

Narzędzia programistyczne wspierające Pythona

Niniejszy artykuł powstał jako rezultat moich poszukiwań narzędzi, które wspomagałyby programistę piszącego w Pythonie. Znalezione narzędzia testowałem na projekcie, który wraz z trzema innymi studentami pisałem w ramach pracy licencjackiej na wydziale informatyki Uniwersytetu Warszawskiego. Projekt ostatnio urosł do około 30 tys linii i w związku z tym programowanie z pomocą prostych narzędzi stawało się coraz trudniejsze.

W ramach zajęć na uczelni poznałem wiele narzędzi dla Javy: skomplikowane i potężne IDE, programy do wyszukiwania bugów, do wyszukiwania miejsc podejrzanych o zły styl programowania itp. W związku z tym chciałem przekonać się czy analogiczne narzędzia istnieją dla Pythona. Byłem też przygotowany, że ze względu na inną specyfikę języka i mniejszą popularność w dużych projektach, wynik może nie być zadowalający.

Wszystkie znalezione programy są darmowe (innych nie szukałem), większość z nich można za darmo ściągnąć ze stosownych stron, chociaż niektóre wymagały pewnej pracy w dostosowywaniu do mojego projektu, podejrzewam, że podobnie będzie w przypadku innych. Stąd dodatkiem do artykułu jest plik .zip z lekko zmodyfikowanymi wersjami narzędzi lub prostymi skryptami które pokazują jak je stosować. Również w artykule sporo miejsca poświęciłem sprawom związanym z instalacją i ewentualnymi modyfikacjami omawianym programów.

Wymagania wstępne:

- Python 2.6,
- MS Windows,
- Jakiś program od rozpakowania archiwów *.zip.

Część narzędzi, zwłaszcza tych tworzonych przez pojedynczych entuzjastów, działa też na starszych wersjach Pythona, zazwyczaj właśnie do nich były tworzone, natomiast ich praca w nowej wersji wymaga więcej pracy. Wszystkie omawiane programy powinny też działać pod Linux-em (poza Notepad-em++).

1. Skróty

Wszystkie omawiane programy razem z linkami do ich stron oraz ewentualnym położeniem w załączonej paczce (pliku *.zip). Oprogramowanie które nie wymaga specjalnych modyfikacji nie zostało załączone w paczce, łatwiej je ściągnąć z Internetu.

Nazwa programu	Strona WWW	Położenie w paczce
Eclipse + Pydev	http://www.eclipse.org/downloads/ http://pydev.sourceforge.net/updates/	Brak
Notepad++	http://notepad-plus.sourceforge.net/uk/download.php	Brak
PyLint	http://www.logilab.org/project/pylint	Brak
Pyflakes	http://divmod.org/trac/wiki/DivmodPyflakes	/pyflakes
Pymetrics	http://sourceforge.net/projects/pymetrics/	/pymetrics
Statsvn	http://www.statsvn.org/downloads.html	/statsvn
depgraph	http://www.tarind.com/depgraph.html	/depgraph
Lumpy	http://www.greenteapress.com/thinkPython/swampy/lumpy.html	/lumpy

2. Edytory / IDE

Praktycznie każdy język z popularnych języków programowania ma jedno lub więcej popularnych IDE, natomiast nie znalazłem takiego środowiska dla Pythona. Na pytanie o najlepszy edytor na forach zazwyczaj pojawiają się „standardowe” odpowiedzi, np. Emacs czy gedit, czyli programy które nie są łatwe do opanowania i/lub nie zapewniają bardziej zaawansowanych funkcji: podpowiadanie nazw czy kontrola poprawności kodu. Zapewne wynika to z mniejszej popularności oraz samej specyfiki języka Python: dynamiczny charakter i łatwość użycia refleksji utrudniają automatyczną interpretację czy refactoring. Dalej opisuję dwa najlepsze edytory które osobiście znalazłem.

2.1. Eclipse + Pydev

Eclipse jest jednym ze standardowych IDE dla Javy, obsługuje też wiele wtyczek zarówno z dodatkowymi narzędziami dla Javy, jak i dającymi wsparcie dla innych języków.

W szczególności programiści Pythona powinni spróbować użyć wtyczki Pydev, która daje całkiem dobre wsparcie dla tego właśnie języka. Pełny spis funkcjonalności jest przedstawiony na stronie domowej projektu, jego skrót przedstawiam poniżej:

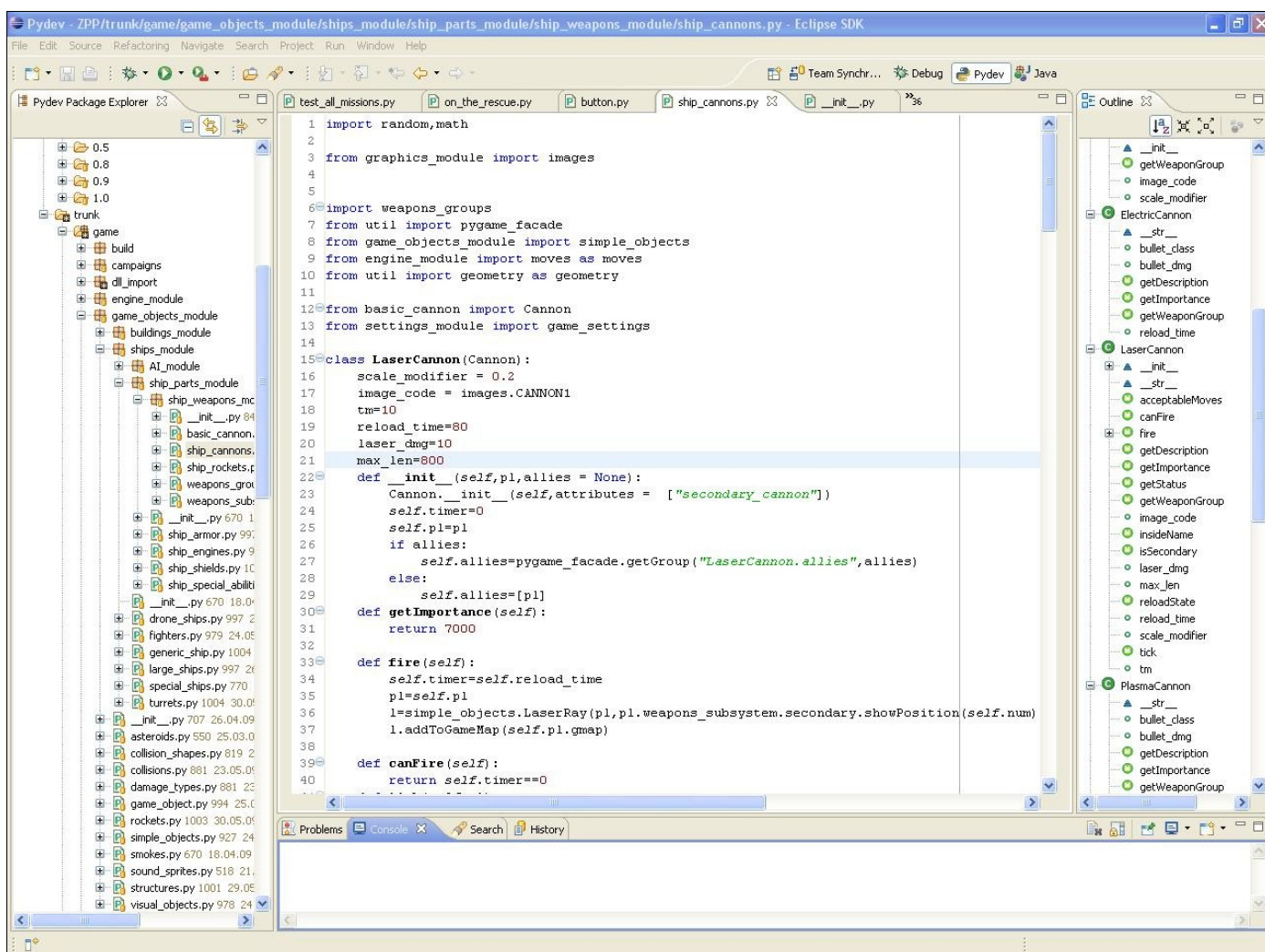
Funkcjonalności:

- Debugger analogiczny jak ten dla Javy, ale nie ma możliwości ustawienia breakpointu na wyjątku,
- Szybkie wyszukiwanie plików i napisów (np. nazw klas, skrótów Ctrl + H, Ctrl + Shift + R),
- Podpowiadanie nazw metod i pól w klasach, nazw zmiennych, klas w modułach itd. Niestety podpowiadanie nie wydaje się bardzo wyszukane, podaje tylko te nazwy co do których jest pewność, że są zdefiniowane. (skrót Ctrl + Spacja),
- Analogiczne wsparcie dla Jython-a,
- Podkreślanie składni, automatyczne wykrywanie błędów składniowych, opcje do porządkowania plików zawierających spacje i tabulatory,
- Automatyczne wcinanie kodu, zgodnie ze składnią,
- Zarządzanie pakietami i projektami, możliwość łączenia z innymi wtyczkami, np. Subclipse (wtyczka do SVN).

Sposób instalacji:

- Należy ściągnąć którąś z wersji Eclipse-a ze strony (wystarczy najprostsza/najmniejsza wersja),
- Rozpakować w wybranej lokalizacji, Eclipse nie wymaga instalacji,
- Uruchomić Eclipse-a, przez odpowiedni plik w podkatalogu \bin,
- Help -> Software Updates, Add site, dodać stronę PyDev-a (drugi link podany w tabelce),
- Dalej postępować zgodnie z instrukcjami, po instalacji można wybrać perspektywę PyDev, stworzyć nowy projekt i zacząć pisać.

Przykładowy screen:



Ilustracja 1: Mimo wszystko lepiej jest mieć szeroki ekran.

Uwagi:

- Im mniej refleksji w kodzie, tym lepiej Pydev (i pozostałe narzędzia) działają,
- Dalej jeszcze opiszę połączenie Pydev-a i Pylint-a.

2.2. Notepad ++

Nie jest to IDE, raczej zaawansowany edytor. Ale ma za to wsparcie dla bardzo dużego zbioru języków i działa na starszym sprzęcie o niższych parametrach. Bardzo dobry wybór dla osób które nie chcą pracować w złożonym i zasobożernym IDE, lub jako substytut Notatnika.

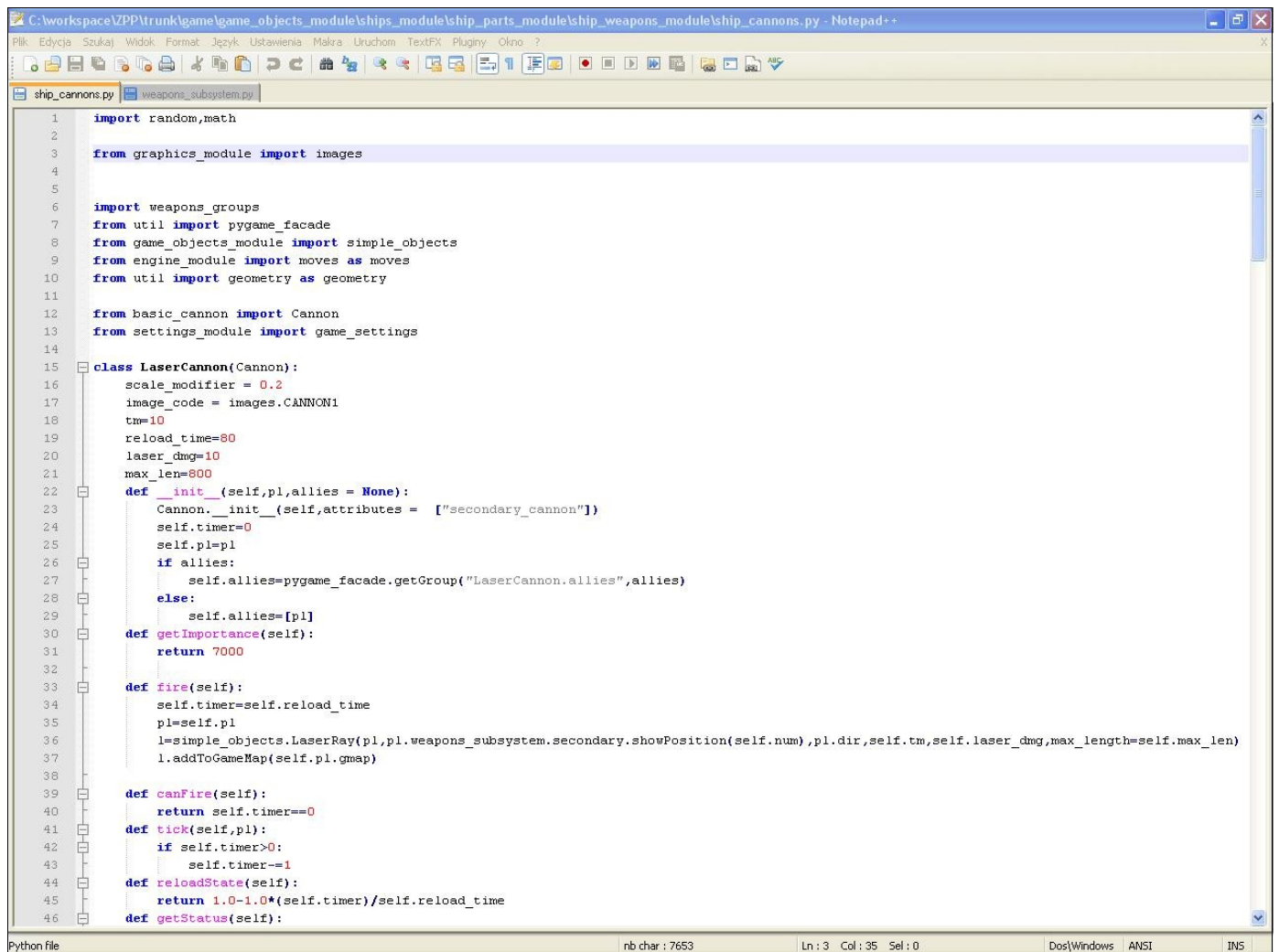
Funkcjonalności:

- Wsparcie dla wcięć i kolorowanie składni,
- Auto uzupełnianie, chociaż znacznie słabsze niż w Pydev-ie,
- Zakładki z wieloma otworzonymi plikami,
- Wyszukiwanie tekstu w wielu plikach, np. w całym drzewie projektu,
- Wsparcie również dla innych języków, edytor przydatny nawet jeśli ma się też Eclipse-a, do szybkiej edycji różnych plików.

Sposób instalacji:

- Ściągnąć i uruchomić instalator ze strony domowej projektu,
- Doradzam wybór pełnej integracji z Explorerem Windows, daje to opcję otwarcia Notepad++ w menu kontekstowym po naciśnięciu prawego przycisku na pliku.

Przykładowy screen:



```
1 import random,math
2
3 from graphics_module import images
4
5
6 import weapons_groups
7 from util import pygame_facade
8 from game_objects_module import simple_objects
9 from engine_module import moves as moves
10 from util import geometry as geometry
11
12 from basic_cannon import Cannon
13 from settings_module import game_settings
14
15 class LaserCannon(Cannon):
16     scale_modifier = 0.2
17     image_code = images.CANNON1
18     tm=10
19     reload_time=80
20     laser_dmg=10
21     max_len=800
22     def __init__(self,pl,allies = None):
23         Cannon.__init__(self,attributes = ["secondary_cannon"])
24         self.timer=0
25         self.pl=pl
26         if allies:
27             self.allies=pygame_facade.getGroup("LaserCannon.allies",allies)
28         else:
29             self.allies=[pl]
30     def getImportance(self):
31         return 7000
32
33     def fire(self):
34         self.timer=self.reload_time
35         pl=self.pl
36         l=simple_objects.LaserRay(pl,pl.weapons_subsystem.secondary.showPosition(self.num),pl.dir,self.tm,self.laser_dmg,max_length=self.max_len)
37         l.addToGameMap(self.pl.gmap)
38
39     def canFire(self):
40         return self.timer==0
41     def tick(self,pl):
42         if self.timer>0:
43             self.timer-=1
44     def reloadState(self):
45         return 1.0-1.0*(self.timer)/self.reload_time
46     def getStatus(self):
```

3. Kontrola jakości i poprawności kodu

W idealnym przypadku samo IDE po zauważeniu literówki czy też poważniejszego błędu powinno grzecznie :) uprzedzić programistę że jego wytwór może nie zadziałać. Dla Javy to dość często się udaje, Eclipse z paroma sprytnymi wtyczkami potrafi znajdować takie problemy, że programista nierzadko sam nie wpadł by na to że taki bug mógłby wystąpić. Oczywiście, w wielu przypadkach to podpowiadanie jest przesadzone, ale zazwyczaj warto się nad takimi miejscami zastanowić. Dla Pythona takim wykrywaczami podejrzanych fragmentów są dwa omówione niżej programy. Trzecim jest Pychecker, ale ja osobiście go nie testowałem.

3.1 Pylint

Nazwa Pylint nawiązuje do programu lint który spełniał podobną rolę dla języka C, jakieś 30 lat temu. Narzędzia tego można używać na dwa sposoby, albo jako niezależny program służący do testowania wybranych zbiorów plików (tak jak programy opisywane dalej), albo można go podłączyć do PyDev-a.

Ja osobiście wypróbowałem drugi sposób.

Funkcjonalności:

- Wyszukuje podstawowe błędy programistyczne, np. literówki,
- Wyszukuje też bardziej nietrywialne błędy i nakłania ostrzeżeniami do pisania bardziej czytelnego kodu, zarówno pod względem wizualnym jak i logicznym.

Przykład błędu nietrywialnego: Nie należy używać list jako parametrów domyślnych w metodach, bo taki parametr wylicza się tylko raz i ewentualna jego modyfikacja może wpłynąć na działanie metody przy kolejnym wywołaniu

- Zbiór ostrzeżeń można modyfikować, chociaż nie jest to zbyt wygodne,
- Wszystkie błędy/ostrzeżenia są zaznaczane przy odpowiadających im liniach w sposób standardowy dla Eclipse-a,
- Błędy mają swoje kody i krótkie opisy, na stronie Pylint-a jest do nich instrukcja.

Sposób instalacji (Instrukcja instalacji zakłada posiadanie już Eclipse-a + PyDev-a):

- Ściągnąć Pylint-a ze strony do wybranego katalogu na dysku, rozpakować,
- Standardowo zainstalować Pylint-a jako bibliotekę za pomocą:

```
python setup.py
```

- Wejść w ustawienia Pydev-a (Window → Preferences → Pydev → Pylint),
- Włączyć Pylint-a i podać ścieżkę do niego ścieżkę,
- Pylint powinien zacząć działać po przebudowaniu projektu.

Krótki help na stronie Pydev-a: <http://pydev.sourceforge.net/pylint.html>

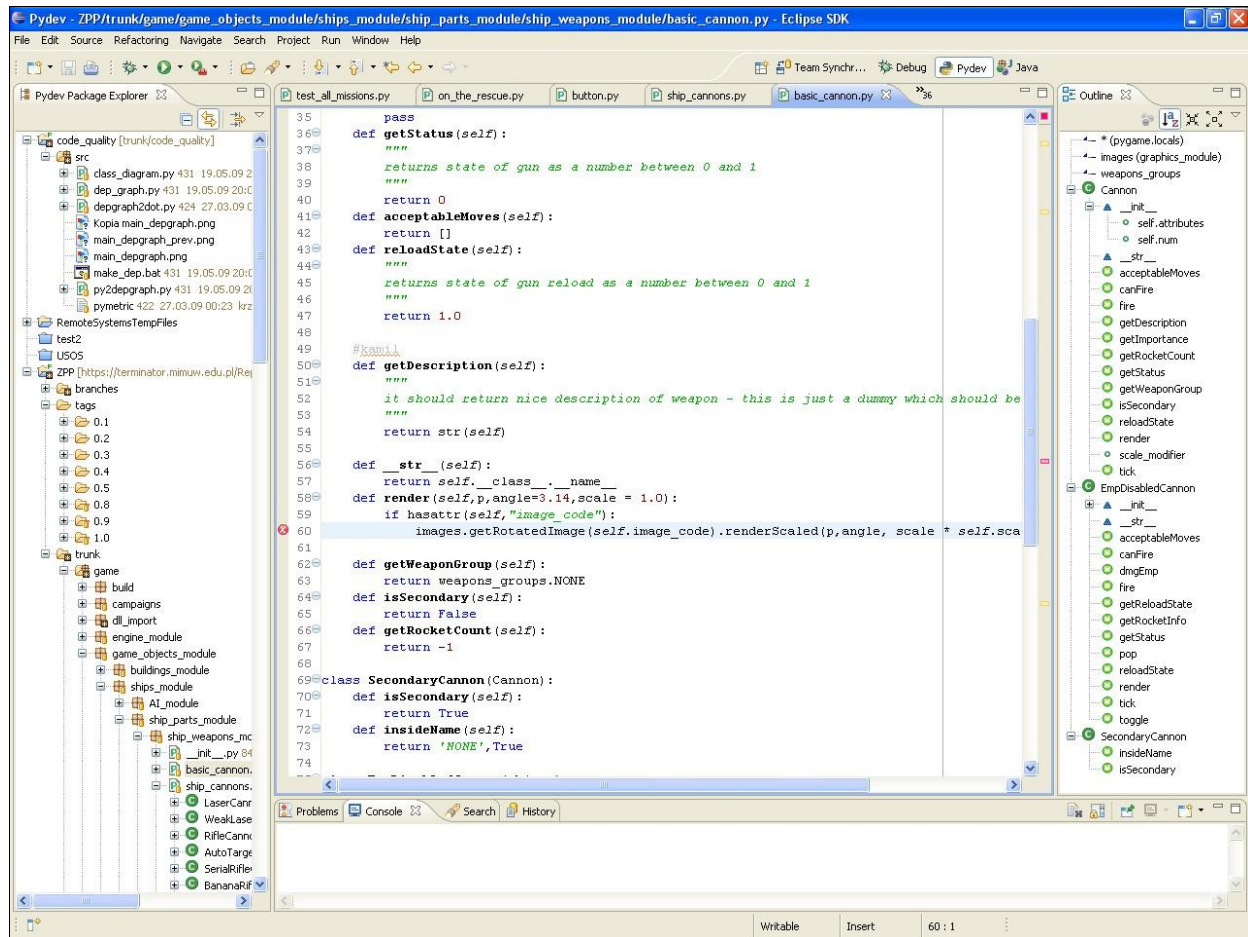
Powyższy help wspomina głównym problemie z używaniem Pylint-a - działa on raczej wolno. Dla domyślnych ustawień po zapisaniu jakiegokolwiek pliku Eclipse-a praktycznie się zawiesza na chwilę, jeśli sprawdzany projekt jest w miarę duży. Jeśli używamy SVN-a i nagle zmieni nam się wiele plików, to proces sprawdzania właściwie nie kończy się w rozsądnym czasie. Help opisuje jak sobie z tym radzić, dodatkowo można spróbować używać opcji „Only analyze open editors” w sekcji Builders. Dodatkową wadą Pylinta jest to, że nie widać czy już skończył analizę, jedynym wyznacznikiem jest fakt pojawienia się nowych błędów/ostrzeżeń.

Drugą wadą/zaletą Pylinta jest zbyt (jak na mój gust) „marudzenie” na temat stylu. W naszym projekcie na początku używaliśmy tabów zamiast spacji, na co pylint zaznaczył ostrzeżeniem każdą linię i praktycznie zamordował wydajnościowo Eclipse-a. To i inne denerwujące mnie ostrzeżenia wyłączyłem wpisując w okienko ustawień ciąg:

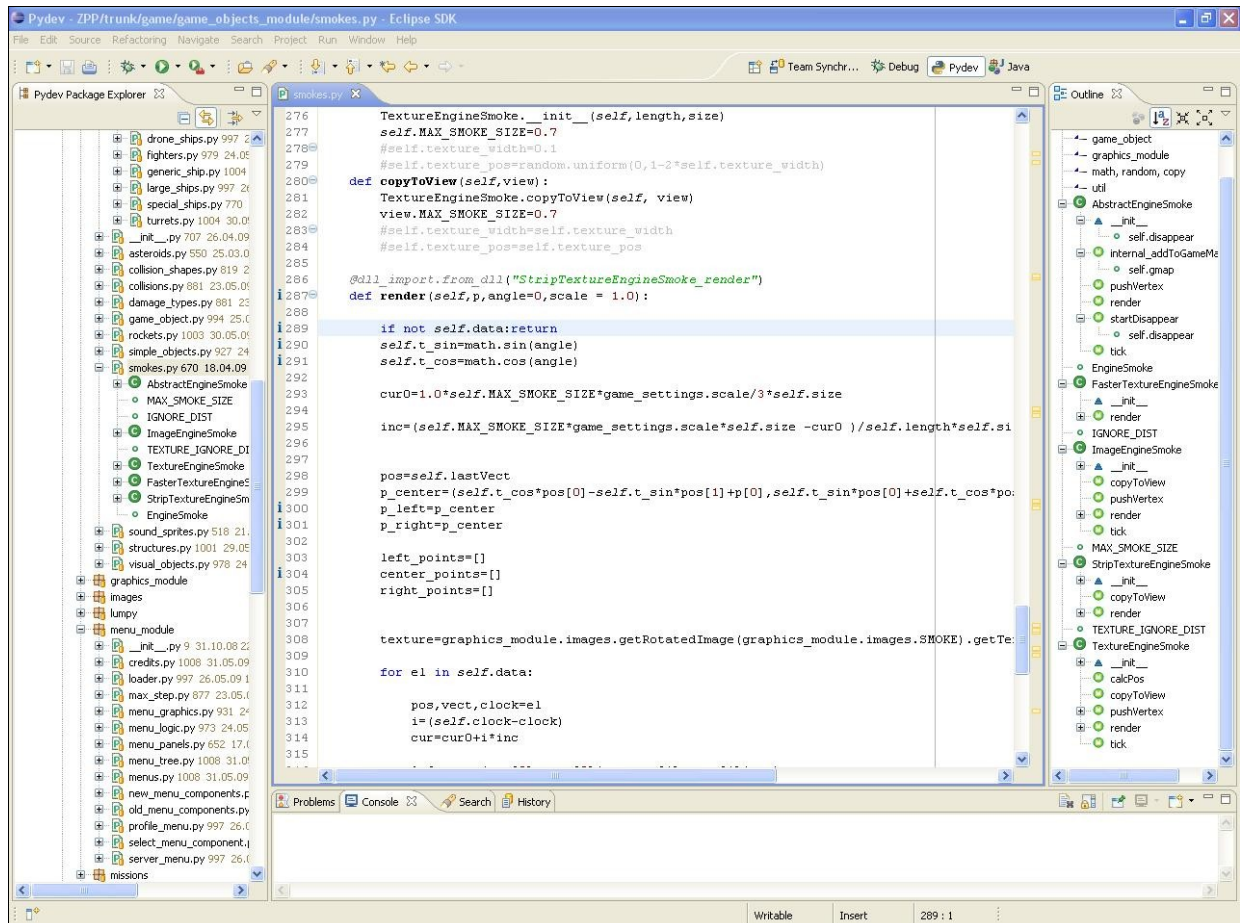
```
--indent-string=t      --disable-msg="W0312" --disable-msg="C0322"  
--disable-msg="C0103" --disable-msg="C0324" --disable-msg="C0301" --disable-msg="C0111"  
--disable-msg="R0201" --disable-msg="R0904" --disable-msg="R0903"
```

Dla projektów które bardziej dbają o styl, część z tych ustawień może być niepotrzebna, pozostałym może uda się oszczędzić trochę czasu na szukaniu tych opcji. Polecam sprawdzenie na stronie Pylint-a co właściwie wyłączamy i wybrać zbiór optymalny dla danego zastosowania.

Przykładowe screeny:



Ilustracja 3: Czepia się, że `self.image_code` jest niezdefiniowany. A `hasattr` nie zauważył! :



Ilustracja 4: A tu znowu parę warningów.

3.2 Pyflakes

Prostsze (i znacznie szybsze) narzędzie.

Funkcjonalności:

- Analizuje podany zbiór plików, bez ich uruchamiania/importowania,
- Wykrywa użycie niezdefiniowanych zmiennych,
- Wykrywa nadpisanie metody przez zdefiniowanie jej wiele razy,
- Wykrywa nadmiarowe i brakujące importy,
- Ostrzega o użyciu importów z *.

Instalacja/użycie:

- Należy ściągnąć lub zainstalować z paczki package Pyflakes.

Dla testowanego przeze mnie projektu najłatwiejszą metodą użycia Pyflakes-a było uruchomienie prostego skryptu w Pythonie który sprawdzał wszystkie pliki podane jako argumenty w linii poleceń. Znajduje się on w załączonej paczce w katalogu Pyflakes. Taki sposób szukania i poprawiania kodu jest o tyle wygodny że łatwo ograniczyć liczbę sprawdzanych plików (gdy wrzucimy zbyt dużo naraz to potem trudniej analizuje się, które pliki są już poprawione) a jednocześnie nie trzeba ich podawać pojedynczo. Warto mieć przy tym zainstalowanego Bash-a (*/*.py bardzo pomaga :)), również pod Windows.

```
python run_pyflakes.py lista_plikow
```

Pomimo, że Pylint potrafi znaleźć znacznie więcej błędów, trudno jest go używać stale ze względu na długi czas czekania na odpowiedź. Natomiast uruchomienie Pyflakes-a na całkiem sporym projekcie jest bardzo szybkie. Pyflakes wypisuje tylko błędy i ostrzeżenia, więc nie ma potrzeby tworzenia dodatkowego narzędzia do przetwarzania danych otrzymywanych z narzędzia. Jeśli błędów nie ma, to wyjście jest puste.

4. Obliczenia metryk dla kodu

4.1 Pylint

Jak się okazuje, Pylint uruchomiony z konsoli sam liczy podstawowe metryki dla sprawdzanego pliku. Jeśli używamy Pydeva, możemy je obejrzeć po zaznaczeniu odpowiedniej opcji w jego konfiguracji (Redirect pylint output to console). Metryki mają kilka wad:

- Pylint generuje całkiem sporo tekstu, zwłaszcza jeśli coś mu się nie spodoba i trudno cokolwiek znaleźć,
- Metryki są naprawdę podstawowe (np. procent komentarzy),
- Włączenie powyższej opcji jeszcze bardziej spowalnia Eclipse-a.

Z drugiej strony można się dowiedzieć jak w skali 0/10 pylint ocenia nasz kod. Ogólnie im więcej błędów tym niższy wynik (osobiście udało mi się uzyskać wynik -54/10 przy źle skonfigurowanym Pylint-ie i dość długim pliku). Zobaczmy dalej co potrafią narzędzia stworzone wprost z myślą o liczeniu statystyk dla kodu.

4.2 Pymetrics

Narzędzie do obliczania różnych metryk dla pojedynczych plików źródłowych (znowu, brak funkcji agregujących z wielu modułów).

Funkcjonalności:

- Wypisuje wartości metryk dla wybranego modułu,
- Podaje metrykę McCabe-a, SLOC i podstawowe (np. liczba linii),
- Jest możliwość łatwego definiowania własnych metryk (a przynajmniej tak twierdzi autor),
- Wyniki są zapisywane jako CSV i SQL, co powinno pomóc przy generowaniu raportów dla większego projektu.

Instalacja/użycie:

Pomimo, że narzędzie teoretycznie działa na dowolnym systemie operacyjnym, twórcy popełnili bardzo prosty błąd uniemożliwiający bezpośrednią instalację pod systemem Windows: w zamieszczonym na stronie archiwum znajduje się plik i katalog o nazwach różniących się tylko wielkością liter. Aby umożliwić łatwą instalację pod Windowsami, w archiwum dołączonym do tego artykułu zamieściłem katalog /pymetrics z rozpakowaną zawartością i nazwą pliku „pymetrics” zmienioną na „win_pymetrics.py”.

- W katalogu zawierającym plik win_pymetrics.py (albo oryginalny Pymetrics) uruchamiamy z konsoli:

python win_pymetrics.py pliki_do_analizy

- Uruchomienie programu bez parametrów daje listę dodatkowych opcji.

Przykładowe screeny:

```
C:\WINDOWS\system32\cmd.exe - bash
- SingleLocalGame

McCabe Complexity Metric for file ../game/engine_module/game.py

1   AbstractGame.__init__
3   AbstractGame.__pauseLoop
2   AbstractGame.changeSoundMode
5   AbstractGame.doCalculations
2   AbstractGame.doOutput
1   AbstractGame.freeze
1   AbstractGame.frozeed
1   AbstractGame.getDisplay
1   AbstractGame.getKeyMapping
1   AbstractGame.getMap
1   AbstractGame.get_changes
11  AbstractGame.handleEsc
1   AbstractGame.handleEsc.doIt
13  AbstractGame.on_key_press
2   AbstractGame.on_key_release
1   AbstractGame.prepareGraphics
1   AbstractGame.prepareKeyboard
1   AbstractGame.prepareMoves
6   AbstractGame.run
2   AbstractGame.runHook
1   AbstractGame.setGmap
1   AbstractGame.setNoTickWait
3   AbstractGame.showFPS
7   AbstractGame.tick
1   AbstractGame.unfreeze
1   DeathMatchClient.__init__
1   DeathMatchClient.doCalculations
1   DeathMatchClient.getDisplay
1   DeathMatchClient.get_changes
3   DeathMatchClient.on_key_press
1   DeathMatchClient.runHook
1   DeathMatchClient.setGmap
1   DeathMatchClient.updateInitialState
1   DeathMatchServer.__init__
3   DeathMatchServer.cleanup
7   DeathMatchServer.doCalculations
1   DeathMatchServer.doOutput
1   DeathMatchServer.getDisplay
1   DeathMatchServer.getInitialState
3   DeathMatchServer.get_changes
1   DeathMatchServer.prepare
1   DeathMatchServer.prepareGraphics
1   DeathMatchServer.prepareKeyboard
1   DeathMatchServer.set_options
1   SingleLocalGame.__init__
1   SingleLocalGame.run
1   __main__

COCOMO 2's SLOC Metric for ../game/engine_module/game.py
```

```
C:\Zaznacz C:\WINDOWS\system32\cmd.exe - bash

25.00 %ClassesHavingDocStrings
4.50 %Comments
0.68 %CommentsInline
13.04 %FunctionsHavingDocStrings

Functions DocString present(+) or missing(-)

- AbstractGame.__init__
+ AbstractGame.__pauseLoop
- AbstractGame.changeSoundMode
- AbstractGame.doCalculations
- AbstractGame.doOutput
- AbstractGame.freeze
- AbstractGame.frozeed
- AbstractGame.getDisplay
+ AbstractGame.getKeyMapping
+ AbstractGame.getMap
- AbstractGame.get_changes
- AbstractGame.handleEsc
- AbstractGame.handleEsc.doIt
- AbstractGame.on_key_press
- AbstractGame.on_key_release
- AbstractGame.prepareGraphics
- AbstractGame.prepareKeyboard
- AbstractGame.prepareMoves
+ AbstractGame.run
+ AbstractGame.runHook
- AbstractGame.setGmap
+ AbstractGame.setNoTickWait
- AbstractGame.showFPS
- AbstractGame.tick
- AbstractGame.unfreeze
- DeathMatchClient.__init__
- DeathMatchClient.doCalculations
- DeathMatchClient.getDisplay
- DeathMatchClient.get_changes
- DeathMatchClient.on_key_press
- DeathMatchClient.runHook
- DeathMatchClient.setGmap
- DeathMatchClient.updateInitialState
- DeathMatchServer.__init__
- DeathMatchServer.cleanup
- DeathMatchServer.doCalculations
- DeathMatchServer.doOutput
- DeathMatchServer.getDisplay
- DeathMatchServer.getInitialState
- DeathMatchServer.get_changes
- DeathMatchServer.prepare
- DeathMatchServer.prepareGraphics
- DeathMatchServer.prepareKeyboard
- DeathMatchServer.set_options
- SingleLocalGame.__init__
- SingleLocalGame.run
```

```

Zaznacz C:\WINDOWS\system32\cmd.exe - bash
FontTextureCreator find_game_deps.sh muzyka-propozycje.txt wymagania.txt
[root 01:36:11 /workspace/ZPP/trunk/pymetrics-0.8.1]# python win_pymetrics ../game/engine_module/*.py
=== File: ../game/engine_module/__init__.py ===
Basic Metrics for module ../game/engine_module/__init__.py
-----
1 numBlocks
1 numCharacters
1 numLines
3 numTokens

McCabe Complexity Metric for file ../game/engine_module/__init__.py
-----
1 __main__

COCOMO 2's SLOC Metric for ../game/engine_module/__init__.py
-----
0 ../game/engine_module/__init__.py

=== File: ../game/engine_module/campaign.py ===
Module ../game/engine_module/campaign.py is missing a module doc string. Detected at line 1
In file ../game/engine_module/campaign.py, function DefaultCampaign.runMission has 1 extra exit at line 44
Basic Metrics for module ../game/engine_module/campaign.py
-----
4 maxBlockDepth
39 numBlocks
6019 numCharacters
3 numClasses
4 numComments
17 numFunctions
124 numKeywords
131 numLines
1 numMultipleExitFens
1255 numTokens
3.05 %Comments

Functions DocString present(+) or missing(-)
-----
- CampaignList.__init__
- CampaignList.getCampaign
- CampaignList.getMenu
- DefaultCampaign.__init__
- DefaultCampaign._updateMissions
- DefaultCampaign.getCampaignState
- DefaultCampaign.getDescription
- DefaultCampaign.getMenu
- DefaultCampaign.getMission

```

4.3 Statsvn

Co prawda to narzędzie nie jest przeznaczone dla Pythona, a raczej dla wszystkich projektów korzystających z repozytorium Subversion, wydaje mi się, że warto je tutaj omówić. Dla osób nie znających/używających SVN, polecam Wikipedię + stronę:

<http://subversion.tigris.org/>

Funkcjonalności:

- Graficzna (w postaci strony WWW + wykresy/diagramy) reprezentacja różnego rodzaju statystyk dla projektu w repozytorium SVN. Naprawdę warto zobaczyć co ciekawego wie o nas nasze repozytorium :),
- Ilustracja zmian tych parametrów w czasie, no wykres liczby linii kodu czy liczby plików w repozytorium w zależności od czasu (domyślnie od chwili zerowej rewizji),
- Statystyki dla poszczególnych programistów, procentowy udział w dodawanych i zmienianych liniach kodu itp.

Instalacja/użycie:

- Należy ściągnąć plik *.jar ze strony projektu z tabelki,
- Wykonać SVN update na katalogu dla którego chcemy dostać statystyki (najlepiej trunk),
- Utworzyć log z SVN w postaci pliku XML, np. za pomocą polecenia:

```
svn --xml --verbose log > log.xml
```

- Uruchomić w wybranym przez siebie katalogu generowanie strony za pomocą polecenia:

```
java -jar sciezka_do_statsvn/statsvn.jar sciezka_do_logu/log.xml sciezka_do_logu
```

- Można otworzyć utworzoną stronę w przeglądarce, strona tytułowa to (standardowo) index.html,
- Za pierwszym razem generowanie statystyk trwa dość długo, później, przy aktualizowaniu strony, program wylicza tylko różnice względem poprzedniego stanu.

Skrypt który wykonuje powyższe kroki, zapisany w bashu, jest zamieszczony w paczce, w katalogu statsvn. Jeśli log z SVN zostanie utworzony w innym katalogu niż katalog dla którego chcemy mieć statystyki (a zarazem którego ten log dotyczy), to ten właśnie katalog należy podać jako drugą ścieżkę przy uruchomieniu jara. W razie problemów wraz z jarem autorzy tego narzędzia dają krótkie readme.

5. Dokumentacja kodu

Nie będę omawiał standardowego narzędzia – pydoc. Skupię się zamiast tego na dwóch ciekawych projektach, które mają na celu wizualizację zależności w tworzonym kodzie.

Niestety oba narzędzia mają swoje wady – w dużej mierze polegają na wewnętrznych mechanizmach interpretera Pythona i w związku z tym często pojawiają się nieoczekiwane problemy (zwłaszcza, że narzędzia te były pisane pod już dawno nieaktualne interpretery).

5.1 depgraph

Wizualizacja zależności pomiędzy modułami w projekcie.

Funkcjonalności:

- Rysuje graf modułów, krawędzie odpowiadają importowaniu jednego modułu przez inny,
- Poszczególne katalogi są rysowane różnymi kolorami, co ułatwia znalezienie importów które nie wydają się mieć sensu, modułów umieszczonych w złych pakietach itp.,
- Nie wymaga uruchomienia programu w trakcie generowania grafu (ale wymaga żeby istniały odpowiednie pliki *.pyc),
- Domyślnie nie rysuje importów z i do pakietów (w sensie plików __init__.py),
- Trzeba się mocno postarać żeby na grafie nie pojawiły się biblioteki używane przez nasz projekt.

Instalacja/użycie:

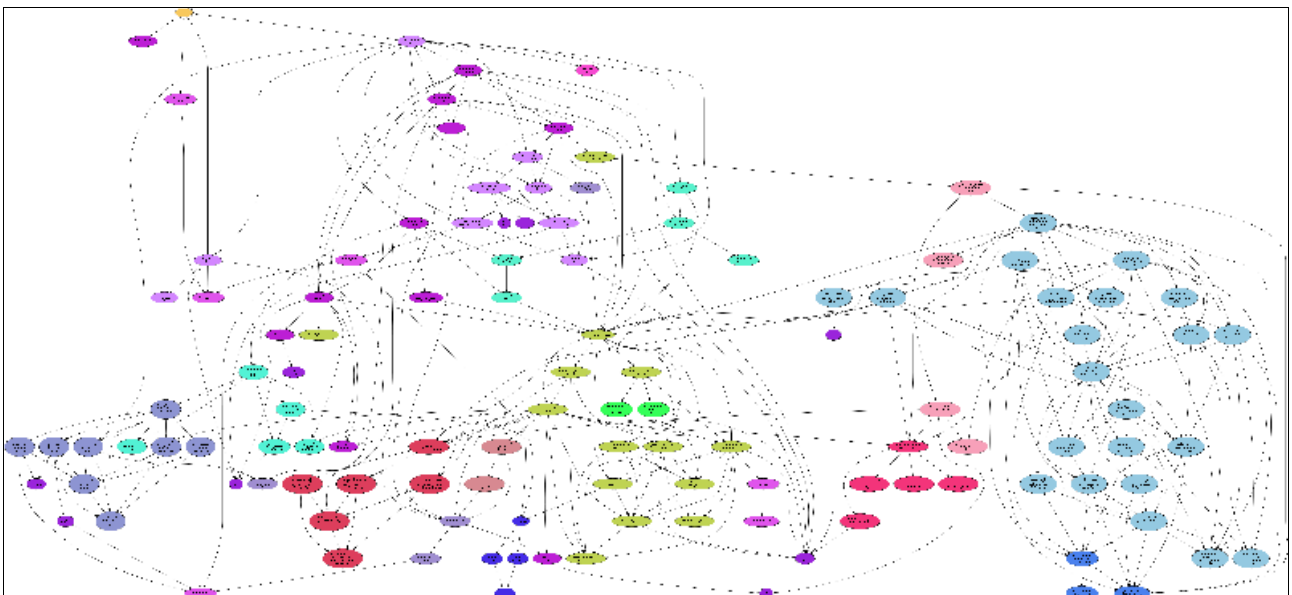
- Należy ściągnąć program Graphviz (<http://www.graphviz.org/>),
- Instrukcja obsługi depgraph podana na stronie autorów praktycznie nie działa pod Windowsami, a przynajmniej nie jest prostą ją do tego zmusić, polecam więc użycie wersji z mojego archiwum,
- znajdujemy plik w naszym projekcie który będzie korzeniem grafu (zazwyczaj główny plik wykonywalny).

```
python dep_graph.py sciezka_do_pliku
```

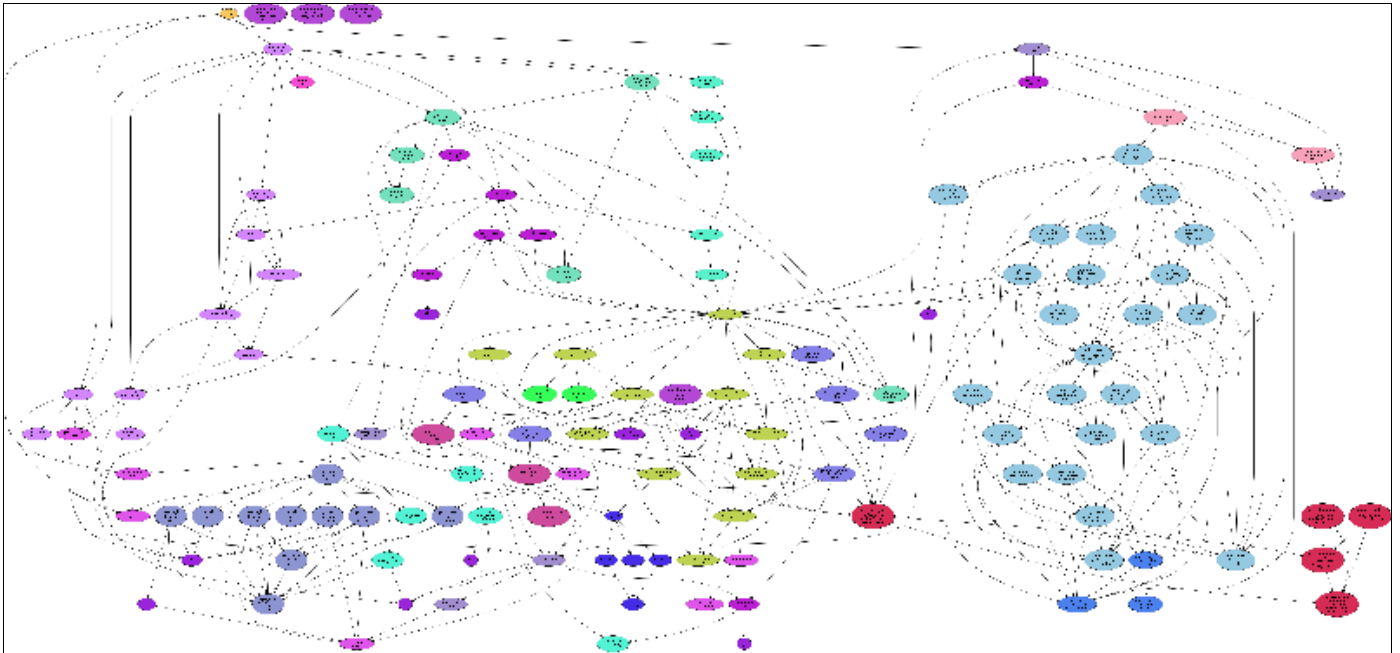
- W katalogu powinien wygenerować się plik main_depgraph.png. Plik ten jest często całkiem spory i stanowi wyzwanie dla większości przeglądarek grafiki. W projekcie nad którym pracowałem miał on wielkość około 7000x3000 pikseli,
- Oglądamy nasz graf szukając modułów które nas nie interesują. Jeśli takowe znajdziemy, dopisujemy je do listy zakazanych (gdzieś koło 65 linijki pliku dep_graph.py),
- Jeśli z drugiej strony widzimy że brakuje pakietów które znowu importują inne moduły i przez nasz graf się rozspójnia, to dopisujemy je do listy widocznych pakietów w linii 49.

Czasem niestety nawet takie ręczne poprawianie nic nie daje i niektóre moduły „wiszą” w powietrzu.

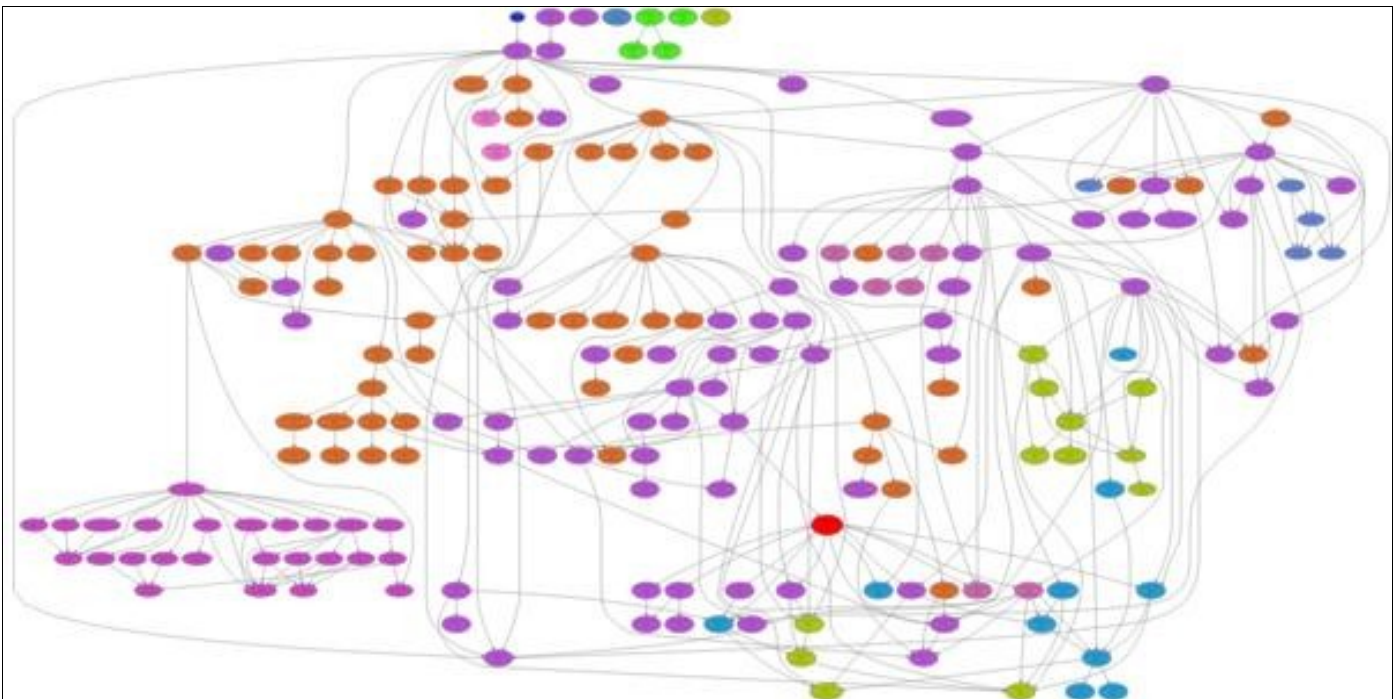
Przykładowe wygenerowane grafy (Obrazki w oryginalnym rozmiarze są dostępne w archiwum):



Ilustracja 8: Diagram modułów naszego projektu, w sumie bardzo źle nie jest.



Ilustracja 9: Niestety po zmianach w strukturze pakietów uzyskaliśmy trochę sierot, widać je w rzędku po lewej u góry, żadne próby zmuszenia depgraph-a do ich połączenia nie pomogły.



Ilustracja 10: A oto graf przykładowy ze strony autorów, widać że im też nie wszystko się do końca udało.

5.2 Lumpy

Narzędzie Lumpy , w oryginalnej wersji, służy do tworzenia diagramów obiektów i klas w notacji zbliżonej do UML.

Tworzenie diagramów obiektów dla dużych projektów tworzy bardzo złożone i nieczytelne obrazy, podobnie staje się gdy próbuje się zaznaczyć pola i metody obiektów. Stąd wersja umieszczona w załączonej paczce wypisuje tylko i wyłącznie diagramy klas z nazwami klas i strzałkami które oznaczają dziedziczenie.

Niestety utworzenie diagramu klas wymaga zaimportowania interesujących nas modułów. Czasami powoduje to niechciane skutki uboczne, jeśli w modułach ktoś umieścił statyczny kod uruchamiany przy imporcie.

Załączony moduł do_lumpy.py ma możliwość stworzenia diagramu dla zadanego (w kodzie) modułu, lub zbioru modułów, dla całego pakietu (katalogu). Wystarczy zastąpić wpisany tam jako przykład pakiet lumpy ścieżką do interesującego nas katalogu.

Można też ręcznie zaimportować wybrane moduły (from moduł import *), wówczas lumpy wypisze tylko klasy z tych modułów oraz ich nadklasy itd. Ważne jest aby uczynić to pomiędzy odwołaniami do lumpy.make_reference() oraz lumpy.class_diagram().

Jeśli chcemy zobaczyć też pola i metody zdefiniowane w klasach, należy zastąpić plik w module lumpy przez te dostarczone przez

autorów (ich kopia znajduje się w katalogu oryginal_lumpy). Można też zmienić tylko nazwę pakietu importowanego na początku pliku do_lumpy.

Funkcjonalności:

- Rysuje hierarchię dziedziczenia klas,
- Ma możliwość zaznaczania pól i metod w klasach,
- Ma możliwość rysowania diagramu obiektów w danym momencie działania programu.

Instalacja/użycie:

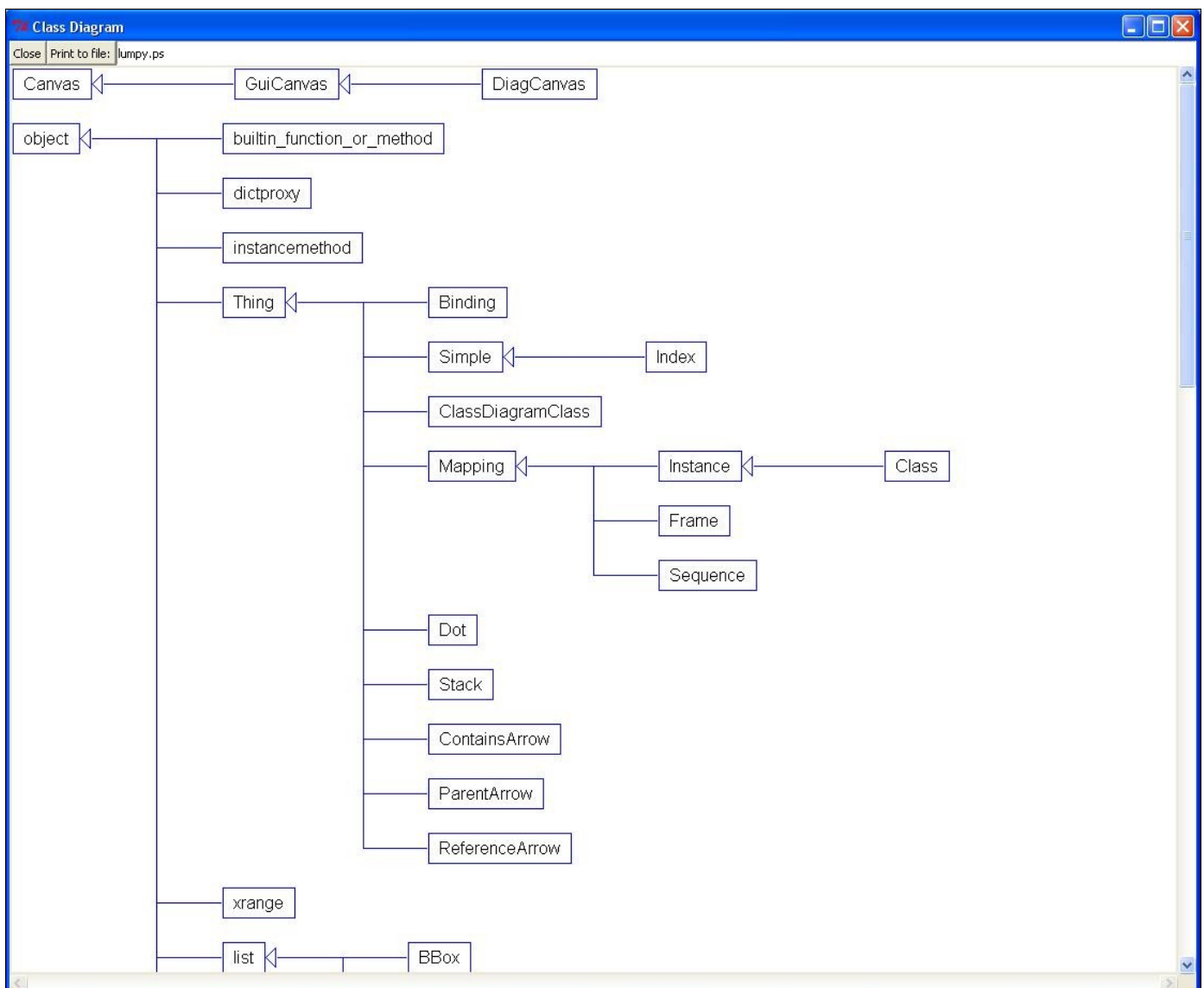
- Zmodyfikować plik do_lumpy.py zgodnie z zaleceniami powyżej,
- Uruchomić go przez:

```
python do_lumpy.py
```

- Powinno utworzyć się graficzne okno w którym możemy poprawić ręcznie otrzymany diagram, tak żeby wizualnie lepiej wyglądał,
- Wpisujemy nazwę pliku w lewym górnym rogu i zapisujemy plik.

Zapisane diagramy dobrze nadają się do dokumentacji technicznej projektu, ułatwiają też wstępne zorientowanie się w kodzie. Za pomocą diagramu obiektów można by próbować stworzyć wizualny debugger, o ile projekt jest wystarczająco prosty (mój był o rzędy wielkości za duży).

Przykładowy screen z okienka graficznego:



Ilustracja 11: do_lumpy uruchomiony na samym lumpy.

6. Podsumowanie

Jak się okazało i co pewnie widać z tego artykułu, narzędzia dla Pythona co prawda istnieją, ale większość z nich jest albo niezbyt dopracowana (np. depgraph) albo są to dawno zapomniane i nie wspierane projekty (np. depgraph, Pymetrics itd.). Aktywne są na pewno statsvn, Pydev i Pylint.

Każdy z wymienionych programów pomaga w zarządzaniu projektem tworzonym w Pythonie, ale brakuje jednego dużego środowiska które by udostępniało wszystkie te funkcjonalności w przystępny sposób. Wydaje mi się, że największe nadzieje rokuje PyDev + Pylint, zwłaszcza że narzędzia te i tak potrafią już analizować kod źródłowy w Pythonie. Dołożenie do nich dodatkowych funkcjonalności graficznej reprezentacji zależności w kodzie czy też mierzenia metryk nie powinno być aż tak dużym problemem. Pewnym ograniczeniem jest tu konieczność dostosowania się do środowiska Eclipse, ale być może da się stworzyć wtyczkę do wtyczki (czyli do PyDev-a) i w ten sposób rozszerzyć ich funkcjonalność bez konieczności ingerencji w istniejący projekt.

Krzysztof Niemkiewicz

Student 3 roku MISMAP UW

[K.Niemkiewicz\[at\]students.mimuw.edu.pl](mailto:K.Niemkiewicz@students.mimuw.edu.pl)