Masarykova univerzita
Fakulta informatiky


# Nástroje na analýzu kódu v jazyku Python a vizualizácia ich výstupu

Bakalárska práca

**Ján Vorčák**

Brno, 2012

## Prehlásenie

Prehlasujem, že táto bakalárska práca je mojím pôvodným autorským dielom, ktoré som vypracoval samostatne. Všetky zdroje, pramene a literatúru, ktoré som pri vypracovaní používal alebo z nich čerpal, v práci riadne citujem s uvedením úplného odkazu na príslušný zdroj.

Ján Vorčák

**Vedúci práce:** Mgr. Marek Grác

## Poďakovanie

Ďakujem Mgr. Marekovi Grácovi, vedúcemu tejto bakalárskej práce a Ing. Martinovi Sivákovi za podnetné návrhy pri jej tvorbe, ale aj za možnosť zrealizovať tento projekt formou bakalárskej práce.

## Zhrnutie

Cieľom tejto bakalárskej práce je návrh a implementácia nástroja, ktorý umožní analyzovať a následne vizualizovať projekt v jazyku Python. V úvodnej kapitole si predstavíme jazyk Python a jeho vlastnosti, ako aj existujúce nástroje na jeho analýzu. Neskôr navrhneme a analyzujeme knižnice, ktoré nám pomôžu k samotnej implementácii. Nástroj navrhneme, v poslednej kapitole popíšeme jeho implementáciu a zhodnotíme dosiahnuté výsledky.

## Kľúčové slová

analýza kódu, Python, Pylint, Pyreverse, Gtk, Graphviz, UML, introspekcia

# Obsah

# 1 Úvod

V dnešnej dobe sa pri vývoji aplikácií čoraz viac kladie dôraz nielen na samotné programovanie, ale najmä na analýzu a návrh systému, dôkladné otestovanie, ale aj spätnú analýzu samotného kódu. Z tohto dôvodu vzniká pre takmer každý programovací či značkovací jazyk množstvo nástrojov, ktoré nám pomáhajú automaticky objaviť potencionálne chyby či porušenie konvencií v zdrojových kódoch. Častokrát sú tieto nástroje síce efektívne a ľahko konfigurovateľné, no najmä výstupom nie príliš užívateľsky prívetivé. Asi najpoužívanejším nástrojom na analýzu kódu v jazyku Python je konzolová utilita Pylint [8]. Pokiaľ si chceme vďaka tomuto programu vytvoriť obraz o väčšom projekte, nestačí nám iba textový výstup, no je potrebné tieto dáta interaktívne vizualizovať.

Cieľom tejto bakalárskej práce je analyzovať nástroje na analýzu a vizualizáciu Python kódu ako aj vlastnosti jazyka, ako je napríklad introspekcia, ktoré nám túto analýzu umožnujú. Z dôvodu vizualizácie je nutné taktiež naštudovať základy modelovania hierarchických štruktúr v jazyku UML. Výstupom bakalárskej práce je nástroj, ktorý poskytuje funkcionalitu, ktorú v existujúcich nástrojoch nenájdeme. Nástroj teda zjednoduší orientáciu najmä vo väčšom projekte pomocou vizualizácie dát najmä za pomoci vygenerovania interaktívnych UML diagramov. Program bude taktiež dopĺňať funkcionalitu, ktorú nám klasický program Pylint neumožnuje, ako je napríklad filtrácia falošných poplachov.

Práca pozostáva zo šiestich kapitol. V druhej kapitole si predstavíme jazyk Python a jeho vlastnosti, neskôr rozanalyzujeme existujúce nástroje na analýzu kódu v jazyku Python a ich nedostatky. Pred implementačnou časťou si predstavíme knižnice, ktoré nám neskôr pomôžu pri implementácií nástroja, ako je napríklad knižnica Gaphas [3] na vykresľovanie grafiky v GTK+ [4] programoch alebo samotný Pylint. Nasledovať bude kapitola o návrhu a samotnej implementácii. Nakoniec zhodnotíme, či implementovaný nástroj spĺňa určené požiadavky či už po stránke funkcionality, alebo škálovateľnosti a navrhneme zmeny na jeho zlepšenie.

# 2 Python

## 2.1 Charekteristika jazyka Python

Python je vysokoúrovňový, objektovo orientovaný dynamický jazyk, ktorý vytvoril holandský programátor Guido van Rossum ako následníka jazyka ABC [19]. Vyznačuje sa prehľadnou syntaxou, jednoduchosťou a modulárnosťou. Python podporuje viacero programátorských paradigiem, najmä objektovo orientované, imperatívne a čiastočne aj funkcionálne paradigmy.

Existuje viacero implementácií jazka Python. Medzi najznámejšie patria:

- CPython
- Jython
- Python pre .NET
- IronPython
- PyPy

Najpožívanejšou a najpodporovanejšou implementáciou je ale CPython. Každá z implementácií sa môže líšiť špecifickými vlastnosťami mimo štandardnej Python dokumentácie. Zdrojové kódy sú pri interpretácii preložené do byte kódu a zvyčajne uložené v `.pyc` alebo `.pyo` súboroch, ktoré sú neskôr spúšťané virtuálnym strojom.

V štandardnej knižnici je dostupných mnoho dátových typov, ako napríklad reálne a komplexné čísla, celé čísla s neobmedzenou dĺžkou, znakové reťazce, zoznamy a slovníky. Dátové typy sú silno a dynamicky typované. Operácia nad nekompatibilným typom spôsobí vyvolanie výnimky.

Python podporuje objektovo orientované programovanie vrátane viacnásobnej dedičnosti. Kód je sústredený do modulov a balíkov s možnosťou importovať špecifický modul, triedu, funkciu alebo iný objekt. Za účelom ošetrenia chýb Python podporuje vyvolávanie a odchytávanie výnimiek. Automatická správa pamäti nahrádza nutnosť manuálne alokovať a uvoľnovať pamäť v kóde. [1]

## 2.2 Introspekcia v jazyku Python

Introspekcia je schopnosť programovo preskúmať daný objekt a rozhodnúť o jeho identite, vlastnostiach a schopnostiach. Jazyk Python podporuje rozsiahlu introspekciu objektov. Medzi hlavné informácie, ktoré potrebujeme

o objektoch v jazyku Python zistiť, patrí ich meno, typ, identita, vlastnosti, schopnosti a ich pôvod. Jazyk Python ponúka množstvo nástrojov, ktoré nám tieto vlastnosti umožnujú zistiť. Môžeme ich rozdeliť do dvoch hlavných skupín. V prvej sú funkcie či už zo štandardnej knižnice, alebo z pomocných modulov, akým je napríklad modul `inspect` [5]. Do druhej skupiny radíme atribúty objektov, ktoré uchovávajú užitočné informácie priamo v objekte.

- Funkcia `dir()` je jedným z hlavných nástrojov introspekcie v jazyku Python a vracia zotriedený zoznam mien atribútov objektu, ktorý bol uvedený ako jej argument. Funkcia je súčasťou štandardnej knižnice, takže nemusíme importovať žiaden modul pre jej použitie. Na výpis funkcií zo štandardnej knižnice môžeme teda využiť samotnú funkciu `dir()`.

```
1 In [1]: print dir(__builtin__)[-10:]
2 ['str', 'sum', 'super', 'tuple', 'type', 'unichr',
      'unicode', 'vars', 'xrange', 'zip']
```

  V prípade, že je funkcia `dir()` použitá bez argumentov, vracia zoznam mien, ktoré sú momentálne definované. Zaujímavá je funkcia `locals()`, ktorá vracia slovník momentálne definovaných premenných a ich hodnôt.

```
1 In [2]: print dir()
2 ['In', 'Out', '_', '__', '___', '__builtin__', '
    __builtins__', '__name__', '_dh', '_i', '_i1',
    '_i2', '_ih', '_ii', '_iii', '_oh', '_sh', '
    exit', 'get_ipython', 'help', 'quit']
```

- Atribút `__dict__` uchováva slovník atribútov a metód daného objektu. Nie sú tu však zahrnuté metódy definované v triede, z ktorej bol objekt odvodený.

```
1 In [13]: class A(object):
2    ....:     x=1
3    ....:     y=''
4    ....:     def helloWorld(self):
5    ....:         pass
6    ....:
7
8 In [14]: a=A()
9
```

```
10 In [15]: a.__dict__
11 Out[15]: {}
12
13 In [16]: A.__dict__
14 Out[16]:
15 <dictproxy {'__dict__': <attribute '__dict__' of '
      A' objects>,
16  '__doc__': None,
17  '__module__': '__main__',
18  '__weakref__': <attribute '__weakref__' of 'A'
      objects>,
19  'helloWorld': <function __main__.helloWorld>,
20  'x': 1,
21  'y': ''}>
```

- Funkcia `vars()` je bez argumentov ekvivalentná funkcii `locals()`. Pri použití s argumentom `x` je to ekvivalent `x.__dict__`.

- Atribút `__doc__` obsahuje komentáre, ktoré popisujú objekt. V prípade, že prvý v module, triede alebo v metóde je znakový reťazec, je automaticky považovaný za `__doc__` atribút. V opačnom prípade sa jeho hodnota nastaví na `None`. Pri použití optimalizácie sa hodnoty `__doc__` z dôvodu kompaktnosti nevkladajú do byte kódu.

```
1 In [3]: print __doc__.__doc__
2 str(object) -> string
3
4 Return a nice string representation of the object.
5 If the argument is a string, the return value is
     the same object.
```

- Atribút `__name__` obsahuje názov objektu odvodený z jeho typu. Niektoré objekty, ako napríklad znakové reťazce, tento atribút neobsahujú. Tento atribút obsahujú napríklad moduly. V prípade, že spúšťame skript priamo pomocou Python interpretu, je atribút `__name__` nastavený na hodnotu `'__main__'`, nakoľko Python interpret je považovaný za hlavný modul, a to aj v prípade, že Python skript spúšťame z príkazového riadku. Je teda často používaný na rozpoznanie, či daný modul len importujeme, alebo priamo spúšťame.

```
1 In [4]: def my_function():
```

```
2     ....:     pass
3     ....:
4
5  In [5]: my_function.__name__
6  Out[5]: 'my_function'
7
8  In [6]: __name__
9  Out[6]: '__main__'
```

- Funkcia `type()` zo štandardnej knižnice vracia typ jej argumentu. Ten vracia v podobe typového objektu, ktorý môže byť porovnávaný s typmi definovanými v module `types`.

```
1  In [7]: type(my_function)
2  Out[7]: function
3
4  In [8]: type(1)
5  Out[8]: int
```

- Funkcia `id()` vracia unikátnu identitu objektu. Táto funkcia je užitočná, nakoľko viacero premenných môže odkazovať na rovnaký objekt. Funkcia `id()` vracia pamäťovú adresu objektu.

```
1  In [9]: id('string')
2  Out[9]: 3078023712L
3
4  In [10]: id(id)
5  Out[10]: 3078187660L
```

- Funkcie `hasattr()` a `getattr()` - v prípade, že je potrebné zistiť prítomnosť alebo hodnotu atribútu, štandardná knižnica ponúka funkcie `hasattr()` a `getattr()`.

```
1  In [11]: hasattr(id, '__name__')
2  Out[11]: True
3
4  In [12]: getattr(id, '__name__')
5  Out[12]: 'id'
```

- Funkcia `callable()` - v niektorých prípadoch môžu objekty slúžiť na vyvolanie určitého druhu udalostí. Pomocou funkcie `callable()` sa dá overiť, či je daný objekt spustiteľný.

```
In [13]: callable(id)
Out[13]: True

In [14]: callable(1)
Out[14]: False
```

- Funkcie `isinstance()` a `issubclass()` - funkcia `isinstance()` vracia hodnotu v závislosti od toho, či je objekt inštanciou danej triedy. Funkcia vracia pravdivú hodnotu aj v prípade, že je objekt inštanciou jej potomka.

  Funkcia `issubclass()` vracia pravdivú hodnotu v prípade, že objekt reprezentujúci triedu je podtriedou druhého argumentu.

```
class Person(object):
    pass
class Student(Person):
    pass
p=Person()
s=Student()

In [15]: isinstance(s, Student)
Out[15]: True

In [16]: isinstance(s, Person)
Out[16]: True

In [17]: issubclass(Student, Person)
Out[17]: True
```

# 3 Nástroje na analýzu kódu pre jazyk Python

## 3.1 Nástroje na detekciu chýb pre jazyk Python

### 3.1.1 PEP8

Ide o nástroj, ktorý vyvíja Johann C. Rocholl. Tento nástroj slúži na otestovanie kódu Python podľa konvencií definovaných v PEP 8 [6]. Je to konzolový program, ktorý pomáha zvyšovať prehľadnosť a lepšiu štruktúru kódu.

Program PEP8 umožnuje programátorovi pomocou rôznych prepínačov ovládať analyzátor. V predvolenom móde vypíše program každú chybu len raz, čo je pre analýzu rozsiahlejšieho projektu nevhodné. Voľba `-repeat` vypíše každú chybu bez ohľadu na to, či sa už v kóde vyskytuje viackrát, alebo nie. Voľba `-filename=patterns` obmedzuje spracovávané súbory len na tie, ktoré vyhovujú zadanému regulárnemu výrazu. Zaujímavým prepínačom je `-show-pep8`, ktorý k danej chybe vypíše na štandardný výstup aj text z definície PEP 8, ktorý daná časť kódu porušuje. Utilita Pep8.py je nástroj, ktorý výrazne dopomáha k prehľadnosti a ucelenosti kódu. Chýbajú mu však možnosti detekovania chýb v kóde [6].

### 3.1.2 PyChecker

PyChecker je program zachytávajúci problémy, na ktoré upozorňuje u nedynamických jazykov kompilátor v podobe varovaní a chýb. Medzi problémy, ktoré tento program zachytí, patrí napríklad zlý počet parametrov predaných funkcii, metóde či konštruktoru, používanie neexistujúcich metód a tried, ako aj použitie premennej pred jej inicializáciou. Detekuje taktiež nepoužité inštancie globálnych či lokálnych premenných, kontroluje definíciu `self` ako prvej premennej u metód tried, ako aj úroveň dokumentácie pre triedy, moduly a metódy [7].

Jednou z predností je aj priamy import do zdrojového kódu. Pokiaľ má užívateľ prístup a práva k modifikácii zdrojového kódu, je veľmi jednoduché PyChecker naimportovať priamo. Jeho hlavnou nevýhodou je, že daný kód spustí a vykoná. PyChecker je teda nevhodný na detekciu chýb v aplikáciách, ktoré napríklad pracujú nad databázou, alebo pracujú v ostrom prostredí, ktoré je nevhodné na testovanie.

### 3.1.3 Pylint

Pylint je jeden z najpoužívanejších, ľahko konfigurovateľných nástrojov na analýzu kódu Python, využívaný či už manuálne, alebo automaticky. Je vyvíjaný za podpory Logilab.org a vydaný pod licenciou GNU GPL [11]. V zdrojových kódoch analyzuje potencionálne chyby a varovania, ale aj porušenie konvencií podľa štandardu PEP 8. S projektom Pylint je dodávaný aj program Pyreverse, ktorý pre daný kód vygeneruje UML diagram v podobe *dot* [14] formátu. Daný *dot* súbor je možné priamo pretransformovať do *PNG* alebo iného formátu. Výhodou programu Pylint oproti PyChecker je, že daný súbor väčšinou nespúšťa, ale analyzuje staticky. Je teda vhodný do každého prostredia. Pylint dopĺňa PyChecker hlavne v kontrole dĺžky riadkov a kontrole názvov premenných za pomoci regulárnych výrazov. Overuje taktiež implementáciu deklarovaných rozhraní a detekuje duplicitný kód. Obzvlášť zaujímavou vlastnosťou je generovanie metrík a externých závislostí [8].

Správanie sa programu je možné upraviť pomocou prepínačov špecifikovaných na príkazovom riadku alebo pomocou konfiguračného súboru. V konfiguračnom súbore je možná filtrácia varovaní a chýb rôznych druhov, ignorovanie rôznych typov súborov. Taktiež je možné načítať rôzne typy doplnkov, nastavenie minimálneho počtu znakov v riadkoch, obmedzenie veľkosti modulu, nastavanie znakov využitých na odsadenie, vlastné definície zastaralých modulov.

Najmä u väčších projektov s požiadavkami na prehľadnosť kódu a dobrý návrh užívateľ ocení možnosť nakonfigurovať maximálny počet atribútov triedy pomocou prepínača *max-attributes*, rodičov triedy pomocou prepínača *max-parents*. Počet výrazov *return* a *yield* je možné obmedziť s využitím prepínača *max-returns*. U niektorých existuje možnosť špecifikovať aj minimálny počet, napríklad u počtu verejných metód triedy pomocou prepínačov *min-public-methods* alebo *max-public-methods* [9].

### 3.1.4 Zhrnutie

Pylint patrí medzi najpoužívanejší program na analýzu kódu Python, nakoľko je ľahko konfigurovateľný a kombinuje funkcionalitu ostatných nástrojov. Do každého prostredia je vhodné vybrať nástroj na analýzu vždy podľa špecifických potrieb. Aj keď Pylint môžeme nazvať najpoužívanejším nástrojom, projekty ako Moap [20] využívajú pre testovanie PyChecker.

Program PEP8 má taktiež široké použitie, no len s obmedzením na porušenie konvenčných chýb, ktoré sú v niektorých prípadoch akceptovateľné,

hlavne za účelom sprehľadnenia kódu alebo z historických dôvodov projektu.

## 3.2 Nástroje na vizualizáciu projektu v jazyku Python

Nástroje na vizualizáciu nám slúžia nielen na vygenerovanie UML diagramov, ale aj na zobrazenie dodatočných informácií o projekte, akými sú napríklad metriky alebo chyby v kóde. V tejto kapitole si predstavíme dostupné nástroje na vizualizáciu Python kódu.

### 3.2.1 Pyreverse

Pyreverse je program dodávaný spoločne s utilitou Pylint. Jeho úlohou je vygenerovať UML diagramy pre Python projekt. Vykonáva synaktickú analýzu Python balíkov a vygeneruje UML diagram vo formátoch `dot` alebo `vgc`. Pyreverse je za pomoci programu dot [14] schopný vygenerovať aj UML diagram vo formáte PNG.

Medzi hlavné výhody programu Pyreverse patrí jeho rýchlosť a široká ponuka prepínačov, ktoré umožňujú ignorovať moduly a súbory, filtrovať typy atribútov alebo ovládať výzor výsledného UML diagramu. Zaujímavou vlastnosťou je vygenerovanie diagramu tried súvisiacich s jednou konkrétnou triedou. Túto vlastnosť umožnuje prepínač `-class=<class>`.

Nevýhodou programu Pyreverse je jeho neinteraktívnosť. Pri zmene v projekte musíme daný graf pregenerovať. Navyše nemôžeme pomocou programu Pyreverse upravovať projekt.

```
1 $ pyreverse -o png .
```

### 3.2.2 Pylint-gui

Pylint-gui je program, ktorý bol vytvorený najmä pre užívateľov operačného systému Windows ako alternatíva k spúštaniu Pylint pomocou príkazového riadku. Grafické rozhranie programu využívajúce `Tk` je veľmi jednoduché a ide len o prevedenie konzolového výstupu do vizuálnej podoby. Pylint-gui teda neodstránil nedostatky programu Pylint. Pri editácii projektu treba výstup znova pregenerovať, nakoľko chyby sú identifikované číslom riadku, ktorý sa pri editácii môže zmeniť.

### 3.2.3 Integrácia programu Pylint do vývojových prostredí

Program Pylint bol integrovaný do väčšiny vývojových prostredí (IDE), ktoré jeho výsledky graficky interpretujú. Táto integrácia je často len v podobe doplnkov a modulov bez priamej podpory IDE. Ďalšou z nevýhod je obmedzenosť na dané vývojové prostredie.

Jedným z príkladov týchto doplnkov je modul PyDev[21] pre vývojové prostredie Eclipse. Jeho výhodou je jeho automatické spustenie po zmene súboru, ktoré však ale pri väčších projektoch môže spôsobiť spomalenie. Výhodou je aj možnosť prefiltrovať zobrazenie podľa typu chyby, ktorú Pylint zaznamená. Obmedzuje sa ale len na nasledujúcich päť skupín chýb, nie však na konkrétne chybové správy.

- FATAL
- ERRORS
- WARNINGS
- CONVENTIONS
- REFACTOR

PyDev[21] taktiež nepodporuje filtráciu falošných poplachov ani generáciu UML.

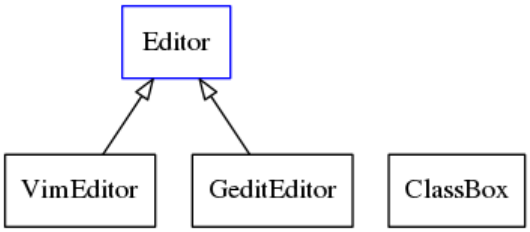
### 3.2.4 Graphviz

Graphviz je software na vizualizáciu grafov. Graphviz pozostáva z jazyka `dot` popisujúceho grafy a množstva iných nástrojov, ktoré generujú alebo inak spracovávajú súbory v tomto formáte. Tieto nástroje môžu pre výstup zvoliť množstvo užitočných formátov ako napríklad PDF, PNG alebo SVG. Dot súbor môže vyzerať nasledovne.

```
1  digraph "name" {
2  charset="utf-8"
3  rankdir=BT
4  "4" [color="blue", shape="record", label="Editor"];
5  "5" [shape="record", label="VimEditor}"];
6  "6" [shape="record", label="GeditEditor"];
7  "16" [shape="record", label="ClassBox"];
8  "5" -> "4" [arrowtail="none", arrowhead="empty"];
9  "6" -> "4" [arrowtail="none", arrowhead="empty"];
10 }
```

Po spustení nasledujúceho príkazu sa vygeneruje graf vo formáte PNG.

```
1 $ dot -Tpng graph.dot -o graph.png
```




Obrázok 3.1: Graf vygenerovaný programom dot

### 3.2.5 Zhrnutie

Existuje množstvo utilít a programov, ktoré pomáhajú výsledky analýzy Python kódu vizualizovať. Medzi základné problémy, ktoré tieto programy majú, patrí ich neinteraktívnosť, nutnosť výsledky pregenerovať po editácii kódu, alebo chýbajúca funkcionalita. Pri implementácii projektu sa pokúsime zamerať práve na tieto chýbajúce vlastnosti. Výsledný program by mal byť rýchly a neobmedzovať programátora na jeden druh editoru, ako je to napríklad pri doplnkoch IDE.

# 4 Analýza nástrojov potrebných k implementácii

## 4.1 Analýza nástroja Gaphas

### 4.1.1 Základná charakteristika nástroja Gaphas

Gaphas predstavuje zoskupenie knižníc a nástrojov na vykresľovanie grafických objektov na určené elementy grafického rozhrania GTK. Je naprogramovaný v jazyku Python a vydaný pod LGPL [10] licenciou.

### 4.1.2 Popis Gaphas API

Gaphas API využíva MVC návrhový vzor a môžeme ho teda rozdeliť na 3 hlavné časti – `Model`, `View` a `Controller`.

`Model` je časť, ktorá obsahuje doménovú reprezentáciu projektu. Časť `View` prevádza dáta reprezentované modelom do podoby vhodnej k interaktívnej prezentácii užívateľovi. Časť `Controller` reaguje na udalosti a zaisťuje zmeny v častiach `Model` a `View`. [24]

- Model – je časť API, ktorá obsahuje:
  - `api/canvas` – plátno na vykresľovanie, ktoré sa správa ako kontajner pre vykresľované položky. Canvas v sebe zahŕňa atribúty, ktoré súvisia s vykresľovaním, napríklad atribút `gaphas.Solver` alebo atribút `__connections` uchovávajúci informácie o prepojeniach medzi jednotlivými položkami
  - `api/items` – položky `gaphas.item.Element` a `gaphas.item.Line` odvodené od triedy `gaphas.item.Item`, ktoré sú uchovávané a vykresľované na plátne. Jednoduchým odvodením novej triedy z `gaphas.item.Item` môžeme vytvoriť vlastný prvok na vykresľovanie
  - `api/connectors` – v tejto kategórií ide o triedy `gaphas.connector.Handle` a `gaphas.connector.Port`, ktoré umožňujú spájať jednotlivé položky na plátne
  - `api/solver` – umožňuje definovať podmienku medzi dvomi a viacerými premennými a udržiavať jej platnosť pri zmene premenných
  - `api/constraint` – väzby pre premenné, ktoré sú zaregistrované na plátne. Každá väzba obsahuje zoznam premenných, ktoré

sú zaregistrované v objekte typu `gaphas.solver.Solver`

- `api/utils` – obsahuje pomocné funkcie týkajúce sa najmä vykresľovania textu na plátno

- View – obsahuje všetky triedy súvisiace so zobrazovaním a vykresľovaním jednotlivých elementov:

  - `api/view` – trieda, ktorej úlohou je spravovať vykresľovanie Gaphas položiek, uchováva vykresľovacie plátno, informácie o označených objektoch a objektoch, nad ktorými sa nachádza myš
  - `api/painters` – objekty, ktoré vykresľujú jednotlivé objekty
  - `api/gtkview` – trieda implementujúca funkcionalitu `gaphas.view.View` v `Gtk.DrawingArea`. Slúži teda na vykreslenie plátna na Gtk+ element. Používa nástroje z časti controller a objekty z časti view

- Controller – časť API slúžiaca na interakciu s plátnom a objektami obsahuje:

  - `api/tools` – nástroje, ktoré poskytujú pohľadu interaktívnosť tým, že spracovávajú udalosti, ktoré sú ním posielané. Gaphas API poskytuje nástroj `HoverTool` slúžiaci k označeniu položky, ktorá sa nachádza pod myšou, nástroj `ItemTool` poskytuje výber a premiestňovanie položiek, `HandleTool` výber a pohyb s objektami typu `gaphas.connector.Handle`. Okrem nich ponúka aj nástroje ako `PanTool` pre pohyb plátna alebo `PlacementTool` pre umiestnenie nových položiek na plátno. Nástroje môžu byť nakombinované a zreťazené do jedného nástroja kombinujúceho ich vlastnosti s použitím triedy `ToolChain`. Nástroje sú implementované pomocou udalostí. Nástroj môže udalosť obslúžiť alebo ignorovať. Existuje teda jednoduchá možnosť v prípade potreby naimplementovať vlastný nástroj
  - `api/aspects` – definujú funkcionalitu na rozmedzí položiek a nástrojov. Vysporiadajú sa teda s pohybom položiek alebo ich označením

### 4.1.3 Zhrnutie

Gaphas knižnica umožnuje vykresľovanie grafiky na plátno v prostredí GTK+. Jej výhodou je možnosť prispôsobiť vykresľované objekty priamo na mieru. Výborne spracovaná je aj interakcia medzi objektami. Vďaka väzbám môžeme dva objekty udržovať v definovanej pozícii, napríklad na jednej čiare, v jednom bode alebo na pozícii definovanou rovnicou.

## 4.2 Analýza nástroja Pylint

### 4.2.1 Pylint

Kľúčová trieda v štruktúre programu Pylint je trieda `pylint.lint.PyLinter`, ktorá spravuje nastavenia, doplnky, aktiváciu a deaktiváciu správ na úrovni modulov. Uchováva taktiež informácie o počte tried, metód a iné jednoduché štatistiky. Táto trieda je spúšťaná hlavnou triedou `Run` pri spustení programu. Spravuje taktiež triedy typu `Checkers`, ktoré analyzujú kód a triedu typu `Reporter` - triedu zopovednú za výstup.

V nasledujúcom odseku si podrobne opíšeme najdôležitejšie prvky programu Pylint.

- Triedy typu Checkers

  Ide o triedy, ktoré rozširujú aspoň jednu z tried

  - IRawChecker
  - IASTNGChecker

  Tieto triedy sú zaregistrované v triede `pylint.lint.Pylinter` pomocou jej metódy `register_checker`. Po zaregistrovaní prebehne spracovanie v metóde `pylint.lint.Pylinter.check`, kde sa rozdelia zaregistrované triedy podľa ich predkov a začne sa prehľadávanie a kontrola modulov. V triedach typu Checker sa zavolá metóda `process_module`, ktorej implementácia je ponechaná na každej triede samostatne.

  Každá takáto trieda má zaregistrovaný slovník chybových správ, ktorý dokáže detekovať. Ako kľúč je použitý kód chyby a ako hodnota dvojica obsahujúca textovú reprezentáciu chyby a jej popis.

Ukážka kódu 4.1: Slovník chybových hlášok triedy `EncodingChecker`

```
1 MSGS = {
2     'W0511': ('%s',
3               'Used when a warning note as FIXME
                   or XXX is detected.'),
4     }
```

Pri náleze chyby checker zaregistruje chybu pomocou metódy `add_message` zdedenej z triedy `pylint.checkers.BaseChecker`, ktorá chybu predá inštancii triedy `pylint.lint.Pylinter`.

```
1     def add_message(self, msg_id, line=None, node=
          None, args=None):
2         """add a message of a given type"""
3         self.linter.add_message(msg_id, line, node
              , args)
```

- Triedy typu Reporters – Tento typ tried je odvodený od triedy `pylint.reporters.BaseReporter`. Triedy slúžia na formátovanie a vygenerovanie samotného výstupu. Pri inicializácii sa triede pomocou metódy `set_output` nastaví výstupné zariadenie, ktoré je v predvolenom režime nastavené na štandardný výstup. Je možné ho predefinovať na súbor alebo iný typ výstupného zariadenia. Tak ako pri triedach typu `Checker`, je aj v tomto prípade veľmi jednoduché vytvoriť vlastnú implementáciu.

  V programe Pylint ale nájdeme pomerne široký výber výstupov.

  - HTMLReporter
  - GUIReporter
  - TextReporter
  - ParseableTextReporter
  - VSTextReporter
  - ColorizedTextReporter

Jednou z nevýhod programu Pylint je neprítomnosť nástroja, ktorý by dokázal ignorovať konkrétne chyby v zdrojovom kóde. Prvou možnosťou programu Pylint je ignorovať daný typ chyby pomocou prepínača `-disable`.

```
1 $ pylint --disable=E1101,R0904
```

Druhou možnosťou je spustiť len zvolené triedy typu Checker pomocou prepínača `-enable`.

```
1 $ pylint --enable=basic,variables,classes,metrics,
    similarities
```

Táto funkcionalita je v praxi veľmi potrebná, nakoľko sa v kóde často vyskytujú miesta, kde istú chybu tolerujeme. Nechceme však jej detekciu vypnúť v celom projekte.

### 4.2.2 Pyreverse

Pri prehľadávaní projektu si program Pyreverse vygeneruje z daných argumentov inštanciu triedy `logilab.astng.manager.Project` za pomoci triedy `ASTNGManager` z balíčka `logilab.astng`, ktorý je závislosťou projektu Pylint (viď Ukážka kódu 4.2, riadok 2). Táto trieda obsahuje informácie o názve, ceste, moduloch a premenných analyzovaného objektu.

Za pomoci triedy `logilab.astng.Linker` prehľadá daný projekt a vygeneruje vzťahy medzi jednotlivými triedami (viď Ukážka kódu 4.2, riadok 3).

Inštancie týchto dvoch tried sú predané ako parametre objektu typu `DiadefsHandler` (viď Ukážka kódu 4.2, riadok 4), ktorý za pomoci triedy `ClassDiadefGenerator` vygeneruje diagram tried (viď Ukážka kódu 4.2, riadok 5). Diagram tried je neskôr predaný triede zodpovednej za zápis.

Nasledujúca časť kódu je hlavnou kostrou programu Pyreverse.

Ukážka kódu 4.2: Hlavná časť programu Pyreverse

```
1  try:
2      project = self.manager.project_from_files(args)
3      linker = Linker(project, tag=True)
4      handler = DiadefsHandler(self.config)
5      diadefs = handler.get_diadefs(project, linker)
6  finally:
7      sys.path.pop(0)
8
9  if self.config.output_format == "vcg":
10         writer.VCGWriter(self.config).write(diadefs)
11 else:
12     writer.DotWriter(self.config).write(diadefs)
```

Triedou zodpovednú za zápis rozumieme triedu, ktorá rozširuje triedu `pylint.pyreverse.writer.DiagramWriter`.

Príkladmi takýchto tried sú napríklad nasledujúce triedy:

- DotWriter
- VCGWriter

Úlohou týchto tried je implementovať medódu `set_printer`, ktorá nastaví backend, ktorý bude zapisovať do súboru. Implementujú aj medódy `get_title`, `get_values` a `close_graph`. Prácu za nich vykonáva funkcia `write`, ktorú zdedia po triede `DiagramWriter`, ktorá pomocou interných metód `write_packages` a `write_classes` vyvoláva metódy backendu pre zápis. Pri implementácii novej triedy typu `writer` je teda dôležité naimplementovať triedu, ktorá sa využíva ako backend a inicializovať a nastaviť ju vo vnútri metódy `set_printer`. Dôležité je taktiež backend triedu korektne uzavrieť v metóde `close_graph`, ktorá je taktiež volaná metódou `write`.

# 5 Implementácia nástroja

## 5.1 Cieľ

Naším cieľom je naimplementovať nástroj, ktorý bude schopný prehľadne a interaktívne zobrazovať chyby v kóde v rámci hierarchie modulov, tried a funkcií a taktiež ich zobraziť v zdrojovom kóde bez nutnosti pregenerovania. Nástroj bude zobrazovať zdrojové kódy a umožňovať ich editáciu, pričom nebude obmedzený na jeden typ editoru. Taktiež by mal fungovať v plnej funkčnosti multiplatformne.

## 5.2 Zdôvodnenie výberu knižníc a nástrojov pre implementáciu

Program bude implementovaný ako aplikácia s grafickým rozhraním v programovacom jazyku Python, nakoľko vďaka jeho introspekcii je väčšina programov a knižníc pre jeho analýzu naprogramovaná práve v ňom.

Ako grafické rozhranie sme zvolili GTK+, nakoľko spĺňa naše požiadavky, a tými sú multiplatformnosť, kompaktnosť a rýchlosť. Aj napriek tomu, že bol pôvodne vyvíjaný pre `X Window System` je možné ho využiť aj na platforme `Microsoft Windows` a `Quartz`.

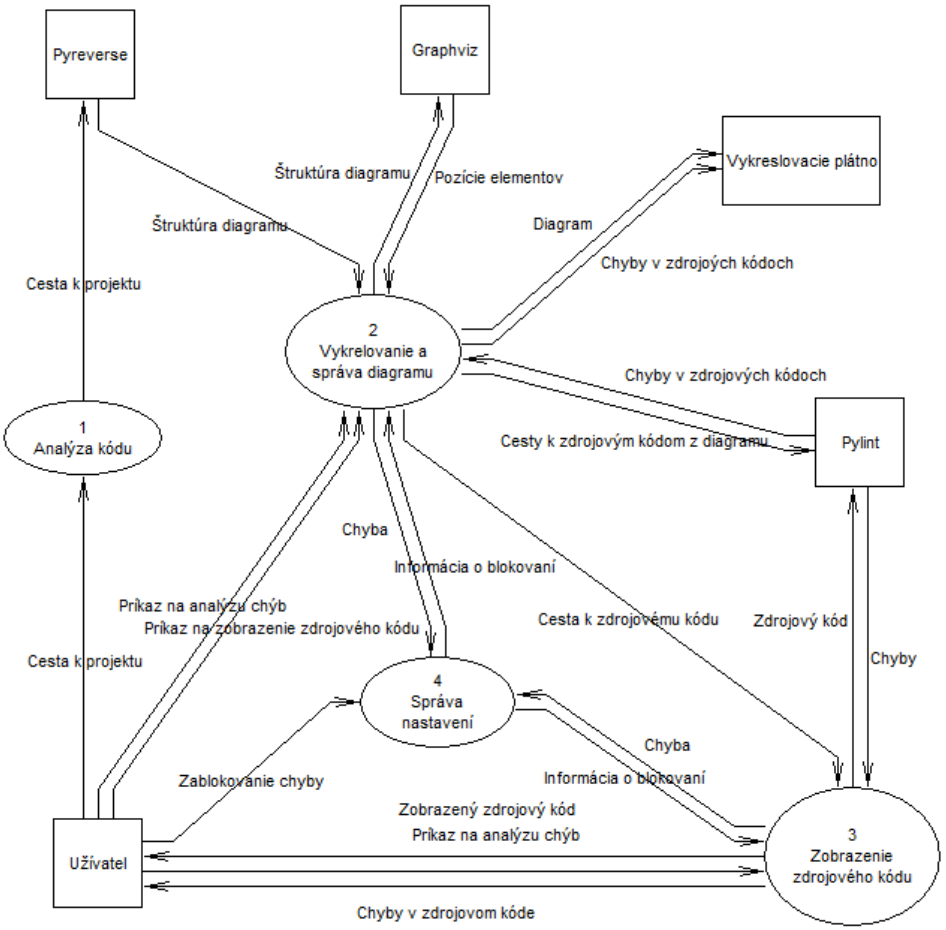
Backend systému bude zabezpečovať program Pylint, nakoľko je dostupný na všetkých platformách a je dlhodobo používaný a otestovaný. Fakt, že je implementovaný do viacerých populárnych IDE prostredí, potvrdzuje prítomnosť výborne navrhnutej API. Okrem generovania chýb poskytuje taktiež služby Pyreverse, vďaka ktorým bude možné naimplementovať generovanie diagramu tried a modulov.

Nakoľko bude ako grafické prostredie použitá knižnica GTK+, je dobrou voľbou použiť pre zobrazenie zdrojového kódu komponentu GtkSourceView [15], ktorú ponúka táto knižnica. Nakoľko ale nechceme obmedziť program len na jeden druh editoru, vytvoríme jednoduché rozhranie, pomocou ktorého integrujeme napríklad editor Vim a umožníme integrovať ostatné editory v prípade potreby. V prípade editorov je nutné, aby boli schopné zobrazovať na chybových miestach príslušné značky. Pri editore GtkSourceView využijeme jeho funkcionalitu značiek, pri editore Vim možnosť zvýraznenia určitých riadkov.

Pre zobrazenie grafu modulov a tried využijeme knižnicu Gaphas dostupnú pre GTK+. Knižnica nám pomôže vizualizovať výsledky analýzy. Jej jediným nedostatkom je neprítomnosť prvkov pre UML. Prvky pre zobrazenie tried, modulov a funkcií bude nutné doprogramovať manuálne.

## 5.3 Analýza a návrh

Nakoľko program využíva vnútornú štruktúru viacerých programov a knižníc, úlohou anlýzy je vhodne navrhnúť ich prepojenie. Nasledujúci DFD [22] diagram popisuje základný tok dát v aplikácii.




Obrázok 5.1: DFD diagram navrhnutej aplikácie

Pri spustení programu užívateľ zvolí cestu k projektu pomocou grafického rozhrania, alebo ako parameter na príkazovom riadku. Ten sa predá triede, ktorá používa metódy z knižnice Pyreverse. Vygenerovaná štruktúra reprezentujúca diagram sa predá knižnici Graphviz [13], ktorá rozhodne o pozíciách jednotlivých elementov diagramu tak, aby bol diagram čo najpre-

hľadnejší. Pre každý element v štruktúre diagramu sa vytvorí objekt knižnice Gaphas, ktorý sa vykreslí na plátne. Referencia na diagram a položky na plátne sú uložené ako kontext plátna pre neskoršie využitie. Pri spustení analýzy kódu projektu sa pomocou Pylint knižnice vygenerujú chybové hlásenia, prefiltrujú sa tak, aby sa užívateľovi nezobrazovali hlásenia, ktoré v minulosti zablokoval.

Rozpoznanie týchto chýb prebieha na základe typu chybového hlásenia, názvu objektu, na ktorom sa chyba vyskytla a čísle stĺpca. Nezohľadňujeme teda číslo riadku výskytu chyby, pretože táto informácia sa môže zmeniť pridaním nových riadkov pred výskyt chyby.

Následne sa pomocou kontextu plátna predajú informácie o počte nájdených chýb a informácie o nich priamo jednotlivým položkám na plátne a zabezpečí sa ich prekreslenie. Tieto položky si uchovávajú napríklad aj informácie o pozícii v súbore. Po kliknutí na položku plátna sa vytvorí nové okno s editorom podľa aktuálnych nastavení. Pri kontrole zdrojového kódu danej triedy sa pošle požiadavka danému editoru, ktorý sa stará o zobrazovanie chybových hlásení.

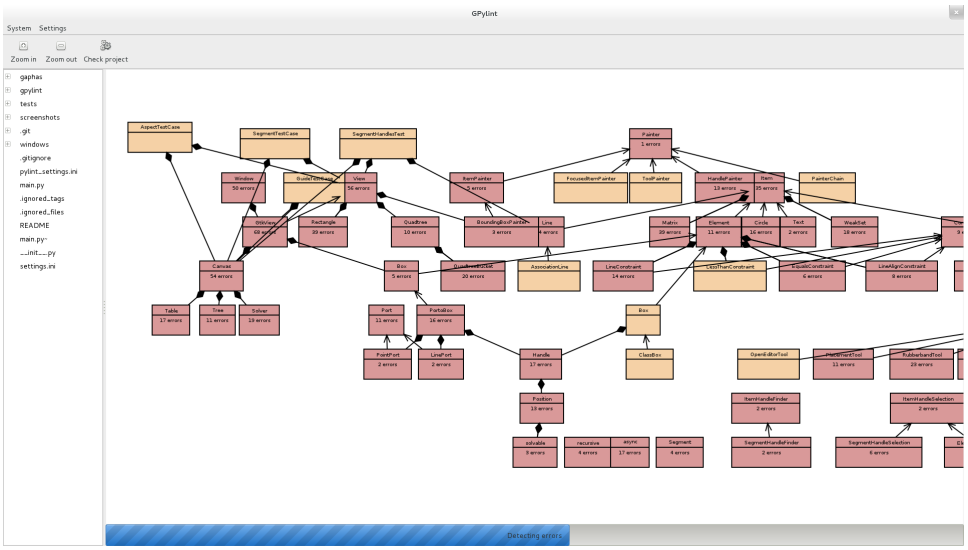
Do tejto chvíle sme naimplementovali dva typy editorov. Grafický editor sme naimplementovali pomocou komponenty GtkSourceView [15]. Editor Vim sme integrovali pomocou komponenty VteTerminal.

```
1 def open_file(self):
2     self.component = Vte.Terminal()
3     self.component.fork_command_full(Vte.PtyFlags.
          DEFAULT, None, ['vim', self.filepath], [], \
4         GLib.SpawnFlags.SEARCH_PATH, None, None)
5     self.component.show()
```

Program je implementovaný spôsobom využívajúcim API nástrojov Pylint, Pyreverse a Gaphas. Je teda závislý aj na ich korektnosti. Pri testovaní programu sme objavili napríklad chybu nástroja Pyreverse `www.logilab.org/ticket/92362`.

## 5.4 Aplikácia jednotlivých nástrojov

Pre požiadavky aplikácie sme potrebovali vytvoriť triedy využívajúce alebo upravujúce funkcionalitu jednotlivých nástrojov. V nasledujúcej podkapitole popíšeme využitie jednotlivých knižníc a ich implementáciu v našom projekte.




Obrázok 5.2: GPylint – Screenshot implementovaného nástroja

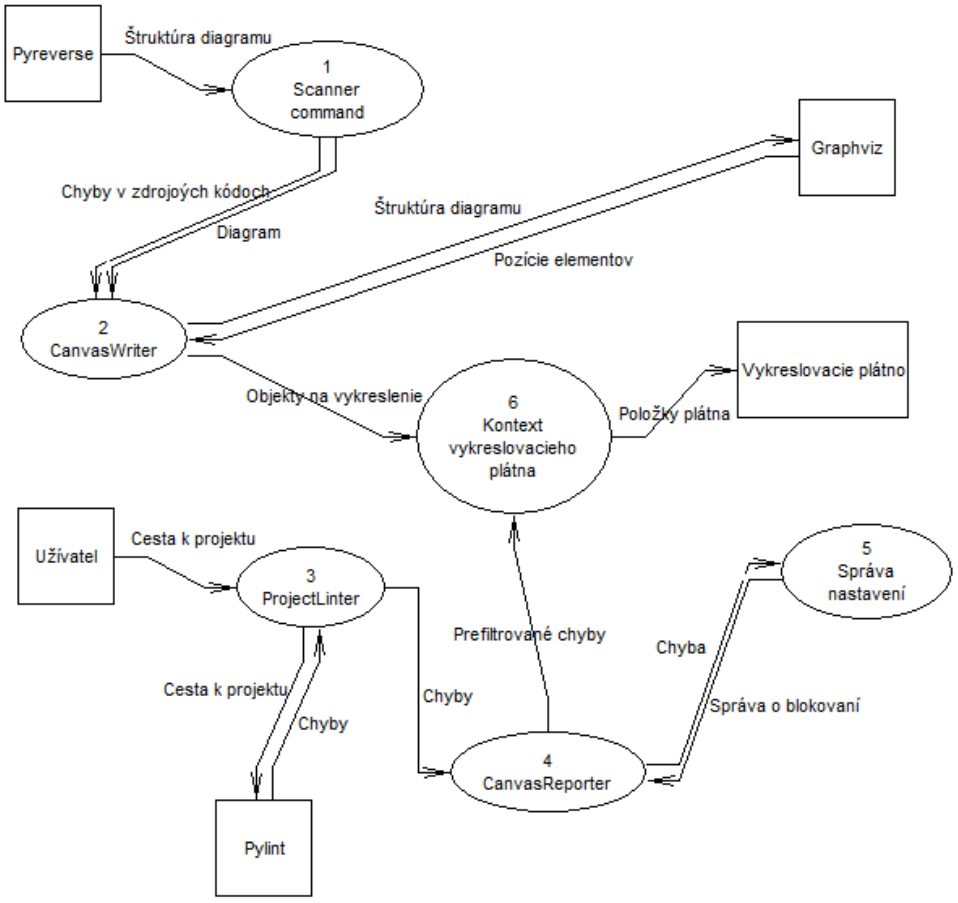
### 5.4.1 Aplikácia nástroja Pyreverse

Pri popise nástroja Pyreverse sme poukázali na hlavnú kostru programu Pyreverse, ktorá vracia diagram tried, predávaný triede zodpovednej za zápis. Vytvorili sme teda triedu `ScannerCommand`, ktorá spúšťa kostru programu Pyreverse v samostatnom vlákne, prefiltruje výsledky pre potreby aplikácie a výsledky predá zapisujúcej triede, ktorú naimplementujeme.

Triedu zodpovednú za zápis na plátno sme nazvali `CanvasWriter` (viď Obrázok 5.3, proces 2). Táto trieda implementuje metódu `get_values(self, obj)`, ktorá rozhoduje, aké informácie budú predané jednotlivým položkám na plátne. V tejto metóde teda rozhodneme, ktoré údaje od programu Pyreverse predáme k ďalšiemu spracovaniu. Inak len deleguje funkcionalitu na triedu `CanvasBackend`.

Trieda `CanvasBackend` rozširuje triedu `DotBackend` z programu Pylint. Všetky výstupy tak generuje do dot [14] formátu, ktoré v prekrytej metóde `generate(self, filename)` použijeme ako vstup pre knižnicu Gaphas.

Vďaka nej dostaneme pozície jednotlivých elementov na plátne tak, aby bol výsledný graf prehľadný a uložíme potrebné informácie do kontextu plátna. Daný graf následne vykreslíme na plátno.




Obrázok 5.3: DFD diagram procesu vykresľovania a správy diagramu

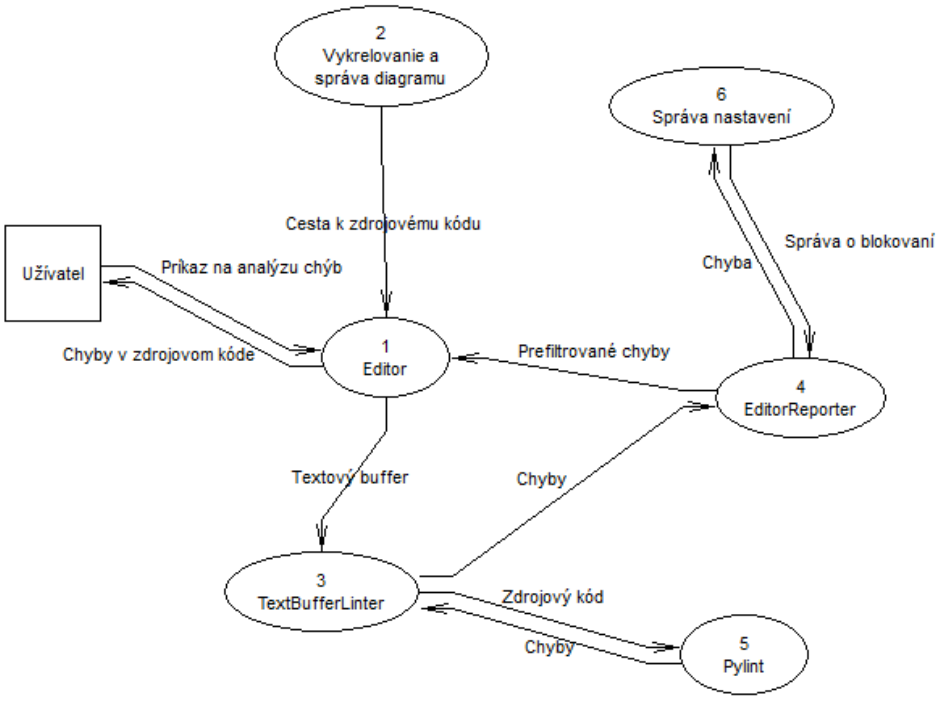
### 5.4.2 Aplikácia nástroja Pylint

Nástroj Pylint využívame dvomi rôznymi spôsobmi. Potrebujeme preskenovať celý projekt, aby sme chyby pre jednotlivé triedy vykreslili na plátno, a zároveň potrebujeme preskenovať práve otvorený súbor. Pre každý tento účel vytvoríme triedy implementujúce funkcionalitu triedy `PyLinter`, ktoré budú bežať v samostatnom vlákne.

Pri kontrole samostatného súboru funkcionalitu analýzy chýb rieši každý editor samostatne. V nasledujúcom odseku popíšeme spôsob, akým túto funkcionalitu implementuje grafický editor využívajúci komponentu GtkSourceView [15]. Ten vytvorí inštanciu triedy `TextBufferLinter` (viď Obrázok 5.4, proces 3), ktorú sme naimplementovali tak, aby jednotlivým

triedam kontrolujúcim kód predala obsah textového bufferu, ktorý obdržala pri inicializácii. Nakoľko každá inštancia triedy `PyLinter` potrebuje triedu zodpovednú za výstup výsledkov, naimplementovali sme triedu `EditorReporter` (viď Obrázok 5.4, proces 4).

Trieda `EditorReporter` po obdržaní chybovej správy skontroluje, či nie je užívateľom blokovaná (viď Obrázok 5.4, proces 6). Ak blokovaná nie je, vykreslí informácie o chybe v zdrojovom kóde.




Obrázok 5.4: DFD diagram procesu zobrazenia zdrojového kódu

Pri kontrole celého projektu sa vytvorí inštancia triedy `ProjectLinter`, ktorej sme nastavili triedu `CanvasReporter` ako triedu zodpovednú za výstup. Trieda `CanvasReporter` pri obdržaní správy skontroluje, či táto správa nie je blokovaná a následne ju nastaví jednotlivým položkám na plátne.

### 5.4.3 Aplikácia nástroja Gaphas

Nástroj Gaphas použijeme na vykresľovanie diagramu tried na plátno. Jeho jedinou nevýhodou je prítomnosť len základných elementov, ako je úsečka,

obdĺžnik alebo text. Vďaka výbornému návrhu knižnice Gaphas ich môžeme využiť na zostavenie vlastných elementov. Z triedy `gaphas.item.Element` odvodíme triedu `Box`, ktorá bude na plátne reprezentovať jednotlivé triedy prehľadávaného projektu. Bude taktiež uchovávať dodatočné informácie, ako je cesta k súboru triedy, jej názov, číslo riadku jej výskytu ale aj informácie o chybách, ktoré detekujeme.

Vzťahy medzi triedami vykreslíme za pomoci triedy `AssociationLine`, ktorá spája dve triedy `Box`. Podľa vzťahu medzi jednotlivými triedami vykreslí vzor typický pre špecializáciu alebo kompozíciu.

Jednotlivé komponenty reprezentujúce triedy môžeme po plátne presúvať. Z tohto dôvodu sme boli nútení vytvoriť väzbu, ktorá bude udržiavať asociačnú čiaru medzi danými dvomi triedami. Triedu sme nazvali `HandlesConstraint`. Pri pohybe komponenty si parametricky vyjadrí úsečku medzi stredmi spájaných objektov a nájde prieniky medzi ich hranami. Na dané prieniky umiestni konce spájanej úsečky.

Nakoľko chceme vytvoriť rozhranie čo najviac interaktívne, potrebujeme nástroj, ktorý po dvojkliku na položku plátna zobrazí jej zdrojový kód v nastavenom editore.

Knižnica Gaphas ponúka výborné rozhranie pre tvorbu vlastného nástroja. Nástroj `OpenEditorTool` odvodíme z triedy `gaphas.tool.Tool` a v metóde `on_double_click(self, event)` pristúpime k vykreslenej položke na plátne, nad ktorou je umiestnená myš. Z jej atribútov zistíme cestu k zdrojovému kódu a kód zobrazíme.

### 5.4.4 Aplikácia ostatných nástrojov

Pre potreby aplikácie potrebujeme využiť nasledujúce Python knižnice:

- ConfigParser
- cPickle
- argparse

Pre uloženie nastavení použijeme triedu ConfigParser [16], ktorá umožňuje bezproblémovú prácu s nastaveniami. Jej správanie sme pre aplikáciu prispôsobili vytvorením manažérov nastavení, ktoré implementujeme pomocou `Singleton` [23] návrhového vzoru. Takýmto spôsobom implementujeme nastavenia blokovania Pylint chybových hlásení, nastavenie použitého editora alebo uchovanie naposledy otvoreného projektu.

Vlastnosťou nášho nástroja je aj možnosť ignorovať konkrétne chyby na jednotlivých riadkoch, ktoré ale nechceme ignorovať globálne. Informácie o nich teda uložíme do dátového kontajnera, ktorý pomocou knižnice cPickle [17] serializujeme. Pri opätovnom spustení aplikácie prevedieme deserializáciu. Deserializovaný objekt neskôr použijeme na prefiltrovanie chybových správ.

Module argparse [18] použijeme na získanie informácií, ktoré užívateľ zadá na príkazovom riadku. Modul poskytuje veľmi jednoduché rozhranie na prácu s parametrami príkazového riadku, ako je napríklad automatická generácia nápovedy. Pomocou príkazového riadku bude môcť užívateľ zadať napríklad cestu k projektu, ktorý sa má otvoriť.

# 6 Záver

Našou úlohou bolo analyzovať nástroje na analýzu a vizualizáciu Python kódu a vytvoriť nástroj, ktorý poskytuje funkcionalitu, ktorú doteraz vytvorené nástroje neposkytujú.

Po predstavení základných vlastností jazyka Python sme preskúmali vlastnosti jeho introspekcie, ktoré nám vytvorili prehľad o možnosti programovo preskúmať vlastnosti jednotlivých objektov.

Neskôr sme si predstavili nástroje na analýzu kódu PEP8 [6], PyChecker [7] a Pylint [8] a zhodnotili sme, že najvhodnejší nástroj pre backend našej aplikácie je program Pylint, nakoľko je ľahko prispôsobiteľný a okrem detekcie chýb poskytuje funcionalitu vygenerovania UML diagramu. Navyše je široko používaný, či už ako samostatná utilita, alebo integrovaný do rôznych IDE modulov.

Analyzovali sme taktiež existujúce nástroje na vizualizáciu kódu a zhodnotili sme ich či už po stránke funkcionality, kompaktnosti, alebo iných kritérií.

Nakoniec sme preskúmali a vybrali nástroje, ktoré nám pomohli k ľahšej implementácii projektu a projekt sme navrhli a implementovali. Implementovaný program nám umožnuje zobraziť ľubovoľný Python projekt v podobe UML diagramu. Výsledný UML diagram sa výzorovo podobá na diagram vygenerovaný programom Graphviz [13]. Poskytuje nám okrem iného interaktívnosť a možnosť zobraziť chyby v rámci hierarchie modulov a tried. Pomocou nastavení máme možnosť ignorovať celé skupiny chýb, konkrétne typy chýb, či ignorovať chyby len na konkrétnych riadkoch. Zdrojové kódy môžeme upravovať v jednoduchých editoroch vizuálneho, alebo konzolového charakteru. Aj vďaka zvoleniu vhodných knižníc program veľmi dobre škáluje.

Pri implementácii sme narazili na viaceré problémy, medzi ktoré patrí napríklad chyba v programe Pyreverse, ktorá bude opravená v ďalšej verzii balíčka `logilab.astng.inspector`.

Medzi nevýhody programu môžeme zaradiť fakt, že slúži na analýzu len jedného programovacieho jazyka, alebo fakt, že niektorá funkcionalita, ako je napríklad filtrácia chýb, by mohla byť implementovaná na nižšej úrovni, aby bola prístupná aj spustením z príkazového riadku bez spustenia grafického rozhrania.

V budúcnosti sa program bude môcť spustiť na iných platformách, medzi ktoré patrí napríklad aj `Microsoft Windows`. Ďalej bude možné

viac prispôsobiť program pre jednoduchšie editovanie zdrojových kódov, akými sú napríklad klávesové skratky a iné vlastnosti pokročilých editorov.

# Literatúra


[1] PYTHON V2.7.3 DOCUMENTATION. *The Python Language Reference* [online]. Apr 24, 2012 [cit. 2012-04-24]. Dostupný z URL: `http://docs.python.org/reference/introduction.html`.

[2] PYTHON.ORG. *Package Index, Pylint 0.25.1* [online]. [cit. 2012-04-24]. Dostupný z URL: `http://pypi.python.org/pypi/pylint`.

[3] PYTHON.ORG. *Package Index, Gaphas 0.7.2* [online]. [cit. 2012-04-24]. Dostupný z URL: `http://pypi.python.org/pypi/gaphas`.

[4] GTK.ORG. *Gtk+ website* [online]. [cit. 2012-04-24]. Dostupný z URL: `http://www.gtk.org/`.

[5] PYTHON SOFTWARE FOUNDATION. *The Python Standard Library, Python Runtime Services* [online]. Apr 24, 2012 [cit. 2012-04-24]. Dostupný z URL: `http://docs.python.org/library/inspect.html`.

[6] PYTHON SOFTWARE FOUNDATION. *Python style guide checker* [online]. 1990 [cit. 12-11-2011]. Dostupný na URL: `http://pypi.python.org/pypi/pep8/`.

[7] PYCHECKER CONTRIBUTORS. PyChecker home page [online]. [cit. 12-11-2011]. Dostupný na URL: `http://pychecker.sourceforge.net/`.

[8] SYLVAIN, THENAULT. *Pylint home page* [online]. 27-09-2006 [cit. 12-11-2011] Dostupný na URL: `http://www.logilab.org/project/pylint`.

[9] PYCHECKER CONTRIBUTORS. *Pylint features* [online]. [cit. 2012-04-24]. Dostupný na URL: `http://www.logilab.org/card/pylintfeatures`.

[10] FREE SOFTWARE FOUNDATION, INC.. *GNU Lesser General Public License* [online]. [cit. 2012-04-24]. Dostupný na URL: `http://www.gnu.org/copyleft/lesser.html`.

[11] FREE SOFTWARE FOUNDATION, INC.. *Gnu General Public Licence* [online]. Dostupný na URL: `http://www.gnu.org/licenses/gpl-2.0.html`.

[12] GUIDO VAN ROSSUM, BARRY WARSAW. *Style Guide for Python Code* [online]. 05-07-2001 [cit. 12-11-2011] Dostupný na URL: http://www.python.org/dev/peps/pep-0008/.

[13] GRAPHVIZ CONTRIBUTORS. *Graphviz - Graph Visualization Software* [online]. [cit. 12-11-2011] Dostupný na URL: http://www.graphviz.org.

[14] GRAPHVIZ CONTRIBUTORS. *The DOT Language* [online]. [cit. 12-11-2011] Dostupný na URL: http://www.graphviz.org/content/dot-language.

[15] THE GTKSOURCEVIEW TEAM. *Gnome.org* [online]. [cit. 2012-04-24]. Dostupný na URL: http://projects.gnome.org/gtksourceview/.

[16] PYTHON SOFTWARE FOUNDATION. *Configuration file parser* [online]. [cit. 12-11-2011] Dostupný na URL: http://docs.python.org/library/configparser.html.

[17] PYTHON SOFTWARE FOUNDATION. *Python object serialization* [online]. [cit. 12-11-2011] Dostupný na URL: http://docs.python.org/library/pickle.html#module-cPickle.

[18] PYTHON SOFTWARE FOUNDATION. *Parser for command-line options, arguments and sub-commands* [online]. [cit. 12-11-2011] Dostupný na URL: http://docs.python.org/library/argparse.html#module-argparse.

[19] WIKIPEDIA CONTRIBUTORS. *ABC (programming language)* [online]. [cit. 12-11-2011] Dostupný na URL: http://en.wikipedia.org/wiki/ABC_%28programming_language%29.

[20] MOAP CONTRIBUTORS. *MOAP - Maintenance of a Project* [online]. [cit. 12-11-2011] Dostupný na URL: http://thomas.apestaart.org/moap/trac/.

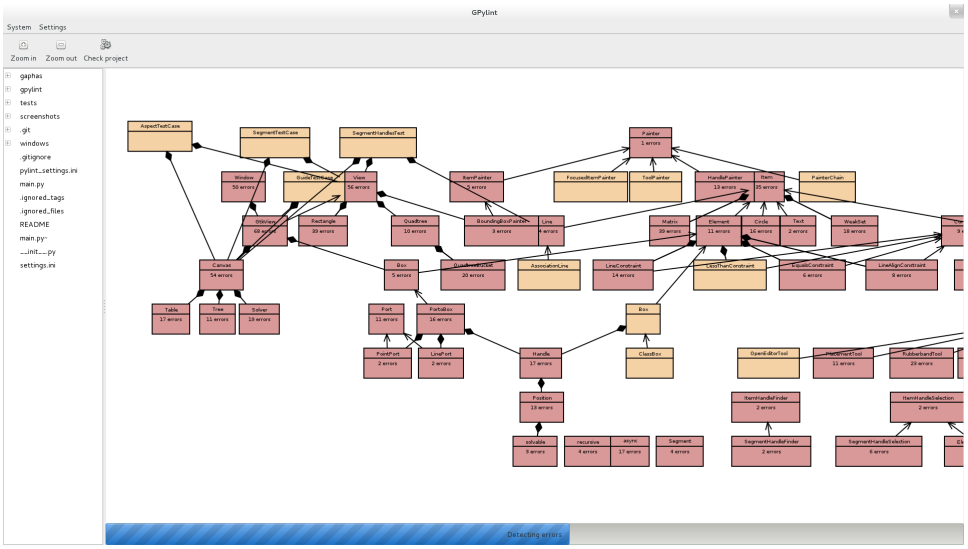
[21] PYDEV CONTRIBUTORS. *PyDev project* [online]. [cit. 12-11-2011] Dostupný na URL: http://http://pydev.org/.

[22] JAROSLAV RÁČEK. *Strukturovaná analýza systémů*. [cit. 12-11-2011]

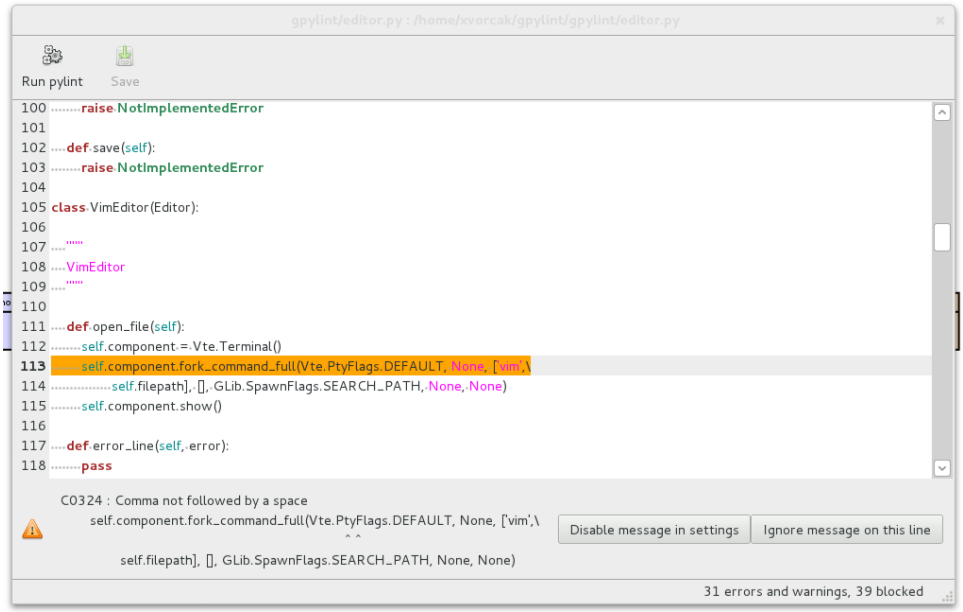
[23] WIKIPEDIA CONTRIBUTORS. *Singleton pattern* [online]. [cit. 12-11-2011] Dostupný na URL: http://en.wikipedia.org/wiki/Singleton_pattern.

[24] WIKIPEDIA CONTRIBUTORS. *Model-view-controller* [online]. [cit. 12-11-2011] Dostupný na URL: http://cs.wikipedia.org/wiki/Model-view-controller.

# A Screenshot grafického rozhrania

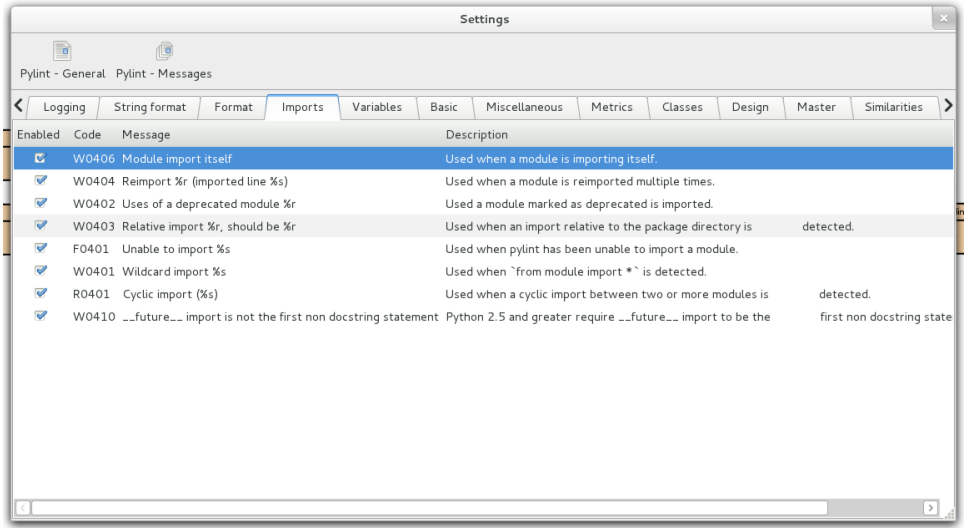



Obrázok A.1: Diagram projektu spolu s knižnicou Gaphas zobrazený na vykresľovacom plátne počas detekcie chýb




Obrázok A.2: Zobrazenie zdrojového kódu vo vizuálnom editore


Obrázok A.3: Zobrazenie zdrojového kódu v editore Vim




Obrázok A.4: Okno nastavení chybových hlásení

# B Obsah priloženého CD

- Aplikácia GPylint
  - *gpylint* – zložka obsahujúca Git repozitár programu
  - *logilab.patch* – patch, ktorý odstraňuje bug balíčku logilab.astng.inspector, `http://www.logilab.org/ticket/92362`
  - *gpylint/README* – súbor obsahujúci základné informácie pre beh programu
- Text bakalárskej práce
  - *thesis.tex* – text bakalárskej práce vo formáte latex s využitím šablóny fithesis
  - *thesis.pdf* – text bakalárskej práce vo formáte PDF
  - *images* – zložka obsahujúca obrázky použité v bakalárskej práci
  - *fithesis2.cls* – šablóna fithesis2