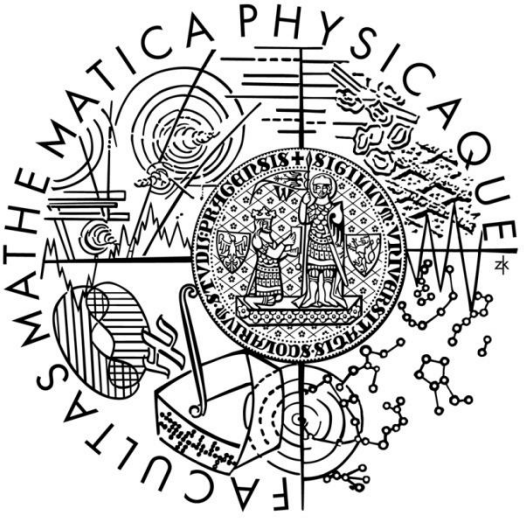
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jakub Míšek
IntelliSense implementation of a dynamic language
Katedra softwarového inženýrství
Vedoucí diplomové práce: *RNDr. Filip Zavoral, Ph.D.*
Studijní program: *Informatika, Softwarové systémy*

Děkuji Filipu Zavoralovi za jeho odbornou podporu a též za jeho trpělivost při vedení této práce.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

Jakub Míšek

V Praze dne

1 Contents

Název práce: Implementace *IntelliSense pro dynamický jazyk*
Autor: *Jakub Míšek*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *RNDr. Filip Zavoral, Ph.D.*
e-mail vedoucího: *zavoral@ksi.mff.cuni.cz*

Abstrakt: *Vývojáři během samotného programování často opakují stejná slova a stále opisují běžné výrazy. Dále také prohledávají dokumentace a zdrojové kódy, například kvůli znění deklarací metod a jiných symbolů. Z tohoto pohledu je velmi užitečná asistence ze strany vývojového prostředí. Konkrétně metody IntelliSense značně usnadňují práci tím, že shromažďují informace o používaných symbolech a programátorovi je automaticky nabízejí. Během práce s dynamickými jazyky je ale tato pomoc velmi problematická, jelikož sémantika jednotlivých symbolů není definitivně známa, dokud program není spuštěn.*

Součásti projektu Phalanger tak byli implementovány metody pro syntaktickou, ale také sémantickou analýzu zdrojového kódu, zaměřené hlavně na dynamický jazyk PHP. Je tak možné sestavit seznam slov, které je možné zapsat na určité místo ve zdrojovém kódu; jako jsou klíčová slova jazyka, deklarované symboly či parametry funkce, včetně informací o nich.

Klíčová slova: *IntelliSense, PHP, dynamické jazyky, Phalanger*

Title: *IntelliSense implementation of a dynamic language*
Author: *Jakub Míšek*
Department: *Department of Software Engineering*
Supervisor: *RNDr. Filip Zavoral, Ph.D.*
Supervisor's e-mail address: *zavoral@ksi.mff.cuni.cz*

Abstract: *In the context of computer programming, the importance of computer assistance is being understood by many developer communities. Developers are e.g. using the same well known expressions or searching method signatures in library documentations. Code sense or IntelliSense methods make most of these actions unnecessary because they serve the available useful information directly to the programmer in a completely automated way. Recently, with the increased focus of the industry on dynamic languages a problem emerges - the complete knowledge on the source code is postponed until the runtime, since there may be ambiguous semantics in the code fragment.*

As a part of the Phalanger project the methods for syntax and semantic analysis of the dynamic code were designed, especially targeted for the PHP programming language. These methods produce a list of valid possibilities which can be then used on a specified position in the source code; such as declarations, variables and function parameters. This collected information can be also used to a fine-grained syntax highlighting.

Keywords: *IntelliSense, PHP, dynamic language, Phalanger*

2 Introduction

Many programmers still use simple text editors to write their programs. They are either used to these editors or they don't have any other, better available development environment for their programming language. It is then hard to avoid syntax errors or typing errors. Furthermore keeping the names of all classes, class members, namespaces, API functions in mind and remembering the names of variables visible in all current scopes is almost impossible. They must browse the source code and all the documentation manually to figure out function parameters and function return value types. They also have to look for comments in the source code and some remarks in SDK and API documentations to check additional information about used functions, classes and other objects.

This is not a major concern when writing small applications developed by small teams of programmers or maybe even by one single programmer. On the other hand large development teams typically produce huge amounts of various software modules with a lot of large documentation files.

Here is a place for an intelligent module used by the source code editor. Using the source code it would process every documentation file and every source code file. It would then suggest the useful information to the programmer in the time they need it.

Additionally, more helpful features can be implemented. For example an intelligent source code editor that understands the programming language can suggest the possible symbols that can be typed. It is also able to replace the typing errors automatically with the correct expression. When the available information is ambiguous it is able to suggest a list of possible options.

Another practical issue is the possibility to navigate through the project's source code files. An intelligent editor knows the structure of the source code. It allows the programmer to navigate to a specific declaration. It can also display declarations within the current file, allowing the programmer to quickly jump wherever they need. It is obvious that working in this environment has to make programming faster and more reliable.

As a part of the Phalanger project [18] we have implemented language integration of PHP dynamic language into the Microsoft Visual Studio environment. This implementation supports most of the common IntelliSense features. This environment already contains an editor with a very good language integration framework, so we designed the *language integration* for PHP and dynamic languages in general.

Since PHP is a dynamic language, there are no static declarations. There can be different variables, classes or class members every time the same source code runs. No declaration has a definite

meaning until the runtime and until the declaration is used. However IntelliSense works with type information which in dynamic languages is missing until runtime. Another issue with some of the dynamic languages is the dynamicity of the source file inclusion statement. The compiler doesn't know which source file will be inserted in the place of the inclusion statement. These are issues for the language integration – all successive alternatives on every code fragment have to be populated.

The main contribution of this thesis is a description of dynamic language source code analysis and information gathering in a well-arranged way. Unlike the static languages, the analysis of dynamic languages has to estimate the types and possible values from the code fragment semantics. These methods also take into account non-static and non-typed implicit and explicit declarations, so that the resulting suggestions cover most of the developer's common tasks.

3 Language integration

Almost every bigger development environment (integrated development environment, IDE) and even some simple text editors have the ability of custom language integrations. That means that in a form of plug-ins it is possible to add a new programming language support into the environment. This support mostly enables a new project type for that programming language and then the ability to automatically compile, run and debug the applications inside the environment using its common controls. Among other things the integration may implement text editing features based on the specific language syntax, like colorizing specified words in the source code by their meaning. The popular feature of integrations is also the auto-completion known as IntelliSense.

In the presented work the language integration for Microsoft Visual Studio 2008 is designed and described methods are implemented. The work is targeted mainly to the IntelliSense support implementation of the PHP dynamic language, which can be compiled by the Phalanger PHP compiler.

3.1 IntelliSense

IntelliSense is an implementation of auto-completion methods by Microsoft. Since 1996 it is used in their integrated development environment - Visual Studio. The symbol names the programmer is typing are completed. In addition to this, IntelliSense gathers documentation for variable names, functions, types and methods, using some kinds of source code analysis and reflection. For the programmer it is an easy way to access specific function description, its parameters list and other information from the documentation. It speeds up software development because the amount of names needed to be memorized is reduced as well as keyboard presses. The reference documentation is also used for gathering information about specific symbols like types and functions, so the programmer can see all available text information about the symbol he is just typing in an interactive way. When he is typing a symbol, the list of auto-completion hints are displayed in a form of a list box under the cursor together with the text information

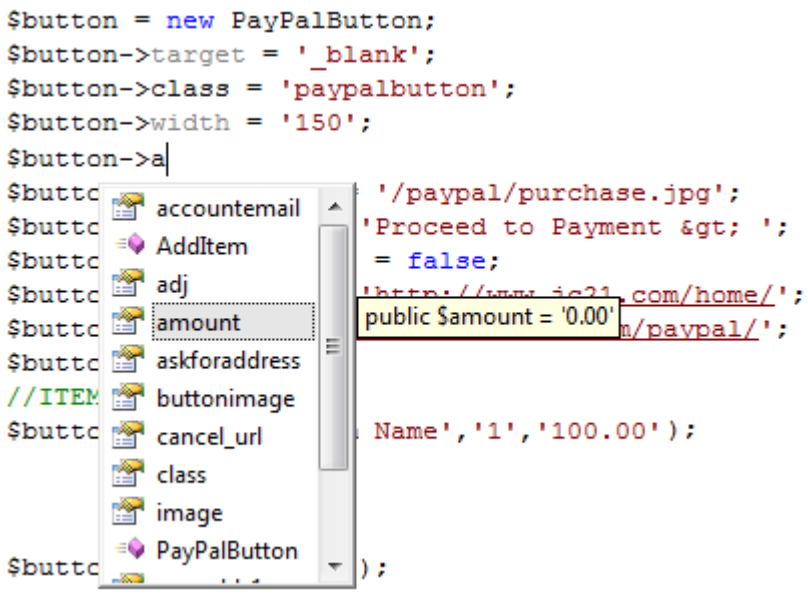


Figure 1 IntelliSense example, suggesting an object members.

about every item. Figure 1 shows an example of these suggestions. Then it suffices to select the desired option and IntelliSense completes the word automatically. Also when the programmer hovers with the mouse cursor over the symbol the information about that symbol simply pops up in the form of a tooltip.

The methods of IntelliSense automatically generate an in-memory database of symbols referenced by or defined in the application that is being edited. This database contains symbols like types, variable names, functions, namespaces and useful information about them.

The implementations work differently for specified languages. Mostly the marker characters are detected, such as dots, arrows or other separator characters or sequences of characters which the language syntax defines. When one of these markers are typed and the marker follows an entity having some accessible members, IntelliSense uses the type information about the entity and offers its members as suggestions in a popup list box. Then the user can select and accept any suggestion or continue typing the symbol. The more characters of the symbol user types the less suggestions are offered because they are filtered by the word fragment typed so far.

The suggestions aren’t available only after known marker characters. IntelliSense can detect typing of any sequence of characters; they don't have to follow the marker. Possible symbol full names are offered too using the current code context (local variables, functions, keywords etc.). The programmer can save yet more keyboard presses when an appropriate naming convention is used. Moreover IntelliSense fixes the size of characters and other typing errors of the word typed so far by changing the word to the chosen suggestion.

3.2 Integration architecture

The mechanism of the code sense, called language integration, has to respond on several events. The user is supposed to get help while he is typing or when he moves the mouse cursor over an unknown symbol. These are the basic events which should be followed by a corresponding response. There are other events; e.g. when the source code editor needs to redraw the text it asks the language integration for the color of the words. All the events are raised by the source code editor and are handled by the language integration. This mechanism is depicted on Figure 2.

The source code editor requests the *language integration* to parse the source code every time it was changed and as a result the *language integration* returns a list of tokens. The tokens have some specified properties. The main properties are the position in the source code, token identifier and the color of the token. Other properties are flags used by the editor to determine possible actions which are supported by this token. These flags enable the language integration to enable or disable future actions on each token.

After parsing the editor can send specific requests. The most important are requests for the list of available declarations (which can be typed at the cursor position) and requests for text information about the token under the mouse cursor, usually displayed as a tooltip. The first one is requested every time the user inserts a character, which is a part of the token with a flag implying that this token enables a suggesting (list of possible declarations). The second one is called when the user moves the mouse cursor over a token with a flag specifying that this token is able to offer textual information.

In this place the intelligence of the language integration comes into place. The language integration has to manage its own in-memory database of symbols from every source file in the project. This database is used to find a declaration or a list of declarations corresponding to the specific token, representing a finished or an unfinished symbol. Therefore the language integration stores the source code in an abstract syntax *tree* (IntelliSense tree) which is built for all files placed in or referenced by the application being edited.

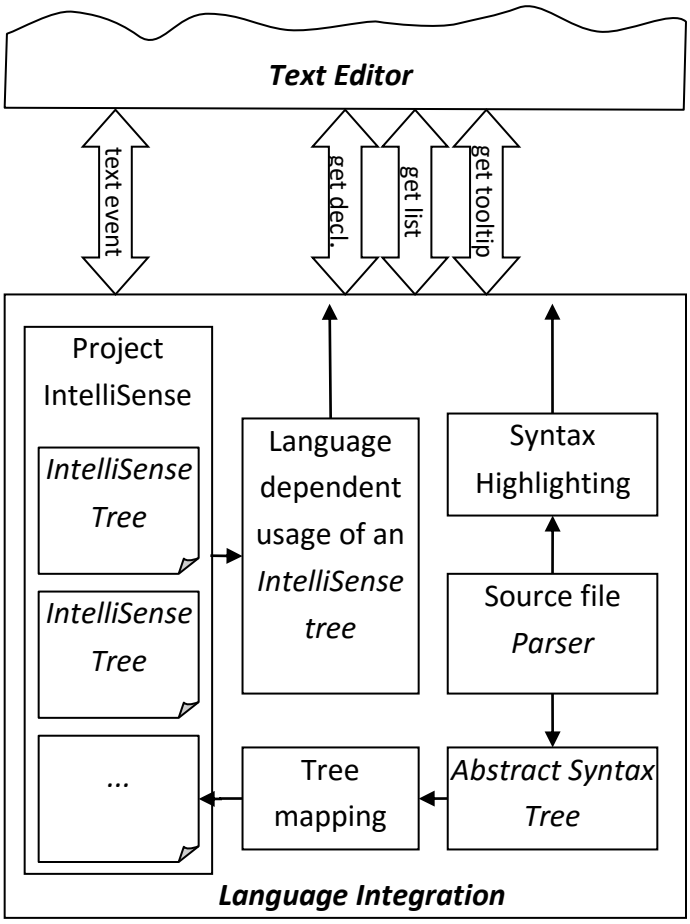


Figure 2 Language integration architecture.

The tree contains all declarations and definitions organized in the same scope structure as they are in the source code. The tree is optimized for searching for specific declarations accessible within the scope in the source code. It is important to note that while the trees of the source code files are being built, the information about every symbol is gathered subsequently using semantic analysis. Every symbol usage must be processed and resulting information is added into the appropriate symbol declaration (the creation of the trees is described in section 6).

The IntelliSense tree used by the language integration is designed to be language independent. It can be created from language specific syntax trees or other sources by mapping them into the IntelliSense tree. Then the language integration works with these trees so it doesn't have to solve any differences between original languages. It accesses the trees in a same unified way. The usage

of these trees depends on the source code where the IntelliSense is used, so the specific usages for specific language syntaxes have to be defined there.

The IntelliSense tree of every source file is cached in the memory and its last version is available all the time. This is important because after the user has typed something into the source code, the source code is mostly no longer syntax-valid and therefore a new updated IntelliSense tree can't be created directly. Then the cached one has to be used and only the node positions should be modified to reflect the new token positions.

3.3 Language Integration for Microsoft Visual Studio implementation

Described IntelliSense methods are implemented as a part of the language integration for Microsoft Visual Studio 2008. The main module of this language integration is the Language Service, which represents specific language, PHP language in this case. Besides the IntelliSense there are other components needed to have the Language Service working, such as a source code parsing component. Therefore the *Phalanger Core* [3][13] is used, especially its PHP parser module which is able to process the PHP source code, parses the code and generates the abstract syntax tree from that. In general used language integration comes from the IronPython [2] language integration project, which is a good example of designing a language integration solution for the Visual Studio environment.

3.3.1 Visual Studio Language Service architecture

The language integration for Microsoft Visual Studio uses .Net Visual Studio SDK [1][12]. The integration consists of COM-visible objects. They are registered for specified file extension so they can be instantiated by the Visual Studio environment when a file with the name containing this extension is opened in the Visual Studio text editor. These objects implement the logic of the language; the text editor uses them to obtain the language-specific information about the source code.

In this work, the main language integration object, describing the PHP language, is named *PhpLanguage* and it is placed in *PHP.VisualStudio.PhalangerLanguageService* namespace. The *PhpLanguage* is inherited from *Microsoft.VisualStudio.Package.LanguageService* type [1], which is the base class for a language service that supplies language features including syntax highlighting, brace matching, code collapsing, auto-completion, IntelliSense support, and code snippet expansion.

Figure 3 describes the simple language service architecture. The *LanguageService* provides methods that Visual Studio uses to obtain information about the source code and to request for IntelliSense objects. Every time the source code changes or the cursor moves the *ParseSource* method is called along with its call reason and the *AuthoringSink* object reference. The *AuthoringSink* is a container

used for collecting various source code information. The code is processed and the *AuthoringScope* object instance, which implements the IntelliSense functionalities, is returned. Within the *ParseSource* method, various information about the source code text, such as syntax errors, mathing braces position or code blocks positions, can be gathered and stored within the *AuthoringSink* object. The Visual Studio then uses this object to enable appropriate text-oriented features like code collapsing or errors underlining. LanguageService also provides methods that enable Visual Studio instantiating objects used for parsing the source code (*IScanner*) and for handling other source code text events (*Source*). The last one handles changes in the source code text other text editor events.

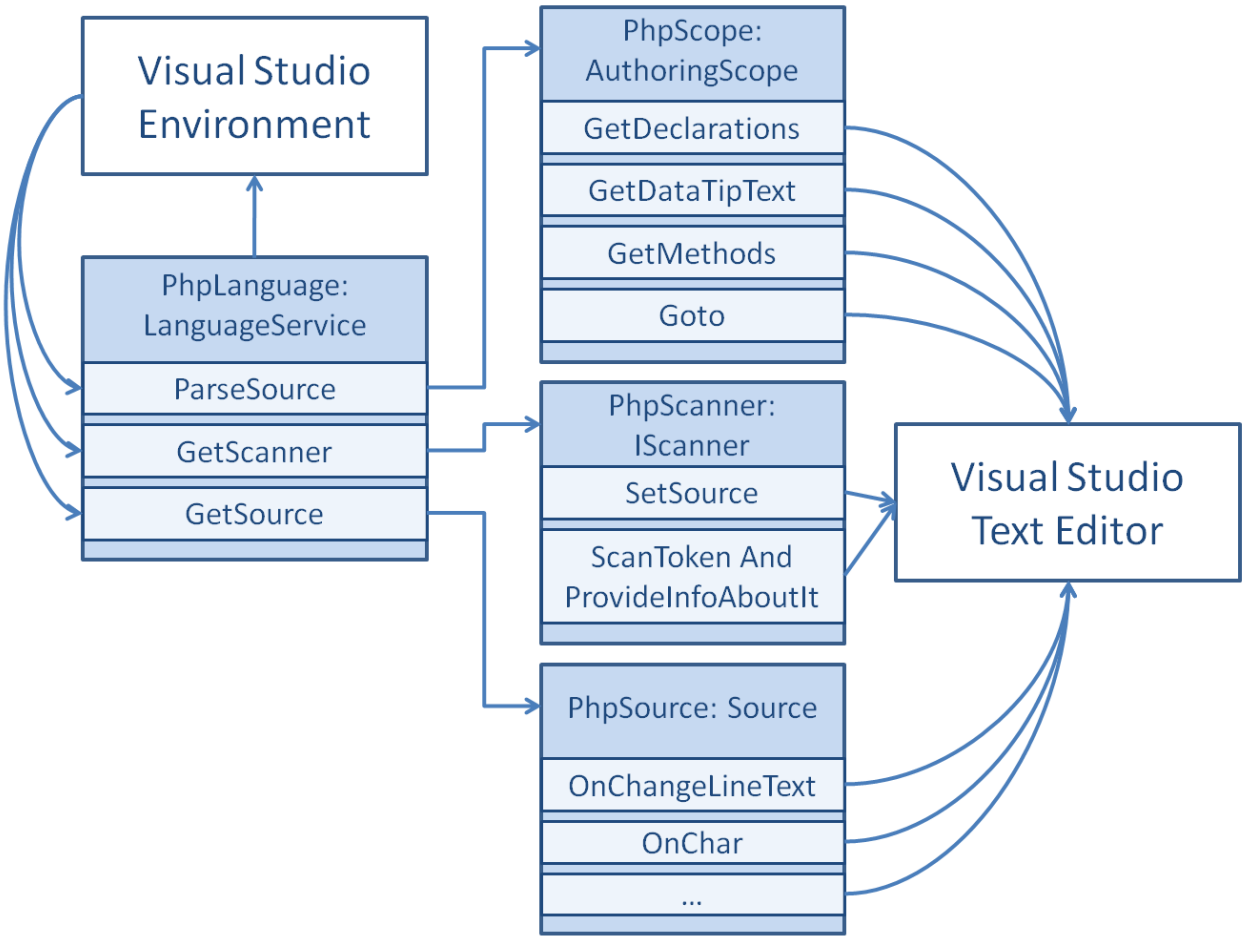


Figure 3 Visual Studio Language Service architecture

As a response to the *LanguageService.ParseSource* method call, the *AuthoringScope* object instance is returned. It offers methods for IntelliSense support, such as getting a list of declarations (auto-completing), text descriptions about the symbols and getting a position of the specific symbol declaration. Also the AuthoringScope defines a method for obtaining a list of possible function call parameters list.

The implementation described in this work inherited from the *AuthoringScope* type is named *PhpScope* and it uses the IntelliSense tree to manage the in-memory database of symbols and to manage the structure of the source code. Then it is used for implementing the IntelliSense support in the VisualStudio environment (for the tree creation see section 6 and for its usage see section 7).

3.3.2 Visual Studio Language Service features

In an addition to IntelliSense Visual Studio language service defines other text editing features which could improve the development process. Using the code collapsing the programmer can easily hide parts of the source code which are not important for him at the moment. Likewise the text editor automatically highlights the source code syntax, highlights the matching braces around the cursor and underlines syntax errors. So the programmer can see more information about the source code and it makes the development faster.

3.3.2.1 Syntax highlighting

Syntax highlighting is an ability of the text editor to colorize the words by its meaning. It is used mainly for differentiating words by its syntactic meaning, mostly language specific keywords, function names, type names and string or other literals.

Through the *IScanner* object instantiated by *LanguageService.GetScanner* method call, the Visual Studio obtains the tokens. The implementations of IScanner object parses the given source code text and creates a list of tokens for every line of the text. Tokens then describe symbols on the line, including its color index. These indexes point into the array of color specifications which are defined by the language service implementation and describe foreground text color, background text color and other font properties. Syntax highlighting feature is very useful for better source code overlook, it is easier for the programmer to navigate along the code and see what he's just looking for.

3.3.2.2 Brace matching

When the cursor is placed near a pair symbol, this symbol can be highlighted together with its corresponding pair symbol, like pairs of brackets or parentheses. This feature is called brace matching and it is available in most modern source code editors. Especially in complex expressions it is very helpful to have an overview about braces and other pair symbols. Also the programmer can see immediately if some pair symbol is missing.

Visual Studio gets positions of pair symbols from the *AuthoringSink* object, which is filled with this information during parsing the source code. Then the pair symbols near the cursor are automatically highlighted.

3.3.2.3 Error underlining

Syntactic or other errors are automatically underlined by a red waved line. So the programmer can note them immediately without a need of running the compilation process. The error information,

their positions and text descriptions, are inserted into the *AuthoringSink* object during the *LanguageService.ParseSource* method call. Visual Studio then automatically displays them in a special errors list window and underlines the errors in the source code.

3.3.2.4 Code collapsing

Code collapsing, or hidden blocks, is a text editor feature which enable hiding of specific blocks of source code text using the small plus symbol on the left side of the text. By clicking onto this symbol or pressing special keyboard shortcut inside the code block, the block is collapsed into a small box. Mostly it is used for hiding function bodies or class declarations. The programmer can then take care about the rest of the code only, on which he's just working on. It improves the code overlook; the programmer can concentrate on the important parts of the code.

The information about the collapsible block positions is inserted into the *AuthoringSink* object during the *LanguageService.ParseSource* method call. Visual Studio text editor uses these positions automatically to enable the code collapsing functionality of specified code blocks.

3.3.2.5 Navigation bar

Global declaration names within the whole current source code file can be listed in an extra combo box. It enables the programmer fast navigation through the file. Mostly there are listed class declarations and function declarations, but navigation bar is being used even for variables and constants. In addition if selected declaration has any member declarations (like class and its methods and properties) they are displayed in the second combo box called *members list*. The programmer can see all the declarations in the file and he can easily navigate to them.

Visual Studio environment gets the lists of declarations and their members using the special object (*TypeAndMemberDropdownBars)*. The object is instantiated by the *LanguageService* implementation during the *LanguageService.CreateDropDownHelper* method call. The implementation of the object then fills the list of declarations of the given source code when its member method *OnSynchronizeDropdowns* is called. Also it fills the *members list* of the currently selected declaration. This method is called every time the cursor position is changed so Visual Studio needs to update the lists.

4 Source code parser

It is necessary to perform lexical and syntactic analysis before the semantic analysis of the source code. The whole process flow is illustrated on Figure 4. For the syntax highlighting support it is actually sufficient to implement lexicalization only. Also other features work with the tokens only such as brace matching.

More advanced methods require the syntax tree so the syntactic analysis follows. It consists of the token analyzer and the abstract syntax tree builder. After these steps the parser knows all syntax errors so the programmer can be alerted immediately. The syntax tree might be used for code block collapsing and navigation bar support.

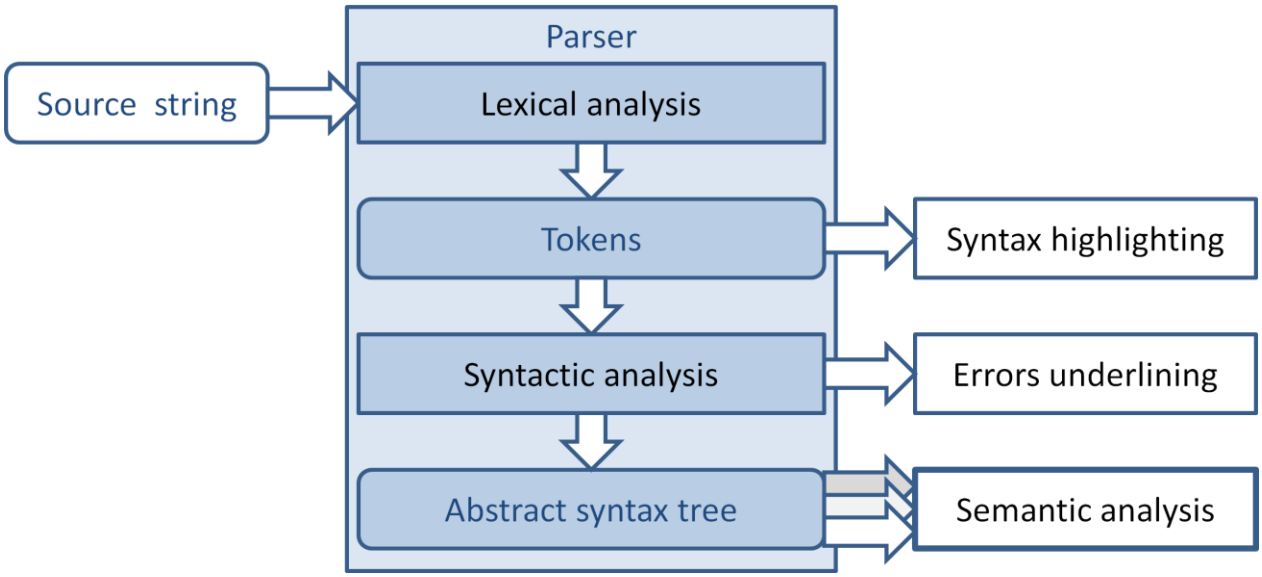


Figure 4 Parser flow

The resulting data such as tokens together with the abstract syntax trees might be used for a semantic analysis. It gathers the symbols and relevant information about them to generate an in-memory database of symbols. This is used for enabling IntelliSense support which generally requires the type information; in case of dynamic languages it has to be estimated based on the semantic analysis. In static languages the type of symbols is known already from the syntax tree.

It is important to note that a common compiler for the specified language can be used, or more precisely its parser module only. In this work it is the Phalanger project which contains the PHP parser written in .NET language. In general it is possible to use the Phalanger with small modifications to parse the PHP code and to generate tokens and an abstract syntax tree (AST).

4.1 Phalanger project

Phalanger is an implementation of a compiler and runtime of PHP dynamic language running under the .NET Framework; therefore it is able to parse, compile and then run the PHP applications. The project is available as an open source. It was started on Charles University in Prague in 2003 and at present it is fully capable to compile and run most of the PHP applications.

As a part of the project it manages many components and various functionalities. There are listed the most important ones:

- **.NET interoperability** - It allows an interoperability of dynamic PHP with static .NET languages so that it is possible to use .NET objects in PHP program. Presented implementation of language integration is primarily designed for source codes compiled by Phalanger therefore it has to support PHP symbols along with the .NET symbols from .NET assemblies. The interoperability allows using the PHP for more various targets such as Microsoft Silverlight or Windows Forms.
- **Lexer/Parser** - Among other things the usability of Phalanger is in its individual modules implementation. Many of them are used to realize the implementation of presented work. As a regular compiler there must be a lexicalization and parser modules. The first one is generated using the modified GPLEX project [14] which is an open source generator for lexical scanners. The second one is built by the GPPG [15]; it is the similar project for generating LALR parsers. These two modules are able to work without the rest of the project; they are used here only with small modifications for generating tokens from the source code string and parsing them to build an abstract syntax tree (AST).
- **Doc comments** - The Phalanger parser was modified to process the user comments too. If they are in a specified format then corresponding AST nodes may contain some extra information extracted from them. The documentary comments are used mostly for explicit description of functions. They contain not only the function text description, but most important is the possibility to specify the function return type and function signature, especially its parameters type. In dynamic languages these information are not a part of the language syntax so this is the alternative way how to specify them. This extension of AST is normally used for generating better documentation but it could be very helpful for implementing IntelliSense which is based on symbol types.
- **Compiler** - Phalanger also provides the compiler which generates the code for Microsoft Intermediate Language (MSIL). Optionally the compiler adds the debug information into the MSIL code, so it enables the support of debugging right in the Visual Studio environment.
- **Runtime** - Since the PHP is a dynamic language, it requires a non-standard runtime environment. Most of simple operations, which are normally resolved within the

compilation process in static languages, have to be solved during the runtime as callback methods. Therefore there are also runtime support modules.

4.2 Lexical analysis

For the further analyses (syntactic, semantic and an others) it is necessary to convert the source code string into basic elements called tokens first. These tokens are used also by the text editors as the smallest text elements which define the color and properties of the corresponding part of the text. Since the text editor needs this information updated all the time, the lexicalization is performed every time the source code is changed; therefore it has to be very fast. The token properties enables us to tell the text editor what functionalities the current language integration supports on the specific token, such as pair symbols highlighting (find pair of parentheses), displaying of quick info texts (tip texts of identifiers) or text completing (IntelliSense, auto-completion).

In general any lexical analyzer for the specified language, or shortly lexer, can be used for converting the source code string into tokens. In principle the text is parsed by the scanner component, which is mostly the simple finite machine defined by the language specification, to get lexemes. Since the finite machine is used we can remember its states on every processed line of the code for further optimizations (4.2.1). Lexemes define a block of text on the specific position in the code; they are then categorized by their meaning and converted into the tokens within the tokenization process. The token category is then used for specifying the color and properties of corresponding text block. Also tokens with no importance like "white-space" may be ignored completely.

The scanning and tokenization processes are generally not separated into single components. They work together within the lexical analyzer and the whole process can be performed in a single pass. In practice these analyzers are generated by tools such as flex or its modification GPLEX used in this project. Tools generate lexers from the set of rules which specify the programming language syntax.

4.2.1 Optimizing lexicalization

The process of lexicalization is performed every time the source code is changed. The changes mostly appear in a single line while the rest of the code stays unchanged. Therefore it is pointless to reanalyze parts of the text not affected by the changes.

Hence the tokens on every line of the source code text are cached along with the scanner state on accordant line end. When the code is changed only changed lines are rescanned. The scanner starts with the state initiated by the state from the previous line end. After the line reanalysis, if the state of the scanner differs from the previous state then the scanner has to process following line too.

Thereby only lines where tokens might be changed are reanalyzed; even large source code texts can be rescanned quickly.

This optimization is easy to implement by the simple modification of the scanner component to process the source code text by single lines. Only its state has to be initiated before the scanning with the state from the previous line end.

4.2.2 Syntax highlighting

Language integrations typically offer one of two types of syntax highlighting ability, if any. They differ primarily in the computing complexity.

- **Simple, syntax based** - Even older text editors offer the functionality of text symbol colorization based on the language syntax. Typically it uses the tokens as a product of the lexical analysis. Based on the correspondent token category every symbol has assigned a specific color and eventually a different font. This color resolving is performed every time the source code is changed for every newly created token.
- **Semantic based** - The syntax based functionality used to be extended - the symbols are highlighted depending on the syntactic and semantic analysis. For these methods tokens don't contain enough information and it is necessary to resolve declaration types and other properties of highlighted symbols. Then using different fonts and colors the local declarations can be distinguished from the global ones or private methods from public ones etc.

4.2.3 Brace matching

Highlighting the pair of parentheses (or any pair symbols) is another functionality which is being implemented using tokens. To find the pair starting from the token under the cursor, simple grammar can be used. The grammar is working with the tokens list. The rules of the grammar depend on the specific language syntax, but most of the language symbols may be ignored so the grammar can be designed to be almost universal. Simple grammar for matching the parentheses is represented on Figure 5, rule "*P*".

```
P := "(", X, ")"

X := (*, P, *) | *

* := [^"(", ")"]
```

Figure 5 Parentheses matching

Note that the grammar has to be defined over the tokens and not over the source code string only. The string contains same symbols with different meaning, like the parentheses as a language symbol and parentheses as a part of the string literal.

If given rule matches the list of tokens where the first symbol in the rule matches the token under the cursor, then the tokens representing the parentheses in the rule may be highlighted.

The grammar describes the formal problem. In practice it is much easier to have a counter, scan the tokens starting on the token under the cursor; increase the counter whenever an open parenthesis appears and decrease it when a close parenthesis appears. All the other tokens will be ignored. The direction of scanning depends on the value of counter; if the open parenthesis is the first scanned token, the counter will have a positive value and scanner will go right, otherwise left. If the counter reaches zero then the procedure ends, the first scanned token and the last scanned token are the pair symbols. The range of scanned lines has to be limited because sometimes the second pair symbol is missing and the whole source code would be scanned. This algorithm (on Figure 6) finds the same symbols like the grammar above, but it is easier to implement.

```
L = {any left pair symbol}, R = {any right pair symbol}
t = {array of tokens}, i = X = {current token index}, d = 0 /*scanning direction*/
l = 0/*nested level*/

if (t[i] in L) then (d = +1), else if (t[i] in R) then (d = -1) /*init direction*/
otherwise return (nothing) /*current token is not a pair symbol*/

l = d /*init nested level*/

while (l != 0) do begin
        i += d /*update position*/
        if (t[i] in L) then (l = l+1), else if (t[i] in R) then (l = l-1) /*update nested level*/
        if (|X - i| > {max processed tokens limit}) then return (nothing) /*no pair in limit*/
end

return (X, i) /*location of outer pair symbols*/
```

Figure 6 Brace matching pseudocode

4.3 Syntactic analysis

In language integration the parsing, or more formally syntactic analysis, is used for a preliminary check of the source code syntax validity and for building the syntax tree for afterwards semantic analyses. In general it is the process of analyzing a sequence of tokens (from the preceding lexical analysis) to determine their grammatical structure.

A parser is being a part of compilers and may be semi-automatically generated by a tool from the specific language grammar. In the presented work the parser component from Phalanger is used. It finds the syntax errors and builds the abstract syntax tree for syntactically valid parts of the source string.

Generally any parser for specific language can be used to determine source string grammatical structure and to build some data structure describing it. In practice there are many tools for generating parsers from a grammar written in Backus-Naur form (4.5). Also some existing components use this notation for implementing very simple language integrations of static languages.

4.3.1 Syntax tree

As a product of parsing the abstract syntax tree, or AST, is created. It is a data structure in a form of the tree representing the syntactic structure of the code. This structure describes the source code in an abstract way; the tree consists of tree nodes and each node corresponds to a construct from the source code. In common syntax trees many details from the source code, typically not needed for the compilation process, are missing, e.g. comments and preprocessor commands.

This source code representation is then used for next analyses. Therefore it is useful to gather as much information as it is available in the source code, including the comments mentioned above. During the semantic analysis the comments might be used and additional information might be added to the syntax tree by subsequent processing.

Note that syntax errors may cause an incomplete syntax tree. They have to be caught within the parsing. That is an issue for language integration because it needs an abstract description of the whole source code, even if it isn't syntactically valid.

4.3.2 Syntax errors

The mechanism of underlining errors in the source code is very useful and allows users to see specific notifications as soon as it is detected by the language integration. There are several typical usages of this:

- **Lexical errors** - It is very easy to implement. It just underlines unknown symbols that are reported within the lexical analysis.
- **Syntax errors** - Still simple detection, syntactic errors are reported by the specific language parser. It is basically a product of the syntactic analysis.
- **Spelling errors** - For the text in comments and string literals there can be used some spell checker component to report spelling errors too.
- **Variable usage warnings** - Not used variables and usage of not initiated variables can be detected by the semantic analysis and reported to the user.
- **Unreachable code** - Parts of the code that might not be traversed during execution can be detected and the user can be warned. The implementation can be realized by building the control flow graph (CFG) [17].

- **Undefined symbol** - It means that usage of a symbol in the source code which has no definition is underlined, such as unknown function call, unknown variable name or not existing class member. It is implemented typically by running compilation process on a background thread; this method works in general and covers all the error underlining implementations mentioned above and many others. The negative side of this is higher CPU usage.

Parsing usually checks for syntax errors, which are simply the tokens not matching any grammar rule (they are not expected in their location); parser produces errors messages, warnings or another notifications. Source code editors are then able to automatically underline them in the source code and to display a list of all the errors. These functionalities are managed through the special object, which is filled with all the notifications in the source code by the parser. It is used then by the editor environment to retrieve the current list of errors.

Since parser has to describe a concrete syntax error to help the user identify it; the language grammar typically contains additional rules to catch frequent errors for a better error description. They are used also for enabling the parser to continue processing the source code. Any other syntax errors are marked as an "unexpected symbol" and may cause parsing abortion.

With the view of generated documentation from the source code comments, the language integration should prevent spelling errors. Therefore even source code comments and string literals might be checked using some spell checker component (e.g. Microsoft Word spell checker). Found spelling errors are then added into the same ErrorSink object used for syntax errors.

4.3.3 Code collapsing

As a part of AST nodes there are locations of corresponding constructs in the source code. They can be used for defining collapsible blocks. Typically the syntax tree is scanned to find the nodes describing class bodies, function bodies and namespaces. These nodes are used then to determine the block location.

The whole mechanism of hiding and showing these blocks is then managed by the source code editor environment. It has to remember the state of blocks, their locations and if they are shown or hidden, and properly render the source code text.

Collapsible blocks are recreated every time the source code is parsed using the current abstract syntax tree. Then previously initiated blocks have to be mapped onto the newly initiated blocks to keep collapsible parts of the code in its previous state. This can be done by simple matching range of lines of currently created blocks onto the last blocks lines, moved to reflect changes which was made in the source code since last parsing (lines added, lines removed).

4.4 Local AST

Sometimes it is needed to determine a syntactic structure of the code which is not syntactically valid. It is important especially for the IntelliSense implementation. To keep the IntelliSense working it is useful to have a syntax tree of an expression which has to be analyzed further for enabling IntelliSense features, such as auto-completing or tip texts. These invalid expressions occur while the source code is just being edited by the user, but at the same time the user is awaiting working IntelliSense, e.g. to auto-complete syntactically invalid expression.

Therefore it is needed to build syntactic structure of an incomplete expression, where the specific language grammar rules do not match yet. For these purposes another and much simpler grammar can be defined (e.g. Figure 7). It enables to create simple syntax tree of a small part of the code in an abstract way. To do that it suffices to use the tokens from the lexical analysis and to have a trivial grammar describing most used expressions. The grammar rules might be defined from the right to the left. They are than matched typically starting with the token under the cursor continuing to its left side. The longest matching sequence of tokens is then found.

```
A := A->X | B::X | X

X := X\[.*\] | X\(.*\) | Y

X := \{.*\} | $X

Y := Identifier

B := B::Y | Y
```

Figure 7 Simple expression grammar, starting from A.

This light-weight grammar may ignore many language elements, such as array indexes or unexpected tokens. The point is to ignore syntax errors and to get a simple and mostly valid abstract syntax tree, which describes the expression structure and contains just its important parts. The grammar rules have to define variable usages, function calls, member chains and indirect usages. It is basically enough to cover most used expressions that IntelliSense is able to work with.

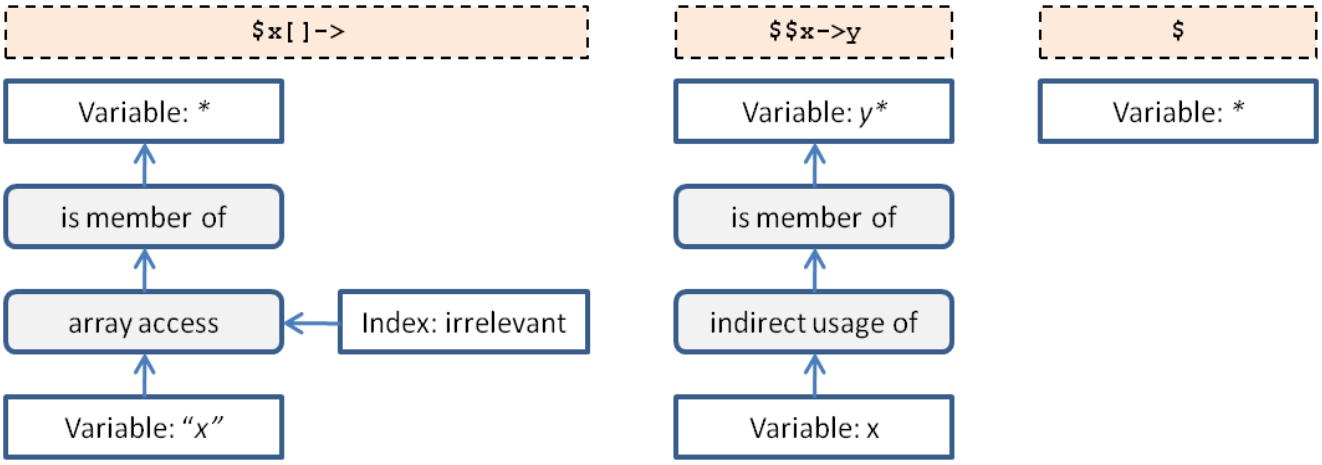


Figure 8 Invalid expressions and corresponding syntax trees.

Some of required expressions and corresponding syntax trees are depicted on Figure 8. Despite the fact that these expressions are not syntax valid, the syntactical structure is needed. The simpler and much more tolerant grammar can be used to determine any usable syntax tree.

4.5 Backus-Naur form

Backus-Naur form, shortly BNF [14] [15] [19], is a meta-language used to formally define context-free grammars. Formal languages might be described in this way mostly by defining their syntax with two sets of rules - lexical rules and syntactic rules. In the context of source codes analyzers BNF is widely used as a description of programming language syntax.

Many source code analyzers used in IntelliSense implementations are able to work with this notation almost in general (eclipse). Just note that in the presented work the complete parser from Phalanger is used. It is faster, more sophisticated and adapted to the PHP language. Of course, Phalanger contains a formal definition of the language too, in the form of BNF, but there are other components integrated to its parser, e.g. PHP syntax tree builder, all built for the .NET framework environment.

5 Code analysis

The main part of the *language integration* mechanism is the ability to manage information about the source code, typically gathered using semantic analysis of the syntax trees. It has to provide methods for getting the source code structure, defined symbols (declarations) and related information about them.

This symbol information should contain common entities of object oriented languages and data resulting from the semantic analysis too:

- **Symbol name** - Symbol text identifier as it is used in the source code (declared identifier name, e.g. variable name, type name, function name etc.). In dynamic languages it might be a result of an expression evaluation. The name is needed for identifying the symbol and is defined by the user.
- **Symbol type** - There may be many different symbols, each describing a different language entity. Fortunately there is some well known set of entity types in programming languages, such as *function*, *variable*, *class*, *namespace* etc. These types are important to identify symbols more closely. So every symbol should contain its type identifier.
- **Location** - The symbol position is needed; it is locations in the source code to enable us easily navigate to it, but also it is a position in the context of other symbols. So there have to be a line number and column number range, and all the symbols have to be stored in a hierarchical structure similar to the original syntax tree.
- **Variable type** - IntelliSense is practically based on the knowledge of the variables type. Hence the symbols describing variables have to contain their type if it is known and if it could be resolved. Moreover in dynamic languages one variable can carry different type instances in different parts of the code and in different program executions. Also the variable can be initiated dynamically, and its type is known only at runtime. Therefore the symbol should contain some list of possible types resolved within the semantic analysis. But in general it cannot be resolved definitely in dynamic languages; even the single program execution won't help in some cases.
- **Possible values** - In dynamic languages sometimes the value of a variable has to be known to determine type of another variable. Hence it has to be possible to try to evaluate the variable and remember the result. This can be used for next processes, such as analyzing of indirect variable usages etc.
- **Description** - The textual description helps the user to identify the symbol. Also it can provide related information about it. The symbol info may be obtained from user documentary comments, online documentation or other sources. Also related texts can be attached as well as the symbol file location.

- **Function signature** - In case of a function declaration, IntelliSense requires its parameters. So the symbol describing a function should provide a list of parameters. They can be described in the same way as the regular symbol is - everyone should contain name, type, possible values and text description.
- **Object members** - In object oriented languages, single symbols (especially type declarations and its instances) contain object members (properties and methods). In case of variables their members implies from their type. In dynamic languages the type need not to be known, but possible members of a variable can be gathered using the semantic analysis. In general any symbol can have members (e.g. class members, function return value members, namespace members like inner declarations etc.).
- **Extensibility** - In the future other data as a result of analyses might be added. The information management (in-memory database of symbols) must have the ability to be extended and must be allowed to store the custom data. Particular symbol types may require their own memory space for additional related information, such as base type of the class declaration or return expressions of the function, used for next analyses.

Therefore, in order to make this mechanism usable in general, the structure of a typical source code should be stored in a tree. Actually the tree structure produced by parsers and used by compilers called *abstract syntax tree* (4.3.1, AST) is very flexible and satisfies most of the specified requirements. With some modifications it can be used for gathering information about the source code to be usable for implementing IntelliSense features.

For the purposes of IntelliSense we've designed the universal *IntelliSense tree* which is derived from the notion of *AST*. Classic AST doesn't describe all the information needed by the IntelliSense, like single node descriptions, suggested types or other data used for declaration analysis. IntelliSense trees work then like in-memory databases of symbols, defined by the user in the source code as well as defined by the language specifications. Database of native language symbols is stored in the same way as the user-defined symbols.

Having the specified ASTs as a product of the source code parsing, it is possible to define mappings from these types of AST to the *IntelliSense tree*. All IntelliSense operations use the *IntelliSense tree* to obtain the information about the source code. The tree gathers the information from the original AST and other sources, such as other ASTs, documentation files or user comments. Thanks to this method it is possible to use more source files written in different languages within the one *language integration* mechanism. When a language is integrated with other languages, like PHP is integrated with its extensions or C# with .Net assemblies and XML documentation, the *language integration* then does not see any difference – any source file can be mapped to this universal *IntelliSense tree* by providing appropriate mapping (6). Once the tree is built, additional information is added to the tree by subsequent processing (semantic analysis) of the tree or its source data.

5.1 IntelliSense tree architecture

The tree described in this section is a data structure for managing information about the symbols defined in the source file. In addition to that this in-memory database of symbols stores the results of semantic analysis. It offers processed data in a unified way, adapted for usage in IntelliSense implementations.

The structure of the tree has to reflect the structure of the source code. The nodes represent single language entities, such as function bodies, class scopes and general code scopes, which are defined in the source code. They are not initialized until the information about them is requested. Thus the tree is built lazily starting from the root scope, to save computing time. This laziness saves time and memory, only the needed parts of the tree have to been built, the other parts are not touched.

There are two base types of nodes in the tree, describing two main languages entities:

- **Scope node** - The first one is the *scope node*, also used as a root of the tree. It works like a container for other *scope nodes* and *declaration nodes*. Typically it will correspond to some code block which contains other blocks and symbols declaration.
- **Declaration node** - The second one, the *declaration node*, can specify any implicitly or explicitly declared symbol in the source code, such as function declaration, class declaration, namespace, variable, constant, function parameter or even a language keyword. Also a *declaration node* can be internally linked to a *scope node*, as the class declaration with the scope containing member' declarations.

All the nodes among other things also contain their current location in the source code, because it enables us to find the tree node representing the code scope at specified position and reversely to navigate to the specified declaration location in the source code.

In general the IntelliSense tree describes the source code file in the same way as the syntax tree does. Hence it would be possible to define mapping from ASTs or another source file interpretation to the IntelliSense tree. Note that some elements of typical AST (e.g. function calls) do not need to be used in the process of mapping, but they could be used in the next processes of semantic analysis; so these AST elements should be just remembered in the corresponding scope node.

In addition to this, tree nodes provide methods for language integration support, such as selecting local declarations, getting symbol description or listing object members; therefore their single implementations have to be able to perform analysis of the tree to gather required data. These analyses are only performed when the data are requested.

5.1.1 Scope node

The scope node works like a container for other scope nodes and declarations. It corresponds practically to block of code in the source file. Also it is used as the root node of IntelliSense tree. See Table 1 for the base scope node member properties.

Additionally there are properties important for implementing features of the language service, needed for identifying the scope node and other information used mainly for determining a list of locally reachable declarations:

- **File name** - A path to the source file, where the scope is located, is used for identifying particular declarations. It can be served to the user within the symbol descriptions. Practically the file name value is equal to the file name value of the parent scope node if it exists. Only the root scope node has to be able to store the file name, other nodes just point to this value.
- **Used namespaces** - A list of namespaces used by the code in this scope (and child scopes) is used for gathering locally reachable declarations. Used namespaces are typically named within the "uses" or "import" statements.
- **Inclusions** - In any code scope other source files can be included, typically using the "include" statement. Therefore the list of included files has to be stored in the scope. Reversely the root scope can be included by other scopes (because the root scope represents the whole tree which describes the source file). Then the scope has to be able to get a list of scopes where it is being included. These lists are used for gathering possible declarations visible in the scope, because the declarations from the included files and from the parent scopes are mostly visible too. For resolving the inclusions see 5.3.

This scope node specification represents an abstract object. Then the node derivations have to be implemented (Appendix B. Figure 13) for the specific scopes and specific sources, such as various syntax trees etc. Particular implementations manage the mapping from the source and semantic analysis of the source. For example the PHP global scope (root scope of the PHP source code) and also the generic PHP code scope have to be implemented. Both of them will use the PHP syntax tree as the source of their child scopes and declarations and as the source of next analyses.

Scope property	*Description*
Position	Location span of the scope in the source code string.
Declarations	List of declared symbols (*declaration*s) in the scope.
Scopes	List of other code *scopes* written within this scope

Inclusions	List of included file names is remembered. IntelliSense trees representing listed files are then used to obtain additional local declarations.
Used namespaces	List of namespaces used in the scope. Their declarations are implicitly visible in the current scope and its child scopes.
Parent scope	Reference to the scope containing this scope, it is *empty* in case of the root scope.
File name	Path to the file where this scope is placed.
Scope type	Specified scope type, such as *code block*, *function body*, *class body* or *namespace body*. Not important.

Table 1 Scope node interface specification

5.1.2 Declaration node

The base type of the declaration node is more complex (see Table 2 for its interface specification) and it is the main part of the IntelliSense tree. It is designed to describe symbols declared in the source code and to manage various information about them. Especially the symbols important for IntelliSense implementations should be supported, such as variables, constants, class declarations, functions or namespaces; declared implicitly or even explicitly.

Therefore this node contains not only the symbol name and text description, but also the information for specific symbol type must be present; such as possible object members or function signature. There are also many properties which describe the symbol type, possible values of variable, object instance members, object static members, symbols in a namespace, array items or indirectly used symbols. This is solved using following members of the base declaration node:

- **Name** - The name of the symbol, as it is used in the source code.
- **Description** - Known text information about the declaration (symbol description) can be taken from various meta-data such as comments near the declaration or the online documentation; the file name and position where the declaration is placed can be appended too. Finally it contains all the available information about the declaration. The *language integration* can offer this information to the user.
- **Symbol type** - The declaration is also described by its *symbol type*; it describes what specified node declares. There are many symbol types like variable, constant, function, namespace, class and keyword. It is used by the *IntelliSense integration* to select appropriate symbols in the specified context. It is also used for displaying an appropriate icon next to the symbol name, when a declaration list is populated.

- **Visibility** - Especially objects within a class declaration have defined a visibility (access modifier). It tells the compiler (and in this case the language integration) in what code scopes and how the symbol is reachable. Typical access modifiers are public, protected, private, static and constant. The resulting *visibility* flag can be their combination.
- **Function signature** - In case of function, all its signatures have to be stored. This information has to be available through the IntelliSense as the method parameter lists. Also function parameters can be used during the analysis of the function body code, to declare local variables from parameters and to determine their possible type or values.
- **Analyzers** - The information about every symbol can be obtained from many various expressions. Symbol values, its type members or dynamically added members imply from single symbol usages. The declaration node then has to collect these usages, analyze them and offer the results (object members, static members, possible values and array items declarations). Therefore an analyzer object is created for every symbol usage. There must be different implementations of analyzer for different symbol usages, e.g. assignment, add operation, function call etc. These analyzers are saved within the declaration node. Whenever its members or values are needed, all the requested results from analyzers are unified and returned. This separation saves CPU time and memory, because a single analyzer can be shared by many declaration nodes and the analysis is performed only if some information is requested. For the analyzer object description see 5.1.3 and Table 3. The analyzers generalize the type resolving problem; they are used instead of types to obtain symbol information by subsequent processing of the code. Even dynamically added object members are then collected within the analyzer; in this way these members can be displayed by IntelliSense.

Most of the properties might be described as an expression dynamically and the particular declaration node implementation (an analyzer) has to analyze them and try to evaluate them on demand. That's an issue with dynamic languages, where even the declarations name can be described as an expression which’s value is known definitely only at runtime.

Using the analyzers, the declaration node may offer a list of possible values. It is used when the node describes a variable. These values are important for resolving dynamic indirect usages and indirect function calls. The values are expressed as expressions which have to be evaluated to obtain the result. These results are used as another declaration name at runtime or they can be used as a part of symbol description. See 5.2.2 how the expression evaluation can be bypassed and why.

Declaration node property	*Property description*	
Position	Location span in the source code. Used for navigating to the declaration.	
Name	The text name of the symbol.	
Analyzers	List of symbol analyzers. Primarily used for obtaining members and values.	
Parent scope	Reference to the scope node where the declaration is placed originally.	
Symbol type	The type of the declaration, describes what the symbol means.	
	Type	A class or a value type.
	Variable	A variable declaration.
	Constant	A constant declaration.
	Namespace	Namespace container.
	Function	Function declaration.
	Keyword	A language keyword.
Description	Text description of the declaration.	
Visibility	Declaration visibility flags, e.g. *public*, *protected*, *private* or *static*	
Function signature	List of signatures used for function call assistance; every signature defines a list of parameters with parameter name, type declaration and parameter description.	

Table 2 Declaration node specification

5.1.2.1 Declaration node implementation

There have to be several node implementations (Appendix B. Figure 14), each one for different symbol type and other source data, e.g. variable symbol initiated from specific PHP syntax tree node etc. The node has to be able to work with its source data and offer relevant semantic information. For example each implementation can offer text description, which is more relevant to the specific symbol type. Implementing specific declaration nodes manages the mapping into the IntelliSense tree (6).

Typical declaration node implementations, which are initiated from the Phalanger AST (PHP syntax tree), are listed below:

- **Type declaration** - It uses the class declaration and analyzes its body to obtain class members. The symbol description is constructed from the documentary comment above the declaration if it exists. The analysis is performed only if class members are requested. Also special keywords which can be written within the class scope are added.
- **Variable declaration** - Variable declaration node is created if a variable usage is found for the first time. Every other usage (especially assignment) is analyzed using a new analyzer object. This simply solves the type resolving problem by tracking the variable value through the assignments. Also the comment near the variable declaration can be used and analyzed to obtain variable description or even its type name.
- **Constant symbol** - Constant constructs from the AST can be caught and used for initializing the constant declaration node. Its name, description from comment and initial value is used.
- **Namespace block** - Symbols declared within the specified namespace are grouped and stored in the namespace declaration node. They are then accessible through this node. The code scope corresponding to the namespace contains this namespace name in the "Used namespaces" list in default.
- **Function declaration** - The declaration node implementation, which is initiated by the function declaration construct, offers function signatures, description from the user comment above and also it uses all the analyzers initiated from the expressions within the *return* statements. So it is able to manage possible function return values and its return value object members.
- **Keyword symbol** - Special declaration node which enables to add a keyword into the list of symbols visible in the local code scope (and its child scopes). It is typically used to add keywords into the context of scopes like "self", "return", "break" or "continue".

5.1.3 Analyzers

The declaration node needs to manage its possible object members and its possible values. In static languages the members are typically obtained through the object type. But in dynamic languages they can be collected from various sources; the *analyzers* are used for that. The *analyzer* is a helper abstract object (Table 3) which offers a list of object members (in a form of declaration nodes) and a list of string values, using the analysis of a part of the source code (typically specific expression) and the knowledge of the rest of the analyzed code.

Every *declaration node* contains the list of *analyzers* and every *analyzer* gathers required data from different source, such as an AST expression node or .NET type object. There are many possible sources and therefore one or more analyzers are used within one declaration node. For every

symbol usage, an analyzer is created and added into its declaration node. Also one analyzer, which describes specific expression, can be shared by more declaration nodes. It saves memory and CPU time; in addition to that it partially solves the type resolving problem through the tracking of a variable value.

An analyzer is a certain replacement for the object type. It manages only object properties, such as its members and its possible values. Every symbol is described by the set of analyzers, where each one gathers data from different source. Properties of the object are able to be extended by adding a new analyzer into the set. Analyzers are reusable, it is then possible to inherit or copy properties of another object by referencing its existing analyzer.

Analyzer property	*Property description*
Values	List of possible text values.
Object members	Object instance member declarations. (after "->")
Static members	Static member declarations. (after "::")
Namespace members	Declarations in the namespace. (after ":::" in Phalanger)
Indirect declarations	Declarations used from indirect usages of this object.
Array items	Analyzers of array items.

Table 3 An analyzer interface

Typical *analyzer* object implementations listed below have to manage various source expression analyses:

- **Class declaration** - The class declaration content is scanned and its member declarations are saved into the "Object members" and "Static members" lists. Also the members from the analyzers of the parent class declaration are added, if the parent is defined. This analyzer is then able to provide class members.
- **New expression** - A result of the "new" expression is the analyzer which returns public members of the specified class. The class name has to be resolved and the corresponding declaration node is used. Using the analyzers from this declaration (typically one "Class declaration**"** analyzer) their public member declarations are referenced in the "Object members" list.
- **Variable use expression** - A product of the variable use expression is the analyzer, which just references all the analyzers of the specified variable and returns all their members. It can be

used only if the variable name can be resolved and the corresponding declaration node can be found.

- **Assignment expression** - This works like a variable use. The left operand is used and its corresponding declaration node is resolved. As a result the declarations analyzers are referenced without any change.
- **Indirect variable use** - The expression within the indirect use is semi-evaluated and its possible values are collected. Then all the declaration nodes matching some string value are used and all their analyzers are referenced.
- **Custom analyzer** - For the symbol information which is gathered within the semantic analysis there is also custom analyzer. Its properties can be filled by declarations manually and it is used for collecting dynamically added members. When a non-existing object member is used, a dynamic analyzer is created and added into the declaration node corresponding to this object. The analyzer contains the dynamically used member in one of its lists. In this way the type of specific object can be extended.

5.2 Dynamic information

With dynamic languages it is necessary to perform the semantic analysis to collect information about each declaration from the whole source code semantics. Therefore even particular expressions have to be processed and used for declaring new or modifying existing declarations. These declarations are stored in the IntelliSense tree, where they can be easily found or created and modified.

The analyzers are used for expanding an existing declaration with new information. If there are new object members or variable values used in the source code, a new custom analyzer containing these extensions is added into the appropriate declaration node.

Also a declaration can be declared or used in an expression as its evaluation. Even the compiler doesn't know the result of every expression, so it cannot know all the declared symbols too. But the language integration has to offer all the possibilities. Therefore it has to estimate what can be possibly written into the source code.

The only exact way how to list all the possibilities is to execute the source code. But the execution has usually many side effects, and every time the program runs it can produce different results depending on the current hardware, time, files, random generators etc. Hence making the execution is not a good idea.

The particular expressions have to be bypassed somehow to get an approximate result and meaning. Some simple expressions can be actually executed, other ones can match predefined

patterns and then semi-evaluated. The rest of the expressions can be ignored with the awareness of incomplete information stored in the IntelliSense tree.

5.2.1 Type resolving

IntelliSense is based on types. Among other things in dependence on the symbol type (e.g. variable, class or function return value) it offers object members defined within the object type. But in dynamic languages one symbol can represent instances of more types in different program executions and different locations. In addition to that the symbol can be extended by adding new member declarations dynamically besides the type's members. Therefore the single type is not sufficient for getting all possible symbol members. See 5.1.2 and 5.1.3 how the symbol members are managed.

Hence the language integration must watch for particular symbol usages. Most important are all the assignment expressions where the symbol gets a new value. The type resolving is based on this; within the assignment all the properties of the right value are added (without duplicities) into the declaration node describing the left value. The single properties are expressed within the analyzer objects; they can be simply referenced and added into the declaration node. Just note that the value on the right side must be resolved first before using its analyzers. This mechanism works even for dynamically added members, because usage of a non-existing member extends the symbol properties with a custom analyzer. It is then referenced within the assignment operation with all the other analyzers.

This way the symbol properties (especially its members) are collected by tracking the symbol assignments recursively, until its possible values source is reached (e.g. "new" expression, literal or some irresolvable expression). The symbol is then described by all the analyzers which could be resolved. They can then be used to obtain all the possible object members.

In case of function calls, the return statements must be found in the function definition body. Then the expressions that are specified within these statements are used. The expressions are analyzed and the resulting analyzer objects are used for describing the function call symbol.

5.2.2 Expression patterns

The expression patterns are helper objects used to recognize and semi-evaluate specific expressions. The pattern is an abstract object which manages the matching with a part of AST and its evaluation in general. The set of predefined patterns are used to recognize most used expressions in the source code. Expressions not handled by any expression pattern are typically ignored.

Every pattern can define a small sub-tree of the *AST* (expression without values) with the semi-evaluation code which returns an *analyzer* (5.1.3). This *analyzer* may contain approximate results

for the given expression in its value lists or it can offer a list of object members – this depends on the specific pattern implementation.

Method	*Returns*	*Description*
IsMatch	True of False	Checks if the given parameter (typically a part of AST) matches the pattern.
GetAnalyzer	Analyzer	Using own semi-execution routine it creates an analyzer with given parameter results.

Table 4 The *expression pattern* interface, functionality depends on the specific implementation.

The specific expression patterns (Table 4) must be implemented; their usage is very flexible and works well. For a specific language there are numerous frequently used expressions. These expressions can be recognized by the *language integration* using the *expression patterns* and they can be semi-evaluated. Thanks to this mechanism, safe expressions with no side effects can be processed and analyzed.

Pattern	*Resulting analyzer*
Variable use	Returns all the analyzers from the declaration node which describe used variable.
Literal	Creates the analyzer that offers given literal value in its values list.
new (*type name)*	It tries to resolve the type name. Then it fills the member list with accessible members of the *resolved* declaration node.
Indirect Variable use	Declaration node describing the variable symbol is resolved. Then the string values within this node are used to obtain possible indirectly used symbol names. The resulting analyzer then offers members from all the declaration nodes matching obtained symbol names.
Otherwise	No analyzer is created.

Table 5 Example patterns and their semi-execution procedure.

It is then sufficient to define patterns e.g. for indirect usage, simple expression evaluation and other basic dynamic and non-dynamic operations; see Table 5 for an example. Expressions which match the defined patterns are transformed to the specific analyzer by the *language integration*. Then the *analyzers* can be used to obtain a result value or expression object members in general.

5.2.3 Dynamic members

In dynamic languages it is usually possible to add new members to an object during runtime. Therefore wherever an object member is used in the source code, it has to be checked by the language integration whether it is an existing member of the parent object declaration. If it is not, then this dynamic member has to be added into the list of object members. See 5.1.2 and 5.1.3 for managing the symbol object members.

The *custom analyzers* are used for adding new information about the symbol during the semantic analysis. A new *custom analyzer* with this new dynamic member declaration is created and added into the parent object declaration. As a result the origin declaration members are mixed with other members. In this case the dynamically used member is added into the set of current object members.

5.2.4 Dynamic evaluation

As well as dynamic file inclusion, in the same way the dynamic code evaluation works. Each of these features can be implemented by the second one. Typically it is possible to get the code string and then evaluate it, by using the language native function (e.g. "*include*" or "*eval*"). The dynamic evaluation has the same effect as the evaluated code string would be simply pasted into the place of evaluation method call. The problem for the language runtime and for language integration implementation too is that the evaluated code string is known only at runtime. Among other things the dynamic evaluation can change the set of defined symbols and their meaning too.

The file inclusion is simpler, because only the file name must be resolved (see 5.3). But general dynamic string evaluation is practically irresolvable until runtime. It is used typically to evaluate complex strings, containing declarations and calls, constructed from various source values with many possible results. Hence the general code evaluation should be ignored by language integration completely.

5.3 Linking trees

File inclusion statements are very important and they should be handled by the language integration. The inclusion enables to use symbols declared in source files which are referenced from the current source code. Dynamic languages typically support the ability to express the included file name as a generic expression which has to be evaluated at runtime.

There are several methods how to manage symbols from the referenced source files. In general it is sufficient to manage the referenced file names. Then the symbols accessible through the included file can be obtained by using the corresponding IntelliSense tree. Methods listed below can be used:

- **Ignore inclusions** - In simple implementations of language integration the inclusion statements are ignored completely. Referenced files within the current source code are not processed. Only symbols declared in the current source code and maybe in some predefined source files are available.
- **Include everything** - The second method offers all the referenced symbols and moreover it doesn't have to evaluate the included files name expression. It simply uses all the symbols in all the source files within the current application directory and all subdirectories. But therefore even not accessible symbols are offered by the IntelliSense. For large applications it can be annoying and sometimes unusable.
- **Resolve inclusions** - Most of expressions within the inclusion statements can be resolved before the runtime. That's why the scope nodes (5.1.1) contain the list of IntelliSense trees which are included there (referenced source files and the corresponding IntelliSense trees). The accessible symbols then might be obtained through this. The included file name must be resolved to obtain the corresponding tree. This file name is a generic expression placed within the inclusion statement, which can be fully evaluated only during runtime in dynamic languages. Therefore the expression has to be semi-evaluated, e.g. using patterns (see 5.2.2). Fortunately, in most cases, the target file name is expressed as a simple string literal, but otherwise the semi-evaluation might fail; then the corresponding tree is not linked with the code scope and the declarations within the referenced file are not available.

In dynamic languages, inclusion statements can be placed within the code blocks whose execution depends on some conditional expression (e.g. "if" or "switch" - evaluated at runtime). In this case all the inclusion statements should be processed, independently on the condition - because the language integration should offer every possibility and the included files can differ at runtime. Therefore conditions evaluation is not important for language integration and proper IntelliSense working.

The file inclusion works in the opposite way too. Accessible symbols can be obtained not only from the included source files, parent scopes are used too. Note that in case of root scopes the parent scope is also the code scope from which the current source code file is included. Therefore the scopes which references current code scope should be managed too. This is needed to obtain every possible declaration (see 7.1.1).

5.3.1 Reference cycling

Using the referenced trees linking it is possible to build a graph of linked trees. This graph can be used then to detect possible cycles, which can be reported to the user as a warning (e.g. using the common visual studio error sink together with syntax errors). However, cycles can be detected wrong e.g. in case of conditioned inclusion statements.

Note when the locally accessible symbols are collected, the cycle in linked trees must be detected to avoid cycling. During runtime it is typically ignored and the application crashes (stack overflow), but the language integration just needs to list available symbols.

5.3.2 Memory leaks

The trees should not be referenced directly. Only the file name (as tree identifier) should be used instead of the direct tree reference. Every time the linked tree is requested it is obtained from the tree cache (5.4) by its file name. Then the last updated version of the linked tree is used every time and trees are managed at the one place (tree cache). All old versions of the trees are not used anywhere and they can be garbage collected automatically.

5.4 Tree cache

The *IntelliSense tree* of every used source file within the application directory is stored in the global in-memory cache. Every tree is identified by the corresponding full file name. Then unchanged source codes need not to be reparsed and the last version of its tree can be obtained here.

5.4.1 Trees updating

Each source file has its own tree built by the *language integration*. This tree is rebuilt every time a source file is changed. Hence the *language integration* is trying to process source codes which are not necessarily syntactically valid. This situation occurs when the source code is just being modified by the user. Then the IntelliSense tree can't be created directly. But the tree is needed even for non-syntax valid source codes to keep the IntelliSense working.

For the IntelliSense when the source code cannot be parsed, the last cached tree of the processed source file is used, but with some modifications. When the source code is being edited, the source positions of the scopes and declarations in the cached tree do not reflect their current positions in the source code. Hence with every added or removed line or character in the source code, the positions of the tree nodes have to be moved in the same way by the same amount of characters and lines. Let us note that only the positions located below the changes must be corrected. When some part of the source code is removed, tree nodes in this area should be removed too. This modified tree can be used even for the changed syntax invalid source file; only the new code text added within the syntax invalid code is not parsed and it is not contained in the IntelliSense tree.

6 Mapping into IntelliSense tree

Creation of the *IntelliSense tree* for the specific source file depends on the origin language (language, file format, grammar). The source has to be analyzed and then particular language elements are converted into the tree nodes subsequently. The typical process consists of the source code parsing and creation of the *AST*. The visitor pattern is used on the *AST* and source code elements are converted into the corresponding implementations of *IntelliSense tree* nodes. This process is re-executed whenever the source is changed. Typical process is depicted on Figure 9.

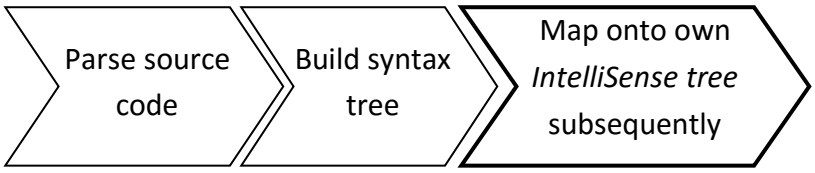


Figure 9 An IntelliSense tree creation

The tree can be created from various source files, like .NET assemblies or PHP source files. Whatever the sources are, the *IntelliSense tree* implementation unifies them. Once built, additional information is added to the tree by subsequent processing (semantic analysis). Note that the initial tree build contains all the static declarations; the following semantic analysis gathers dynamically used symbols and then they are inserted into the tree subsequently.

There must be implemented a special IntelliSense tree node for every source code construct. These nodes are derived from the scope and the declaration base node types. The source file is then scanned and every relevant construct is converted to the corresponding IntelliSense tree node implementation. Newly created node is initiated by the origin element, so the node remembers its source data; its child nodes can be created by the node when they are requested only. Also additional information, such as description or dynamic object members, can be extracted later - typically when they are resolved during scanning (subsequent processing).

In dynamic languages the tree node information is gathered from various source elements. The whole source file, in a parsed form - AST, must be processed and every expression has to be analyzed. As a result it can extend some existing node information or add new declaration node (implicit declarations). In static languages it is typically sufficient to scan declaration constructs only (class declaration, variable declaration etc.); single expressions are not needed for collecting available symbols and their description.

Following source files are used in the presented work to enable the support for PHP language. Corresponding source constructs are mapped onto own IntelliSense tree nodes implementation.

Using the same architecture of IntelliSense tree, it is possible to use different source files within one language integration. Various source files (.NET assembly, PHP source, PHP extension) might be referenced from another source code file, and language integration mechanism needs not to solve any differences. Defined symbols from many various source files may be used then together. The list of file types follows:

- **PHP source file** - The file contains PHP source code and references to another PHP files. The whole PHP project might reference .NET assemblies - their symbols must be available during the development in the same way as the PHP constructs are. PHP source file is identified by the full file name.
- **.NET assembly** - Type declarations are defined in .NET assemblies. Every type is placed in a namespace and contains members, such as methods and properties. The declarations are static only. .NET assembly can reference other .NET assemblies. An assembly is identified by the full assembly name (it is not a file name).
- **PHP .NET extension** - Special case of .NET assemblies are PHP extensions generated by Phalanger. These assemblies are just wrappers for PHP native extensions. They contain type declarations within one specified namespace. Every type is described by an attribute which specifies the PHP class name. Methods can be also denoted by .NET attributes to specify the regular PHP name or define global functions (the method can be used as a global function). Type properties can be denoted as PHP constants. One of these special .NET assemblies also contains most of default PHP functions (PHP class library) and it is referenced by all source codes in default.
- **Native PHP symbols** - Native PHP symbols ("echo", "include", "isset", "_GLOBAL" etc.) must be defined within the IntelliSense tree too. The special tree is created and its global scope contains these symbol declarations. This tree is referenced by all other source codes in default.

6.1 PHP source file

The conversion of PHP source code into the IntelliSense tree consists of several steps. First of all the code must be parsed and the syntax tree then created (4.3, 4.3.1). In the AST statically declared constructs are defined directly. They can be inserted into the IntelliSense tree. Finally by analyzing single expressions, implicit declarations and some additional information are collected.

6.1.1 Static information

Starting from the root, by walking through the all particular AST elements, corresponding implementations of IntelliSense tree nodes are instantiated and inserted into the IntelliSense tree. The resulting tree reflects the same hierarchical structure of code scopes as it was in the origin source code. As a result statically defined PHP constructs (classes, functions) are mapped into a

representation of IntelliSense tree. The following AST nodes are caught by the visitor pattern (static code):

- **Global code** - Global code represents the root of AST. It is converted into the root of IntelliSense tree, which is implemented by *GlobalScope* type and it is derived from generic *Scope* node. Its parent scope is empty and FileName property is set to the source full file name. The child AST nodes are processed and their results are placed within the *GlobalScope* node (other scope and declaration nodes).
- **Code scope** - Code scopes, such as function body, if/else body or general code scopes, are converted into the *CodeScope* node (derived from *Scope* node). Its child elements are placed within this scope. It is used for encapsulating local declarations of variables, in order to use them only in current and child scopes. The source code hierarchical structure is then preserved. FileName property of the code scope points to the value of its parent scope node.
- **Function declaration** - In PHP, function is declared and defined within the single function declaration statement. It is converted into two IntelliSense tree nodes - the declaration and code scope. Function declaration node (*PhpFunctionDecl*) contains the function name and function signature (parameters list). Also the function declaration description is initiated if the corresponding AST node contains function documentary comments. The declaration is linked with the code scope node, which is created from the function body scope.
- **Function parameter** - Visitor catches the function parameter nodes. They are then added into the corresponding function declaration parameters list. Also the parameter is converted into the variable declaration node, this is added into the function code scope (it represents the local variable declaration). The variable type can be estimated using the documentary comments above the function. If it is specified, the type analyzer from the suggested type is created and added into the created variable declaration node (same as it would be assigned with the "new" expression of the suggested type).
- **Class declaration** - AST contains statically declared classes. They are converted into the class declaration node (*PhpClassDecl*) and placed within the current scope node. The class name is typically a simple string literal. The analyzer which collects class members from the class code scope is used then and added into the class declaration node. If the class is inherited from another class, the base class declaration node is found and its analyzers are used to gather its public and protected members.
- **Import statement** - Phalanger adds non-standard keywords into the PHP syntax. Because of .NET interoperability there is the "import namespace" statement. The namespace name follows and it is a string literal always, because this must be known during the compilation to generate MSIL code properly and to add correct assembly references into the generated code. Imported namespace is used by the visitor and added then into the "Used

namespaces" list (5.1.1) of the current scope node. The language integration then collects the locally visible symbols from this used namespaces in default (7.1.1).

6.1.2 Dynamic information

In addition to statically defined symbols another declarations are added using the semantic analysis of the abstract syntax tree. Particular expressions are scanned to find e.g. implicitly declared variables and dynamically added object members. Also possible variable values and types (over the list of analyzers) are discovered. The AST and so far built IntelliSense tree are used for that. Among the others following AST nodes are caught and used for this analysis:

- **Assignment expression** - There are the basics of the type resolving; the analyzers of the R-value (collected using the expression patterns) are just added into the L-value. The L-value is a variable use expression, this must be processed too and its declaration node must be found. The specific symbol is then described by all the analyzers from every assignment operation (without duplicities).
- **Include statement** - The inclusion statement parameter value (included file) has to be resolved (e.g. using expression patterns). Mostly it is simple string literal. Then the file name is saved within the current scope - into the "Inclusions" list.
- **Variable use** - VariableUseExpr construct consists of the whole member chain. There is the variable name and the IsMemberOf property which points to the parent object expression and which may be null. If it is null, the VariableUseExpr describes some local variable (e.g. $xxx), otherwise the VariableUseExpr is its member (e.g. some_parent->xxx).
    - If it is a member (IsMemberOf is not null) - the IsMemberOf expression must be processed recursively and the resulting analyzers are then collected. Using their "Object members list" the declaration with VariableUseExpr name is found. If there is no such declaration, the custom analyzer with this new object member is added into the parent object declaration (dynamic member).
    - If it is a variable - the corresponding declaration node is found in the tree, starting from the current code scope (7.1.1 - locally accessible declarations). If there is no such declaration, the declaration node for new variable must be created and inserted into the current code scopes "Declaration list" (implicit declaration).
- **Return statement** - The return statement is used to describe the corresponding function return value. The expression within this statement is analyzed (using the expression patterns) and the resulting list of analyzers is added into the function declaration node. It works like the assignment expression with the L-value of function declaration and R-value of the returning expression.
- **Function call** - Function calls are processed to gather possible input parameters properties (analyzers). Input parameters are analyzed (using expression patterns) and resulting

analyzer objects are added into the local variables of called function matching the corresponding parameter name.

6.1.3 Documentary comments

The language parser can catch specially formatted comments above the function and class declaration. These comments are called documentary comments (see Figure 10 for an example). They describe parameter types in dynamic languages. In PHP the notation of these comments come from the java doc comments.

When the comments are caught, the parser parses them and the result is placed into the corresponding AST node (as Annotations). The symbol description and optionally function return type and parameters description are extracted from the comment. The parameter types and return value type is used as a type hint.

```
/**
* Multiline description
* used for symbol defined below
*
* @param   type   description
* @return  type   description
*/
```

Figure 10 Documentary comment example.

6.1.4 Error underlining

Beside syntactic errors there are other possible problems that can appear during runtime. It is possible to detect them within the semantic analysis. In dynamic languages these problems mostly cannot be marked as an error, but as a warning - a possible issue. Therefore in practice it is usually possible to turn these features off in IDE. Eventual warnings are detected using the semantic analysis - during the mapping from the source code into the IntelliSense tree. Particular detected warnings are underlined in the source code, reported through the same mechanism used for syntax errors (error sink, 3.3.2.3, 4.3.2), together with their simple text description. Common issues and their implementations are listed below:

- **Use of uninitialized variable** - In dynamic languages there are typically no uninitialized variables, because every variable used for the first time is automatically initiated with some zero value. But sometimes the programmer wants to see the variables used without his own initialization. To properly detect these usages it is needed to build control flow graph (CFG) and find where the variable is used for the first time. Then only assignments (variable as L-value) or input function parameters (variable from current function parameter) are allowed as the first usages. Other usages should be reported. But in dynamic languages it may be impossible to determine CFG, e.g. the code can be executed within any other source code as an inclusion.

- **Unreachable code** - Using the control flow graph (CFG), parts of the source code which cannot be reached are detected. This can be reported to the user as a warning, since it typically means that something is wrong.
- **Undefined function** - Every function call can be easily caught (*FunctionCallExpr* AST node) and the corresponding function name is found in the IntelliSense tree built so far. When the symbol is missing, this function call can be reported.
- **Ambiguous variable type** - To keep the source code clean it is possible among other things to report variables containing different object types. Dynamic languages allow that but it can be a source of other problems in general. Assignment expressions can be analyzed; if the variable in L-value contains then analyzers describing different types, it can be then reported.
- **Inclusions of non-existing file -** During the resolving of the included file name (include/require statement) the language integration can check if the file specified here really exists. The inclusion of non-existing files can be reported, but the language integration cannot determine if the file will not exist during runtime.
- **Unused variable** - For every variable symbol all its usages can be managed (especially reading their value). These usages can be stored within the variable declaration node in IntelliSense tree. After the semantic analysis all the implicitly or explicitly declared variables (and function parameters) without any readings can be underlined. Typically the variable must be in any expression excepting the L-value part of assignments. Otherwise it is an unused variable.

In dynamic languages there are many situations which can be determined only during runtime. All of the possible issues described above can be resolved at runtime, e.g. by dynamic evaluation ("eval", which is not handled by language integration) or inclusion of the source file which couldn't be resolved before.

6.2 .NET assembly

Assemblies (MSIL DLL) are referenced by the application - it is possible to use the symbols defined in that assemblies in the PHP source code. Language integration must collect the symbols (types and their methods and properties) and be able to use the information about them. The mapping from .NET assembly is simpler, because it contains static declarations only and next semantic analysis is not needed. Every referenced assembly is loaded, IntelliSense tree is created and particular type declarations are mapped. The assembly is specified by its full assembly name; all assemblies referenced by the application must be processed, including the assemblies referenced by other referenced assemblies.

As a result for every assembly there is one IntelliSense tree in the cache of trees, which contains type declarations managed in the hierarchical structure of namespaces. It is because of faster selection; all types (class declarations) within specified namespace must be found quickly - when IntelliSense is offering a list of local declarations, even all the symbols from used namespaces must be listed. There may be thousands of declarations in the assembly; therefore every symbol is placed within its namespace declaration node in the IntelliSense tree. Declarations of the specified namespace (classes and other child namespaces) are obtained using the corresponding namespace declaration node analyzers, especially their "Namespace members" property (5.1.3).

6.2.1 Assembly processing

In the resulting IntelliSense tree there are only declaration nodes describing particular namespaces, types and their members. In the assembly there aren't corresponding elements for the code scopes, also the position property in the declaration node is ignored here. Since there are only static declarations, the process of IntelliSense tree creation consists of simple assembly load and listing its elements using the reflections. Following list describes the mapping from specified assembly elements:

- **System.Type** - These objects obtained from the assembly are converted into the type declaration nodes. Only public and visible types are used, the other ones are not accessible and they are not important for IntelliSense. First of all the type namespace is used and the corresponding namespace declaration node is found or created. For the namespace e.g. A::B::C, there is the namespace declaration node "A" in the root of the tree, then its namespace member "B" (through the analyzers of "A") and then its member "C" - the last one is then used as a container for the types placed within A::B::C. Using the reflections the newly inserted type declaration node is analyzed and its members are populated (using the specified Analyzer object which is then added into the type declaration node).
- **System.Reflection.MethodInfo** - For the method object, the function declaration node is created and added into the parent type declaration node analyzer. Non public methods are ignored. In .NET (as a static language) every method has return value and parameters list with the type specified. But there may be more than one method with the same name and different signatures - in this case the previously added function declaration node can be found and its list of signatures will be extended.
- **System.Reflection.FieldInfo** - FieldInfo is the named symbol containing a constant value placed within the type declaration. The declaration node is created and inserted into the parent type declaration node analyzer. Then new analyzer containing the value of the field is created and added into the field declaration node.
- **System.Reflection.PropertyInfo** - Property is the named member of the type declaration, which has the type specified and is able to get or set the value. The declaration node is

created and inserted into the parent declaration as a member. The type of the property can be used and analyzer object describing this type can be added into the property declaration node. The language integration need not to check if the set or get methods are allowed. The property is just offered within the parent object members list (like the fields or the methods are).

- **System.Reflection.EventInfo** - The event is a special object which holds the list of delegates (references to methods). It allows calling all of them within one event call. In C# there are overloaded operators (+= and -=) to adding and removing references into/from the event, however the PHP is not designed to work with events at all. Phalanger adds special methods (Add and Remove) to manipulate with the event object. Therefore the language integration has to do the same - the declaration node for the event is created and the member functions Add and Remove are added into its analyzer in order to be able to see them in the IntelliSense.

6.2.2 XML documentation

Assemblies and descriptions of assembly elements are placed separately. Documentary comments of elements from single assembly are stored in .NET XML documentation file. In order to offer the description of every declaration node, XML files corresponding to .NET assemblies must be loaded and processed. Then particular descriptions of elements must be obtained from this XML file.

6.3 Native PHP symbols

Built-in declarations of the PHP language must be present in a form of IntelliSense tree too. They can be then processed in the same way and integrated with the user defined symbols. Therefore there must be an extra IntelliSense tree which contains these declarations, mostly native functions such as "echo", "include", "isset", "empty" etc. This extra tree is included by all the PHP source files in default, thus the symbols defined here are visible everywhere.

The extra IntelliSense tree must be created in order to present all built-in symbols. In addition to that there are also standard keywords which are offered to the user when he's typing a word (auto-completion) together with the declarations. The keywords can be defined in the IntelliSense tree too, or they can be added into the resulting list of possibilities only when the list is populated (7.1).

6.3.1 Native symbols definition

To create the extra IntelliSense tree, the source database of built-in symbols is needed. They are used then to fill the tree. The process of tree filling (initialization) depends on the form of the source representation. The best option would be to use some already defined mapping, e.g. from the PHP source code or .NET assembly, then the extra tree creation would be easier. The list below describes possible options:

- **Hardcoded** - The built-in PHP declarations can be easily initiated in the language integration source code by creating corresponding declaration nodes directly; they are inserted into the empty extra IntelliSense tree root. This way is fastest, but in the context of programming it is not good-looking. The program and data are mixed together.
- **PHP source code** - Built-in PHP declarations can be defined in extra PHP source file. Then the existing mapping from the PHP code can be used to create IntelliSense tree containing these symbols. Existing symbols and their additional information can be easily changed and new symbols can be added without a need of language integration source code modifications.
- **.NET assembly** - The PHP compiler (Phalanger) defines most of the PHP native functions using the special .NET assembly. This assembly can be processed in the same way as the other assemblies are (6.2) through the reflections. All the built-in functions are placed within the predefined namespace. Therefore this namespace must be imported by all the source files in default.
- **Documentation or another text representation** - The amount of natively defined symbols can be obtained from some text representation, such as online language documentation. Methods of semantic web or another semantization processes may be used. The source text is parsed, e.g. using regular expressions, and specified elements are collected. These elements can represent a PHP symbol declaration, corresponding symbol description or link to another source text. In this way big amounts of built-in declarations can be extracted from already existing documentation.

In the presented work, declarations that are a part of the language syntax ("echo", "include", etc) are defined in the hardcoded way, because there are only a few of them. The rest of declarations (standard PHP functions and types) are defined in the special .NET assembly, because Phalanger provides it. Therefore there are two extra IntelliSense trees containing standard PHP symbols.

6.4 Optimizing

The process of single IntelliSense tree creation is performed every time the corresponding source code changes, hence it is suitable to implement some optimizations. Also there are trees containing symbols from large static language libraries, e.g. from .NET assemblies, which manage typically thousands of symbols. Initializing all of them may be too slow at language integration startup.

Some of additional information initialization can be postponed until it is requested, because not all of the source elements must be analyzed and mapped immediately. Also multi-core systems may be utilized and creation of tree nodes can be parallelized in some cases.

Even then the database of symbols may not be ready immediately. Especially in case of large system libraries, collecting of all the symbols information takes some time. Still the language integration must handle user requests from the beginning; so the initialization process should be executed on a

background thread and IntelliSense typically displays a message informing the user, that symbols database is not built yet.

6.4.1 Lazy mapping

The process of mapping doesn't need to be done immediately. In fact a single tree node can remember a reference to the origin source data element; then the mapping of child elements can be done on-demand and not before the content of the tree node is used.

Only the top level tree nodes are initiated immediately and the rest of the tree can be created later. This works for static languages, in dynamic languages all the tree must be created in order to perform semantic analysis.

In case of .NET assemblies, only types are added into the tree, their members need not to be scanned immediately. The declaration node describing the type references the source System.Type object instance. When the type members are requested, the type declaration node initialization is finished first.

Dynamic languages are typically able to modify local or global declarations everywhere. Dynamic members of global variables can be added within e.g. class member function call. Also global variables can be assigned here. Therefore the class content, its members and member function bodies have to be scanned and processed without postponing. But optionally theses cases might be ignored in order to speed up the process; the symbols information then will not be complete until the content of all the source elements is used and analyzed.

6.4.2 Multi-threading

Some tasks within the tree creation process may be performed independently. They can run on separate threads, so the process is parallelized. Especially different tree nodes can be created separately, as products of mapping source static elements.

For the following semantic analysis of the dynamic language code, the complete tree is needed. Various tree elements are modified and additional information is added, that is then used for the rest of analysis. Its eventual parallelization will require many locks, and then it does not have the desired effect.

6.4.3 Disk cache

Most of analyzed files are static, as a part of language libraries. But they are also large and their IntelliSense trees are needed every time the language integration starts. Hence when the tree is created once, it can be serialized into a persistent storage. Then the next time the trees from this cache can be easily restored.

Only trees of static libraries can be cached in this way. They do not contain any results from semantic analysis and they do not reference any elements from other trees. The serialization process is easy to implement and no additional objects are needed to be saved with. It is basically used for caching syntax trees of static languages used by IntelliSense, typically of .NET assemblies or C/C++ header files.

7 IntelliSense features implementation

The IntelliSense tree provides an in-memory database of symbols. The symbols are organized in a structuralized form as a tree of source code scopes and declared symbols. The IntelliSense tree nodes also contain additional information as a result of the syntactic and semantic analysis. That's used for implementing specific IntelliSense features, such as obtaining a list of available declarations in the specified code scope and getting corresponding information about them as well. Optionally the resulting list can be filtered to get one or more specified declarations which match a specific word or symbol type.

7.1 Declarations list

The main IntelliSense feature is the ability to offer a list of declaration (symbols) corresponding to the specific sequence of characters (token, word) in the specific location. This word can be used as a full declaration name or as its part only. If it is a part only, the resulting list of declarations may be offered as the possible word suggestions. If the word specifies the full declaration name, the additional information from the matching declarations is typically shown to the user or it is used for internal purposes. By implementing this functionality it is typically possible to implement also the rest of IntelliSense features, such as tip texts, method info or determining the specific declaration location.

Therefore IntelliSense must analyze the specific word in the context of other symbols and provide possible declaration nodes from the IntelliSense tree. This word can represent a single variable name, function name, type name or a member of some object. Typically the expression ending by this word must be recognized and its syntactic structure must be determined. See 4.4 how to determine rough syntactic structure of a part of the code.

The code context is needed for determining the accessible symbols. It is suitable to use the code scope node, which is determined using the specific word location. This code scope contains local declarations, a reference to the parent code scope and additional scope information (5.1.1).

7.1.1 Local declarations

A list of visible local declarations depends on the current language. In most cases the declarations can be collected from the current code scope and parent code scopes subsequently. In addition to that included trees root scope and optionally scopes which include the current scope might be used too. The last one allows the user to work with declarations which may be visible, if the source code file is executed as an inclusion.

If the language supports the namespaces, all of *IntelliSense trees* within the project should be processed to find namespace declarations imported in the current code scope. Then also the declarations in these namespaces are listed as local visible declarations. A typical selection process is depicted on Figure 11.

```
include "header.php";

function foo($a, $b, $c)
{
    ...
}

function bar($d)
{
    include "methods.php";

    if ($x)
    {
        $y = $x + 1;
        ????
    }
}
```

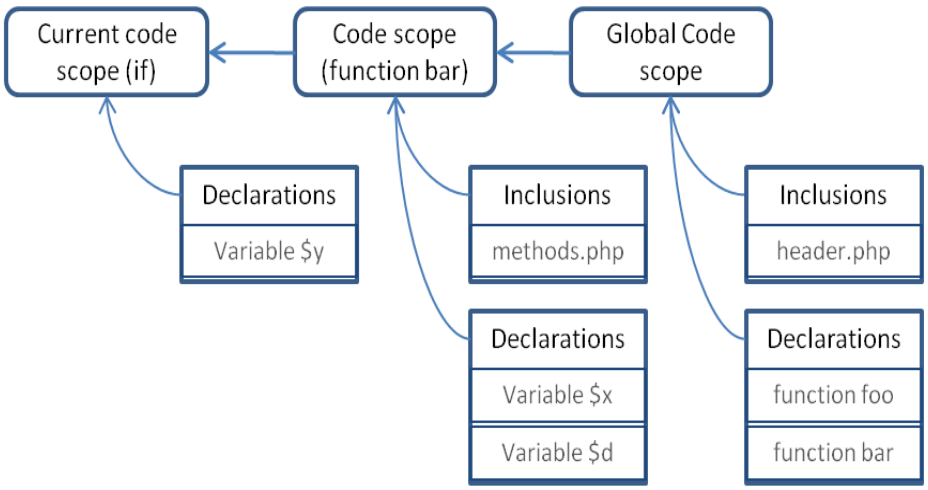


Figure 11 Source code example and selecting visible declarations in specified code scope.

7.1.2 Expression declarations

To enable functionalities of the IntelliSense, the declaration nodes within the IntelliSense tree that correspond to the specific symbol must be found. The symbol can typically occur as a part of a more complex expression, e.g. as a member of some object. Therefore the syntactical structure of the expression has to be determined first (4.4). This structure is used to select the right declaration node in the context of the expression.

Not all the expressions have to be recognized; only variable usages, type usages or function calls are important. The issue with these expressions is that they can be used as a part of some expression chain, optionally with array indexers or indirect usages. See Figure 12 for an example of possible expression chains. All the symbols must be recognized subsequently to get the declaration nodes of the last symbol in the chain.

```
$symbol

$x->symbol

(new MyClass)->$symbol

{$x}[$y]->symbol()
```

Figure 12 Possible expression chains

First of all the expression has to be parsed and transformed into some kind of a unified data structure (4.4). Typically this is an expression that ends under the cursor and it is transformed into a part of the language-specific AST (sub-AST). It is the appropriate unified data structure describing the syntax of the expression with the already defined visitor pattern functionality. Just note that many of syntax tree properties need not to be specified.

`$A[123+456]->C`		
Variable `$A`	Indexer `[…]`	Member `->C`
Local declarations with the name "A" and type Variable	Select all array analyzers from *"$A"*	Select all object members with the name equals "C" from *"$A[…]"*

Table 6 Example of an expression chain and its declaration nodes

The sub-AST of the expression describes the way how to obtain the matching declarations from the IntelliSense tree. Table 6 shows an example of such processing. Following list describes the way how to process specific elements of AST:

- **Variable use** - This construct describes a variable. It has the name specified and it can occur as a part of a members chain. That's why there are two possible forms of this construct:
  - Variable is a member of an expression (IsMemberOf property points to an expression). The parent expression must be processed first to obtain its list of analyzers (expression patterns can be used). The resulting list is then used; all analyzer's object members matching the variable name are selected and returned.
  - Local variable use (IsMemberOf property is empty). In this case all the local declarations (7.1.1) matching the variable name are selected.

  Also the selected declaration nodes are filtered; the nodes with the type of variable remain only.
- **Array access** - The expression can be accessed through the array indexer. When the analyzers of such expression are requested (e.g. to obtain object members), its analyzers are not returned immediately. They are used first to collect array items analyzers instead. The "Array items" property of analyzers of declaration nodes describing the expression are used for that. (Expression patterns solves processing of such expressions, see 5.2.2)
- **Indirect usage** - An expression can be used for describing the name of another symbol. The declaration node corresponding to that expression is resolved, and through its analyzers possible string values are collected. The resulting list of values is used to collect the indirectly used symbol declaration nodes. (Expression patterns solves that, see 5.2.2)

- **Type usage** - This AST node describes the type name. The local declaration nodes (7.1.1) are filtered to contain the nodes describing the types only.
- **Function call** - The process of selecting corresponding function declaration nodes is very similar to the processing of Variable use. Only at the end, declaration nodes with the type of function are filtered.

The stack of declaration node lists is used. Within processing specific AST node, the top of the stack can be popped and the list of newly selected declaration nodes can be pushed here. When the selection process ends, the declaration node list on the top of the stack corresponds to the symbol on the end of the expression. It implies from the hierarchical structure of the AST and the order of the processing particular AST nodes. The single expressions (in case of processing the IsMemberOf property) can be analyzed using the expression patterns to obtain a list of expression analyzers.

7.1.3 Word completion

When the symbol (token) is being typed, an editor environment requests the language integration for a list of declarations - as possible completions of the so far typed symbol. This list is requested every time the user presses a keyboard character or when the user requests for the auto-completion manually (typically by pressing some keyboard shortcut (e.g. Ctrl + Space in Visual Studio)).

Then the expression under the cursor has to be parsed (4.4) and processed to get corresponding expression declaration nodes (7.1.2). The only difference is, that the names of listed declarations need not to match so far typed symbol exactly - the symbol string just represents a part of some existing declaration. Therefore all the declarations that are a superset of the symbol string are listed. They don't necessarily have to be the same; small changes in the processing of single sub-AST nodes during the selection process (7.1.2) are required. If the AST node describes the symbol under (before) the cursor, e.g. user is just typing this symbol, the processing method is a little bit modified:

- **Variable use, Type usage, Function call** - All the declaration nodes selected in the previous implementation are listed if they just contain the symbol name. Also the list is not filtered by the specific declaration type, which is typically unknown until the symbol is completed.

If there is no expression to process, meaning that user wants to auto-complete an empty space, the list of all local declarations can be selected to display all the possibilities which could be used.

The resulting list of declarations is offered to the user. In case of only one item in the list, the symbol (token) under the cursor can be auto-completed immediately.

7.1.4 Special cases

In case of word completion, the language integration should modify the offered declarations list by the current code context. The behavior of the modifications typically depends on the language. Current code scope type can be used to determine possible modifications. Also the tokens before or after the processed expression might be used.

Various list adaptations are usually performed. The list below describes situations within the PHP language. Let's assume that the expression under the cursor is processed already, the list of possible declaration nodes for the auto-completion is populated, and the token before this expression is known:

- **Token before the expression is "new"** - The resulting list of declaration nodes is filtered to contain only the nodes describing the types.
- **The symbol under the cursor is not in the list** - There are no matching declarations for the word typed by the user. Typically it means that the user is declaring a new variable implicitly - by using the variable name for the first time. This new declaration should be added into the list with a description saying "new variable".
- **Code scope is not class declaration, just generic code block** - Special items are added into the list. These items are called snippets. They describe frequently used constructs in the source code, e.g. "if", "switch", "function" etc. Typically all the possible language keywords must be added into the list to enable the user to type them. By selecting the snippet from the list by the user, the whole construct is inserted into the code, including possibly following parentheses.
- **Code scope is "switch" block** - The snippets for "case", "default" and "break" keywords are added into the list.
- **Code scope is "for", "while" or "foreach"** - The snippets "break" and "continue" are added into the list.
- **Code scope is function code block** - The snippet "return" is added into the list.

Many similar situations can be handled. The intention is to offer the list of words as meaningful as it is possible. The pointless suggestions have to be removed and other reasonable ones should be added, depending on the current code context. This can be automated using the language grammar.

7.2 Tip texts

The next IntelliSense feature is displaying the tip texts about the symbol (token) under the mouse cursor. The expression under the cursor is parsed, processed and the list of declaration nodes is populated (7.1.2). The list should contain one item only; otherwise the first one might be used. The

declaration node in the list provides the description property. This can contain a symbol description and/or additional information gathered from the syntactic and semantic analysis, like possible variable or constant value, function parameters or source file name. This text description is returned and displayed then by the text editor environment.

7.3 Method info

Information about a method consists of a method text description and a list of possible method signatures. When the user is typing parameters of a function call, the language integration has to recognize that it is a method under the cursor (e.g. cursor is after the left parenthesis and the identifier token is before the left parenthesis). Then it has to found the method information.

Like the tip texts above, the corresponding declaration node must be found. It corresponds to the expression which describes the function call. Its description and list of its possible parameters are returned.

7.4 Go to declaration

The last IntelliSense feature enables the user to quickly jump to the declaration of the symbol under the cursor (e.g. by pressing F12 in Visual Studio). The expression under the cursor is processed and the declaration node is resolved; the declaration node location span and the parent code scope file name are returned. The text editor environment automatically opens the file name and jumps onto the specified location.

8 Other solutions

The IntelliSense is becoming a common part of modern source code editors. Most of well-known programming languages have its own support in development environments - in default or in the form of third party plug-in.

There are working commercial language integrations even for dynamic languages, like PHP or JavaScript. Typically they ignore the language dynamicity or some of their features. For user-defined languages there are also low-level solutions, like the IntelliSense tree designed in this work.

Next to the designed IntelliSense tree, there are other possible solutions how to implement IntelliSense functionalities. However they are basically not focused on the dynamic languages. Therefore non-static information is not resolved and typically no semantic analysis is needed.

8.1 Non-dictionary predictive text

The methods of text predicting without knowledge of the language are very easy to implement. It consists of the lexicalization module and self-learning database of symbols only. The source texts are lexicalized using some general rules and found tokens are stored within the database. The database typically doesn't contain any more information about the symbols.

Text oriented methods without a predefined dictionary work for any source code or generally for any text. These techniques [8][10][11] do not know the specific language - they learn from the already existing source texts. Only words (tokens) from existing source files are used. While writing the text the editor just offers the list of known words which contain the word under the cursor. The list of available words is taken from the database of symbols. A set of characters or regular expressions have to be defined only, just to let the editor know what the simple word could be.

These simple methods are very fast; they should help and speed up the process of programming. Also it is language independent and it is easy to implement. But when using this solution the programmer has no more information about the source code and particular declarations. The editor lacks any intelligence so it offers also meaningless words combinations.

8.2 Static language

In the IntelliSense implementation of a dynamic language, all the dynamic features may be ignored completely. The source codes are lexicalized and parsed only. Following semantic analysis is not needed. Only the abstract syntax tree, as a product of parsing, is used directly as an in-memory database of symbols.

This method is still simple, and it is practically usable in most cases (e.g. VS.Php, see 8.3). It works like the IntelliSense tree implementation without the additional semantic analysis, which uses expression patterns (5.2.2) and analyzer objects (5.1.3). Then static and explicit declarations are collected only. Even the symbols description can be obtained from the user comments or XML doc file. The particular IntelliSense features must be implemented, such as gathering the list of right symbols available at the specific code scope.

Optionally the semantic analysis or CFG are used to determine possible errors or code warnings (6.1.4), like unused variable declaration or unreachable code detection.

8.3 VS.Php

VS.Php [4] is the commercial PHP language integration for the Visual Studio 2005 and 2008. At present it offers the IntelliSense functionalities and syntax highlighting for PHP, HTML, JavaScript and CSS. Also the integration is able to use third party software for debugging the PHP source code inside the Visual Studio environment.

It is a complete commercial solution for developers. However, this product does not handle the dynamic operations at all. It ignores object dynamic members and also it does not handle indirect calls and indirect variable usages. The developer cannot easily implicitly declare new variable, because anything the developer types is changed into something already declared. Then in fact it changes the PHP into a simple static language.

Finally the file inclusions are ignored completely. The language integration just offers everything what is explicitly declared in all the source files in the whole project folder. Then even not accessible symbols are offered by the IntelliSense. For large applications it can be annoying and sometimes unusable.

8.4 Microsoft DLR

Microsoft DLR [5] is an open source project, which is still under development. It enables compilation of user-defined dynamic language (or also static language). As a result it provides runtime environment and it allows interoperability between .NET and dynamic languages built on DLR. The objects defined in any of these languages can be used from any .NET based runtime.

Basically all the implementation leans on building the DLR expression tree from the source code. It is a syntax tree, which describes the source code and which is specially designed for dynamic languages.

In the future once the building of that expression will be managed it will also automatically serve simple language integration solution for Visual Studio. It will support syntax highlighting, debugging and probably IntelliSense functionalities too.

8.5 JavaScript IntelliSense

Microsoft Visual Studio has an integrated support for JavaScript dynamic language [6] which tries to resolve the types of variables and then manage their possible object members. The integration supports IntelliSense features and offers even the dynamic object members. All the built-in JavaScript functions and other constructs are managed and the IntelliSense is able to work with them. Also all the current HTML entities are included in complete DOM model.

However JavaScript language is simpler than other dynamic languages, especially it misses an include statement, so the single source code file is analyzed only. Language properly does not support any extensions, which makes the language integration even easier to implement.

9 Conclusion and future work

This thesis describes methods used for gathering information from the source code, so they can be presented in a well-arranged way. Originally it was targeted on the PHP dynamic language, but the data structures are designed to support other static and dynamic languages and to integrate them.

As a part of the work, the language integration for Microsoft Visual Studio was implemented. The integration, where the presented techniques are used, is called Phalanger IntelliSense - it is the significant extension of the previous Phalanger language integration. It processes the PHP language source code together with the .NET static assemblies to suggest available symbols, in order to support abilities of the Phalanger PHP compiler. The text prediction and source code analysis works well even for non static expressions without a need of script code execution.

Future enhancements of the Phalanger IntelliSense should recognize more well known expressions. Also the language specific processes, such as analyzing the function or class declarations, could be programmed in a declarative way, so it would be easier to integrate more languages quickly. As well as the language built-in functions and keywords - the formal language grammar, consisting of lexical and syntactical rules, can be used directly.

10 References

[1] Visual Studio SDK: Language Services, http://msdn.microsoft.com/en-us/library/bb165099(VS.80).aspx

[2] IronPython project, http://www.codeplex.com/IronPython

[3] Phalanger compiler, http://php-compiler.net/

[4] jcx software: VS.Php, http://www.jcxsoftware.com/vs.php

[5] Microsoft DLR, http://www.codeplex.com/dlr

[6] VisualStudio JScript IntelliSense, http://msdn.microsoft.com/en-us/library/bb385682.aspx

[7] IntelliSense, http://en.wikipedia.org/wiki/IntelliSense

[8] Predictive text, http://en.wikipedia.org/wiki/Predictive_text

[9] Tobias Lindahl, Konstantinos Sagonas: Practical Type Inference Based on Success Typings, Proceedings of PPDP, ACM, Venice 2006

[10] I. Scott MacKenzie: Keystrokes per Character as a Characteristic of Text Entry Techniques. Proceedings of MobileHCI 2002

[11] O'Riordan et. al.: Investigating Text Input Methods for Mobile Phones. J. Computer Sci, I (2):189-199, 2005.

[12] Visual Studio SDK, http://msdn.microsoft.com/en-us/library/bb166441(VS.80).aspx

[13] Phalanger project, source codes, http://www.codeplex.com/Phalanger

[14] GPLEX, http://plas.fit.qut.edu.au/gplex/

[15] GPPG, http://plas.fit.qut.edu.au/gppg/

[16] Joel Jones, "Abstract Syntax Tree Implementation Idioms", Pattern Languages of Programs 2003, Illinois

[17] Joseph Poole: A Method to Determine a Basis Set of Paths to Perform Program Testing, NIST 1991

[18] Abonyi A., Balas D., Beno M., Misek J. and Zavoral F.: Phalanger Improvements, Technical Report, Department of Software Engineering, Charles University in Prague, 2009

[19] ISO/IEC 14977:1996(E) Information technology - Syntactic metalanguage - Extended BNF

Appendix A. CD content

·**Documents**

·**IntelliSense implementation of a dynamic language.pdf**

·Phalanger

·Developer1-1.doc - Phalanger developers documentation.

·User.doc - Phalanger documentation, user's guide.

· **Installation**

· **Phalanger.msi** - Phalanger 2.0, php compiler installation.

· **VSIIP.msi** - Visual Studio integration installation.

· **VSIIP.txt** - Visual Studio integration installation instructions.

· **Source codes**

· **Phalanger** - Phalanger project source codes.

· **doc** - Phalanger generated documentation.

· **Source/Orcas** - Language Integration source codes.

· **Source/Orcas/doc** - Language Integration generated documentation.

Appendix B. Class diagrams

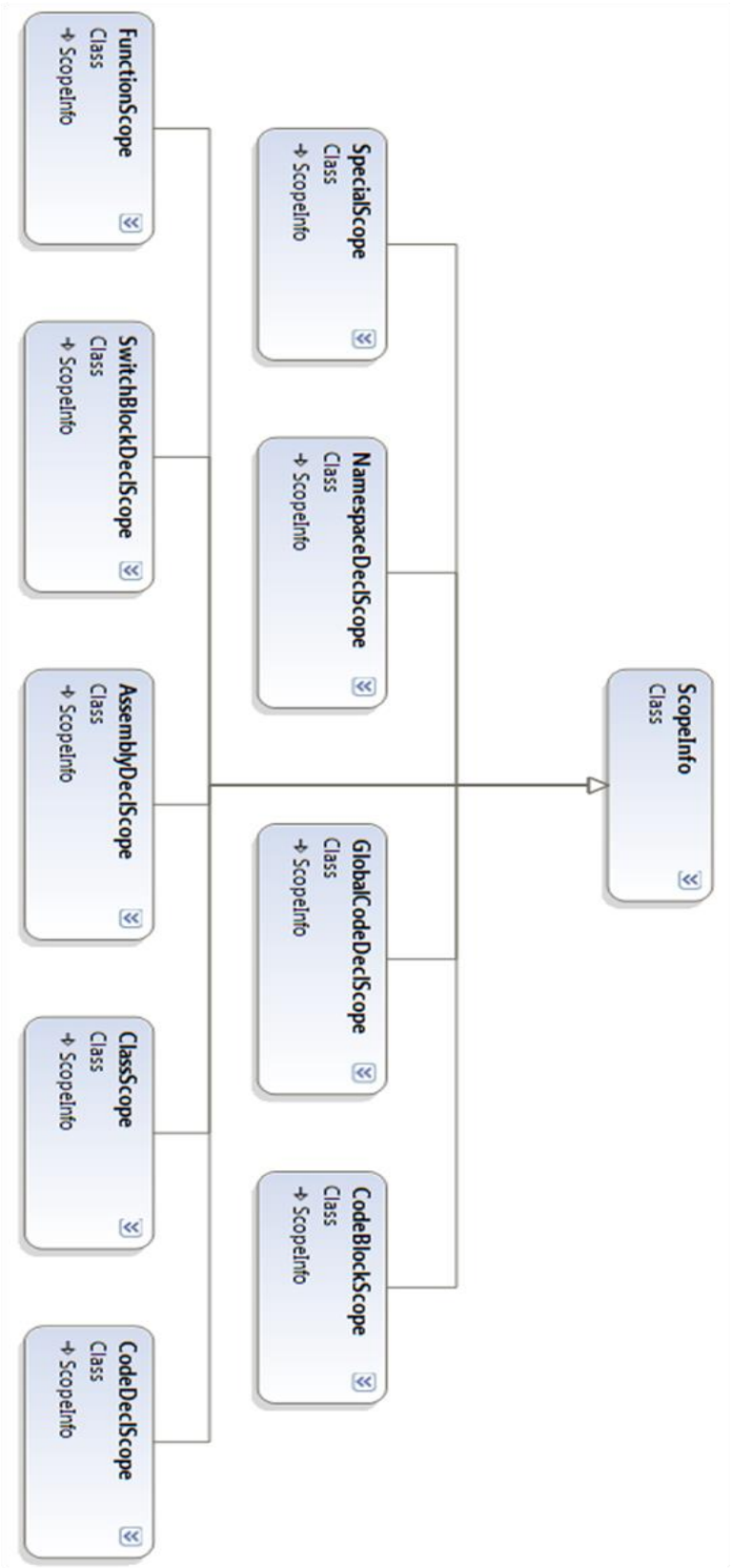


Figure 13 Scope nodes implementation diagram

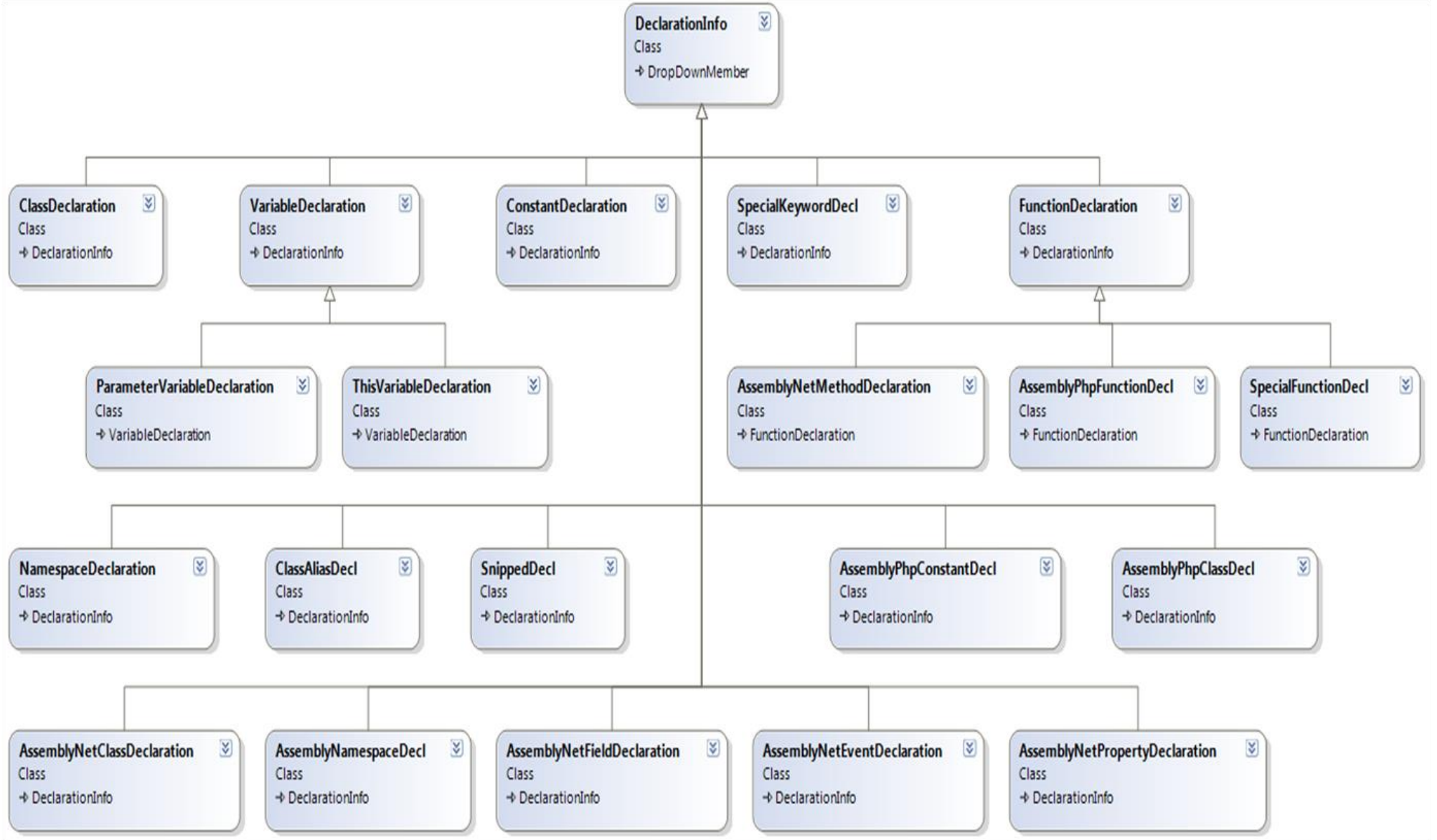


Figure 14 Declaration nodes implementation diagram