Politechnika Łódzka
Instytut Informatyki

INŻYNIERSKA PRACA DYPLOMOWA

MAPOWANIE OBIEKTOWO-RELACYJNE W JĘZYKU C++

Wydział: Fizyki Technicznej, Informatyki i Matematyki Stosowanej
Promotor: dr inż. Arkadiusz Tomczyk
Dyplomant: Marcin Maciaszczyk
Nr albumu: 165466
Kierunek: Informatyka
Specjalność: Inżynieria Oprogramowania i Analiza Danych

Łódź, 10 maja 2014r.

Instytut Informatyki
90-924 Łódź, ul. Wólczańska 215, budynek B9
tel. 042 631 27 97, 042 632 97 57, fax 042 630 34 14 email: office@ics.p.lodz.pl

Spis treści

Rozdział 1

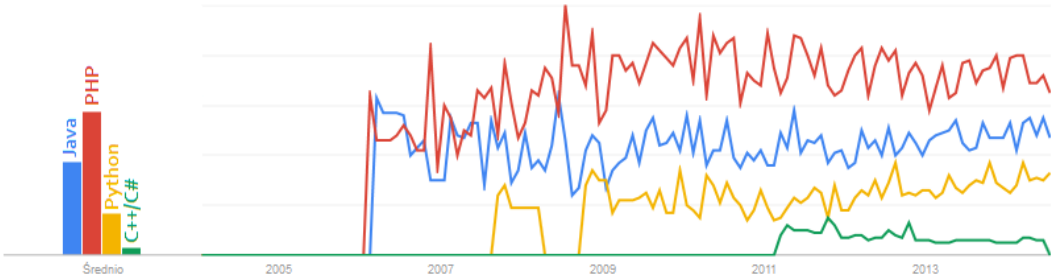
Wstęp

1.1 Uzasadnienie wyboru tematu

Wraz z rozwojem informatyki proces tworzenia oprogramowania wymaga od informatyków co raz większej wiedzy w poszczególnych dziedzinach takich jak na przykład bazy danych czy sieci komputerowe. Ciężko jest być specjalistą w każdej z nich, dlatego też podczas tworzenia oprogramowania programiści co raz częściej sięgają po różnego rodzaju narzędzia programistyczne, które mają za zadanie ułatwić im pracę wykonując jej część za nich.

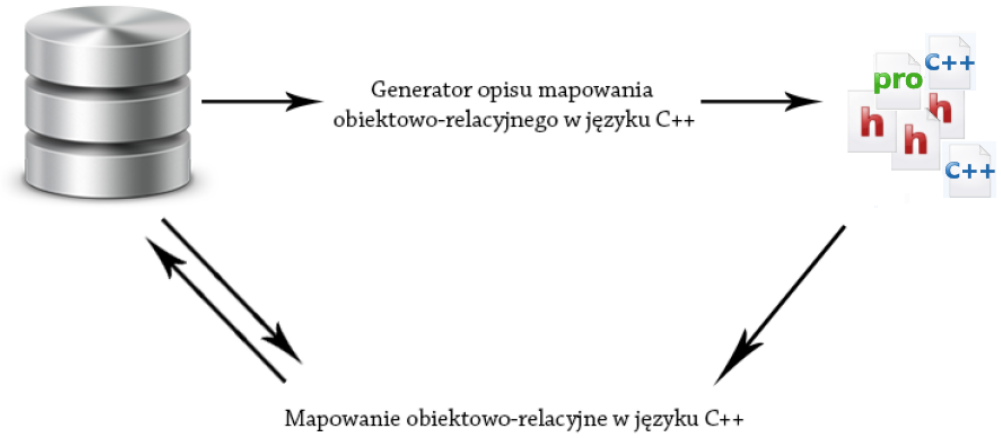
Przykładem takich narzędzi są aplikacje szkieletowe wykorzystywane jako fundamenty dla tworzonych aplikacji czy też biblioteki programistyczne udostępniające zestawy określonych funkcji. Oczywiście nie zawsze mamy możliwość skorzystania z wspominanych narzędzi, jednak jeśli taka istnieje warto wziąć to pod uwagę podczas fazy planowania tworzenia oprogramowania, ponieważ decydując się na korzystanie z nich jesteśmy w stanie zaoszczędzić sporo czasu, a także uniknąć wielu błędów związanych z nieznajomością danej dziedziny.

Wraz z kolegą z tego samego roku studiów – Sebastianem Florkiem, zdecydowaliśmy się w ramach pisamia pracy dyplomowej na wspólne stworzenie aplikacji szkieletowej, która będzie realizowała mapowanie obiektowo-relacyjne w języku C++. Wybór C++ jako języka programowania tłumaczymy dość dobrą jego znajomością, a także pewnym doświadczeniem w programowaniu w tym języku nabytym w trakcie studiów. Mapowanie obiektowo-relacyjne jest to obecnie zagadnienie co raz bardziej powszechne, szczególnie w językach programowania takich jak PHP czy Java. Programiści C++ nie mają już tak dużego wyboru wśród dostępnych narzędzi służących do mapowania obiektowo-relacyjnego, co także braliśmy pod uwagę ustalając temat nad jakim będziemy pracować.



Rys. 1.1: Wykres przedstawiający popularność mapowania obiektowo-relacyjnego na przestrzeni czasu w wybranych językach programowania [9]

Podział naszej pracy zostanie opisany na początku rozdziału 4, jednak na wstępie warto zaznaczyć, że tematem pracy Sebastiana jest generator opisu mapowania obiektowo-relacyjnego, który jest wstępnym etapem pracy naszej aplikacji szkieletowej. Celem wspomnianego generatora jest wygenerowanie pliku projektu, a także plików nagłówkowych oraz klas na podstawie istniejącej już bazy danych. Moim zadaniem będzie samo mapowanie obiektowo-relacyjne, które będzie się opierało na wcześniej wygenerowanych plikach.



Rys. 1.2: Schemat współpracy modułów tworzonej aplikacji szkieletowej, dalej występującej też pod nazwą Qubic

1.2 Problematyka i zakres pracy

Programowanie obiektowe jest obecnie jednym z najpopularniejszych paradygmatów programowania, a pojęcia takie jak klasa czy obiekt znane są wszystkim programistom. Podobnie jest z relacyjnym modelem organizacji baz danych i terminami takimi jak relacja czy krotka. Chcąc wykorzystać oba te podejścia w jednej aplikacji musimy zadbać o obustronną konwersję pomiędzy danymi z tabel relacyjnej bazy danych a obiektami aplikacjii. Tym właśnie zajmuje się mapowanie obiektowo-relacyjne, które wraz z tworzeniem aplikacji szkieletowych w języku C++ jest główną problematyką niniejszej pracy.

Tutaj powstaje pytanie czy na prawdę warto korzystać z bibliotek i aplikacji szkieletowych służących do mapowania obiektowo relacyjnego? Odpowiedź nie jest jednoznaczna w wszystkich przypadkach, ale warto wymienić jego podstawowe wady i zalety, których dokładniejsza analiza znajduje się w dalszej części pracy. Zacznijmy od zalet wykorzystania narzędzi ORM[1]:

- Znacznie zredukowana ilość pracy wymagana na oprogramowanie dostępu do bazy danych.
- Nieobowiązkowa znajomość języka SQL[2]. W celu tworzenia zapytań do bazy danych korzystamy z interfejsu udostępnianego przez daną bibliotekę czy też aplikację szkieletową.
- Uniezależnienie się od rodzaju systemu zarządzania bazą danych, możemy korzystać równie dobrze z MySQL, Oracle, PostgreSQL czy też Microsoft SQL Server niekoniecznie posiadając o nich rozbudowaną wiedzę.
- Aby skorzystać z transakcji, połączenia z bazą danych a także wieu innych funkcjonalności baz danych wystarczy zazwyczaj wywołać pojedyńczą metodę.
- Cały model danych przechowywany jest w jednym miejscu oraz nie jest ściśle związany z wykorzystywanym systemem zarządzania bazą danych dzięki czemu łatwiej jest nam wprowadzać kolejne modyfikacje w kodzie oraz nawet zmieniać system zarządzania bazą danych na inny.

Wszędzie gdzie pojawiają się zalety mamy też do czynienia z wadami, nie inaczej jest w tym przypadku:

[1] mapowanie obiektowo-relacyjne (ang. Object-Relational Mapping)
[2] strukturalny język zapytań (ang. Structured Query Language)

- Konfiguracja jest najczęściej skomplikowana i wymaga sporo czasu.
- Aby efektownie korzystać z narzędzi do mapowania obiektowo-relacyjnego wymagana jest od nas ich dobra znajomość.
- Proste zapytania są obsługiwane bardzo sprawnie, jednak gdy przetwarzamy duże ilości złożonych zapytań wydajność nie dorówna nigdy zapytaniom napisanym przez specjalistę znającego język SQL.
- Abstrakcja wprowadzona przez narzędzia ORM może okazać się uciążliwa, ponieważ nie zawsze zdajemy sobie sprawę z tego co dzieje się za kulisami w trakcie wykonywania poszczególnch operacji.

Naszym głównym celem jest uczynienie Qubica dobrą alternatywą dla nielicznych, ale istniejących już bibliotek oraz aplikacji szkieletowych realizujących mapowanie obiektowo-relacyjne w języku C++. Wszystkie nam obecnie znane rozwiązania są dostępne za darmo, jednak nie udostępniają one generatora opisu będącego w stanie wygenerować cały projekt aplikacji a ich interfejsy nie należą do intuicyjnych. Wprowadzenie generatora oraz intuicyjnego interfejsu użytkownika powinno uczynić Qubica istotną alternatywą dla istniejących już narzędzi zakładając, że pozostała funkcjonalność mapowania obiektowo-relacyjnego zostanie zrealizowana poprawnie.

Analizę istniejących rozwiązań przedstawia kolejny rozdział a zestawienie z rezultatami naszej pracy znajduje się natomiast w końcowej części pracy, gdzie zostaną przedstawione uzyskane przez nas wyniki oraz podsumowanie wykonanej przez nas pracy.

1.3 Cele pracy

Do najważniejszych celów niniejszej pracy dyplomowej należą:

- Analiza istniejących bibliotek oraz aplikacji szkieletowych realizujących mapowanie obiektowo-relacyjne w języku C++ – przeanalizowanie istniejących już narzędzi umożliwi dokładniejsze zapoznanie się z tematyką mapowania obiektowo-relacyjnego a także ze sposobem działania istniejących już rozwiązań, co umożliwi zaczerpnięcie najciekawszych pomysłów dla Qubica a także wskaże elementy, które mogą ulec w nim poprawie.

- Stworzenie własnej aplikacji szkieletowej realizującej mapowanie obiektowo-relacyjne w języku C++ – ułatwi dokładniejsze poznanie mechanizmów działających podczas mapowania obiektowo-relacyjnego oraz sposobów rozwiązania pojawiających się problemów.

- Porównanie szybkości działania oraz ilości kodu wymaganego do stworzenia przykładowej aplikacji stworzonej w oparciu o Qubic a o inne istniejące narzędzia – przeprowadzenie testów pozwoli stwierdzić czy przyjęte założenia i zastosowane rozwiązania wpłynęły na stworzenie wartej uwagi aplikacji szkieletowej realizującej mapowanie obiektowo-relacyjne.

Do celów części praktycznej należą:

- Stworzenie intuicyjnego interfejsu użytkownika – aby sprawdzić w jakim stopniu udało się zrealizować założenia zostanie przeprowadzony test polegający na napisaniu tej samej aplikacji wykorzystującej Qubica oraz inne aplikacje szkieletowe i bibioteki, a następnie porównaniu ilości linii kodu stworzonych aplikacji.

- Stworzenie generatora opisu mapowania obiektowo-relacyjnego – jest to przedmiotem pracy Sebastiana, naszym wspólnym celem jest integracja obu modułów.

- Poprawne zrealizowanie założeń mapowania obiektowo-relacyjnego, a także jak najlepsza optymalizacja zapytań – aby sprawdzić w jakim stopniu udało się zrealizować założenia zostanie przeprowadzony test polegający na napisaniu tej samej aplikacji wykorzystującej Qubica oraz inne aplikacje szkieletowe i bibioteki, a następnie porównaniu ich wydajności.

- Uczynienie konfiguracji Qubica jak najprostszą.

1.4 Metoda badawcza

1.4.1 Studia literaturowe

Literatura wykorzystana podczas pisania niniejszej pracy dotyczy głównie czterech zagadnień, czyli tworzenia aplikacji szkieletowych w języku C++, tworzenia zapytań w języku SQL, aplikacji szkieletowej Qt oraz mapowania obiektowo-relacyjnego. Pierwsze dwa należą są dość popularne, dlatego też wybór pozycji książkowych jest dość spory. Na temat mapowania obiektowo-relacyjnego trudniej jednak znaleźć

podobną ilość książek, jednak istnieje spora ilość artykułów w wersji elektronicznej. Najczęściej są one napisane w języku angielskim. Tytuły wykorzystanych pozycji bibliograficznych wraz z ich krótkimi opisami znajdują się w kolejnym rozdziale.

1.4.2 Analiza istniejących rozwiązań

Poza podstawowym źródłem informacji jakim są studia literaturowe podczas pisania tej pracy przeprowadzona została analiza istniejących już narzędzi realizujących mapowanie obiektowo-relacyjne. Analiza taka umożliwia poznanie praktycznych rozwiązań problemów pojawiających się podczas prac badawczych nad daną tematyką.

1.4.3 Stworzenie własnej aplikacji szkieletowej

Praktyka jest najczęściej najlepszą z dostępnych metod nauki i to właśnie podczas tworzenia własnej aplikacji szkieletowej realizującej mapowanie obiektowo-relacyjne można się najbardziej z danym tematem zapoznać. Wszystkie problemy, które pojawiały się poczas pisania Qubica musiały zostać w pewien sposób rozwiązane i to właśnie analiza tych problemów i ich rozwiązywanie było główną metodą badawczą wykorzystaną podczas pisania niniejszej pracy.

1.4.4 Analiza porównawcza oraz testy

Analizując wcześniej istniejące już rozwiązania i powównując je z własnym można dojść do najtrafniejszych wniosków. To właśnie na tym etapie często dowiadujemy się czy przyjęte przez nas założenia i zaproponowane rozwiązania były lepsze niż te z istniejących już rozwiązań.

1.5 Przegląd literatury w dziedzinie

1.5.1 Literatura dotycząca języka C++ oraz Qt

W celu zasięgnięcia informacji na temat języka C++ i zagadnień z nim związanych najczęściej wykorzystywaną pozycją książkową była „Symfonia” Jerzego Grębosza [1]. Najlepszym jej określeniem jest „kurs programowania w języku C++”, opisane zostały w niej jednak zagadnienia dotyczące nie tylko języka C++, a także co istotne dla autora niniejszej pracy zagadnienia dotyczące obiektowości.

Poza tym wartościowym źródłem wiedzy podczas tworzenia Qubica była specyfikacja języka C++ [6] oraz dokumentacja aplikacji szkieletowej Qt [7].

1.5.2 Literatura dotycząca języka SQL

Kluczowym zadaniem Qubica jest tworzenie jak najefektywniejszych zapytań w języku SQL, wiedza autora na ten temat pochodzi w głównej mierze z książki Johna Viescasa o tytule „SQL Queries for Mere Mortals" [4]. Wyjaśnione zostały w niej zagadnienia dotyczące tworzenia zapytań w języku SQL oraz podstawy związane z bazami danych.

W tym przypadku także wartościowe okazały się specyfikacje na przykład języka MySQL [8].

1.5.3 Literatura dotycząca mapowania obiektowo-relacyjnego

Głównym źródłem wiedzy autora na temat mapowania obiektowo-relacyjnego były książki „Hibernate w akcji" [2] oraz „Java Persistence with Hibernate" [3] napisane przez Christiana Bauera oraz Gavina Kinga. Ich tytuły mogą być mylące, wstępne rozdziały dokładnie opisują tematykę mapowania obiektowo-relacyjnego nie uzględniając kontekstu języka Java czy aplikacji szkieletowej Hibernate.

Ponadto liczne źródła elektroniczne, wśród których dominują artykuły naukowe również okazały się pomocne. Odniesienia do nich znajdują się w rozdziale 2, gdzie pojęcia dotyczące mapowania są objaśniane.

1.6 Układ pracy

Tematem niniejszej pracy jest mapowanie obiektowo-relacyjne w języku C++, zaś za jej główny cel przyjęto przeanalizowanie istniejących bibliotek oraz aplikacji szkieletowych realizujących mapowanie obiektowo-relacyjne oraz stworzenie własnej aplikacji szkieletowej.

Najważniejszym celem pracy jest przeanalizowanie istniejących narzędzi służących do mapowania obiektowo-relacyjnego w języku C++ oraz stworzenie na podstawie tej analizy własnej aplikacji szkieletowej realizującej to samo zagadnienie.

Rodział 1 otwiera uzasadnienie wyboru tematu pracy, opisane zostały w nim także cele pracy, jej problematyka, zakres, wykorzystane metody badawcze, przegląd literatury oraz jej układ.

Rozdział 2 zawiera objaśnienia podstawowych zagadnień teotetycznych związanych z mapowaniem obiektowo-relacyjnym, czyli między innymi pojęć takich

jak trwałość danych, relacyjne bazy danych czy też język SQL.

Rozdział 3 przedstawia analizę istniejących już narzędzi służących do mapowania obiektowo-relacyjnego napisanych w języku C++.

Rozdział 4 przedstawia projekt aplikacji szkieletowej Qubic, postawione wymagania, opis jego implementacji, wyniki testów oraz końcowe porównanie z wcześniej analizowanymi narzędziami w rozdziale 3.

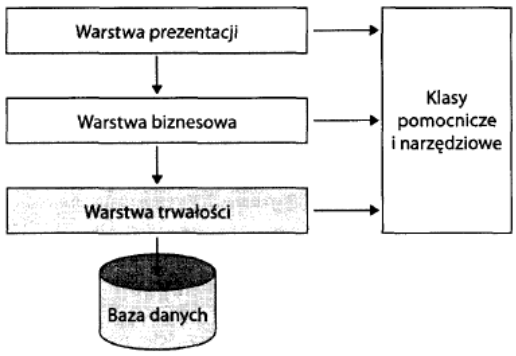
Rozdział 5 zawiera podsumowanie niniejszej pracy inżynierskiej, biorąc pod szczególną uwagę dyskusję wyników oraz dalsze prespektywy rozwoju pracy.

Rozdział 2

Zagadnienia teoretyczne dotyczące mapowania obiektowo-relacyjnego

2.1 Architektura warstwowa

Architektura warstwowa opisuje interfejs pomiędzy kodem, który implementuje różne zadania, by zmiana w jednym z zadań i sposobie jego wykonania nie odbijała się na zmianach w innych warstwach. W ten sposób programista jest w stanie sprawniej wprowadzać poprawki oraz nowe elementy do tworzonej aplikacji. Typowa i sprawdzona architektura aplikacji wysokiego poziomu składa się z trzech warstw: prezentacji (nazywanej też warstwą widoku), logiki oraz trwałości danych.



Rys. 2.1: Warstwowa architektura aplikacji [2]

Zadaniem poszczególnych warstw jest:

- Warstwa prezentacji – zawiera logikę interfejsu użytkownika, czyli między innymi kod odpowiedzialny za wyświetlanie okien, formularzy czy też tabel.

- Warstwa logiki – jej postać bywa zróżnicowana, przyjmuje się jednak, że powinna zawierać implementację reguł biznesowych i wymagań systemowych, które zostały uznane za dziedzinę rozwiązywanego problemu.
- Warstwa trwałości danych – stanowi grupę klas i komponentów odpowiedzialnych za zapamiętywanie danych i odczytywanie ich z wybranych źródeł. Zawiera ona także model elementów dziedziny biznesowej.

2.2 Trwałość danych

Kluczowym zadaniem podczas tworzenia aplikacji jest zapewnienie trwałości danych, co oznacza zapewnienie, że po zakończeniu działania aplikacji zebrane dane nie mogą zostać utracone [3]. W przeciwnym wypadku znalezienie zastosowania dla takich aplikacji byłoby znacznie cięższym zadaniem a rozwiązanie wielu problemów za ich pomocą byłoby niemożliwe. Nie możemy przecież wyobrazić sobie systemu bankowego po którego ponownym uruchomieniu dane na temat wszystkich klientów i posiadanych przez nich środków zostałyby utracone.

2.3 Relacyjne bazy danych

Większość programistów problem trwałości danych rozwiązuje poprzez wykorzystanie relacyjnych baz danych jako „magazynu danych" oraz języka SQL w roli „zarządzającego danymi". Pojęcia te są powszechnie znane, jednakże warto przypomnieć ich definicje:

Definicja 1 *Relacyjna baza danych – zbiór danych w postaci tabel połączonych ze sobą relacjami [10].*

Definicja 2 *Język SQL – strukturalny język zapytań pozwalający na wprowadzanie zmian w strukturze bazy danych, a także zmian w samej bazie czy też pobieranie z niej informacji. Język ten opiera się na silniku bazy danych, który pozwala tworzyć zapytania [4].*

Duża popularność relacyjnych baz danych wynika z ich licznych zalet, do których należą między innymi łatwość modyfikacji przechowywanych w nich danych, zmniejszona możliwość popełnienia pomyłki czy też duża elastyczność i szybkość w zarządzaniu danymi. Ma to miejsce kosztem zmniejszonej wydajności w stosunku do innych modeli danych [3]. Relacyjne bazy danych swoje zastosowanie

znajdują w wielu systemach i platformach technologicznych, są one najczęściej podstawową reprezentacją dla elementów biznesowych [2].

Aby jak najskuteczniej korzystać z narzędzi mapowania obiektowo-relacyjnego warto zaznajomić się z relacyjnymi bazami danych oraz językiem SQL. Wiedza ta umożliwia tworzenie aplikacji działających sprawniej i bardziej odpornych na wszelkiego rozdzaju błędy. Szczególnie przydatna może okazać się znajomość języka SQL, który jest fundamentem mapowania obiektowo-relacyjnego oraz wszystkich innych aplikacji wykorzystujących relacyjne bazy danych w celu zapewnienia trwałości danych.

2.4 Programowanie obiektowe

W aplikacjach stworzonych w oparciu o paradygmat programowania obiektowego, trwałość danych umożliwia przechowanie obiektów po zakończeniu działania aplikacji aż do momentu kiedy przy jej ponownym uruchomieniu nie zajdzie potrzeba jego wczytania z powrotem. Obiekt jest przechowywany na dysku twardym, a nie jak wcześniej w pamięci operacyjnej komputera, a ich ilość nie jest ograniczona, „utrwalany" może być pojedyńczy obiekt, ale także całe ich struktury. Warto pamiętać, że większość obiektów nie jest trwała. Obiekty ulotne mają ograniczony czas życia, związany z tworzonym je procesem czy też metodą [2].

Obecne relacyjne bazy danych udostępniają zestaw mechanizmów umożliwiających manipulowanie, sortowanie, wyszukiwanie czy też zbieranie danych. Odpowiedzialne są one także za nadzorowanie operacjami współbieżnymi oraz nad integralnością danych. W przypadku gdy z bazą połączonych jest kilka aplikacji klienckich do jej zadań należy zarządzanie wszystkimi operacjami oraz unikanie błędów. Decydując się na wykorzystanie relacyjnych baz danych wymienione mechanizmy wykonują za nas sporą część pracy [3].

2.5 Wykorzystanie SQL w C++

Podczas korzystania z bazy danych w aplikacjach C++ wykorzystywane są łączniki, które przesyłają instrukcje SQL do bazy danych. Instrukcje mogą być tworzone ręcznie lub generowane przy użyciu kodu C++, zadaniem łącznika jest przesłanie zapytania, odebranie odpowiedzi bazy danych oraz powiadomienie o ewentualnych błędach.

Większość programistów jednak najbardziej zainteresowana jest problemem biznesowym który musi rozwiązać, a tworzenie zapytań oraz zarządzanie bazą danych

im to w pewien sposób utrudniają. System, który wykona wszystkie zadania związane z trwałością danych za programistę tak, żeby ten mógł poświęcić się jedynie rozwiązaniu postawionego przed nim problemu wydaje się być najlepszym rozwiązaniem takiej sytuacji.

W związku z tym, że oprogramowanie dostępu do relacyjnej bazy danych z poziomu obiektowych aplikacji nie należy do najłatwiejszych zadań wiele osób będzie się zastanawiać czy na prawdę warto to robić. Czy nie lepiej skorzystać z mało popularnych, ale istniejących przecież obiektowych baz danych? Panująca sytuacja pokazuje, że nie. Relacyjne bazy danych okazują się najbardziej sprawdzoną technologią i zdecydowana większość aplikacji je wykorzystuje.

2.6 Niedopasowanie paradygmatów

Jak zauważają autorzy książki „Hibernate w akcji" podstawowym problemem w przypadku wykorzystania relacyjnych baz danych oraz programowania obiektowego jest niedopasowanie paradygmatów. Problem ten może zostać podzielony na kilka mniejszych problemów:

- Problem szczegółowości – programiści mogą tworzyć nowe klasy z różnym poziomem szczegółowości. Mniej znacząca klasa, na przykład `Adres`, może zostać osadzona w klasie bardziej znaczącej, na przykład `Użytkownik`. W bazie danych szczegółowość może zostać zaimplementowana jedynie na poziomie tabeli `Użytkownik` oraz kolumny `Adres`.

- Problem podtypów – dziedziczenie oraz polimorfizm to podstawowe mechanizmy programowania obiektowego i w praktycznie każdej aplikacji znajdziemy ich wykorzystanie, relacyjne bazy danych nie udostępniają jednak tych mechanizmów co jest kolejną przyczyną niedopasowania.

- Problem identyczności – w przypadku bazy danych encje są identyczne gdy ich klucze główne są takie same. W programowaniu obiektowym obiekty są sobie równe gdy mają takie same wartości pól lub gdy są identyczne (mają tę samą referencję).

- Problemy dotyczące asocjacji –

- Problem nawizgacji po grafie obiektów –

2.7 Koszt niedopasowania

2.8 Mapowanie obiektowo-relacyjne

Poza przedstawionym wcześniej klasycznym podejściem zapewniania trwałości danych w aplikacjach, w którym programista sam tworzy bazę danych a potem nią zarządza konstruując zapytania w języku SQL, istnieje także inne rozwiązanie tego problemu nazywane najczęściej mapowaniem, bądź też odwzorowaniem obiektowo-relacyjnym [2].

W pierwszym rozdziale podczas opisu problematyki pracy została przedstawiona uproszczona definicja mapowania obiektowo-relacyjnego, biorąc pod uwagę pojęcia opisane w tym rozdziale można ją tym razem zapisać w sposób następujący:

Definicja 3 *Mapowanie obiektowo-relacyjne – automatyczna i niewidoczna dla użytkownika realizacja trwałości obiektów w aplikacji zamieniająca je na wpisy w relacyjnej bazie danych na podstawie metadanych opisujących klasę. Mapowanie działa jako dwukierunkowy translator danych z jednej reprezentacji w inną [2].*

Narzędzia realizujące mapowanie obiektowo-relacyjne składają się z w większości przypadków następujących elementów [2]:

- Interfejsu do przeprowadzania podstawowych operacji CRUD[1] na obiektach klas zapewniających trwałość.
- Interfejsu umożliwiającego tworzenie zapytań związanych z klasami oraz ich właściwościami.
- Narzędzia do określania metadanych.
- Technik takiej implementacji mapowania, aby poprawnie współgrało ono z obiektami transakcyjnymi wykonując wszystkie dostępne operacje.

W zasadzie więc terminu mapowania obiektowo-relacyjnego można użyć w odniesieniu do dowolnej warswy trwałości, która zapytania do bazy danych generuje automatycznie na podstawie opisu w postaci metadanych. Tworzenie jednak takiego generatora dla pojedyńczej aplikacji często mija się z celem i lepszym rozwiązaniem jest skorzystanie z gotowego narzędzia służacego do mapowania bądź też własnoręczna implementacja warstwy trwałości w wybranej aplikacji. W przypadku wyboru jednego z gotowych narzędzi w ogóle nie musimy zajmować się implementacją warstwy trwałości, ponieważ korzystamy z już wcześniej przygotowanej.

Do sposobów implementacji mapowania obiektowo-relacyjnego należą [2]:

[1] utwórz, odczytaj, zaktualizuj oraz usuń (ang. Create, Read, Update and Delete)

- Pełna relacyjność – cała aplikacja, włączając w to interfejs użytkownika, zaprojektowana jest wokół modelu relacyjnego i podstawowych operacji języka SQL. Aplikacje takie część logiki mogą mieć przeniesione do warstwy bazy danych.

- Lekkie odwzorowanie obiektów – encje są reprezentowane przez ręczne napisane klasy odpowiadające tabelom relacynym. Kod SQL zostaje schowany przed logiką aplikacji dzięki wzorcom projektowym.

- Średnie odwzorowanie obiektów – aplikacja jest zaprojektowana wokół modelu obiektowego. Kod SQL jest generowany dynamicznie przez szkielet systemu.

- Pełne odwzorowanie obiektów – obsługuje wyrafinowane modele obiektowe stosując kompozycję, dziedziczenie i polimorfizm. Warstwa trwałości w sposób niewidoczny implementuje zapis i odczyt danych. Ten poziom funkcjonalności jest najtrudniejszy do osiągnięcia i uzyskiwanie go na potrzeby pojedyńczej aplikacji nie ma sensu.

Dlaczego warto korzystać z mapowania obiektowo-relacyjnego? Podstawową zaletą dla programisty jest z pewnością możliwość uniknięcia pisania własnoręcznie wszystkich zapytań, a zamiast tego skorzystanie z gotowych metod wybranego narzędzia. Mogłoby to sugerować, że decydując się na korzystanie z mapowania nie musimy znać się ani na relacyjnych bazach danych, ani na języku SQL, jest to jednak nieprawda, ponieważ wydajność tworzonych w taki sposób aplikacji byłaby znacznie mniejsza od tych napisanych przez programistów którym te pojęcia nie są obce. Przyjrzyjmy się jeszcze raz podstawowym zaletom mapowania obiektowo-relacyjnego [2]:

- Produktywność – warstwa trwałości jest bardzo niewdzęczną warstwą do oprogramowania, decydując się na skorzystanie z narzędzi mapujących problem ten zostaje wyeliminowany a cała uwaga programisty może zostać poświęcona innym warstwom aplikacji.

- Konserwacja – brak warstwy logiki w kodzie napisanym przez programistę czyni system bardziej zrozumiałym, czytelniejszym, a co za tym idzie łatwiejszym do przekształcenia.

- Wydajność – prawdą jest stwierdzenie, że warstwa trwałości napisana ręcznie przez programistę potrafi być co najmniej tak szybka jak ta wygenerowana automatycznie. Podobnie prawdą jest, że dowolny program napisany w C++

można napisać w Assemblerze by działał co najmniej tak szybko jak ten pierwszy. Jednak jak dobrze wiadomo, znacznie trudniejszym zadaniem jest napisanie programu w języku asemblera aby działał on tak samo sprawnie jak ten napisany w C++. Aby to zrobić trzeba bardzo dobrze znać Assemblera, analogiczna sytuacja ma miejsce podczas własnoręcznego tworzenia warstwy trwałości i znajomości języka SQL.

- Niezależność – abstrakcja języka SQL uwalnia programistę od jego szczegółów i różnych jego dialektów w poszczególnych systemach bazodanowych. Większość narzędzi obsługuje wiele systemów bazodanowych, a więc nawet ich zmiana po napisaniu aplikacji nie powinna być problemem.

Mapowanie obiektowo-relacyjne jest rozwiązaniem sprawdzającym się obecnie w bardzo dużej ilości istniejących już aplikacji i jego wybór przed własnoręczną implementacją zapytań jest najczęściej trafną decyzją. W chwili są to dwa najpopularniejsze rozwiązania, przy czym mapowanie jest obecnie najczęściej wybierane w związku z niedopasowaniem obiektowo-relacyjnym.

-

-

Rozdział 3

Analiza istniejących rozwiązań mapowania obiektowo-relacyjnego w języku C++

3.1 Kryteria analizy

-

3.2 Przegląd istniejących rozwiązań

-

3.3 Porównanie istniejących rozwiązań

Nazwa	QxOrm	Debea	SOCI	OpenORM
Typ	-	-	-	-
Cena	0 zł	0 zł	0 zł	0 zł
Obsługiwane bazy	-	-	-	-

Tab. 3.1: Porównanie istniejących rozwiązań

-

Rozdział 4

Aplikacja szkieletowa Qubic

4.1 Moduły tworzonej aplikacji

W rozdziale pierwszym w skrócie został przedstawiony schemat współpracy modułu tworzonego przez autora tej pracy a autora pracy, której tematem jest „Generator opis mapowania obiektowo-relacyjnego w języku C++”. W tym podrozdziale opis ten zostanie rozwinięty.

Chcąc w jak największym stopniu zautomatyzować obsługę połączenia z bazą oraz zminimalizować czas jaki programista będzie musiał poświęcić na oprogramowywanie komunikacji pomiędzy programem a bazą danych zdecydowaliśmy się na wprowadzenie generatora opisu, który tę część pracy wykona za użytkownika Qubica.

Zadaniem generatora jest wygenerowanie plików klas, plików nagłówkowych oraz pliku projektu Qt w oparciu o istniejącą bazę danych. Moduł tworzony przez autora niniejszej pracy będzie wykorzystywany w wygenerowanym kodzie, a także w kodzie napisanym przez użytkownika Qubica. W momencie zakończenia pracy generatora wygenerowana aplikacja będzie w pełni gotowa do przechowywania danych z bazie bez konieczności tworzenia zapytań. W celu przechowania obiektów aplikacji, ich modifikacji, załadowania czy usunięcia z bazy danych wystarczy uruchomienie odpowiednich metod Qubica.

Dalsza część tego rozdziału została poświęcona w zdecydowanej większości modułowi Qubica zajmującego się mapowaniem obiektowo-relacyjnym.

4.2 Analiza wymagań

4.2.1 Wymagania funkcjonalne

Podczas projektowania Qubica przyjęto następujące założenia w celu jak najlepszego odwzorowania cech mapowania obiektowo-relacyjnego:

- Aplikacja musi umożliwiać podstawowe operacje mapowania obiektowo-relacyjnego, a więc zapisywanie obiektów do bazy danych, ich odczyt, aktualizowanie obiektów już zapsanych w bazie oraz ich usuwanie.
- Aplikacja musi udostępniać interfejs do tworzenia zapytań z poziomu kodu, dzięki temu jej użytkownik nie musi znać języka SQL.
- Aplikacja musi udostępniać funkcje dostępu do powiązanych danych w przypadku wystąpienia relacji różnych od jeden do jednego. Użytkownik powinien mieć możliwość uzyskania dostępu do obiektów powiązanych z wybranym obiektem bez konieczności własnoręcznego konstruowania zapytań.
- Aplikacja powinna posiadać wsparcie dla wielu rodzajów baz danych, ewentualnie musi być ona łatwo rozszerzalna.
- Aplikacja musi posiadać możliwość wykorzystania transakcji.
- Aplikacja musi posiadać możliwość konfiguracji.

4.2.2 Wymagania niefunkcjonalne

Do wymagań niefunkcjonalnych postawionych projektowanej aplikacji należą:

- Użytkowanie aplikacji powinno być jak najbardziej intuicyjne, co za tym idzie kod użytkownika mający za zadanie wykonywać podstawowe operacje powinien zajmować jak najmniej linii.
- Aplikacja musi działać możliwie szybko, czas podstawowych operacji nie powinien znacząco odbiegać od tego w przypadku gdy użytkownik sam tworzyłby zapytania do bazy.
- Aplikacja musi rozpoznawać relacje jeden do jednego, jeden do wielu oraz wiele do wielu i odpowiednio je obsługiwać.
- Pamięć w trakcie działania aplikacji musi być odpowiednio zarządzana, niedopuszczalne są żadne wycieki pamięci czy też zapętlenia się programu.

- Błędy pojawiające się w trakcie działania aplikacji powinny być prawidłowo obsługiwane i sygnalizowane użytkownikowi.

4.3 Projekt

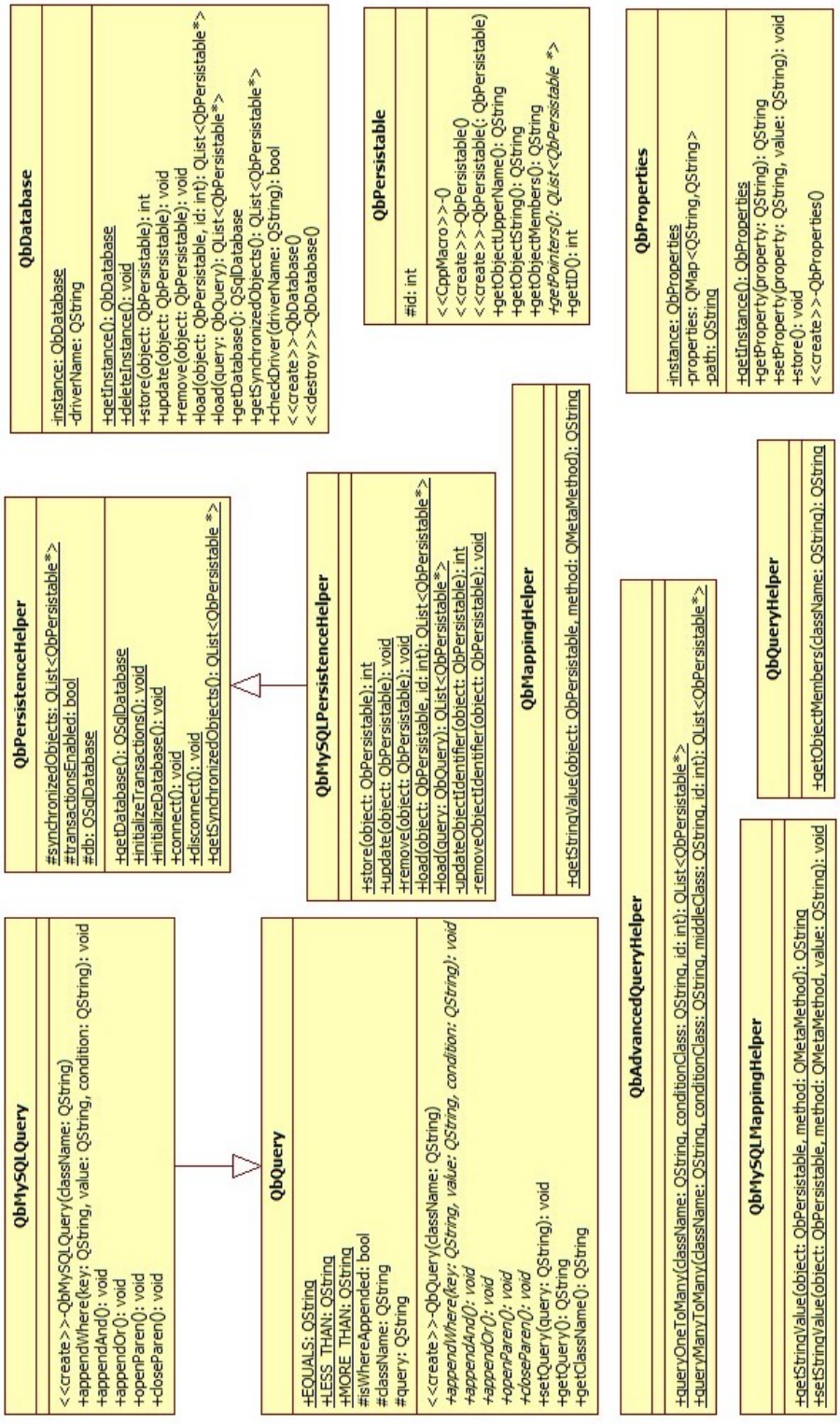
4.3.1 Rodzaj aplikacji

Qubic jest aplikacją szkieletową, czyli strukturą do wykorzystywaną do budowy innych aplikacji. Jego podstawowym zadaniem jest realizacja mapowania obiektowo-relacyjnego, czyli udostępnienie funkcjonalności pozwalającej na przechowywanie obiektów w bazie danych.

W celu stworzenia aplikacji w oparciu o Qubica użytkownik musi przygotować bazę danych na podstawie której wygenerowany zostanie plik projektu Qt wraz z plikami nagłówkowymi oraz plikami klas. W tym momencie aplikacja posiada już oprogramowany dostęp do danych i użytkownik może zająć się implementowaniem własnej logiki czy też warstwy widoku.

4.3.2 Diagram klas

Diagram klas Qubica przedstawia rysunek 4.1:



Rys. 4.1: Diagram klas

4.3.3 Wzorce projektowe

Jednym z wzorców projektowych, które znalazły zastosowanie w Qubicu jest Singleton. Do jego największych zalet należy fakt, że klasa go wykorzystująca może posiadać co najwyżej jedną instancję do której istnieje globalny dostęp. Singleton swoje zastosowanie znajduje w przypadku gdy programista chce ograniczyć liczbę instancji dla wybranych klas a także samemu odpowiadać za ich tworzenie. W przypadku Qubica został on wykorzystany w klasach `QbDatabase` oraz `QbProperties`. Tworzenie więcej niż jednej instancji tych klas nie ma sensu biorąc pod uwagę specyfikę projektu, więc najlepszym rozwiązaniem wydaje się być zablokowanie tej możliwości użytkownikowi.

4.3.4 Środowisko programistyczne

Qubic został napisany w języku C++ w oparciu o aplikację szkieletową Qt, co za tym idzie wybór platformy należy do użytkownika i równie dobrze może to być Windows jak i Linux. Zarówno wybór bazy danych nie został narzucony z góry, co prawda na potrzeby projektu zaimplementowana została obsługa tylko dla bazy MySQL jednak zapewniona została łatwa rozszerzalność i dodanie obsługi dla innych rodzajów baz danych nie powinna być problemem.

Jedyną biblioteką wykorzystywaną przez Qubica jest QsLogger odpowiedzialny za logowanie. Jest to bardzo przydatna funkcja, która umożliwia szybkie zlokalizowanie pojawiających się problemów.

Wykorzystane technologie, poprawiony opis, screeny, czy coś jeszcze?

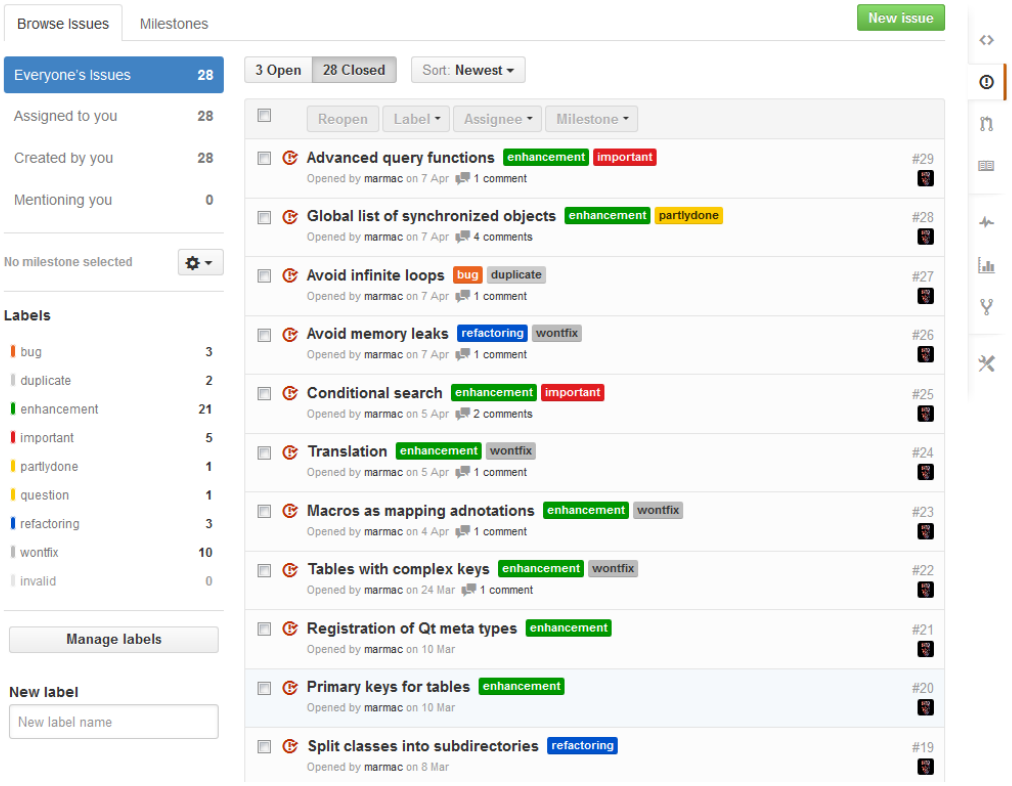
4.3.5 System kontroli wersji

Podczas pracy nad projektami w których bierze udział więcej niż jedna osoba bardzo dobrym rozwiązaniem jest korzystanie z systemów kontroli wersji. Ułatwiają one wspólną pracę oraz organizację tworzonych projektów.

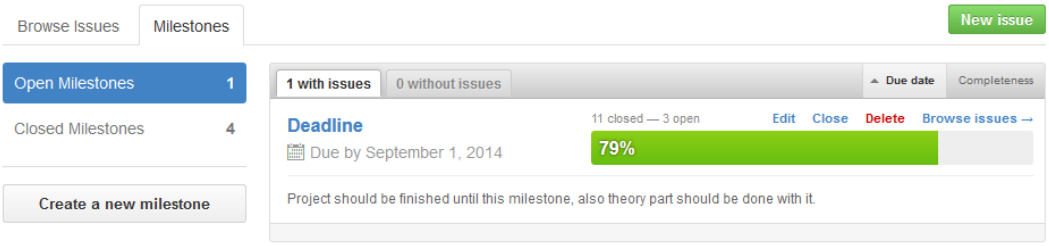
Wybór autorów Qubica padł na dobrze znany serwis GitHub[1]. Poza repozytorium na którym znaleźć można kod źródłowy Qubica podczas implementacji projektu wykorzystany został system Issues and Milestones[2], który ułatwia organizację pracy nad projektem.

[1] oficjalna strona serwisu – `https://github.com/`

[2] problemy oraz kamienie miliowe



Rys. 4.2: Przejrzysty widok systemu GitHub Issues



Rys. 4.3: Przejrzysty widok systemu GitHub Miletsones

4.4 Implementacja

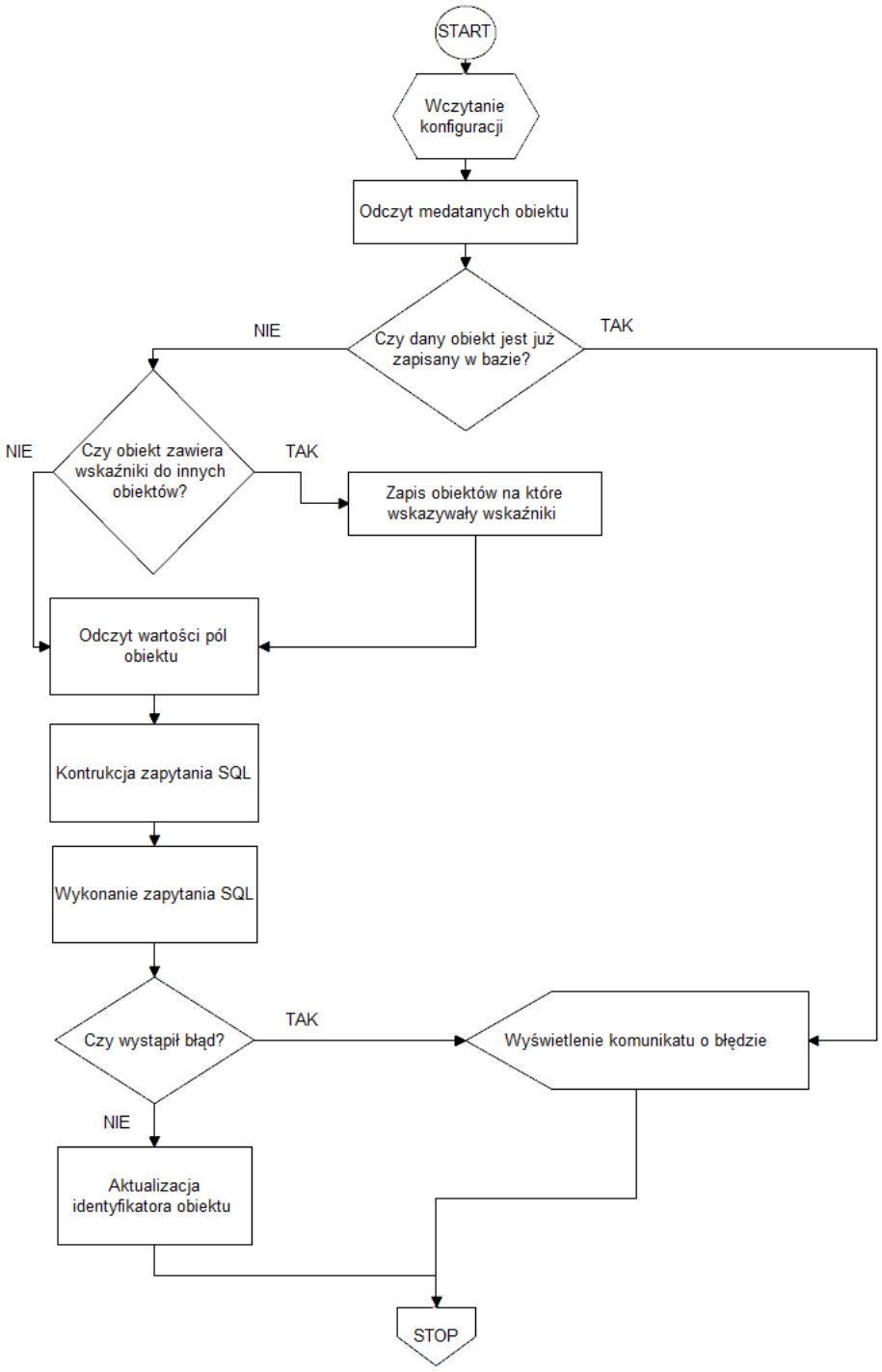
4.4.1 Interfejs CRUD

Kluczowym modułem narzędzi programistycznych realizujących mapowanie obiektowo-relacyjne jest interfejs CRUD umożliwiający manipulowanie danymi w bazie

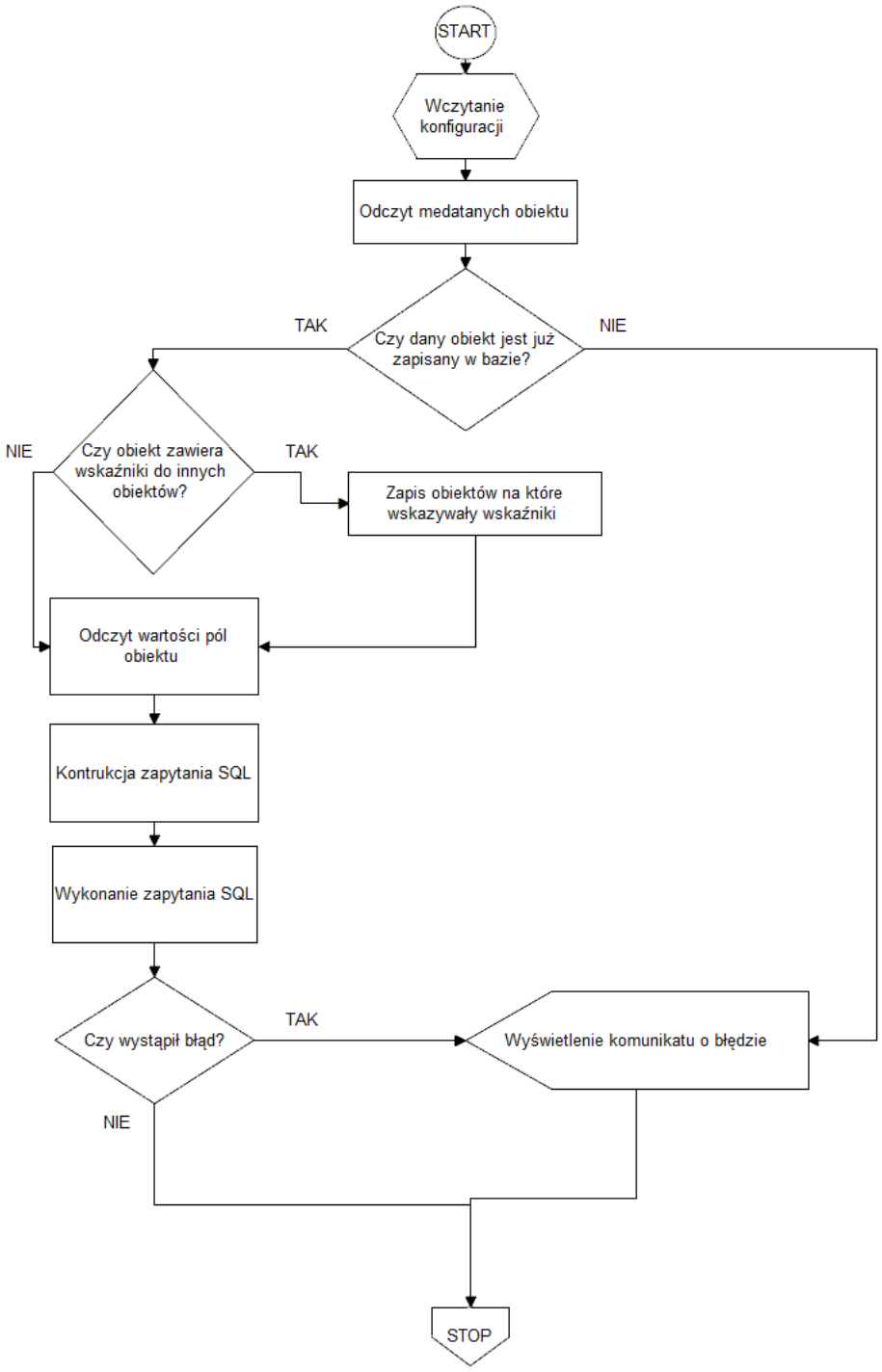
danych z poziomu kodu. Interfejs ten dostępny jest w podstaci co najmniej czterech metod, które jako argumenty przyjmują obiekty dziedziczące po jednej z klas Qubica – `QbPersistable`. Umożliwia to wykorzystanie polimorfizmu, a przecież podczas operacji na obiektach od samego początku nie znany jest dokładny typ obiektu.

Aby mapowanie mogło skutecznie działać potrzebny jest jego jednoznaczny opis, część narzędzi je realizujących korzysta z adnotacji, które określają mapowanie pomiędzy konkretnymi polami klas a konkretnymi tabelami w bazie danych. W Qubicu problem ten został rozwiązany w trochę inny sposób, który umożliwia stworzony generator opisu mapowania obiektowo-relacyjnego. Generowane pliki klas są tworzone według ściśle określonych zasad, co oznacza, że nazwy metod dostępowych odpowiadają w pewien sposób nazwom tabel i ich kolumn. Przykładowo dla kolumny o nazwie `NAME` w tabeli `EMPLOYEE` w pliku klasy `Employee` zostaną wygenerowane metody `getName()` oraz `setName(QString name)`. Wiedza ta została wykorzystana w trakcie korzystania z mechanizmu refleksji. Konfiguracja samego mapowania nazw została zapisana w pliku konfiguracyjnym Qubica.

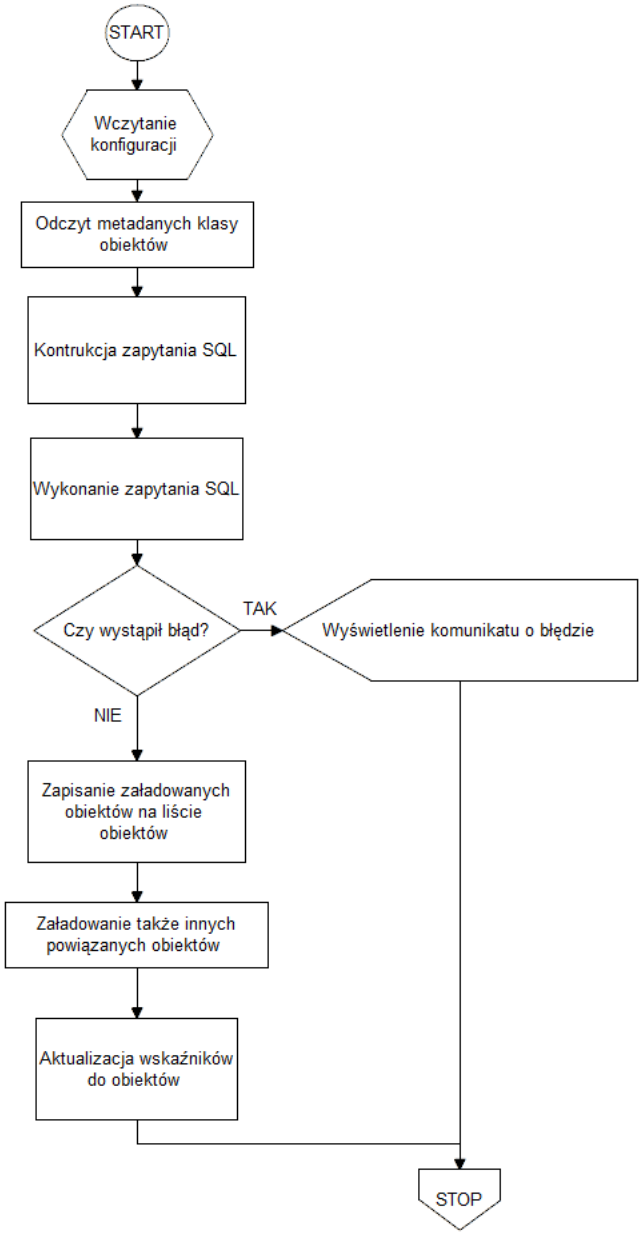
Schematy działania czterech podstawowych metod dostępu przedstawiają poniższe schematy blokowe:



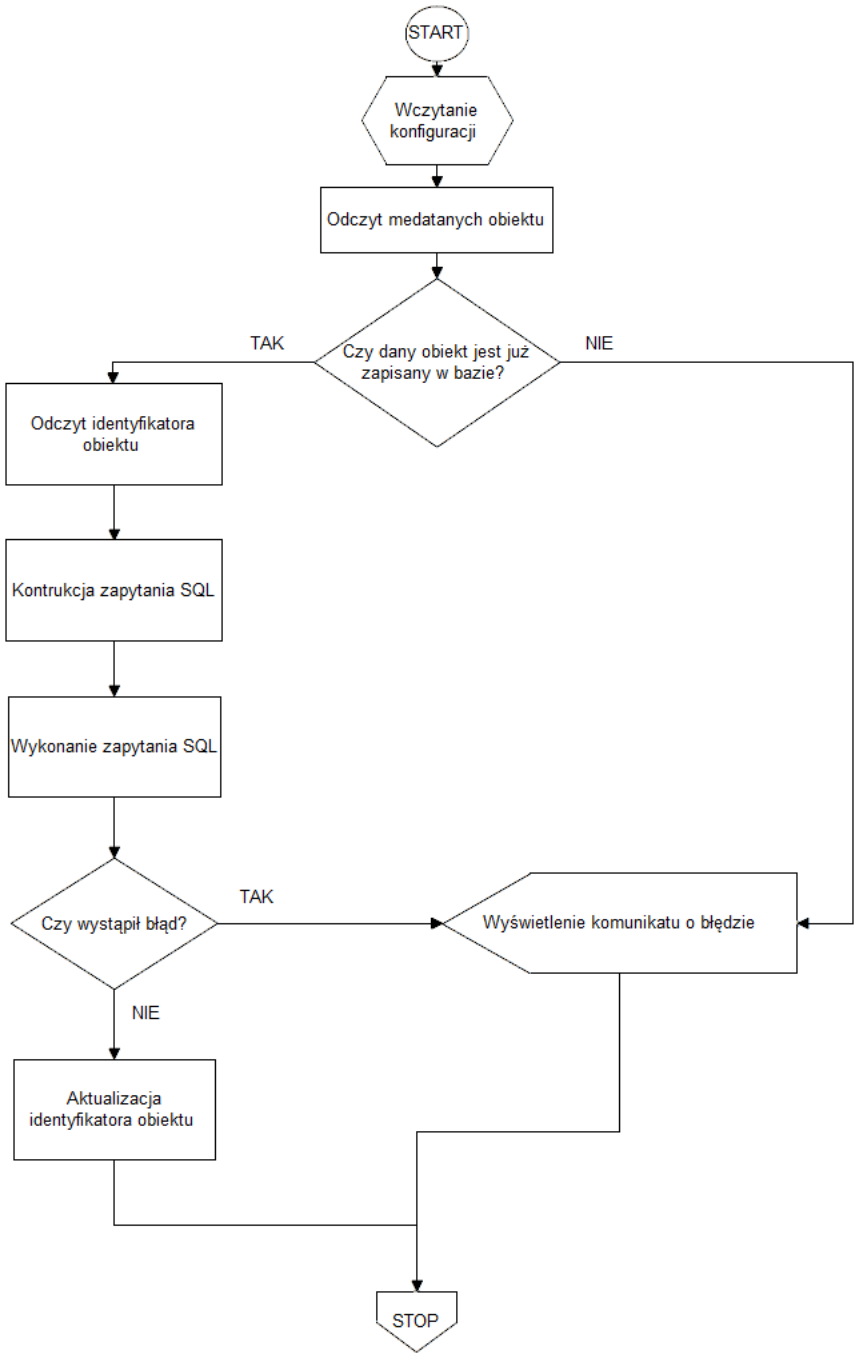
Rys. 4.4: Schemat blokowy metody zapisującej obiekty w bazie danych



Rys. 4.5: Schemat blokowy metody aktualizującej obiekty w bazie danych



Rys. 4.6: Schemat blokowy metody ładującej obiekty z bazy danych



Rys. 4.7: Schemat blokowy metody usuwającej obiekty z bazy danych

-

-

-

4.5 Konserwacja i inżynieria wtórna

Jak przebiega eksploatacja projektu? Jakie wady i zalety ujawniły się po okresie testowania i użytkowania? Jak można skorzystać z tej wiedzy praktycznej pod kątem rozbudowy pracy? Jakie elementy systemu powinny zostać w pierwszej kolejności zmodyfikowane?

4.6 Dokumentacja użytkownika

4.7 Przykładowa aplikacja wykorzystująca Qubica

-

-

-

-

-

4.8 Testy oraz ich wyniki

4.9 Perspektywy rozwoju Qubica

W celu dalszego rozwoju stworzonej aplikacji szkieletowej warto rozważyć wprowadzenie następujących usprawnień:

- System adnotacji – obecnie na opis mapowania składają się odpowienie nazwy funkcji oraz makra Qt. Istnieje jednak możliwość wprowadzenia własnych makr, które miałyby opisywać mapowanie pomiędzy nazwami tabel z baz danych a odpowiednimi polami klas napisanych z języku C++. Dzięki temu zaistniałaby możliwość uniezależnienia nazw pól klas od nazw tabel w bazie danych.

- Interfejs zapytań – choć jest już zaimplementowany, nadal nie udostępnia on wszystkich możliwych funkcjonalności języka SQL. Implementacja obsługi takich poleceń jak JOIN czy UNION z pewnością byłaby dodatkowym atutem.

- Identyfikacja tabel także za pomocą kluczy złożonych – w tej chwili tabele identyfikowane są za pomocą kluczów głównych, co z kolei wymusza ich nadawanie w każdej z tabel.

- Pamięć podręczna – wprowadzenie pamięci podręcznej może znacznie polepszyć wydajność w przypadku ciągłych operacji na tych samych danych.

- Konfiguracja z poziomu kodu – obecnie większość konfiguracji jest zapisana w plikach konfiguracyjnych i tylko tam może być zmieniana, w celu rozwoju wprowadzenie dodatkowej możliwości jego konfiguracji wydaje się być dobrym pomysłem.

- Wsparcie dla różnych rodzajów baz danych – wprowadzenie tego usprawnienia ogranicza się do implementacji kilku interfejsów dla innych niż MySQL rodzajów baz danych. Biorąc pod uwagę możliwość wzorowania się na zaimplementowanej już logice nie powinno to stworzyć problemu gdy zaistnieje taka konieczność.

- Serializacja danych – dodanie możliwości serializacji może okazać się użyteczne w przypadku pracy z dużymi ilościami danych, w tym celu można skorzystać z wielu istniejących już bibliotek udostępniających tę możliwość.

- Internacjonalizacja – w tej chwili wszystkie logi zlokalizowane są w języku angielskim, istnieje jednak możliwość zmiany obecnego stanu poprzez wykorzystanie modułu translacji udostępnianego przez Qt.

- Wielowątkowość – wykorzystanie wielowątkowości w przypadku mapowania-obiektowo relacyjnego z pewnością nie należy do najłatwiejszych zadań, jednak znacznie może to usprawnić wykonywanie bardziej wymagających operacji.

Rozdział 5

Podsumowanie

5.1 Dyskusja wyników

Dzięki zrealizowaniu pracy poprawie uległa wydajność. Ponadto, o ? % skrócony został czas. Które cele pracy udało sie zrealizować? Co z tego wynika? Które cele pracy pozostały niezrealizowane i dlaczego?

5.2 Perspektywy rozwoju pracy

W celu rozwoju niniejszej pracy dyplomowej należy przede wszystkim rozważyć dalsze prace nad stworzoną aplikacją szkieletową. W rozdziale 4.9 przedstawione zostały liczne możliwości rozwoju Qubica, ich realizacja z pewnością byłaby sporym krokiem w przód.

Podjęcie się tego zadania oznaczałoby jednak trzymanie się tematyki mapowania obiektowo-relacyjnego a przecież aplikacja szkieletowa nie musi ograniczać się do realizacji tylko jednego zagadnienia, istnieje wiele innych możliwości rozwoju. Dobrym tego przykładem jest aplikacja szkieletowa Spring, która oferuje bardzo duże możliwości. Qubic można rozwijać w podobnym kierunku, zarazem zmieniając główny temat pracy dyplomowej. Nowym tematem mogłaby zostać na przykład aplikacja szkieletowa na telefony komórkowe czy też aplikacja szkieletowa do tworzenia usług internetowych[1].

[1] ang. webservice

Bibliografia

[1] Gręboszz, Jerzy. Symfonia C++. Standard. Wyd. 3. Kraków, 2013. ISBN 978-83-7366-134-4.

[2] Bauer, Christian, King, David. Hibernate w akcji. Wyd. 1. Gliwice, 2007. ISBN 978-83-246-0527-9.

[3] Bauer Christian, King David. Java Persistence with Hibernate. Greenwich, 2007. ISBN 1-932394-88-5.

[4] Viescas, John. SQL Queries for Mere Mortals. 2001. ISBN 83-7279-152-X.

[5] Ezust, Alan and Paul. Introduction to Design Patterns in C++ with Qt4. Wyd. 1. Soughton, 2006. ISBN 978-0-13-282645-7.

[6] C++ Language Tutorial. http://www.cplusplus.com/doc/.

[7] Qt Project Documentation. http://qt-project.org/doc/.

[8] MySQL Documentation. http://dev.mysql.com/doc/.

[9] Trendy Google. https://www.google.pl/trends/explore#q=orm%20-java%2C%20orm%20php%2C%20orm%20python%2C%20orm%20c&cmpt=q.

[10] Portal UAM w Poznaniu. http://www.staff.amu.edu.pl/p̃si/-informatyka/kluczew/I2_Database.htm.

Spis rysunków

Spis tabel

Załączniki

1. Kod źródłowy Qubica oraz aplikacji przykładowej w wersji elektronicznej.