Uniwersytet Mikołaja Kopernika

Wydział Matematyki i Informatyki

Zakład Baz Danych

Paweł Piątkowski
nr albumu: 187438

Praca magisterska
na kierunku informatyka

Graficzny edytor zapytań do bazy danych db4o

Opiekun pracy dyplomowej
dr hab. Krzysztof Stencel, prof. UMK
Wydział Matematyki i Informatyki

Toruń 2010

Pracę przyjmuję i akceptuję

.....
data i podpis opiekuna pracy

Potwierdzam złożenie pracy dyplomowej

.....
data i podpis pracownika dziekanatu

Streszczenie

Celem niniejszej pracy magisterskiej była implementacja wygodnego interfejsu pozwalającego tworzyć zapytania do obiektowej bazy danych db4o. W wyniku prac powstał graficzny edytor zapytań napisany jako wtyczka do Eclipse IDE. Edytor pozwala wizualnie tworzyć graf zapytania. Graf ten określa warunki jakie muszą spełniać obiekty aby znaleźć się w zbiorze rezultatów. Na podstawie grafu generowany jest kod języka Java realizujący zapytanie. Wygenerowany kod używa SODA API jako metody dostępu do danych utrwalonych w bazie db4o. Aplikacja napisana jest z użyciem biblioteki GEF (Graphical Editing Framework).

Spis treści

ROZDZIAŁ I

Wprowadzenie

1 Omówienie problemu

Niniejszy podrozdział wprowadza czytelnika w specyfikę obiektowych baz danych. Następnie pojawia się omówienie obiektowej bazy danych db4o wraz z przedstawieniem dostępnych w niej interfejsów dostępu do danych.

1.1 Obiektowe bazy danych

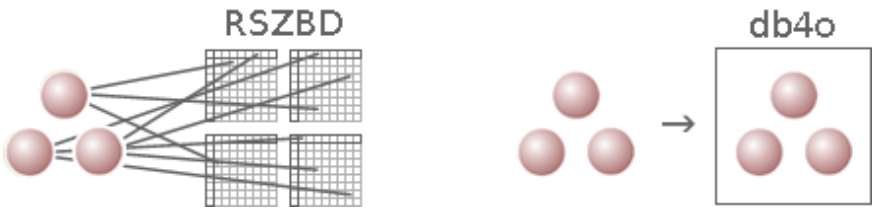
Jednym z głównych powodów rozwoju obiektowych baz danych było rozwiązanie tzw. problemu niezgodności impedancji (ang. impedence mismatch). Problem ten powstaje ze względu na fundamentalne różnice pomiędzy paradygmatem relacyjnym a obiektowym. Na paradygmacie relacyjnym oparte są Relacyjne Systemy Zarządzania Bazą Danych (RSZBD). Natomiast obiektowe języki programowania wyrastają z paradygmatu obiektowego. Gdy łączymy ze sobą te dwa światy nieuchronnie napotykamy trudności wynikające z różnic pomiędzy nimi. Programista piszący program w duchu OOP (Object Oriented Programming) zmuszony jest wyjść ze świata obiektowego chcąc uzyskać dostęp do danych zapisanych w relacyjnej bazie danych.

Załóżmy, że w programie napisanym obiektowo istnieje potrzeba utrwalenia (ang. persist) pewnych obiektów. Typowym rozwiązaniem takiego problemu za pomocą RSZBD byłoby stworzenie w bazie danych tabeli odwzorowującej klasę obiektów, które chcemy utrwalać. Następnie należy napisać kod odpowiedzialny za zapisywanie informacji zawartych w obiektach do utworzonej tabeli. Odbywa się to poprzez wykonanie odpowiedniego zapytania SQL. Analogicznie postępujemy chcąc odczytać utrwalone dane. Wykonujemy zapytanie, powołujemy do życia nowe obiekty i

wypełniamy je danymi otrzymanymi z bazy danych. Takie podejście zdaniem autora jest kłopotliwe. Wymaga ono od programisty ciągłego manewrowania na granicy świata relacyjnego i obiektowego. Ponadto, konieczne jest używanie w kodzie zapytań SQL co jest nieczytelne, czyni kod niepodatnym na automatyczny refactoring i jest źródłem zagrożenia bezpieczeństwa danych (SQL injection).

Przedstawiony scenariusz jest oczywiście uproszczony w celu uwypuklenia problemu niekompatybilności paradygmatów relacyjnego i obiektowego. W codziennej praktyce zwykle stosuje się gotowe narzędzia ORM (Object-Relational Mapping) takie jak Hibernate czy Toplink. Ułatwiają one tworzenie odwzorowania obiekt – relacja uwalniając programistę od żmudnego procesu ręcznego odwzorowywania obiektów na tabele. Są one z sukcesem stosowane, jednak mają wiele wad. Z tego powodu wielu ekspertów uważa je za problematyczne - obniżają wydajność, komplikują projekt, zwiększają koszty utrzymania oprogramowania. Ten temat szeroko omawia [3].

Obiektowe Systemy Zarządzania Bazą Danych (OSZBD) wyrastają bezpośrednio z paradygmatu programowania obiektowego. Zasobami przechowywanymi w obiektowej bazie danych są obiekty. Dzięki temu komunikacja programu napisanego w stylu OOP z obiektową bazą danych jest prosta i elegancka. Gdy chcemy utrwalić obiekt czynimy to w jednej linijce kodu. Wczytując dane otrzymujemy obiekty, z którymi możemy pracować. Nie ma tu potrzeby odwzorowywania tabel na obiekty, które to zwykle jest konieczne w przypadku baz relacyjnych. Tę fundamentalną różnicę pomiędzy RSZBD a OSZBD materiały informacyjne obiektowej bazy danych db4o przedstawiają za pomocą poniższej ilustracji.



Ilustracja 1: Odwzorowanie obiektów w RSZBD i OSZBD

1.2 db4o

Początek db4o datuje się na rok 2000 kiedy Carl Rosenberger rozpoczął pierwsze prace nad implementacją własnego silnika obiektowej bazy danych. Efektem jego wysiłków była pierwsza wersja db4o wydana w 2001 roku. Trzy lata po tym wydarzeniu założył on firmę Db4objects Incorporated, która od tej pory zajmowała się rozwojem db4o. Od tego czasu db4o stale się rozwijała zyskując nowe funkcjonalności i popularność. W roku 2008 amerykańska korporacja Versant Corporation kupiła prawa do bazy danych db4o a Carl Rosenberger został pracownikiem tejże firmy.

db4o to obiektowa baza danych o otwartych źródłach. Jest dostępna za darmo o ile opublikujemy źródła projektu, w którym jej używamy. W przypadku projektów o zamkniętych źródłach db4o sprzedawana jest na licencji komercyjnej wraz ze wsparciem technicznym. Db4o jest równolegle rozwijana dla dwóch platform – .NET i Java.

W poprzednim podrozdziale przedstawiono różnice pomiędzy RSZBD a OSZBD. Fundamentalnie różne podejście do przechowywania danych w obu systemach zarządzania bazą danych skutkuje odmiennymi metodami dostępu do nich. W chwili obecnej baza danych db4o posiada cztery interfejsy dostępu do danych:

- Query By Example
- Native Queries
- LINQ
- SODA

Należy w tym miejscu zaznaczyć, iż LINQ dostępny jest tylko w wersji db4o przeznaczonej dla platformy .NET. Natomiast pozostałe trzy metody dostępu do danych znajdziemy w obu wersjach db4o.

Omówmy dostępne w db4o interfejsy dostępu do utrwalonych obiektów.

W przykładach demonstrujących użycie poszczególnych interfejsów będziemy posługiwać się klasą `Punkt` zdefiniowaną następująco[1]:

1 Wszystkie fragmenty kodu zawarte w pracy są napisane w języku Java - o ile nie zaznaczono inaczej.

```
public class Punkt {
   private int x;
   private int y;

   public Punkt() {}

   public Punkt(int x, int y) {
      this.x = x;
      this.y = y;
   }

   public void setX(int x) {
      this.x = x;
   }

   public int getX() {
      return x;
   }

   public void setY(int y) {
      this.y = y;
   }

   public int getY() {
      return y;
   }
}
```

W poniższych przykładach jak i w dalszej części pracy zakładamy, że istnieje obiekt `db` typu `ObjectContainer`, który jest naszym punktem dostępu do danych zapisanych w bazie.

1.2.1 Query By Example

Query By Example (QBE) to prosty mechanizm dostępu do danych zalecany

tylko dla użytkowników zapoznających się dopiero z db4o. Używając QBE przekazujemy db4o wzorzec obiektu. Jako rezultat otrzymujemy zbiór obiektów, których pola zgadzają się z wartościami zdefiniowanymi we wzorcowym obiekcie. Pobranie wszystkich punktów o współrzędnej x równej 100 wygląda następująco:

```
1  Punkt wzorzec = new Punkt();
2  wzorzec.setX(100);
3  ObjectSet result = db.queryByExample(wzorzec);
```

QBE jest proste w użyciu, jednak ma wiele poważnych ograniczeń.

- Pola obiektu wzorcowego i kandydatów zawsze porównywane są równościowo. Nie ma możliwości szukania obiektów wpadających w pewien zakres wartości danego pola.
- QBE nie sprawdza się także jeżeli szukamy obiektów o wartości pola równej wartości domyślnej dla danego typu danych. Gdyby w drugiej linii naszego przykładu nadać polu `x` wartość 0 (zamiast 100), db4o uznałoby że nie nakładamy na to pole żadnego ograniczenia bo wartość 0 to domyślna wartość dla pola typu `int`. Jako rezultat otrzymalibyśmy wszystkie obiekty klasy `Punkt` utrwalone w bazie danych.
- QBE nie pozwala na złożone zapytania uwzględniające logiczne łączenie poszczególnych ograniczeń (AND, OR, NOT, ...)

Z tych powodów Query By Example nie jest zalecaną metodą tworzenia zapytań do bazy danych db4o.

1.2.2 Native Queries

Aby pokonać ograniczenia QBE powstał mechanizm nazwany Native Queries (NQ). Koncepcję tego interfejsu zaproponowali wspólnie Carl Rosenberg i William Cook w roku 2005[2]. NQ posiada cały szereg zalet, dzięki którym jest zalecanym przez

2 C. Rosenberg, W. Cook: *"Native Queries for Persistent Objects, A Design White Paper"*, http://www.cs.utexas.edu/users/wcook/papers/NativeQueries/NativeQueries8-23-05.pdf (2005)

twórców db4o interfejsem dostępu do danych. Zapytania NQ nie używają łańcuchów znakowych. Pisane są w czystej Javie bądź C#, zależnie od wersji db4o. Zatem pisząc zapytania NQ możemy korzystać ze wszystkich ułatwień oferowanych przez nowoczesne środowiska programistyczne takie jak Visual Studio czy Eclipse. Zapytania NQ poddają się automatycznemu refactoringowi a pisząc je możemy korzystać z autouzupełniania składni. W momencie kompilacji zapytanie jest sprawdzane pod względem zgodności typów jak i poprawności syntaktycznej. Te, nieosiągalne dla metod opartych na zapytaniach w postaci łańcuchów znakowych, ułatwienia sprawiają, że NQ uważane jest za bardzo bezpieczny interfejs.

Zasada tworzenia zapytań jest prosta. Programista wypełnia ciało metody `match()`, która jest wywoływana dla każdego obiektu szukanej klasy. Każdy obiekt, dla którego metoda `match()` zwróci wartość `true`, znajdzie się w zbiorze rezultatów zapytania. Oczywiście strategia sprawdzania po kolei każdego obiektu danej klasy nie może być efektywna. Dlatego Native Queries są konwertowane na zapytania typu SODA przed wywołaniem. Silnik db4o optymalizuje zapytania NQ i wykorzystuje indeksy kiedy jest to możliwe - nie ma więc potrzeby faktycznego instancjonowania wszystkich utrwalonych obiektów, żeby wywołać na każdym z nich metodę `match()`. Jest to niewidoczne z punktu widzenia programisty tworzącego zapytanie NQ.

Przypomnijmy zapytanie z poprzedniego podrozdziału - szukamy wszystkich punktów o współrzędnej x równej 100. Za pomocą Native Queries wyrazimy to w następujący sposób:

```
1  List <Punkt> punkty = db.query(new Predicate<Punkt>() {
2      public boolean match(Punkt punkt) {
3          return punkt.getX() == 100;
4      }
5  });
```

Native Queries ma dużo większe możliwości niż QBE co pokazuje kolejny przykład:

```
1  List <Punkt> punkty = db.query(new Predicate<Punkt>() {
2      public boolean match(Punkt punkt) {
3          return (punkt.getX() == 100)
4              || (punkt.getY() > 0 && punkt.getY() < 100);
5      }
6  });
```

Powyższe zapytanie zwraca listę punktów o współrzędnej x równej 100 lub współrzędnej y należącej do przedziału (0, 100). Przykład ten pokazuje siłę NQ - do wyrażania zapytań używamy tylko konstrukcji języka programowania (w tym przypadku Javy). Nie ma tu problemów związanych z użyciem obcych języków wewnątrz programu jak to się dzieje w przypadku SQL, HQL [6] czy OQL [5].

1.2.3 LINQ

Language Integrated Queries, szerzej znane pod nazwą LINQ, to technologia stworzona przez firmę Microsoft i udostępniona wraz z .NET Framework 3.5 w listopadzie 2007 roku. LINQ jest językiem zapytań, który można stosować z wieloma różnymi źródłami danych. Technologia ta tworzy warstwę abstrakcji pomiędzy programistą a źródłem danych. Pozwala stosować jeden język zapytań niezależnie od tego czy źródłem danych jest relacyjna baza danych, obiektowa baza danych, plik XML czy tablica. Składnia LINQ jest podobna do składni SQL-a, jednak w odróżnieniu od niego LINQ jest w pełni zintegrowany z językiem programowania. Dzięki temu tworzenie zapytań w LINQ, podobnie jak w przypadku Native Queries, jest bezpieczniejsze i wygodniejsze niż w przypadku języków opartych na łańcuchach znakowych. Pełna integracja z językiem programowania pozwala na kontrolę typów w momencie kompilacji, autouzupełnianie składni oraz automatyczny refactoring.

LINQ dostarcza mechanizmy dzięki, którym można zastosować go do wielu różnych źródeł danych. Dzięki temu twórcy db4o mogli dodać wsparcie dla LINQ w db4o dla platformy .NET. Używanie LINQ w db4o sprowadza się do użycia w logiczny sposób słów kluczowych dostarczonych przez .NET Framework. Przyjrzyjmy się

przykładowemu zapytaniu LINQ napisanemu w języku C#.

```
1  IEnumerable<Punkt> result = from Punkt p in db
2                              where p.getX() == 100
3                              select p;
```

Tak jak w poprzednich przykładach, zapytanie to zwraca listę punktów o współrzędnej x równej 100. Przyjrzyjmy się nieco bardziej skomplikowanemu zapytaniu. Użyjemy w tym celu przykładu, którym posłużyliśmy się już podczas omawiania Native Queries - "zbiór punktów o współrzędnej x równej 100 lub współrzędnej y należącej do przedziału (0, 100)"

```
1  IEnumerable<Punkt> result = from Punkt p in db
2                              where
3                              (p.getX() == 100) ||
4                              (p.getY() > 0 && p.getY() < 100)
5                              select p;
```

Składnia LINQ jest intuicyjnie zrozumiała dla programistów zaznajomionych z językiem SQL co przyczynia się do rosnącej popularności tej technologii.

1.2.4 SODA

Skrót SODA oznacza Simple Object Database Access co tłumaczy się jako prosty dostęp do obiektowej bazy danych. Jest to niskopoziomowe API umożliwiające tworzenie zapytań do bazy danych db4o. SODA było pierwszym interfejsem dostępu do utrwalonych obiektów oferowanym w db4o. Powstałe później interfejsy zostały napisane w oparciu o istniejący mechanizm SODA – zapytania napisane za pomocą LINQ, Query By Example czy Native Queries są najpierw tłumaczone na zapytania SODA przez silnik db4o i dopiero w takiej postaci są faktycznie wykonywane. Zapytania typu SODA są szybkie. Twórcy db4o zalecają wręcz przepisywanie zapytań NQ czy LINQ na zapytania typu SODA w przypadkach kiedy tamte są zbyt wolne. Chociażby z tego

powodu warto znać i używać SODA API.

Z drugiej strony SODA nie jest pozbawiona wad. Tworzenie zapytań wymaga używania łańcuchów znakowych, co sprawia iż SODA w przeciwieństwie do Native Queries czy LINQ, nie poddaje się refactoringowi. Co więcej, błąd w nazwie pola nie będzie wykryty podczas kompilacji ani podczas wykonywania programu. Odwołanie poprzez łańcuch znakowy do nieistniejącego pola klasy zaowocuje pustym zbiorem wynikowym co może być źródłem trudnych do znalezienia błędów. SODA jest również mało intuicyjna. Z tych powodów Native Queries są zalecaną metodą tworzenia zapytań dla typowych zastosowań.

W niniejszej pracy język zapytań SODA spełnia bardzo ważną rolę, z tego powodu poświęcimy mu więcej miejsca niż interfejsom omawianym na poprzednich stronach.

Przyjrzyjmy się przykładom zapytań typu SODA. Powtórzymy zapytanie, które zrealizowaliśmy już wcześniej za pomocą QBE, NQ i LINQ - "zbiór punktów o współrzędnej x równej 100":

```
1  Query query = db.query();
2  query.constrain(Punkt.class);
3  query.descend("x").constrain(100);
4  ObjectSet result = query.execute();
```

Tworzenie zapytania SODA polega na dosłownym, niskopoziomowym wyrażaniu warunków, które mają spełniać obiekty. W drugiej linii powyższego przykładu określamy, że szukamy obiektów klasy `Punkt`. W trzeciej linii definiujemy kolejne warunki - obiekt musi posiadać pole o nazwie "x" i pole to musi mieć wartość 100.

Zobaczmy jak za pomocą SODA można zrealizować zapytanie - "zbiór punktów o współrzędnej x równej 100 lub współrzędnej y należącej do przedziału (0, 100)"

```
1  Query query = db.query();
```

```
2  query.constrain(Punkt.class);
3  Constraint c1 = query.descend("x").constrain(100);
4  Constraint c2 = query.descend("y").constrain(0).greater();
5  Constraint c3 = query.descend("y").constrain(100).smaller();
6  c3.and(c2);
7  c1.or(c3);
8  ObjectSet result = query.execute();
```

Pierwsze trzy linie to niemal powtórzenie poprzedniego zapytania. W czwartej i piątek linii widzimy zastosowanie metod zmieniających tryb ewaluacji ograniczeń. Domyślny tryb ewaluacji ograniczeń traktuje argument metody `constrain()` jako wzorzec. Ewaluacja takiego zapytania zwraca tylko obiekty o ograniczanym polu równym podanemu wzorcowi. Wywołanie metody `greater()` na obiekcie klasy `Constraint` zmienia sposób interpretacji ograniczenia. Sprawia, że silnik db4o zwróci obiekty o wartości ograniczanego pola większej niż wartość podana jako wzorzec. W liniach szóstej i siódmej definiujemy logiczne relacje pomiędzy zdefiniowanymi wcześniej ograniczeniami. Linia szósta jest na dobrą sprawę zbędna, gdyż ograniczenia są domyślnie łączone relacją AND. W tym przypadku autor zdecydował się pozostawić ten nadmiarowy kod dla celów demonstracyjnych jak i dla podkreślenia intencji stojącej za kodem.

Poniżej przedstawiamy krótkie omówienie dostępnych modyfikatorów trybu ewaluacji ograniczeń.

- `greater()`, `smaller()` - tworzy warunki odpowiednio "większe niż", "mniejsze niż". Mogą być stosowane do argumentów typu String. Wówczas porównywanie napisów następuję według porządku leksykograficznego.
- `equal()` - stosowane w połączeniu z `greater()`, `smaller()` do tworzenia ograniczeń odpowiednio "większe lub równe", "mniejsze lub równe".
- `like()` - obiekt spełnia warunek, jeżeli ograniczane pole zawiera w sobie napis podany jako argument metody `constrain()`. Wielkość znaków nie ma znaczenia.

- `contains()` - podobnie jak `like()`. Wielkość znaków ma znaczenie.
- `startsWith()`, `endsWith()` - obiekt spełnia warunek jeżeli ograniczane pole odpowiednio zaczyna się lub kończy od napisu podanego jako argument metody `constrain()`. Metody te przyjmują jako argument zmienną typu `boolean`. Określa ona czy podczas ewaluacji ograniczenia brać pod uwagę wielkość znaku.

W porównaniu do LINQ czy Native Queries zapytania typu SODA są rozwlekłe i niezrozumiałe. Na pierwszy rzut oka nie jesteśmy w stanie stwierdzić jakie rezultaty będzie generowało dane zapytanie. Dopiero po dokładnym prześledzeniu kodu zapytania staje się to jasne. Z drugiej strony niskopoziomowe API daje programiście dużą kontrolę nad grafem zapytania, które ostatecznie będzie wykonywane przez silnik db4o.

Z naszego punktu widzenia SODA ma bardzo ważną zaletę - proces tworzenia zapytania jest w gruncie rzeczy prosty. Za pomocą metody `descend()` możemy dostać się do dowolnego pola klasy, następnie nałożyć na nie ograniczenie metodą `constrain()`, dodać do ograniczeń modyfikatory (`greater()`, `smaller()`, etc), łączyć zapytania logicznie za pomocą metod `and()`, `or()`, `not()`. Dzięki tej prostocie tworzenie zapytań SODA łatwo jest opisać za pomocą algorytmu co jest kluczowe w niniejszej pracy.

2 Cel pracy

Założonym celem pracy było stworzenie narzędzia pozwalającego na wygodny i intuicyjny dostęp do bazy danych db4o. Planowane funkcjonalności:

- odczytywanie bazy danych db4o i wyświetlanie klas przechowywanych w niej obiektów
- tworzenie i edytowanie grafów zapytań
- generowanie kodu zapytania na podstawie grafu

- zapisywanie grafów w postaci plików graficznych

3 Przyjęte rozwiązania

Squared, czyli edytor zapytań będący efektem niniejszej pracy powstał jako wtyczka do Zintegrowanego Środowiska Programistycznego Eclipse[3]. Decyzja ta była motywowana dużą popularnością platformy Eclipse, co ułatwia dostęp do edytora potencjalnym zainteresowanym. Rozszerzanie istniejącej, stabilnej platformy daje również komfort korzystania ze sprawdzonej bazy kodu, który w innym przypadku autor zmuszony byłby napisać, a który nie jest istotny dla tematu niniejszej pracy. Dokładniejsze omówienie platformy Eclipse znajduje się w rozdziale drugim niniejszej pracy w podrozdziale 5.1 Platforma Eclipse.

Platforma Eclipse oferuje rozszerzenie przeznaczone specjalnie do tworzenia graficznych edytorów o nazwie Graphical Editing Framework (GEF)[4]. W niniejszej pracy jest on intensywnie wykorzystywany. Użycie biblioteki GEF pozwoliło autorowi skupić się na centralnym problemie niniejszej pracy – wizualnym tworzeniu zapytań typu SODA. Czytelników zainteresowanych rozszerzeniem GEF zachęcam do zapoznania się z treścią podrozdziału 5.2 Graphical Editing Framework.

Spośród czterech dostępnych interfejsów dostępu do danych, autor wybrał SODA jako API, w którym generowane będą zapytania. Zapytania SODA są szybkie. Ponadto zapytanie typu SODA da się w naturalny sposób przedstawić w postaci grafu. Te dwie zalety zdecydowały o wyborze tego właśnie interfejsu. QBE dyskwalifikują jego ograniczone możliwości. Kod zapytań LINQ i Native Queries byłby trudny do wygenerowania ze względu na skomplikowanie tych interfejsów. Ponadto LINQ nie jest dostępny w db4o dla języka Java. Z tych powodów SODA wydaje się być naturalnym wyborem.

3 Eclipse IDE - http://eclipse.org/
4 Graphical Editing Framework - http://www.eclipse.org/gef/

4 Organizacja pracy

Mając za sobą wprowadzenie w kolejnych rozdziałach skupimy się na graficznym edytorze zapytań stworzonym w ramach niniejszej pracy.

Rozdział drugi to opis stworzonego oprogramowania. Zawiera on omówienie projektu i przedstawienia architektury oprogramowania. Następnie następuje szczegółowy opis funkcjonalności wraz z typowymi przypadkami użycia. Rozdział kończy podsumowanie użytych bibliotek i technologii oraz uzasadnienie ich użycia.

Rozdział trzeci przedstawia rezultaty pracy - co udało się osiągnąć, jakie problemy pozostały nierozwiązane. Następnie omówione zostały wady i zalety stworzonego oprogramowania, propozycje jego rozwinięcia oraz krótkie porównanie z innymi dostępnymi na rynku produktami o podobnych funkcjonalnościach. Następnie zaprezentowane są ciekawe problemy implementacyjne napotkane podczas prac nad edytorem Squared.

Najistotniejsze fragmenty pracy to te ukazujące ideę i możliwości stworzonego oprogramowania - odpowiednio rozdziały.

Do pracy dołączone są następujące załączniki:

- Załącznik A – Nośnik DVD zawierający niniejszą pracę magisterską w postaci pliku PDF a także edytor zapytań Squared w postaci kodu źródłowego oraz binarnej.
- Załącznik B – Podręcznik Użytkownika

ROZDZIAŁ II

Implementacja

1 Analiza wymagań

Graficzny edytor zapytań powinien pozwalać użytkownikowi stworzyć graf zapytania. Następnie na podstawie tego grafu edytor powinien wygenerować poprawny syntaktycznie i semantycznie kod zapytania używający SODA API. Jest to ogólnie sformułowany podstawowy cel, który został postawiony przed oprogramowaniem stworzonym w ramach niniejszej pracy. Podczas analizy wymagań zidentyfikowano konkretnie funkcjonalności, które powinno posiadać tworzone oprogramowanie:

- Edytor powinien otwierać dowolny plik bazy danych db4o i wyświetlać obiekty zawarte w tej bazie
- Edytor powinien pozwalać na stworzenie grafu zapytania poprzez dwukrotne kliknięcie na dowolny obiekt z wczytanej bazy danych
- Edytor powinien pozwalać na rozwinięcie dowolnego węzła o ile jest to logicznie możliwe
- Edytor powinien pozwalać na dodawanie ograniczeń do dowolnego węzła grafu
- Edytor powinien pozwalać na ustanawianie logicznych relacji pomiędzy istniejącymi ograniczeniami
- Edytor powinien generować kod zapytania na podstawie grafu
- Edytor powinien zapisywać graf zapytania w postaci pliku graficznego

Według zamierzeń autora głównym polem zastosowań stworzonego oprogramowania jest dydaktyka. Edytor ma wspomagać proces zapoznawania się z

SODA API. Dlatego ważne jest aby było proste w obsłudze a interfejs użytkownika był intuicyjny i przejrzysty.

2 Graf zapytania

W zapytaniach typu SODA programista tworzy graf zapytania za pomocą wywołań SODA API. Graf ten określa jakie warunki muszą spełniać obiekty aby znaleźć się w zbiorze wynikowym zapytania. Tworzenie zapytań typu SODA opiera się na wywołaniach metody `descend()`, która przechodzi w głąb klasy, będącej punktem wejścia, pozwalając programiście dostać się do jej zagnieżdżonych obiektów. Za pomocą metody `constrain()` określa się ograniczenia nakładane na obiekty kandydujące do znalezienia się w zbiorze wynikowym.

Dołączany do dystrybucji db4o dokument „db4o Reference Documentation” opisuje proces tworzenia zapytania SODA jako kombinację poniższych pięciu podstawowych idiomów:

- utwórz główny węzeł grafu zapytania

```
Query root = db.query();
```

- dodaj ograniczenia do dowolnego węzła grafu

```
root.constrain(Foo.class);
```

- przejdź z dowolnego węzła do dowolnego z jego podwęzłów

```
Query foo = root.descend("foo");
Query bar = root.descend("bar");
```

- dodaj kolejne ograniczenia do dowolnego węzła

```
Constraint fooConstraint = foo.constrain(5);
Constraint barConstraint = bar.constrain("F");
```

- ustaw tryb ewaluacji ograniczenia

```
fooConstraint.greater().not();
```

```
barConstraint.startsWith(true);
barConstraint.like();
```

Zdaniem autora warto do tej listy dodać punkt szósty:

- ustanów logiczne relacje pomiędzy ograniczeniami

```
fooConstraint.or(barConstraint);
fooConstraint.and(barConstraint);
```

Powyższe idiomy można dowolnie, logicznie łączyć w celu uzyskania złożonych zapytań z wieloma warunkami. Prześledźmy proces tworzenia zapytania na przykładach.

2.1 Przykłady

Zdefiniujmy klasę `Pracownik`, która posłuży nam jako przykład:

```
public class Pracownik {
      private Pracownik szef;
      private String imie;
      private int placa;
}
```

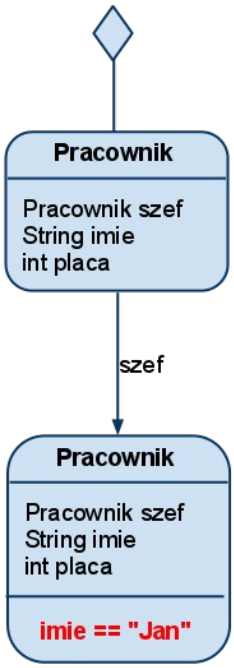
Przykład 1 - „pracownicy, których szef ma na imię Jan”

Szukamy wszystkich pracowników, których szef ma na imię Jan. Za pomocą SODA takie zapytanie możemy wyrazić w następujący sposób:

```
1  Query query = db.query();
2  query.constrain(Pracownik.class);
3  query.descend("szef").descend("imie").constrain("Jan");
4  ObjectSet result = query.execute();
```

W pierwszej linii uzyskujemy dostęp do obiektu `Query` wywołując metodę `query()` na obiekcie `ObjectContainer`. Następnie określamy, że chcemy uzyskać obiekty klasy `Pracownik`. Trzecia linia jest zdecydowanie najciekawsza – za pomocą metody `descend()` przechodzimy w głąb klasy `Pracownik`. Najpierw przechodzimy do pola `szef`, a z niego do pola `imie`. W tym momencie mamy dostęp do interesującego nas pola zatem nakładamy na nie ograniczenie. W linii czwartej wykonujemy zapytanie otrzymując kolekcję obiektów klasy `Pracownik`.

Kluczowym elementem SODA API jest metoda `descend()`. Pozwala ona dostać się do dowolnego pola klasy co jest niezbędne, w przypadku gdy chcemy nałożyć na nie ograniczenie. Tę podróż w głąb klasy można intuicyjnie przedstawić w postaci diagramu przedstawionego na poniższej ilustracji.



Ilustracja 2: Proponowana postać grafu zapytania

Romb reprezentuje punkt wejścia a każdy węzeł grafu odpowiada obiektowi utrwalonemu w bazie danych. Węzeł połączony z rombem określa klasę obiektów zwracanych przez zapytanie – w tym przypadku będą to obiekty klasy `Pracownik`.

Niżej widzimy wychodzącą z węzła krawędź oznaczoną etykietą „szef”. Krawędź ta oznacza, że sięgamy do pola `szef` klasy `Pracownik`. W rozwiniętym polu `szef` sięgamy do pola `imie` i stawiamy warunek `imie == „Jan”`.

Zdaniem autora powyższy graf w sposób czytelny i intuicyjny reprezentuje zapytanie „pracownicy, których szef ma na imię Jan”. Za pomocą podobnych grafów można wizualizować zapytania typu SODA, co ułatwia i przyspiesza ich analizę. Można argumentować, iż kod zapytania jest na tyle prosty, że nie ma potrzeby odwoływać się do pomocy wizualnych. O ile w przypadku powyższego zapytania można się z taką argumentacją zgodzić, to w przypadku bardziej skomplikowanych zapytań diagram jest nieocenioną pomocą. Postaram się pokazać to za pomocą następnego przykładu.

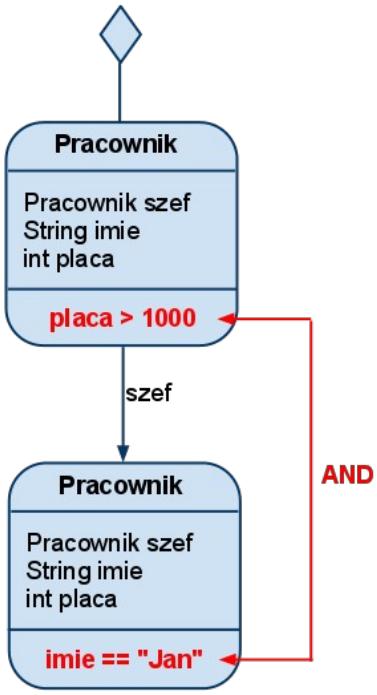
Przykład 2 - „pracownicy, których szef ma na imię Jan zarabiający więcej niż 1000”

W przykładzie drugim pokażemy, w jaki sposób można łączyć ze sobą pojedyncze ograniczenia tworząc złożone zapytania. Weźmy zapytanie z pierwszego przykładu jako punkt wyjścia i dodajmy do niego dodatkowe ograniczenie.

```
1  Query query = db.query();
2  query.constrain(Pracownik.class);
3  Query imieSzefa = query.descend("szef").descend("imie");
4  Constraint imieJan = imieSzefa.constrain("Jan");
5  Query placa = query.descend("placa");
6  Constraint placa1000 = placa.constrain(1000).greater();
7  placa1000.and(imieJan);
8  ObjectSet result = query.execute();
```

Pierwsze cztery linijki odpowiadają zapytaniu z Przykładu 1. W linii piątej przechodzimy do pola `placa`, żeby w linii szóstej nałożyć na nie ograniczenie. Interesują nas pracownicy z pensją większą niż 1000. W tym celu stosujemy metodę `greater()`. W linii siódmej łączymy oba utworzone ograniczenia za pomocą metody `and()`.

Diagram 2 przedstawia graf powyższego zapytania.



Ilustracja 3: Graf zapytania z ograniczeniami

Oczywiście przy odrobinie wprawy analizując kod zapytania z przykładu drugiego można szybko odczytać jakie zapytanie on realizuje. Jednak graficzna reprezentacja zapytania wydaje się być o wiele bardziej czytelna. Jest to istotne szczególnie dla osób, które dopiero zapoznają się z zapytaniami SODA. Zwłaszcza jeżeli jest to ich pierwszy kontakt z obiektowymi bazami danych. Stąd motywacja do stworzenia Squared - graficznego edytora zapytań SODA będącego przedmiotem niniejszej pracy.

3 Projekt oprogramowania

Projekt Squared w dużej mierze opiera się na wzorcu projektowym Model-Widok-Kontroler[5] (MVC, Model-View-Controller). Dzięki zastosowaniu tego wzorca udało się oddzielić kod odpowiedzialny za logikę sterowania (Kontroler), model danych (Model) i interfejs użytkownika (Widok).

5 Wzorzec projektowy Model-Widok-Kontroler - http://www.martinfowler.com/eaaDev/uiArchs.html

Widok to zestaw klas przedstawiających model w formie graficznej. Edytor posiada dwa widoki:

- Widok diagramu zapytania - prezentuje graf zapytania. W skład tego widoku wchodzą klasy opisujące graficzną reprezentację modelu w postaci diagramu.
- Widok kodu zapytania - prezentuje wygenerowany kod zapytania.

Modelem danych w naszym przypadku są klasy opisujące diagram zapytania:

- `Diagram` - klasa zawiera referencje do wszystkich elementów diagramu (zarówno węzłów jak i połączeń między węzłami). W szczególności klasa `Diagram` posiada referencję do korzenia grafu zapytania - punktu wejścia diagramu.
- `Node` - reprezentuje węzeł grafu
- `NodeLink` - reprezentuje połączenie między węzłami grafu

Kontroler - zestaw klas zawierających logikę edytora. Klasy kontrolera przechwytują akcje użytkownika i dokonują odpowiednich zmian zarówno w modelu danych jak i w widoku diagramu zapytania. Widok kodu zapytania nie jest bezpośrednio modyfikowany przez kontroler. Zmiany w modelu inicjowane przez kontroler są propagowane do widoku kodu zapytania poprzez mechanizm notyfikacji implementowany w klasie `Diagram`. Mechanizm ten znany jest w literaturze jako wzorzec projektowy Obserwator[6]. Obiekty, które chcą być informowane o zmianach zachodzących w modelu danych rejestrują się jako obserwatorzy w klasie `Diagram`. Takie obiekty będą notyfikowane o dodaniu nowego węzła, dodaniu ograniczenia do węzła, etc. Klasa reprezentująca widok kodu zapytania rejestruje się jako obserwator zmian w modelu danych. Gdy tylko nastąpi zmiana w modelu, widok zapytania zostanie poinformowany o konieczności wygenerowania odświeżonego kodu zapytania. Zadanie

6 Wzorzec projektowy Obserwator - http://www.research.ibm.com/designpatterns/example.htm

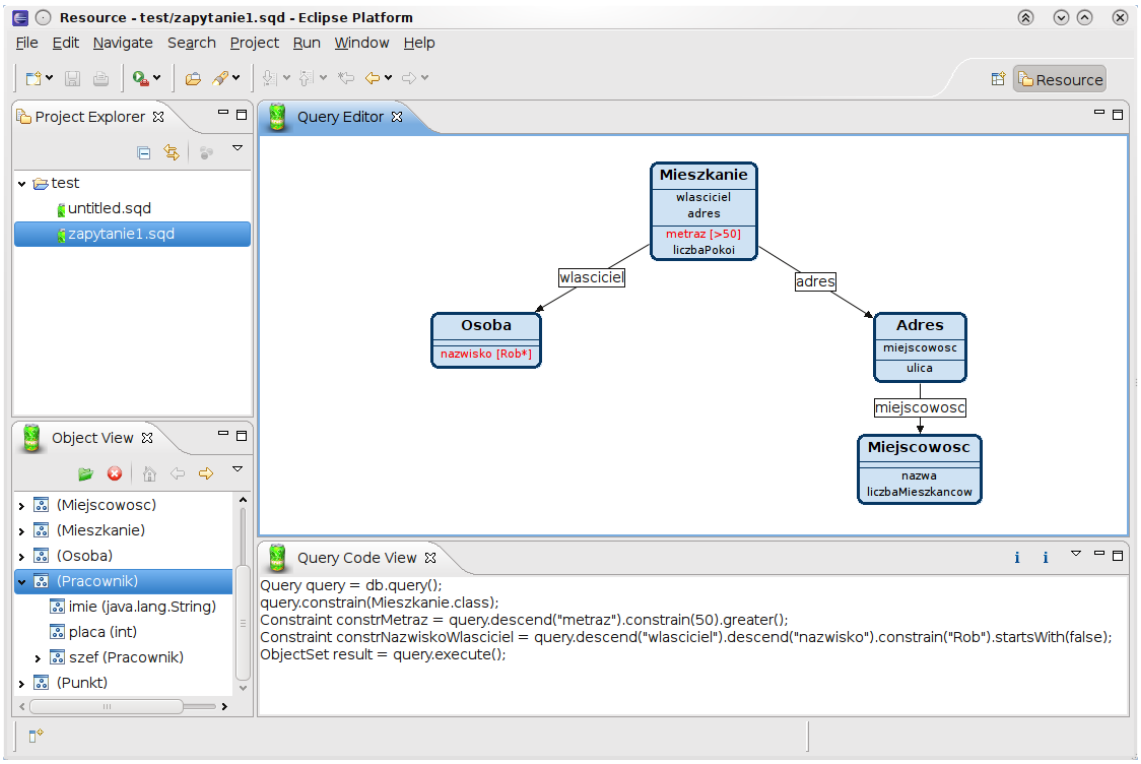
to zleca klasie `QueryBuilder`, która analizując model danych tworzy kod zapytania odpowiadający aktualnemu diagramowi. Dzięki takiemu rozwiązaniu nie musimy wywoływać żądania odświeżenia kodu zapytania w każdym miejscu w kodzie, w którym modyfikowany jest model danych. Każda zmiana w modelu wyzwala generowanie kody zapytania - mamy pewność, że kod zapytania zawsze odzwierciedla aktualny stan modelu.

4 Możliwości i funkcjonalności

W tym rozdziale omówimy możliwości Squared oraz przedstawimy typowe przypadki użycia. Szczegółowy podręcznik użytkownika znajduje się w załączniku A dołączonym do niniejszej pracy.

4.1 Interfejs edytora

Poniższy zrzut ekranu przedstawia widok edytora podczas pracy.



Ilustracja 4: Edytor Squared podczas pracy

Squared posiada cztery obszary robocze nazywane również widokami. Ta druga nazwa jest częściej stosowana w społeczności Eclipse. Nie należy mylić widoków w rozumieniu wzorca MVC z widokami w Eclipse. W przypadku naszego projektu istnieją pewne powiązania pomiędzy nimi ale są to byty z zupełnie różnych kategorii.

Pokrótce omówimy przeznaczenie każdego z widoków.

Project Explorer

Widok umiejscowiony w lewym górnym rogu ekranu. Jest to widok ogólnego przeznaczenia dostarczany przez platformę Eclipse. Służy do zarządzania zasobami projektów. W naszym przypadku zasobami projektu są pliki diagramów zapytań.

Object View

W lewym dolnym rogu widzimy widok obiektów (Object View). Widok ten zapewnia podgląd bazy danych db4o. Pozwala on otworzyć dowolny plik bazy danych db4o i wyświetlić obiekty w nim utrwalone. Dowolny obiekt z listy możemy rozwinąć aby zobaczyć jego pola. Dwukrotne kliknięcie na dowolny obiekt z listy tworzy nowy diagram, którego punktem wejścia jest kliknięty obiekt.

Query View

W dolnej części ekranu widzimy widok kodu zapytania. Tutaj pojawia się kod wygenerowany na podstawie diagramu. Kod zapytania można zaznaczyć i skopiować w celu wygodnego wklejenia do źródeł programu.

Query Editor

Sercem programu jest widok edycji zapytania. Zajmujący większą część przestrzeni roboczej widok pozwala na tworzenie diagramu zapytania. Użytkownik zaczyna pracę z edytorem Squared poprzez dwukrotne kliknięcie na dowolny obiekt w widoku obiektów (Object View). To powoduje wyświetlenie się w widoku edycji zapytania pierwszego węzła grafu zapytania. Mając pierwszy węzeł w widoku edytora użytkownik może rozwinąć pola obiekty wejściowego w nowe węzły diagramu. Przypomnijmy, że każdy węzeł grafu odpowiada obiektowi utrwalonemu w bazie

danych. Nowy węzeł będzie połączony strzałką z jego węzłem-rodzicem. Grot strzałki zawsze wskazuje dziecko. Ponadto strzałka oznaczona jest etykietą określającą, które pole węzła-rodzica rozwija węzeł-dziecko. W taki sposób możemy rozwijać diagram uzyskując dostęp do interesujących nas podobiektów.

4.2 Praca z edytorem

Pracę z edytorem Squared rozpoczynamy od otworzenia bazy danych. Czynimy to za pomocą przycisku Open database zlokalizowanego w widoku obiektów (Object View). Gdy plik bazy danych zostanie załadowany do pamięci edytor wyświetla listę klas obiektów przechowywanych w otwartej bazie danych. Na tej liście użytkownik znajduje interesującą go klasę i klika dwukrotnie. To powoduje, że w widoku edycji zapytania (Query Editor) pojawia się pierwszy węzeł grafu zapytania. W ten sposób użytkownik rozpoczyna tworzenie grafu zapytania.

Istotą języka zapytań jest możliwość wyłowienia z grupy obiektów tych, które z jakiegoś powodu nas interesują. W przypadku SODA API funkcjonalność ta realizowana jest za pomocą kombinacji metod `descend()` oraz `constrain()`. Jak pokazaliśmy we wcześniejszych rozdziałach, metody te pozwalają definiować warunki, które muszą spełniać obiekty aby znaleźć się w zbiorze rezultatów zapytania. Warunki te nazywane są ograniczeniami (ang. constraint).

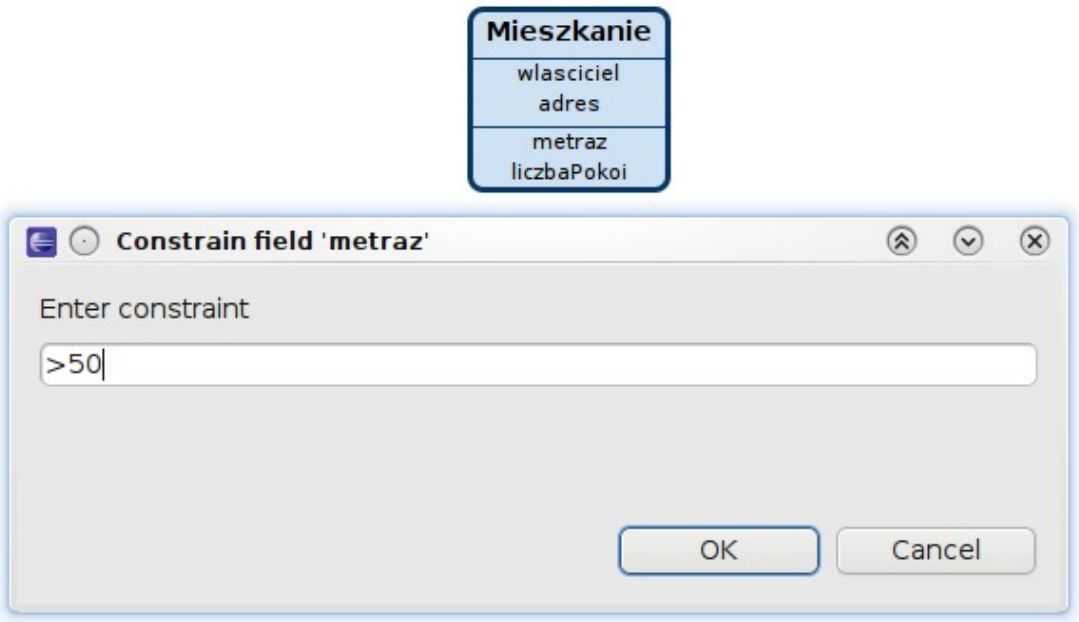
Rozważmy klasę `Mieszkanie`.

```
1  class Mieszkanie {
2    Osoba wlasciciel;
3    Adres adres;
4    double metraz;
5    int liczbaPokoi;
6  }
```

W edytorze jest ona reprezentowana wizualnie jako węzeł grafu, jak widać na poniższej ilustracji.

Ilustracja 5: Węzeł - reprezentacja klasy na diagramie

Pola klasy `Mieszkanie` są na diagramie podzielone na dwie grupy. Przynależność do jednej albo drugiej grupy określa typ danego pola. Do grupy pierwszej, nazwijmy ją grupą pól rozszerzalnych, należą pola `wlasciciel` (klasy `Osoba`) oraz `adres` (klasy `Adres`). Do grupy drugiej należą pola `metraz` typu `double` oraz `liczbaPokoi` typu `int`. Grupę drugą nazwijmy grupą pól ograniczalnych. Należą do niej pola typów nazywanych w języku Java prymitywnymi (`byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`) oraz typ `java.lang.String`. Pozostałe typy danych (w tym typy definiowane przez użytkownika) należą do grupy typów rozszerzalnych. Kliknięcie na pole z grupy ograniczalnych powoduje pojawienie się okna dialogowego, w którym możemy wprowadzić warunek - ograniczenie nakładane na to pole. Zobaczmy jak wygląda to w praktyce.



Ilustracja 6: Nakładanie ograniczenia na pole metraz

Po kliknięciu w pole `metraz` pojawia się okno dialogowe pozwalające ograniczyć to pole. Wpisaliśmy ograniczenie '>50' co oznacza, że interesują nas obiekty klasy `Mieszkanie` o wartości pola `metraz` większej niż 50. Po zatwierdzeniu wprowadzonego ograniczenia widzimy, że węzeł na diagramie zmienił wygląd.

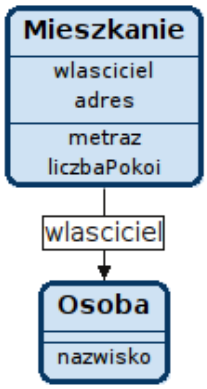
Ilustracja 7: Węzeł Mieszkanie z ograniczonym polem metraz

Pole `metraz` jest oznaczone czerwonym kolorem - w ten sposób edytor zwraca uwagę na pola z ograniczeniami. Ponadto obok nazwy ograniczonego pola w nawiasach kwadratowych pojawia się warunek ograniczający to pole. Jednocześnie w widoku kodu zapytania (Query Code View) obserwujemy pojawienie się wygenerowanego kodu zapytania:

```
1  Query query = db.query();
2  query.constrain(Mieszkanie.class);
3  Constraint constrMetraz =
query.descend("metraz").constrain(50). greater();
4  ObjectSet result = query.execute();
```

W powyższym scenariuszu udało się nam utworzyć proste zapytanie za pomocą edytora Squared.

Rozważmy bardziej skomplikowany przypadek. Tym razem chcemy otrzymać listę mieszkań, których właścicielem jest osoba o zadanym nazwisku. Chcemy zatem nałożyć ograniczenie na pole klasy `Mieszkanie`. Rozpatrywane przykład jest analogiczne do poprzedniego. W poprzednim zapytaniu nakładaliśmy ograniczenie na pole `metraz`, natomiast w aktualnym ograniczane jest pole `wlasciciel`. Istotna różnica polega na typie pola, na które nakładamy warunek. Pole `metraz` jest typu `double` należy więc do grupy pól ograniczalnych. Natomiast pole `wlasciciel` należy do grupy pól rozszerzalnych - nie możemy go w bezpośredni sposób ograniczyć. Możemy natomiast je rozwinąć. Czynimy to poprzez kliknięcie na nazwę pola, któremu chcemy się przyjrzeć.



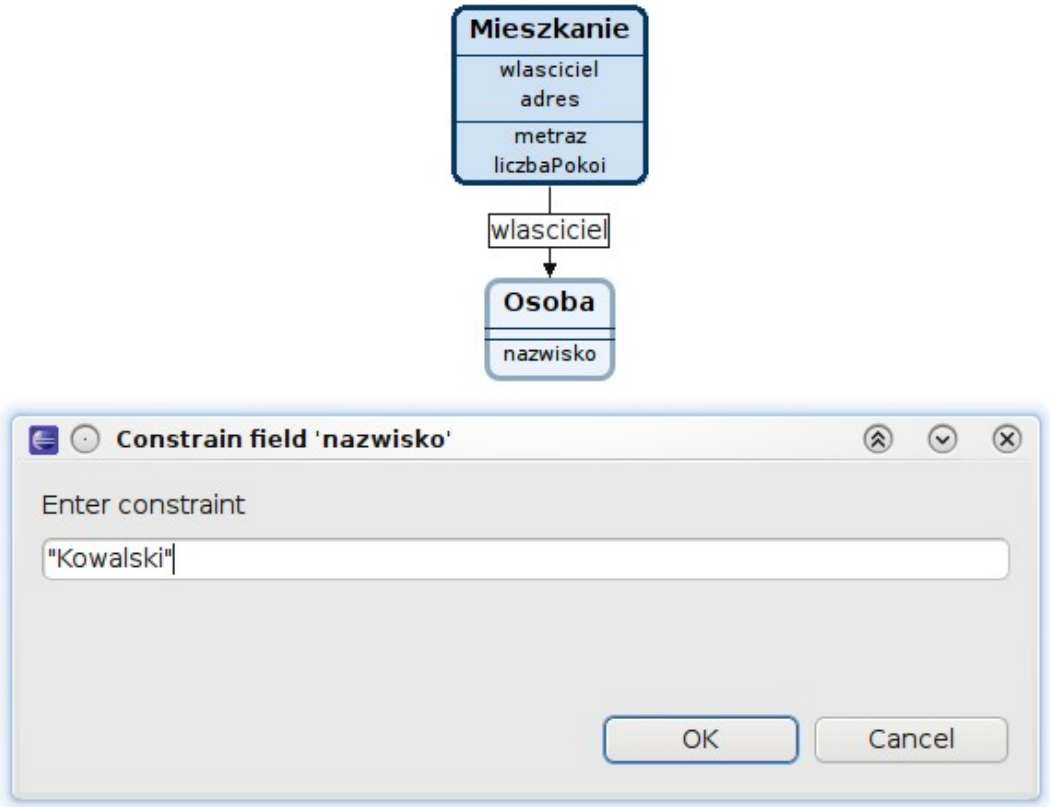
Ilustracja 8: Węzeł Mieszkanie z rozwiniętym polem wlasciciel

W efekcie na ekranie pojawia się nowy węzeł grafu zapytania. Odpowiada on polu `wlasciciel`, które rozszerzyliśmy. Pole to jest obiektem klasy `Osoba` co widać w widoku edycji edytora Squared i na powyższej ilustracji.

Przyjrzyjmy się klasie `Osoba`.

```
1  class Osoba {
2    String nazwisko;
3  }
```

Interesujące nas pole jest typu `String`. Wiemy już, że pola tego typu są ograniczalne. Zatem nasz problem jest rozwiązany - klikamy na pole nazwisko i wprowadzamy ograniczenie:



Po wprowadzeniu ograniczenia wygenerowany został kod zapytania:

```
1  Query query = db.query();
2  query.constrain(Mieszkanie.class);
3  Constraint constrNazwiskoWlasciciel =
query.descend("wlasciciel").descend("nazwisko").
constrain("Kowalski");
4  ObjectSet result = query.execute();
```

4.3 Definiowanie ograniczeń

W przykładach w poprzednim podrozdziale utworzyliśmy warunki określające, które obiekty klasy Mieszkanie mają znaleźć się w zbiorze rezultatów zapytania. Warunki te zdefiniowaliśmy używając prostego języka wyrażeń, który dokładnie omówimy w niniejszym podrozdziale.

Każde wyrażenie składa się z argumentu i operatora (lub operatorów).

Dostępne operatory:

- `*`

  Stosowany do argumentów typu `String` pozwala budować warunki odpowiadające metodom SODA API takim jak `startsWith()`, `endsWith()`, `contains()`.

  Przykładowe wyrażenia:

  - *abc - pole kończy się na 'abc'
  - abc* - pole zaczyna się na 'abc'
  - *abc* - pole zawiera podsłowo 'abc'

- `>, <, <=, >=`

  Operatory pozwalają definiować warunki odpowiadające metodom SODA API takim jak `greater()`, `smaller()`. Mogą być stosowane zarówno do pól o typach liczbowych jak i typu `String`. W drugim przypadku napisy będą porównywane według porządku leksykograficznego.

Przykładowe wyrażenia:

- >=50' - pole ma wartość większą lub równą 50. Warunek ten jest poprawne tylko dla pól o typach liczbowych.
- >"50" - pole ma wartość większą niż "50" według porządku leksykograficznego. Warunek ten jest poprawny tylko dla pól typu `String`.

- ==

Operator porównania równościowego. Pozwala definiować warunki określające dokładną wartość pola. Operator opcjonalny. Przykładowo wyrażenie '==50' wyprodukuje ten samo kod zapytania co wyrażenie '50'. Podobnie w przypadku argumentów typu `String`. Wyrażenie '=="Kowalski"' jest równoważne wyrażeniu '"Kowalski"'.

Przykładowe wyrażenia:

- "Kowalski" - pole ma wartość "Kowalski"
- 50 - pole ma wartość 50

Jeżeli użycie powyższych operatorów wydaje się być niejasne zachęcamy do zapoznania się z treścią kolejnego podrozdziału. W nim przedstawimy przypadki użycia , które ukazują jak w praktyce definiuje się ograniczenia za pomocą edytora Squared.

4.4 Przypadki użycia

Niniejszy rozdział pokazuje jak wygląda praca z edytorem Squared w praktyce. Rozpatrzymy w nim szereg przykładów, które zademonstrują możliwości oprogramowania powstałego w wyniku niniejszej pracy.

Przykład 1

"Wszystkie obiekty klasy Osoba"

Przykład najprostszy z możliwych. Jeżeli chcemy uzyskać listę wszystkich obiektów typu `Osoba` wystarczy, że dwukrotnie klikniemy na klasę `Osoba` w widoku obiektów edytora (Object View). W widoku edycji zapytania pojawi się wówczas węzeł grafu reprezentujący klasę `Osoba`.

Ilustracja 10: Węzeł Osoba

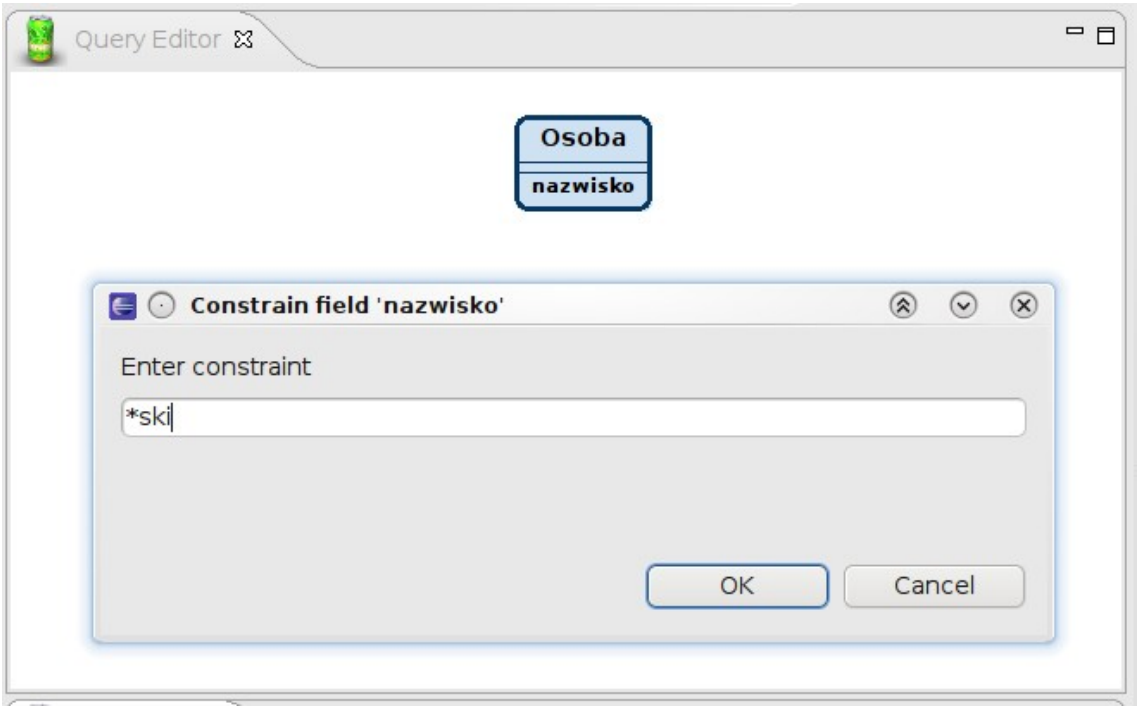
Jednocześnie w widoku kodu zapytania (Query View) pojawi się poniższy kod zapytania:

```
1  Query query = db.query();
2  query.constrain(Osoba.class);
3  ObjectSet result = query.execute();
```

Przykład 2

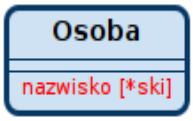
"Osoby o nazwisku kończącym się na 'ski' "

W drugim przykładzie interesują nas tylko osoby o nazwisku kończącym się sylabą 'ski'. Procedura postępowania w tym przypadku rozpoczyna się identycznie jak w pierwszym przykładzie. Gdy w widoku edycji zapytania pojawi się węzeł reprezentujący klasę `Osoba` możemy dodać ograniczenie. Aby to zrobić klikamy na pole nazwisko, co powoduje pojawienie się okna dialogowego. W pole tekstowe wprowadzamy wyrażenie '*ski' tak jak na poniższej ilustracji.



Ilustracja 11: Wprowadzanie ograniczenia na pole nazwisko

Zatwierdzamy przyciskiem OK (lub klawiszem Enter). W ten sposób dodaliśmy ograniczenie do węzła grafu. Na poniższej ilustracji widzimy jak zmienił się wygląd węzła grafu po dodaniu ograniczenia. Ograniczone pole jest teraz wyróżnione czerwonym kolorem a wyrażenie użyte do jego ograniczenia jest wyświetlone obok nazwy pola w nawiasach kwadratowych.



Ilustracja 12: Węzeł Osoba

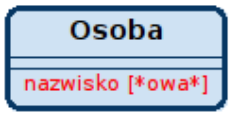
Poniżej widzimy kod zapytania wygenerowanego po przeprowadzeniu opisanej procedury.

```
1  Query query = db.query();
2  query.constrain(Osoba.class);
3  Constraint constrNazwisko =
query.descend("nazwisko").constrain("ski").endsWith(true);
4  ObjectSet result = query.execute();
```

Przykład 3

"Osoby, które mają w nazwisku podsłowo 'owa'"

Trzeci przykład demonstruje jak szukać obiektów z polami typu `String`, które zawierają dane podsłowo. Przykład jest analogiczny do poprzedniego. Jedyną różnicą jest inne wyrażenie ograniczające.



Ilustracja 13: Węzeł Osoba

Wygenerowany kod zapytania:

```
1  Query query = db.query();
2  query.constrain(Osoba.class);
3  Constraint constrNazwisko =
query.descend("nazwisko").constrain("owa").contains();
5  ObjectSet result = query.execute();
```

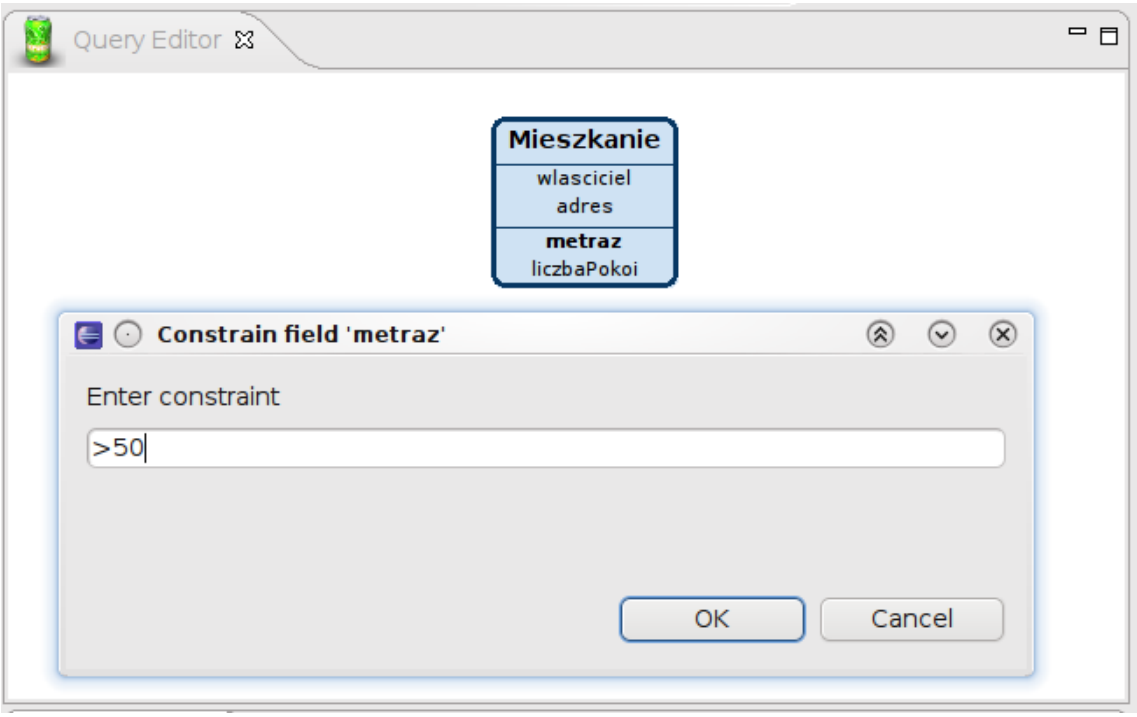
W kolejnych przykładach będziemy korzystać z klas `Mieszkanie`, `Adres` oraz `Miejscowosc` zdefiniowanych poniżej.

```
class Miejscowosc {
      String nazwa;
      int liczbaMieszkancow;
}

class Adres {
      Miejscowosc miejscowosc;
      String ulica;
}

class Mieszkanie {
      Ososba wlasciciel;
      Adres adres;
      double metraz;
      int liczbaPokoi;
}
```

Przykład 4

"Mieszkania o powierzchni powyżej 50 metrów kwadratowych"

W kolejnym przykładzie będziemy nakładać ograniczenie na pole typu liczbowego. Z punktu widzenia użytkownika edytora nie ma to większego znaczenia. Użytkownik klika nazwę pola `metraz` i wprowadza wyrażenie ograniczające to pole. W tym przypadku wyrażenie to ma postać '>50' jak widzimy na poniższej ilustracji.



Ilustracja 14: Wprowadzanie ograniczenia na pole metraz

Po wprowadzeniu ograniczenia obserwujemy zmieniony wygląd węzła z nałożonym ograniczeniem.

Ilustracja 15: Węzeł z ograniczeniem na polu metraz

Jednocześnie w widoku kodu zapytania (Query View) widzimy wygenerowany kod zapytania.

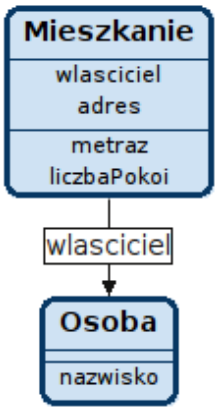
```
1  Query query = db.query();
```

```
2  query.constrain(Mieszkanie.class);
3  Constraint constrMetraz =
query.descend("metraz").constrain(50).greater();
4  ObjectSet result = query.execute();
```

Przykład 5

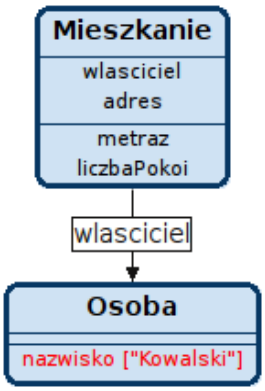
"Mieszkania, których właścicielem jest Kowalski"

Poprzednie przykłady miały na celu zademonstrowanie różnych rodzajów ograniczeń. Niniejszy przykład wprowadza pewną nowość. Będziemy w nim ograniczać pole rozszerzalne. Jak już wiemy takiego pola nie możemy ograniczyć bezpośrednio. Możemy je natomiast rozwinąć. W tym celu klikamy jego nazwę, w tym przypadku pole ma nazwę `wlasciciel`, i obserwujemy nowy węzeł w widoku edycji.



Ilustracja 16: Węzeł Mieszkanie z rozwiniętym polem właściciel

Rozwinąwszy pole `wlasciciel` w podwęzeł mamy bezpośredni dostęp do pól klasy `Osoba`. Teraz możemy bez przeszkód dostać się do interesującego nas pola `nazwisko`. Klikamy je i w znany nam już sposób wprowadzamy ograniczenie - "Kowalski".



Ilustracja 17: Węzeł Osoba z ograniczonym polem nazwisko

Po wprowadzeniu ograniczenia zwróćmy uwagę na wygenerowany kod zapytania. Szczególnie warto zwrócić uwagę na linię trzecią. Widzimy w niej dodatkowe wywołanie metody `descend()`, które edytor dodał po rozwinięciu pola `wlasciciel`.

```
1  Query query = db.query();
2  query.constrain(Mieszkanie.class);
3  Constraint constrNazwiskoWlasciciel =
query.descend("wlasciciel").descend("nazwisko").constrain("Kowal
ski");
4  ObjectSet result = query.execute();
```

Przykład 6

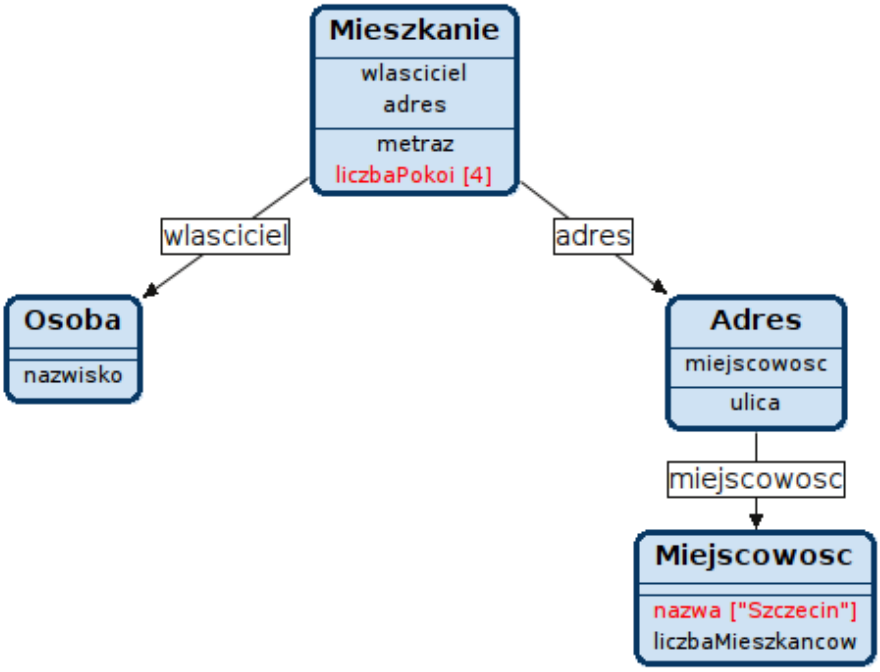
"Czteropokojowe mieszkania w Szczecinie"

W tym przykładzie szukamy mieszkań spełniających dwa warunki. Liczba pokoi musi być równa 4 i mieszkanie musi mieścić się w Szczecinie. Zaczynamy tworzyć diagram od węzła reprezentującego klasę `Mieszkanie`. Widzimy, że ta klasa posiada pole `liczbaPokoi`. Zatem nakładamy na nie ograniczenie. W tym momencie wygenerowany kod zapytania odzwierciedla wprowadzone ograniczenie.

```
1  Query query = db.query();
2  query.constrain(Mieszkanie.class);
3  Constraint constrLiczbaPokoi =
query.descend("liczbaPokoi").constrain(4);
4  ObjectSet result = query.execute();
```

Powyższy kod zwróci listę mieszkań, które mają cztery pokoje. Nas jednak interesują tylko czteropokojowe mieszkania w Szczecinie. Zatem kontynuujemy pracę z edytorem. Rozwijamy pole `adres`. W wyniku tej czynności pojawia się podwęzeł klasy `Adres`. W nim klikamy na pole `miejscowosc` rozwijając podwęzeł reprezentujący klasę `Miejscowosc`. Ma on pole `nazwa`, które zawiera nazwę miejscowości. Klikamy na nią aby nałożyć ograniczenie. W oknie dialogowym wprowadzamy wyrażenie ograniczające i zatwierdzamy przyciskiem OK.

Na poniższej ilustracji przedstawiono diagram powyższego zapytania. Widzimy na nim również rozwinięte pole `wlasciciel`. Jest ono zbyteczne. Nie zdefiniowaliśmy dla niego żadnego ograniczenia i dlatego nie zmienia ono wygenerowanego kodu.



Ilustracja 18: Rozbudowany graf zapytania z dwoma ograniczeniami

Pełny kod powyższego zapytania wygenerowany przez Squared:

```
1  Query query = db.query();
2  query.constrain(Mieszkanie.class);
3  Constraint constrLiczbaPokoi =
query.descend("liczbaPokoi").constrain(4);
4  Constraint constrNazwaMiejscowoscAdres =
query.descend("adres").descend("miejscowosc").descend("nazwa").c
onstrain("Szczecin");
5  ObjectSet result = query.execute();
```

5 Użyte narzędzia i biblioteki

Celem niniejszej pracy było stworzenie narzędzia zapewniającego łatwy i intuicyjny dostęp do bazy danych db4o nawet użytkownikom niezaznajomionym z tym

OSZBD. W niniejszym rozdziale przedstawimy narzędzia użyte do zrealizowania tego celu.

5.1 Platforma Eclipse

Eclipse to projekt rozpoczęty w listopadzie 2001 roku przez firmę IBM. Od roku 2004 nad jego rozwojem czuwa Eclipse Foundation - organizacja typu non-profit założona przez IBM. Eclipse jest napisanym w języku Java frameworkiem służącym do tworzenia aplikacji. Jego najważniejszą cechą jest rozszerzalność. Platforma Eclipse sama w sobie posiada pewne wbudowane funkcjonalności jednak są one bardzo ogólne. Filozofia tworzenia aplikacji w oparciu o platformę Eclipse polega na tworzeniu wtyczek (ang. plug-in), które rozszerzają funkcjonalności platformy. Dzięki temu z jednej ogólnej bazy mogą powstać tak różne, wyspecjalizowane aplikacje jak edytor diagramów UML, klient serwisu Twitter czy IDE do tworzenia aplikacji dla systemu operacyjnego Android.

Najbardziej znaną, flagową wtyczką do platformy Eclipse jest Java Development Tools (JDT)[7]. Jest to zestaw rozszerzeń platformy, który czyni zeń zintegrowane środowisko programistyczne dla języka Java. Platforma Eclipse wraz z JDT i innymi wtyczkami jest dystrybuowana pod nazwą Eclipse IDE for Java Developers. Dzisiaj ta dystrybucja jest często utożsamiana z platformą chociaż w rzeczywistości jest to tylko zestaw dodatków dystrybuowany w jednym pakiecie.

Elastyczność Eclipse oraz prostota rozszerzania uczyniły z niego popularną platformę do tworzenia aplikacji typu Integrated Development Environment (IDE). W chwili pisania tych słów istnieją rozszerzenia platformy wspierające proces tworzenia oprogramowania w językach takich jak C, C++, Python, Ruby, PHP, Perl, Scala, Ada, COBOL. Warto wspomnieć, że firmy takie jak Google czy Samsung stworzyły IDE do tworzenia aplikacji dla swoich mobilnych systemów operacyjnych używając platformy Eclipse. W przypadku Google jest to wtyczka Android Development Tools (ADT)[8] przeznaczona do rozwijania oprogramowania dla systemu Android. Natomiast Samsung

7 Java Development Tools (JDT) - http://www.eclipse.org/jdt/
8 Android Development Tools (ADT) - http://developer.android.com/sdk/eclipse-adt.html

udostępnia bada IDE[9] jako zalecane środowisko do tworzenia aplikacji dla urządzeń działających pod kontrolą systemu operacyjnego Samsung bada.

Wyżej wymienione zalety platformy Eclipse skłoniły autora do stworzenia edytora zapytań będącego przedmiotem niniejszej pracy z użyciem opisanej platformy.

5.2 Graphical Editing Framework

GEF[10] jest frameworkiem o otwartych źródłach przeznaczonym do tworzenia aplikacji zorientowanych na graficzne edytowanie. Pozwala on wizualizować niemal dowolny model danych oraz ułatwia oprogramowanie interakcji użytkownika z wizualną reprezentacją modelu.

Graphical Editing Framework składa się z dwóch wtyczek.

- org.eclipse.draw2d

  Używany do wyświetlania obiektów i rozmieszczania ich na ekranie. Nie posiada sam w sobie funkcjonalności związanych z edytowaniem.

- org.eclipse.gef

  Dodaje do funkcjonalności Draw2D możliwość interakcji z wyświetlanymi obiektami. Zapewnia wsparcie dla zaawansowanych funkcjonalności takich jak drag and drop czy zaznaczanie grup obiektów.

GEF jest bardzo wszechstronnym frameworkiem. Z jego użyciem powstały już aplikacje takie jak:

- Edytor Diagramów UML
- Projektant Interfejsów Użytkownika
- Edytor Diagramów Blokowych

GEF daje programiście narzędzia znacznie ułatwiające tworzenie aplikacji,

9 Samsung bada SDK - http://developer.bada.com/apis/index.do
10 Graphical Editing Framework (GEF) - http://www.eclipse.org/gef/

których istotną częścią jest graficzna prezentacja danych. Biblioteka ta zawiera zestaw klas implementujących graficzne obiekty takie jak figury, połączenia między nimi, etykiety oraz mechanizmy obsługi interakcji użytkownika z tymi obiektami. Te udogodnienia bardzo przyspieszają proces tworzenia oprogramowania zorientowanego na wizualną prezentację danych.

Z powyższego opisu wyłania się obraz biblioteki idealnej do zastosowania w projekcie takim jak graficzny edytor zapytań Squared. I tak jest w istocie. Dzięki bibliotece GEF praca nad graficznym edytorem jest zajęciem, które nie wymaga pisania dodatkowych tysięcy linii kodu aby móc wyświetlić złożone obiekty graficzne - te funkcjonalności daje nam Graphical Editing Framework.

ROZDZIAŁ III

Rezultaty Pracy

1 Stan oprogramowania

Oprogramowanie powstałe w ramach niniejszej pracy jest w chwili obecnej w stanie pozwalającym na bezproblemową pracę. Jakkolwiek nie wszystkie planowane funkcjonalności udało się zaimplementować. Dokładne omówienie stanu oprogramowania znajduje się w podrozdziałach 2 Osiągnięte cele i 3 Nierozwiązane problemy. W trakcie pracy nad kodem edytora Squared autorowi nasunęło się wiele pomysłów na rozwinięcie planowanych od początku funkcjonalności, jak i na dodanie zupełnie nowych. Te pomysły przedstawione są w podrozdziale 4 Propozycje rozwinięcia oprogramowania.

Kod edytora w stanie dołączanym do niniejszej pracy jest raczej zaawansowanym prototypem niż ukończonym narzędziem. Nie ulega wątpliwości, że Squared wymaga jeszcze wiele pracy do osiągnięcia dojrzałości. Pisanie edytora było dla autora interesującym zajęciem oraz jednocześnie praktyczną nauką tworzenia wtyczek dla platformy Eclipse jak i programowania z użyciem Graphical Editing Framework.

2 Osiągnięte cele

Założonym celem pracy było stworzenie narzędzia pozwalającego na wygodny i intuicyjny dostęp do bazy danych db4o. Zdaniem autora ten podstawowy cel udało się osiągnąć. Edytor Squared umożliwia wizualne tworzenie grafów zapytań i generuje na tej podstawie poprawny kod realizujący to zapytanie.

Wymieńmy szczegółowe cele, które udało się osiągnąć:

- odczytywanie dowolnego pliku bazy danych db4o oraz wyświetlanie przechowywanych w nim obiektów
- tworzenie i edytowanie grafu zapytania poprzez rozwijanie węzłów grafu
- dodawanie ograniczeń do węzłów grafu zapytania
- generowanie kodu zapytania na podstawie grafu zapytania
- zapisywanie grafu zapytania w postaci pliku graficznego

3 Nierozwiązane problemy

Pomimo osiągnięcia głównych celów pracy, pozostały pewne zamierzenia, które nie doczekały się realizacji.

Najważniejszym z nich jest możliwość logicznego łączenia ograniczeń. Wizualnie ta funkcjonalność pozwalałaby użytkownikowi utworzyć krawędź łączącą dwa ograniczenia. Krawędź ta reprezentowałaby relację logiczną pomiędzy ograniczeniami. Użytkownik mógłby wybrać typ relacji lub wprowadzić wyrażenie, które byłoby tłumaczone na odpowiednie wywołania SODA API. Koncepcję tej funkcjonalności szerzej przedstawiono w rozdziale drugim, podrozdział 2 Graf zapytania.

Drugim problemem, który nie doczekał się rozwiązania jest brak wsparcia dla zapytań zawierających ograniczenia pól typu `String`, w których nieważna jest wielkość znaku. W chwili obecnej Squared w takich przypadkach zawsze generuje zapytanie rozróżniające napisy różniące się wielkością znaku. Użytkownik jest niestety zmuszony ręcznie modyfikować kod wygenerowany przez edytor jeżeli potrzebuje zapytania, w którym wielkość znaku nie ma znaczenia. Na szczęście w większości przypadków modyfikacja jest trywialna. Jeżeli wygenerowane przez Squared zapytanie zawiera wywołanie `startsWith(true)` lub `endsWith(true)` wystarczy zmienić wartość argumentu na **`false`**. Ponadto zamiast metody `contains()` należy stosować `like()`, która działa identycznie z wyjątkiem odmiennego traktowania

różnic w wielkości znaku - `contains()` rozróżnia znaki o różnej wielkości, podczas gdy `like()` te różnice ignoruje.

4 Propozycje rozwinięcia oprogramowania

Podczas prac implementacyjnych pojawiło się wiele pomysłów, których realizacja może być ciekawym rozwinięciem obecnego stanu edytora Squared.

Najbardziej interesującym z nich wydaje się być dodanie do edytora funkcjonalności wizualizowania kodu zapytania w postaci grafu. Obecnie za pomocą Squared tworzymy graf i na jego podstawie generuje on kod zapytania. Rozpatrywana funkcjonalność miałaby działać przeciwnie. Użytkownik wprowadzałby kod zapytania a oprogramowanie analizowałoby kod i budowało graf zapytania. W ten sposób edytor mógłby być pomocny podczas szukania błędów w zapytaniach napisanych ręcznie przez użytkownika. Ta funkcjonalność po wstępnej analizie wydaje się być skomplikowanym przedsięwzięciem godnym oddzielnej pracy. Głównym problemem tutaj jest analiza kodu zapytania, który może być przecież zapisany na wiele równoważnych sposobów.

Rozpatrzmy zapytanie znajdujące mieszkania, które mają trzy pokoje lub przynajmniej pięćdziesiąt pięć metrów kwadratowych powierzchni.

```
1  Query query = db.query();
2  query.constrain(Mieszkanie.class);
3  Constraint constrMetraz =
query.descend("metraz").constrain(55).greater();
4  Constraint contrLiczbaPokoi =
query.descend("liczbaPokoi").constrain(3);
5  constrMetraz.or(contrLiczbaPokoi);
6  ObjectSet result = query.execute();
```

To samo zapytanie zapisane nieco inaczej.

```
1  Query query = db.query();
```

```
2  query.constrain(Mieszkanie.class);
3  Constraint constrMetraz =
query.descend("metraz").constrain(55).greater();
4  Constraint contrLiczbaPokoi =
query.descend("liczbaPokoi").constrain(3).or(constrMetraz);
5  ObjectSet result = query.execute();
```

Jeszcze jedna wersja tego samego zapytania.

```
1  Query query = db.query();
2  query.constrain(Mieszkanie.class);
3  Constraint constrMetraz =
query.descend("metraz").constrain(55).greater().or(query.descend
("liczbaPokoi").constrain(3));
4  ObjectSet result = query.execute();
```

Powyższe trzy urywki kodu realizują to samo zapytanie. Pokazuje to jak rozbudowanym zadaniem jest stworzenie narzędzia zdolnego do wizualizacji grafu zapytania na podstawie kodu.

W miarę postępu prac programistycznych nad projektem autorowi nasuwały się kolejne funkcjonalności, których dodanie powiększyłoby wartość Squared. Wymieniamy je tutaj wraz z krótkim opisem:

1. Możliwość wykonania wygenerowanego zapytania na otwartej bazie danych. Użytkownik mógłby sprawdzić zapytanie w czasie rzeczywistym.
2. Możliwość jednoczesnej pracy nad wieloma zapytaniami. Użytkownik mógłby tworzyć kolejne zapytania i pracować nad nimi równocześnie. Każde zapytanie byłoby otwarte w oddzielnej zakładce edytora.
3. Wsparcie dla operacji Cofnij, Ponów.
4. Możliwość przybliżania i oddalania widoku edytora. Dzięki temu praca z dużymi grafami byłaby dużo łatwiejsza. Szczególnie dla użytkowników pracujących na małych monitorach.

5. Kolorowanie składni w widoku kodu zapytania. Edytor jest także narzędziem dydaktycznym i może służyć do zapoznania się z SODA API. Dzięki kolorowaniu składni kodu generowanych zapytań ta nauka z pewnością byłaby łatwiejsza.
6. Proponowane udogodnienia interfejsu użytkownika:
   - przycisk pozwalający wyczyścić aktualny diagram
   - przycisk pozwalający wyczyścić aktualny diagram pozostawiając jednak jego węzeł wejściowy
   - przycisk kopiujący wygenerowany kod zapytania do schowka - obecnie ta funkcjonalność istnieje, ale wymaga od użytkownika zaznaczenia tekstu zapytania i wciśnięcia kombinacji klawiszy CTRL + C. Wydaje się, że pojedyncza czynność naciśnięcia przycisku zwiększyłaby komfort pracy i produktywność użytkownika.
   - animacje podczas rozwijania węzłów - obecnie po kliknięciu na rozwijalne pole węzła w jednej chwili pojawia się nowy węzeł. Zdaniem autora dobrze przemyślana animacja będzie lepiej oddawała to, co właśnie stało się na ekranie. To może przełożyć się na lepsze zrozumienie idei edytora przez nowych użytkowników.
   - rozróżnienie pól rozwijalnych i ograniczalnych poprzez oznaczenie ich ikonami czytelnie opisującymi ich rolę.
   - możliwość zmiany domyślnej nazwy kontenera danych używanej w kodzie generowanego zapytania. Wygenerowany kod zakłada, że istnieje obiekt o nazwie `db` typu `ObjectContainer`. Użytkownik mógłby zdefiniować dowolną, własną nazwę kontenera. Dzięki temu kod wygenerowany przez edytor nadawałby się do wklejenia do programu użytkownika bez żadnych ręcznych modyfikacji.

W trakcie pracy nad oprogramowaniem autor postanowił w pierwszej kolejności skupić się na podstawowych celach. Te niestety okazały się wymagać dużego nakładu

pracy i uniemożliwiły zaimplementowanie dodatkowych funkcjonalności przedstawionych w niniejszym rozdziale.

BIBLIOGRAFIA

1 Edlich S., Paterson J. i inni: *The definitive guide to db4o* (2006)

2 Moore B., Dean D. i inni: *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.* IBM Redbooks (2004)

3 Neward T.: *The Vietnam of Computer Science* (2006), http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx

4 *Eclipse Platform Technical Overview,* http://eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf

5 Strona internetowa db4objects, http://www.db4o.com

6 Strona internetowa http://www.odbms.org

7 Strona internetowa projektu Hibernate, http://www.hibernate.org

8 Strona internetowa projektu Graphical Editing Framework, http://www.eclipse.org/gef/

ZAŁĄCZNIK A

Zawartość dołączonej płyty DVD

Do pracy dołączony jest pojedynczy nośnik DVD zawierający pliki:

1. praca_magisterska/praca_magisterska.pdf

   Niniejsza praca magisterska w formacie PDF

2. squared/squared_1.0.0.20101010.jar

   Archiwum jar zawierające edytor zapytań Squared w postaci gotowej do zainstalowania wtyczki do Eclipse

3. squared/squared.source_1.0.0.20101010.zip

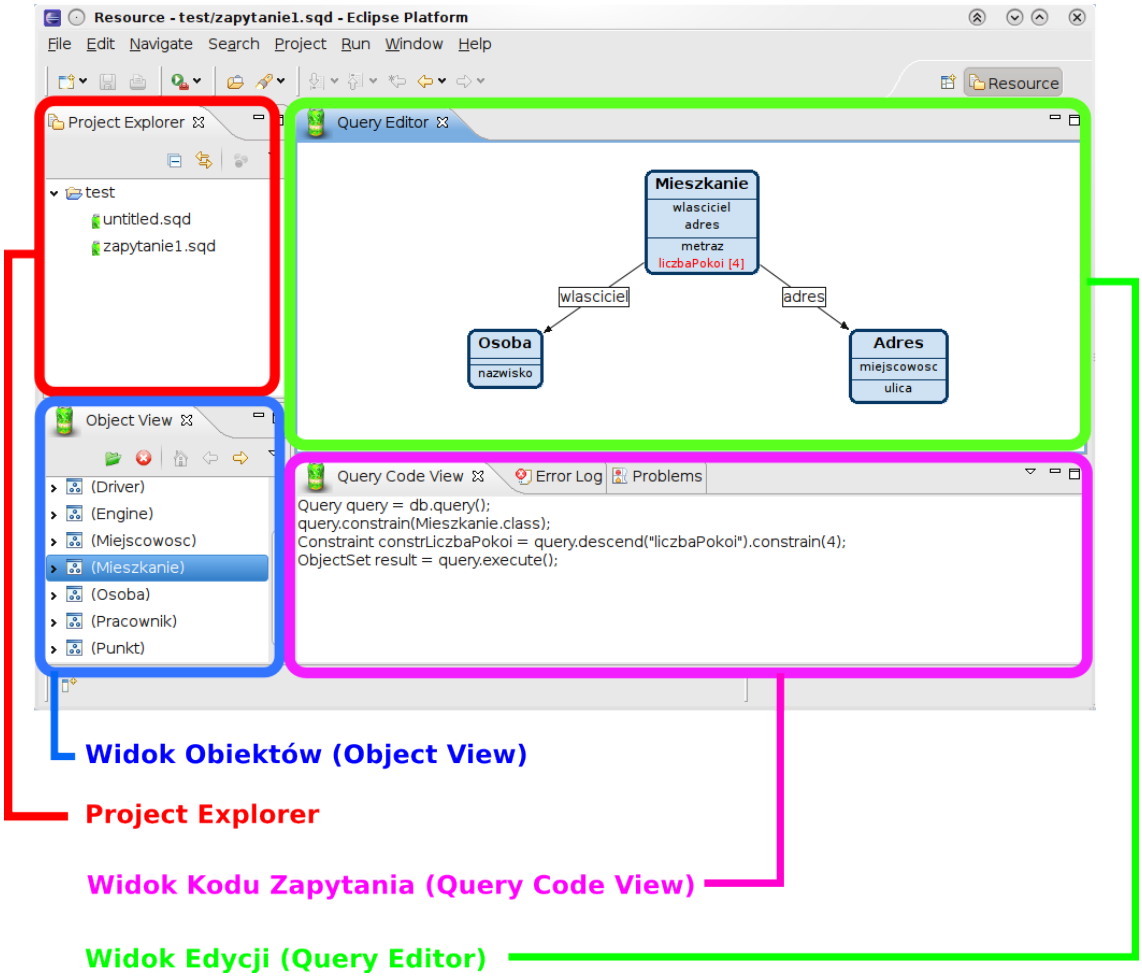
   Kod źródłowy edytora zapytań Squared

ZAŁĄCZNIK B

Podręcznik użytkownika

1 Opis interfejsu użytkownika

Interfejs edytora Squared składa się z czterech widoków przedstawionych na poniższym zrzucie ekranu.



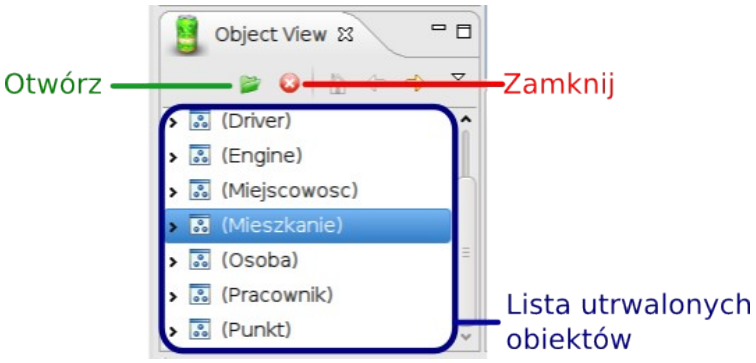
Ilustracja 19: Widoki edytora Squared

Oczywiście użytkownik może zmieniać położenie i rozmiar poszczególnych

widoków według własnego uznania. Przedstawiony wyżej układ jest tylko przykładem.

Przeznaczenie poszczególnych widoków zostało dokładnie opisane w rozdziale 4.1 Interfejs edytora. Niniejszy rozdział ma za zadanie opisać poszczególne elementy interfejsu użytkownika.

Widok obiektów (Object View)



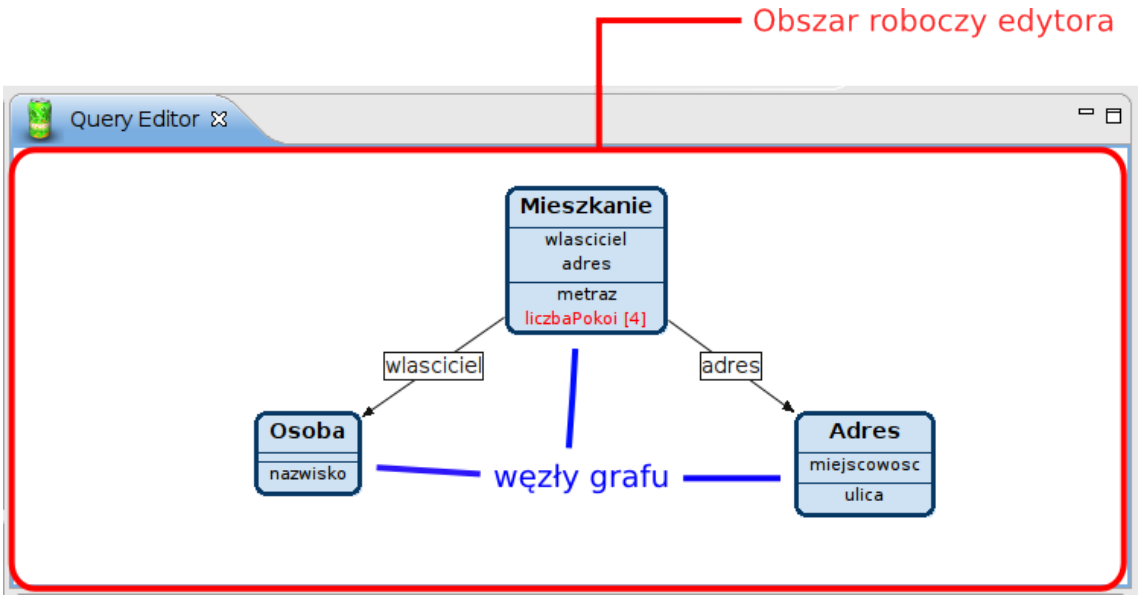
Ilustracja 20: Widok obiektów (Object View)

Większą część widoku obiektów zajmuje lista obiektów utrwalonych we wczytanym pliku bazy danych db4o (oznaczona niebieskim kolorem). Widzimy też dwa przyciski.

Przycisk Otwórz (oznaczony zielonym kolorem) pozwala otworzyć plik bazy danych db4o. Po otworzeniu bazy danych lista utrwalonych obiektów wypełnia się nowo-wczytanymi danymi.

Przycisk Zamknij (oznaczony kolorem czerwonym) pozwala zamknąć aktualnie otwarty plik bazy danych. W większości przypadków użytkownik nie musi używać tego przycisku. Zwykle po zakończeniu pracy z edytorem wystarczy zamknąć Eclipse a plik bazy danych zostanie zamknięty automatycznie. Również w przypadku próby otworzenia nowego pliku bazy danych, automatycznie zostanie zamknięty poprzedni. Gdyby jednak z jakiegoś powodu użytkownik potrzebował zamknąć używany aktualnie plik, może to uczynić za pomocą przycisku Zamknij.

Widok edycji grafu zapytania (Query Editor)

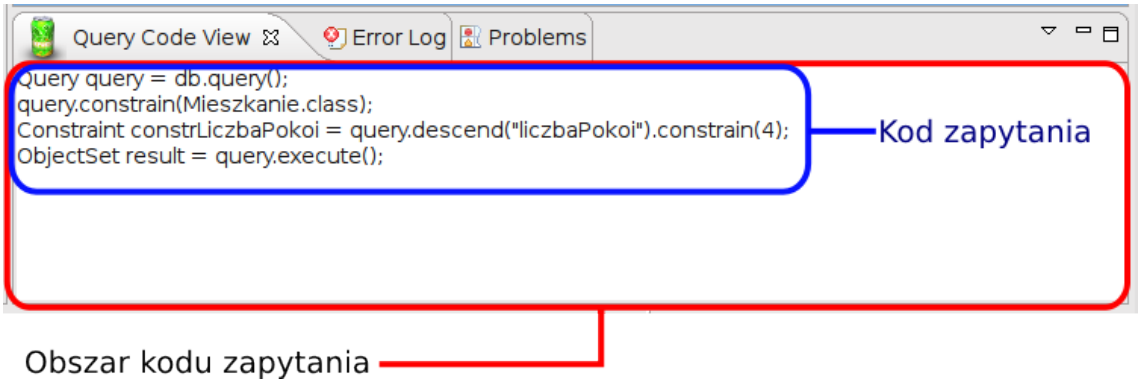


Ilustracja 21: Widok edycji zapytania (Query Editor)

Widok edycji zapytania wypełnia obszar roboczy edytora. Jest to płaszczyzna, na której mieści się graf zapytania tworzony za pomocą edytora Squared. Na tej płaszczyźnie znajdują się węzły grafu oraz połączenia między nimi.

Gdy Query Editor jest aktywnym widokiem platformy Eclipse możemy zapisać graf zapytania jako plik graficzny poprzez opcję menu File -> Save As.

Widok kodu zapytania (Query Code View)



Ilustracja 22: Widok kodu zapytania (Query Code View)

Widok kodu zapytania jest bardzo prostym elementem interfejsu. Jego jedynym zadaniem jest prezentacja wygenerowanego przez Squared kodu zapytania. Użytkownik może zaznaczyć kod zapytania za pomocą myszy i skopiować tekst zapytania do schowka. Dzięki tej funkcjonalności użytkownik może łatwo wykorzystać Squared do szybkiego tworzenia zapytań na potrzeby tworzonych przez niego aplikacji korzystających z bazy danych db4o.

2 Instalacja wtyczki Squared

Do zainstalowania edytora wymagana jest wcześniejsza instalacja platformy Eclipse. Squared był pisany jako wtyczka dla Eclipse w wersji 3.5.2 (Galileo). Był również testowany z wersją 3.6.1 (Helios) jednak powinien z powodzeniem działać również w starszych wersjach platformy. Jeżeli użytkownik nie posiada kompatybilnej wersji platformy musi ją najpierw zainstalować. W tym celu należy ściągnąć dowolną dystrybucję Eclipse dostępną na stronie http://www.eclipse.org/downloads/.

Do poprawnego działania Squared wymagana jest wtyczka Graphical Editing Framework, która jest dołączana do większości dystrybucji platformy Eclipse. Jeżeli jednak GEF nie jest dołączony do Twojej instalacji Eclipse należy go zainstalować zgodnie z instrukcją dostępną na stronie internetowej projektu GEF http://wiki.eclipse.org/index.php/GEF_Developer_FAQ#Download_and_Install.

Instalacja edytora sprowadza się do skopiowania pliku postaci squared_VER.YYYYMMDD.jar do katalogu plugins w swojej instalacji platformy Eclipse. W podanej wyżej nazwie pliku VER oznacza numer wersji wtyczki. Natomiast YYYYMMDD oznacza datę jej zbudowania. Na płycie DVD dołączonej do niniejszej pracy znajdziemy Squared w wersji 1.0.0 zbudowany dziesiątego października 2010 roku - squared_1.0.0.20101010.jar. Ten plik wystarczy skopiować do katalogu plugins w miejscu instalacji platformy Eclipse. Aby móc używać zainstalowanej wtyczki wymagany jest restart Eclipse. Po restarcie możemy potwierdzić poprawną instalacje wtyczki przechodząc do menu Help -> About Eclipse, następnie klikając Installation Details. W zakładce Plug-ins na liście zainstalowanych wtyczek powinna figurować

wtyczka o nazwie Squared.

3 Praca z edytorem Squared

Poniższy rozdział wprowadza użytkownika w zagadnienia związane z używaniem edytora Squared.

3.1 Przygotowanie do pracy i tworzenie projektu

Praca ze Squared polega na używaniu widoków tworzących interfejs użytkownika edytora zapytań. Aby je zobaczyć wybieramy opcję Window -> Show View -> Other. Z widocznej listy widoków wybieramy db4o -> Object View. Tę czynność powtarzamy dla widoku Query View.

Aby rozpocząć pracę z wtyczką należy najpierw utworzyć projekt. W tym celu uruchamiamy Eclipse oraz wybieramy opcję File -> New -> Project. Z listy dostępnych predefiniowanych szablonów wybieramy General -> Project, podajemy nazwę projektu i klikamy przycisk Finish.

Gdy mamy przygotowany projekt możemy dodać plik, na którym będziemy operować. Wybieramy opcję File -> New -> Other. Z listy dostępnych pozycji wybieramy db4o -> Query Diagram i zatwierdzamy przyciskiem Next. Następnie wprowadzamy nazwę pliku lub zostawiamy domyślną - untitled.sqd. Zatwierdzamy przyciskiem Finish. Po zatwierdzeniu w widoku Package Explorer powinien pojawić się utworzony plik. Poprzez dwukrotne kliknięcie jego nazwy możemy otworzyć widok edycji grafu zapytania (Query Editor).

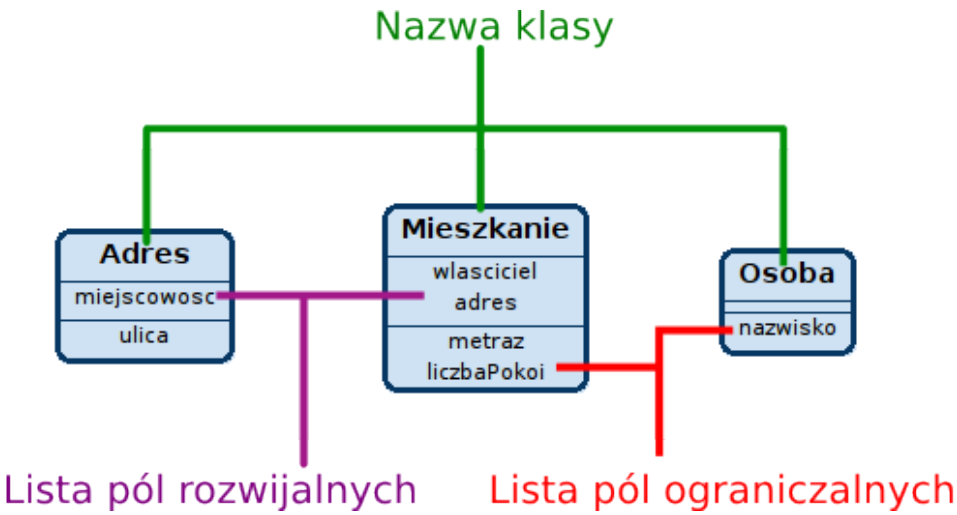
Aby móc rozpocząć faktyczną prace z edytorem Squared potrzebujemy dostępu do dowolnego pliku bazy danych db4o. Możemy go uzyskać klikając przycisk Otwórz w widoku obiektów (Object View). Po wczytaniu pliku bazy danych widok Object View wypełnia się listą klas obiektów utrwalonych we wczytanym pliku.

3.2 Tworzenie grafu zapytania

Aby rozpocząć edycję grafu zapytania klikamy dwukrotnie dowolną klasę z listy widocznej w widoku obiektów (Object View). Po wykonaniu tej czynności w widoku edycji grafu zapytania widzimy węzeł grafu, czyli graficzną reprezentację klasy utrwalonej w bazie danych. Poniżej widzimy widok edycji grafu zapytania (Query Editor) po dwukrotnym kliknięciu klasy `Mieszkanie`.

W tej chwili węzeł `Mieszkanie` jest naszym punktem wejścia. Oznacza to, że zapytanie wygenerowane przez Squared będzie zwracało listę obiektów klasy `Mieszkanie`. Oczywiście w każdej chwili możemy zmienić punkt wejścia. Wystarczy w widoku obiektów (Object View) dwukrotnie kliknąć inną klasę. Edytor zapyta wówczas czy na pewno chcemy usunąć graf aktualnie widoczny w widoku edycji. Po potwierdzeniu rozpocznie się tworzenie nowego grafu, którego punktem wejścia będzie nowa klasa.

Na poniższym zrzucie ekranu widzimy węzły reprezentujące obiekty różnych klas.



Ilustracja 24: Węzły grafu zapytania

Jak widzimy każdy węzeł składa się z trzech segmentów.

1. Nazwa klasy

   Pierwszy segment zawiera nazwę klasy, którą dany węzeł reprezentuje - odpowiednio Adres, Mieszkanie, Osoba.

2. Lista pól rozwijalnych

   Drugi segment zawiera listę pól, które możemy rozwinąć poprzez kliknięcie. Wyjaśnienie czym są pola rozwijalne znajduje się w rozdziale 4.2 Praca z edytorem.

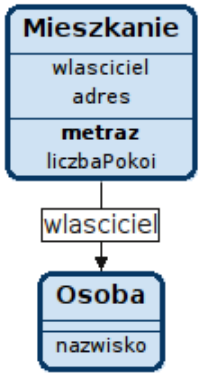
3. Lista pól ograniczalnych

   Trzeci segment zawiera listę pól, na które możemy nałożyć ograniczenia. Wyjaśnienie czym są pola ograniczalne znajduje się w rozdziale 4.2 Praca z edytorem.

3.3 Rozwijanie węzłów grafu

Kliknięcie na dowolne pole rozwijalne powoduje pojawienie się nowego węzła grafu zapytania. Dzięki temu mechanizmowi użytkownik może dostać się do pól głęboko zagnieżdżonych w strukturze obiektu wejściowego. Poniższy zrzut ekranu

przedstawia węzeł reprezentujący klasę `Mieszkanie` z rozwiniętym polem `wlasciciel`.



Ilustracja 25: Węzeł Mieszkanie z rozwiniętym polem wlasciciel

3.4 Definiowanie ograniczeń

Kliknięcie na dowolne pole ograniczalne powoduje pojawienie się okna dialogowego, w którym użytkownik może wprowadzić wyrażenie ograniczające to pole. Ten mechanizm jest kluczowy dla działania edytora Squared. Dzięki niemu użytkownik określa warunki, które muszą spełniać obiekty aby znaleźć się w zbiorze rezultatów tworzonego zapytania.

Dokładne omówienie zagadnienia tworzenia ograniczeń znajduje się w rozdziale 4.3 Definiowanie ograniczeń.

4 Jak odinstalować Squared

Aby odinstalować wtyczkę Squared wystarczy usunąć plik squared_VER.YYYYMMDD.jar z katalogu plugins w Twojej instalacji platformy Eclipse.