Comenius University, Bratislava

Faculty of Mathematics, Physics and Informatics

Integration of Text Editor with Code-Analysing Tool

Diploma Thesis

Bc. Ľubomír Žák, 2014

Comenius University, Bratislava

Faculty of Mathematics, Physics and Informatics

Integration of Text Editor with Code-Analysing Tool

Diploma Thesis

Study program: Informatics

Field of Study: 2508 Informatics

Department: Department of Computer Science

Advisor: RNDr. Tomáš Kulich, PhD.

Bratislava, 2014

Bc. Ľubomír Žák

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Ľubomír Žák
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	9.2.1. informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický

Názov: Integrácia textového editora s nástrojom analyzujúcim kód

Cieľ: Nadizajnovať abstraktnú vrstvu medzi textovým editorom a nástrojom, ktorý automaticky analyzuje kód. Implementovať prototyp tohto prístupu pre niektorý dynamický jazyk a niektorý editor.

Vedúci:	RNDr. Tomáš Kulich, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	doc. RNDr. Daniel Olejár, PhD.
Dátum zadania:	21.11.2012

Dátum schválenia: 28.11.2012

prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

....

študent vedúci práce

I hereby declare I wrote this thesis by myself, only with the help of referenced literature, under the careful supervision of my thesis supervisor.

. . . .

I would like to thank my supervisor RNDr. Tomáš Kulich, PhD. for his great help, advices and supervising.

Abstract

Integrating supportive extensions to various development environments can be a very complicated task. Every IDE has different API providing different functionality for user willing to spend some time on improving given product. Main goal of the thesis was to design and implement solution which would be as IDE-independent as possible while providing better communication and extensibility for other programming languages. Our idea is based on separating computation/functionality part completely from IDE itself. As a demonstration we programmed an implementation of designed interface for Python language and Vim text editor. In the second part of the thesis we were focusing on providing systematic and robust solution for generating abstract syntax tree from given Python source code.

KEYWORDS: Plugin, IDE, Dynamic language, Python, Parser, Lexer, Code analysis

Abstrakt

Integrácia rozšírení do rôznych vývojových prostredí môže byť veľmi komplikovaná úloha. Každé prostredie má rôznu API, ktorá poskytuje rozdielnu funkcionalitu pre používateľa ochotného venovať nejaký čas vylepšovaniu daného produktu. Hlavným cieľom diplomovej práce bolo vytvoriť dizajn a následne implementáciu riešenia, ktoré by bolo maximálne nezávislé od vývojového prostredia a poskytovalo by jednoduchšiu komunikáciu a rozšíriteľnosť pre ďalšie programovacie jazyky. Náš prístup je založený na separácii výpočtov/funkcionality samotnej od vývojového prostredia. Pre demonštráciu nášho prístupu sme naprogramovali riešenie pre jazyk Python a textový editor Vim. V druhej časti diplomovej práce sa zaoberáme tým, ako systematicky a robustne generovať abstraktný syntaktický strom zo zdrojového kódu jazyka Python.

KĽÚČOVÉ SLOVÁ: Rozšírenia, IDE, Dynamický jazyk, Python, Parser, Lexer, Analýza kódu

List of Figures

3.1 Simple abstract tree example. 28

6.1 Results from validation with errors shown. 53
6.2 Error descriptions example. 54
6.3 Auto complete example. 54
6.4 Get all function and class definitions example. 55

Contents

Intro **1**

1 Plugin Creation **4**

1.1 Vim 4

1.2 Emacs 7

1.3 Eclipse 10

1.4 Sublime 11

1.5 Motivation and proposed solution 13

1.5.1 Daemon interface description 14

2 Basics **21**

2.1 Dynamic and dynamically typed languages 21

2.2 Differences 22

2.3 Choosing the language 24

2.4 Why Python ? 25

2.5 About Python 25

3 Parsing **27**

3.1 AST - Abstract syntax tree 27

3.2 Using lexer and parser to produce AST 28

3.2.1 Detection of the invalid parts 29

3.2.2 Lexer 29

3.2.3 Parser 30

3.2.4 AST module 32

3.3 Summarization 32

3.4 Comparison with other solutions 33

3.4.1 PyDev 34

3.4.2 PyLint, PyChecker, PyFlakes 34

3.4.3 Our solution superiority 34

4 Daemon service **36**

4.1 Our daemon implementation . . . 36
4.1.1 Asyncore . . . 37
4.1.2 Json . . . 38
4.1.3 Threading . . . 39
4.1.4 Handling requests . . . 40
4.2 Daemon for other IDE/programming language . . . 43

5 Plugin **44**
5.1 IDE - vim . . . 44
5.2 Vim life cycle . . . 45
5.3 Classes . . . 46
5.4 Functions . . . 47
5.5 Vimrc settings . . . 50
5.6 Client for other IDE/programming language . . . 51

6 Results **52**
6.1 Screenshots . . . 53
6.2 Universality of daemon . . . 55

Conclusion **56**

Bibliography **57**

Appendix **58**

Intro

Dynamically typed languages are recently rising on popularity. And there is a very good reason for that. Once any programmer tries any of the dynamic languages after spending a while programming statically (in languages such as Java or C#) he can truly "feel" the power given to him. No variable declarations, assigning almost anything to anything else, shorter code written and more. The compile-time control used by statically typed languages can indeed slow people down in certain situations. But it can save the programmer from hours of debugging since there is a lot of errors being caught during it.

Summing it up, dynamic languages are cool but that compile-time protection is something hard to give up on once you are used to it. This raises an interesting question: Could we provide the protection of static languages to certain dynamic language ? If it is possible, how "good" can the protection be ? How many errors are we able to detect before run-time given only source code of dynamic language ? And that is our goal: to try to answer these questions and implement a solution for given language and text editor.

Whole project consists of three main parts: Parsing part, Inference part and Plugin for given text editor. Main goal of this thesis was to design and then implement an abstract layer between the chosen text editor and a code-analyzing tool. The tool itself should be independent enough so that its integration with another editor in the future would be as simple as possible. Additionally, the tool should be easily extensible for other programming languages using the same API. We decided to divide Plugin part into client and service parts which helps with meeting the above mentioned criteria. Parsing part provides reliable way of generating output usable by inferencer even for syntactically invalid code. The code-analyzing tool itself (the inference part) was implemented by my colleague, Bc. Dominik Kapisinsky in his diploma thesis *Type-Awareness in Dynamic Languages* [1].

1

Plugin Creation

Creating plugin to certain text editor can be very complicated issue. One needs to learn the principles of creating such a plugin including how to display something in an IDE, how to get some triggers from it and more. Additionally, if plugin is actually supposed to do something useful it needs to have some features implemented in it.

In the next sections we will provide brief list of representatives of certain IDE types with descriptions of what simple plugin creation in them look like. Moreover, we will name an example of plugin related to our further work, e.g. related to Python language (see section 2 for further details on why Python language was selected) and we will look on how it is linked with the editor. At the end of the chapter we will describe our solution idea which is much less IDE-dependent and more universal.

1.1 Vim

Vim is an open source text editor used mostly on Unix-like systems. Vim is a modal editor, meaning that it behaves differently, depending on which mode user is currently in. Possible modes are [2]:

- Normal mode - used for entering editor commands
- Visual mode - like Normal mode, but movement commands extend a highlighted

area

- Select mode - typing a printable character deletes the selection and starts insert mode
- Insert mode - text inserted is written into buffer
- Command line mode - used for executing commands by command line located at the bottom of vim editor
- Ex mode - like Command line mode, but after entering command user remains in Ex mode

Generally the most used modes are Normal, Visual, Insert and Command line. Any customization to the vim behaviour is done either by editing configuration files, usually called *.vimrc* and located at *$HOME/.vimrc* (where $HOME refers to home directory location) or by adding customized extensions called plugins. Each of the plugins loaded to vim have *.vim* extension and is (or at least some part of it) written in **Vim script** language.

Configuration file can contain useful settings like color scheme setting, enabling of the line numbers, setting the tabulator width when it is pressed and more. Additionally, any plugin written later can fetch custom setting from this file, enabling the user to customize some of its own settings. Last thing we will mention about *vimrc* configuration files is remapping key bindings. For example, default option to call auto completion in vim is usually $< C - X >< C - O >$, which stands for pressing Ctrl + X followed by Ctrl + O. To remap it, for example to pressing "." key, user can add following lines:

```
inoremap <expr> . AutoCompletion()

func AutoCompletion()
return ".\<C-X>\<C-O>"
endfunc
```

The *inoremap* command remaps "." to *AutoCompletion()* function call, which simply acts as if the $< C - X >< C - O >$ combination was pressed. More complicated

things are usually stored separately in vim script plugin files, tough.

Vim script

Vim script is a scripting language built into the Vim editor. Vim script supports well basic principles of modern programming languages, such as control flow support or object-oriented programming support. Vim script also provides a lot of functionality via built in commands and functions. Additionally vim script supports adding pieces of codes written in other interpreted languages (including Perl, Python, Lua or Ruby). For example following construct is perfectly fine in vim script plugin with python enabled:

```
function! CustomFunction()
%some vim script code here

python << PYTHONEND

#some python code here

PYTHONEND
endfunction
```

Previous example defines vim script function called *CustomFunction()*. This function includes an arbitrary long piece of python code starting at *python* $<< PYTHONEND$ line and ending at $PYTHONEND$ line.

This provides excellent support for not giving up on users favourite language. If user for example wants to script plugins in python, he needs to study and write only minimum of vim script code and do most of the functionality in python sections of the plugin code.

Last thing we will mention under vim script is robust support of auto triggered calls called **auto commands**. User can easily specify what should be done when he opens vim, moves cursor, saves file, leaves buffer and many more.

Commonly used plugins

Pylint is being executed every time the file is saved. We call this approach a "standard" way of triggering - some static analysis is being done on whole text once the trigger has been activated.

Another commonly used solution, **Pyflakes** uses more triggers: BufEnter/Leave (triggered once buffer is being open/closed), InsertEnter/Leave (switching between modes), CursorHold (user did not move cursor for given amount of time) and CursorMoved (triggered every time user moved the cursor around. Summing it up, this is again the static approach - trigger causes update and return values are displayed.

Conclusion

Plugins written for vim text editor are linked with the vim usually in a regular static way: plugin waits for some triggers provided by vim script and calls customizable actions accordingly. Then gathers results from any form of computation provided by either *.vim* plugin file itself or some linked source file and displays them to the user. Whole computation process is being called as some kind of a method returning some values for the plugin. Communication between computation and the editor itself is usually done via passing copy of current buffer or the name of file which is being modified.

1.2 Emacs

GNU Emacs is an extensible, customizable text editor. For extending its functionalities it uses Emacs modes written in Emacs Lisp to be able to support various languages and texts such as C codes, python codes or more generic stuff like emails or plain texts.

Lisp

Lisp is functional programming language, which was widely used mostly for artificial intelligence research. The name LISP is derived from LISt processing, which implies that the original purpose of this programming language was working with lists and various other data structures derived from it. Lisp uses fully-parenthesized prefix notation, meaning that every function calls and expressions must be surrounded by parentheses. For example while in most "modern" programming languages it is allowed to write

```
1 + 2 * 3
```

Same construction in LISP looks like this:

```
(+ 1 (* 2 3))
```

Arguments are listed after operand itself. The only other functionality we will list here is list creating in Lisp. List is Lisp object containing zero or more other Lisp objects:

```
(x y (z))
```

Overview

Emacs is similar to vim in many ways. Mostly they provide a little unusual (compared to "modern" IDEs) navigation between and on files. It is not (at least by default) graphical interface, therefore previously mentioned navigation around is mostly done via keyboard shortcuts. Knowing these shortcuts is vital for using Emacs effectively.

Emacs provides, though, way to remap default shortcuts to basically any other key combination via simple call of *(global-set-key* command:

```
(global-set-key keysequence command)
```

where *keysequence* is sequence of keys needed to be pressed to invoke given command (usually a name of a function). Key sequence is usually mapped to key combinations

containing either CONTROL (ctrl) or META (alt) keys (or both) followed by another key. For example placing following command in *.emacs* configuration file would cause emacs to call help whenever Ctrl + ? is pressed:

```
(global-set-key * \C-?" 'help-command)
```

This way we can allow user to remap auto completion method to whatever he wants.

Emacs Lisp contains numerous macros with built-in functions which gives user an easy way how to do . Grabbing whole buffer is pretty straightforward via *buffer-string* command. For getting characters or even words under current cursor position, *thing-at-point* function is used.

Another example of often proposed functionality is providing keyword or auto completion list to the user. This is solved via *display-completion-list*, usually preceded by opening temporary buffer in which the options will be shown (*with-output-to-temp-buffer*):

```
(with-output-to-temp-buffer "*Completions*"
    (display-completion-list
        (all-completions (buffer-string) my-alist)
            (buffer-string)))
```

where *my-alist* contains completion options and *buffer-string* in this case represents variable, on which the completion was called.

Conclusion and commonly used plugins

Summing it up, Emacs Lisp provides a lot of functionality by using built-in functions and macros. It does not support auto commands in a vim-like way. Instead, it does provide different way to achieve similar results by defining custom modes for open buffers.

Generally code analyse tools for Emacs are using either *FlyMake* or its newer alternative *Flycheck*. The best description of these modes is by the creator himself: "Flymake is implemented as an Emacs minor mode. It runs the syntax check tool in

the background, passing it a temporary copy of the current buffer, and parses the output for known error/warning message patterns. Flymake then highlights erroneous lines (that is, lines for which at least one error or warning has been reported), and displays an overall buffer status in the mode line." [5] In other words, both these modes provide a simple "interface" which expects standard output from various checking tools to be in specific format (doing regex matching for this purpose). To integrate tool with these modes one needs to provide results from calculations in it.

1.3 Eclipse

Eclipse is an open source developing tool which is mostly known as an IDE for *Java* programming. On the top of that Eclipse provides an extensible way of adding various plugins for support of other programming languages, such as C, C++, Python, PHP and more.

Writing a plugin for the Eclipse IDE is a little bit more complicated. Eclipse by itself is just some kind of "motherboard". All additional extensions to Eclipse are done through plugins, and plugins integrate with each other through extensions on extension points. Eclipse plugins typically provide extensions to the platform that support some additional capability or semantics. Whole plugin code needs to be written in *Java* language. Unlike *Vim script* and *Emacs Lisp* we will not provide description of *Java* language here since it is not needed (*Java* is much more popular and used widely all over the world).

First thing while creating Eclipse plugin is adjusting the manifest file, which is something like a header of the whole project. In manifest there are numerous things specified including:

- **Overview** - describes what is plugin supposed to do plus contains general information such as ID, version, name of the plugin and so on
- **Dependencies** - other plugins on which newly created one depends
- **Extensions and Extension points** - main mechanism for selecting what is

plugin supposed to do. By specifying extension points we define what we expect from Eclipse and what we want to offer.

- **Build** - specifying build information

Once extension points have been chosen, user can import these packages and use provided "triggers" by extending given classes.

Commonly used plugins

PyDev is an example and most likely the most used Python development plugin for Eclipse platform. Although it is a very complex solution providing massive functionality support, it is created in a "traditional" way - there is no way that one could use PyDev features in other IDEs for one simple reason: It is almost entirely written in Java language and is dependant on very specific interfaces and extension points provided only by Eclipse itself. We do not want to judge PyDev qualities by this, though, since PyDev never had the ambition to be multi-IDE development tool and was specifically designed for Eclipse.

Conclusion

Eclipse provide huge support for plugin creation since the whole Eclipse idea is based on linking different plugins together. By providing excessive amount of extension points and triggers and by taking advantages of Java programming language, creating plugin doing basically anything (in terms of functional capabilities) is possible.

1.4 Sublime

Sublime Text is modern text editor, which is based mainly on simple and intuitive graphical interface and its adaptability.

Writing a plugin for Sublime is very user friendly, since unlike vim or Emacs it does

not require some specific language learning: whole plugin can be written in Python language. Sublime comes with functionality that generates a skeleton of Python code needed to write a simple plugin which includes importing of two necessary modules: *sublime* and *sublime_plugin*.

The API provided contains all of the needed methods. For example grabbing text in the current view can be done by simple *self.view* call. Tracking view IDs is handled by *self.view.id()*. This provides simple way of how to not mix up various requests if processing is being done by threads for example.

What is the most noticeable thing about sublime is its very nice graphical interface with very easy theme or color customizations.

Unfortunately, Sublime is not free IDE and license must be bought: "Upgrade Policy A license is valid for Sublime Text 3, and includes all point updates, as well as access to prior versions (e.g., Sublime Text 2). Future major versions, such as Sublime Text 4, will be a paid upgrade." [6]

Conclusion and commonly used plugins

Sublime enables extending its functionality by writing plugins in high level language - Python. Having that granted writing custom plugin is by far the easiest from IDEs listed here. Not only that because high level programming language is used (because as mentioned above, Eclipse plugin creation is done in high level language - Java), but that API provided is much easier to understand.

Pylint was also modified to work with Sublime. Integration of the plugin is pretty simple and is provided by afore mentioned pair of sublime support modules. Whole process is again a "static" one.

1.5 Motivation and proposed solution

We have provided a brief list of different types of IDEs with descriptions. For each environment we also listed several basic commands to demonstrate how it is possible to create a simple plugin/mode/extension in each of them. Moreover, we have listed several commonly used solutions in each of the IDEs for working with Python. We can sum up problems of plugin creating into two different non-overlapping sections:

1. Complexity of the plugin needed for tool to be able to do its job
2. Communication between the plugin and the tool

After setting up the criteria, we can divide existing solutions into two different categories depending on how do they meet each of them:

- **IDE independent** - easier integration but generally bad tool-plugin communication and with fewer functionality (Pyflakes or Pylint in Emacs)
- **IDE dependent** - usually more complex but with more complicated (often too specific) integration (Pydev, Pyflakes for Vim, Anaconda for Sublime)

Both afore mentioned categories have their pros and cons. And that is why our proposed solution is indeed a little different. We wanted to keep up the positive things from both of the approaches. Therefore we have decided to create universal service running on given computer somewhere in the background listening on certain port. Once user starts any of the IDEs, respective plugin is only connecting to the service which is doing all of the computations. Plugin is only responsible for grabbing the data and providing feedbacks to the user from what has been received by the service.

How is it different/better than existing solutions ?

- **Easier IDE integration** - plugin is responsible only for sending requests to the service and then receiving the output. Although since some kind of socket-client communication is needed, integration can be easier in some other existing solutions, which statically call given tool for the answer. Our approach has one big advantage in other area of communication though: there is no need to

extract big archives at specific locations or do massive step by step installations per each of the IDEs. One instance of service is running for any of the IDEs. Plugin is there only to connect to it which makes its role much more specific and equivalent at the same time in all of the IDEs.

- **Better communication** - there is no need to parse enormous amount of command line flags and parameters to distinct different requests. Moreover, it is not necessary to somehow parse standard output from the tool to decode it and then provide the feedback. Since service is not accessing any files in any way, it does not depend on root access or any other kind of access. Service just listens and responds to specific question with the specific answer.
- **Extensibility** - it is very easy to extend the daemon such that it would respond to other types of requests (for example possibly other programming languages, another IDEs) without the need of installing additional tools for different languages or different IDEs. One Service can cover all of the programming languages and IDEs.
- **Dynamism** - that opens huge amount of possibilities like that there is no need to send whole file all the time (speeding up the process greatly), computing useful information while idle (no incoming requests for certain amount of time), and generally avoiding doing certain kinds of work all over again on each request (iterating trough imports for example).

The whole idea and its realization for one language and one IDE (but still easily extensible for more) is discussed in further details in chapters 4 and 5.

1.5.1 Daemon interface description

Since we wanted our daemon to be universal and as IDE independent as possible, we designed simple interface for the daemon via which (potentially any) plugin can communicate with it. Following list summarizes requests accepted by the daemon and provides description of the request, input parameters, description of which variables given trigger modifies, what is an output (what is sent back to the client) and whether

this method call is language dependent or not.

Each of the requests is described informally since for further details one could check daemon code documentation. The key thing mentioned above can also be seen here: it would be very simple to extend the daemon in a way that it would listen to other type of requests (possibly for other languages and other IDEs).

Whole interface description is provided in JSON (JavaScript Object Notation)-like syntax since our implementation also uses this approach. Daemon requests are simple maps with key : value pairs (see below).

All requests need to contain given three properties:

1. *type* - specifies which request should be triggered
2. *pid* - process id which helps to distinct between clients
3. *tab_page_number* - additional number used if there are more than one of the buffers open at the same time by single process

Note: Parameters noted in square brackets - [ ] - are optional. All data read by daemon are expected to be in bytes format (string transformed to plain bytes).

sendCurrentWorkingDirectory

Description: receives directory and tries to find root file (specified by parameter) to get all function and class definitions in given project.

Input:

```
{
    "type" : "sendCurrentWorkignDirectory",
    "pid" : int,
    "tab_page_number" : int,
    "current_working_dir" : string,
    ["root_filename" : string, "language" : string]
}
```

Modified variables: updated buffer holding correct list of function and class definitions for given project

Output: -

Language dependent: Yes. Calculating all class and function definitions would need to be slightly extended since this call uses specific parser. By adding parser for another language, though, daemon could switch between programming languages accordingly to additional parameter *language*.

sendWholeFile

Description: client sent whole buffer to the daemon.

Input:

```
{
    "type" : "sendWholeFile",
    "pid" : int,
    "tab_page_number" : int,
    "whole_file" : string
}
```

Modified variables: updated buffer at given ID with data retrieved

Output: -

Language dependent: No. Request helps daemon to keep up buffers data "fresh". For other programming languages the exactly same method can be used since uniqueness of the IDs sent is handled by the client, not by the daemon.

updateLine

Description: updates three lines in given buffer: one being modified and both the one line above and the following one. Number of lines input is used to determine whether deletion of single line or insert of new line was triggered.

Input:

```
{
    "type" : "updateLine",
    "pid" : int,
    "tab_page_number" : int,
    "number_of_lines" : int,
    "line_number" : int,
```

```
        "line_text" : string,
        "previousline_text" : string,
        "nextline_text" : string
    }
```

Modified variables: updated buffer at given ID with retrieved data

Output: -

Language dependent: No. Same as **sendWholeFile**.

parseAndValidate

Description: Most important request: Daemon parses and validates whole buffer (found by the incoming pid and tab number) and responds with list of errors and warnings.

Input:

```
    {
        "type" : "parseAndValidate",
        "pid" : int,
        "tab_page_number" : int,
        "number_of_lines" : int,
        "iterations" : int,
        ["language" : string]
    }
```

where *iterations* specifies how many times the whole validation process should be called.

Modified variables: -

Output: Responds with list of errors and warnings. Each of the problems or warnings is another list containing two elements: line number on which the error/warning occurred and the description of the error:

```
    {
        "problems" : array of arrays,
        "warnings" : array of arrays
    }
```

Language dependent: Yes. Daemon calls specific function on given buffer which is supposed to do whole validation process and respond afterwards. If such a feature was desired by the daemon for other programming languages, whole process needs to be also written for other programming language and the daemon would select between the languages by additional parameter.

getAutocomplete

Description: Daemon runs validation with different goal: do not check whether there are some errors but give me list of attributes of given variable or constant on given line. Afterwards daemon responds with options for auto completion.
Input:

```
{
    "type" : "getAutocomplete",
    "pid" : int,
    "tab_page_number" : int,
    "variable" : string
    "line_number" : int
}
```

where *variable* is text until current cursor position and *line_number* is current line number on which auto completion request was called.
Modified variables: -
Output: Responds with list for auto completion sorted alphabetically.

```
{
"options" : array of arrays
}
```

where each of the arrays in outer array contains string at first index and optionally docstring (if it was present) at second index.
Language dependent: Yes. Same as **parseAndValidate** since it uses same mechanisms.

getAllDefinitions

Description: Responds with a list of all function/class definitions. Additionally recalculates definitions for given buffer to be able to provide correct data (for example user added new class and wants to use it somewhere later in the code).
Input:

```
{
    "type" : "getAllDefinitions",
    "pid" : int,
    "tab_page_number" : int,
    "current_directory" : string
    "filename" : string
}
```

Modified variables: -
Output: Responds with list of all function and class definitions in given project.

```
{
    "options" : array of dictionary objects containing
                string as keys and array as value
}
```

where each of keys is one of the options provided and array contains info whether it is class or function on first index and path to file in which given definition occurred on second index. *Language dependent:* No. Values stored for given ID were already calculated (see **sendCurrentWorkingDirectory** request).

For further details of our implementation of the daemon and plugin communicating with it see chapters 4 and 5.

Conclusion

Daemon service is expecting certain request types. Holding correct data in buffers is language independent since this mechanism is the same for all of the programming languages. On the other hand, parsing methods and methods looking for function

and class definitions are language dependent (at least slightly).

2

Basics

In this chapter we list and discuss some of the basic principles of dynamically typed languages. At the end of the chapter we present our decision on which dynamically typed language did we choose and why.

2.1 Dynamic and dynamically typed languages

Dynamic programming language is a term used broadly in computer science to describe a class of high-level programming languages that execute at run-time many things that other languages might perform during compilation, if at all.

"In recent years the importance of dynamic scripting languages - such as PHP, Python, Ruby and Javascript - has grown as they are used for an increasing amount of software development. Scripting languages provide high-level language features, a fast compile modify-test environment for rapid prototyping, strong integration with database and web development systems, and extensive standard libraries." [8] They also allow programs to be written more easily using high-level constructs such as comprehensions for queries and using generic code.

As scripting languages are used for more ambitious projects, software tools to support these languages become increasingly important. Most dynamic languages are also

dynamically typed.

2.2 Differences

In a statically typed language, every variable name is bound both

- to a type at compile time
- to an object

Once a variable with given name has been bound to a certain type (in other words it has been declared), it can be assigned only to objects of same type. There is no possibility how to assign integer to a string, for example. An attempt to assign such a values (or any other values together) results in type exception.

In a dynamically typed language, every variable name is (unless it is null or none) bound only to an object. Additionally, majority of the type checking is performed at run-time, not at compile-time. That is the biggest difference between dynamically typed languages and statically typed languages.

Short example in python:

```
someVariable = 9
someVariable = 'some string'
```

Short code above would result in use of uninitialized variable exception in statically typed languages. In other words, static compiler says "I do not know what it is, therefore I do not know what the type of that variable is." To avoid this error we could initialize variable like this:

```
int someVariable = 9
someVariable = 'some string'
```

In this case, type error would be raised. Compiler can not assign string to something that has been declared as an integer. On the other hand, this initialization is perfectly fine in dynamically typed languages such as python. Furthermore, the following is

again perfectly fine in python (but would not work in statically typed languages):

```
if True:
    someVariable = 9
else:
    someVariable = 'some string'
```

Again, static compiler has no such a variable initialised. If we actually initialised value before if block, it would not help since compiler would throw type error on else block. One would question this behaviour since the else block would never trigger. That is indeed true, but the static compiler can not allow us to do that. If it can not guarantee types on all variables it can not mark such a code valid, regardless of the fact that anyone can see that it actually is a type safe piece of code.

Going back to dynamically typed languages, names are bound to objects at execution time by means of assignment statements, and it is possible to bind a name to objects of different types during the execution of the program (as seen on above examples). However, this can often lead to numerous hidden errors where the programmer relies on the fact that some variable is for example int but in reality it contains string value. Still, we can say that this is also the major weakness of the statically typed languages: while protecting programmer against all of the type errors he is also protected in cases where it might not be needed.

In dynamically typed languages we will not encounter type errors. The more "freedom" comes with a price tag on it, though. Dynamically typed languages often require a lot of unit testing to cover up the static typing. Although these tests can detect a much wider range of errors, it is often hard or impractical to write full coverage test for a particular program.

Dynamically typed languages are often considered to be slower. It is because in such a language the compiler always has to keep "flags" describing the actual type of the value of the variable, and the program has to perform a data-dependent branch on that value each time it manipulates a variable. It also has to look up all methods and operators on it. The knock-on effect of this on branching and data locality is lethal to general purpose run time performance. That is why the dynamic language

JIT benchmarks emphasize near-C speed on small inner loops but steer clear of large data-structures and data manipulation problems.[9] Summing the above up:

- Every variable can be dynamically-typed: Need type checks
- Every statement can potentially throw exceptions due to type mismatch and so on: Need exception checks
- Every field and symbol can be added, deleted, and changed at runtime: Need access checks
- The type of every object and its class hierarchy can be changed at runtime: Need class hierarchy checks[10]

With modern dynamic language compilers there are several neat tricks they can do to specialize the hot paths you use in order to improve the overall performance.

To conclude the section, we could say that the dynamically typed languages can feel intuitively faster and also creating certain types of solutions might take much less time. The statically typed languages protect the programmer and guarantee that a particular subset of errors will never occur at the cost of more time spent on typing (declaring values, potentially more blocks in code etc.).

2.3 Choosing the language

Here is a brief list of dynamically typed languages we have considered as potential candidates for our thesis.

- Python
- PHP
- Perl
- Ruby
- Javascript

After consideration we have decided to implement our solution for **Python**.

2.4 Why Python ?

Python is becoming more and more popular between programmers for both its simplicity and very strong features. Lately it has been even used as a programming language for numerous beginner programming courses on world renowned universities, such as MIT, Cambridge or Oxford.

Also worldwide community around Python is being very dedicated and determined to keep it as smooth as possible, therefore Python as a language is very mature. Number of projects being developed in it rises, too. Site www.langpop.com even ranks Python as 6th most used programming language, falling behind only to C, Java, PHP, JavaScript and C++.

Another reason for choosing Python is that there are several existing plugins and solutions for Python working on similar topic, so we were able to compare our solution with already existing ones.

2.5 About Python

Python was released by its designer, Guido Van Rossum, in February 1991 as a reaction to interpreted language called ABC. Rossum wanted to keep some of its features but he also wanted to improve it via correcting some specific problems.

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax.[11] It directly provides numerous interfaces to useful system calls and libraries. Furthermore, Python is portable across all major hardware and software platforms.

Strong Python features overview

Let us summarize strongest python features:

- **Simplicity** - python syntax is very simple and minimalistic. It has very good readability and provides programmer an option to concentrate more on the ideas than on code itself.
- **Easy to learn** - as stated above, it is very popular language to start learning programming with.
- **High-Level** - programmer can concentrate only on high end programming. There is no need to care about garbage collecting or memory allocating.
- **Multi platform** - all python scripts can be executed on any machine and operating system, where Python interpreter is present. Interpreter can be installed on almost all known operating systems, including Windows, Linux, Macintosh, Solaris and more.
- **Object - oriented (and still simple)** - python supports object oriented programming (object creating and manipulating)
- **Interpreted** - python is interpreted language. That means it can be run without compilation - directly from the source code.
- **Extensible with C/C++** - in case some speed is required, certain part of the code can be written in C or C++ languages and then just included in main python script.

Conclusion

To sum up whole chapter, we decided to concentrate on a single dynamically typed language - Python. This decision was made due to its maturity, simplicity and due to the fact, that there are some existing solutions written for python on similar subject so we could compare our results with these.

3

Parsing

As I mentioned before in the introduction, the whole project consists of three main parts: Parsing, Inference and Plugin part. In this chapter we will provide description of our parsing method and what is it supposed to do and why.

The easiest way to describe what this part should do is pretty simple: We are given some kind of an input and we need to transform it into something else. In our specific case the input is any Python source code and an output should be something structured enough, with enough information for Inference part to do its job - AST - abstract syntax tree.

3.1 AST - Abstract syntax tree

An AST is a tree representation of source code written in basically any programming language. AST contains only semantically relevant information - irrelevant details are omitted. AST trees are usually represented as n-ary trees.

Figure 3.1 shows simple example of how would abstract syntax tree [7] look for the following code:

```
x * y + z
```

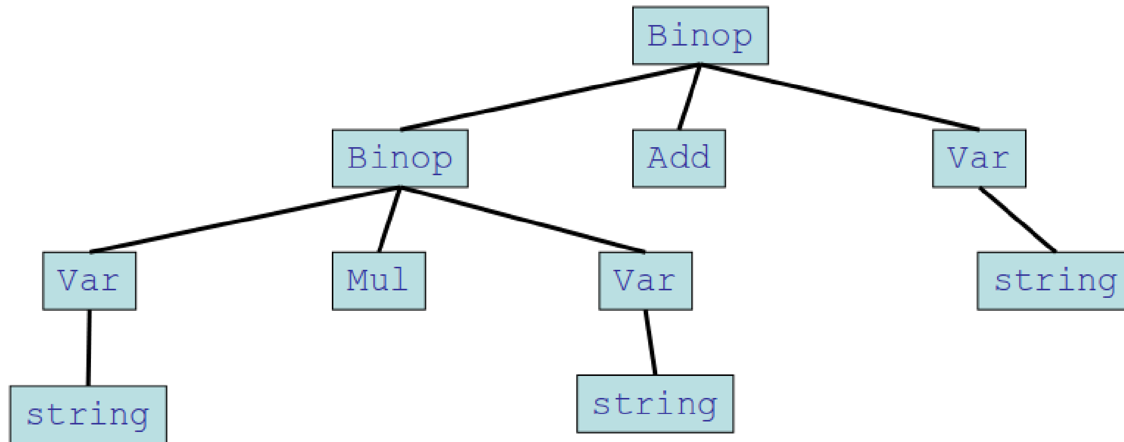

Figure 3.1: Simple abstract tree example.

3.2 Using lexer and parser to produce AST

We do have Python source code as an input. We need to produce AST representation of this code as an output. To achieve this we could intuitively use Python module called **ast**.

This module takes string (a python code) as an input and generates its AST representation. Seems like the problem is already solved, but using module has one big disadvantage: **ast** will not produce anything on an input, which is not syntactically valid. And by syntactically valid we mean that whole input must fit in Python grammar used by "regular" Python parser. If user makes any mistake, like writing *clsas* instead of *class*, ast module will throw an exception and there is no output provided.

At first it looked like the **ast** module could not be used for our solution at all. But what if we could detect and delete invalid parts of code (those which do not fit in python grammar) ? After that we could use **ast** module on the rest of the code to generate the correct tree while showing syntactical errors on the deleted lines.

3.2.1 Detection of the invalid parts

If we wanted to detect the invalid parts of the code we need to split the input code into blocks and then use the *ast* module to validate each of the blocks. Block finding recursion stops on validating single lines and gives us list of lines which were not correct.

To extract blocks from source code we could generally use one of the following two methods:

1. Look for keywords like def, class, for etc. in code and then use some heuristic to split it by values found
2. Generate lexer and parser similar to the original Python one which would be more robust

First technique is intuitively not very good. Keywords could be used in docstrings, in function or class names and writing heuristic method to cover all of the catches would be very difficult (maybe it could not be even possible).

That is why we decided to use the more elegant way - create our own lexer and parser tools.

For lexer and parser generation we used **PLY** or **Python-Lex-Yacc**. PLY is a pure-Python implementation of the popular compiler construction tools lex and yacc. PLY supports LALR(1) parsing as well as providing input validation, error reporting, and diagnostics. [12]

3.2.2 Lexer

Lexer is used for lexical analysis - process during which the input string is broke down into sequence of tokens. Tokens are represented via regular expressions. PLY can generate lexer from file specifying tokens written in Python.

```
@TOKEN(t_RBRACE)
def t_RBRACE(t):
```

```
r"\}"
t.lexer.paren_count -= 1
return t
```

This simple example explains whole idea how tokens written for PLY lexer look like. Annotation specifies that this is indeed a token and it's name is t_RBRACE. This token represents right brace symbol. Second line is used to count number of braces opened to detect syntax errors - opening braces via left brace and then not closing it.

By specifying these rules, or tokens, we generated lexer to work with Python. Notable thing here is the indentation errors detection. In Python language the indentation is used to specify code blocks instead of curly braces which are often used in many popular languages ($Java, C\#$).

Once lexer finds the indent token it remembers that the indentation is now open and expects dedent coming (sooner or later). This process is recursive - we need to close "brackets" (in this case indent and dedent) as many times as we have opened them. In other words, number of indents and dedents in source code must be the same, otherwise an indentation error is reported at the first line where matching did not work.

Once we have specified all of the token regular expressions we generated the lexer itself. Output from our lexer is a stream of tokens. And that is exactly what the next part, *parser*, needs as an input.

3.2.3 Parser

Parser is being used for syntactical analysis. What parser actually does is trying to match given tokens on the series of rules of given grammar.

In this paper we were using PLY parser, which is LALR(1) parser (LALR(1) stands for look-ahead left-to-right parser with one-token lookahead). To match tokens to the rules this parser can use not only the current token in input stream, but can also check which token is coming next, therefore can decided better which rule to use.

Writing the grammar for Python was actually pretty simple because input for PLY parser generator is not some domain specific language, but regular valid Python code. We took the original Python grammar and then we simplified it until it was good enough to fit for our purpose.

In the next few lines there is a brief sketch of our simplified version of grammar (to see the original Python grammar check [13]). Our grammar, compared to the original one, is heavily simplified, having less restrictive rules (Note: capitalized names are tokens - terminal symbols. Lower-cased names are non-terminal symbols):

```
file_input -> suite ENDMARKER
suite -> stmt | suite stmt
block -> block_keyword fragment COLON NEWLINE INDENT suite DEDENT
block -> block_keyword fragment COLON suite
block -> block_keyword fragment COLON NEWLINE
block_keyword -> * all Python keywords: if, elif, class, def etc. *
stmt -> fragment NEWLINE | fragment | NEWLINE | block
fragment -> * basically anything else *
```

Translated to human language: We do have some *file_ input* at the beginning. That is expanded to *suite* and *ENDMARKER*. Each *suite* is either only single *stmt* or *suite* followed by another *stmt*, where *stmt* represents single line of code. This line of code can be either *NEWLINE*, *block* or *fragment*. Blocks represent keywords followed by blocks (see *block* rules). The biggest difference between original python grammar and our robust implementation is hidden in *fragment*. Fragment is something able to cover up much more compared to the original grammar. Lets consider following Python input:

```
def some_invalid_return():
    return return return return #comment
```

This results in **invalid syntax** error in original Python parser. Our parser parses it just fine.

The reason behind whole robust parser idea is very simple: we do not want to delete

whole blocks of code if user makes a trivial mistake. We want to split code into smaller parts and then by using *ast* remove as small blocks/single lines of code as possible.

Result of parsing is therefore list of custom nodes or tokens clamped together.

3.2.4 AST module

With parser output in hands we used *ast* module to validate each of the nodes representing statement - single line of code using function which transforms given parser node back to string representation. After evaluating all of the statements we had list of deleted statements (line numbers). To get correct code (code with invalid lines removed), we transformed root node back to string while ignoring deleted statements.

Each line marked as deleted is in the end returned as **invalid syntax** error.

3.3 Summarization

Summing it up, whole process of creating AST from given source code consist of following steps:

1. Source code is given to lexer
2. Token stream from lexer is given to parser
3. Parser produces list of custom nodes (similar to ast). Each of the nodes contains enough information for recreating part of the code it represents.
4. Special function iterates trough list of nodes and uses ast module to validate each of the statements (lines of code). Syntactically invalid lines are marked.
5. Once recursion is completed, code is restored back to string while deleted nodes are omitted - this gives us syntactically correct subset of original code. If there was no syntactical errors at all, result of whole process is the same as the source code.

6. Call ast module on correct subset to retrieve AST representation of correct part of the code

At the end of the whole process we have the AST representation of correct subset of the original code plus list of deleted parts (lines).

3.4 Comparison with other solutions

In this subsection we will provide brief description of how the other solutions detect syntactically invalid parts in the code.

All of the solutions were tested on following two simple examples of invalid source codes:

Example 1 (indent error on line 3, syntactical errors on lines 4 and 8):

```
1 class A:
2     def function(self):
3     print('wrong indent')
4         return return return something horrible
5
6     def another_function(self):
7         print('correct indent')
8         return return again something bad
```

Example 2 (syntactical errors on lines 4 and 8):

```
1 class A:
2     def function(self):
3         print('correct indent')
4         return return return something horrible
5
6     def another_function(self):
7         print('correct indent')
8         return return again something bad
```

3.4.1 PyDev

PyDev, plugin to Eclipse (see 1.3), has internal parser written in Java. Parser uses Python grammar specified in an external file. PyDev tries to parse source code with standard python parser and if the code is not syntactically valid, returns only first occurrence of the error. This is because whole syntactic or indent error handling is done via catching exceptions in the parser.

In the first example PyDev detects indent error on 3rd line, but does not report anything on lines 4 and 8. In second example (with indent error corrected) syntax error on line 4 is detected but then again error on line 8 is omitted.

3.4.2 PyLint, PyChecker, PyFlakes

For all mentioned solutions the exactly same thing as for PyDev applies. They are again using regular Python parser and therefore they are not able to detect more than one syntactical/indentation error in a single file - they only detect the first one.

3.4.3 Our solution superiority

Our solution is able to detect indent error in first example, but does not show any additional errors. That is given by fact that lexer does not produce any output and therefore no further analysis is possible. The only thing returned is the line number where indentation was not valid.

On the other hand, on the second example, unlike other solutions, we are able to detect both syntactical errors.

As described in 3.2.1, our solution is better in handling invalid inputs thanks to our more robust parser. This key change in approach allows us to detect more syntactical errors at once (not just first occurrence). It would be very complicated to track down indent errors if there is more than one present. Since indentation is not something that is being broken very often we decided to use standard approach here: detect

only first indentation error.

What is even **more** important is the fact that our tool does not detect syntax errors by catching an exception during parsing process. They are all being found iteratively without throwing whole code for validation. And that is something really huge. Why is it so good ? Because even if there is a syntactical error in the code, our tool is still able to parse it and then validate the rest of the code (which might be valid except that one incorrect line) and show errors and warnings or provide auto completion as if the entire code was valid.

This is especially helpful when there is a code with numerous errors (syntactical or even others like non-existent attribute etc.). While using other solutions it would be needed to repair errors one by one, our solution provides better feedback by showing all errors at once. And that is something that can potentially save a lot of tedious work for a programmer.

4

Daemon service

A daemon (or a service) is a background process that is designed to run autonomously, with little or no user intervention. Our daemon works as a pseudo compile-time compiler: it checks for various errors before run-time.

Plugin comunicates with this **daemon** running on certain port, listening for commands and responding with the answers. In other words, the daemon listens and acts as some kind of oracle - client connects and asks questions like if there are any errors or warnings in the code, or asks for an auto completion on certain variable. The daemon does its part of job by receiving request, deciding which action to trigger, computing the answer and responding back to the client with the results.

4.1 Our daemon implementation

Our daemon implementation of the interface described at 1.5.1 is written in Python and is using various python modules. As stated above, daemon is some kind of service. It is a server listening on certain local port waiting for clients to connect. For socket communication we used *asyncore* module - an asynchronous socket handler.

4.1.1 Asyncore

This module provides the basic infrastructure for writing asynchronous socket service clients and servers. The advantage of this module is that it does not use multi-threading. Instead, it uses *select()* system call to juggle between multiple communication channels (connected clients).

Why is multiple client support needed in the first place ? User can open multiple editors resulting in multiple requests from multiple buffers containing different texts. It would be possible to always send whole buffers, but this would not be very effective way how to do it. Therefore daemon is asynchronous and handles read/write operations via queues - always check if there is something in read queue. If it is, then read it and trigger correct action. Once you are done, put response in write queue with correct client address set. If another request came meanwhile, grab it from queue and repeat.

Server holds information about each of the clients and corresponding text buffers. Each text buffer is one 'open file' in editor.

Server consist of two main classes: **Host** and **Remote client**

Remote client

Remote client is a simple class holding information about connected client. These information stored include:

- *self.host* - holds reference to the instance of Host class
- *self.outbox* - messages which should be sent to this client
- *self.address* - information about address of client

Remote client must also support *handle_read*() and *handle_write*() methods. Both are implemented pretty straightforward. Read method is handled by host handler via given message plus client address. Write uses basic socket *send()* method.

Host

Host class is 'brain' of daemon service. It is responsible for reading from queue, writing into queue, waiting in loop forever and handling requests correctly, distributing signals to remote clients and more.

In *__init__* method host initialises everything it needs:

- creates socket on given port
- binds itself to this socket and loops itself forever to listen for requests
- initialize list of open connections - remote clients

Once any client tries to join, *__accept__()* method is called. If the call is successful, client is accepted and added to remote clients list. From this moment on, if client writes anything into the queue, host reads it and calls *handle_read()* handler.

This is where whole magic is happening: host checks which request was received and from which client (note that client must connected before, otherwise accept handler would be called). To do so, it needs to "parse" incoming request. To avoid sending plain strings between client and host delimited by some series of special strings we have decided to use JSON-like formatting of the requests (see 1.5.1).

4.1.2 Json

Json is Python modul used for object serialization and deserialization implementing a lightweight data interchange format based on a subset of JavaScript syntax. Its usage is very simple: single call of *json.dumps()* method serializes given map to string. Similarly *json.loads()* deserializes string into an object.

```
user_completion_response = json.dumps({'options' : listOfOptions})
host.respond(bytes(user_completion_response, 'UTF-8'), client_address)
```

This short example presents actual piece of code from user completion method. Message is first serialized via *json.dumps()* call, which returns serialized string in a JSON-

like format. After that, this string is sent by host to given *client_ address*. Note that before the string is sent it is transformed to bytes since we need to be compatible with lower python version on the client side (plugin side).

Each of the incoming requests are deserialized in *handle_ read()* call:

```
requestDecoded = json.loads(client_request.decode("utf-8"))
```

The *requestDecoded* variable now contains dictionary with key : value pairs. Note that again the request is decoded from bytes to string first.

4.1.3 Threading

We wanted our server to be truly asynchronous, meaning that user should NEVER wait for the response in a way that once the plugin sent something it should not be frozen waiting for the response. Similarly, if the daemon wants to handle multiple requests at the same time (for example while it is still validating code and request for auto completion comes), there needs to be some kind of mechanism which divides work to separate threads, each of them responsible for carrying out one of the requests. Python provides ideal solution for such a situation: *threading* module.

Threading module supports all of the standard constructs used in programming languages working with threads including creating, stopping, locking and using semaphores.

The following example shows simple way of creating and using daemon thread:

```
t = threading.Thread(...)
t.daemon = True
t.start()
```

Constructor takes mainly two arguments: name of the function which should be run and arguments passed to the function. The significance of the daemon flag is that the entire Python program exits when only daemon threads are left. Last line in the example simply starts the thread.

4.1.4 Handling requests

In the next few subsections we will try to provide brief descriptions of each of the requests implementation.

SendCurrentWorkingDirectory

Once user opens *.py file in vim, plugin tries to connect to the daemon (see 4.1.1). Right after connection is established plugin sends this request containing information about current filename and directory. This thread tries to find all function and class definitions in for given project.

Project usually consists of many directories and subdirectories with numerous files in each of them. To find root of project, daemon tries to locate the up-most **__init__.py** file. To find such a file, he starts searching in current directory - this is given in incoming request. If there is no such a file found, daemon next checks for **__init__.py** file in its parent directory. Whole process stops once root directory of whole file system is reached. If daemon finds **__init__.py** file at more than one height of recursion, the deeper (closer to the root of file system) one is marked as the right one. Note: user can specify other root file name than the default by modifying their *.vimrc* file (see 5.5 for further details).

Once init file is located, daemon starts process described in chapter 3:

- open file
- lexer on file data
- parser on file data
- specialised function to retrieve all function definitions and class definitions - **node_to_defs**

Function **node_to_defs** recursively iterates trough nodes and tries to locate *DEF NAME* or *CLASS NAME* pairs in given tree. These pairs indicate that either there is def keyword followed by name of function or class keyword followed by name of the

class.

And that is exactly what we need to extract. Daemon saves all function and class definitions for given file and goes on.

If definition with certain name is provided more than once (e.g. in two separate files), daemon holds information about both definitions and their locations.

At the end daemon holds information about project locations and list of dictionaries, where each dictionary consists of name of definition (either class or function) as a key and file/files, in which this definition occurred, as value.

Additionally daemon stores one char flag containing either F or C indicating that given record is either function or class definition.

Another thread initialized during startup is responsible for keeping these data correct - every 300 seconds daemon iterates trough all recently open projects and recalculates all definitions. Once done, it falls asleep via *time.sleep(300)* call.

SendWholeFile

Daemon holds information about buffers of all connected clients plus all of the open tabs of each client connected.

All buffers of all clients connected to the daemon are being held in simple dictionary where the key is PID of the process plus tab number (this is needed to cover situations where user opens more than one tab in single running Vim instance) and value is plain string.

SendWholeFile request therefore contains PID and tab id number plus buffer data as value. After deserialization of request daemon checks buffer dictionary for given key. If such a key is no present, daemon does an insert of new key with new value. Otherwise daemon updates identified record.

UpdateLine

To minimise communication complexity we do not want to send whole buffer (containing all lines of open file) all the time on each request. Instead, UpdateLine method updates only small window around current line which is being modified. Term small window in this case represents one line above and one line under the current line (the line where the cursor is).

Such a small window is needed since vim sometimes detects changes in current line too late. During testing some of the changes being made were either not detected correctly or, in worse case, were not detected at all. For example if the user asks for an auto complete and selects any of the values, new line is inserted before plugin could detect which option was actually selected. That would result in corrupted buffer data at daemon side.

ParseAndValidate

Once daemon receives this request, it knows that plugin is asking: "Is there anything wrong with this code ?" To answer this question, daemon needs to run complex process consisting from few steps where each of the steps needs previous one's output as its input.

1. Grab actual buffer data by given PID and tab number ID.
2. Call **lexer** on given data
3. Call **parser** on given data to retrieve abstract syntax tree (AST)
4. Evaluate output from previous step via inference package and retrieve errors and warnings
5. Respond to the client

Whole process of how lexer and parser together produce ast tree was described in chapter 3 on page 27.

GetAutocomplete

After receiving this request daemon calls specialised function with incoming variable name and line number as parameters.

Whole idea of what is done next is this: instead of asking "Does variable x have attribute y ?" we ask "Which attributes does variable x have at line n ?". We pass line number and variable name which should be resolved to inferencer and it returns list of attributes on given line (if any are found).

GetAllDefinitions

This request comes once user asks for list of all definitions in given project. Since this information was aleady calculated (see *SendCurrentWorkingDirectory*) daemon responds with correct options.

4.2 Daemon for other IDE/programming language

Using our daemon for Python code validation and auto completion in other IDEs than Vim (Vim is used in our implementation as chosen IDE, see section 5) is pretty easy since all work which needs to be done is writing plugin for given IDE which would sent requests described at 1.5.1 and then it would be able to display results received to the user.

Using daemon for other programming languages means that the daemon needs to be extended by another tool, which would be able to provide analysis for it. The extensibility of the daemon itself is very useful and helps tremendously here: there is no need to rewrite whole API since the requests can stay exactly the same. All that needs to be done is attaching another analysing tool to the daemon and then implement simple mechanism which would allow daemon to switch between the tools accordingly to requests coming. As stated in interface description 1.5.1, this would require only checking additional incoming parameter.

5

Plugin

Daemon service is running, ready to answer questions and provide valuable advice and info. To be able to actually use its potential, we needed to implement client side of process which would both communicate with daemon service and translates given information to readable form for user.

5.1 IDE - vim

After considering all of the given facts we decided to implement our client side for vim text editor. Main arguments speaking for Vim were:

- experienced community
- vim script and its ability to easily implement own plugins with support of other scripting languages (including Python)
- extensive and helpful documentation for vim script
- a lot of users are using vim for Python development

After choosing the IDE we had to summarize the features we needed to implement at client side of the project:

1. Connect to the daemon

2. Support request/receive operations in specified format while communicating with the daemon
3. Calculate correct requests by listening to the actions of the user
4. Show errors and warnings
5. Provide custom auto complete functions

5.2 Vim life cycle

As mentioned before in section 1.1, vim script supports powerful feature called **auto commands**. For our purpose we have used these auto commands:

- VimEnter - handles startup function plus sends whole buffer at start, sets auto complete function and user complete function
- VimLeavePre - closes open connection
- BufWritePost - synchronizes buffers by sending whole buffer to the daemon service
- CursorHold and CursorHoldI - triggered once user does not move cursor for set period of time. This is where code validation handler is triggered
- BufAdd - user opens new tab (send whole buffer to daemon) and clear marked errors
- TabEnter - once tab is entered/switched - again clear the errors
- CursorMovedI and CursorMoved - updates given line while user is moving around

5.3 Classes

All of classes used by the plugin are written in Python language and are being created during *StartupFunction()* call.

References to variables and class and function definitions are being passed between various vim script functions so there is no need to store all results vim variables and then transform them back and vice versa.

Client

Client class is asyncore implementation of client side of communication process. During initialization it tries to connect to the daemon service and establish socket for further exchange of information. Client holds information about server address and socket opened if the connection was successful. If for some reason the connection failed, plugin stops all other activity.

Implementation of message sending is pretty simple: client holds queue of messages which should be sent. Method *send_message()* simply appends given message to the queue. After that *handle_write* actually calls *send()* method.

Noticeable thing is that both write and read methods are called only if client actually can write/read to/from queue. If the server is busy responding to some other client, daemons queue is closed for write and client waits given amount of time if it opens up. Similar mechanism is used on read: client calls *handle_read()* method. If there was no check implemented on socket availability then this scenario could trigger pretty often: Client sends request to daemon. Daemon works on getting an answer, but since client wants that answer straight away (via possibly unprotected *recv()* call) there is nothing in the queue and the client is left with empty hands. Server responds a second later but client does not wait for the answer anymore since it gave up already. Whole process results in unaccepted message appearing in queue which further results in problems during next processing next of the requests.

To avoid this behaviour, availability check method is being called before every write or

read. Client checks if socket is ready (using *select.select()* call in specialised function). If it's not, client falls asleep for small period of time and tries again until the timeout occurs.

This elegant solution grants us consistent behaviour for each client connected to the daemon.

5.4 Functions

As described in section 4, plugin needs to create requests in such a way that the daemon would be able to deserialize them, trigger correct action and respond with the answer. Lets take a look at how our implementation handles this. List of functions used to create requests for the daemon is:

1. StartupFunction
2. SendWholeBuffer
3. SendCurrentLine
4. ParseAndValidate
5. GetOmnicomplete
6. GetFuncAndClassDefs

Again, all of the functions which not only send some data to the daemon but expect something in return all being handled by separate threads. This approach avoids "freezing" of the vim while waiting for the response. Threaded calls are: *ParseAndValidate*, *Getomnicomplete* and *GetFuncAndClassDefs*.

StartupFunction

Startup function is called whenever new instance of Vim is started. It contains definitions of all python classes (see above section 5.3). Initializes client and connects to

daemon (if possible).

Once client is connected, it sends information about current directory so the daemon can get all function and class definitions in whole project.

SendWholeBuffer

As the name implies, this function grabs whole buffer and sends it to the daemon. To avoid opening current file again and again to get contents of file, vim script methods are used instead. One simple command written right returns current file in a list:

```
let lines = getbufline(bufnr("%"), 1, "$")
```

After that, it is easy to just grab contents of this vim variable in Python part of the function and send it to the daemon. To avoid mixing the buffers, identifiers are appended to the request: PID of current process plus the tab number.

SendCurrentLine

Similar to *SendWholeBuffer*, but tracks only line on which user is either editing or just moving cursor. To avoid sending this request whenever cursor is moved, plugin holds local buffer and compares current line under cursor to the one in buffer. If there is any change, plugin sends changed line plus both previous and next line.

ParseAndValidate

Whenever cursor moving stops, vim tracks how much time passed since this occurred. Once the given timeout is reached, CursorHold/CursorHoldI (given on current editor mode) auto command is triggered.

This provides us with nice functionality - whenever user does stop writing or moving around, we can validate current code to get errors and warnings. And that is exactly what is happening - request for validation is sent.

Once plugin receives response from server, it needs to analyse it. Response contains two lists: list of errors and list of warnings. Plugin iterates trough both and calls either *ErrorHighlight* or *WarningHighlight* functions for each of the problems/warnings.

For errors there is red "!" on the left side of the screen shown. For warnings it is yellow "!" (see figure 6.1).

User can also call *:cope* command to show error and warning descriptions for each of the given errors.

Whole function call is being executed by separated thread (same as at **??**). This is because once the request is sent, plugin waits for the answer. If daemon does not respond immediately, user might be experiencing time span (lasting from few milliseconds to even few seconds for complicated and long codes) during which he could not interact with vim at all. Once the function is being executed separately, user can do further changes to the code while waiting for respond from the daemon.

Once one request is sent, given flag is set to true until thread is done with its job so the daemon is not spammed with requests from same client while it is still calculating the previous one.

Description contains generic information about the error, line on which the error/warning occurred and symbol, which is most likely the victim of causing the problem.

GetOmnicomplete

Omnicomplete, or auto complete function is heavily desirable functionality which does not work really well in other solutions. To call auto complete method user needs to simply press "." symbol. Whenever "." is pressed, GetOmnicomplete function is called.

In first cycle function finds beginning of the word until cursor. This prevents accidentally removing text written before position on which complete was called. Plugin sends request to daemon once it knows correct starting position. This request contains text until cursor (place where "." was placed) and line number.

Server responds with possible options. To display these options plus some additional info , plugin iterates trough given options and adds each of them to return list in specialized dictionary format containing word plus info. Info contains docstring of given function or class if it is present. Resulting visual effect can be seen on picture below.

GetFuncAndClassDefs

Sometimes while writing code user knows that he has some class or function defined somewhere in other files under given project but he does not remember the exact name. This feature is especially well suited for these problems.

User complete function call triggered by vim results in sending this type of request to the daemon. Once the daemon responds with options, plugin iterates trough them and shows both names and their corresponding occurrences in project on the right.

5.5 Vimrc settings

Since we wanted to give some space to the potential user we implemented three main parameters which can be modified via *.vimrc* file. The modifiable parameters are:

1. **Pynfer_root_filename** -specifies the name of the root file for which daemon should look when trying to calculate all definitions of classes and functions for given project. Default value is *__init__.py*.
2. **Pynfer_port_number** - specifies the number of port on which client tries to connect.
3. **Pynfer_number_of_iterations** - specifies how many times should be validation process run. The higher the number is, the longer the process might take.

For further details on how to change these parameters see Appendix chapter on page 58.

5.6 Client for other IDE/programming language

To sum it up, our client is "almost" language independent. We say almost since vim auto commands are limited to the *.py* files. Rewriting this last section of the client would make our plugin perfectly usable for other languages also. That is possible because one of the biggest features of our approach: plugin is not calculating anything. It is just responsible for creating requests, sending them to the daemon and afterwards handling the responses.

Rewriting client for other IDEs is a little more complicated, since this would require to study new materials about plugin creation in other development environments. Still, there is no need to rewrite parsing or validating process since that is being handled separately at the daemon side. And that makes it much, much easier.

6 Results

In this chapter we will provide results of our thesis divided into two sections. In the first section there are some screenshots from working solution shown. They cover each of the possible options shown by plugin in vim. In the second section we discuss how the daemon was created and that our plugin - daemon solution is truly more complex and universal than the usual approach to plugin creation.

6.1 Screenshots

```
 1 class A:
 2     def __init__(self, x, y):
 3         self.x=x
 4
 5         self.y=y
 6
 7     def method(self):
 8         self.x=5
 9
10         self.z=#sdfjsdgakdjgajd40000
11
12 a.counter
13
14 a.method()
15 a = A(3,4)
16 a.method()
17 a.x =a.z
18
19 a.method()
20
21 b,c,d = (1,2,3)
22 z = b+  c
23
24
~
~
~
```

Figure 6.1: Results from validation with errors shown.

```
test.py [+]
  1 test.py|10 error| invalid syntax
  2 test.py|12 error| Problem(message: NameError: Name 'a' is not defined, symbol: a, node: <class '_ast.Attribute'>, line: 12)
  3 test.py|14 error| Problem(message: NameError: Name 'a' is not defined, symbol: a, node: <class '_ast.Attribute'>, line: 14)
  4 test.py|17 error| Problem(message: nonexistent_attribute, symbol: z, node: <class '_ast.Attribute'>, line: 17)
~
~
~
~
~
~
[Quickfix List]
:cope
```

Figure 6.2: Error descriptions example.

```
    1 class A:
    2     def __init__(self, x, y):
    3         self.x=x
    4
    5         self.y=y
    6
    7     def method(self):
    8         self.x=5
    9
!  10         self.z=#sdfjsdgakdjgajd40000
   11
!  12 a.counter
   13
!  14 a.method()
   15 a = A(3,4)
   16 a.method()
!  17 a.x =a.z
   18
   19 a.method()
   20
   21 b,c,d = (1,2,3)
   22 z = b+  c
   23
   24 a.
~         __bases__
~         __class__
~         __getattr__
~         __init__
~         __setattr__
~         method
~         x
~         y
```

Figure 6.3: Auto complete example.

Figure 6.4: Get all function and class definitions example.

6.2 Universality of daemon

Our goal was to create daemon service that would not depend on certain IDE. If we wanted to integrate our solution to another IDE, we would only need to implement plugin, which would communicate with the service with way described in more detail in chapters 4 and 5.

Any client able to connect on given port on localhost and communicate via JSON format messages with the daemon is able to use daemon services while the daemon itself handles all the dirty work consisting of validating the code, providing needed feedbacks and doing all of the calculations.

Conclusion

We successfully completed all of our main goals given for this diploma thesis:

1. Propose and implement new approach to plugin creation and integration
2. Implement this approach for one dynamically typed language and one text editor
3. Provide reliable and better method of how to generate input for symbolic execution on dynamically typed language - Python

In the parsing part we first created the lexer and the parser tools robust enough to transform as much of the input Python source code as possible to abstract syntax tree while keeping track of deleted lines which are marked as invalid in the output provided for the user. This approach is better than existing solutions, which generally use only regular python parser for syntactical error detection and therefore can only detect single error (while our approach allows us to detect possibly all of them).

In the integration part we provided an unusual approach to plugin creation by separating IDE from the tool itself entirely. Afterwards we designed an abstract interface for communication between these two by describing the API. Lastly we implemented this solution for single IDE - Vim and programming language - Python. Plugin is communicating with the daemon service which is running independently on its own. Our implementation keeps door open for further extending the whole project with more programming languages and for more IDEs easily, which is indeed the key feature of this approach.

Bibliography

[1] Dominik Kapišinský *"Type-Awareness in Dynamic Languages"* Faculty of mathematics, physics and informatics, Comenius University, Bratislava, 2014.

[2] *"Vim documentation: intro"*, http://vimdoc.sourceforge.net/htmldoc/intro.html

[3] Steve Oualline *"Vi iMproved (VIM)"*, New Riders Publishing, 2001.

[4] Bob Glickstein *"Writing GNU Emacs extensions"* Kismet McDonough-Chan and Ellie Fountain Maden, 1997.

[5] Pavol Kobiakov, Sam Graham *"FlyMake"* http://www.emacswiki.org/emacs/FlyMake

[6] *"Sublime upgrade policy"*, http://www.sublimetext.com/sales_faq

[7] Asger Feldhaus, Jan Midtgaard, Michael I. Schwartzbach, *"Abstract Syntax Trees"* http://cs.au.dk/~mis/dOvs/slides/37a-abstractsyntaxtrees.pdf

[8] Paul Biggar and David Gregg *"Static analysis of dynamic scripting languages."* Draft: Monday 17th August (2009).

[9] William Edwards *"Why Dynamic Programming Languages Are Slow"* http://williamedwardscoder.tumblr.com/post/19538827844/why-dynamic-programming-languages-are-slow

[10] Kazuaki Ishizaki *"Adding Dynamically-Typed Language Support to a Statically-Typed Language Compiler"* http://www.cl.cam.ac.uk/research/srg/netos/vee_2012/slides/vee18-ishizaki-presentation.pdf

[11] "The Python Programming Language" http://groups.engin.umd.umich.edu/CIS/course.des/cis400/python/python.html

[12] "PLY (Python Lex-Yacc)." http://www.dabeaz.com/ply/ply.html.

[13] "Python grammar" https://docs.python.org/3.4/library/ast.html

Appendix

Installation and usage

To install and use our tool either use following steps are required:

1. Make sure that python3 is installed (easily doable with running *python3* command in console) and vim text editor is installed. To check that try running *vim* command in console. If unknown command error is returned, install vim by running following:

   ```
   sudo apt-get install vim
   ```

2. Download the solution from `https://github.com/pynfer/pynfer`

3. Extract the downloaded archive.

4. Either run *pynfer_install_vim.sh* script (which will try to complete all of the steps automatically) by typing

   ```
   sudo sh pynfer_install_vim.sh $1
   ```

   where $1 is location, where pynfer should be installed (for example */opt/pynfer*) or follow step-by-step guide provided in the following steps.

5. **Note: Only follow these steps if *install.sh* was not successful**.

   Move extracted folder to somewhere more appropriate (for example */opt/pynfer* folder (Note: location of your download might be different):

   ```
   sudo mv ~/Downloads/pynfer-master/tool /opt/pynfer
   ```

6. Make sure that there exists directory in which vim checks for plugins. To create such a directory write

   ```
   mkdir ~/.vim/plugin
   ```

7. Copy plugin.vim file to the directory created in the above step

```
cp ~/Downloads/pynfer-master/plugins/pynfer.vim ~/.vim/plugin
```

8. *Optional*: Add port number and number of iterations to local vim configuration file (**vimrc** - usually located at */.vimrc*). There are default values specified, so this step is completely optional. Example of lines added to */.vimrc* file:

```
"Default value for python is __init__.py
let g:Pynfer_root_filename ='something.py'

"Default value is 10003
let g:Pynfer_port_number = 10003

"Default value is 10
let g:Pynfer_number_of_iterations = 1

"Default value is 0
let g:Pynfer_default_python_settings = 0
```

9. Run daemon.py at */opt/pynfer* directory:

```
python3 /opt/pynfer/daemon.py
```

To specify other than the default port to be used, add it as an integer argument following the command:

```
python3 /opt/pynfer/daemon.py *PortNumber*
```

10. *Optional:* To avoid always navigating to the source folder where daemon.py is located and running above mentioned command, create symbolic link to shell script executing this command for you. First of all set executable permission on *daemon_ start.sh* file located in */opt/* directory:

```
chmod +x /opt/pynfer/daemon_start.sh
```

After that create symbolic link to this script:

```
sudo ln -s /opt/pynfer/daemon_start.sh /usr/bin/pynfer
```

From now on, running

```
pynfer
```

in console starts our daemon service.

11. Open any *.py file with vim and enjoy our tool !

    ```
    vim example.py
    ```

Note: Daemon gets closed on reboot or shut down of the computer, therefore before next usage it needs to be started again. Once our project will be ready for release our installation will add daemon to "run on system start" list and this step will be omitted.

Settings

Plugin also contains three settings especially useful for setting vim as python editor. To enable these, set *g:Pynfer_ default_ python_ settings* in */.vimrc* file to 1 by adding the following line:

```
let g:Pynfer_default_python_settings = 1
```

Settings included are:

- number - shows line numbers on the left side of the editor.
- showmode - shows current vim mode
- tabstop = 4 - changes tab width from 8 (vim default setting) spaces to 4 (used in Python).