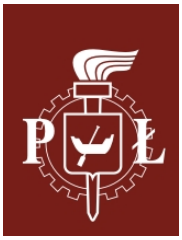


Politechnika Łódzka
Wydział Fizyki Technicznej, Informatyki i
Matematyki Stosowanej

GENERATOR OPISU MAPOWANIA OBIEKTOWO-RELACYJNEGO W C++

Sebastian Florek

Praca inżynierska wykonana pod
przewodnictwem dr. Arkadiusza
Tomczyka

Sierpień 2014

Spis treści

Rozdział 1

Wstęp

1.1 Problematyka i zakres pracy

Wzorzec architektoniczny typu Model View Controller [1] jest obecnie szeroko używany przy projektowaniu aplikacji. Implementacja modelu warstwy danych w rozbudowanych aplikacjach opartych o bazy danych jest czasochłonna i kosztowna. Rozwiązaniem tego problemu jest zlecenie generowania warstwy danych aplikacji zewnętrznej.

Patrząc na średniej wielkości bazę danych z 30 tabelami, po 5 wierszy na tabelę, człowiek musiałby napisać 30 klas po minimum 80 linijek każda, co przekłada się na 2400 linii kodu. Oszczędność czasu, a co za tym idzie i pieniędzy przy zleceniu tego zadania programowi jest ogromna, również szansa popełnienia błędu maleje znacząco, zakładając spełnienie przez użytkownika kryteriów narzuconych przez tego typu aplikacje.

Narzędzia czy aplikacje szkieletowe służące do mapowania obiektowo-relacyjnego [2] używane przez programistów muszą być zazwyczaj dopasowywane do ich potrzeb. Pozwalają one zachować połączenie między bazą danych, a ich systemem. W fazie deweloperskiej cyklu produkcji można zauważyć, że procesy projektowania i implementacji warstwy danych są bardzo podobne, co za tym idzie, stają się one rutynowymi zadaniami cyklu produkcyjnego.

Po zbadaniu obecnie dostępnych narzędzi służących do mapowania obiektowo-relacyjnego można znaleźć pewne wzorce i programy służące do generowania warstwy

[1] Model View Controller (MVC) - wzorzec służący do organizowania struktury aplikacji [12].

[2] Mapowanie Obiektowo-Relacyjne (ang. ORM, Object Relational Mapping) - przedstawia sposób odwzorowania obiektowej architektury systemu informatycznego na bazę danych lub inny element systemu o relacyjnym charakterze [2].

danych aplikacji, m. in.: wzorzec Data Access Object [3] czy aplikację Hibernate, jednakże ich niska skuteczność i wydajność związane z wieloma zapytaniami do bazy oraz czasem konfiguracji sprawiają, że szukamy czegoś innego. Większość tego typu narzędzi bardziej skupia się na rozwiązaniu konkretnych problemów związanych z bazą danych. Zaniedbują natomiast warstwę danych, która jest bardzo ważnym komponentem i najniższą warstwą aplikacji opartych o bazy danych.

Głównym problemem staje się więc stworzenie warstwy danych, która zapewni bezpieczeństwo, wydajność oraz będzie optymalna, a następnie generatora, który pozwoli zautomatyzować ten proces. W przeciwieństwie do większości istniejących narzędzi, które generują bazę danych na podstawie istniejącej warstwy danych aplikacji, nasza aplikacja będzie generowała warstwę danych aplikacji w oparciu o istniejącą bazę danych.

Proponowanym rozwiązaniem powyższych problemów będzie zaprojektowana aplikacja o nazwie Qubic. Dzielić ona będzie główną warstwę danych na mniejsze warstwy na podstawie istniejącej bazy danych. Kluczem jest pozwolenie innym deweloperom na prace z konkretnymi obiektami, bez praktycznej znajomości języków typu SQL [4] do obsługi baz danych.

Aplikacja będzie składała się z dwóch oddzielnie realizowanych i niezależnych modułów. Moduł generatora opisu mapowania obiektowo-relacyjnego będzie udostępniony jako biblioteka współdzielona. Dzięki takiemu rozwiązaniu moduły będą mogły być używane niezależnie. W niniejszej pracy zostanie przedstawiony moduł generujący opis relacyjnej bazy danych w postaci plików klas języka C++ wraz z teoretycznym rozwiązaniem problemu generowania opisu mapowania obiektowo-relacyjnego. Drugi moduł będzie zajmował się procesem mapowania obiektowo-relacyjnego wygenerowanych klas na obiekty baz danych i udostępniał odpowiednie interfejsy do obsługi operacji zapisu i odczytu na bazie danych.

1.2 Cele pracy

1.2.1 Przybliżenie istniejących aplikacji szkieletowych z dziedziny mapowania obiektowo-relacyjnego.

Spełnienie tego celu wiąże się z poznaniem problematyki generowania opisu mapowania obiektowo-relacyjnego. Ważnym krokiem będzie analiza istniejących aplikacji

[3] Data Acces Object (DAO) - komponent używany do oddzielenia i enkapsulacji dostępu do danych [13].

[4] SQL (ang. Structured Query Language) - język programowania stworzony do zarządzania danymi, które trzymane są w relacyjnych bazach danych [5].

zajmujących się mapowaniem obiektowo-relacyjnym i posiadających funkcję generowania warstwy danych aplikacji. Należy zdefiniować oraz zaproponować rozwiązanie problemu generowania warstwy danych aplikacji, a następnie przeanalizować istniejące aplikacje służące do mapowania obiektowo-relacyjnego oferujące funkcję generowania warstwy danych.

1.2.2 Poznanie technik tworzenia i stworzenie aplikacji szkieletowej z zakresu generowania opisu mapowania obiektowo-relacyjnego w C++.

Po rozpoznaniu problematyki tematu i zaproponowaniu własnego rozwiązania należy przybliżyć techniki projektowania aplikacji szkieletowych w oparciu o konkretne źródła wraz z istniejącymi rozwiązaniami architektonicznymi, które nawiązują do generowania opisu mapowania obiektowo-relacyjnego. Kolejnym krokiem jest stworzenie aplikacji w oparciu o zidentyfikowane problemy w celu ich rozwiązania.

1.2.3 Porównanie stworzonej aplikacji z już istniejącymi aplikacjami szkieletowymi w języku C++ oraz w innych językach programowania.

Ostatni cel wiąże się z testami stworzonej aplikacji. Po odpowiednich testach należy zebrać wyniki i porównać z wynikami podobnych aplikacji z zakresu generowania opisu mapowania obiektowo-relacyjnego. Na koniec zebrane dane należy przedstawić w postaci podsumowania i opisać wyciągnięte wnioski.

1.3 Metoda badawcza

1. Studia literaturowe z dziedziny generowania opisu mapowania obiektowo-relacyjnego w języku C++.

   Obecnie dostępne źródła z tej dziedziny nie są sformalizowane. Dostępne są jedynie opisy i dokumentacje istniejących aplikacji, które uwzględniają w sposób ogólny ich budowę. Znalezione i użyte w tej pracy źródła nie są dostępne w języku polskim, więc muszą być tłumaczone w większości z języka angielskiego. Jako że nie ma oficjalnych książek dotyczących tematyki generowania opisu mapowania obiektowo-relacyjnego, większość źródeł tu zebranych to źródła elektroniczne, artykuły i dokumentacje.

2. Analiza wymagań aplikacji szkieletowych generujących opis mapowania obiektowo-relacyjnego.

   Narzędzia zajmujące się generowaniem opisu mapowania obiektowo-relacyjnego są zazwyczaj tylko dodatkami do typowych aplikacji typu ORM. Nie znajdziemy tu modelu aplikacji szkieletowej, na którym można bazować. Wymagania postawione takiej aplikacji są zazwyczaj takie same i są one podyktowane przez aplikacje zajmujące się mapowaniem obiektowo-relacyjnym. Podobnie jest i w tym przypadku gdzie wymagania generatora opisu są postawione przez drugi moduł aplikacji zajmujący się mapowaniem obiektowo-relacyjnym.

3. Proces projektowania i tworzenia Qubica.

   W oparciu o zebrane informacje i wymagania aplikacji szkieletowych służących do generowania opisu mapowania obiektowo-relacyjnego zostanie stworzona aplikacja szkieletowa mająca na celu rozwiązanie problemów zidentyfikowanych w procesie analizy.

4. Testy i wnioski dotyczące stworzonego narzędzia do generowania warstwy danych aplikacji w oparciu o bazę danych.

   Metoda ta służy do wyciągnięcia wniosków na temat stworzonej aplikacji. Przeprowadzone zostaną testy porównawcze. Na podstawie wyników testów wyciągnięte zostaną odpowiednie wnioski na temat sposobu rozwiązania przedstawionych w pracy problemów oraz Qubica. Wszystko to pozwoli stwierdzić czy proponowane rozwiązanie jest lepsze, tańsze, szybsze od porównywanych.

1.4 Przegląd literatury w dziedzinie

Źródła z zakresu języka C++

Użyte w tej pracy źródła dotyczące języka C++ służą przede wszystkim poznaniu technik programowania bibliotek współdzielonych oraz technik metaprogramowania. Dodatkowym celem przy pisaniu samej aplikacji jest chęć poznania nowego standardu języka C++11, który również jest przedstawiony w użytych źródłach. Szczegółowe omówienie tego standardu zostało przedstawione na stronie twórcy języka i służyć będzie jako główne źródło wiedzy [7]. Sposób tworzenia bibliotek i techniki metaprogramowania zostały opisane w książce Advanced C++ Metaprogramming [4].

Źródła z zakresu narzędzi i aplikacji do mapowania obiektowo-relacyjnego

Tematyka generowania opisu mapowania obiektowo-relacyjnego jest związana z narzędziami ORM i brak jest książek dedykowanych tej tematyce. Do zrozumienia samej idei działania generatora należy przybliżyć działanie narzędzi do mapowania obiektowo-relacyjnego. W pozycjach EJB 3 Java persistence API [1] oraz Hibernate w akcji [2] znajdziemy opis działania narzędzi typu ORM oraz techniki mapowania obiektowo-relacyjnego.

Źródła z zakresu działania frameworka Qt

Framework[5] Qt to zestaw bibliotek i narzędzi przydatnych programistom. Dzięki mechanizmowi refleksji, wsparciu dialektów SQL czy prostej budowie aplikacji graficznych znacznie ułatwia tworzenie dużych aplikacji. Użyta książka Introduction to Design Patterns in C++ with Qt [3] opisuje w prosty sposób mechanizm refleksji, wzorce czy tworzenie bibliotek przy użyciu tego frameworka. Dodatkowo głównym narzędziem w etapie tworzenia aplikacji będzie dokumentacja Qt dostępna w internecie pod adresem [8].

Źródła z zakresu SQL

W celu generowania opisu bazy danych potrzebna jest znajomość struktury bazy, typów pól, połączeń. Wymaga to dla niektórych dialektów SQL pisania dość nietypowych zapytań. Potrzebne informacje zostały zasięgnięte ze źródeł elektronicznych i odpowiednich dokumentacji konkretnych dialektów, m. in: stron internetowa z dokumentacją dialektu MySQL [6].

1.5 Układ pracy

Tematem pracy jest: Generator opisu mapowania obiektowo-relacyjnego w C++, zaś za główny cel przyjęto rozwiązanie problemu automatycznego generowania warstwy danych w aplikacjach opartych o relacyjne bazy danych.
Rozdział 1 zawiera szczegółowy opis problemu. Przedstawione jest w nim proponowane rozwiązanie problemu wraz z opisem metod badawczych użytych do analizy tematu. Podsumowane zostają również główne założenia i cele pracy. Na koniec przeprowadzony zostaje przegląd literatury z dziedziny generowania opisu mapowania obiektowo-relacyjnego. Zostają w nim wyróżnione najważniejsze tematyki dotyczące prezentowanego tematu wraz z krótkim opisem użytych źródeł.

[5] Z ang. framework - aplikacja szkieletowa

W rozdziale 2 przybliżona zostaje tematyka tworzenia aplikacji opartych o warstwy. Znajduje się tam opis warstwy dostępu do danych, która musi zostać wygenerowana. Następnie po kolei przedstawione zostają tematyki związane z bazami danych, mapowaniem obiektowo-relacyjnym oraz użytymi wzorcami projektowymi. Na końcu znajduje się opis sposobu działania bibliotek współdzielonych oraz aplikacji szkieletowej Qt.
Rozdział 3 zawiera analizę istniejących aplikacji służących do generowania opisu mapowania obiektowo-relacyjnego. Wymienione zostają same aplikacje, wspierane systemy, opisane zostają ich słabe i mocne punkty.
Rozdział 4 jest zbiorem technologii i metod użytych przy tworzeniu części praktycznej pracy. Opisane zostają: wybrany język programowania, użyte narzędzia, środowisko programistyczne, a także biblioteki i system baz danych użyte w fazie projektowej i implementacyjnej.
Kolejny rozdział opisuje fazę projektowania i implementacji aplikacji Qubic. Spisane są wymagania funkcjonalne aplikacji, a także ograniczenia projektu. Przybliżony zostaje projekt w postaci diagramów klas, diagramów użycia. Pokazane zostają użycia wzorców projektowych. Wskazane zostają kluczowe punkty aplikacji wraz z kodem źródłowym i opisem. Następnie następuje faza testów stworzonej aplikacji oraz zestawienie i porównanie wyników testów podobnych aplikacji.
W podsumowaniu pracy przedstawiono uzyskane w fazie testowania wyniki stworzonej aplikacji. Opisane zostają zrealizowane cele, słabe i mocne punkty przedstawionego rozwiązania. Na podstawie wyników następuje ocena możliwości i przydatności zaproponowanego rozwiązania. Na końcu omówione zostają możliwe perspektywy rozwoju generatora opisu mapowania obiektowo-relacyjnego.

Rozdział 2

Generator opisu mapowania obiektowo-relacyjnego w C++

2.1 Podstawowe definicje

Użyte koncepcje i terminy używane w dalszej części pracy muszą zostać wyjaśnione w celu lepszego zrozumienia opisywanej problematyki. W kolejnych rozdziałach zostają objaśnione podstawowe pojęcia związane z tematyką generowania opisu mapowania obiektowo-relacyjneg w C++. Są opisane terminy związane z bazami danych, aplikacjami z zakresu ORM, wzorcami projektowymi, aplikacjami zajmującymi się generowaniem kodu. Na koniec przybliżone zostają biblioteki współdzielone oraz framework Qt użyty przy tworzeniu części praktycznej tej pracy.

2.1.1 Warstwa dostępu do danych

Warstwa dostępu do danych aplikacji jest najniższą warstwą w architekturze aplikacji. Jej głównym zadaniem jest stworzenie mostu pomiędzy bazą danych, a samą aplikacją, tak aby możliwe było wykonywanie podstawowych operacji na bazie danych z poziomu aplikacji, tj. odczytu, zapisu oraz tworzenia i usuwania rekordów. W programowaniu, warstwa danych służy zwróceniu referencji obiektu wraz z jego atrybutami, gdzie obiekt odpowiada tabeli w bazie danych, a jego atrybuty odpowiednim kolumnom tej tabeli.

Aplikacje używające warstwy dostępu do danych mogą być zależne lub niezależne od bazy danych. Jeśli warstwa dostępu do danych wspiera wiele typów baz danych, aplikacja staje się bardziej generyczna. Ułatwia to przystosowanie aplikacji do innego typu baz danych bez dużego wysiłku. Tego typu praktyki są stosowane często właśnie w narzędziach zajmujących się mapowaniem obiektowo-relacyjnym

[14].

2.1.2 Normalizacja bazy danych

Normalizacja bazy danych jest procesem organizacji tabel oraz ich pól w relacyjnych bazach danych, w taki sposób aby zminimalizować redundancję danych. Proces ten zazwyczaj ma na celu zdefiniowanie połączeń między tabelami, a następnie ich podział na mniejsze logiczne części, które zmniejszą powtarzalność danych w bazie [5].

2.1.3 SQL

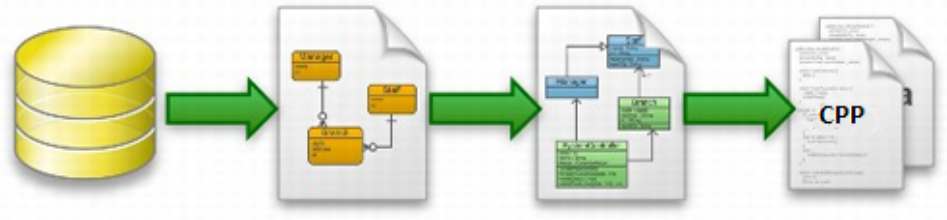
SQL jest językiem programowania stworzonym do zarządzania danymi, które są trzymane w relacyjnych bazach danych. Głównym zadaniem tego języka jest dodawanie, usuwanie, odczyt oraz aktualizacja danych. Na bazie języka SQL zostało stworzonych wiele systemów zarządzania relacyjnymi bazami danych oraz nowych dialektów SQL, które są obecnie powszechnie używane. Jednym z takich systemów zarządzania relacyjnymi bazami danych jest MySQL, który został użyty jako główny system przy projektowaniu i implementacji części praktycznej tej pracy [5].

2.1.4 Mapowanie Obiektowo-Relacyjne

Mapowanie obiektowo-relacyjne jest techniką programowania używaną w celu konwersji danych pomiędzy niekompatybilnymi systemami. Rozwiązaniem tego problemu jest stworzenie w pamięci programu wirtualnej bazy danych obiektów, które mogą być używane bezpośrednio przez aplikacje [2]. Obiekty przebywające w pamięci muszą być powiązane z danymi w bazie danych. Tabele relacyjnej bazy danych mają swoje odwzorowanie w obiektach języka programowania. Tworzone są one na podstawie metadanych opisujących to odwzorowanie. Mapowanie obiektowo-relacyjne musi być zatem procesem dwukierunkowym, tak aby obie strony relacji operowały zawsze na aktualnych danych. Samo rozwiązanie mapowania składa się z czterech elementów:

- interfejsu do przeprowadzania podstawowych operacji CRUD na obiektach klas zapewniających trwałość
- języka lub interfejsu programistycznego do określania zapytań związanych z klasami lub ich właściwościami
- narzędzia do określania metadanych

- technik implementacji ORM, zachowujących integralność między obiektami



Rys. 2.1: Uproszczony przebieg generowania opisu mapowania obiektowo-relacyjnego [15].

2.1.5 Wzorzec projektowy

Wzorce projektowe są obecnie szeroko stosowane przy tworzeniu oprogramowania. Pozwalają one na wyodrębnienie często używanych części algorytmów, tak aby mogły być używane przez różne części systemu. Nie są to gotowe rozwiązania, a jedynie opisy oraz szablony zawierające wskazówki rozwiązania tych samych problemów występujących w różnych sytuacjach.

Wzorce programowania obiektowego zazwyczaj pokazują relacje i powiązania pomiędzy obiektami bez specyfikacji tych obiektów. Są one stosowane na poziomie projektowania aplikacji.

Typy wzorców projektowych możemy podzielić na:

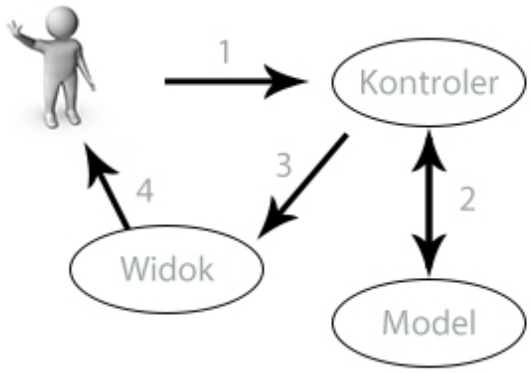
- Wzorce konstrukcyjne - opisujące proces tworzenia nowych obiektów. Ich zadaniem jest tworzenie, inicjalizacja oraz konfiguracja obiektów.
- Wzorce strukturalne - opisujące struktury powiązanych ze sobą obiektów.
- Wzorce czynnościowe - opisujące zachowanie i powiązania współpracujących ze sobą obiektów.

2.1.6 Wzorzec Model-Widok-Kontroler

Wzorzec projektowy model-widok-kontroler oddziela od siebie warstwy: dostępu do danych, kontrolera oraz widoku. Każda warstwa odpowiada za inne zadania.

- Warstwa modelu - zarządza dostępem do danych oraz wszelkimi zmianami z nimi związanymi
- Warstwa kontrolera - interpretuje sygnały płynące z warstwy widoku. Informuje warstwy dostępu do danych oraz widoku o wymaganych zmianach.
- Warstwa widoku - zarządza wyświetlaniem informacji.

Przykładowy obrazek przedstawia w prosty sposób schemat działania tego wzorca oraz przepływ informacji między warstwami.



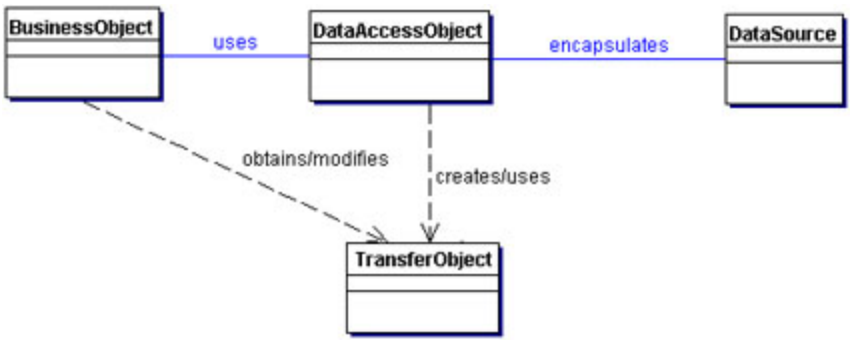
Rys. 2.2: Wzorzec Model-Widok-Kontroler. [17]

2.1.7 Wzorzec DAO

DAO jest komponentem, którego zadaniem jest dostarczanie jednolitego interfejsu do komunikacji między aplikacją, a źródłem danych(np. bazą danych czy plikiem). DAO dostarcza możliwość manipulacji danymi bez wykonywania bezpośrednich operacji na źródle danych, a jedynie na obiekcie. Głównymi zaletami takiego podejścia są prostota użycia i czytelność. Zmiana logiki nie wymaga zmian w źródle danych i na odwrót. Pozwala to również ukryć szczegóły związane ze źródłem

danych oraz w łatwy sposób zmienić je w razie potrzeby [13].

Wadami tego rozwiązania jest pojawienie się kolejnej warstwy interfejsu oraz zwiększenie ilości kodu, który musi zostać wykonany do realizacji dostępu do danych.



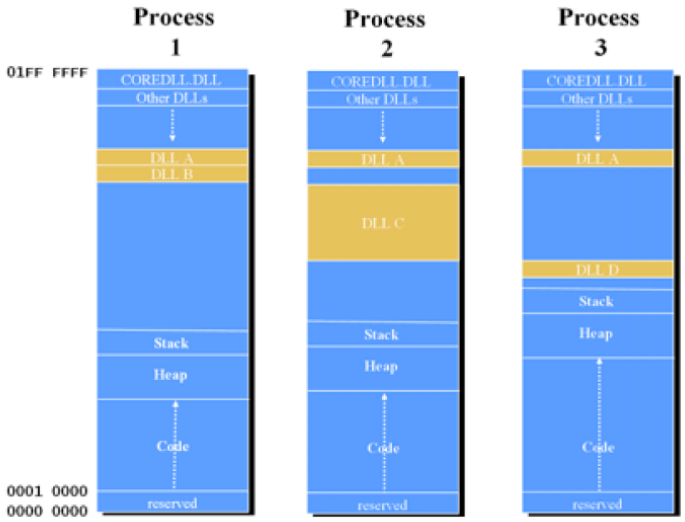
Rys. 2.3: Wzorzec DAO [18]

Jak widać w skład wzorca wchodzą:

- Obiekct biznesowy(Business Object) - obiekt przechowujący dane biznesowe
- Obiekt dostępu do danych(Data Access Object) - zapewnia transparentny dostęp do źródła danych
- Źródło danych(Data Source) - baza danych, plik XML, itp.
- Obiekt transferowy(Transfer Object) - przenosi dane z warstwy danych do obiektu biznesowego

2.1.8 Biblioteka współdzielona

Biblioteka współdzielona jest ładowana tylko raz do pamięci systemu. Każdy proces który załaduje taka bibliotekę do własnej pamięci procesu jedynie mapuje adresy do oryginalnych wywołań funkcji. W przeciwieństwie do bibliotek statycznych kod funkcji nie jest kopiowany do pamięci procesu, który je wywołuje. Przechowywane są jedynie wirtualne adresy funkcji do których odwołuje się proces. Raz załadowana biblioteka współdzielona może być więc używana przez wiele procesów jednocześnie co zmniejsza zużycie pamięci [9].



Rys. 2.4: Schemat działania biblioteki współdzielonej [19].

2.1.9 Qt framework

Framework Qt jest gotowym narzędziem skierowanym do programistów C++, jak również osób preferujących język CSS i Javascript do budowania aplikacji. Posiada moduły potrzebne do budowy kompletnej aplikacji, m. in.:

- Biblioteki języka C++ przeznaczone na wiele platform
- Wsparcie dla budowania interfejsu użytkownika przy użyciu różnych narzędzie
- Zintegrowane środowisko deweloperskie

Główną zaletą używania Qt jest możliwość pisania kodu skierowane pod wiele platform. Moduły obsługi bazy danych, budowy interfejsu użytkownika i wiele innych pozwalają znacznie skrócić czas potrzebny na tworzenie aplikacji. Jest również jednym z niewielu narzędzi w języku C++, które bardzo dobrze wspierają refleksję [8].

Rozdział 3

Analiza istniejących aplikacji do generowania opisu mapowania obiektowo-relacyjnego

W tym rozdziale zostaną przeanalizowane istniejące aplikacje zajmujące się generowaniem opisu mapowania obiektowo-relacyjnego. Sprawdzone zostaną ich słabe i mocne stron oraz wymagania.

3.1 Istniejące rozwiązania w dziedzinie

W języku C++ brak jest podobnych aplikacji zajmujących sie generowaniem warstwy dostępu do danych aplikacji. Istnieją aplikacje zajmujące się mapowaniem obiektowo-relacyjnym jednak bez funkcjonalności generowania jego opisu. W języku Java istnieje kilka aplikacji zajmujących się generowaniem opisu mapowania obiektowo-relacyjnego. Są nimi: JOOQ [11] oraz Hibernate [10]. Obie aplikacje zajmują się mapowaniem obiektowo-relacyjnym i posiadają funkcjonalność generowania warstwy dostępu do danych z istniejących systemów relacyjnych baz danych.

3.1.1 Wymagania aplikacji

Obie aplikacje, tj. JOOQ oraz Hibernate zostały stworzone przy użyciu języka Java. Dzięki temu mogą one być używane zarówno na systemach unixowych jak i tych z rodziny Windows. Sam Hibernate nie posiada opcji generowania modelu warstwy danych aplikacji. Dostępne są w tym celu odpowiednie wtyczki do środowisk programistycznych, takich jak Eclipse czy Intellij. Aplikacja JOOQ

natomiast rozwiązała ten problem i udostępnia funkcję generatora kodu z linii poleceń wywołując odpowiednią klasę. Jedynym wymogiem oprócz posiadania odpowiedniego środowiska z którego wygenerowany zostanie projekt na podstawie bazy danych jest stworzenie odpowiedniego pliku konfiguracyjnego.

3.2 Wady i słabe punkty istniejących rozwiązań

Jako pierwsze weźmy pod uwagę wady aplikacji Hibernate:

- Bardzo duża ilość API[1]. do nauki, przez co nie jest łatwy w obsłudze.
- Ze względu na budowę modelu warstwy danych aplikacji wymaga dużej ilości zapytań do bazy. Często przy bazach z dużą ilością rekordów wydajność zauważalnie spada.
- Wymaga użycia odpowiedniego środowiska programistycznego i wtyczek w celu generowania warstwy dostępu do danych aplikacji.

Druga aplikacja JOOQ rozwiązuje dwie z trzech wyżej wymienionych wad. Niestety nadal oferuje one generowanie modelu danych z zachowaniem czystego wzorca DAO, co wiąże się z większym zużyciem pamięci i w pewnych sytuacjach stratami wydajności. Oczywiście tego typu rozwiązania posiadają również bardzo wiele zalet:

- Nie wymagają od programisty znajomości języka SQL.
- Oszczędzają dużo czasu, ponieważ same zajmują się komunikacją z bazą i pozyskiwaniem danych.
- Pozwalają skupić się na logice biznesowej aplikacji, zamiast zarządzaniu bazą danych.
- Są niezależne od systemów baz danych. Mogą współpracować z wieloma róznymi systemami.

Są to najważniejsze zalety tego typu rozwiązań. Jednak jeśli chodzi o analizowanie w tym rozdziale aplikacje, żadna z nich nie współpracuje z aplikacjami pisanymi w języku C++. Pozwala to zaproponować własne rozwiązanie i wypełnić tę lukę.

[1] API(ang. Application Programming Interface) - dostarcza odpowiednie specyfikacje podprogramów, struktur danych, klas obiektów i wymaganych protokołów komunikacyjnych.

Rozdział 4

Technologie i metody użyte w części badawczej

W tym rozdziale zostaną opisane wszelkie użyte narzędzia oraz technologie, które posłużyły do stworzenia aplikacji Qubic. Zostanie opisany wybrany serwer bazy danych, a następnie środowisko deweloperskie. W dalszej części zostaną przedstawione użyte technologie wraz z metodykami programistycznymi użytymi w fazie projektowania aplikacji.

4.0.1 Język programowania

Do stworzenia aplikacji został wybrany język C++. Najważniejszym czynnikiem jego wyboru jest wydajność. W porównaniu podstawowych operacji między innymi popularnymi językami język ten jest zdecydowanie szybszy. Poniższa tabela przedstawia porównanie czasów wykonania: pętli, operacji tablicowych, podstawowych operacji matematycznych.

Language	CPU time			Slower than		Language version	Source code
	User	System	Total	C++	previous		
C++ *(optimized with -O2)*	1,520	0,188	1,708	-	-	g++ 4.5.2	link
Java *(non-std lib)*	2,446	0,150	2,596	52%	52%	1.6.0_26	link
C++ *(not optimized)*	3,208	0,184	3,392	99%	31%	g++ 4.5.2	link
Javascript (SpiderMonkey)	see comment (SpiderMonkey seems as fast as C++ on Windows)						
Javascript (nodejs)	4,068	0,544	4,612	170%	36%	0.8.8	link
Java	8,521	0,192	8,713	410%	150%	1.6.0_26	link
Python + Psyco	13,305	0,152	13,457	688%	54%	2.6.6	link
Ruby	see comment (Ruby seems 35% faster than standard Python)						
Python	27,886	0,168	28,054	1543%	108%	2.7.1	link
Perl	41,671	0,100	41,771	2346%	49%	5.10.1	link
PHP 5.4	roga's blog results (PHP 5.4 seems 33% faster than PHP 5.3)						
PHP 5.3	94,622	0,364	94,986	5461%	127%	5.3.5	link

Rys. 4.1: Porównanie wydajności popularnych języków względem C++. [16]

Drugim ważnym czynnikiem wyboru języka C++ jest brak istnienia aplikacji realizujących mapowanie obiektowo-relacyjne wraz z możliwością generowania warstwy dostępu do danych w tym języku.

4.0.2 System baz danych

Jako system zarządzania bazą danych został wybrany serwer MySQL. Czynniki które o tym zdecydowały to przede wszystkim mała zajętość pamięci oraz łatwa i szybka konfiguracja. Dodatkowo posiada on bardzo dobrą integrację z użytym frameworkiem Qt, który został użyty m.in.: w celu połączenia z bazą danych oraz odczytu jej struktury z poziomu języka C++.

4.0.3 Środowisko deweloperskie

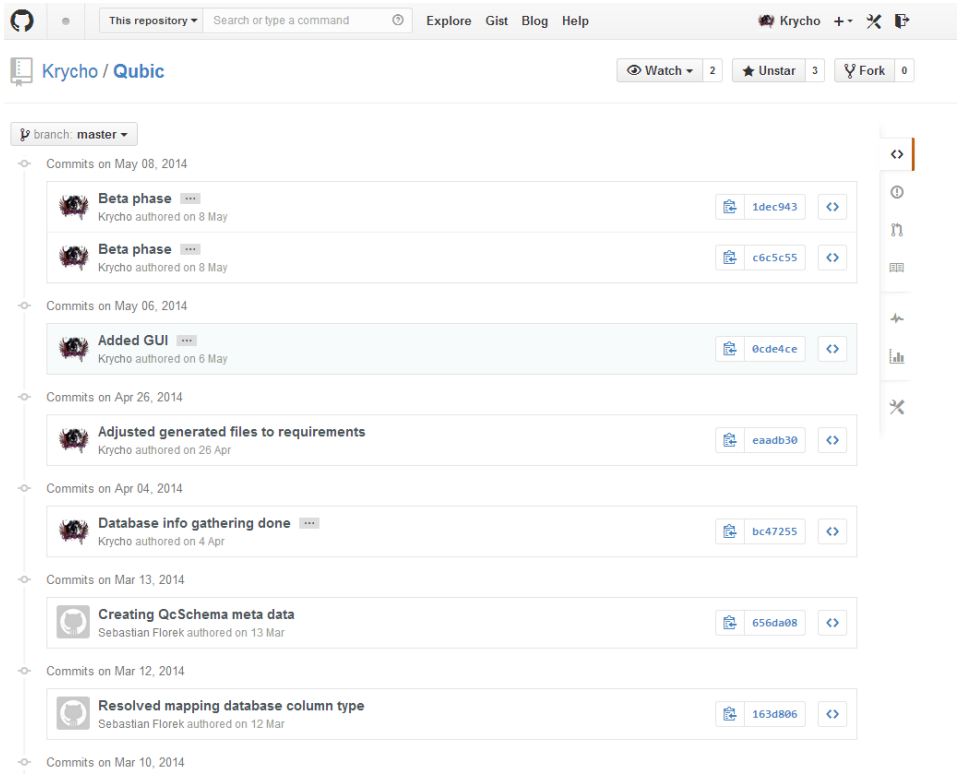
Wybrane środowisko deweloperskie jest częścią użytego frameworka Qt. Qt Creator jest zintegrowany z zestawem bibliotek użytego frameworka i udostępnia wsparcie składni oraz informacje o interfejsie programowania aplikacji.

4.0.4 Biblioteki

Użyty zestaw bibliotek należy do frameworka Qt. Udostępnia on biblioteki przenośne i narzędzia programistyczne dedykowane dla języków C++, Java czy QML. Głównym atutem są klasy służące do budowy graficznego interfejsu użytkownika, wsparcie połączenia z wieloma systemami baz danych oraz obsługiwany mechanizm refleksji. Wszystkie te rzeczy w znacznym stopniu ułatwiają tworzenie aplikacji oraz skracają czas potrzebny na jej stworzenie.

4.0.5 Inne narzędzia

Dodatkowym narzędziem użytym w procesie tworzenia aplikacji był system kontroli wersji o nazwie GIT. Pozwalało to kontrolować cały proces powstawania Qubica. Całość jest zintegrowana z portalem Github i przechowywana na prywatnym repozytorium.



Rys. 4.2: Log kontrolny zmian projektowych z portalu Github.

Rozdział 5

Projektowanie aplikacji szkieletowej Qubic

5.1 Analiza wymagań

W tym rozdziale zostanie przedstawiona analiza oraz część implementacyjna aplikacji Qubic. W pierwszej części są zdefiniowane wymagania aplikacji oraz opisane wymagania funkcjonalne wraz z ograniczenia projektu. Następnie zostają przedstawione główne diagramy dotyczące działania aplikacji. Kolejna część opisuje technologie i metodologie użyte w projekcie z podziałem na warstwy modelu, widoku i kontrolera w aplikacji. Kolejna część zawiera przedstawienie proponowanego rozwiązania wraz z częścią implementacyjna. Przybliżone zostają najważniejsze części kodu wraz z opisem. Ostatnim krokiem są testy i ocena funkcjonalności stworzonej aplikacji.

5.1.1 Studium możliwości

Główną ideą projektu jest rozwiązanie problemu generowania warstwy dostępu do danych aplikacji. W celu minimalizacji zużycia pamięci oraz uzyskania jak najlepszej wydajności została stworzona jedna warstwa obiektów, gdzie każdy obiekt klasy odzwierciedla tabelę w bazie danych. Nie istnieje podział na obiekty transakcyjne i biznesowe. Aplikacja daje nam możliwość generowania warstwy dostępu do danych z większości obecnie istniejących systemów zarządzania relacyjnymi bazami danych.

5.1.2 Wymagania funkcjonalne

Ze względu na budowę własnego systemu mapowania obiektowo-relacyjnego moduł generowania danych musi spełniać pewne wymagania:

1. Generowanie warstwy dostępu do danych w oparciu o relacyjne bazy danych.
2. Generowanie plików klas oraz plików źródłowych.
3. Generowane pliki powinny mieć czytelną i prostą budowę, aby były proste w użyciu dla programistów.
4. Aplikacja powinna trzymać dane struktury bazy w pamięci podręcznej w celu zachowania wydajności i zlikwidowania problemu częstego dostępu do bazy danych.
5. Wszelkie ustawienia powinny być w prosty sposób konfigurowalne, tak by zmiana w bazie nie wymuszała zmian kodu aplikacji.
6. Powinna generować dodatkowe funkcje pozwalające wyciągnąć powiązane dane w oparciu o relacje między tabelami.
7. Aplikacja powinna wspierać wiele systemów baz danych lub udostępniać łatwy sposób dodania ich wsparcia.
8. Aplikacja powinna w prosty sposób definiować mapowanie typów danych z bazy danych na typy języka.

5.1.3 Ograniczenia projektu

Ze względu na czytelność kodu oraz sposób działania modułu zajmującego się mapowanie obiektowo-relacyjnym na bazę danych zostały nałożone pewne ograniczenia:

- Każda tabela w bazie musi zawierać klucz główny.
- Każdy typ danych użyty w bazie musi zostać zarejestrowany w aplikacji i zmapowany na odpowiedni typ języka.
- Nazwy kolumn w tabelach mogą zawierać jedynie litery i cyfry ze względu na czytelność generowanego kodu.

Powyższe ograniczenia są rozpoznawane przez aplikację, a ich nie spełnienie wiąże się z brakiem możliwości generowania warstwy dostępu do danych. Kolejnym ograniczeniem jest brak integracji generatora z innymi aplikacjami do mapowania obiektowo-relacyjnego oraz współpraca jedynie z relacyjnymi bazami danych. Również błędne wskazanie relacji w tabelach może wiązać się z błędami logicznymi w wygenerowanym kodzie.

5.2 Projekt

5.2.1 Projekt warstwy danych

Poniżej przedstawiony kod został użyty do budowy bazy danych. Baza zawiera najważniejsze relacje potrzebne do testów tworzonej aplikacji, tj. jeden do wielu oraz 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)
```

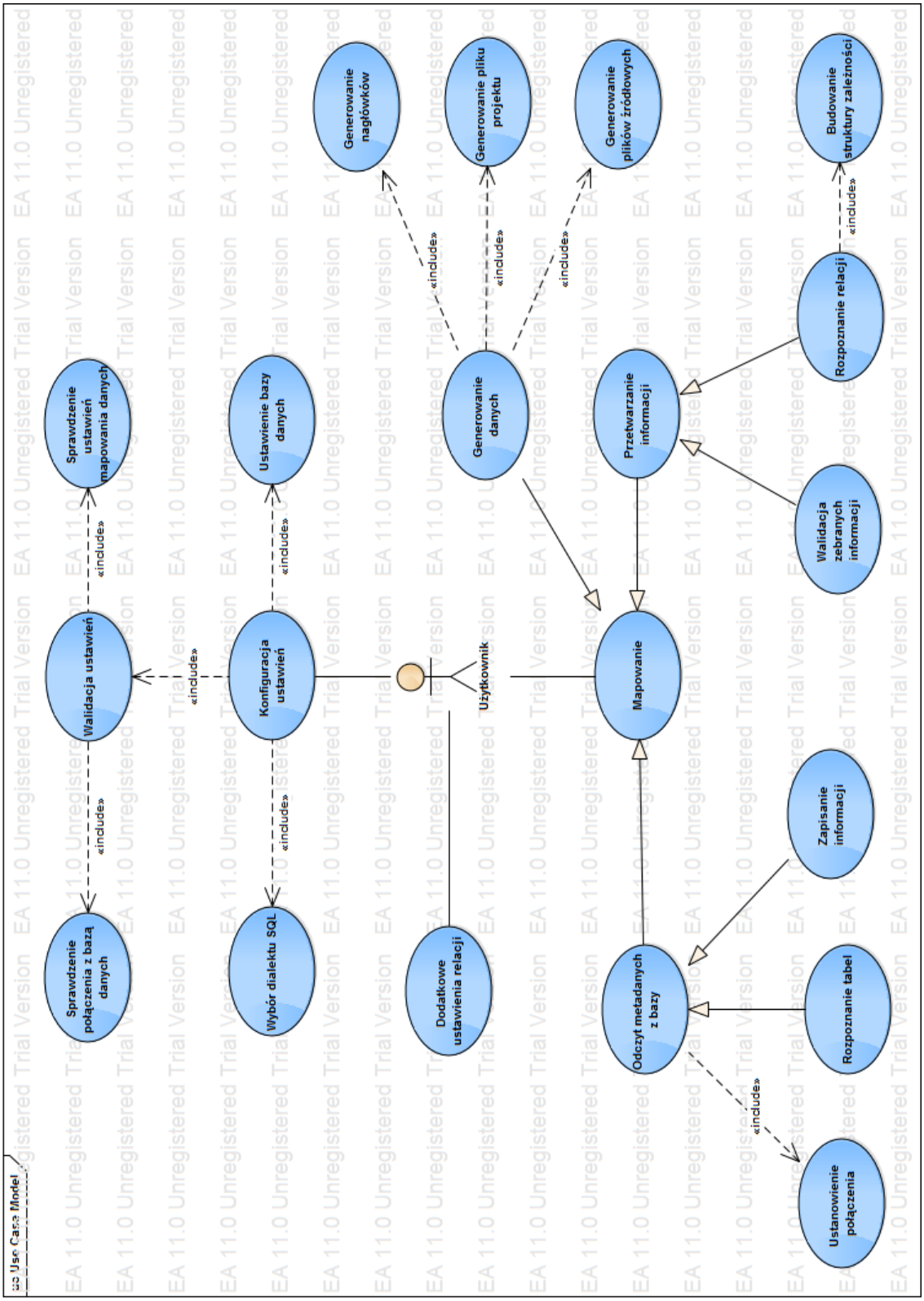
```
);

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;
```

Użytkownik bazy danych, który posłuży nam do ustanowienia połączenia musi posiadać uprawnienia do odczytu informacji z tabel opisujących budowę bazy danych. Przykładowo baza MySQL posiada dodatkową bazę przechowująca informacje o strukturze danych w systemie. Zaleca się więc używanie konta administratora bazy w celu generowania plików warstwy danych.

Poniższy rysunek przedstawia stworzoną bazę danych w postaci graficznej wraz z zaznaczeniem relacji między tabelami.

5.2.2 Projekt aplikacji

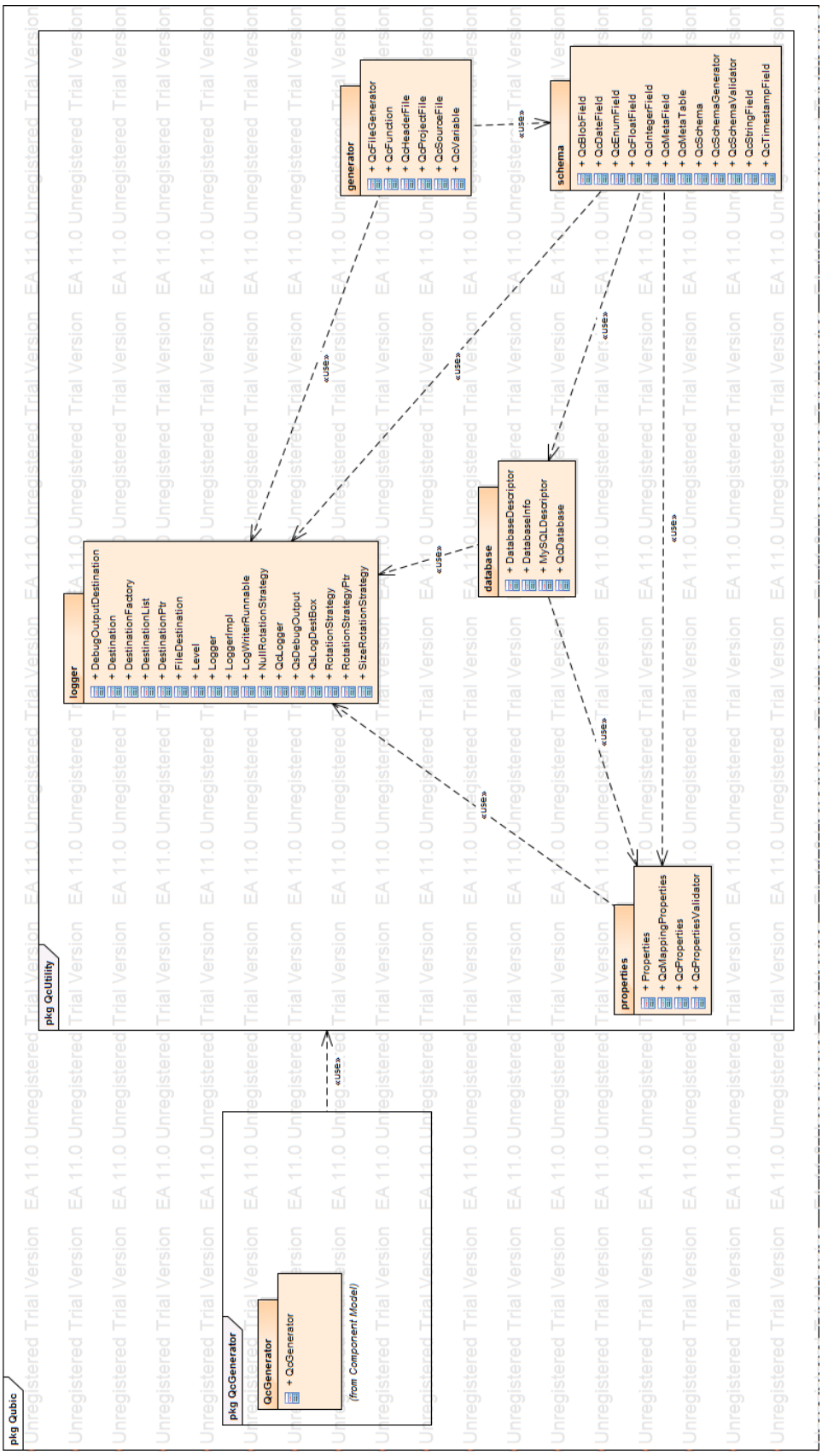
Projektowana aplikacja została oparta o wzorzec model-widok-kontroler, gdzie modelem jest baza danych, kontrolerem biblioteka współdzielona, natomiast widokiem przykładowa aplikacja korzystająca z funkcji udostępnianych przez bibliotekę. Poniższy diagram przypadków użycia przedstawia możliwości generatora:



Rys. 5.1: Diagram przypadków użycia modułu generatora.

Użytkownik ma możliwość konfiguracji ustawień aplikacji. Wszystkie ustawienia są sprawdzane przed połączeniem z bazą danych. Kolejną rzeczą jest opcja ręcznego definiowania relacji wiele do wielu między tabelami. Inne rodzaje relacji są rozpoznawane automatycznie. Ostatnim krokiem jest złożony proces mapowania, który najpierw odczytuje dane z bazy i zapisuje je do pamięci, następnie przeprowadza walidacje zebranych informacji oraz buduje struktury zależności. Ostatnim krokiem jest sam proces generowania projektu gotowego użycia w środowisku programistycznym.

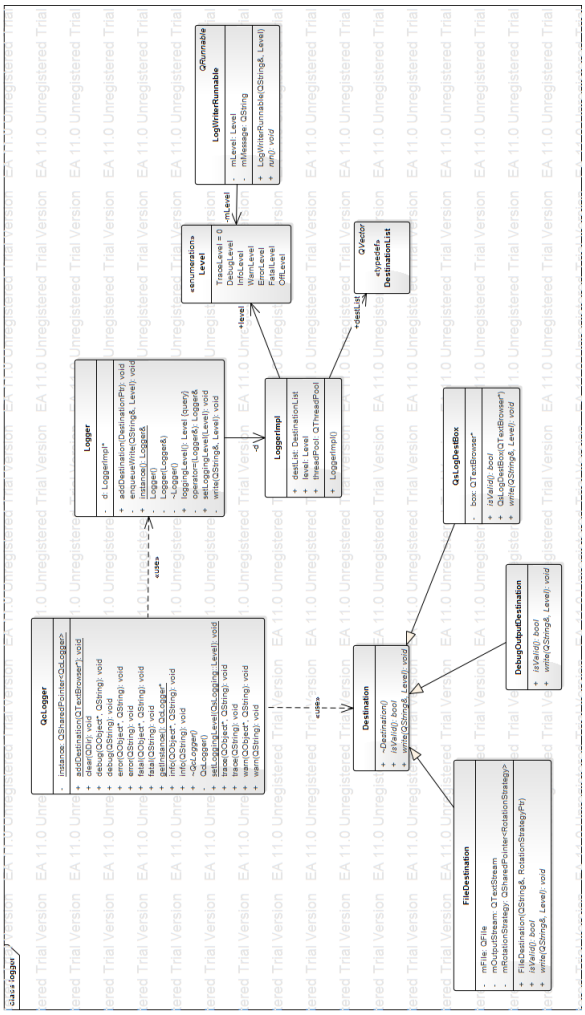
Poniższy diagram koncepcyjny przedstawia strukturę aplikacji:



Rys. 5.2: Diagram koncepcyjny generatora.

Paczka QcUtility jest w istocie biblioteką współdzieloną realizującą generowanie opisu mapowania obiektowo-relacyjnego. Moduł QcGenerator natomiast jest aplikacją opartą o funkcjonalność udostępnioną przez bibliotekę. Warstwą modelu danych jest baza danych, której struktura została przedstawiona we wcześniejszym rozdziale. Nie jest ona częścią samej aplikacji, dlatego nie została przedstawiona na diagramie.

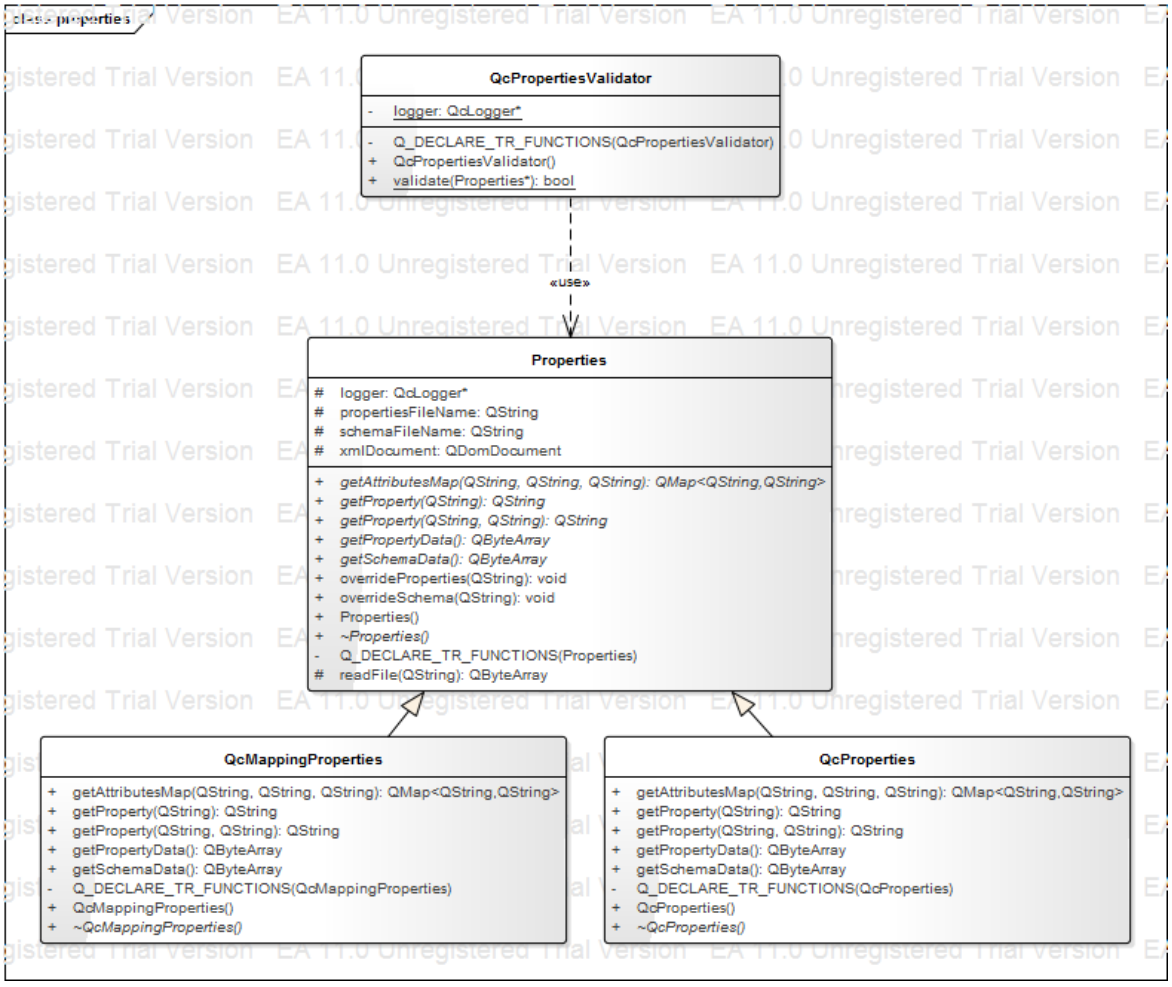
Moduł główny zajmujący się generowaniem został podzielony na logiczne części w celu wyraźnej separacji odpowiedzialności każdego modułu[1].



Rys. 5.3: Diagram klas modułu logera.

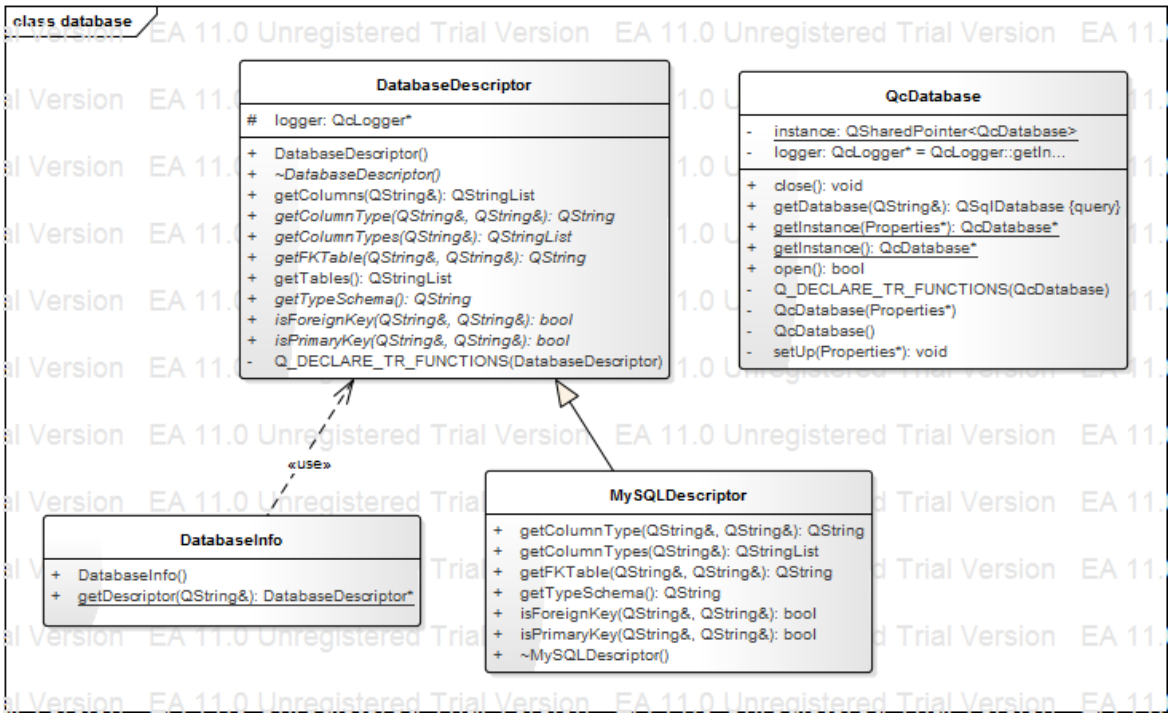
[1]Zasada pojedynczej odpowiedzialności mówi że procesy powinny być od siebie niezależne i zaimplementowane w postaci oddzielnych klas lub modułów, które komunikują się ze sobą przy pomocy publicznych interfejsów.

Moduł logujący jest używany przez wszystkie inne części systemu. Pozwala on w prosty i przejrzysty sposób śledzić przebieg procesu mapowania oraz wychwycić ewentualne błędy. Oparty on został o istniejący projekt logera o nazwie QsLog.



Rys. 5.4: Diagram klas modułu ustawień.

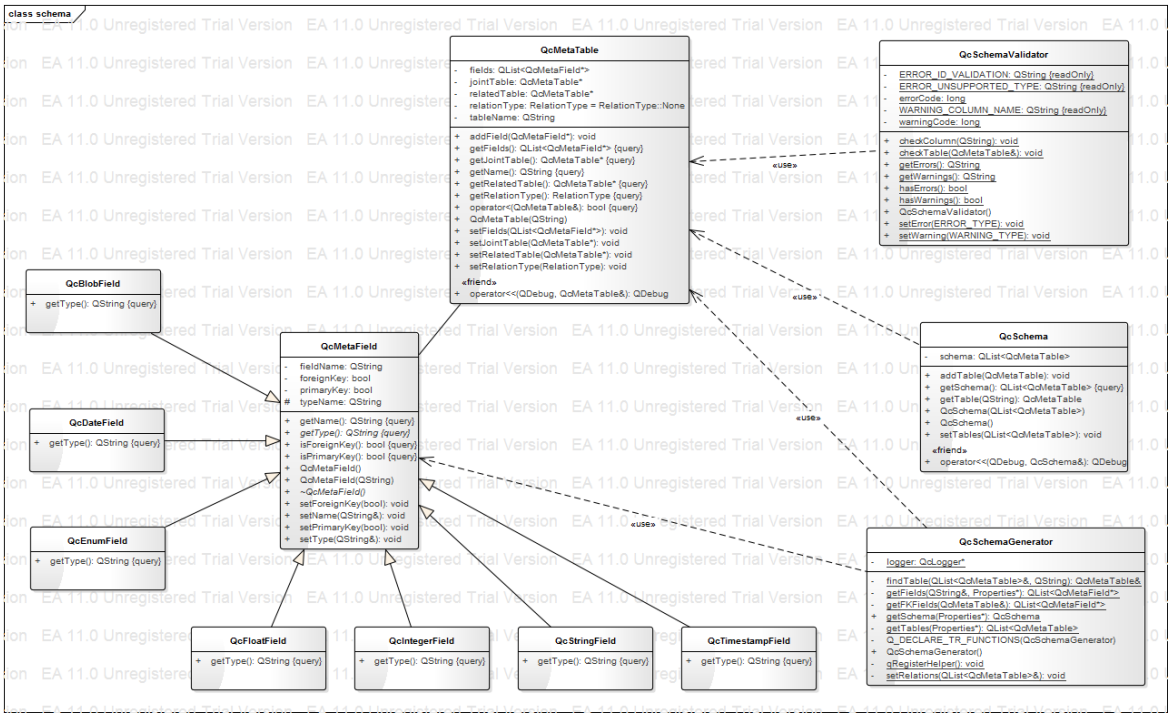
Wszelkie ustawienia potrzebne do połączenia z bazą danych oraz do przeprowadzenia procesu mapowania tabel na klasy są obsługiwane przez powyższy moduł. Udostępniony został również interfejs sprawdzający poprawność ustawień.



Rys. 5.5: Diagram klas modułu ustawień.

Moduł bazy danych pełni kilka funkcji:

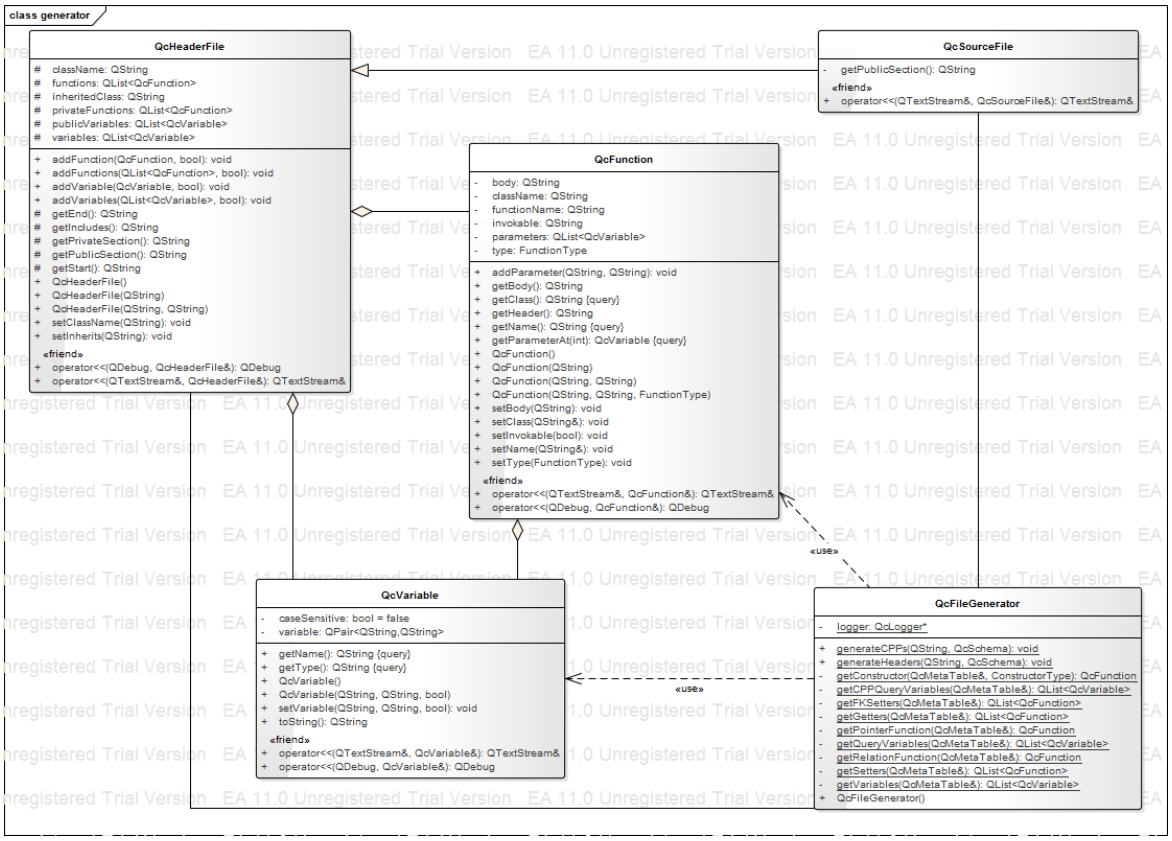
- jest odpowiedzialny za utworzenie połączenia z bazą danych i odczyt metadanych z bazy
- zbiera informacje o nazwach tabelach, typach danych w tabelach, kluczach głównych, kluczach obcych oraz powiązanych tabelach i przekazuje je do modułu zajmującego się przetwarzaniem tych informacji.



Rys. 5.6: Diagram klas modułu budowania struktury danych.

Powyższy moduł jest najważniejszą częścią aplikacji. Jego zadaniem jest przetworzenie zebranych informacji.

- tworzy w pamięci strukturę odczytanych tabel
- automatycznie rozpoznaje relacje między tabelami(jeden do jednego, jeden do wielu)
- mapuje typy kolumn bazy danych na odpowiednie typy języka
- sprawdza czy zebrane dane spełniają postawione założenia
- udostępnia interfejs generujący



Rys. 5.7: Diagram klas modułu generatora projektu.

Ostatni moduł udostępnia generator plików, który bazując na wygenerowanej strukturze danych tworzy gotowy projekt dla programisty. Wszystkie zebrane informacje są przekształcane w gotowe pliki nagłówkowe i źródłowe języka C++.

5.3 Implementacja: punkty kluczowe

W tej części pracy zostaną przedstawione kluczowe fragmenty kodu odpowiadające ze główny przepływ danych przy generowaniu opisu mapowania obiektowo-relacyjnego. Główny moduł biblioteki współdzielonej oraz przykładowa aplikacja z interfejsem graficznym wykorzystująca udostępniane przez stworzoną bibliotekę funkcje.

Do połączenia z bazą została użyta klasa zaprojektowana zgodnie z wzorcem

projektowym o nazwie singletone[2].

C++ Code 5.1: QcDatabase

```
#ifndef QCDATABASE_H
#define QCDATABASE_H

#include <QSharedPointer>
#include <QtSql/QSqlDatabase>
#include "qclogger.h"
#include "properties/properties.h"

class QcDatabase
{
    Q_DECLARE_TR_FUNCTIONS(QcDatabase)

public:
    static QcDatabase* getInstance(Properties *
       properties);
    static QcDatabase* getInstance();
    bool open();
    void close();
    QSqlDatabase getDatabase(const QString &
       connectionName = "defaultConnection")const;
private:
    QcLogger *logger = QcLogger::getInstance();
    static QSharedPointer<QcDatabase> instance;
    QcDatabase(Properties *properties);
    QcDatabase();

    void setUp(Properties *properties);
};

#endif // QCDATABASE_H
```

Kolejna klasa udostępnia interfejs zajmujący się odczytywaniem metadanych z

[2]Singletone - wzorzec projektowy ograniczający możliwość tworzenia obiektów danej klasy do jednej instancji oraz zapewniający globalny dostęp do stworzonego obiektu.

bazy. Została tutaj użyta klasa abstrakcyjna w celu umożliwienia łatwego dodania wsparcia nowych dialektów SQL.

C++ Code 5.2: QcDatabase

```
#ifndef DATABASEDESCRIPTOR_H
#define DATABASEDESCRIPTOR_H

#include <QCoreApplication>

#include "qclogger.h"

class DatabaseDescriptor
{
    Q_DECLARE_TR_FUNCTIONS(DatabaseDescriptor)

protected:
    QcLogger *logger;
public:
    explicit DatabaseDescriptor();
    virtual ~DatabaseDescriptor() {}

    QStringList getTables();
    QStringList getColumns(const QString &tableName)
        ;
    virtual QString getColumnType(const QString &
       tableName, const QString &columnName) = 0;
    virtual QStringList getColumnTypes(const QString
        &tableName) = 0;
    virtual QString getTypeSchema() = 0;
    virtual bool isPrimaryKey(const QString &
       tableName, const QString &columnName) = 0;
    virtual bool isForeignKey(const QString &
       tableName, const QString &columnName) = 0;
    virtual QString getFKTable(const QString &
       tableName, const QString &columnName) = 0;
};
```

```
#endif // DATABASEDESCRIPTOR_H
```

Przykładowa implementacja takiej klasy dla dialektu MySQL widoczna jest poniżej.

C++ Code 5.3: QcDatabase

```
#include "mysqldescriptor.h"

#include <QSqlQuery>
#include "qcdatabase.h"

QString MySQLDescriptor::getColumnType(const QString
    &tableName, const QString &columnName) {
    QSqlDatabase db = QcDatabase::getInstance()->
        getDatabase("info");
    QSqlQuery query(QString("SELECT␣data_type␣FROM␣
        information_schema.columns␣WHERE␣table_schema
        ␣='%1'␣AND␣table_name='%2'␣AND␣column_name
        ='%3'").arg(
                        QcDatabase::getInstance()->
                            getDatabase().
                            databaseName(),tableName,
                            columnName), db);

        ...

    return query.value(0).toString();
}

bool MySQLDescriptor::isPrimaryKey(const QString &
   tableName, const QString &columnName) {
    QSqlDatabase db = QcDatabase::getInstance()->
        getDatabase("info");
    QSqlQuery query(QString("SELECT␣column_key␣FROM␣
        information_schema.columns␣WHERE␣table_schema
```

```
        ␣='%1'␣AND␣table_name='%2'␣AND␣column_name
        ='%3'").arg(
                            QcDatabase::getInstance()->
                                getDatabase().
                                databaseName(),tableName,
                                columnName), db);

          ...

    return query.value(0).toString().contains("PRI")
        ;
}

bool MySQLDescriptor::isForeignKey(const QString &
   tableName, const QString &columnName) {
    QSqlDatabase db = QcDatabase::getInstance()->
        getDatabase("info");
    QSqlQuery query(QString("SELECT␣column_key␣FROM␣
        information_schema.columns␣WHERE␣table_schema
        ␣='%1'␣AND␣table_name='%2'␣AND␣column_name
        ='%3'").arg(
                            QcDatabase::getInstance()->
                                getDatabase().
                                databaseName(),tableName,
                                columnName), db);

          ...

    return query.value(0).toString().contains("MUL")
        ;
}

QString MySQLDescriptor::getFKTable(const QString &
   tableName, const QString &columnName) {
    QSqlDatabase db = QcDatabase::getInstance()->
        getDatabase("info");
    QSqlQuery query(QString("SELECT␣
```

```
        REFERENCED_TABLE_NAME␣FROM␣information_schema
        .KEY_COLUMN_USAGE␣where␣table_schema='%1'␣and
        ␣table_name='%2'␣and␣column_name='%3'").arg(
                            QcDatabase::getInstance()->
                                getDatabase().
                                databaseName(),tableName,
                                columnName), db);

          ...

    return query.value(0).toString();
}

QString MySQLDescriptor::getTypeSchema() {
    return "information_schema";
}

QStringList MySQLDescriptor::getColumnTypes(const
   QString &tableName) {
    QStringList result;
    QSqlDatabase db = QcDatabase::getInstance()->
       getDatabase("info");
    QSqlQuery query(QString("SELECT␣data_type␣FROM␣
       information_schema.columns␣WHERE␣table_schema
       ␣='%1'␣AND␣table_name='%2'").arg(
                            QcDatabase::getInstance()->
                                getDatabase().
                                databaseName(),tableName)
                                , db);

          ...

    return result;
}
```

Poniższa klasa QcMetaField jest odpowiednikiem kolumny w tabeli i przechowuje informacje o jej typie oraz właściwościach, np. czy jest kluczem głównym.

Jest to również klasa abstrakcyjna i w celu zarejestrowania nowego typu, który będzie obsługiwany przez aplikację należy rozszerzyć tą klasę, dodać wpis do funkcji w klasie QcSchemaGenerator w celu zarejestrowania klasy w systemie refleksji Qt oraz zarejestrować nowy typ w odpowiednim pliku z ustawieniami mapowania danych.

C++ Code 5.4: QcDatabase

```
#ifndef QCMETAFIELD_H
#define QCMETAFIELD_H

#include <QString>
#include <QMetaType>

class QcMetaField
{
private:
    QString fieldName;
    bool primaryKey;
    bool foreignKey;
protected:
    QString typeName;
public:
    QcMetaField();
    QcMetaField(QString fieldName);
    virtual ~QcMetaField() {}

    void setName(const QString &fieldName);
    void setForeignKey(bool fk);
    void setPrimaryKey(bool pk);
    void setType(const QString &typeName);

    virtual QString getType()const = 0;
    QString getName()const;
    bool isPrimaryKey()const;
    bool isForeignKey()const;
};
```

```
#endif // QCMETAFIELD_H
```

Jedna z najważniejszych klas modułu generatora to klasa QcSchemaGenerator, która zajmuje się przetworzeniem informacji pobranych z bazy danych na odpowiednią strukturę danych przechowywaną w pamięci. Główna funkcja getTables jest odpowiedzialna za odczytanie tabel, nazw i typów kolumn oraz rozpoznanie części relacji między tabelami. W trakcie przetwarzania wykonywana jest również walidacja danych.

C++ Code 5.5: QcDatabase

```
QList<QcMetaTable> QcSchemaGenerator::getTables(
    Properties *properties) {
    QList<QcMetaTable> result;
    DatabaseDescriptor *descriptor = DatabaseInfo::
        getDescriptor(QcDatabase::getInstance()->
        getDatabase().driverName());

    for(QString tableName : descriptor->getTables())
        {
        if(QcSchemaValidator::hasErrors()) {
            return result;
        }
        QcMetaTable table(tableName);
        logger->debug("Mapping table: " + tableName)
            ;
        table.setFields(getFields(tableName,
            properties));
        QcSchemaValidator::checkTable(table);
        result.append(table);
    }

    setRelations(result);

    for(QcMetaTable table : result) {
        switch(table.getRelationType()) {
        case(RelationType::OneToMany):
```

```
                logger->debug("One to many");
                logger->debug(table.getRelatedTable()->
                    getName());
                break;
            case(RelationType::ManyToMany):
                logger->debug("Many to many");
                logger->debug(table.getRelatedTable()->
                    getName());
                logger->debug(table.getJointTable()->
                    getName());
            }
        }

        return result;
}

QList<QcMetaField*> QcSchemaGenerator::getFields(
    const QString &tableName, Properties *properties)
    {
        qRegisterHelper<QcMetaField>();

        QList<QcMetaField*> result;
        DatabaseDescriptor *descriptor = DatabaseInfo::
            getDescriptor(QcDatabase::getInstance()->
            getDatabase().driverName());
        QMap<QString,QString> typeMapping = properties->
            getAttributesMap("map","fromType","toType");

        for(QString columnName : descriptor->getColumns(
            tableName)) {
            QcSchemaValidator::checkColumn(columnName);
            QString columnType = descriptor->
                getColumnType(tableName,columnName);
            QString columnTypeMap = typeMapping[
                columnType];
            int id = QMetaType::type(columnTypeMap.
                toStdString().c_str());
```

```
        logger->debug("Mapping field: '" +
            columnType + "' to type '" +
            columnTypeMap + "'");
        if(id == 0) {
            logger->error("Type: '" + columnType + "
                ' is not supported");
            QcSchemaValidator::setError(ERROR_TYPE::
                UnsupportedType);
            break;
        }
        QcMetaField *field = static_cast<QcMetaField
            *>(QMetaType::create(id));
        logger->debug("Setting column name: " +
            columnName);
        field->setName(columnName);
        if(descriptor->isPrimaryKey(tableName,
            columnName)) {
            logger->debug("Column: " + columnName +
                " is primary key.");
            field->setPrimaryKey(true);
        }

        if(descriptor->isForeignKey(tableName,
            columnName)) {
            logger->debug("Column: " + columnName +
                " is foreign key.");
            field->setType(descriptor->getFKTable(
                tableName,columnName));
            field->setForeignKey(true);
        }

        result.append(field);
    }

    return result;
}
```

```
void QcSchemaGenerator::setRelations(QList<
    QcMetaTable> &tables) {
  logger->debug("Setting up relations");
  for(QcMetaTable &table : tables) {
      QList<QcMetaField*> fkFields = getFKFields(
          table);
      if(fkFields.size() == 1) {
          logger->debug("Found one to many
              relation");
          QcMetaTable &relationTable = findTable(
              tables, fkFields.at(0)->getName());
          logger->debug("Table: " + table.getName
              () + " got fk to: " + relationTable.
              getName() + " table");
          relationTable.setRelationType(
              RelationType::OneToMany);
          relationTable.setRelatedTable(&table);
      }

      if(fkFields.size() == 2) {
          logger->debug("Found many to many
              relation");
          QcMetaTable &leftSide = findTable(tables
              ,fkFields.at(0)->getName());
          QcMetaTable &rightSide = findTable(
              tables,fkFields.at(1)->getName());
          QStringList split = table.getName().
              split("_");
          if(split.size() == 2 &&
                  (leftSide.getName().compare(
                      split.at(0)) == 0 ||
                      rightSide.getName().compare(
                      split.at(0)) == 0) &&
                  (leftSide.getName().compare(
                      split.at(1)) == 0 ||
                      rightSide.getName().compare(
                      split.at(1)) == 0)) {
```

```
                logger->debug("Left: " + leftSide.
                    getName() + " Center: " + table.
                    getName() + " Right: " +
                    rightSide.getName());
                leftSide.setRelationType(
                    RelationType::ManyToMany);
                leftSide.setRelatedTable(&rightSide)
                    ;
                leftSide.setJointTable(&table);

                rightSide.setRelationType(
                    RelationType::ManyToMany);
                rightSide.setRelatedTable(&leftSide)
                    ;
                rightSide.setJointTable(&table);
            }

        }
    }
}
```

Na podstawie utworzonej struktury danych generowane są pliki projektu. Odpowiada za to klasa QcFileGenerator.

C++ Code 5.6: QcDatabase

```
#ifndef QCFILEGENERATOR_H
#define QCFILEGENERATOR_H

#include "schema/qcschema.h"
#include "schema/qcmetatable.h"
#include "qclogger.h"
#include "qcfunction.h"

#include <QString>
```

```
#include <QList>

class QcFileGenerator
{
private:
    static QcLogger *logger;

    static QcFunction getConstructor(QcMetaTable &
       table, ConstructorType type);
    static QList<QcFunction> getGetters(QcMetaTable
       &table);
    static QList<QcFunction> getSetters(QcMetaTable
       &table);
    static QList<QcVariable> getVariables(
       QcMetaTable &table);
    static QList<QcVariable> getQueryVariables(
       QcMetaTable &table);
    static QList<QcVariable> getCPPQueryVariables(
       QcMetaTable &table);
    static QList<QcFunction> getFKSetters(
       QcMetaTable &table);
    static QcFunction getPointerFunction(QcMetaTable
        &table);
    static QcFunction getRelationFunction(
       QcMetaTable &table);
public:
    QcFileGenerator() = delete;
    static void generateHeaders(QString dirPath,
       QcSchema schema);
    static void generateCPPs(QString dirPath,
       QcSchema schema);
};

#endif // QCFILEGENERATOR_H
```

Poniżej przedstawiony jest częściowy kod prostej aplikacji graficznej opartej o funkcje udostępniane przez stworzoną bibliotekę. Odczytuje ona odpowiednie

pliki ustawień, łączy się z bazą danych, sprawdza ustawienia, a następnie generuje projekt wykorzystywany przez moduł mapowania obiektowo-relacyjnego.

C++ Code 5.7: QcDatabase

```
// ==================== SLOTS ====================
    //

void QcGenerator::loadDBProperty() {
    QFileInfo propertyFile(QC_PROPERTIES_DIR + QDir
        ::separator() + QC_PROPERTIES_FILE);
    if(!propertyFile.exists()) {
        QMessageBox::critical(this,"Property load
            error","Could not find " + QString(
            QC_PROPERTIES_FILE) + " in resources
            folder.");
        ui->actionConnect->setEnabled(false);
        return;
    }

    properties = new QcProperties();

    if(!QcPropertiesValidator::validate(properties))
        {
        QMessageBox::critical(this,"Property
            validation error", "Validation of " +
            QString(QC_PROPERTIES_FILE) + " failed.")
            ;
        ui->actionConnect->setEnabled(false);
        return;
    }

    ui->usernameEdit->setText(properties->
        getProperty("username"));
    ui->passwordEdit->setText(properties->
        getProperty("password"));

    ui->dbNameEdit->setText(properties->getProperty(
```

```
        "database","name"));

    ui->dbHostEdit->setText(properties->getProperty(
        "host","address"));
    ui->dbPortEdit->setText(properties->getProperty(
        "host","port"));

    QMessageBox::information(this,"Validation␣
        successfull", "Properties␣and␣schema␣
        validated␣successfully.");
    ui->actionConnect->setEnabled(true);
}

void QcGenerator::loadMappingProperty() {
    QFileInfo mappingFile(QC_PROPERTIES_DIR + QDir::
        separator() + QC_MAPPING_FILE);
    if(!mappingFile.exists()) {
        QMessageBox::critical(this,"Property␣load␣
            error","Could␣not␣find␣" + QString(
            QC_MAPPING_FILE) + "␣in␣resources␣folder.
            ");
        ui->actionGenerate->setEnabled(false);
        return;
    }

    mapping = new QcMappingProperties();

    if(!QcPropertiesValidator::validate(mapping)) {
        QMessageBox::critical(this,"Mapping␣
            validation␣error", "Validation␣of␣" +
            QString(QC_PROPERTIES_FILE) + "␣failed.")
            ;
        ui->actionGenerate->setEnabled(false);
        return;
    }

    QMessageBox::information(this,"Validation␣
```

```
        successfull", "Mapping␣and␣schema␣validated␣
        successfully.");
    ui->actionGenerate->setEnabled(true);
}

void QcGenerator::connectToDB() {
    db = QcDatabase::getInstance(properties);
    if(!db->open()) {
        QMessageBox::critical(this,"Database␣error",
            "Could␣not␣connect␣to␣database.");
        return;
    }

    QMessageBox::information(this,"Database","
        Successfully␣connected␣to␣database.");
}

void QcGenerator::generate() {
    QcSchema schema = QcSchemaGenerator::getSchema(
        mapping);

    if(!QcSchemaValidator::hasErrors()) {

        if(QcSchemaValidator::hasWarnings()) {
            QMessageBox::warning(this, "Warning",
                QcSchemaValidator::getWarnings());
        }

        QString dir = QFileDialog::
            getExistingDirectory(this, tr("Open␣
            directory"),"",QFileDialog::ShowDirsOnly
            | QFileDialog::DontResolveSymlinks);

        QcFileGenerator::generateHeaders(dir,schema)
            ;
        QcFileGenerator::generateCPPs(dir,schema);
```

```
            QMessageBox::information(this,"Generation","
                Generating completed.");
        } else {
            QMessageBox::critical(this,"Error",
                QcSchemaValidator::getErrors());
        }

}
```

5.4 Testy stworzonej aplikacji

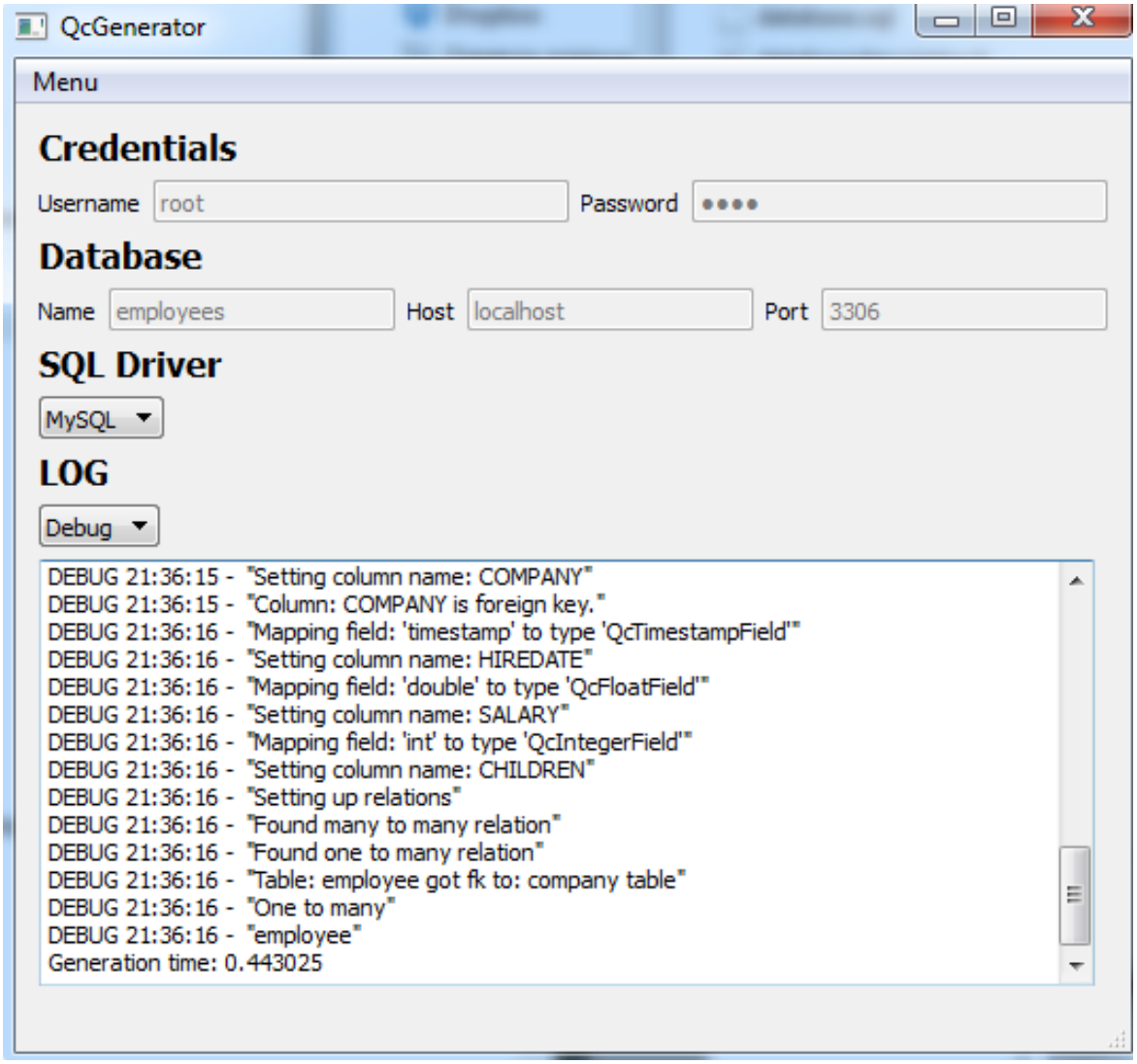
5.4.1 Testy wydajności

Test wydajności został przeprowadzony w porównaniu z aplikacją JOOQ napisaną w języku Java. Porównany został czas potrzebny na wygenerowanie plików gotowego projektu. Użyta została baza danych opisana w rozdziale 5.2.1. Oto logi przebiegu procesu generowania wraz z czasami.

Listing 5.8: JOOQ Log

```
...
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Enums fetched
: 0 (0 included, 0 excluded)
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating table
: Company.java [input=company, output=company, pk
=KEY_company_PRIMARY]
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating table
: Department.java [input=department, output=depar
tment, pk=KEY_department_PRIMARY]
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating table
: Employee.java [input=employee, output=employee,
```

```
 pk=KEY_employee_PRIMARY]
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Tables generated                : Total: 619.231ms
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating table references
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Table refs generated
: Total: 623.451ms, +4.22ms
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating Keys
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Keys generated
: Total: 631.825ms, +8.373ms
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating table records
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating record
: AssignmentRecord.java
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating record              : CompanyRecord.java
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating record
: DepartmentRecord.java
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Generating record              : EmployeeRecord.java
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Table records generated
: Total: 666.048ms, +34.222ms
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Routines fetched
: 0 (0 included, 0 excluded)
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: Packages fetched
: 0 (0 included, 0 excluded)
sie 17, 2014 9:34:20 PM org.jooq.tools.JooqLogger info
INFO: GENERATION FINISHED!
: Total: 681.673ms, +15.624ms
```



Rys. 5.8: Interfejs aplikacji przykładowej wraz z logiem procesu generowania.

Framework	Czas generowania	Poprawa wydajności(w %)
Qubic	443.025 ms	-
JOOQ	681.673 ms	35

Tab. 5.1: Porównanie czasów generowania projektu z istniejącej bazy danych.

5.4.2 Porównanie funkcjonalności

Porównanie funkcjonalności samego generatora opisu mapowania obiektowo-relacyjnego opiera się na sprawdzeniu wsparcia różnych dialektów SQL. Porównane zostały jedynie najpopularniejsze dialekty.

Framework	Qubic	JOOQ	Hibernate	Funkcjonalność
	tak	tak	tak	Wsparcie MySQL
	tak	tak	tak	Wsparcie PostgreSQL
	tak	tak	nie	Wsparcie SQLite
	tak	tak	tak	Wsparcie Oracle
	tak	nie	tak	Wsparcie MSSQL
	tak	tak	tak	Wsparcie Sybase
	tak	tak	tak	Wsparcie IBM DB2

Tab. 5.2: Porównanie funkcjonalności wybranych aplikacji zajmujących się generowaniem opisu mapowania obiektowo-relacyjnego.

Rozdział 6

Podsumowanie

Ten rozdział zawiera podsumowanie pracy oraz uzyskanych wyników. Omówione zostają dodatkowe możliwości rozwoju tematu jak i całej aplikacji Qubic.

6.1 Dyskusja wyników

Przeprowadzone testy generowania warstwy dostępu do danych aplikacji wykazały około 35% wzrost wydajności na korzyść zaproponowanego rozwiązania. Należy tu zaznaczyć, że w języku C++ nie znaleziono podobnych aplikacji, a różnice wydajności wynikają częściowo z budowy języka Java, w którym zostały napisane podobne aplikacje. Dodatkowym atutem stworzonej aplikacji jest wsparcie większej ilości dialektów SQL od konkurencyjnych rozwiązań oraz brak istnienia podobnych aplikacji w języku C++, dzięki czemu Qubic staje się bezkonkurencyjnym rozwiązanie w swojej dziedzinie. Prostota użycia, brak zależności od zewnętrznych aplikacji czy dodatkowego ręcznego konfigurowania to tylko kilka z wielu atutów stworzonego generatora opisu mapowania obiektowo-relacyjnego.

6.2 Ocena możliwości wdrożenia Qubica

Qubic pozwala zaoszczędzić programistom dużą ilość czasu w fazie tworzenia aplikacji, dzięki czemu mogą sie oni skupić na innych częściach projektu. Moduł generujący warstwę dostępu do danych jest prosty i szybki w użyciu oraz dalszym rozwoju. W razie potrzeby jest prosto konfigurowalny, a dzięki generycznej budowie aplikacja może być łatwo dostosowywana do własnych potrzeb. Dzięki temu jej wdrożenie nie wiąże się z dużymi kosztami, a pozwala na późniejsze zaoszczędzenie cennego czasu w fazie implementacyjnej.

6.3 Perspektywy dalszego rozwoju

Dodatkowym atutem stworzonej aplikacji byłby moduł zajmujący się procesem odwrotnym, czyli generowaniem bazy danych z istniejącego schematu obiektów klas lub pliku tworzącego bazę danych. Pozwoliło by to na pewną swobodę programistom, których zadaniem jest stworzenie bazy, a łatwiejsze jest dla nich stworzenie struktury w języku programowania niż bezpośrednie tworzenie bazy danych używając język SQL.

Bibliografia

[1] M. Keith, M. Schincariol, Pro EJB 3 Java persistence API. 2006. ISBN-13 978-1-59059-645-6

[2] C. Bauer, G. King, Hibernate w akcji, 2007, ISBN: 978-83-246-0527-9

[3] A. Ezust, P. Ezust, Introduction to Design Patterns in C++ with Qt ISBN 978-0-13-282645-7

[4] D. Gennaro Advanced C++ Metaprogramming ISBN-13 978-1460966167

[5] P. Wilton, J. Colby, Beginning SQL ISBN 0-7645-7732-8

[6] http://dev.mysql.com/doc/refman/5.6/en/ - [dostęp 05.08.2014]

[7] http://www.stroustrup.com/C++11FAQ.html - [dostęp 10.08.2014]

[8] http://qt-project.org/ - [dostęp 07.01.2014]

[9] http://msdn.microsoft.com/en-us/library/windows/desktop/ms681914(v=vs.85).aspx - [dostęp 29.07.2014]

[10] http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ - [dostęp 02.08.2014]

[11] http://www.jooq.org/javadoc/3.4.x/ - [dostęp 02.08.2014]

[12] http://msdn.microsoft.com/en-us/library/ff649643.aspx - [dostęp 28.07.2014]

[13] http://www.oracle.com/technetwork/java/dataaccessobject-138824.html - [dostęp 28.07.2014]

[14] http://msdn.microsoft.com/en-us/library/ee658127.aspx - [dostęp 27.07.2014]

[15] http://www.visual-paradigm.com/VPGallery/img/orm/Overview/ORM-Overview.png – [dostęp 02.11.2013]

[16] http://blog.famzah.net/2010/07/01/cpp-vs-python-vs-perl-vs-php-performance-benchmark/ - [dostęp 02.07.2014]

[17] http://webroad.pl/wp-content/uploads/2012/12/images_old_mvs-schemat.jpg - [dostęp 28.07.2014]

[18] http://www.oracle.com/ocom/groups/public/@otn/documents/digitalasset/146804.jpg - [dostęp 28.07.2014]

[19] http://i.msdn.microsoft.com/dynimg/IC34006.gif - [dostęp 29.07.2014]

Spis rysunków

Spis tabel