figure 4.2), perform queries e.g. track occurrences of PHP's *magic methods*[2] (Figure 4.3), visualize and browse namespace constellations (Figure 4.4 and 4.5) and analyze annotation constellations and relationships (Figure 4.6 and 4.7), to name a few examples.

4.3 Discussion

We did create a problem oriented implementation to enable basic PHP source code and annotation meta data analysis. Whereas we achieved the desired requirements to the system such as extensiblitiy and

[1] calculated using the CLOC tool `http://cloc.sourceforge.net/`
[2] interceptor methods

System	Description	Version	Files	SLOC	Failures
Wordpress	*Blog Platform*	3.6	495	131'075	0
PHPUnit	*Testing framework*	3.7.24	374	29'338	0
Drupal	*Web CMS*	7.22	274	83'460	1
phpMyAdmin	*Mysql database management software*	4.0.5	410	133'980	0
Zend Framework	*Application framework*	2.2.2	2202	150'477	13
CodeIgniter	*Application framework*	2.1.4	147	24'382	1
Symfony	*Application framework, (standard distribution with vendors)*	2.3.3	2510	155'020	10
Doctrine	*Persistence & Component framework*	2.3	575	45'189	0
Monolog	*Logging framework*	1.6	52	2'523	0
Assetic	*Asset management framework*	1.1.2	97	5'239	0
Swiftmailer	*Mailer library*	5.0.1	166	8'633	0
Twig	*Templating engine*	1.13.2	185	8'299	0
Joomla	*WebCMS (full package)*	3.1.5	1850	188'894	0
TYPO3	*Web CMS*	6.1.3	3557	305'590	19
Total:			12'894	1'272'099	44

Table 4.1: Parsing results on a corpus of PHP systems, resulting in an error ratio of ~0.003

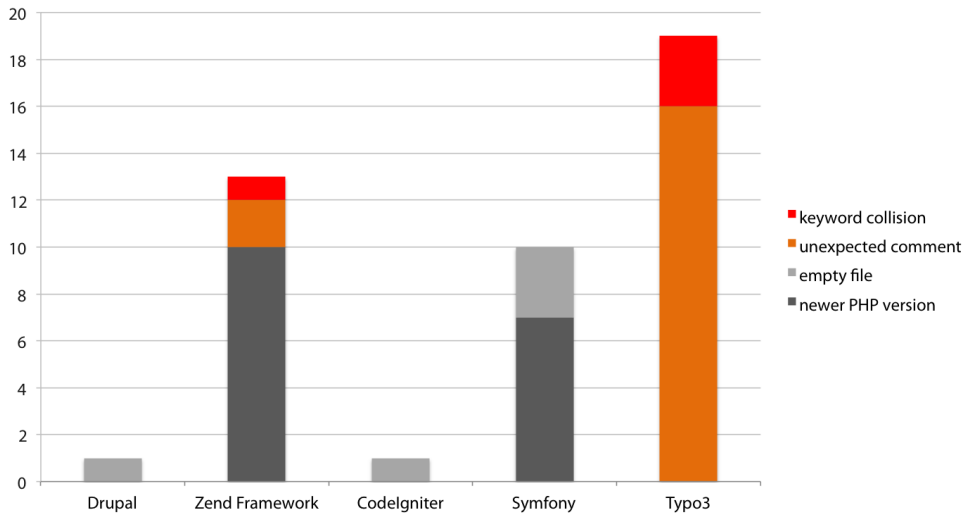

Figure 4.1: Error distribution in the tested frameworks.

flexibility, the implementation leaves room for improvements.

The processing steps after the initial parsing ignore procedural code constructs and only track structural entities such as classes and interfaces including their members, functions and all containing namespaces. To fully exploit *Moose*'s analysis capabilities, further investigations in a more fine grained analysis of statements and expressions are necessary. Also, it is nearly impossible to reason on the quality of the parsing results and expressiveness of the grammar. There are no dedicated specifications or validation possibilities than to collect empirical data which also is true for the annotation handling. To prevent the annotation analysis from suffering from possible interferences with documentation tags we added filters to the system that prevent it from including annotations not defined in the system possibly excluding elements of interest (its possible to overcome these limitations by manually querying the system in *Moose* or generating custom reports).

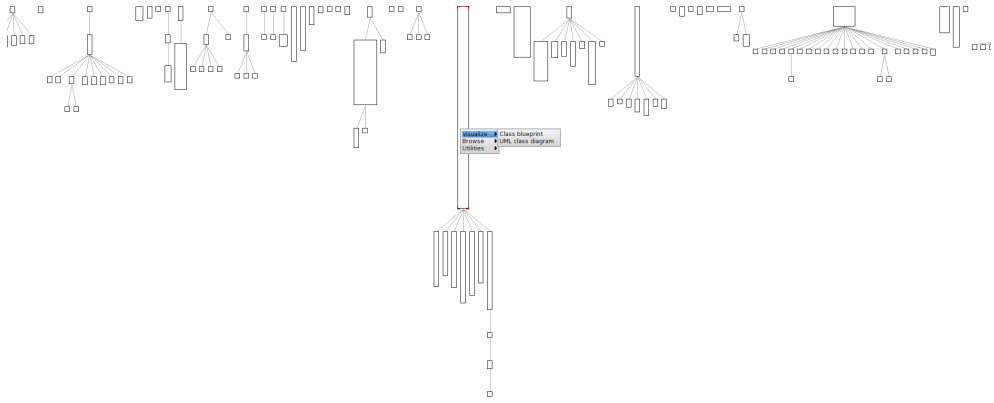

Figure 4.2: Excerpt of an interactive system complexity view.

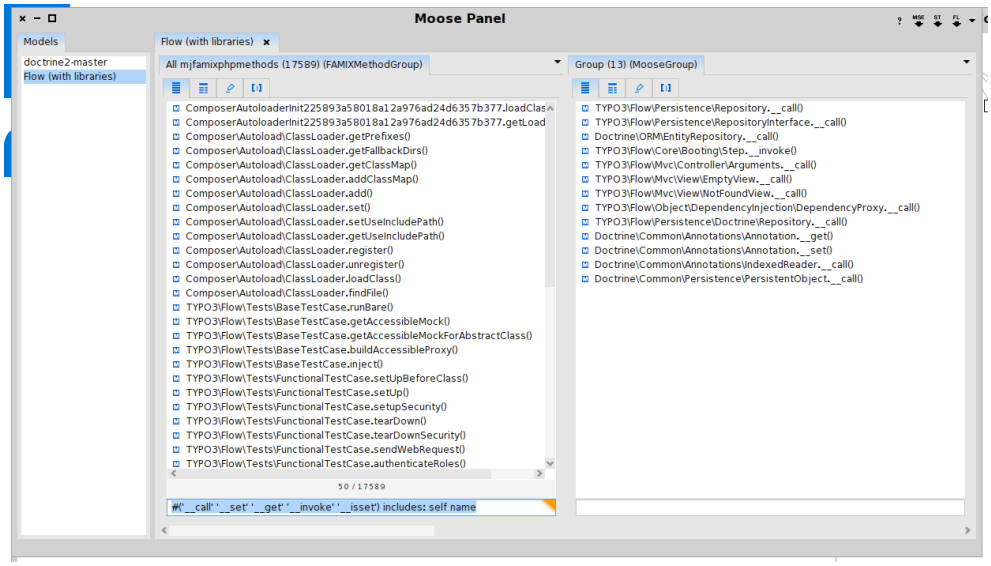

Figure 4.3: Perfoming queries is possible directly in the browser e.q. querying magic methods (interceptor methods) in all methods of the system by executing Smalltalk code

```
#('__call' '__set' '__get' '__invoke' '__isset') includes: self name
```

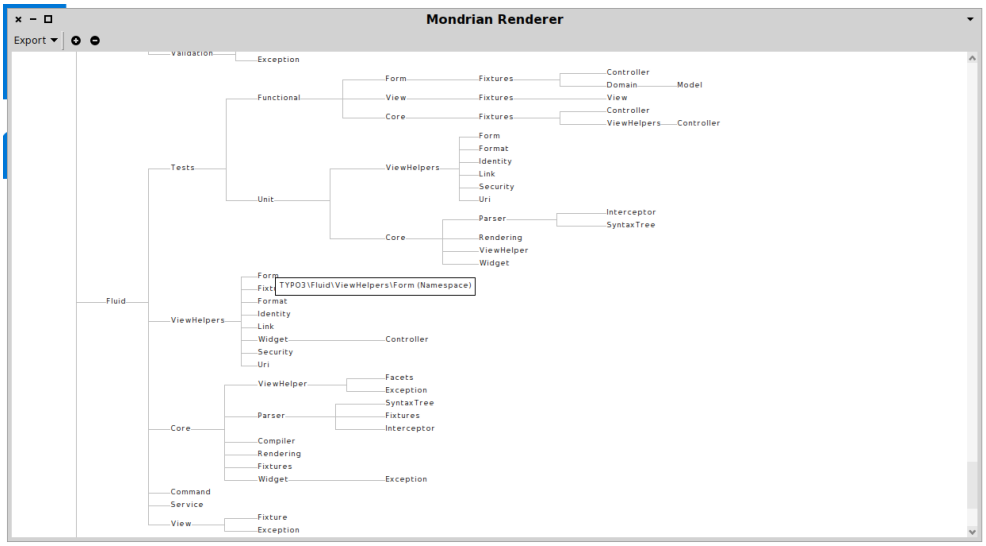

Figure 4.4: Hierarchical and interactive view of namespaces.

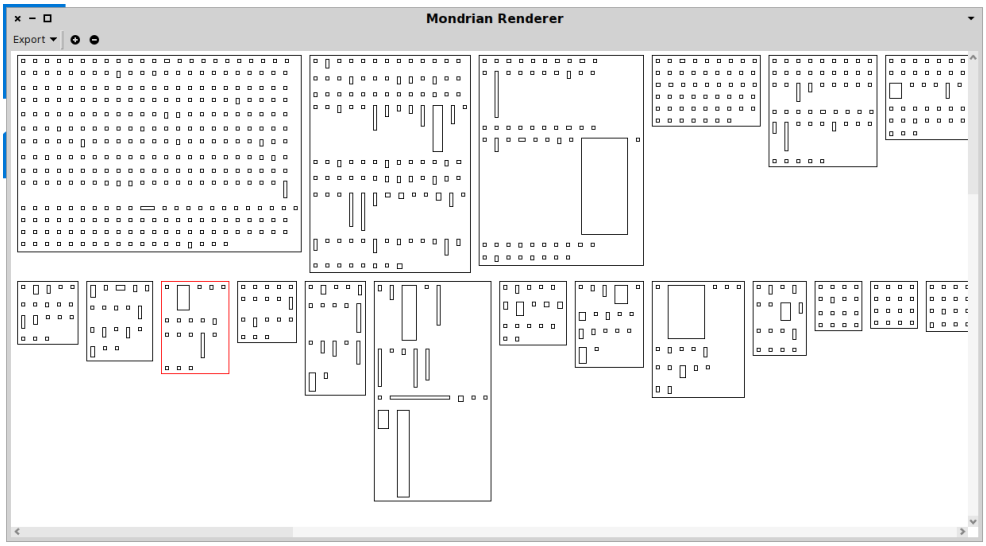

Figure 4.5: Namespace and member visualization view, allows to browse and analyze type in the context of their respective namespace.

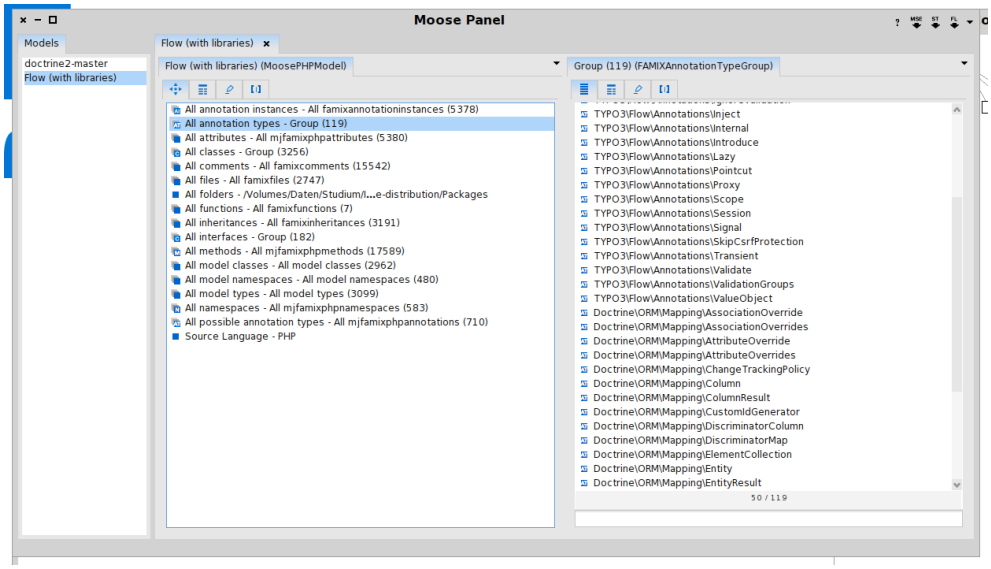

Figure 4.6: Browsing of annotation types which lists all annotation defined in the systems and exposes their corresponding classes and annotated entities.

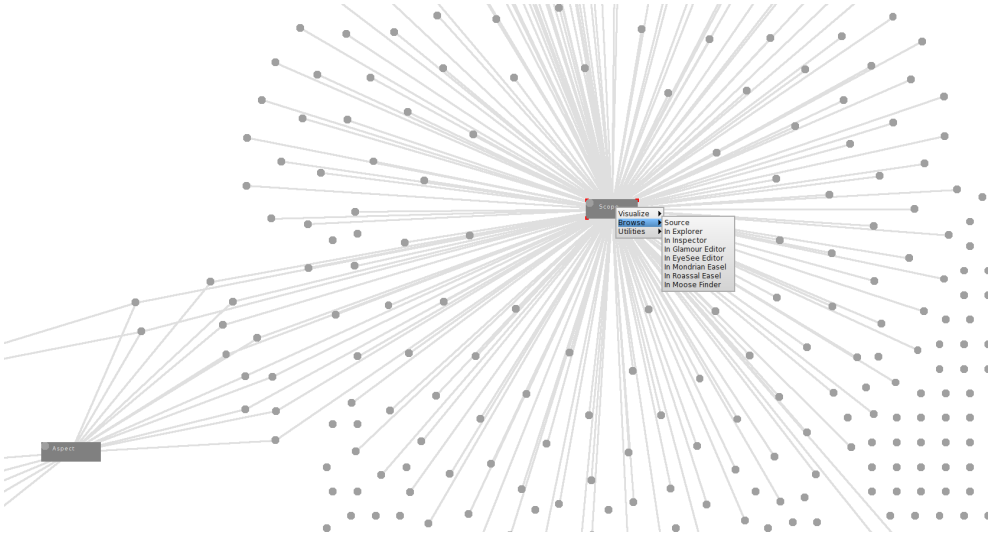

Figure 4.7: Interactive annotation constellation visualization which allows direct browsing of annotations and annotated entities.

5

Conclusions and Lessons Learned

The proposed solution enables static analysis of PHP source codes and especially nested meta data in a reusable, dynamic and extensible way. Even though our implementation suffers from inaccuracies concerning PHP's expression grammar, the *PetitParser* framework provides flexibility in terms of the implemented grammar and enables the parser to be adapted dynamically to syntax changes e.g. caused by updates of the PHP core. Furthermore, the integration into the *Moose* suite allows to perform static analysis tasks that go beyond the focus of this work. The user interface of *Moose* enables an interactive, reusable and flexible analysis approach for parsed sources of any kind without the need for tedious external configuration such as rulesets.

As a critical point it should be mentioned that the *PetitParser* framework is not necessarily the most suitable tool to parse grammars similar to PHP's. The missing possibility to directly transform the PHP reference implementation's parser specification into a *PetitParser* requires further analysis of the grammar's characteristics or tools that generate LR parsers suitable for the usage with *Moose*. As a consequence the achieved results cannot be directly verified and reasoned upon. Our implementation is mostly derived from real-world examples and hence may suffer from inaccuracies, especially concerning the utterly complex expression grammar of PHP. Moreover we left out most of the procedural parts subsequent to the parsing since they do not influence the model required for the targeted analysis.

Nevertheless, the implemented solution covers a variety of entry points for future work and enhancements. PHP's backwards compatibility equals the principle of inheritance and parsers for newer versions can easily be derived from the core parser as soon as the remaining procedural elements are implemented. The work on the parsing and importing components exposed several points worth considering for future work on the topic and treatment of similar work in general.

1. **Research:** To do enough research upfront avoids recurring work and fundamental design flaws. We took up work without any theoretical knowledge of parser theory and only the most basic awareness of grammar classifications. The identification of possible difficulties or misfits between the chosen tools, the approach and the targeted goals are essential for the success of such a project. Once more, a constant reevaluation of assets and requirements in the sense of an agile workflow would have prevented the implementation from being stagnant. In our case an early evaluation of the sources mentioned in Section 2.2.1 would indicate possibly critical parts of the grammar concerning top-down recursive parsing as in *PetitParser* and LALR parsers.

2. **Requirements:** A clearly and precisely specified goal enables us to validate our work. Enabling PHP meta data analysis is an interesting topic but lacks a reference point to validate against. Neither the parsing, nor the established analysis can be properly verified and tested.

3. **Implementation:** "Make it work, make it right, make it fast." may not be the best approach for our implementation. Some refactorings are still pending due to *dirty* implementations which are non-trivial to be rewritten e.g. the model population or the namespace resolution which both could be solved more elegantly using visitors.

4. **Testing:** To test the components in systems as interleaved as the proposed solution is indispensable especially when not being able to rely on a specification. The grammar we implemented is a complex

and fragile construct. During the parsing of bigger systems to identify potential weaknesses we iteratively made subtle changes. Without an according testing suite we are not able to guarantee that allegedly fixes do not break other parts of the implementation. While we used Pharo and *PetitParser*'s internal testing suites future work would probably benefit from a continuous integration server parsing a large codebase on a regular basis.

Appendices

A Installation and Quick Start

The installation of the importer also installs the parsing components. The easiest way to install is to load the components into a Moose image downloadable on from their website[1]. The code in Listing 12 installs the importer and the parser. To install the parser as a standalone component, consider the code of Listing 13.

```
Gofer new
  url: 'http://smalltalkhub.com/mc/FabrizioPerin/MooseEE/main';
  package: #ConfigurationOfMooseJEEPHP;
  load.
(Smalltalk at: #ConfigurationOfMooseJEEPHP) perform: #loadWithoutPetit
```

Listing 12: Installing the importer and the parser (requires *Moose* to be installed).

```
Gofer new
  url: 'http://smalltalkhub.com/mc/Moose/PetitPHPParser/main';
  package: #ConfigurationOfPetitPHPParser;
  load.
(Smalltalk at: #ConfigurationOfPetitPHPParser) perform: #loadDefault
```

Listing 13: Load the PHPParser

After the installation, opening the `MoosePanel` can be done directly using the dedicated world menu *World →Moose →MoosePanel* or by simply evaluating `MoosePanel open`. The interface is mostly self explanatory and provides context menus for all its elements. To import PHP source code, the menu in the upper right corner of the panel includes a command labeled “Import PHP sources from the file system.” which opens a file system browser which allows you to select a directory including your sources[2] Below one can find an entry named “Import PHP sources from the file system and debug on error.” which allows you to inspect the system state after the importing process if there were errors. Figure 1 presents the panel and the menu containing the dedicated import commands.

[1] http://www.moosetechnology.org/download

[2] Potential parsing errors will be alerted after the importing.

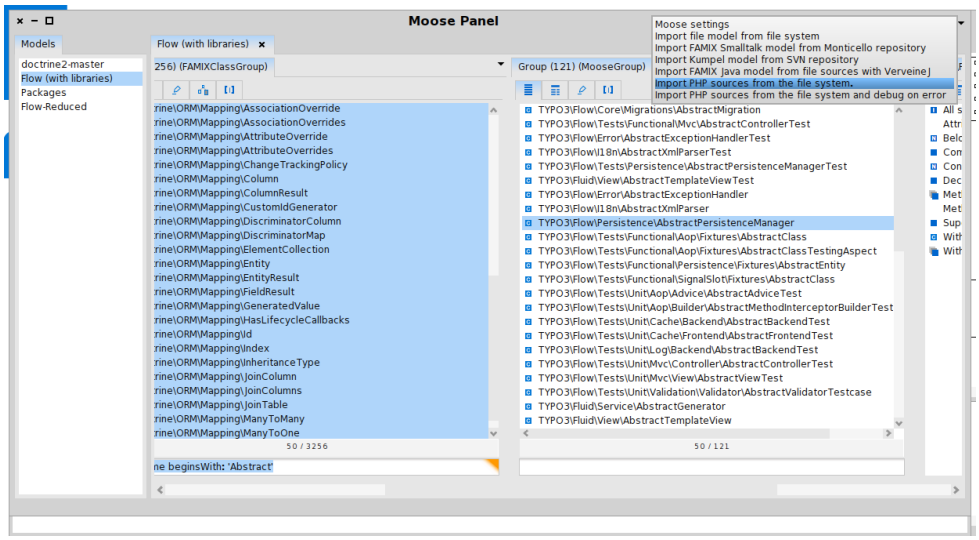

Figure 1: The moose panel and its import commands listed in the menu in the upper right corner.

B Doctrine Annotation Grammar

```
Annotations   ← Annotation ('*'* Annotation?)*
Annotation    ← '@' AnnotationName ('(' Values? ')')?
AnnotationName ← QualifiedName / SimpleName
QualifiedName  ← NameSpacePart '\\' (NameSpacePart '\\')* SimpleName
NameSpacePart  ← identifier / null / false / true
SimpleName     ← identifier / null / false / true
Values         ← Array / Value (',' Value)*
Value          ← PlainValue / FieldAssignment
Constant       ← integer / string / float / boolean
identifier     ← T_STRING
PlainValue     ← float / integer / string / boolean / array / Annotation
FieldAssignment ← FieldName '=' PlainValue
FieldName      ← identifier
Array          ← '{' ArrayEntry (',' ArrayEntry)* ','? '}'
ArrayEntry     ← Value / KeyValuePair
KeyValuePair   ← Key ('=' / ':') PlainValue
Key            ← integer / string
```

Listing 14: Doctrine Annotation grammar in terms of PEG

List of Figures

List of Tables

Bibliography

[Der69] Franklin Lewis Deremer. PRACTICAL TRANSLATORS FOR LR(K) LANGUAGES. Technical report, 1969. URL: http://publications.csail.mit.edu/lcs/pubs/ps/MIT-LCS-TR-65.ps.

[DKSJ12] Marc A. De Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static Analysis and Compiler Design for Idempotent Processing. *SIGPLAN Not.*, 47(6):475–486, June 2012. URL: http://doi.acm.org/10.1145/2345156.2254120, doi:10.1145/2345156.2254120.

[DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000. URL: http://scg.unibe.ch/archive/papers/Duca00bMooseCoset.pdf.

[DLT01] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. The Moose reengineering environment. *Smalltalk Chronicles*, August 2001. URL: http://scg.unibe.ch/archive/papers/Duca01bMoose.pdfhttp://www.smalltalkchronicles.net/edition3-2/Pages/moose.htm.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, October 2001. URL: http://doi.acm.org/10.1145/383845.383853, doi:10.1145/383845.383853.

[For02] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, volume 37/9, pages 36–47, New York, NY, USA, 2002. ACM. URL: http://pdos.csail.mit.edu/~baford/packrat/icfp02/packrat-icfp02.pdf, doi:10.1145/583852.581483.

[For04] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/964001.964011, doi:10.1145/964001.964011.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[HKV13] Mark Hills, Paul Klint, and Jurgen Vinju. An Empirical Study of PHP Feature Usage: A Static Analysis Perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 325–335, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2483760.2483786, doi:10.1145/2483760.2483786.

[Hut92] Graham Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992.

[Joh75] S.C. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

[Mun12] Alex Munroe. PHP: a fractal of bad design. http://me.veekun.com/blog/2012/04/09/php-a-fractal-of-bad-design/, April 2012.

[RDGN10] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010. URL: http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf.

[RS70] D.J. Rosenkrantz and R.E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17(3):226 – 256, 1970. URL: http://www.sciencedirect.com/science/article/pii/S0019995870904468, doi:http://dx.doi.org/10.1016/S0019-9958(70)90446-8.

[Rü13] Michael Rüfenacht. Enabling software analysis using petitparser and moose, August 2013.

[Sev12] Charles Severance. Inventing PHP: Rasmus Lerdorf. *Computer*, 45(11):6–7, 2012. doi:10.1109/MC.2012.379.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001. URL: http://scg.unibe.ch/archive/phd/tichelaar-phd.pdf.

[Vis97] Eelco Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. URL: http://www.cs.uu.nl/people/visser/ftp/P9707.ps.gz.

[ZPY+12] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. The HipHop Compiler for PHP. *SIGPLAN Not.*, 47(10):575–586, October 2012. URL: http://doi.acm.org/10.1145/2398857.2384658, doi:10.1145/2398857.2384658.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname: .....

Matrikelnummer: .....

Studiengang: .....

Bachelor ☐ Master ☐ Dissertation ☐

Titel der Arbeit: .....

.....

.....

LeiterIn der Arbeit: .....

.....

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

.....

Ort/Datum

.....

Unterschrift