Omelette

Raport z projektu

Sławomir Blatkiewicz Jakub Górniak Piotr Piechal
Bartosz Pieńkowski Barnaba Turek Michał Zochniak

10 czerwca 2011

Część I.
Historia projektu

1. Interfejs programu

Na samym początku projektu, po wybraniu jego tematu, planowaliśmy stworzyć narzędzie obsługiwanie z linii komend do "kompilacji" plików tekstowych w diagramy UML. Mieliśmy również pomysł aby stworzyć moduł do języka LaTeX służący do generacji diagramów, jednak szybko go zarzuciliśmy. Z upływem czasu doszliśmy do wniosku, że łatwiejsze i wygodniejsze w użyciu będzie proste IDE umożliwiające zarówno tworzenie "kodu diagramu" jak i jego "kompilację". W pierwszej wersji okno aplikacji podzielone było na dwie części: część zawierającą kod i diagram. Diagram był statycznym obrazem generowanym na podstawie kodu w sąsiedniej części okna. Kolejnym krokiem było umożliwienie użytkownikowi przeciąganie elementów na diagramie, który przestał być statyczny. Również okno z kodem otrzymało nową funkcjonalność jaką było podstawowe kolorowanie składni opartej o syntaktykę języka Python. Dodano również przycisk pozwalający na wygenerowanie diagramu w postaci rastrowej i zapisanie jej do pliku. Następnie okno kodu zostało wyposażone w obsługę składni stworzonego przez nas języka. Kolejnym etapem było wykorzystanie architektury paneli do budowy wewnętrznej struktury okna oraz dodanie panelu obsługi błędów przetwarzania kodu. Dopiero w ostatniej fazie rozwoju aplikacji powstał moduł umożliwiający kompilację kodu do grafiki rastrowej z linii komend.

2. Język opisu diagramów UML

Od samego początku zakładaliśmy strukturę języka opartą o system klucz-wartość, którą utrzymaliśmy do końca trwania projektu. Naszym głównym celem było uzyskanie jak

największej elastyczności języka, która miała umożliwiać definiowanie własnych elementów diagramów bez konieczności ingerencji w wewnętrzne struktury aplikacji. Zostało to uzyskane przez zastosowanie architektury modułów dołączanych dynamicznie przy uruchomieniu aplikacji. Użytkownik może stworzyć własne moduły, czyli elementy z których mogą składać się diagramy. Przez to możliwości rozbudowy naszego programu są praktycznie nieograniczone. W tym momencie użytkownik nie jest związany z notacją UML. Dopisując własne moduły można adaptować naszą aplikację np. do tworzenia diagramów BPMN. W obecnej formie możliwa jest również walidacja kodu poddawanego kompilacji oraz zwrócenie wykrytych błędów wraz z numerami linii w których wystąpiły oraz prostą diagnostyką. Użytkownik w momencie kompilacji otrzymuje informację zwrotną na temat popełnionych błędów w konstrukcji kodu diagramu, które nie zawsze muszą być widoczne wprost.

3. Sposób przyłączania relacji do obiektów

Istotnym z punktu widzenia projektu architektury aplikacji zagadnieniem jest sposób w jaki relacje są podłączane do obiektów na diagramie. Na samym początku planowaliśmy by obiekty posiadały zbiór punktów na obrysie jego reprezentacji, do których relacja może zostać przyłączona. Takie rozwiązanie jednak stwarzało pewne problemy. Po pierwsze nie byliśmy w stanie z góry przewidzieć ile takich punktów na każdej krawędzi należy wyznaczyć aby wszystkie podłączone relacje wyglądały naturalnie. Natomiast zmiana zbioru tych punktów przy już istniejących dołączonych relacjach powodowałaby konieczność przeliczania ich położenia i przesuwania dołączonych relacji. Rozwiązanie na które zdecydowaliśmy się ostatecznie jest nieporównywalnie prostsze zarówno ideowo jak i w implementacji. Mianowicie każdy obiekt ma jeden punkt, do którego przyłączane są wszystkie relacje, umieszczony w jego centrum. Takie rozwiązanie wymusiło na nas rozwiązanie problemu przesłaniania relacji przez reprezentację graficzną obiektu. Realizowane jest to poprzez znajdowanie przecięć (w geometrycznym znaczeniu) reprezentacji graficznej z relacją, a następnie przycinaniu linii ją symbolizującej.

4. Integracja wszystkich modułów

Na samym początku projektu nie było wyraźnego podziału funkcjonalności aplikacji pomiędzy członków zespołu. Każdy tworzył fragmenty które stanowiły odrębne byty, między którymi nie istniały ustalone interfejsy komunikacyjne. Jednak w momencie w którym byliśmy gotowi na zapewnienie podstawowej funkcjonalności pierwszej wersji aplikacji okazało się, że doskonale działające w odosobnieniu moduły nie mają wspólnych interfejsów. Co więcej stworzenie ich nie było trywialną sprawą. Pomimo tego powstała pierwsza, i choć bardzo niedoskonała, to działająca wersja aplikacji zapewniająca podstawową funkcjonalność. Ponieważ wersja ta, ze względu na wspomniane problemy, była bardzo niedoskonała pod względem programistycznym, zdecydowaliśmy się na daleko idącą refaktoryzację kodu wraz z redefinicją kanałów komunikacyjnych. Dzięki temu krokowi powstała wersja aplikacji cechująca się pełną integracją modułów, jasno wyzna-

czonymi granicami funkcjonalności oraz jednoznacznie określonymi interfejsami między nimi. Także pojawiły się wyraźne podziały odpowiedzialności za poszczególne części aplikacji pomiędzy członków zespołu. Od tej pory praca nad rozwojem aplikacji była o wiele łatwiejsza i przebiegała bez większych problemów. Każdy z programistów rozwijał kod, który bardzo dobrze znał z poprzedniego etapu rozwoju aplikacji, co znacznie przyspieszyło pracę i współpracę. Jednak co ważniejsze, od tego czasu aplikacja jest w pełni zintegrowana, a poprawki wprowadzane przez programistów nie zaburzały tej integralności. Praktycznie w każdej chwili aplikacja była gotowa do uruchomienia. Wraz z tą zmianą, nadszedł czas wzajemnego testowania własnego kodu. Ponieważ aplikacja po wprowadzeniu zmian nie przestawała działać, wszyscy programiści dysponujący najnowszą wersją uruchamiając ją, chcąc czy nie chcąc, testowali kod nie tylko swój ale także innych członków zespołu. Od tej pory zaczęliśmy wzajemnie zgłaszać sobie zauważone usterki co drastycznie przyspieszyło tempo ich znajdowania.

5. Zarządzanie projektem

Ze względu na rozwojowy charakter projektu i niedookreśloną formę produktu końcowego do prowadzenia projektu starano się zaadaptować zwinną metodykę *Scrum*. Jako długość sprintu przyjęto stały okres dwóch tygodni, po upłynięciu których, począwszy od ostatniego tygodnia pierwszego semestru, sporządzany był zbiorowy raport relacjonujący pracę wykonaną przez każdego członka zespołu.

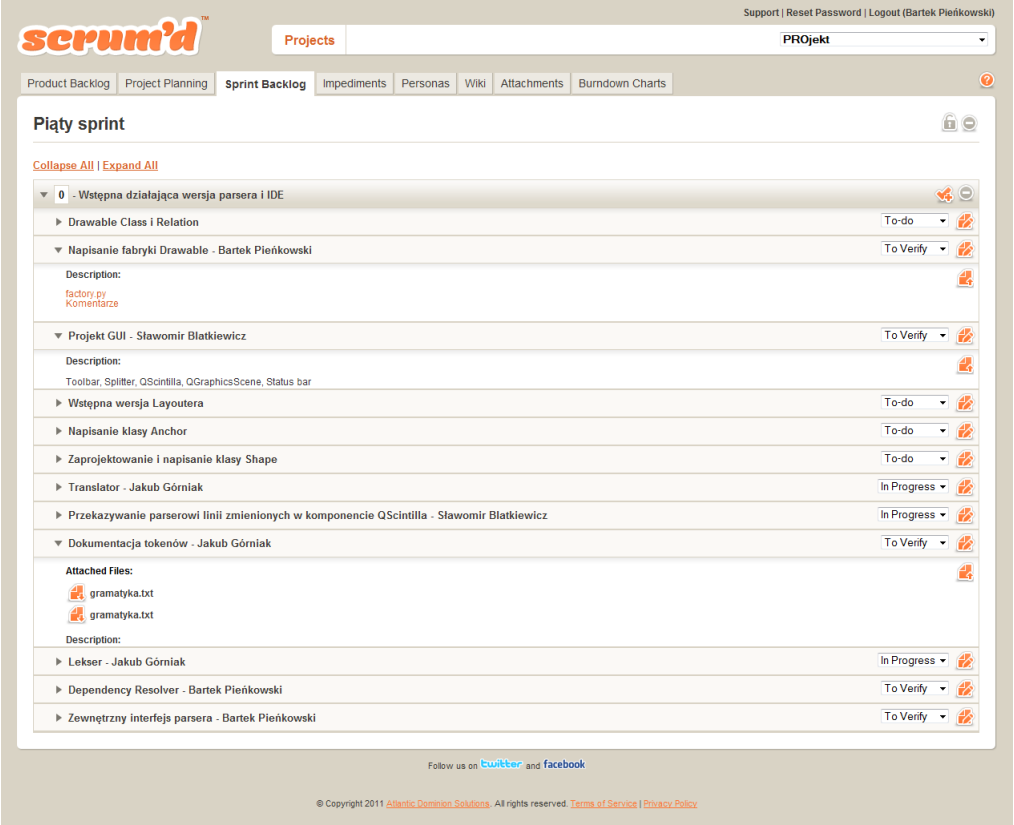
W początkowej fazie trwania projektu przed rozpoczęciem każdego sprintu ustalane były zadania, które następnie podlegały estymacji przy użyciu techniki o nazwie *Planning Poker*, w której udział brali wszyscy członkowie zespołu. Ustalanie terminarza sprintów, zarządzanie zadaniami (w tym przypisywanie ich konkretnym osobom oraz śledzenie stanu wykonania), jak również monitorowanie kondycji całego projektu możliwe było dzięki wykorzystaniu serwisu *Scrum'd*.

Dążąc do zapewnienia wysokiej jakości tworzonego kodu każdy nowo powstały fragment programu, przed dostaniem się do głównego repozytorium, weryfikowany był przez kierownika projektu i pozostałych członków zespołu w celu wskazania błędów i odnotowania ewentualnych uwag. Komentowanie kodu z dokładnością do pojedynczej linii możliwe było dzięki bogatej funkcjonalności serwisu *GitHub*.

W miarę upływu czasu każdy z członków zespołu zaczął zajmować się określonym obszarem projektu, dzięki czemu podział zadań stał się oczywisty. Podczas wzmożonej pracy nad stworzeniem demonstracyjnej wersji programu większość zadań przerodziła się w drobne poprawki, czego efektem była (prawdopodobnie niesłuszna) rezygnacja z dalszej estymacji zadań.

W celu przyspieszenia procesu integracji zorganizowano w tamtym okresie kilka spotkań w laboratorium 225, na których stawiła się większość członków zespołu. Zalety takiego sposobu pracy (szybkie tempo, ułatwiona komunikacja wewnątrz zespołu) były niepodważalne. W efekcie proces pełnej integracji kodu zakończył się sukcesem.

Nowy sposób pracy narzucił wymóg korzystania z bardziej odpowiedniego narzędzia do zarządzania nim, zrezygnowano więc z używania serwisu *Scrum'd* na rzecz o wiele

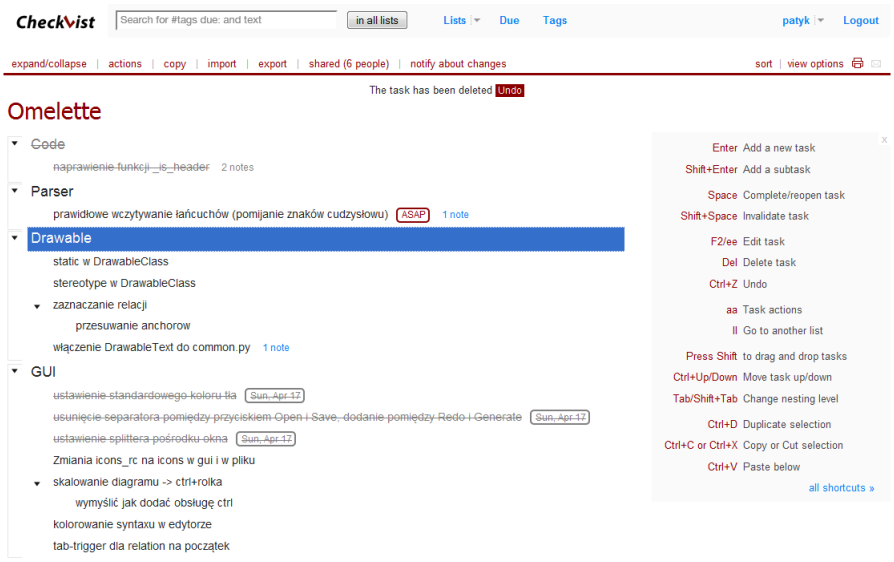


Rysunek 1: Wygląd listy zadań w serwisie *Scrum'd*

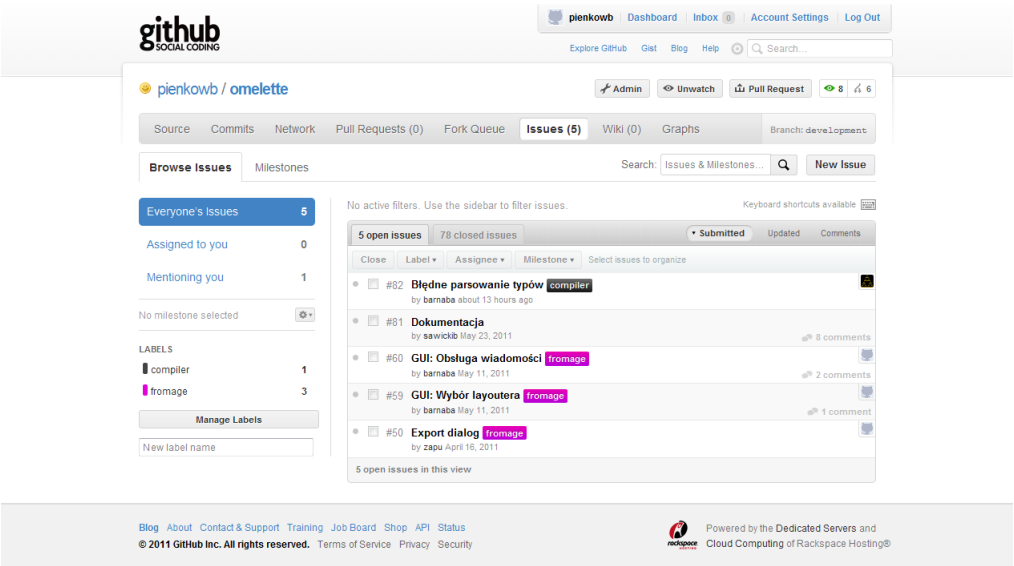
prostszego w założeniach i obsłudze serwisu *Checkvist*, będącego interaktywną listą zadań dającą możliwość współtworzenia wszystkim członkom zespołu. Zakres możliwości tego narzędzia obejmuje dodawanie i usuwanie zadań, oznaczanie ich jako wykonanych, komentowanie oraz ustalanie terminarza.

W końcowej fazie projektu zaczęto wykorzystywać nowy *issue tracker* udostępniony w ramach serwisu *GitHub*. Zarządzanie zadaniami przy pomocy tego narzędzia obejmowało m.in. przypisywanie ich konkretnym osobom, ustalanie kamieni milowych i dodawanie etykiet. Korzyścią wynikającą z jego używania była jawność zadań i osób za nie odpowiedzialnych, jak również możliwość zamykania wykonanych zadań przez wszystkich członków zespołu.

Niewątpliwą trudnością w zarządzaniu zespołem były opóźnienia w dostarczaniu kodu przez niektórych jego członków oraz problemy z egzekwowaniem przydzielonych zadań. Ostatecznie udało się w dużym przybliżeniu zrealizować zamierzoną funkcjonalność (która nie została przecież dokładnie sprecyzowana), więc projekt nie okazał się porażką. Z drugiej strony, mimo negatywnego nastawienia niektórych członków zespołu, można było utrzymać bardziej wydajne tempo pracy.



Rysunek 2: Wygląd listy zadań w serwisie *Checkvist*



Rysunek 3: Nowy *issue tracker* oferowany przez serwis *GitHub*

Część II.
Podział obowiązków

6. Sławomir Blatkiewicz

W czasie trwania projektu zajmowałem się głównie warstwą Graficznego Interfejsu Użytkownika. Wraz z kolegą sprawdzaliśmy możliwości biblioteki graficznej PyQt które można spożytkować w naszej aplikacji. Szczególną moją uwagę na tym etapie przykuwał komponent edytora kodu źródłowego. Badałem możliwość generowania diagramu w czasie rzeczywistym, już de facto podczas wprowadzania kodu w edytorze(czego nie udało się osiągnąć). Pomagałem również przy panelu diagramów. Do moich obowiązków należało również przystosowanie interfejsu zewnętrznego do zmian jakie występowały w innych modułach.

7. Piotr Piechal

W projekcie zajmowałem się layouterem diagramów. Zarówno wyborem algorytmów jak i ich implementacją oraz projektowaniem struktur wykorzystywanych przy ich realizacji. Na początku projektu wraz z kolegami opracowaliśmy interfejs modułu realizującego zadania layoutera. Następnie rozpocząłem poszukiwanie algorytmów zapewniających tę funkcjonalność. W ich wyniku stworzyłem opracowanie algorytmów, które nadają się do naszych zastosowań. Były to głównie algorytmy grawitacyjne lub sprężynowe. Następnie wspólnie z kolegami ustaliliśmy kolejność w jakiej będziemy chcieli implementować wybrane algorytmy. W pierwszej kolejności opracowałem, zaimplementowałem i zoptymalizowałem algorytm rozkładu kołowego. Działa rozkładając wierzchołki diagramu na okręgu minimalizując liczbę przecięć krawędzi poprzez umieszczanie sąsiadujących wierzchołków blisko siebie. Algorytm ten daje doskonałe wyniki w przypadku grafów gęstych o niewielkiej liczbie wierzchołków. Po zakończeniu fazy testów przeszedłem do implementacji pierwszego z algorytmów wytypowanych do umieszczenia w naszym projekcie - algorytmu Eadesa. Pseudokod algorytmu był bardzo ogólny i wymagał w dużej mierze własnej interpretacji. Po zaimplementowaniu go okazało się, że nie działa poprawnie. Po wielu nieudanych próbach poprawienia efektów jego działania zdecydowaliśmy się skorzystać z gotowej implementacji z biblioteki Graphviz. Niestety spędziłem tak wiele czasu na próbach poprawienia wyników algorytmu że nie wystarczyło mi czasu na implementację żadnego innego algorytmu z naszej listy. Planowałem również zaimplementować algorytm grupujący niektóre węzły, których sposób rozłożenia jest z góry znany (np. generalizacja). Algorytm rozkładania miał operować zarówno na pojedynczych wierzchołkach jak i na ich grupach. Niestety tego pomysłu również nie udało się zrealizować ze względu na brak czasu.

8. Bartosz Pieńkowski

Poza pracą wynikającą z pełnienia obowiązków kierownika projektu, skupiającą się na administrowaniu głównym repozytorium, sporządzaniu raportów i dbaniu o równy przydział pracy, aktywnie uczestniczyłem w podejmowaniu wszystkich decyzji projektowych i tworzeniu architektury programu. Miałem duży wpływ na ostateczny wygląd składni języka – wiele z pomysłów, które zaproponowałem, miało przełożenie na obowiązujący projekt gramatyki.

Samodzielnie zaprojektowałem moduł `code`, odpowiedzialny za wstępne dzielenie kompilowanego kodu na części zawierające pojedyncze obiekty. Ostatecznie potencjał tego modułu nie został w pełni wykorzystany, gdyż kod za każdym razem kompilowany jest w całości, a napisana przeze mnie klasa **Code** pozwalała ustalić, które obiekty uległy zmianie, pod warunkiem aktualizowania kodu w trakcie edycji.

Odpowiadam także za pierwszą wersję klasy **DrawableFactory**, która zapewniała dynamicznie ładowanie klas z konkretnych modułów na podstawie typu obiektu UML. Zaletą takiego rozwiązania była możliwość rozszerzania języka o kolejne elementy bez zmian w module rysującym. Klasa ta została później udoskonalona przez innego członka zespołu i stała się częścią modułu `diagram`, w którym przeprowadziłem gruntowną modyfikację zmieniając architekturę modułów z klasami rysującymi.

Kolejnym moim dokonaniem jest klasa **Compiler**, której zadaniem jest przeprowadzanie kolejnych etapów kompilacji począwszy od parsowania, a na walidacji kończąc. Integruje ona kilka innych klas – m.in. klasę **DependencyResolver** realizującą rozwiązywanie relacji dziedziczenia pomiędzy obiektami oraz klasę **Validator** odpowiedzialną za weryfikację zdefiniowanych kluczy obiektu na podstawie danych walidacji. W przypadku obu tych klas miałem ogromny wpływ na ich projekt oraz implementację.

9. Barnaba Turek

W początkowej fazie projektu zajmowałem się głównie planowaniem architektury i sposobu działania kompilatora. Potem zacząłem implementować klasy, które miały leżeć u podstaw tej architektury (takie jak **UMLObject**). Na tym etapie projektu znacząca część mojej aktywności polegała na pomaganiu niektórym kolegom w opanowaniu stosowanych technologii (głównie obsługi repozytorium) i czytaniu książki "Dive into Python". Zajmowałem się też zarządzaniem repozytorium — tworzeniem struktury katalogów, paczek, wybieraniem i sprawdzaniem możliwości frameworków, używanych do testowania.

W dalszych etapach projektu zajmowałem się też modułem common, który zawierał generyczne klasy, na których miały bazować konkretne klasy środowiska graficznego reprezentujące obiekty na diagramie. Wprowadzałem też drobne zmiany w modułach `actions`, `code`, `parser`, `layouter`, `logging`. Moduły `common` i `uml` (a także ich testy) wymagały wielokrotnego poprawiania. Dzięki temu zyskałem praktykę w pisaniu dobrych (nie specyficznych) testów jednostkowych.

10. Michał Zochniak

W projekcie zajmowałem się projektowaniem oraz implementacją warstwy aplikacji służącej do rysowania diagramów. Analizowałem oraz podejmowałem decyzje na temat komunikacji modułów kompilatora, rysowania oraz GUI. Przez cały czas życia projektu utrzymywałem moduł tworzenia diagramów z linii komend który wymagał częstej refaktoryzacji ze względu na dynamikę zmian w innych modułach, z których korzysta. Podczas projektu mocno podniosłem swoją wiedzę z zakresu Pythona oraz biblioteki Qt. Cały czas szukałem nowych rozwiązań dla istniejących (oraz tych które miały dopiero się ukazać) problemów. Jedyną niepokonaną przeszkodą okazała się generacja diagramów z poziomu konsoli w środowisku bez X-Server. Po wielu dniach poszukiwań oraz nieprzespanych nocy spędzonych na debugowaniu, doszedłem do wniosku, że jedyną możliwością jest użycie zewnętrznej biblioteki do rysowania czcionek - Qt polega na X-Serverze w sprawach czcionek. Uznaliśmy, że koszt jest większy niż zysk i zrezygnowaliśmy z możliwości generowania diagramów w środowisku bez „X-ów".

Część III.
Opis architektury aplikacji

11. Opis i uzasadnienie technologii

11.1. Python

Jako język programowania wybraliśmy język **Python**.

Python to interpretowany język ogólnego zastosowania, pozwalający programować na wysokim poziomie abstrakcji. Język oparty jest na wielu paradygmatach programowania: programowaniu obiektowym, programowaniu funkcjonalnym i programowaniu imperatywnym. Typowanie w **Pythonie** jest dynamiczne, a typy są mocne. Zarządzanie pamięcią odbywa się dynamicznie.

Interpretery **Pythona** dostępne są na wszystkie popularne systemy operacyjne, wiele z nich jest otwartym oprogramowaniem. Sama specyfikacja języka zarządzana jest przez *Python Software Foundation* — niezależną organizację non-profit.

Głównym powodem, dla którego zdecydowaliśmy się na język **Python** to jego popularność. Język ten ma opinię języka o bardzo dobrej dokumentacji. Popularność wpływa także na dostępność dużej ilości otwartych bibliotek, z których wiele jest dojrzałych i wysokiej jakości.

Wielu z nas korzysta na co dzień z systemu GNU/Linux, gdzie wiele aplikacji jest napisanych w języku **Python**. Znajomość języka **Python** pozwoliłaby nam więc robić zmiany w aplikacjach, z których korzystamy na co dzień.

Jednym z powodów, dla którego wybraliśmy język **Python** był także fakt, że nikt

z nas go nie znał. Wybranie nieznanego dotąd języka miało sprawić, że projekt będzie bardziej interesujący oraz zwiększyć kompetencje zawodowe członków zespołu [1] .

11.2. pyparsing

Pyparsing to jedna z bibliotek, które skłoniły nas do wybrania języka **Python** do realizacji tego projektu. Biblioteka **Pyparsing** to otwarte oprogramowanie.

Biblioteka pozwala w prosty sposób zbudować rekursywny analizator składniowy zstępujący. Gramatyka, pod kątem której analizowany ma być plik źródłowy, określana jest za pomocą języka **Python** w plikach źródłowych projektu (W naszym przypadku w pliku `lexer.py`). Pyparsing jest używany w takich projektach jak Django, pydot czy Graphite.

Parsowaną gramatykę opisuje się tworząc odpowiednie obiekty. Obiekty te mogą reprezentować symbole terminalne (wyrażenia regularne, zestawy znaków, pojedyncze znaki lub ich ciągi) lub ich produkcje. Każdemu obiektowi można przypisać akcję, która zostanie wykonana, gdy dany symbol zostanie wczytany.

11.3. Qt i pyQt

Qt to zestaw przenośnych bibliotek i narzędzi programistycznych dedykowanych do języków *C++* i *Java*. Pozwala budować graficzne interfejsy użytkownika w sposób zorientowany obiektowo.

Środowisko Qt to otwarte oprogramowanie. Środowisko dostępne jest na platformy *X11*, *Windows*, *Mac OS X*, *Haiku*, oraz na urządzeniach przenośnych opartych na Linuksie, *Windows CE* i *Symbian*.

Biblioteki **Qt** oprócz obsługi interfejsu użytkownika, zawierają także niezależne od platformy systemowej moduły obsługi procesów, plików, sieci, grafiki trójwymiarowej (OpenGL), baz danych (SQL), języka XML, lokalizacji, wielowątkowości, zaawansowanej obsługi napisów oraz wtyczek.

Dzięki bibliotece **pyQT** mogliśmy skorzystać ze środowiska **Qt** z poziomu języka **Python**.

Do graficznego projektowania interfejsu użytkownika użyliśmy programu **Qt Designer**.

Zdecydowaliśmy się na środowisko **Qt** ze względu na jego popularność, dostępność na platformy mobilne (wierzymy, że znajomość tego środowiska będzie dla nas cenna w przyszłości), fakt że jest to środowisko w pełni zorientowane obiektowo, oraz dostępność dojrzałej biblioteki do języka **Python**.

[1] W razie gdyby projekt nie okazał się hitem na miarę Napstera i musielibyśmy się jeszcze kiedykolwiek starać o pracę.

11.4. Nosetests

Nosetests, to biblioteka, której używaliśmy do testowania wytwarzanego przez nas oprogramowania. Biblioteka bazuje na module `unittest` dostarczanym w standardowej bibliotece języka **Python**.

Głównym powodem, dla którego zdecydowaliśmy się na korzystanie z tej biblioteki jest jej zdolność do automatycznego dodawania potrzebnych ścieżek. Ta cecha jest niezbędna do uruchamiania testów bez IDE, a alternatywą jest dopisanie kilku linii, ustawiających ścieżki na prawidłowe w każdym teście.

Ponadto **Nosetests** udostępnia wiele rozszerzeń, ułatwiających pisanie testów, takich jak generatory testów.

11.5. Git i Github

Git to otwarty rozproszony system kontroli wersji. Pozwala na łatwe tworzenie i łączenie gałęzi rozwoju projektu, szybkie przemieszczanie się pomiędzy wersjami i sprawdzanie różnic pomiędzy nimi. **Git** jest stosowany w takich projektach, jak jądro systemu Linux i umożliwia bardzo wiele (nawet w stosunku do innych nowoczesnych rozproszonych systemów kontroli wersji). Niestety, powoduje to, że nie jest to system łatwy w nauce.

Zdecydowaliśmy się na system **Git**, ze względu na doskonały serwis *Github*[2]

Serwis *Github* posłużył nam nie tylko, jako przestrzeń do współdzielenia kodu. Serwis ten udostępnia wiele narzędzi, które były bardzo przydatne w czasie prac nad projektem. Korzystaliśmy z wykresów, które pomagały się zorientować, który członek zespołu co robi. Inna przydatna opcja, z której korzystaliśmy, aby zapewnić wysoką jakość kodu, to komentowanie kodu napisanego przez innych. Od kiedy został wprowadzony ulepszony system zadań (*Issues 2.0*), korzystaliśmy z niego w celu przechowywania wymagań i komunikacji z opiekunem projektu.

Ponadto serwis pozwala na automatyczne powiadamianie o nowych zmianach w projekcie na kanale IRC, co było bardzo pomocne — motywowało do przeglądania kodu innych i usprawniało integrację.

11.6. Graphviz

Kiedy stało się jasne, że nie uda nam się napisać dobrego algorytmu rozkładającego elementy grafów, zaczęliśmy rozglądać się za gotową alternatywą. Znaleźliśmy program **Graphviz** — otwarte narzędzie służące do wizualizacji grafów, które obsługuje wiele metod rozkładania grafów.

Sam program **Graphviz** przyjmuje grafy opisane za pomocą prostego języka i generuje obrazki. Jednak twórcy tego programu udostępnili jego funkcjonalności w formie biblioteki.

[2] `http://github.com`

Skorzystaliśmy z biblioteki **pygraphviz**, która oferuje możliwość wywoływania funkcji udostępnianych przez bibliotekę **Graphviz** z poziomu **Pythona**.

11.7. Inne narzędzia

11.7.1. Zintegrowane środowisko programistyczne

W zespole nie zostało ustalone żadne środowisko programistyczne. Część członków zespołu używała środowiska *Eclipse*, część *Netbeans*.

Niektórzy członkowie zespołu tworzyli kod za pomocą zwykłych edytorów tekstowych.

11.7.2. Kanały komunikacji

Głównymi kanałami komunikacji wewnątrz zespołu były rozmowy w rzeczywistości i na kanale IRC.

Ponadto, przez pewien czas korzystaliśmy z serwisu *Scrumd*[3], służącego do zarządzania projektami prowadzonymi w metodyce Scrum. Głównym zastosowaniem tego narzędzia było przechowywanie wymagań. Wprowadzenie systemu *issues 2.0* w serwisie *Github* sprawiło, że serwis przestał być przydatny.

Korzystaliśmy także z serwisu *piratepad*[4]. Serwis ten pozwala na jednoczesną edycję plików tekstowych i łatwe dzielenie się nimi. Z serwisu korzystaliśmy we wczesnych wersjach projektu, kiedy ustalaliśmy ogólne cele i założenia.

12. Diagramy Klas

13. Działanie programu

13.1. Kompilacja

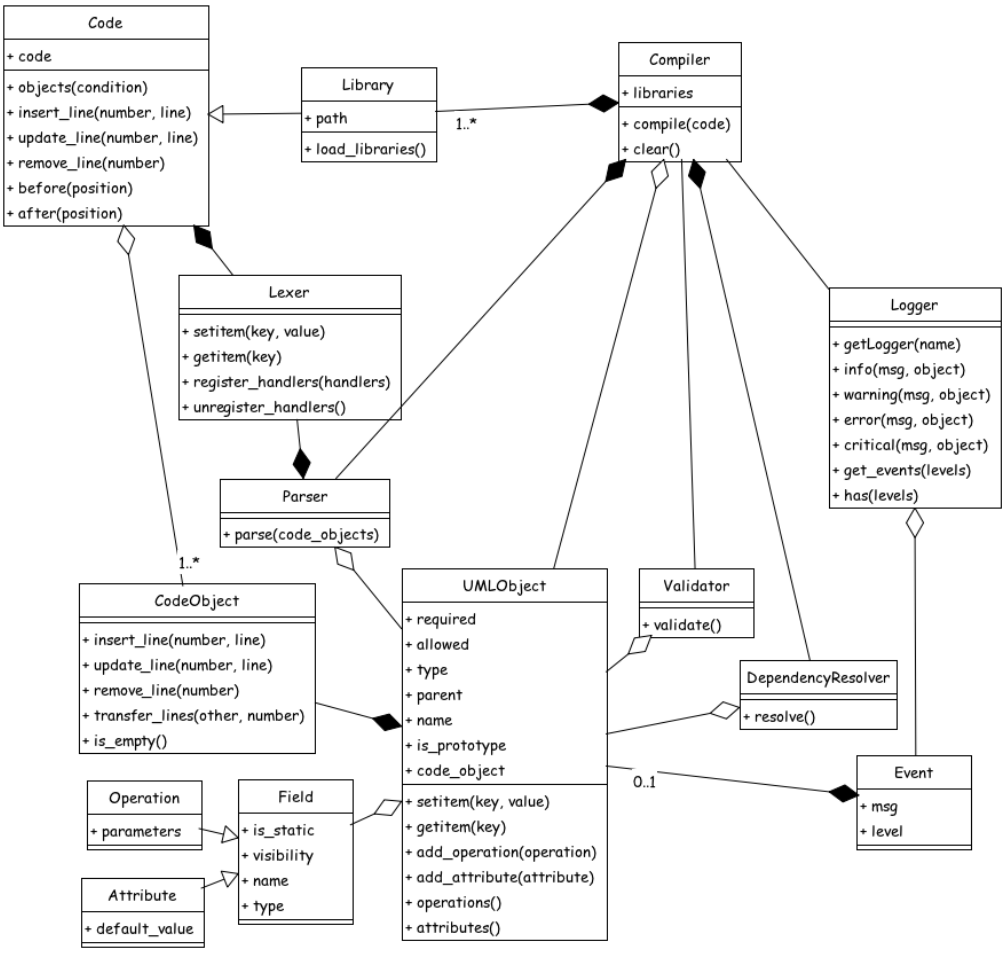
Kompilacja diagramów w programie **Omelette** składa się z następującego zestawu czynności:

- Wczytanie kodu i jego wstępny podział na obiekty
- Sparsowanie kodu
- Uzupełnienie obiektów danymi z ich prototypów
- Odrzucenie prototypów
- walidacja skompilowanych obiektów

13.1.1. Wczytanie kodu

Za ten etap odpowiedzialna jest klasa `Code`. Klasa przyjmuje napis, zawierający plik źródłowy. Za pomocą klasy `Lexer` sprawdzane są kolejne linie. Jeżeli linia jest definicją obiektu, tworzony jest nowy obiekt typu `_CodeObject`, zawierający linie należące do

[3] `http://scrumd.com`
[4] `http://piratepad.net`



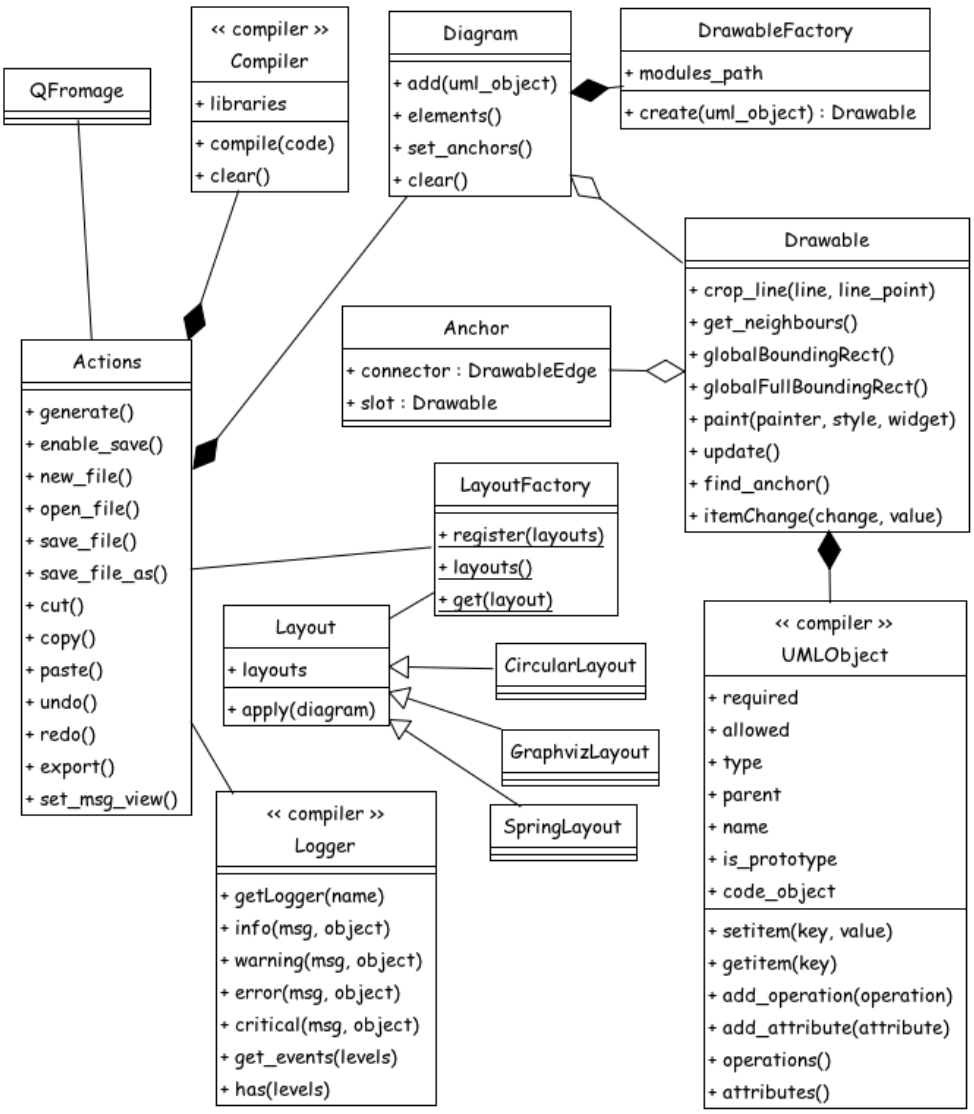
Rysunek 4: Diagram klas modułu `compiler`

obiektu następującego po znalezionym nagłówku. Poza samymi liniami `_CodeObject` przechowuje także informacje pozwalające określić numery linii w źródle.

Ponieważ pierwszy wiersz źródła nie musi być nagłówkiem obiektu, tworzony jest specjalny obiekt zerowy, przechowujące wszystkie linie przed pierwszym obiektem.

13.1.2. Parsowanie kodu

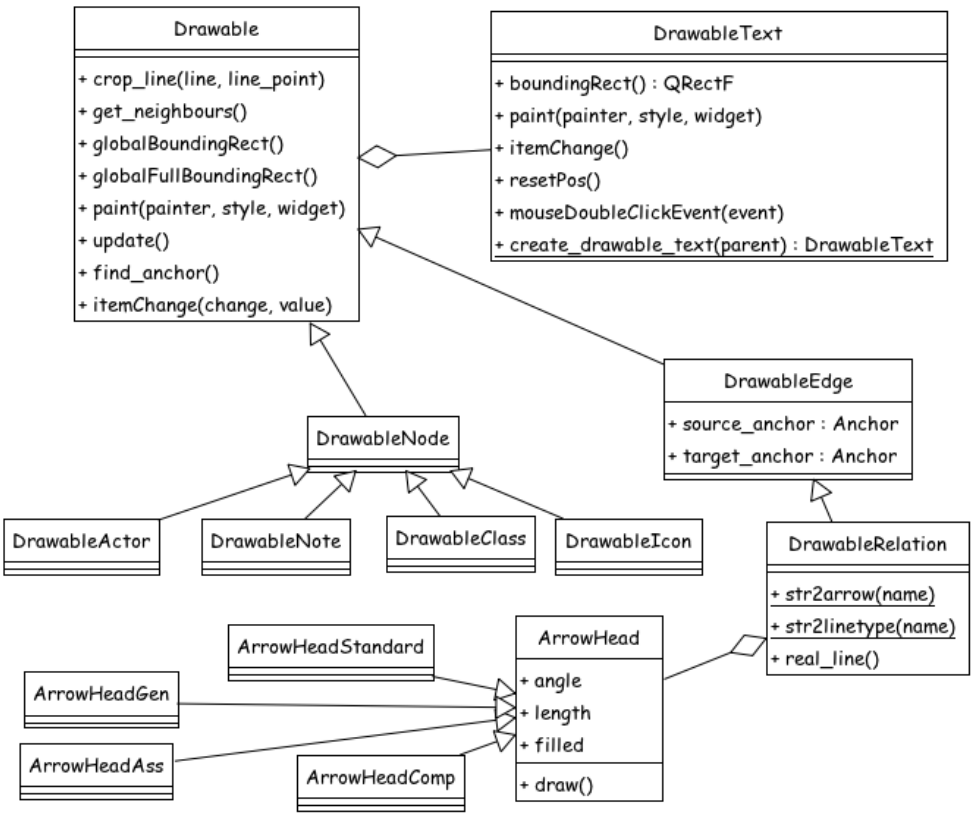
Obiekt klasy `Code`, zawierający obiekty `_CodeObject` jest następnie przesyłany do klasy `Parser`. Klasa ta zawiera funkcje budujące obiekty klasy `UMLObject`, na podstawie rozpoznanych symboli. Obiekt klasy `Parser` Tworzy obiekt klasy `Lexer`, w którym zde-



Rysunek 5: Diagram klas modułu `fromage`

finiowana jest gramatyka.

Następnie metody klasy `Parser`, służące do budowania obiektów `UMLObject` są dodawane jako obiekty obsługujące symbole gramatyki. Ostatecznie metoda `parse_string` klasy `Lexer` zostaje wywołana i zostają utworzone obiekty `UMLObject` odpowiadające plikowi źródłowemu.



Rysunek 6: Diagram klas modułu `modules`

13.1.3. Uzupełnienie obiektów

Za uzupełnienie obiektów odpowiada obiekt klasy `DependencyResolver`. Obiekt ten otrzymuje wszystkie obiekty `UMLObject`, które zostały stworzone w poprzednich krokach (oraz obiekty biblioteczne, tworzone przy inicjalizacji kompilatora).

Na początku obiekt sprawdza, czy nie występują cykliczne zależności pomiędzy obiektami (t.j. czy dwa lub więcej obiektów nie jest wzajemnie swoimi prototypami).

Następnie dla każdego obiektu znajdowany jest obiekt, będący jego prototypem i właściwości tego obiektu są dopisywane do aktualnego obiekt (pod warunkiem, że obiekt sam takich właściwości nie określa).

Po dojściu do obiektu, którego prototypem jest `base`, uzupełnianie danej gałęzi obiektów zostaje zakończone.

13.1.4. Odrzucanie prototypów

Prototypy muszą być odrzucone z puli obiektów z dwóch powodów:

- Nie powinny być rysowane na diagramie
- Nie muszą być spójne (spełniać wszystkich wymagań określonych przez ich prototypy). Pozostawienie ich przed następnym krokiem spowodowałoby niepotrzebne błędy.

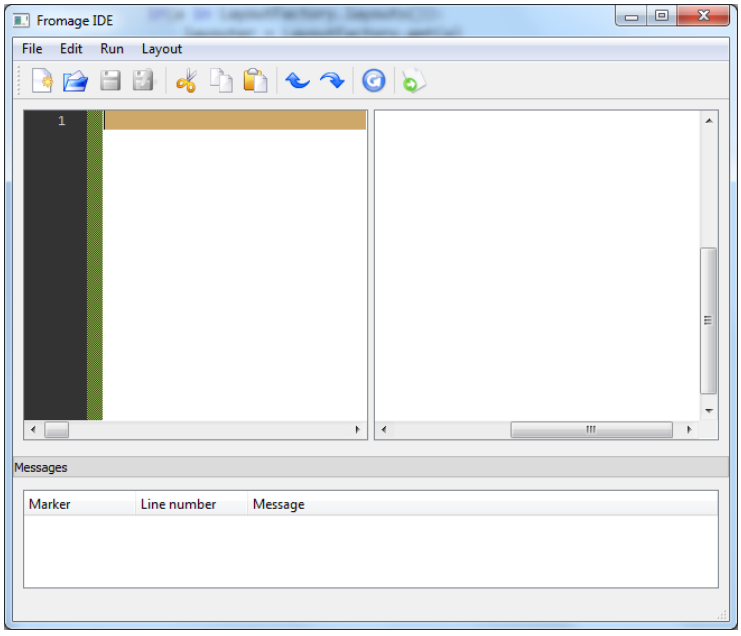
13.1.5. Walidacja obiektów

Obiekt klasy `Validator` sprawdza zgodność z danymi walidacji określonymi przez jego prototypy.

Normalnie sprawdzane jest istnienie, bądź nie istnienie konkretnej właściwości obiektu, oraz ew. zgodność typu tej właściwości. Jeżeli prototypy określają wymagane wartości o typie `Object`, to sprawdzane jest także, czy wskazane przez te wartości obiekty istnieją.

Część IV.
Instrukcja użytkownika

14. Interfejs graficzny



Po uruchomieniu programu powinniśmy zobaczyć okno główne składające się z trzech paneli:

- Edytor - panel służący do edycji aktualnie otwartego dokumentu.
- Podgląd - panel podglądu diagramu
- Lista błędów - sygnalizująca błędy w dokumencie

14.1. Pasek menu oraz pasek narzędzi

Opcje programu można wywoływać poprzez ikony na pasku narzędzi lub poprzez pasek menu.

Kolejno na pasku narzędzi znajdują się:

- „New" - otwiera nowy (pusty) dokument
- „Open document" - otwiera istniejący dokument
- „Save" - zapisuje aktualnie otwarty dokument
- „Save as" - umożliwia zapisanie aktualnie otwartego dokumentu z wyborem nowej nazwy
- „Cut" - funkcja „wytnij"
- „Copy" - funkcja „kopiuj"
- „Paste" - funkcja „wklej"
- „Undo", „Redo" - „cofnij", „potwórz"
- „Generate" - generuje diagram i wyświetla go w panelu podglądu
- „Export" - zapisuje diagram widoczny w panelu podglądu do pliku obrazka

Dodatkowo na pasku menu znajduje się kategoria „Layout". Menu „Layout" służy do wyboru algorytmu rozkładania diagramów przy generacji. Do wyboru są różne algorytmy, zależnie od platformy (a mianowicie obecności biblioteki **Graphviz**).

14.2. Praca z programem

Zazwyczaj praca z programem rozpoczyna odczytania istniejącego dokumentu lub rozpoczęcia pracy nad nowym dokumentem - w drugim przypadku można zacząć pracować na pustym dokumencie który jest dostępny od razu po otwarciu aplikacji. Użytkownik powinien najpierw zamodelować diagram opisując jego strukturę, zgodnie z regułami języka, w panelu edycji diagramu. Następnie kliknięcie użycie funkcji Generuj spowoduje wygenerowanie diagramu oraz wyświetlenie go w oknie podglądu. Nie stanie się tak w przypadku błędów krytycznych. Wszystkie błędy oraz ostrzeżenia sygnalizowane są w panelu błędów. Po wprowadzeniu poprawek do diagramu, można znowu kliknąć Generuj. W panelu podglądu powinniśmy wtedy zobaczyć uaktualniony diagram. W panelu podglądu możemy również przemieszczać elementy diagramu, wykonując gest „przeciągnij i upuść" za pomocą myszy. Jeśli jesteśmy zadowoleni z diagramu, możemy użyć funkcji Eksportuj, która zapisze diagram do obrazka.

15. Interfejs konsolowy

Diagramy można generować wywołując skrypt konsolowy. Pomoc do programu można wywołać używając flagi **-h** lub –**help**.

```
Usage: cli.py -h --help -i --input -o -output -l --layout -m --margin
       -s --scale

-h --help Displays this info
-i --input Input .uml file. If not provided, stdin is used.
-o --output Output image file.
-m --margin Picture margins. Default: 10px
-s --scale Scale of diagram. Default: 1
-l --layout Sets the layouter
Default layouter is Circular layout.
Available layouters:
        - Neato layout
        - Circular layout
        - Spring layout
        - Circo layout
        - SFDP layout
        - TWOPI layout
        - FDP layout
        - Dot layout
```

Funkcja *help* wyświetla dostępne flagi oraz dostępne algorytmy rozmieszczenia diagramów (zależnie od tego, czy jest dostępna biblioteka **Graphviz**).

15.1. Plik wyjściowy, wejściowy

Za pomocą flagi **-i** lub –**input** można zdefiniować położenie pliku wejściowego. Jeśli nie zostanie zdefiniowany plik wejściowy, program będzie czytał dokument ze standardowego wejścia.

Za pomocą flagi **-o** lub –**output** należy zdefiniować położenie pliku wyjściowego. Program nie zadziała, jeśli nie będzie możliwości pisania do podanego pliku.

15.2. Margines oraz skala diagramu

Za pomocą flagi **-m** lub –**margin** można zdefiniować marginesy diagramu. Definiowany jest margines z każdej strony płótna. Domyślny margines to 10 pikseli.

Za pomocą flagi **-s** lub –**scale** można zdefiniować skalę diagramu. Wartość skali 1 oznacza diagram „normalnej" wielkości, skala 2 utworzy diagram dwa razy większy a skala 0.5 diagram dwukrotnie mniejszy. Należy tutaj wspomnieć, iż margines nie podlega skalowaniu.

15.3. Wybieranie algorytmu rozkładania

Flaga **-l** lub –**layout** wybiera algorytm rozkładania. Domyślny algorytm to algorytm kołowy („Circular layout"). Można wybrać dowolny algorytm dostępny w danym środo-

wisku. Dostępne algorytmy wypisywane są w pomocy do programu, wywoływanej flagą **-h** lub –**help**.

16. Instrukcja użycia języka

Głównym elementem języka jest **obiekt**. Obiekty mogą posiadać nazwę. Właściwości obiektu definiuje się za pomocą **par klucz-wartość**. Ważną właściwością obiektu jest nazwa obiektu bazowego, tj. obiektu który jest prototypem dla danego obiektu.

```
class MojaKlasa
        name: "Moja Klasa"
        stereotype: "Fajna"
```

Przykładem jest obiekt o nazwie „MojaKlasa". Jego prototypem jest obiekt o nazwie „class", zdefiniowany w **bibliotece standardowej**. Innymi słowy: „class" jest obiektem bazowym obiektu „MojaKlasa".

Opisywanie obiektów odbywa się poprzez definiowanie par klucz-wartość. Wartości wpływają na wygląd obiektów na diagramie. „MojaKlasa" posiada dwie pary klucz-wartość: „name" oraz „stereotype".

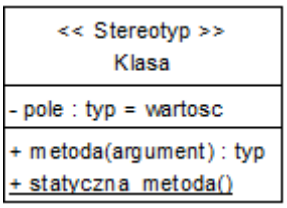
16.1. Przykłady użycia języka

Aplikacja potrafi wygenerować podstawowe elementy diagramu klas oraz diagramu przypadków użycia zdefiniowane w standardzie UML 2.0.

16.1.1. Klasa

Klasę definiuje się na podstawie prototypu **class**.[5]

```
class Klasa
    stereotype : "Stereotyp"
    +metoda(argument) : typ
    -pole : typ = "wartosc"
    _+statyczna_metoda()
```



Utworzony został obiekt o identyfikatorze **Klasa**. Na diagramie będzie to nazwą klasy. Można również nadać inną nazwę korzystając z klucza **name**(którego nie ma w przykładzie). Klucz **stereotype** pełni rolę stereotypów w języku UML. Przykładowa klasa

[5] O prototypach będzie mowa w dalszej części

posiada metodę(wraz ze statyczną) oraz pole. Na uwagę zasługuje określenie widoczności metod i pól: +, -, , #. Statyczne metody lub pola oznacza się poprzez _. Mając już zdefiniowaną klasę **Klasa** można utworzyć na jej podstawie następny obiekt, kopiując wszystkie ustawione wartości kluczy z **Klasa**: **Klasa** NewObject.

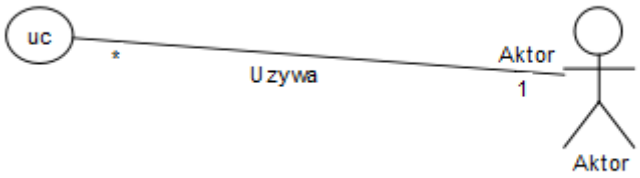
16.1.2. Relacja

Relację określa się na podstawie prototypu **relation**.

```
relation
    source-object : Aktor
    source-count : 1
    source-role : "Aktor"

    target-object : uc
    target-count : *
    name : "Uzywa"

actor Aktor
usecase uc
```



Relację jak wyżej można zdefiniować bez nazwy, jeżeli nie ma potrzeby wykorzystywać do tworzenia nowych obiektów pochodnych od relacji. Klucz **source-object** wskazuje obiekt na diagramie od którego relacja wychodzi, natomiast **target-object** wskazuje obiekt docelowy dla końca relacji. Klucze **source-role** oraz **target-role** odpowiadają za nadanie obiektowi źródłowemu i docelowemu ról zgodnie z notacją UML. Oczywiście nazwa relacji na diagramie poprzez **name**. Ponadto relacja może zawierać klucze: **arror** odpowiadający za określenie rodzaju strzałki relacji oraz **direction**, który odpowiada za miejsce umieszczenia tej strzałki.

16.1.3. Asocjacja

Jako przykład asocjacji można użyć powyższy listing. Oczywiście należałoby zmienić **relation** na **association**. Oparta jest również na odpowiednim prototypie. Asocjacja ma już na wstępie zdefiniowane klucze: **arrow** i **direction**. Można traktować asocjację jako podstawową formę relacji, tzn. bez kierunków.

16.1.4. Generalizacja

Jest relacją o określonym zwrocie i kierunku strzałki.

```
generalization
    source-object : Aktor
    target-object: Klasa

actor Aktor
class Klasa
```

16.1.5. Agregacja

Różni się od generalizacji typem strzałki oraz jej kierunkiem.

```
aggregation
    source-object : Klasa
    target-object: InnaKlasa

class InnaKlasa
class Klasa
```

16.1.6. Kompozycja

Kompozycja różni się od generalizacji i agregacji typem strzałki oraz ma kierunek zdefiniowany taki jak agregacja.

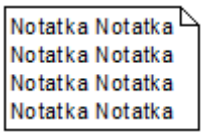
```
composition
    source-object : Aktor
    target-object : Notatka

actor Aktor
note Notatka
```

16.1.7. Notatka

Notatka może posiadać tylko tekst. Nie ma zdefiniowanych innych kluczy oprócz **text**.

```
note
    text : "Notatka Notatka Notatka Notatka Notatka Notatka Notatka Notatka"
```

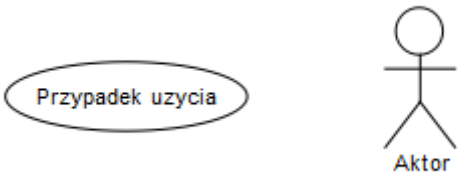


16.1.8. Aktor i przypadek użycia

Aktor posiada tylko klucz **name**. Podobnie przypadek użycia.

```
actor Aktor

usecase uc
    name : "Przypadek uzycia"
```



16.2. Prototypy

Uwaga - dalsza wiedza o języku nie jest wymagana do wygodnego użytkowania programu.

Na początku pokazaliśmy obiekt bazujący na obiekcie „class". „class" jest tzw. prototypem, zdefiniowanym następująco:

```
prototype base class
    allow name STRING
    allow stereotype STRING
```

Słowo kluczowe **prototype** sprawi, że obiekt będzie ignorowany przez moduł rysujący diagram. Stanie się „niewidocznym obiektem" używanym jako obiekt bazowy. **base** mówi, że obiekt nie bazuje na żadnym innym obiekcie. Deklaracje **allow** definiują ograniczenia nałożone na obiekty bazujące na tym prototypie. Sprawiają, ze bazując na „class" będzie można ustawiać tylko klucze „name" oraz „stereotype" które mogą być typu **STRING** czyli ciąg znaków.

16.2.1. Ograniczenia prototypów

Oprócz **allow** można używać również **require**. **Require** sprawi, że użytkownik będzie zmuszony ustawić podany klucz, inaczej kompilacja diagramu nie powiedzie się.

Przykład **require**:

```
prototype base relation
    require source-object OBJECT
    require target-object OBJECT
```

Poznaliśmy właśnie kolejny rodzaj wartości jaką można ustawić klucz - **OBJECT**. Wartość oczekiwana w tym wypadku to nazwa zdefiniowanego wcześniej obiektu, np. (ale niekoniecznie) prototypu.

Dozwolone ograniczenia na wartości to:

- STRING - ciąg znaków
- OBJECT - nazwa zdefiniowanego obiektu
- MULTIPLICITY - UMLowa krotność, czyli liczba lub zakres, dozwolony znak *
- Lista wartości - lista dozwolonych identyfikatorów podawana w formacie: [a, b, c].

16.2.2. „Metody" oraz „Atrybuty" dla klas UML

Język **Omelette** umożliwia definiowanie klas oraz atrybutów notacji UML. Wg przykładu:

```
        +metoda(argument) : typ
    -atrybut : typ = "wartosc"
    _+statyczna_metoda()
```

+ lub **-** przed nazwą metody/atrybutu oznacza widoczność (*public* lub *private*). **_** przed widocznością pozwala zaznacza, że metoda jest statyczna. Po **:** można zdefiniować typ oraz, dla atrybutów, domyślną wartość.

16.3. Prototypowanie

Tworząc obiekt bazujący na innym obiekcie (prototypie), obiekt przejmuje wszystkie ustawione klucze. Spójrzmy na przykład prototypu z ustawionymi kluczami:

```
prototype relation generalization
    arrow: "generalization"
    direction: target
```

Teraz definiując obiekt bazujący na prototypie „generalization", automatycznie przejmujemy wartości dla kluczy „arrow" oraz „direction". Znacznie ułatwia to tworzenie samych diagramów.

16.4. Obiekty wbudowane oraz ich prototypy

Obiektem wbudowanym nazywamy obiekty które są interpretowane przez **Omelette** i potrafią być reprezentowane przez fizyczny [6] obiekt na diagramie. Dostępne typy wbudowane to:

- actor
- class
- usecase
- note
- relation

Każdy z obiektów wbudowanych rysuje elementy diagramu używając zdefiniowanych kluczy. Biblioteka standardowa używa ograniczeń allow/require na kluczach używanych przez typy wbudowane w celu umożliwienia kompilatorowi sygnalizowania błędów.

Biblioteka standardowa udostępnia następujące prototypy używane do pisania diagramów. Obiekty bazowe odpowiadają wbudowanym obiektom, tzn. każdy obiekt bazowy ma swoją reprezentację na diagramach.

16.4.1. Actor

Reprezentuje aktora UML.

Ograniczenia:

- **name** (*STRING*) Nazwa aktora. Wyświetlana pod symbolem aktora.

16.4.2. Class

Reprezentuje klasę UML.

Ograniczenia:

- **name** (*STRING*) Nazwa klasy.
- **stereotype** (*STRING*) Stereotyp.

Typ wbudowany **class** zwraca uwagę na zdefiniowane *metody i atrybuty UML*.

[6] instancja pochodnych klasy Drawable

16.4.3. Usecase

Reprezentuje Use Case UML.
Ograniczenia:

- **name** (*STRING*) Nazwa przypadku użycia.

16.4.4. Note

Reprezentuje notatkę.
Ograniczenia:

- **text** (*STRING*) Tekst notatki

16.4.5. Relation

Relacja służąca do łączenia obiektów.
Ograniczenia:

- **name** (*STRING*) Nazwa relacji. Wyświetlana na środku odcinka łączącego obiekty.
- **arrow** Rodzaj strzałki. Dostępne strzałki to: „association" „composition" „aggregation" „generalization".
- **linetype** Rodzaj linii. Dostępne: „solid", „dash", „dot", „dashdot".

Dodatkowo **wymagane** (require) są: **target-object** oraz **source-object**, definiujące obiekty połączone relacją.
Definiować można również: **target-count** oraz **source-count** jako *MULTIPLICITY* - są to krotności UML; **target-role** oraz **source-role** jako *STRING* - są to role UML.
Kierunek relacji można zdefiniować na dwa sposoby: Ustawić **target-arrow** lub **source-arrow** na wybraną strzałkę (tak jak w kluczu **arrow**) lub użyć kluczy **arrow** (definiujący rodzaj strzałki) oraz **direction** przyjmujący [none, source, target, both], definiujący położenie strzałki.
Dodatkowo, zdefiniowane są następujące prototypy, bazujące na **relation**. Odpowiadają one danym relacją w notacji UML.

- generalization
- association
- aggregation
- composition