Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Wojciech Wąs
Nr albumu: 209224

Uniwersalny Manipulator Bajtkodem do Realizowania Aplikacji

Praca licencjacka
na kierunku INFORMATYKA
w zakresie OPROGRAMOWANIA I METOD INFORMATYKI

Praca wykonana pod kierunkiem
dra Jacka Chrząszcza
Instytut Informatyki

wrzesień 2005

Oświadczenie kierujĄcego pracĄ

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje siŚ do przedstawienia jej w postŚpowaniu o nadanie tytułu zawodowego.

Data Podpis kierujĄcego pracĄ

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiĄzujĄcymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur zwiĄzanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załĄczonĄ wersjĄ elektronicznĄ.

Data Podpis autora (autorów) pracy

Spis treści

Rozdział 1

Wprowadzenie

Umbra jest narzędziem przeznaczonym do przeglądania i edycji bajtkodu aplikacji napisanych w języku programistycznym Java, zintegrowanym ze środowiskiem Eclipse. Posiada rozbudowaną funkcjonalność ulatwiajacą uzytkownikowi zrozumienie i pracę z bajtkodem Javy. Zostało napisane jako wtyczka (ang. plug-in) do środowiska programistycznego Eclipse w wersji 3.0.1. Projekt został stworzony w ramach zajęć z Zespołowego Projektu Prorgamistycznego w roku akademickim 2004/2005 pod kierunkiem dra Jacka Chrząszcza. W skład grupy realizującej projekt weszli: Jarosław Paszek (kierownik zespołu), Wojtek Wąs i Tomasz Batkiewicz. Projekt spotkał się z ciepłym przyjęciem podczas prezentacji 20 czerwca 2005 roku.

W pierwszym rozdziale zamieszczony został dokładniejszy opis funkcjonalności naszego programu. Rozdział ten mówi także o użytych w projekcie narzędziach i technologiach. Sposób instalacji naszego projektu znajduje się na dołączonej płycie CD.

W kolejnych rozdziałach jest opisany podział prac nad projektem między poszczególnych członków zespołu oraz zawartość płyty CD dołączonej do niniejszej pracy.

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3. Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.3. INFORMATION STORAGE AND RETRIEVAL
H.3.3. Systems and Software

Rozdział 2

Opis funkcjonalności projektu

Głównym celem powstania projektu UMBRA bylo utworzenie narzędzia umożliwiającego pracę z bajtkodem wygenerowanym przez kompilator Javy.

Nasze narzędzie umożliwia lepsze zrozumienie istoty działania kompilatora Javy i stosowanych przez niego metod. Dzięki temu pozwala na wypracowanie efektywnego sposobu programowania i optymalizacji działania tworzonych aplikacji.

Projekt zostal zrealizowany jako plugin środowiska programistycznego Eclipse w wersji 3.0. Napisany w całosci w języku Java (J2EE) z wykorzystaniem różnych metod tworzenia oprogramowania m.in. extreme programming. W implementacji posłużylismy sie dodatkowo bibliotekami Bytecode Engeneering Library (BCEL) w wersji 5.1.

Nasz projekt, aby zrealizować wyznaczony mu cel, zostal"wyposażony w następujące możliwosci:

1. generowanie bajtkodu z pliku zawierajacego kod źródłowy w języku Java - w niezależnym oknie wyświetlone zostają instrukcje bajtkodu odpowiadające poszczególnym liniom kodu źródłowego Javy. Całość może zostać zapisana w pliku reprezentującym specjalnie utworzony przez nas typ o rozszerzeniu ”btc”.

2. możliwosc modyfikacji bajtkodu wygenerowanego w ten sposob - użytkownik naszego projektu może dodawać, usuwać, modyfikować, zamieniać kolejność instrukcji bajtkodu. Zmiany te można zatwierdzić, wówczas zostaną one uwzględnione w trakcie uruchamiania, ponadto w oknie zawierającym bajtkod pojawi sie uaktualniona wersja programu. Jednocześnie cały czas dostępna jest opcja przywrócenia oryginalnego bajtkodu na podstawie pierwowzoru kodu źródłowego.

3. sprawdzanie poprawności - przez cały czas pracy widoczne jest oznaczenie kontolne zapalające się w razie wystąpienia niepoprawnych linii w bajtkodzie i wskazujące miejsce wystąpienia pierwszej z nich.

4. synchronizacja pozycji kursorów - korzystając z tej opcji użytkownik może przenieść kursor do linii kodu źródłowego odpowiadającej instrukcji bajtkodu zaznaczonej w danym momencie. Może tez wyznaczyć i podświetlić obszar bajtkodu odpowiadający danej linii kodu źródłowego.

5. uruchomianie programu, którego bajtkod został zmodyfikowany - użytkownik może przetestować sposób, w jaki wprowdzone zmiany wpłyną na działanie programu. Polecenie uruchomienia programu spowoduje uruchomienie jego najnowszej wersji.

6. scalanie zmian w kodzie źródłowym i w bajtkodzie - użytkownik może łączyc ze soba zmiany wprowadzane w kodzie źródłowym i w bajtkodzie. Zmiany te muszą jednak odnosić się do

różnych metod. W obrębie tej samej metody zmiany w bajtkodzie mają pierwszeństwo przed zmianami w kodzie źródłowym.

7. historia zmian - użytkownik może przechowywać wybrane wersje bajtkodu w specjalnych plikach. Możliwe jest odtworzenie bajtkodu na podstawie zapisanego pliku, co pozwala porownywać ze soba bezpośrednio rożne wersje programu.
8. pomoc dla użytkownika - krótkie wskazówki ułatwiające pracę.
9. możliwosc dostosowania scieżki źródeł
10. wybór spośród wielu wersji kolorystycznych

Rozdział 3

Zastosowane rozwiazania techniczne

3.1. Edytor

Podstawowym, stworzonym przez nas narzędziem do pracy z bajtkodem, jest edytor (*Bytecode Editor*) wprowadzony jako jedno ze standardowych rozszerzeń pluginu w środowisku Eclipse. Edytor został zdefiniowany jako domyślny dla plików o rozszerzeniu ".btc" (nasz własny typ plików przechowujących bajtkod). Wykorzystaliśmy standardowe narzędzia oferowane nam przez edytor:

- wszystkie skróty klawiszowe charakterystyczne dla edytorów (*Ctrl+C*, *Ctrl+V*, *Ctrl+D*, *Ctrl+Z*, *Ctrl+S*) zostały uwzględnione
- wprowadziliśmy system automatycznego kolorowania (m.in. rozpoznawanie slów kluczowych związanych z bajtkodem)

Ponadto edytor zawiera dowiązania do struktury *Java Class* zdefiniowanej w bibliotece BCEL, odpowiadającej edytowanej klasie, a także do powiązanego z nim edytora kodu źródłowego, który jest standardowym, zdefiniowanym w środowisku Eclipse, edytorem języka Java.

3.2. Wczytywanie bajtkodu

Wczytywanie bajtkodu odbywa się na podstawie pliku binarnego (dowiązywanego do kodu źródłowego bezpośrednio przez środowisko Eclipse). Na jego podstawie tworzona jest struktura *Java Class* (zdefiniowana w bibliotece BCEL). Z tej struktury wyodrębniane są poszczególne metody, następnie zaś zamieniane na postać tekstową - charakterystyczne mnemoniki bajtkodu wraz towarzyszącymi liniami pomocniczymi. Całość zapisywana jest w pliku tekstowym o rozszerzeniu ".btc", który z kolei jest wczytywany do edytora.

3.3. Kontrola zmian

W celu umożliwienia reagowania na zmiany w edytorze, wprowadziliśmy klasę kontrolującą edycję pliku (*Bytecode Contribution*). Klasa ta zawiera element wykrywający modyfikacje (*Bytecode Listener*), jest też odpowiedzialna za wyświetlanie kontrolnej informacji o błędnych instrukcjach.

Struktura bajtkodu jest odzwierciedlana w strukturze *Bytecode Controller*. Zawiera ona listę obiektów odpowiadających poszczególnym liniom dokumentu i, niezależnie od niej, listę tych linii, które są instrukcjami bajtkodu. Przechowywane obiekty reprezentują, zależnie od swojego typu, różne podklasy klasy *Bytecode Line Controller*, jedna z nich - *Instruction Line Controller* jest dodatkowo podzielona ze względu na typ parametrów charakterystycznych dla poszczególnych instrukcji.

Każda instrukcja zawiera uchwyt do listy instrukcji będącej elementem struktury *JavaClass* - głównej struktury odpowiedzialnej za generowanie bajtkodu w bibliotece BCEL.

Modyfikacja bajtkodu wiąże się z przeszukaniem obu list, w razie potrzeby usunięciem z nich obiektów, utworzeniem nowych lub zmianą parametrów istniejących. Typ każdej nowej linii ustalamy na podstawie zawartych w niej słów kluczowych. Gdy nowo wprowadzana linia jest instrukcją, tworzymy odpowiadający jej uchwyt do listy instrukcji i aktualizujemy zawierajacą ją metodę. W razie usuwania instrukcji, usuwamy dany uchwyt z listy, a wszystkie odwolania, ktore go dotycza (np. przez instrukcje "goto") przenosimy na instrukcję nastepną (wyjątek: usuwanie ostatniej instrukcji danej metody). Na koniec całościowo sprawdzamy poprawność bajtkodu, znaleziony błąd jest sygnalizowany użytkownikowi za pomoca ikony na pasku narzędzi (odpowiada za to klasa *Bytecode Contribution*). Modyfikacje sa uwzględniane w strukturze *Java Class* i zapisywane w pliku binarnym (o rozszerzeniu ".class") w momencie zapisywania bajtkodu lub odswiezania widoku w edytorze. Pierwotna wersja tego pliku jest zapisywana w pliku pomocniczym (oznaczanym przez "_" na początku) i może zostać w każdej chwili przywrócona (wiaże się to z utratą modyfikacji).

3.4. Moduł synchronizacji

Synchronizacja pozycji kodu źródłowego i bajtkodu odbywa się poprzez przeglądanie tablic zawierajacych pary numerów odpowiadających sobie linii (tablice te również zawarte są w strukturze BCEL). Każdej linii kodu źródłowego jest w nich przypisana instrukcja bajtkodu, od której zaczyna sie jej implementacja. Synchronizacja z bajtkodu w kod źródłowy polega na znalezieniu linii, której przypisana jest najbliższa instrukcja powyżej instrukcji przypisanej. Synchronizacja w przeciwną stronę polega na znalezieniu całego obszaru między instrukcją przypisaną danej linii a instrukcją przypisaną linii następnej. Cała operacja odbywa się dzięki nowo wprowadzonemu typowi dokumentu: *Bytecode Document*, ktory oprócz standardowych atrybutów dokumentu ma bezpośredni dostęp do struktury *JavaClass*, co umożliwia szybkie rozpoczęcie wyszukiwania, a także do powiązanego edytora kodu źródłowego, dzięki czemu po wykonanej operacji następuje automatyczne przekierowanie.

3.5. Scalanie zmian

Problem scalania zmian w bajtkodzie i w kodzie źródłowym rozwiązujemy dzięki trzymanej dodatkowo we wspomnianej juz klasie *Bytecode Controller* tablicy modyfikowanych metod. Podczas wykonywania odpowiadajacego za to polecenia, tworzony jest nowy plik binarny. Konieczne jest rownież wczesniejsze utworzenie dodatkowego pliku binarnego powiązanego z kodem źródłowym. Na podstawie tablicy, metody o modyfikowanym bajtkodzie przepisywane sa ze struktury *Java Class* związanej z edytorem bajtkodu, pozostałe z pliku binarnego związanego z kodem źródłowym.

3.6. Wersje historyczne

Historyczne wersje bajtkodu zapisywane są w plikach pomocniczych o rozszerzeniach ".bt0", ".bt1", ".bt2", wraz z odpowiadąjacymi im plikami binarnymi. Przywrócenie nazwy historycznej polega na zmianie nazwy pliku, wyczyszczenie historii - na usunięciu wszystkich takich plików.

Rozdział 4

Wkład własny

W tym rozdziale przedstawię pokrótce, w których elementach projektu i etapach jego tworzenia miałem osobisty udział.

4.1. Dokumentacja

Mojego autorstwa są:

- fragment wizji dotyczący potencjalnych użytkowników oraz hierarchia priorytetów
- większa część przypadków użycia
- fragmenty opisu architektury systemu zawierające graficzne przedstawienie zależności

Ponadto byłem odpowiedzialny za stronę językową i poprawność ortograficzną wszystkich tworzonych dokumentów.

4.2. Sposób prezentacji

Problem, który pojawił się na samym poczatku kodowania, dotyczył wyboru, czy generowany bajtkod powinien być prezentowany jako edytor (z duża dowolnościa ręcznego manipulowania tekstem), czy widokiem o ściśle określonej strukturze linii. Głównymi wadami związanymi z edytorem były: gorsza kompozycja wizualna z kodem źródłowym - okna otwierane były w tym samym obszarze i nigdy naraz (ostatecznie udało się wprowadzić podział pionowy okna na dwie części, przy czym jednak nadal użytkownik musi to robić ręcznie), a także utrudniony bezpośredni dostęp do jego zawartości. Ostatecznie jednak argument o dowolności manipulowania przeważyl.

4.3. Rozpoznanie edytora

Jako wzorzec edytora posłużył nam edytor plikow ”xml” stanowiący jedno ze standardowych rozszerzeń w pluginach środowiska Eclipse. Jednym z pierwszych etapów implementacji było dostosowanie tego edytora do potrzeb bajtkodu. Moim pomysłem było wprowadzenie rozszerzenia ”.btc”, dla którego nowy edytor miał być edytorem domyślnym.

Dodatkowa funkcjonalność edytora była zawarta w kilkunastu klasach generowanych standardowo przez Eclipse. Moim, wcale nie banalnym, zadaniem było poznanie mechanizmów rządzących edytorem na podstawie zagłebienia się w kod tych klas. Dzięki temu udało mi się zlokalizować klasy odpowiedzialne za kolorowanie tekstu i dodać własny, bardzo efektowny, mechanizm kolorowania słów kluczowych.

4.4. Ograniczenia związane z edytorem

Przy tej okazji doświadczyłem poważnych ograniczeń, z jakimi związany jest edytor w Eclipsie. Już podczas kolorowania wyszło na jaw, że zdefiniowanie wzorca nie może obejmować zależności między parametrami w linii, a jedynie ograniczać się do wskazania podświetlanych słów kluczowych dla każdego z kolorów i specyfikacji znaków, które mogą wchodzic w ich skład.

Podczas definiowania edytora było oczywiste, że musi on przechowywać pewne dane związane ze specyfiką swojego zastosowania. Lista tych danych zmieniała się stopniowo, często jednak pojawiała się wśród nich (i pozostała w ostatecznej wersji) struktura *Java Class* z biblioteki BCEL. Aktywowanie edytora odbywa sie przez metodę zdefiniowaną w klasie *WorkbenchPage*, która jest klasą standardową Eclipse'a i w związku z tym nie może być modyfikowana. Z tego powodu niemożliwe było dodanie nowych parametrów do tej metody. Ostatecznie przyjąłem bardziej pracochłonne i mniej eleganckie rozwiązanie polegające na dołączeniu potrzebnych danych do już aktywnego edytora. Wymagało to jednak rozprowadzenia tych danych po licznych obiektach pomocniczych edytora tworzonych w momencie jego aktywacji, w momencie dołączania już aktywnych.

Bardzo bolesnym ograniczeniem był brak możliwosci automatycznej zmiany treści edytowanego dokumentu (lub niektórych jego ustawień, np. koloru) w obrębie edytora. Z tego powodu funkcja *refresh* i wszystkie z niej korzystające w obecnej wersji projektu, aby odswieżyć dokument, zapisują jego treść, następnie zamykają edytor i otwierają jego nową kopię.

4.5. Podstawowa rejestracja zmian

Pierwszym etapem oprogramowania modyfikacji w bajtkodzie było stworzenie klasy *Bytecode Listener* wychwytującej odpowiednie zdarzenia i dołączenie jej do dokumentu zawierającego bajtkod. Główny problem wiązał się z podziałem kompetencji między klasy - dołączanie do dokumentu obiektów wykrywajacych zdarzenia odbywało sie w klasie *Bytecode Document Provider* (jednej z głównych klas pomocniczych edytora), natomiast większość zdarzeń, które ten obiekt mogłby wywołac (np. wyświetlenie kolorowej etykiety) była definiowana w zupełnie od niej niezależnej klasie *Bytecode Contribution*. W tej sytuacji jedynym możliwym wyjściem było odstąpienie od czystej obiektowości i statyczne zdefiniowanie instancji *Bytecode Listener*.

4.6. Instalacja biblioteki BCEL

Jednym z problemów, który na dłuższy czas wstrzymał prace naszego zespołu był problem z instalacją nowej biblioteki. Po pierwszych próbach program potrafił rozpoznać zawarte w niej klasy, ale nie był w stanie wykonać ich metod. Moim dość brutalnym, ale skutecznym pomysłem było stworzenie nowego pluginu, do którego wrzuciłem źródła biblioteki BCEL, następnie ustawiłem zależnosci między pluginami, w ten sposób żeby nasz mogł w pełni wykorzystywać zasoby BCEL.

4.7. Koncepcja listy instrukcji

Mojego autorstwa jest główna koncepcja klasy *Bytecode Controller* przechowującej listę linii bajtkodu i listę instrukcji. Alternatywna, niewątpliwie gorszą koncepcją było generowanie klasy od początku po każdej modyfikacji. Realizacja koncepcji była w znacznym stopniu zadaniem pozostałych czlonków zespołu.

4.8. Obsługa uchwytów

Mojego autorstwa jest znaczna część tych metod klas *Bytecode Controller* i *Instruction Line Controller*, które są odpowiedzialne za przydzielanie tym obiektom uchwytów do listy instrukcji w postaci BCEL. Moim zadaniem było również w dużej części testowanie działania tego mechanizmu oraz poprawianie niezwykle licznych błędów, które sie przy tej okazji pojawiły. W sumie ta część pracy pochłonęła najwięcej czasu poświeconego na implementację programu.

Rozdział 5

Dodatki

5.1. Bibliografia

Korzystaliśmy ze zródeł: **Eclipse:**

`http://www.eclipse.org/`

Bajtkod:

`http://java.sun.com/docs/books/vmspec/html/Mnemonics.doc.html`
`http://mrl.nyu.edu/~meyer/jvmfer/`
`http://java.sun.com/docs/books/html/ClassFile.doc.html`

BCEL:

`http://www-128.ibm.com/developerworks/java/library/j_dyn0414/`
`http://jakarta.apache.org/bcel/manual.html`

5.2. Zawartość CD

Dołączona do pracy płyta CD zawiera:

- **plik ”umbra.rar” - spakowane źródła projektu**
- **katalogi ”umbra_1.0.0” i ”org.apache.bcel_5.1.0” służące do instalacji projektu**
- **instrukcję instalacji i pomoc użytkownika**
- **dokumentację projektową (Wizja, Biznesowe przypadki użycia, Plan rozwoju projektu, Opis architektury systemu)**
- **3 prezentacje przygotowane w pierwszym semestrze**
- **kopię niniejszej pracy**