Projekt zespołowy - wstępne rozpoznanie problemu

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

25 października 2010

Spis treści

0.1 Przedmiot projektu

Projekt obejmuje cały proces powstawania oprogramowania umożliwiającego generowanie diagramów UML w postaci graficznej na podstawie plików tekstowych o określonej strukturze opartej na konkretnej składni.

0.2 Ograniczenia projektu

0.2.1 Zasoby czasowe

Projekt ma trwać około 9 miesięcy. Nieprzekraczalny termin zakończenia prac nad projektem to 10 czerwca 2011.

0.2.2 Zasoby ludzkie

Do realizacji projektu przydzielony został zespół 6 programistów w składzie:

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

0.3 Rozpoznanie problemu

Podstawą całego projektu jest stworzenie języka, który umożliwi precyzyjny opis elementów diagramu klas w notacji UML, zgodnie ze standardami wersji 2.0. Dodatkowymi modułami potrzebnymi do realizacji tego rozwiązania będą parser oraz generator plików graficznych.

0.3.1 Język

Język powinien umożliwiać opisanie podstawowych elementów diagramu klas, opisanych w specyfikacji notacji UML 2.0:

- klasa
- relacja (asocjacja, agregacja, generalizacja) wraz z określeniem liczebności i ról
- notatka
- paczka/moduł

Dodatkowo zakładamy iż język powinien umożliwiać definiowanie:

- relacji n-arnej
- klasy asocjacyjnej
- ograniczenia (constraint)

0.3.2 Parser

0.3.3 Generator plików graficznych

Generator plików graficznych powinien domyślnie korzystać z formatu PNG. Dodatkowo powinien automatycznie optymalizować ułożenie elementów na diagramie.

0.4 Założenia projektu

0.4.1 Cel podstawowy

Celem podstawowym jest stworzenie programu sterowanego z linii komend, który wygeneruje plik graficzny zawierający diagram klas odwzorowujący wskazany plik tekstowy zawierający kod w utworzonym języku.

0.4.2 Cele dodatkowe

Celami dodatkowymi, których realizacja rozważona zostanie po osiągnięciu celu podstawowego są:

1. utworzenie zintegrowanego środowiska programistycznego (IDE) do tego języka, w skład którego wchodziłyby następujace elementy:
   - edytor tekstowy oferujący kolorowanie składni, oraz inteligentne formatowanie tekstu
   - podgląd diagramu na bieżąco
2. rozszerzenie funkcjonalności IDE o możliwość redefiniowania położenia poszczególnych elementów na diagramie w trybie graficznym (*drag and drop*).

0.5 Proponowane rozwiązania

0.5.1 Składnia języka

Składnia języka z założenia ma być nieco podobna do CSS. Główną rolę w określeniu relacji odgrywają pary klucz - wartość, gdzie wartością może być także lista wartości. Pierwszy przykład, to asocjacja łącząca dwie klasy:

Prosta asocjacja

Listing 1: przyklad 1

```
1 association
2   target: Student 1..*
3   source: University 1 "teaches"
```

association określa na podstawie jakiego obiektu ma zostać utworzony nowy obiekt; W naszym przypadku tworzymy nowy obiekt na podstawie istniejącego (w bibliotece standardowej języka) obiektu asocjacji. Następnie w tej samej linii może wystąpić (koniecznie unikalny) identyfikator obiektu. W *przykładzie 1* assocjacja jest anonimowa, co oznacza, że nie będziemy mogli się później do niej odwołać.

Po utworzeniu obiektu możemy modyfikować jego właściwości. Dwukropek oddziela klucze od wartości. Ustawiamy wartość klucza **target** na **Student 1..***. Spacja (lub inny biały znak) oddziela od siebie elementy listy wartości. Student to identyfikator innego obiektu (prawdopodobnie klasy, ale w przykładzie nie widać deklaracji tego obiektu), a 1..* to liczebność. Analogicznie **source** w linii 3 wskazuje na drugi koniec asocjacji. Tutaj oprócz nazwy przyłączonego obiektu i jego liczebności możemy też zauważyć jego rolę.

Liczebności, napisy i identyfikatory obiektów to najczęściej występujące typy danych używane jako wartości.

Klasa

Oczywiście najważniejsze w diagramie UML są klasy. Zwykła klasa to obiekt zbudowany na podstawie obiektu class:

Listing 2: przyklad 2

```
class Student
  + ucz_sie_pilnie()
  + przychodz_na_wyklady() : Wiedza

  # wiedza : Wiedza = FabrykaWiedzyInformatycznej.Zrob_wiedze
  - ocena = 5
  +_ liczba_studentow
```

W linii pierwszej tworzymy nowy obiekt na bazie klasy, którego identyfikatorem jest Student. Identyfikator ten posłuży nam do wiązania studenta relacjami z innymi obiektami takimi jak notatki, inne klasy, a nawet relacje. Przy okazji identyfikator automatycznie staje się nazwą klasy. Można to zmienić ustawiając nową nazwę (napis) jako wartość klucza **name**.

Student jak student, ma kilka metod o określonych argumentach, widocznościach (znaki +, # i -) oraz zwracanym typie. Ma też kilka pól, które poza typem mogą przyjmować domyślną wartość. Znak podkreślenia oznacza, że dane pole lub metoda jest statyczne.

Dziedziczenie

Przypuśćmy, że nie wszyscy studenci są tak dobrzy, jak przewidział analityk. Okazuje się, że potrzebujemy klasy **PrzecietnyStudent** ktory zaczyna z oceną 3. Pisanie całej klasy od nowa zużyłoby i tak zużyte klawisze ctrl, c i v, a ponadto zaciemniło kod. Zbudujemy wiec nowego Studenta bazujac na poprzednim, i zmienimy mu tylko domyslna ocene.

Listing 3: przyklad 3

```
class Student
  + ucz_sie_pilnie()
  + przychodz_na_wyklady() : Wiedza

  # wiedza : Wiedza = FabrykaWiedzyInformatycznej.Zrob_wiedze
  - ocena = 5
  +_ liczba_studentow

Student PrzecietnyStudent
  - ocena = 3
```

Tak samo jak tworząc studenta skopiowalismy obiekt reprezentujacy klasę i dodaliśmy kilka wartości, tak teraz skopiowaliśmy obiekt **Student** i zmieniliśmy jedno z pól obiektu wynikowego.

Tak zdefiniowane dziedziczenie pozwala na wiele użytecznych skrótów. Wyobraźmy sobie, że modelujemy sieć i często używamy klas o stereotypie router. Zamiast ciągle tworzyć nowe klasy i ustawiać im klucz **stereotype**, tworzymy raz klasę-prototyp Router (słowo kluczowe prototype oznacza, że nie będzie ona narysowana na diagramie), a następnie robimy nowe Routery. W ten sam sposób możemy utworzyć asocjację one-to-many:

Listing 4: przyklad 4

```
prototype association one-to-many
  source-count : 1
  target-count : *
```

```
4
5 // one Student owns many books
6
7 one-to-many
8   source : Student
9   target : Book
10
11   label : "owns"
```

Jeżeli przyjrzymy się *przykładowi 4* dokładnie, zauważymy że liczebność możemy zmieniać zarówno za pomocą klucza **source** jak i klucza **source-count**. To samo dotyczy kluczy **target** i **target-count**. To nie przypadek. **target** i **source** to klucze główne, a **count**, **role** i **object** to ich podklucze. Można ustawiać wszystkie podklucze jednocześnie używając klucza głównego, lub dokładnie specyfikować, które podklucze mają zostać ustawione używając notacji *klucz-podklucz*.

Relacja to obiekt pierwszej klasy

Ponieważ tworzymy relacje za pomocą tej samej składni co klasy, notatki i moduły, to możemy nadać im identyfikatory. Następnie inna relacja może za pomocą takiego identyfikatora wykorzystać wcześniej utworzoną relację jako swoje źródło lub cel. Pozwala to na:

1. Klasy asocjacyjne
   Wystarczy, że dwie klasy są połączone nieanonimową relacją. Następnie klasę asocjacyjną łączymy odpowiednią relacją z tamtą relacją.
2. Ograniczenia
   Ograniczenie to tylko relacja łącząca dwie asocjacje ze sobą. Relacji takiej możemy ustawić klucz **label**.

Inne planowane obiekty

Ponadto planujemy w późniejszych wersjach parsera dodać obiekty **n-ary** (reprezentujący romb używany do modelowania n-arnej asocjacji), **note** (notatka) i **module** lub **package**.

Oczywiście biblioteka standardowa będzie zawierać wiele gotowych relacji, takich jak agregacje, kierunkowe asocjacje, kompozycje, relację łączącą notatki i klasy asocjacyjne z ich celami itd.

Użyte w przykładzie klucze nie wyczerpują wszystkich kluczy, które planujemy obsługiwać (takich jak np. kierunek etykiety relacji, zwrot samej relacji, treść notatki).

0.6 Proponowane technologie

Proponujemy do osiągnięcia celu głównego wykorzystanie technologii języka Python. Za takim rozwiązaniem przemawiają następujące argumenty:

1. przenośność rozwiązania spowodowana skryptowym charakterem języka
2. łatwość użytkowania - brak potrzeby instalacji oprogramowania do jego poprawnego działania
3. aspekt dydaktyczny - chęć zapoznania się z proponowaną technologią

Do realizacji IDE proponujemy użycie biblioteki Qt.