Spis treści

Rozdział 1

Wstęp

1.1 Uzasadnienie wyboru tematu

Wraz z upływem czasu postęp technologiczny ma wpływ na życia co raz szerszej rzeszy ludzi na całym świecie. Niezliczone ilości urządzeń zagościły na stałe w domach i mało kto wyobraża sobie bez nich swoje życie. Zaczynając od artykułów gospodarstwa domowego, a kończąc na elektronice użytkowej do której zaliczają się komputery, telewizory czy też smartfony[1]. Wszystkie te urządzenia mają na celu ułatwianie życia swoim użytkownikom.

W parze z licznymi zaletami urządzeń elektronicznych idą jednak pewne wady. Jedną z istotniejszych jest wpływ czasu spędzanego przed różnego rodzaju wyświetlaczami na zdrowie. Badania przeprowadzone na bazie danych Nielsen Audience Measurement pokazują, że przeciętny Polak spędza dziennie średnio 4,5 godziny przed ekranem telewizora[2]. Nie oznacza to jednak, że przez cały ten czas ogląda on telewizję. Oglądanie filmów z dysku komputera, za pomocą serwisów VOD[3] czy granie na konsoli także są wliczone w ten czas. Gdyby jednak dodać do tego czas spędzony przed ekranem smartfona czy też komputera wynik byłby zapewne dwukrotnie większy.

Pogorszenie wzroku czy też wysychanie gałki ocznej są wymieniane jako najczęstsze skutki zbyt dużej ilości czasu spędzanego przed ekranem. Poza próbą jego ograniczenia, jedną z częstszych porad jest próba zmniejszenia kontrastu pomiędzy ekranem a jego otoczeniem.

[1] Przenośne urządzenia łączące w sobie zalety telefonów komórkowych oraz przenośnych komputerów (z ang. smartphone).

[2] Badania zostały przeprowadzone z uwzględnienem osób powyżej 4 roku życia w okresie od stycznia do czerwca 2015 roku [1].

[3] Wideo na życzenie (z ang. video on demand).

Do celów niniejszej pracy należy złożenie i oprogramowanie systemu oświetlenia, który ma rozszerzać obraz widziany na ekranie na jego otoczenie. Poza zmniejszeniem kontrastu, a więc aspektem zdrowotnym, system ma także na celu zwiększyć wrażenia wizualne dostarczane przez oglądany obraz.

System oświetlenia składa się z taśmy diod elektroluminescencyjnych[4] podłączonych do mikrokontrolera Arduino Uno, który z kolei ma współpracować z komputerem z zainstalowanym systemem operacyjnym MacOS. Oprogramowanie mikrokontrolera, którego zadaniem jest sterowanie diodami zostało przygotowane w języku Arduino, natomiast aplikacja kontrolująca cały system przeznaczona na komputer z systemem MacOS zostałą napisana w języku Swift. Wybór języków jest ściśle związany z koniecznością uzyskania jak najlepszej wydajności oraz użyciem najnowszych technologii.

Podobne systemy oświetlenia dostępne są już od pewnego czasu na rynku, jednak to właśnie nowoczesne technologie, prostota wykonania i niskie koszta powinny uczynić z Ligtning, bo taką nazwę otrzymał projekt, pełnowartościowego konkurenta.

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[5]:

- Oszczędność – znacznie zredukowana zostaje ilość pracy wymagana na oprogramowanie dostępu do bazy danych.

[4] LED (z ang. light-emitting diode).

[5] mapowanie obiektowo-relacyjne (ang. Object-Relational Mapping)

- Uniezależnienie się od rodzaju systemu zarządzania bazą danych – możliwość korzystania z wielu rozdzajów baz danych, a także możliwość jego zmiany w dowolnym momencie bez ponoszenia większych strat czasowych.
- Nieobowiązkowa znajomość języka SQL[6] – w celu tworzenia zapytań do bazy wykorzystywany jest udostępniany interfejs.
- Liczne funkcjonalności – aby skorzystać z transakcji, połączenia z bazą danych a także wieu innych funkcjonalności baz danych wystarczy zazwyczaj wywołać pojedyńczą metodę.
- Model danych przechowywany w jednym miejscu – dzięki czemu łatwiej jest zarządzać kodem.

W parze z przedstawionymi zaletami pojawiają się także pewne wady:

- Konfiguracja jest najczęściej skomplikowana i wymaga sporo czasu.
- Aby efektownie korzystać z narzędzi do mapowania obiektowo-relacyjnego wymagana jest 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.

Głównym celem autorów Qubica jest sprawienie aby stał się on dobrą alternatywą dla nielicznych, ale istniejących już narzędzi realizujących mapowanie obiektowo-relacyjne w języku C++. Wszystkie znane rozwiązania są dostępne za darmo, jednak albo nie udostępniają one generatora opisu będącego w stanie wygenerować cały projekt aplikacji albo 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.

[6] strukturalny język zapytań (ang. Structured Query Language)

1.3 Cele pracy

Do najważniejszych celów niniejszej pracy dyplomowej należą:

- Analiza istniejących bibliotek programistycznych 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 wykorzystanie najciekawszych pomysłów w Qubicu 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 Qubica z wcześniej analizowanymi narzędziami – porównanie to pozwoli stwierdzić czy przyjęte założenia i zastosowane rozwiązania okazały się słuszne oraz czy Qubic wnosi coś nowego do grona istniejących rozwiązań.

Do celów części praktycznej należą:

- Stworzenie intuicyjnego interfejsu użytkownika – im mniej linii kodu musi zostać napisane w celu wykonania podstawowych operacji bazodanowych tym interfejs jest uważany za bardziej intuicyjny.
- Stworzenie generatora opisu mapowania obiektowo-relacyjnego – jest to celem pracy Sebastiana Florka. Wspólnym celem obu autorów jest integracja utworzonych modułów.
- Poprawne zrealizowanie założeń mapowania obiektowo-relacyjnego, a także jak najlepsza optymalizacja zapytań – mapowanie musi odbywać się możliwie szybko, jednak najważniejsze jest zrealizowanie wszystkich jego założeń.
- Uczynienie konfiguracji Qubica jak najprostszą – najlepszym rozwiązaniem wydaje się być przeniesienie całej konfiguracji do pliku konfiguracyjnego, tak aby nie musiała ona zalegać w tworzonym kodzie.

1.4 Metoda badawcza

1.4.1 Studia literaturowe

Literaturę wykorzystaną podczas pisania niniejszej pracy można podzielić na trzy kategorie:

- Literatura dotycząca języka C++ oraz aplikacji szkieletowej Qt.
- Literatura dotycząca relacyjnych baz danych oraz języka SQL.
- Literatura dotycząca mapowania obiektowo-relacyjnego.

Na temat dwóch pierwszych kategorii powstało wiele książek oraz artykułów naukowych, ponadto ogromną ilość informacji można znaleźć w różnego rodzaju źródłach elektronicznych. W przypadku ostatniej kategorii wybór ten jest mniejszy, jednakże nadal można znaleźć na ten temat sporo informacji, szczególnie w języku angielskim. Kolejny podrozdział przedstawia najważniejsze pozycje w kazdej z wymienionych powyżej kategorii.

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 można się najbardziej z wybranym 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 które przyjęli autorzy istniejących już rozwiązań.

1.5 Przegląd literatury w dziedzinie

1.5.1 Literatura dotycząca języka C++ oraz Qt

W celu zasięgnięcia informacji na temat programowania w języku C++ i zagadnień z nim związanych najczęściej wykorzystywaną pozycją książkową była „Symfonia” [**?**] Jerzego Grębosza. 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 wartościowa okazała się też specyfikacja 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 jak i do wszystkich innych wykorzystanych źródeł najłatwiej znaleźć z bibliografii, znadującej się na ostatnich stronach pracy.

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.

Praca rozpoczyna się od uzasadnienia wyboru tematu pracy, a także opisu celów pracy, jej problematyki, zakresu, wykorzystanych metod badawczych, przeglądu literatury oraz jej układu.

W kolejnym rozdziale zawarte zostały objaśnienia zagadnień teoretycznych dotyczących relacyjnych baz danych, programowania obiektowego oraz przede wszystkim z mapowania obiektowo-relacyjnego. Można w nim znaleźć informacje na tematy związane także z architekturą aplikacji, trwałością danych w aplikacjach czy z językiem SQL.

Następnie autor przedstawia analizę istniejących już narzędzi służących do mapowania obiektowo-relacyjnego napisanych w języku C++ biorąc pod uwagę wcześniej wymienione kryteria analizy.

Czwarty rozdział przedstawia projekt aplikacji szkieletowej Qubic tworzonej w ramach części badawczej niniejszej pracy, a w tym między innymi postawione wymagania, opis implementacji czy też porównanie z wcześniej analizowanymi narzędziami.

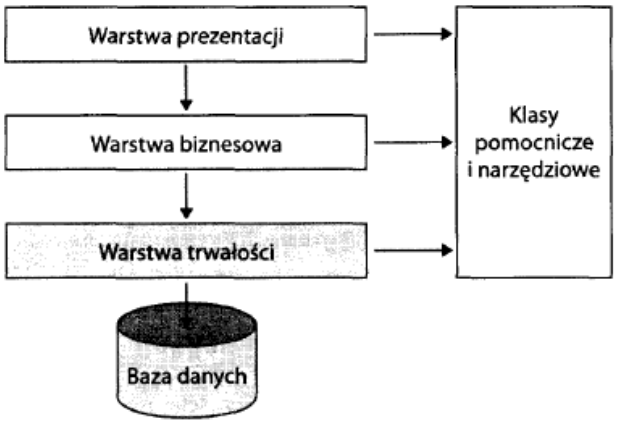
Praca kończy się podsumowaniem, które zawiera dyskusję wyników oraz dalsze prespektywy jej rozwoju.

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 biznesowej oraz trwałości danych [2].



Rysunek 2.1: Warstwowa architektura aplikacji [2]

Zadaniem poszczególnych warstw jest [2]:

- 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 informacji zapisanych w ściśle określony sposób w strukturach odpowiadających założonemu modelowi danych. Model relacyjny został operty na postulatach relacyjności, a jego twórcą jest Edgar Frank Codd. Dane w modelu relacyjnym reprezentowane są jako zbiór krotek, które w znormalizowanych bazach danych są unikatowe a ich kolejność nie gra roli. Dostęp do nich zapewnia algebra relacji wraz z operatorami takimi jak rzutowanie, selekcja, złączenie, suma, różnica czy produkt kartezjański. Zbiory danych powiązane są logicznie za pomocą encji, w ten sposób uniezależnia się widziany przez użytkownika obraz bazy danych od jej postaci fizycznej [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 sporą część pracy, która wcześniej musiała zostać wykonana przez programistę [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” [2] 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 `Adress`, może zostać osadzona w klasie bardziej znaczącej, na przykład `User`. W bazie danych szczegółowość może zostać zaimplementowana jedynie na poziomie tabeli `USER` oraz kolumny `ADRESS`. Jest to jednak problem bardzo łatwy do rozwiązania za pomocą tabel powiązanych ze sobą relacjami.

- 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 – w modelu obiektowym asocjacje reprezentują związki między klasami, na przykład pomiędzy klasami `User` i `Adress`. Języki obiektowe używają w tym celu referencji czy też wskaźników, stąd też łatwo zauważyć, że asocjacje w modelu obiektowym są kierunkowe lub też dwukierunkowe jeśli obie klasy mają do siebie referencje. W przypadku baz danych nie istnieją asocjacje dwukierunkowe, wyróżniamy jedynie relacje jeden do wielu oraz jeden do jednego. Relacja wiele do wielu jest tak na prawdę złożeniem dwóch relacji jeden do wielu z dodatkową tabelą łącznikową, nazywaną też asosjacyjną.

- Problem nawizgacji po grafie obiektów – sposób dostępu do obiektów znacznie różni się od dosępu do danych w relacyjnych bazach. Chcąc na przykład uzyskać informacje o ulicy na jakiej mieszka wybrany użytkownik systemu wywołamy metodę `user.getAdress().getStreet()`, wywołanie takie nazywane jest chodzeniem po grafie obiektu. Przedstawiony sposób niestety nie sprawdza się w przypadku nawigacji w relacyjnych bazach danych. Rozwiązaniem jest minimalizacja ilości zapytań kierowanych do bazy danych, a więc wykorzystanie mechanizmów złączeń jednakże wydajność nigdy nie będzie taka sama jak w przypadku modelu obiektowego.

2.7 Koszt niedopasowania

Rozwązanie wymienionych wcześniej problemów dotyczących niedopasowania paradygmatów może okazać się bardzo czasochłonne. Zazwyczaj około 30% kodu aplikacji to kod warstwy trwałości danych oraz ręczne generowanie zapytań w celu przechowania danych w bazie. Pomimo tak dużych nakładów pracy, często okazuje się, że wymagane są liczne poprawki, ponieważ przyjęte rozwiązania okazały się nietrafne.

Największy koszt jest związany z fazą modelowania i trudnościami z dopasowaniem dobrze znanego programistom modelu obiektowego do niekiedy trudniejszego do zrozumienia modelu relacyjnego. Problem ten udaje się często rozwiązać kosztem zalet modelu obiektowego takich jak niewykorzystanie dziedziczenia czy polimorfizmu. Problem niedopasowania modeli prowadzi do niskej elastyczności i

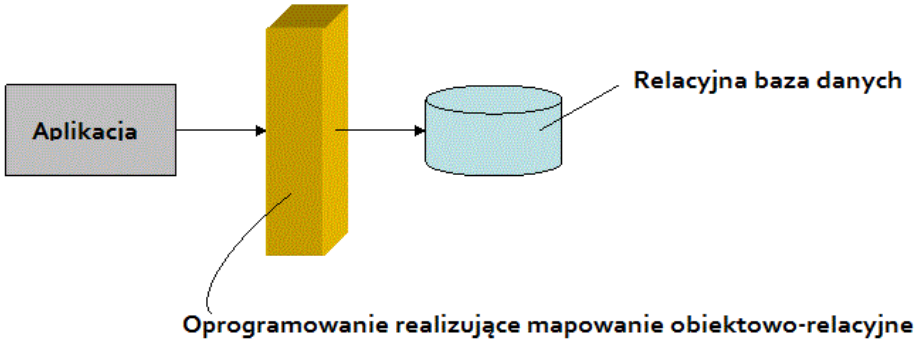
utraty produktywności tworzonych aplikacji, co prowadzi w ostateczności ku wyższym kosztom tworzenia oprogramowania [2].

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].*



Rysunek 2.2: Mapowanie obiektowo-relacyjne

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.

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

- 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]:

- 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.

2.9 Aplikacje szkieletowe

Na wstępie aplikacje szkieletowe zostały opisane jako „fundamenty” dla tworzonych przez programistów aplikacji, warto przypomnieć tutaj dokładniejszą definicję

tego pojęcia:

Definicja 4 *Aplikacja szkieletowa*[2] *– szkielet budowy aplikacji. Definiuje strukturę aplikacji oraz jej ogólny mechanizm działania, dostarcza zestaw komponentów i bibliotek ogólnego przeznaczenia do wykonywania określonych zadań [11].*

Charakterystycznymi cechami aplikacji szkieletowych są między innymi [11]:

- Gotowy szkielet dla tworzonych aplikacji.
- Narzucony przepły sterowania.
- Domyślna konfiguracja.
- Komponenty do rozbudowy.

Do zalet aplikacji szkieletowych można zaliczyć [11]:

- Szybka realizacja projektu – część kodu jest gotowa, a więc oszczędzamy sporo czasu potrzebnego na jego napisanie.
- Poprawa jakości kodu oraz większa niezawodność – gotowy kod został wielokrotnie przetestowany oraz był używany przez wielu innych programistów, a więc jeśli coś nie działa to w większości przypadków wina błędu użytkownika.
- Wsparcie twórców frameworka – jeśli jednak uda nam się znaleźć jakiś błąd, zawsze możemy liczyć na wsparcie ze strony twórców oprogramowania.

Do ich wad natomiast zaliczamy [11]:

- Złożoność – zakres działań aplikacji szkieletowych jest bardzo szeroki, a co za tym idzie ich sposób działania nie jest zawsze oczywisty.
- Koszt szkoleń – aby poradzić sobie ze złożonością, najlepszym wyjściem jest poświęcenie dodatkowego czasu na szkolenie w zakresie wybranego oprogramowania.
- Zredukowana wydajność – w bardziej skomplikowanych systemach wykorzystanie aplikacji szkieletowych może mieć wspłw na spadek wydajności systemu, dzieje się tak w związku z niedopasowaniem wybranej aplikacji do systemu.

[2] ang. Framework

Rozdział 3

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

3.1 Kryteria analizy

W pierwszym rozdziale niniejszej pracy pojawiła się informacja, że istnieją już biblioteki oraz aplikacje szkieletowe realizujące mapowanie obiektowo-relacyjne w języku C++, jednakże jest ich mniej niż w przypadku innych znanych języków programowania. W tym rozdziale zostanie przeprowadzona ich dokładniejsza analiza, do której potrzebne jest wskazanie kryteriów, które zostaną przeanalizowane. Pod uwagę zostałe wzięte następujące aspekty:

- Charakter oprogramowania – przede wszystkim istotny jest fakt, czy wybrane narzędzie należy do tak zwanego „wolnego oprogramowania”[1], a co za tym idzie czy jego użytkownik może podejrzeć i w razie potrzeby zmodyfikować jego kod źródłowy oraz podzielić się wprowadzonymi przez siebie zmianami z innymi jego użytkownikami. W tensposób oprogramowanie ulega ciągłemu rozwojowi, a jego działanie łatwiej zrozumieć.
- Licencja – wielce istotny jest fakt czy wybrane oprogramowanie jest darmowe oraz na jakich zasadach może zostać wykorzystywane. Mało restrykcyjna licencja jest z pewnością zaletą.
- Wspierane systemy zarządzania bazą danych – tutaj zasada jest prosta, im więcej rodzajów baz danych jest wspieranych tym lepiej. Dużą zaletą jest

[1] ang. open-source software

także rozszerzalność orpgoramowania, czyli możliwość dodania wsparcia dla wybranych baz samodzielnie.

- Wykorzystywane biblioteki i aplikacje szkieletowe – część narzędzi wykorzystuje inne, takie jak aplikacja szkieletowa Qt czy bibliotekę boost. Istotne jest to czy użytkownik też musi wykorzystywać dane rozwiązania w swojej aplikacji.
- Poziom skomplikowania interfejsu – konieczność wywoływania sekwencji wielu metod w celu wykonania podstawowych operacji w bazie danych czy też w celu połączenia się z nią nie należy z pewnością do zalet oprogramowania.
- Dostępność generatora opisu – generator opisu jest w stanie znacznie uprościć wykorzystanie narzędzi mapowania obiektowo-relacyjnego, więc jego obecność jest dodatkową zaletą.
- Dodatkowe możliwości – wiele narzędzi posiada specyficzne dla siebie zalety, w tym punkcie zostaną one wyszczególnione.

3.2 Porównanie istniejących rozwiązań

3.2.1 Biblioteka QxOrm

Analizę rozpoczyna najpopularniejsza biblioteka realizujaca mapowanie obiektowo-relacyjne w języku C++, czyli QxOrm.

Rysunek 3.1: Biblioteka QxOrm [13]

Analizę wyznaczonych kryteriów przedstawia poniższa tabela:

Charakter oprogramowania	open-source
Licencja	GNU/GPL
Wspierane bazy danych	MySQL, PostgreSQL, Oracle,
	SQLite, IBM DB2, InterBase, SysBase
Wykorzystywane narzędzia	Qt, boost
Poziom skomplikowania interfejsu	wysoki
Dostępność generatora opisu	tak
Dodatkowe możliwości	edytor encji, walidacja danych,
	serializacja danych, pamięć cache

Tablica 3.1: Analiza biblioteki QxOrm

Konfiguracja QxOrm jest dość rozbudowana, a co za tym idzie skomplikowana. Rejestracja klas wymaga używanie wielu funkcji i makr w pliku nagłówkowym [13]:

```
#ifndef _CLASS_DRUG_H_
#define _CLASS_DRUG_H_

class drug
{
public:
   long id;
   QString name;
   QString description;
   drug() : id(0) { ; }
   virtual ~drug() { ; }
};

QX_REGISTER_HPP_MY_TEST_EXE(drug, qx::trait::
   no_base_class_defined, 1)

#endif // _CLASS_DRUG_H_
```

Listing 3.1: Przykładowy plik nagłówkowy klasy korzystającej z QxOrm

A także w pliku klasy [13]:

```
#include "precompiled.h"
#include "drug.h"
#include <QxMemLeak.h>

QX_REGISTER_CPP_MY_TEST_EXE(drug)

namespace qx {
template <> void register_class(QxClass<drug> & t)
{
  t.id(& drug::id, "id");
  t.data(& drug::name, "name", 1);
  t.data(& drug::description, "desc");
}}
```

Listing 3.2: Przykładowy plik klasy korzystającej z QxOrm

Samo wykorzystanie QxOrm także nie należy do najłatwiejszych zadań, istotne jest poprawne skonfigurowanie bazy danych z poziomu kodu, oraz wykonanie odpowiednich sekwencji metod [13]:

```
#include "precompiled.h"
#include "drug.h"
#include <QxMemLeak.h>

int main(int argc, char * argv[])
{
   QApplication app(argc, argv);
   typedef boost::shared_ptr<drug> drug_ptr;
   drug_ptr d1; d1.reset(new drug()); d1->name = "name1"; d1->
      description = "desc1";
   drug_ptr d2; d2.reset(new drug()); d2->name = "name2"; d2->
      description = "desc2";
   drug_ptr d3; d3.reset(new drug()); d3->name = "name3"; d3->
      description = "desc3";
   typedef std::vector<drug_ptr> type_lst_drug;
   type_lst_drug lst_drug;
   lst_drug.push_back(d1);
   lst_drug.push_back(d2);
   lst_drug.push_back(d3);
   qx::QxSqlDatabase::getSingleton()->setDriverName("QSQLITE");
   qx::QxSqlDatabase::getSingleton()->setDatabaseName("./
```

```
        test_qxorm.db");
    qx::QxSqlDatabase::getSingleton()->setHostName("localhost");
    qx::QxSqlDatabase::getSingleton()->setUserName("root");
    qx::QxSqlDatabase::getSingleton()->setPassword("");
    QSqlError daoError = qx::dao::create_table<drug>();
    daoError = qx::dao::insert(lst_drug);
    d2->name = "name2 modified";
    d2->description = "desc2 modified";
    daoError = qx::dao::update(d2);
    daoError = qx::dao::delete_by_id(d1);
    long lDrugCount = qx::dao::count<drug>();
    drug_ptr d_tmp; d_tmp.reset(new drug());
    d_tmp->id = 3;
    daoError = qx::dao::fetch_by_id(d_tmp);
    qx::serialization::xml::to_file(lst_drug, "./export_drugs.
        xml");
    type_lst_drug lst_drug_tmp;
    qx::serialization::xml::from_file(lst_drug_tmp, "./
        export_drugs.xml");
    drug_ptr d_clone = qx::clone(* d1);
    boost::any d_any = qx::create("drug");
    qx::cache::set("drugs", lst_drug);
    qx::cache::clear();
    drug * pDummy = new drug();
    return 0;
}
```

Listing 3.3: Przykładowy plik klasy korzystającej z QxOrm

3.2.2 Biblioteka Debea

Kolejnym narzędziem poddanym analizie została biblioteka Debea:

Rysunek 3.2: Biblioteka Debea [15]

Analizę wyznaczonych kryteriów przedstawia poniższa tabela:

Konfiguracja Debei jest stosunkowo prosta, odbywa się z poziomu kodu. Wykonywanie podstawowych operacji nie należy jednak do najprostszych, oto przy-

Charakter oprogramowania	open-source
Licencja	wxWindows
Wspierane bazy danych	MySQL, IBM DB2, Oracle, MSSQL
Wykorzystywane narzędzia	brak
Poziom skomplikowania interfejsu	wysoki
Dostępność generatora opisu	tak
Dodatkowe możliwości	serializacja danych

Tablica 3.2: Analiza biblioteki Debea

kładowe połączenie z bazą danych i wykonanie na niej podstawowych operacji z użyciem Debei:

```
ar.open("dbasqlite3-static", "dbname=foobasefile.sqt3");
ar.getOStream().sendUpdate(counter_create);
ar.getOStream().sendUpdate(foo_create);
Foo c1;
c1.mIntVal = 12;
c1.mStrVal = "test string";
dba::SQLOStream ostream = ar.getOStream();
ostream.open();
ostream.put(&c1);
std::cout << "Foo c1 was stored with id = " << c1.getId() <<
   std::endl;
ostream.destroy();
Foo c2;
dba::SQLIStream istream = ar.getIStream();
istream.open(c2);
while (istream.getNext(&c2)) {
    std::cout << "readed Foo c2 with id: " << c2.getId() << std
       ::endl;
};
if ((c1.mIntVal == c2.mIntVal) && (c1.mStrVal == c2.mStrVal))
    std::cerr << "Foo is Foo!" << std::endl;
```

Listing 3.4: Przykładowy użycie Debei

Treść licencji bibliotek wxWindows można znaleźć pod adresem `http://open-source.org/licenses/wxwindows.php`.

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

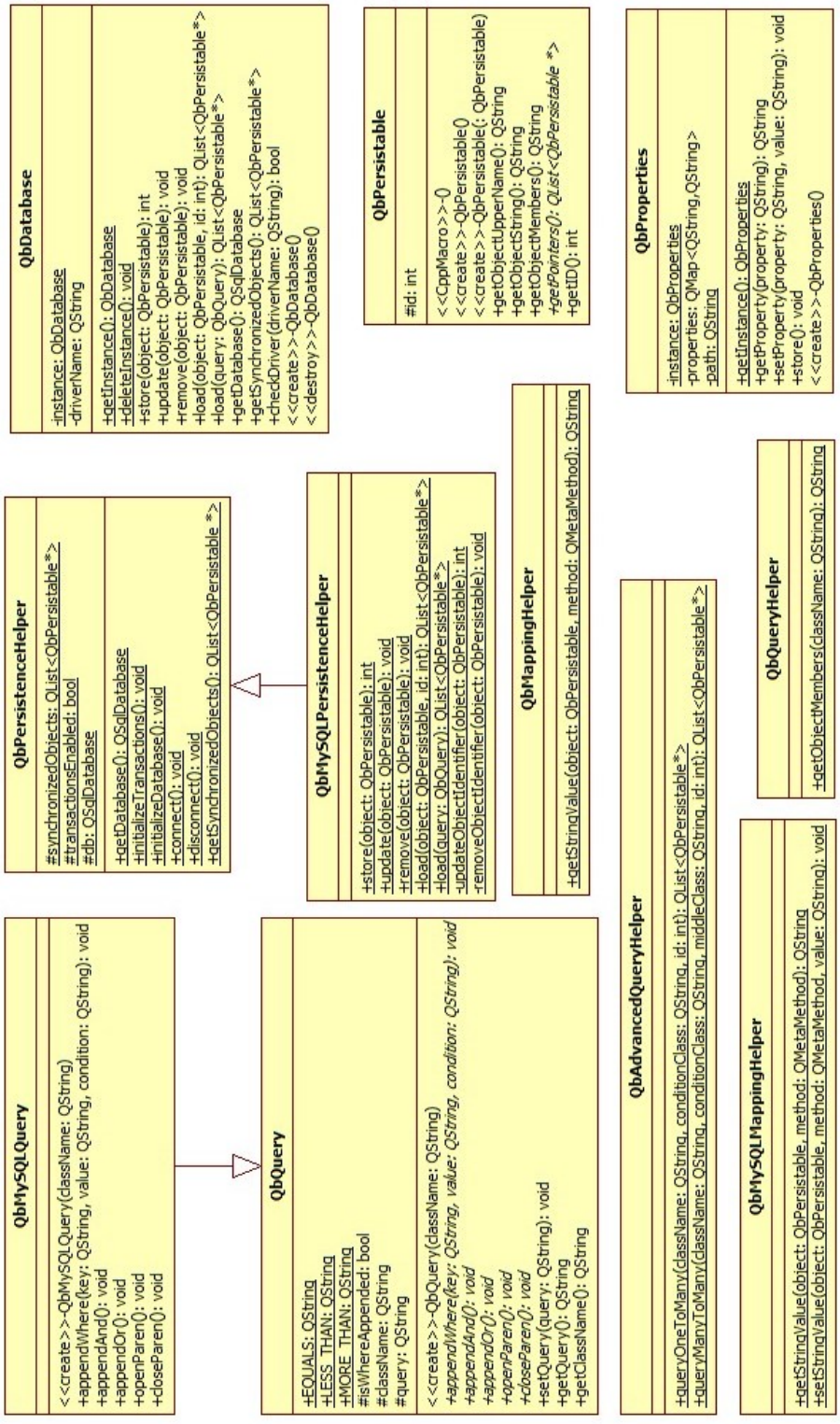
Rysunek 4.1: Logo tworzonej aplikacji szkieletowej

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 nieuwzględniający relacji zależności Qubica przedstawia rysunek 4.2:



Rysunek 4.2: 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.

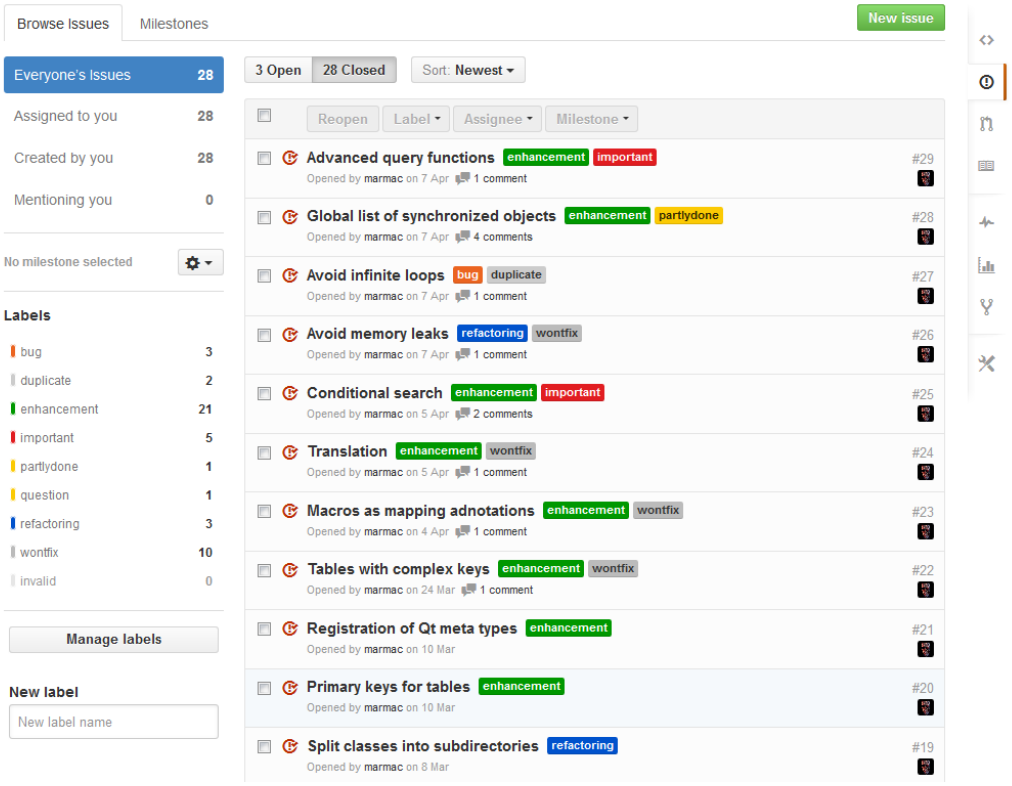
Jedyną biblioteką wykorzystywaną przez Qubica jest QsLog [14] odpowiedzialny za logowanie wydarzeń w konsoli Qt. Jest to bardzo przydatna funkcja, która umożliwia szybkie zlokalizowanie pojawiających się problemów.

4.3.5 System kontroli wersji

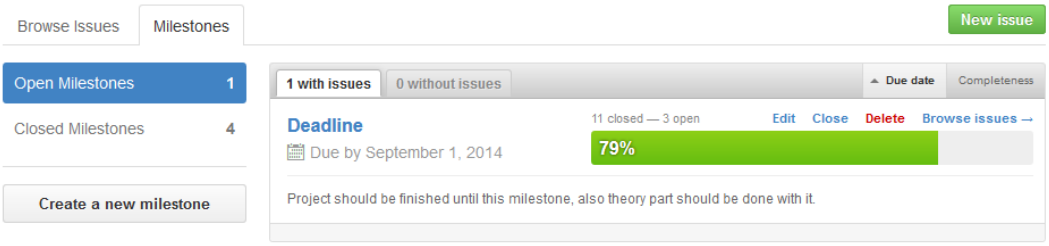
Podczas pracy nad projektami w których bierze udział więcej niż jedna osoba bardzo dobrym rozwiązaniem jest korzystanie z systemów kontroli wersji. Ułatwiają one wspólną pracę oraz organizację tworzonych projektów.

Wybór autorów Qubica padł na dobrze znany serwis GitHub [12]. Poza repozytorium na którym znaleźć można kod źródłowy Qubica podczas implementacji projektu wykorzystany został system Issues and Milestones[1], który ułatwia organizację pracy nad projektem.

[1] problemy oraz kamienie miliowe



Rysunek 4.3: Przejrzysty widok systemu GitHub Issues



Rysunek 4.4: Przejrzysty widok systemu GitHub Miletsones

4.4 Implementacja

W niniejszej sekcji zostaną opisane kluczowe elementy, które zostały zaimplementowane w Qubicu, a także problemy jakie pojawiły się podczas implementacji i zastosowane rozwiązania.

4.4.1 Połączenie z bazą danych

Implementacja dużej aplikacji szkieletowej operującej w większości na bazie danych nie mogła zacząć się od niczego innego niż obsługa połączenia z bazą danych. Jest to element kluczowy dla narzędzi realizujących mapowanie relacyjno-obiektowe i nie inaczej jest w przypadku Qubica. W związku z tym, że w innych analizowanych narzędziach można było się napotkać na uciążliwą konfigurację odbywającą się głównie z poziomu kodu Qubic korzysta z pliku konfiguracyjnego, który przechowywuje całą jego konfigurację. W celu zaimplementowania mechanizmów takich jak połączenie się z bazą czy operacje wejścia i wyjścia wykorzystane zostały zestawy funkcji udostępniane przez aplikację szkieletową Qt oraz sterowniki dla wybranych systemów zarządznia bazą danych. Warto wspomnieć, że sama obsługa połączenia jest niewidoczna dla użytkownika, ponieważ połączenie nawiązywane jest podczas uzyskania dostępu do instancji `Singleton`.

4.4.2 Interfejs CRUD

Innym 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. Nie jest to jednak jedyne możliwe rozwiązanie i w Qubicu problem ten został rozwiązany w trochę inny sposób, który umożliwia stworzony generator opisu mapowania obiektowo-relacyjnego. Generowane pliki klas oraz pliki nagłówkowe 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.

Wspomniany mechanizm refleksji umożliwia konstruowanie obiektów czy też wywoływanie metod dynamicznie – to znaczy w trakcie działania programu. Autor niniejszej pracy nie był przecież w stanie określić dokładnie jakie obiekty i klasy

będą obsługiwane w trakcie działania Qubica. Sama refleksja jest także jednym z powodów dla którego Qubic działa w oparciu o aplikację szkieletową Qt, bo jak dobrze wiadomo sam język C++ w przeciwieństwie do chociażby języka Java jej nie udostępnia.

Aby metody wybranych klas były dostępne za pomocą refleksji Qt wymaga zarejestrowania ich specjalnie do tego przeznaczonym makrem `Q_INVOKABLE` natomiast cała klasa powinna dziedziczyć po klasie `QObject`, w przypadku Qubica drugie wymaganie zostaje spełnione przez dziedziczenie po wcześniej wspominanej klasie `QbPersistable`, natomiast makra zostają dodane na etapie generowania kodu poprzez generator opisu mapowania. Dodatkowa rola klasy `QbPersistable` polega na zagwarantowaniu zestawu pewnych metod w każdej klasie mapowanej, czyli na przykład metody pobierającej klucz główny danego obiektu czy też metody odpowiedzialnej za zwrócenie rzeczywistej i dokładnej nazwy klasy.

W celu zagwarantowania użytkownikowi lepszej kontroli nad tworzoną aplikacją utworzona została także lista aktualnie zsynchronizowanych obiektów, która jest na bieżąco aktualizowana przez metody Qubica. Skorzystanie z niej jest najlepszym sposobem na szybki dostęp do aktualnie zsynchronizowanych obiektów, a udostępniana jest ona podobnie jak interfejs CRUD przez klasę `QbDatabase`.

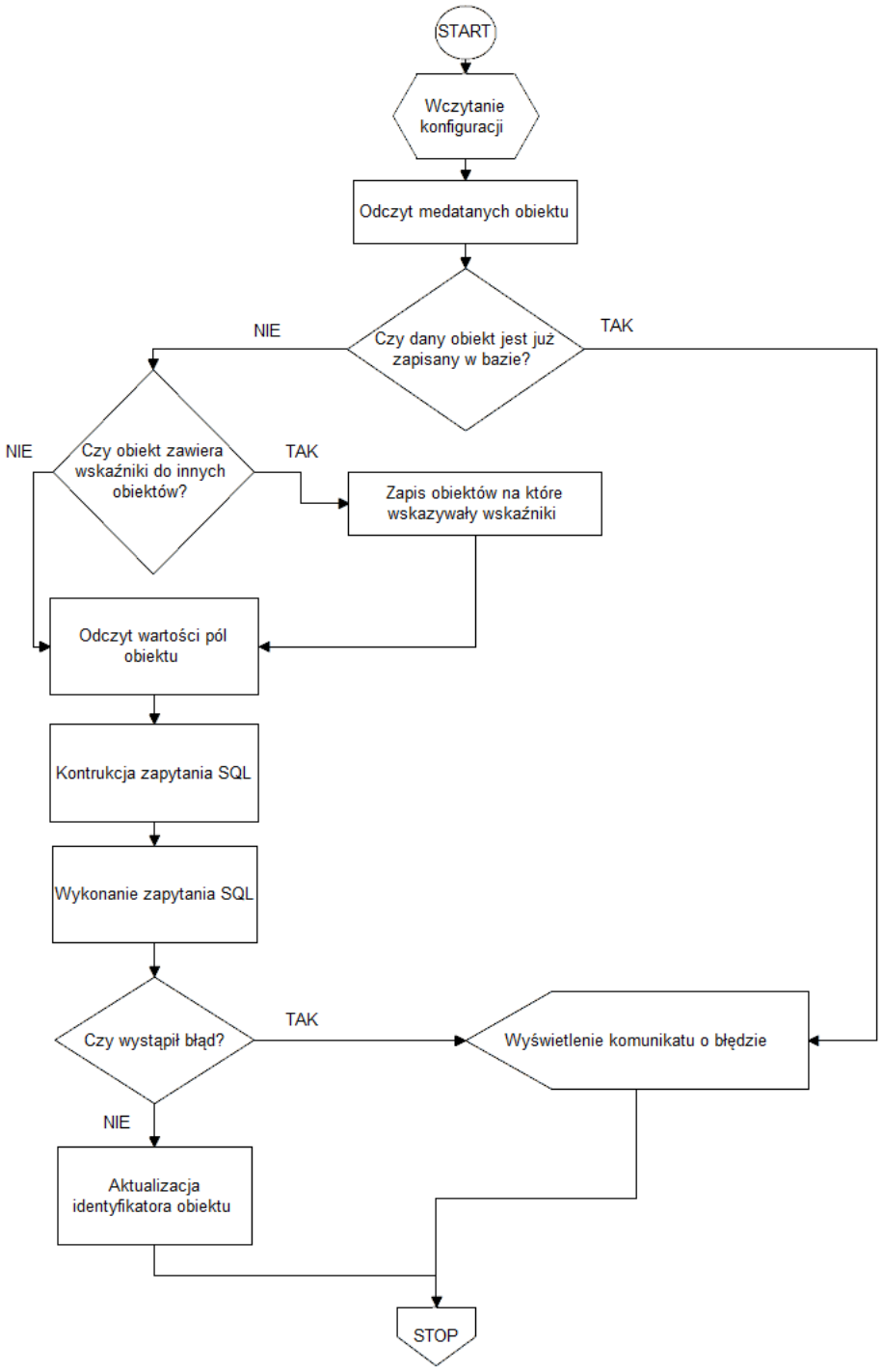
Schematy działania czterech podstawowych metod dostępu przedstawiają schematy blokowe umieszczone na rysunkach 4.5 – 4.8.

Pierwszy ze schematów przedstawia sposób działania metody zapisującej obiekty w bazie danych. Warte uwagi jest w tym przypadku rozwiązanie problemu z wskaźnikami do innychobiektów, które w bazie danych muszą zostać zamienione w odwołania do rekordów w innych tabelach. Aby było to możliwe w trakcie zapisu dowolnego obiektu sprawdzane jest czy posiada on odwołania do innych obiektów i jeśli tak jest pierwsze w bazie zostaną zapisane obiekty podrzędne a dopiero później obiekty, które posiadają do nich odwołania. Ten schemat postępowania został powielony także w przypadku metody aktualizującej obiekty w bazie danych.

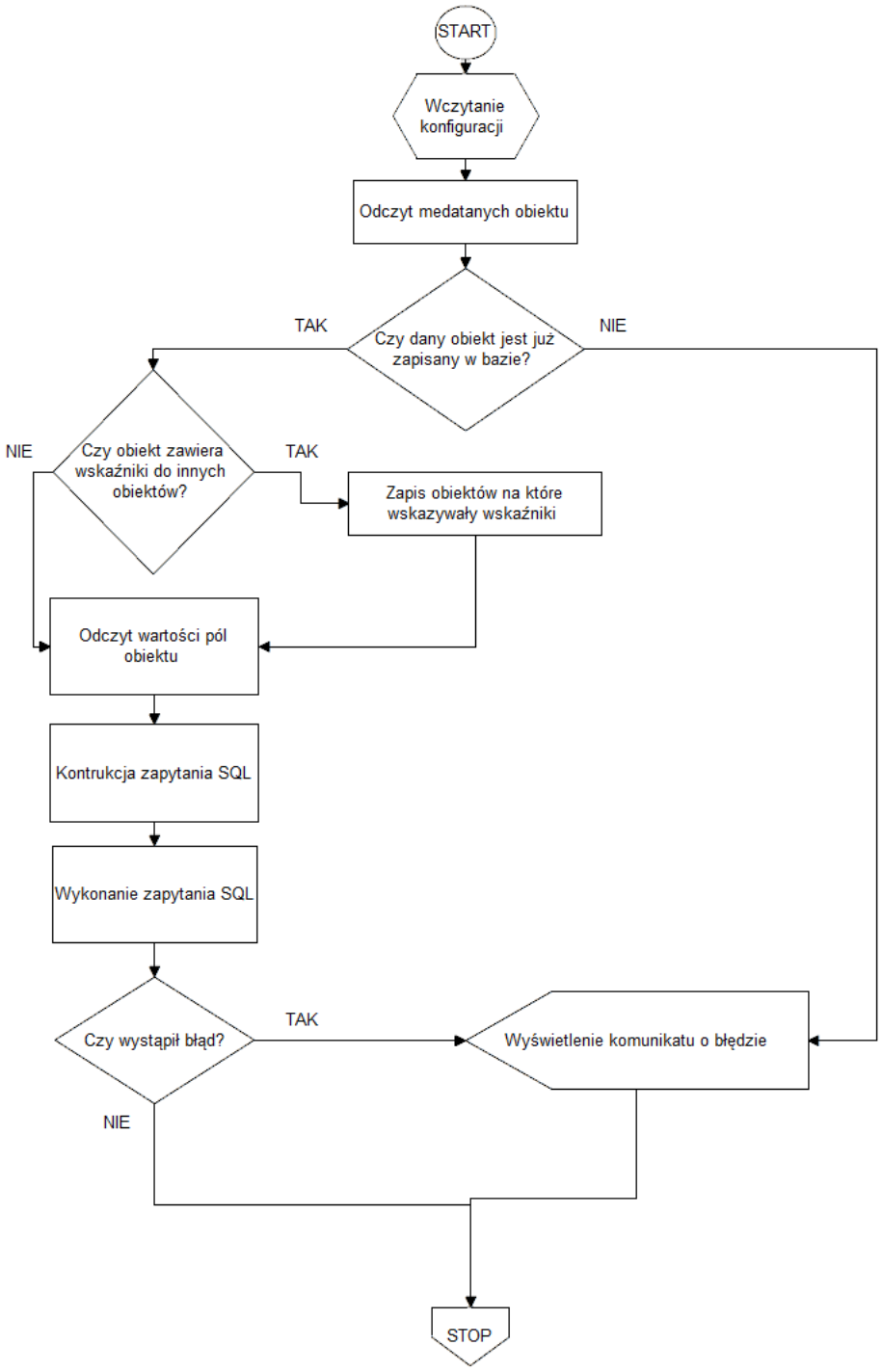
Problem dotyczący odwołań pomiędzy rekordami pojawia się także w trakcie ich ładowania z bazy danych. Tym razem po stworzeniu obiektów powstaje konieczność przekształcenia odwołań do rekordów w innych tabelach w wskaźniki do obiektów utworzonych na ich podstawie. Wywołując metodę `getCompany()` klasy `Employee` liczymy na to, że zwróci nam ona wskaźnik do obiektu klasy `Company` a nie klucz główny z tabeli `COMPANY` w bazie. Transformacja ta ma miejsce w ostatnim momencie konstruowania obiektów, gdy wszystkie inne pola zostały już wypełnione. Wczytywane są wtedy obiekty do których prowadzą odwołania a obiekty w których te odwołania występują uzupełniane są o wskaźniki do nich.

Ostatnia z metod należących do interfejsu CRUD, której zadaniem jest usuwanie obiektów z bazy nie musi dodatkowo dbać o relacje pomiędzy obiektami. Zada-

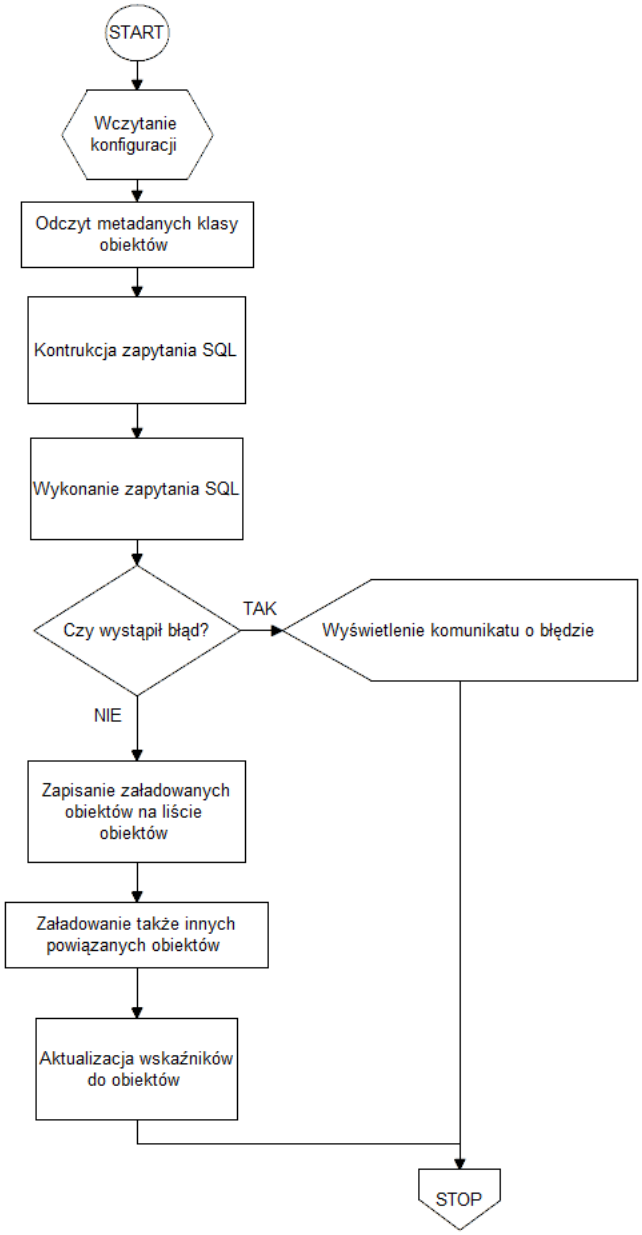
nie to zostaje powierzone bazie danych, która zależnie od preferencji użytkownika będzie usuwała rekordy kaskadowo lub też nie zezwoli na ich usunięcie ze względu na istniejące do nich odwołania.



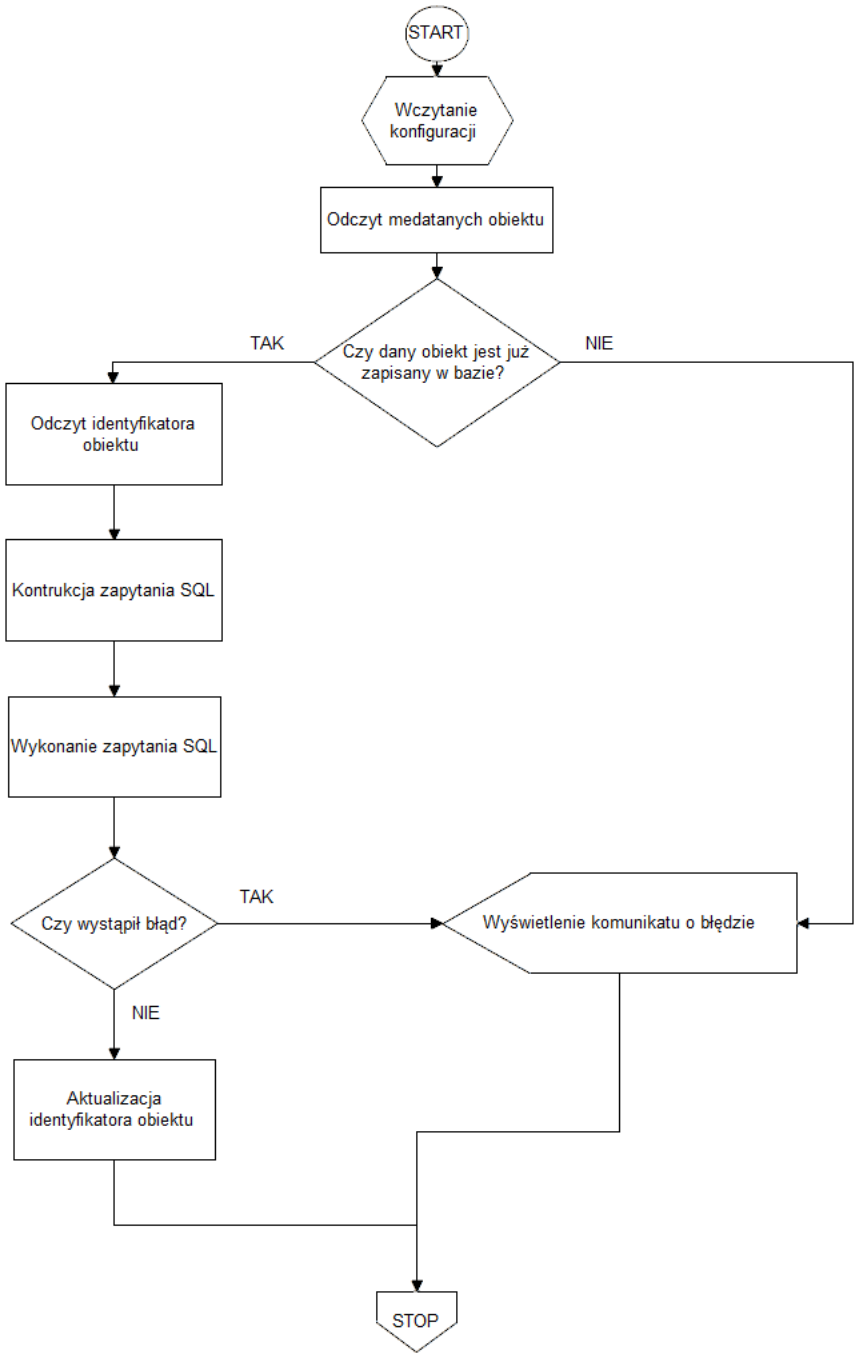
Rysunek 4.5: Schemat blokowy metody zapisującej obiekty w bazie danych



Rysunek 4.6: Schemat blokowy metody aktualizującej obiekty w bazie danych



Rysunek 4.7: Schemat blokowy metody ładującej obiekty z bazy danych



Rysunek 4.8: Schemat blokowy metody usuwającej obiekty z bazy danych

4.4.3 Interfejs do tworzenia zapytań

Operacja ładowania danych udostępniana przez interfejs opisany w poprzedniej sekcji jest w stanie załadować wszystkie dane z wybranej tabeli. Jest to jednak rozwiązanie mało efektywne, szczególnie biorąc pod uwagę systemy przetwarzające duże ilości informacji. Nawet gdy użytkownik przefiltruje odpowiednio dla siebie wyniki ładowania i tak straci dużo czasu na wykonanie tych operacji. Najlepszym sposobem aby usprawnić proces ładowania danych jest skorzystanie z odpowiednio skonstruowanych zapytań SQL. Chcąc podtrzymać ideę mapowania obiektowo-relacyjnego, w której użytkownik nie jest zmuszony do dokładnego poznania wybranego dialektu języka SQL autor niniejszej pracy zdecydował się stworzyć uogólniony interfejs za którego pomocą użytkownicy Qubica mogą tworzyć zapytania SQL bez większej wiedzy na ich temat. Sam sposób tworzenia obiektów zapytań i z nich korzystania został opisany w podręczniku użytkownika.

Sam sposób działania klas tworzących zapytania nie należy do skomplikowanych. Kluczowa jest tutaj klasa wirtualna `QbQuery`, która sama implementuje część metod używanych do tworzenia zapytań – dokładniej są to metody, których ciało nie zmienia się w zależności od wybranego dialektu języka SQL, a także wymusza na klasach dziedziczących zaimplementowanie pozostałych potrzebnych metod. W ten sposób implementując klasę `QbMySQLQuery` zaimplementowana musiała zostać tylko część metod a reszta była już dostępna. Wszystkie tworzone metody operują głównie na łaćuchach znaków stopniowo tworząc obiekty zapytań, które później mogą zostać przekazane do metody `load(query)` w klasie `QbDatabase`.

4.4.4 Metody dostępu do powiązanych danych

Charakterystycznymi elementami relacyjnych baz danych są relacje jeden do wielu oraz wiele do wielu, składające się tak na prawdę z tabeli łącznikowej oraz dwóch relacji jeden do wielu. Dużym usprawnieniem dla aplikacji jest możliwość uzyskania obiektów powiązanych z wybranym obiektem, Qubic dzięki obecności generatora kodu udostępnia taką możliwość.

Weźmy pod uwagę dwie klasy obiektów `Employee` oraz `Company`. W modelu relacyjnym tabela `EMPLOYEE` zawiera odwołanie do tabeli `COMPANY` dzięki czemu posiadamy informacje dla jakiej firmy pracuje wybrany pracownik, w modelu obiektowym zrealizowane to jest za pomocą referencji lub też wskaźników. Co jednak jeśli potrzebujemy informacji o wszystkich pracownikach danej firmy? Ani obiekt ani encja w tabeli nie posiada takich informacji, na pomoc przychodzą wygenerowane przez generator opisu tzw. metody dostępu do powiązanych danych. W przedstawionym przypadku jest to metoda `getEmployees()`, którą możemy wy-

wołać na dowolnym obiekcie klasy `Company`.

Przyjrzyjmy się w jaki sposób odbywa się dostęp do powiązanych danych po wygenerowaniu metod a także jak wyglądają ciała takich metod. Wróćmy do przedstawionego wcześniej przykładu, tzn. do relacji jeden do wielu pomiędzy tabelami `EMPLOYEE` oraz `COMPANY`. Generator opisu po wykryciu wspomnianej relacji w kodzie źródłowym klasy `Company` umieści następującą metodę:

```
QList<QbPersistable*> Company::getEmployees() {
    return QbAdvancedQueryHelper::queryOneToMany(Employee::
        CLASSNAME, CLASSNAME, id);
}
```

Listing 4.1: Ciało metody dostępu do powiązanych danych (jeden do wielu)

Jak widać zadaniem generatora jest jedynie wywołanie metody `queryOneToMany(className, conditionClass, id)`, gdzie argument `className` to nazwa klasy docelowej – w tym przypadku `EMPLOYEE`, `conditionClass` to nazwa klasy obecnej – `COMPANY`, a `id` to unikalny identyfikator wybranego obiektu klasy `Company`. Na podstawie tych trzech argumentów tworzone jest zapytanie z wykorzystaniem wcześniej stworzonego mechanizmu zapytań (poniższy przykład zawiera fragment kodu używanego w przypadku wykorzystania języka MySQL):

```
QbQuery* query = new QbMySQLQuery(className);
query->appendWhere(conditionClass, QString::number(id), QbQuery
    ::EQUALS);
return QbDatabase::getInstance()->load(query);
```

Listing 4.2: Zapytanie dostępu do powiązanych danych (jeden do wielu)

Poza relacjami jeden do wielu Qubic udostępnia dostęp do powiązanych danych także w przypadku relacji wiele do wielu. Tutaj rozważmy przypadek trzech tabel, tzn. `EMPLOYEE`, `DEPARTMENT` oraz `ASSIGNMENT` występującej w postaci tabeli łącznikowej. W przedstawionym przypadku zostaną wygenerowane dwie metody, jedna w klasie `Employee` a druga w `Department`. Przyjrzyjmy się jednej z tych metod, czyli `getEmployees()` w klasie `Department`:

```
QList<QbPersistable*> Department::getEmployees() {
    return QbAdvancedQueryHelper::queryManyToMany(Employee::
        CLASSNAME, CLASSNAME, Assignment::CLASSNAME, id);
}
```

Listing 4.3: Ciało metody dostępu do powiązanych danych (wiele do wielu)

W tym przypadku wygenerowany kod musi zawierać wywołanie metody `queryManyToMany(className, conditionClass, middleClass, id)`, gdzie jedynym dodatkowym argumentem w stosunku do relacji jeden do wielu jest łańcuch znaków `middleName`, który musi zawierać nazwę tabeli łącznikowej. Samo ciało metody dla języka MySQL wygląda następująco:

```
QString tableIdentifier = QbProperties::getInstance()->
    getProperty("qubic.configuration.table.identifier").toUpper
    ();
        QString queryString = "SELECT * FROM " + className.
            toUpper() + " WHERE " ;
        queryString += className.toUpper() + "." +
            tableIdentifier + " IN (SELECT ";
        queryString += middleClass.toUpper() + "." + className.
            toUpper() + " FROM " + middleClass.toUpper();
        queryString += " WHERE " + middleClass.toUpper() + "."
            + conditionClass.toUpper() + "='" + QString::number(
            id) + "')";
        QbQuery* query = new QbMySQLQuery(className);
        query->setQuery(queryString);
        return QbDatabase::getInstance()->load(query);
```

Listing 4.4: Zapytanie dostępu do powiązanych danych (wiele do wielu)

Jak widać w powyższym fragmencie kodu tym razem nie został wykorzystany mechanim zapytań Qubica, który w chwili obecnej nie obsługuje zagnieżdżonych w sobie zapytań, a które były konieczne do otrzymania prawidłowego rezultatu.

4.5 Podręcznik użytkownika

Jednym z głównych celów postawionych przed tworzoną aplikacją szkieletową było uczynienie interfejsu użytkownika jak najbardziej przejrzystym, tak aby tworzenie rozbudowanej instukcji dla użytkowników nie było koniecznością.

Zamiast wspomnianej instrukcji przyjrzyjmy się sposobom realizacji podstawowych operacji za pomocą Qubica oraz odpowiedziom na najczęstsze pytania użytkowników narzędzi mapowania obiektowo-relacyjnego w kontekście tworzonej aplikacji szkieletowej.

Jak wygląda konfiguracja Qubica?

Konfiguracja Qubica została całkowicie przeniesiona do pliku konfiguracyjnego o nazwie `qb.properties`. Za jego pomocą użytkownik może między innymi ustawić w jakim folderze mają zostać zapisywane logi, skonfigurować połączenie z bazą danych czy też zdecydować czy Qubic powinien domyślnie wykorzystywać transakcje czy też nie.

Jak wykorzystać Qubica w tworzonym projekcie?

Po wygenerowaniu pliku projektu, plików nagłówkowych i klas przez generator wystarczy zadbać o to aby wspomniany wcześniej plik konfiguracyjny `qb.properties` znalazł się jednym folderze z plikiem wykonywalnym tworzonego przez nas projektu, a także zaimportować nagłówek `Qubic.h` w tworzonych przez siebie plikach, które korzystają z Qubica.

Jak połączyć się z bazą danych?

Połączenie z bazą danych jest realizowane automatycznie podczas tworzenia instancji klasy Singleton `QbDatabase`. Oznacza to, że użytkownik nie musi samemu troszczyć się o połączenie z bazą danych przed skorzystaniem z operacji na bazie danych, wszystko to zostanie wykonane automatycznie.

Jak korzystać z operacji CRUD?

Operacje CRUD zostały udostępnione przez klasę `QbDatabase`. Użytkownik chcąc zapisać w bazie obiekt `employee` powinen wywołać metodę `QbDatabase::getInstance()->store(employee)`. Warto zauważyć, że może to być pierwsza metoda Qubica wywoływana przez użytkownika a i tak powinna bez problemu zapisać

obiekt w bazie, jest to zapewnione przez mechanizm wspomniany w poprzednim punkcie.

Jak tworzyć własne zapytania za pomocą Qubica?

Przykładowo gdy użytkownik bazy MySQL oraz Qubica chce stworzyć za jego pomocą zapytanie powinien wykorzystać on klasę `QbMySQLQuery`, która umożliwi mu skonstruowanie zapytania na wzór zapytań SQL. Atutem takiego rozwiązania jest to, że użytkownik nie musi znać wybranego dialektu SQL, wystarczy jedynie szerokopojęta znajomość SQL. Przykładowe zapytanie prezentuje się następująco:

```
QbMySQLQuery query = QbMySQLQuery(Employee::CLASSNAME);
query.appendWhere(Employee::FIRSTNAME, "Ryo", QbQuery::EQUALS);
query.appendAnd();
query.appendWhere(Employee::SALARY, "150", QbQuery::MORE_THAN);
list = QbDatabase::getInstance()->load(&query);
```

Listing 4.5: Tworzenie zapytań Qubica

Jak korzystać z metod dostępu do powiązanych danych?

Generator opisu udostępniany wraz z Qubiciem podczas generowania plików uwzględnia wystąpienia relacji jeden do wielu oraz wiele do wielu i automatycznie dodaje nowe metody do generowanych klas. Gdy w bazie pojawia się relacja jeden do wielu, na przykład `Employee - Company`, w klasie `Company` utworzona zostanie nowa metoda dostępu `getEmployees()` umożliwiająca uzyskanie dostępu do wszystkich pracowników pracujących w danej firmie.

4.6 Przykładowa aplikacja wykorzystująca Qubica

W celu zaprezentowania sposobu działania Qubica rozważmy następujący schemat bazy danych, której struktura zawiera zarówno relację jeden do wielu jak i wiele do wielu:

```
DROP DATABASE EMPLOYEES;

CREATE DATABASE EMPLOYEES;
```

```
USE EMPLOYEES;

CREATE TABLE COMPANY (
        ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        COMPANYNAME VARCHAR(30)
);

CREATE TABLE EMPLOYEE (
        ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        FIRSTNAME VARCHAR(10),
        LASTNAME VARCHAR(20),
        BIRTHDAY DATE,
        GENDER VARCHAR(6),
        COMPANY INT,
        HIREDATE TIMESTAMP,
        SALARY DOUBLE,
        CHILDREN INT,
        FOREIGN KEY(COMPANY) REFERENCES COMPANY(ID)
);

CREATE TABLE DEPARTMENT (
        ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        DEPARTMENTNAME VARCHAR(20)
);

CREATE TABLE ASSIGNMENT (
        ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        EMPLOYEE INT,
        DEPARTMENT INT,
        FOREIGN KEY(EMPLOYEE) REFERENCES EMPLOYEE(ID),
        FOREIGN KEY(DEPARTMENT) REFERENCES DEPARTMENT(ID)
);

COMMIT;
```

Listing 4.6: Plik tworzący przykładową bazę danych

Na podstawie zaprezentowanego schematu wygenerowane zostały pliki klas oraz projektu, następnie w celach prezentacyjnych stworzony został zaprezentowany poniżej plik, który wykonuje podstawowe operacje na bazie danych:

```
#include <QCoreApplication>
#include <QbCore/qbdatabase.h>
#include <QbCore/qbpersistable.h>
#include <QbCore/qbmysqlquery.h>
#include <QbTest/employee.h>
#include <QbTest/company.h>
#include <QbTest/assignment.h>
#include <QbTest/department.h>
#include <QsLog/QsLog.h>

int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);

    //storing objects in database
    Company company1 ("Google");
    QDate birthday(1995, 5, 17);
    QDateTime hiredate = QDateTime::currentDateTime();
    Employee employee1 (birthday, "Lee", "Jones", "M", &
       company1, hiredate, 2790.5, 2);
    Employee employee2 (birthday, "Joe", "Smith", "M", &
       company1, hiredate, 2290.5, 0);
    QbDatabase::getInstance()->store(employee1);
    QbDatabase::getInstance()->store(employee2);

    //updating objects in database
    Company company2 ("Facebook");
    employee2.setFirstname("Ryo");
    employee2.setCompanyPtr(&company2);
    QbDatabase::getInstance()->update(employee2);

    //loading objects from database
    qDebug() << "Loaded objects:\n";
    Employee empty;
    QList<QbPersistable*> list = QbDatabase::getInstance()->
       load(empty);
    for(int i=0; i<list.size(); i++)
    {
        Employee* loaded = (Employee*) list.at(i);
        qDebug() << loaded->getID() << "\t" << loaded->
           getFirstname() << "\t" << loaded->getLastname() << "
           \t" << loaded->getCompanyPtr()->getCompanyname()
                << "\t" << loaded->getSalary();
    }
    if(list.size() == 0) qDebug() << "-";
```

```
//query database
qDebug() << "\nQuery result:\n";
QbMySQLQuery query = QbMySQLQuery(Employee::CLASSNAME);
query.appendWhere(Employee::FIRSTNAME, "Ryo", QbQuery::
    EQUALS);
query.appendAnd();
query.appendWhere(Employee::SALARY, "1500", QbQuery::
    MORE_THAN);
list = QbDatabase::getInstance()->load(&query);
for(int i=0; i<list.size(); i++)
{
    Employee* loaded = (Employee*) list.at(i);
    qDebug() << loaded->getID() << "\t" << loaded->
        getFirstname() << "\t" << loaded->getLastname() << "
        \t" << loaded->getCompanyPtr()->getCompanyname()
             << "\t" << loaded->getSalary();
}
if(list.size() == 0) qDebug() << "-";

//advanced query (one to many)
qDebug() << "\nAdvanced query result:\n";
list = company1.getEmployees();
for(int i=0; i<list.size(); i++)
{
    Employee* loaded = (Employee*) list.at(i);
    qDebug() << loaded->getID() << "\t" << loaded->
        getFirstname() << "\t" << loaded->getLastname() << "
        \t" << loaded->getCompanyPtr()->getCompanyname()
             << "\t" << loaded->getSalary();
}
if(list.size() == 0) qDebug() << "-";

//storing new objects in database
Department department1 ("Human Resources");
Department department2 ("Board");
Assignment assignment1 (&department1, &employee1);
Assignment assignment2 (&department2, &employee2);
QbDatabase::getInstance()->store(department1);
QbDatabase::getInstance()->store(department2);
QbDatabase::getInstance()->store(assignment1);
QbDatabase::getInstance()->store(assignment2);

//advanced query (many to many)
qDebug() << "\nAdvanced query result:\n";
list = employee1.getDepartments();
for(int i=0; i<list.size(); i++)
{
```

```
        Department* loaded = (Department*) list.at(i);
        qDebug() << loaded->getID() << "\t" << loaded->
            getDepartmentname();
    }
    if(list.size() == 0) qDebug() << "-";

    //advanced query (many to many)
    qDebug() << "\nAdvanced query result:\n";
    list = department2.getEmployees();
    for(int i=0; i<list.size(); i++)
    {
        Employee* loaded = (Employee*) list.at(i);
        qDebug() << loaded->getID() << "\t" << loaded->
            getFirstname() << "\t" << loaded->getLastname() << "
            \t" << loaded->getCompanyPtr()->getCompanyname()
                 << "\t" << loaded->getSalary();
    }
    if(list.size() == 0) qDebug() << "-";

    //loading list of currently synchronized objects
    qDebug() << "\nSynchronized objects:\n";
    list = *(QbDatabase::getInstance()->getSynchronizedObjects
        ());
    for(int i=0; i<list.size(); i++)
    {
        QbPersistable* synchronized = list.at(i);
        qDebug() << synchronized->getObjectUpperName() + ":" +
            QString::number(synchronized->getID());
    }
    if(list.size() == 0) qDebug() << "-";

    //removing objects from database
    QbDatabase::getInstance()->remove(assignment1);
    QbDatabase::getInstance()->remove(assignment2);
    QbDatabase::getInstance()->remove(employee1);
    QbDatabase::getInstance()->remove(employee2);
    QbDatabase::getInstance()->remove(company1);
    QbDatabase::getInstance()->remove(company2);
    QbDatabase::getInstance()->remove(department1);
    QbDatabase::getInstance()->remove(department2);

    //loading list of currently synchronized objects
    qDebug() << "\nSynchronized objects:\n";
    list = *(QbDatabase::getInstance()->getSynchronizedObjects
        ());
    for(int i=0; i<list.size(); i++)
    {
```

```
        QbPersistable* synchronized = list.at(i);
        qDebug() << synchronized->getObjectUpperName() + ":" +
            QString::number(synchronized->getID());
    }
    if(list.size() == 0) qDebug() << "-";

    //closing database connection to prevent application crash
    QbDatabase::deleteInstance();

    return app.exec();
}
```

Listing 4.7: Plik przedstawiający przykładową aplikację korzystającą z Qubica

Jak widać wykonanie podstawowych operacji CRUD ogranicza się do wywołania pojedyńczej funkcji, bez konieczności wcześniejszego zapewnienia połączenia z bazą ani konfiguracji z poziomu kodu. Jest to bardzo dobry wynik biorąc pod uwagę poprawne działanie aplikacji widoczne na kolejnym listingu, tym razem jest to zrzut z konsolii wykonanego programu, przy użyciu świeżo utworzonej bazy danych:

```
Loaded objects:

1       "Lee"   "Jones"         "Google"        2790.5
2       "Ryo"   "Smith"         "Facebook"      2290.5

Query result:

2       "Ryo"   "Smith"         "Facebook"      2290.5

Advanced query result:

1       "Lee"   "Jones"         "Google"        2790.5

Advanced query result:

1       "Human Resources"

Advanced query result:

2       "Ryo"   "Smith"         "Facebook"      2290.5

Synchronized objects:
```

```
"COMPANY:1"
"EMPLOYEE:1"
"EMPLOYEE:2"
"COMPANY:2"
"DEPARTMENT:1"
"DEPARTMENT:2"
"ASSIGNMENT:1"
"ASSIGNMENT:2"

Synchronized objects:

-
```

Listing 4.8: Plik wynikowy przykładowej aplikacji korzystającej z Qubica

Dodatkowo rezultatem pracy programu był szczegółowy log Qubica, jednakże ze względu na jego długość został on pominięty w niniejszej pracy. Wszystkie operacje zostały wykonane zgodnie z oczekiwaniami, co pozwala sądzić, że mapowanie obiektowo-relacyjne co najmniej w tym kontekście odbywa się poprawnie.

4.7 Analiza Qubica

W rozdziale trzecim przedstawiona została analiza istniejących narzędzi służących do mapowania obiektowo-relacyjnego w języku C++, między innymi na jej podstawie stworzona została aplikacja szkieletowa Qubic realizująca to samo zagadnienie. Jej analiza wygląda następująco:

Charakter oprogramowania	open-source
Licencja	GNU/GPL
Wspierane bazy danych	MySQL oraz łatwa możliwość
	rozszerzenia o inne rodzaje baz
Wykorzystywane narzędzia	Qt
Poziom skomplikowania interfejsu	niski
Dostępność generatora opisu	tak
Dodatkowe możliwości	walidacja danych, metody dostępu
	do powiązanych obiektów

Tablica 4.1: Analiza aplikacji szkieletowej Qubic

W porównaniu do przeanalizowanych wcześniej narzędzi Qubic wyróżnia się przede wszystkim niskim poziomem skomplikowania interfejsu, co można zauważyć szczególnie w poprzedniej sekcji opisującej przykładową aplikacje stworzoną w oparciu o Qubica. Ponadto, udostępnia on wszystkie podstawowe mechanizmy charakterystyczne dla mapowania obiektowo-relacyjnego, a także walidację danych podzczas generowania opisu czy gotowe metody dostępu do obiektów powiązanych.

Qubic należy do wolnego oprogramowania udostępnianego na licencji GNU-/GPL podobnie jak i większość analizowanych narzędzi. Istotną jego cechą jest wykorzystanie aplikacji szkieletowej Qt, podobnie jak analizowana wcześniej biblioteka QxOrm aplikacje tworzone w oparciu o Qubica muszą korzystać także z Qt.

4.8 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

Autorowi niniejszej pracy udało się stworzyć aplikację szkieletową realizującą mapowanie obiektowo-relacyjne, która stanowi dobrą alternatywę dla istniejących już narzędzi. Qubic wspiera wszystkie podstawowe mechanizmy charakterystyczne dla tego typu narzędzi, dodatkowo udostępnia generator opisu mapowania, a jego interfejs należy do najmniej skomplikowanych co zostało przedstawione w poprzednich rozdziałach pracy.

Dzięki realizacji wszystkich postawionych wcześniej celów oraz wymagań udało się osiągnąć zamierzone rezultaty, jednakże przy okazji pisanie niniejszej pracy okazało się być bardzo dobrym sposobem zdobycia wiedzy na temat mapowania obiektowo-relacyjnego, który należy do ciekawszych zagadnień dotyczących inżynierii oprogramowania.

5.2 Perspektywy rozwoju pracy

W celu rozwoju niniejszej pracy dyplomowej należy przede wszystkim rozważyć dalsze prace nad stworzoną aplikacją szkieletową. W rozdziale 4.8 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 przedmiot 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] http://www.wirtualnemedia.pl/artykul/coraz-dluzej-ogladamy-telewizje-najwiec Data dostępu – 15.01.2017.

[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/. Data dostępu – 31.08.2014.

[7] Qt Project Documentation. http://qt-project.org/doc/. Data dostępu – 31.08.2014.

[8] MySQL Documentation. http://dev.mysql.com/doc/. Data dostępu – 31.08.2014.

[9] Trendy Google. https://www.google.pl/trends/explore#q=orm%20-java%2C%20orm%20php%2C%20orm%20python%2C%20orm%20c&cmpt=q. Data dostępu – 31.08.2014.

[10] Tarnowski, Ireneusz. Relacyjne bazy danych. http://gis.pwr.wroc.pl/arch/I%20rok.kons/bazy_danych_wyklad_Ireneusz_Tarnowski.pdf. Data dostępu – 31.08.2014.

[11] Grosser, Andrzej. Szkielety tworzenia aplikacji. http://icis.pcz.pl/ agrosser/wsztap.pdf. Data dostępu – 31.08.2014.

[12] GitHub. https://github.com/. Data dostępu – 31.08.2014.

[13] QxOrm Library for C++. http://www.qxorm.com/qxorm_en/home.html. Data dostępu – 31.08.2014.

[14] QsLog. http://blog.codeimproved.net/tag/qslog/. Data dostępu – 31.08.2014.

[15] Debea. http://debea.net/trac. Data dostępu – 31.08.2014.

[16] OpenORM. https://code.google.com/p/openorm/. Data dostępu – 31.08.2014.

Spis rysunków

Spis tablic

Spis listingów