



Politechnika Łódzka  
Instytut Informatyki

## INŻYNIERSKA PRACA DYPLOMOWA

# GENERATOR OPISU MAPOWANIA OBIEKTOWO- RELACYJNEGO W C++

**Wydział:** Fizyki Technicznej, Informatyki i Matematyki Stosowanej  
**Promotor:** dr inż. Arkadiusz Tomczyk  
**Dyplomant:** Sebastian Florek  
**Nr albumu:** 165397  
**Kierunek:** Informatyka  
**Specjalność:** Inżynieria Oprogramowania i Analiza Danych

Łódź, 7 września 2014r.

Instytut Informatyki

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

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>4</b>
1.1	Problematyka i zakres pracy . . . . .	4
1.2	Metoda badawcza . . . . .	5
1.3	Przegląd literatury w dziedzinie . . . . .	6
1.4	Układ pracy . . . . .	7
<b>2</b>	<b>Generator opisu mapowania obiektowo-relacyjnego w C++</b>	<b>9</b>
2.1	Podstawowe definicje . . . . .	9
2.1.1	Warstwa dostępu do danych . . . . .	9
2.1.2	Normalizacja bazy danych . . . . .	10
2.1.3	SQL . . . . .	10
2.1.4	Mapowanie Obiektowo-Relacyjne . . . . .	10
2.1.5	Wzorzec projektowy . . . . .	11
2.1.6	Wzorzec architektoniczny Model-Widok-Kontroler . . . . .	12
2.1.7	Komponent Data Access Object . . . . .	12
2.1.8	Biblioteka współdzielona . . . . .	13
2.1.9	Aplikacja szkieletowa Qt . . . . .	13
<b>3</b>	<b>Analiza istniejących aplikacji do generowania opisu mapowania obiektowo-relacyjnego</b>	<b>14</b>
3.1	Istniejące aplikacje . . . . .	14
3.2	Wymagania aplikacji . . . . .	14
3.3	Hibernate . . . . .	15
3.4	JOOQ . . . . .	16
3.5	QxEntityEditor . . . . .	16
3.6	Wady i zalety istniejących rozwiązań . . . . .	17
<b>4</b>	<b>Projekt aplikacji szkieletowej Qubic</b>	<b>19</b>
4.1	Analiza wymagań . . . . .	19

4.1.1	Studium możliwości . . . . .	19
4.1.2	Wymagania funkcjonalne . . . . .	19
4.1.3	Ograniczenia projektu . . . . .	20
4.2	Użyte technologie . . . . .	21
4.2.1	Język programowania . . . . .	21
4.2.2	System baz danych . . . . .	21
4.2.3	Środowisko deweloperskie . . . . .	22
4.2.4	Biblioteki . . . . .	22
4.2.5	Inne narzędzia . . . . .	22
4.3	Projekt . . . . .	23
4.3.1	Projekt warstwy danych . . . . .	23
4.3.2	Projekt aplikacji . . . . .	24
4.4	Implementacja: punkty kluczowe . . . . .	33
4.5	Opis użytkowania aplikacji . . . . .	39
4.6	Przykład użycia generatora opisu . . . . .	40
4.7	Możliwości rozszerzania aplikacji . . . . .	45
<b>5</b>	<b>Podsumowanie</b>	<b>48</b>
5.1	Dyskusja wyników . . . . .	48
5.2	Ocena możliwości wdrożenia Qubica . . . . .	48
5.3	Perspektywy dalszego rozwoju . . . . .	49
	<b>Bibliografia</b>	<b>49</b>
	<b>Spis rysunków</b>	<b>51</b>
	<b>Spis tabel</b>	<b>52</b>

# Rozdział 1

## Wstęp

### 1.1 Problematyka i zakres pracy

Wzorzec architektoniczny typu Model View Controller <sup>1</sup> 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 <sup>2</sup> 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 danych aplikacji, m. in.: aplikacje takie jak Hibernate czy JOOQ stworzone

---

<sup>1</sup>Model View Controller (MVC) - wzorzec służący do organizowania struktury aplikacji [13].

<sup>2</sup>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].

w języku Java lub QxEntityEditor napisany w języku C++, jednakże ich niska skuteczność i wydajność związane z wieloma zapytaniem do bazy oraz czasem konfiguracji sprawiają, że szukamy czegoś innego. Większość tego typu narzędzi bardziej skupia się na udostępnieniu jak największej funkcjonalności, przez co zaniedbują 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 developerom na pracę z konkretnymi obiektami, bez praktycznej znajomości języków typu SQL<sup>3</sup> do obsługi baz danych.

Aplikacja została podzielona na dwa moduły i będzie tworzona przez dwie osoby. Pierwszy moduł będzie zajmował się generowaniem opisu relacyjnej bazy danych w postaci plików klas języka C++ i zostanie opisany w niniejszej pracy. Moduł generatora opisu mapowania obiektowo-relacyjnego będzie udostępniony jako biblioteka współdzielona. Dzięki takiemu rozwiązaniu moduły będą mogły być używane niezależnie. Drugi moduł zostanie opisany przez kolegę Marcina Maciaszczyka w pracy pod tytułem: „Mapowanie Obiektowo-Relacyjne w języku C++”. Będzie on 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 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

---

<sup>3</sup>SQL (ang. Structured Query Language) - język programowania stworzony do zarządzania danymi, które trzymane są w relacyjnych bazach danych [5].

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, 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.

## 1.3 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 aplikacji szkieletowej Qt**

Aplikacja szkieletowa 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].

## **1.4 Układ pracy**

Tematem pracy jest stworzenie generatora 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 są w nim różne problemy związane z aplikacjami generującymi warstwę danych aplikacji, 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 zagadnienia dotyczące prezentowanego tematu wraz z krótkim opisem użytych źródeł.

W rozdziale 2 przybliżona zostaje tematyka tworzenia aplikacji opartych o warstwę. 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.

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. Pokazane zostaje wykorzystanie wzorców projektowych na etapie tworzenia aplikacji. 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-relacyjnego 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 klasa 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. Zazwyczaj jest to realizowane

poprzez nadpisanie odpowiednich klas. Tego typu praktyki są stosowane często właśnie w narzędziach zajmujących się mapowaniem obiektowo-relacyjnym [15].

### 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

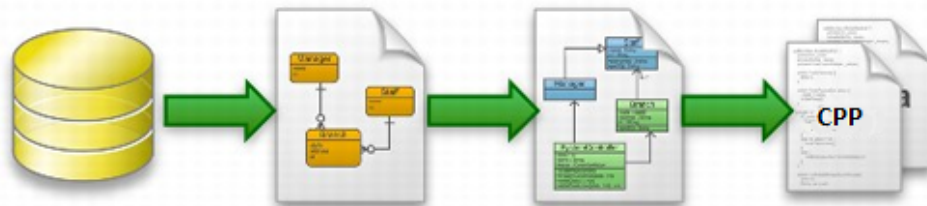
SQL jest językiem programowania stworzonym do zarządzania danymi, które są trzymane w relacyjnych bazach danych. Głównym zadaniem tego języka jest dodawanie, usuwanie, odczyt oraz aktualizacja danych. Na bazie języka SQL zostało stworzonych wiele systemów zarządzania relacyjnymi bazami danych oraz nowych 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 problemu różnej budowy tych systemów 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



Rysunek 2.1: Uproszczony przebieg generowania opisu mapowania obiektowo-relacyjnego [16].

### 2.1.5 Wzorzec projektowy

Wzorce projektowe są wydajnym i eleganckim sposobem na rozwiązanie najczęściej spotykanych problemów przy tworzeniu aplikacji obiektowych. Zostały one spopularyzowane dzięki twórcom książki „Inżynieria oprogramowania: Wzorce projektowe”. Erich Gamma, Richard Helm, Ralph Johnson oraz John Vlissides nazywani także „Bandą Czterech” przeanalizowali w niej 23 wzorce [3]. 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:

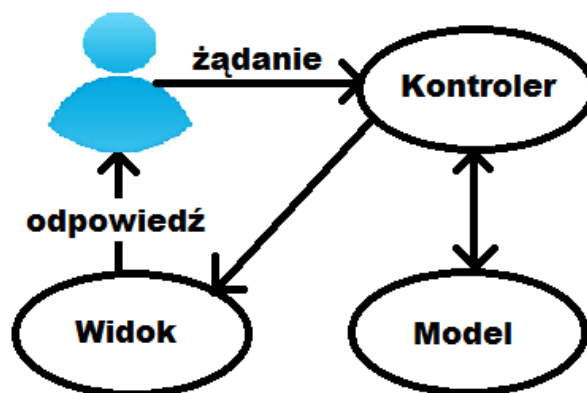
- Wzorce konstrukcyjne - opisujące proces tworzenia nowych obiektów. Ich zadaniem jest tworzenie, inicjalizacja oraz konfiguracja obiektów.
- Wzorce strukturalne - opisujące struktury powiązanych ze sobą obiektów.
- Wzorce czynnościowe - opisujące zachowanie i powiązania współpracujących ze sobą obiektów.

### 2.1.6 Wzorzec architektoniczny Model-Widok-Kontroler

Wzorzec architektoniczny 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 warstwę 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.



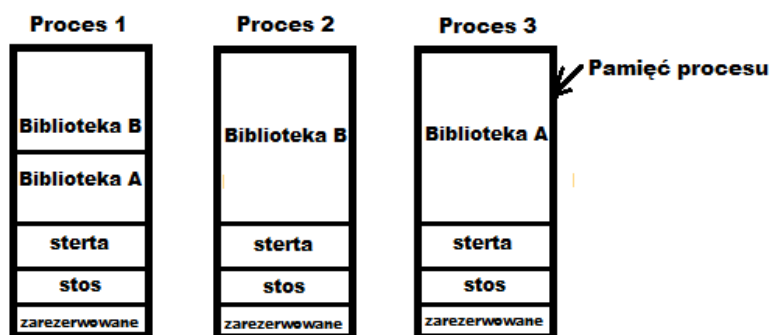
Rysunek 2.2: Wzorzec Model-Widok-Kontroler.

### 2.1.7 Komponent Data Access Object

Data Access Object jest komponentem, którego zadaniem jest dostarczanie jednolitego interfejsu do komunikacji między aplikacją, a źródłem danych (np. bazą danych czy plikiem). Dostarcza on 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 [14].

### 2.1.8 Biblioteka współdzielona

Biblioteka współdzielona jest ładowana tylko raz do pamięci systemu. Każdy proces który załaduje taką 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].



Rysunek 2.3: Schemat działania biblioteki współdzielonej [18].

### 2.1.9 Aplikacja szkieletowa Qt

Aplikacja szkieletowa 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 narzędzi
- Zintegrowane środowisko deweloperskie

Główną zaletą używania Qt jest możliwość pisania kodu skierowanego 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 wspiera refleksję<sup>1</sup> [8].

<sup>1</sup>Mechanizm refleksji - pozwala na modyfikację działania programu w trakcie jego wykonania. Zachowanie funkcji lub metod wcześniej zdefiniowanych może ulec zmianie w trakcie działania programu.

## **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 strony oraz wymagania.

### **3.1 Istniejące aplikacje**

W języku C++ liczba aplikacji zajmujących się generowaniem opisu mapowania obiektowo-relacyjnego jest znikoma. Jedyną wartą uwagi biblioteka to QxORM, a w zasadzie specjalny generator zbudowany na jej podstawie o nazwie QxEntityEditor. W języku Java istnieje kilka aplikacji zajmujących się generowaniem opisu mapowania obiektowo-relacyjnego. Są nimi: JOOQ [11] czy Hibernate [10]. Obie aplikacje zajmują się mapowaniem obiektowo-relacyjnym i posiadają funkcjonalność generowania warstwy dostępu do danych z istniejących systemów relacyjnych baz danych.

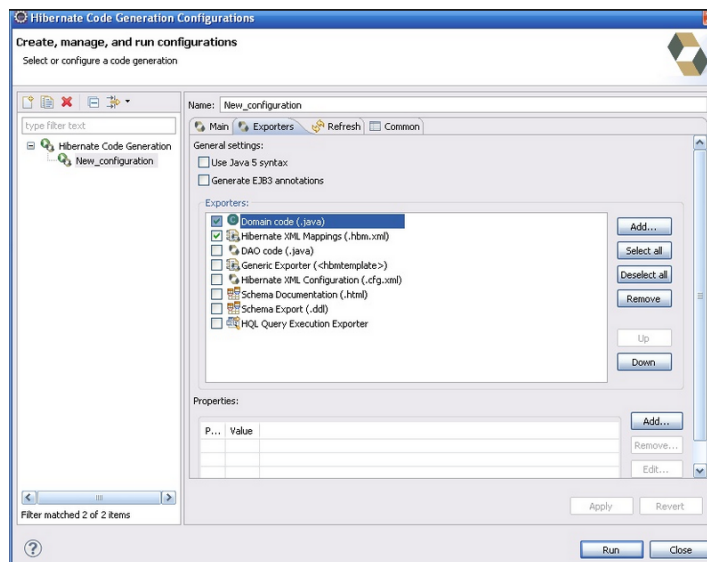
### **3.2 Wymagania aplikacji**

W przeciwieństwie do analizowanych aplikacji Javowych QxEntityEditor został stworzony jako całkowicie niezależna aplikacja, jednak aby generowanie kodu było możliwe wymaga on dostępu do biblioteki QxORM. Pozostałe 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. Jedynym wymogiem oprócz posiadania odpowiedniego środowiska, z którego wygenerowany zostanie projekt na podstawie bazy danych, jest stworzenie odpowiedniego pliku konfiguracyjnego.

### 3.3 Hibernate

Hibernate udostępnia możliwość mapowania modelu danych na relacyjną bazę danych. Dostarcza on również dodatkowe narzędzia, które pozwalają ten proces odwrócić. Są to odpowiednie wtyczki do środowisk programistycznych takich jak Eclipse czy IntelliJ. Udostępniony interfejs jest prosty w obsłudze, a konfiguracja wymaga jedynie dostarczenia odpowiedniego pliku z informacjami o połączeniu. Wadą jest niestety brak możliwości zdefiniowania przez użytkownika dodatkowych informacji, np. dotyczących relacji między tabelami. Dużym plusem jest to, że wszystkie te narzędzia są dostępne za darmo na licencji LGPL<sup>1</sup>



Rysunek 3.1: Interfejs generowania kodu frameworka Hibernate.

<sup>1</sup>LGPL - licencja wolnego oprogramowania. Pozwala między innymi na używanie oprogramowania objętego taką licencją w komercyjnych projektach, bez konieczności zakupu licencji.

## 3.4 JOOQ

Jest stosunkowo nowym i lekkim narzędziem do mapowania obiektowo-relacyjnego. Głównym założeniem twórców było postawienie SQL na pierwszym miejscu. Zamiast tworzyć własny język do generowania zapytań, skupili się na prostym i szybkim generowaniu zapytań SQL. Nie udostępnia on dużej funkcjonalności jak na przykład Hibernate jednak dzięki temu oferuje szybkość i prostotę użycia. Generowanie kodu odbywa się niestety z linii poleceń, jako że nie udostępnia on żadnego graficznego interfejsu. Generowanie odbywa się jednostronnie, czyli z istniejącej bazy danych do projektu języka Java. W celu wygenerowania projektu należy przygotować wcześniej odpowiedni plik ustawień i wywołać odpowiednią klasę.

```
java org.jooq.util.GenerationTool /jooq-config.xml
```

Niestety biblioteka JOOQ w darmowej ofercie ma dość ograniczone możliwości. W celu użycia jej do średnich i większych projektów wymagane jest wykupienie wersji profesjonalnej lub dla firm. W darmowej wersji nie obsługuje ona zbyt dużej ilości baz danych.

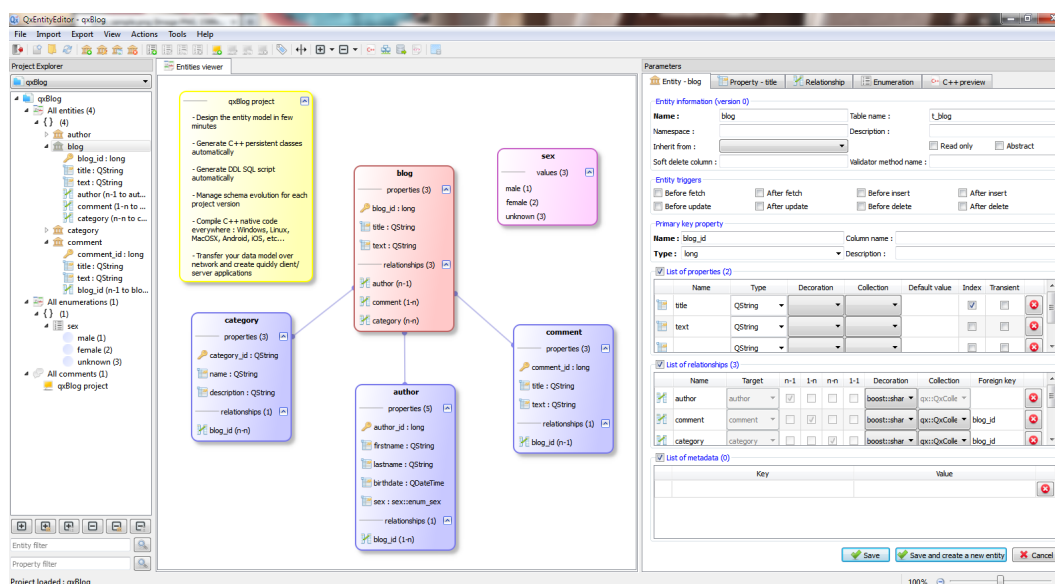
## 3.5 QxEntityEditor

QxEntityEditor jest graficznym edytorem przeznaczonym dla biblioteki QxORM. Pozwala on zarządzać modelem danych w bardzo szerokim zakresie, np.:

- Pozwala generować klasy C++ na podstawie bazy danych przeznaczone do używania z biblioteką QxORM.
- Pozwala generować skrypty tworzące bazę danych.
- Wspomaga tworzenie aplikacji typu klient/serwer.
- Udostępnia własny interfejs modelowania bazy danych. Nie wymaga jej wcześniejszego istnienia w celu stworzenia bazy oraz projektu.

Użytkowanie aplikacji niestety nie jest zbyt proste ze względu na bardzo dużą funkcjonalność i dość niejasne błędy. Próby wygenerowania kodu z bazy MySQL niestety zakończyły się niepowodzeniem. Sama aplikacja jest płatna i nie udostępnia swoich źródeł co znacznie ogranicza jej dostępność dla zwykłego programisty.





Rysunek 3.2: Interfejs aplikacji QxEntityEditor.

### 3.6 Wady i zalety istniejących rozwiązań

QxEntityEditor	JOOQ	Hibernate	
tak	tak	tak	Wsparcie dla popularnych systemów baz danych
duża	mała	duża	Dostępna funkcjonalność
średnia	dobra	dobra	Przejrzystość generowanego kodu
pojedyncza licencja 300\$	od 320\$ do 640\$ w zależności od licencji	darmowy	Cena
duży	średni	średni	Stopień trudności konfiguracji

Tablica 3.1: Porównanie wybranych aplikacji zajmujących się generowaniem opisu mapowania obiektowo-relacyjnego.

Podsumowując powyższą tabelę można stwierdzić, że każde z istniejących rozwiązań ma pewne wady lub braki. Z aplikacji przeznaczonych dla języka Java

Hibernate jest zdecydowanym zwycięzcą jeśli chodzi o funkcjonalność, prostotę i cenę narzędzia do generowania opisu mapowania obiektowo-relacyjnego. JOOQ z drugiej strony swą prostotą użycia sprawia, że dla prywatnych użytkowników jest zdecydowanym liderem. Jeśli chodzi natomiast o język C++, tutaj jedynym kandydatem jest QxEntityEditor. Posiada on dużą funkcjonalność, jednak cena jaką należy zapłacić zdecydowanie odstrasza zwykłych użytkowników. Sprawia to, że staje się on narzędziem przeznaczonym raczej dla firm.

## Rozdział 4

# Projekt aplikacji szkieletowej Qubic

### 4.1 Analiza wymagań

W tej części zostanie przedstawiona analiza oraz część implementacyjna aplikacji Qubic. W pierwszej części są zdefiniowane wymagania aplikacji oraz opisane wymagania funkcjonalne wraz z ograniczeniami 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 implementacyjną. Przybliżone zostają najważniejsze części kodu wraz z opisem. Ostatnim krokiem są testy i ocena funkcjonalności stworzonej aplikacji.

#### 4.1.1 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.

#### 4.1.2 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. Wszelkie ustawienia powinny być w prosty sposób konfigurowalne, tak by zmiana bazy danych nie wymuszała zmian kodu aplikacji.
5. Powinna generować dodatkowe funkcje pozwalające wyciągnąć powiązane dane w oparciu o relacje między tabelami.
6. Aplikacja powinna wspierać wiele systemów baz danych lub udostępniać łatwy sposób dodania ich wsparcia.
7. Aplikacja powinna w prosty sposób definiować mapowanie typów danych z bazy danych na typy języka.

### 4.1.3 Ograniczenia projektu

Ze względu na czytelność kodu oraz sposób działania modułu zajmującego się mapowaniem 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.

## 4.2 Użyte technologie

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

Rysunek 4.1: Porównanie wydajności popularnych języków względem C++. [17]

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

### 4.2.2 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++.

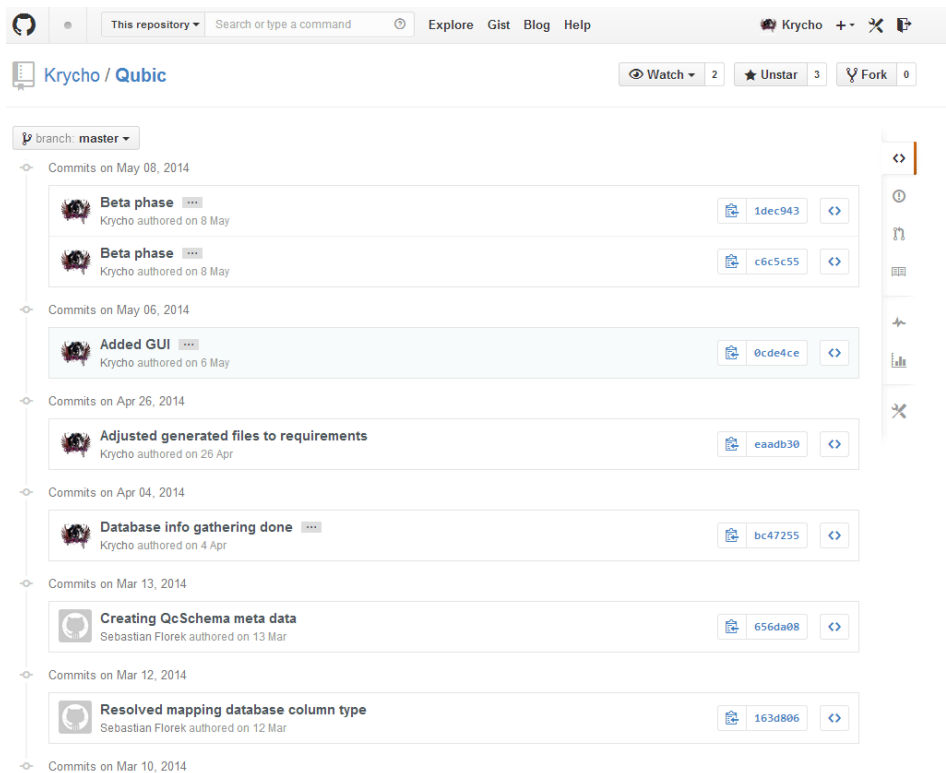
### 4.2.3 Środowisko deweloperskie

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.

### 4.2.4 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.

### 4.2.5 Inne narzędzia



Rysunek 4.2: Log kontrolny zmian projektowych z portalu Github.

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.

## 4.3 Projekt

### 4.3.1 Projekt warstwy danych

Poniżej przedstawiony kod został użyty do budowy testowej 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)
);

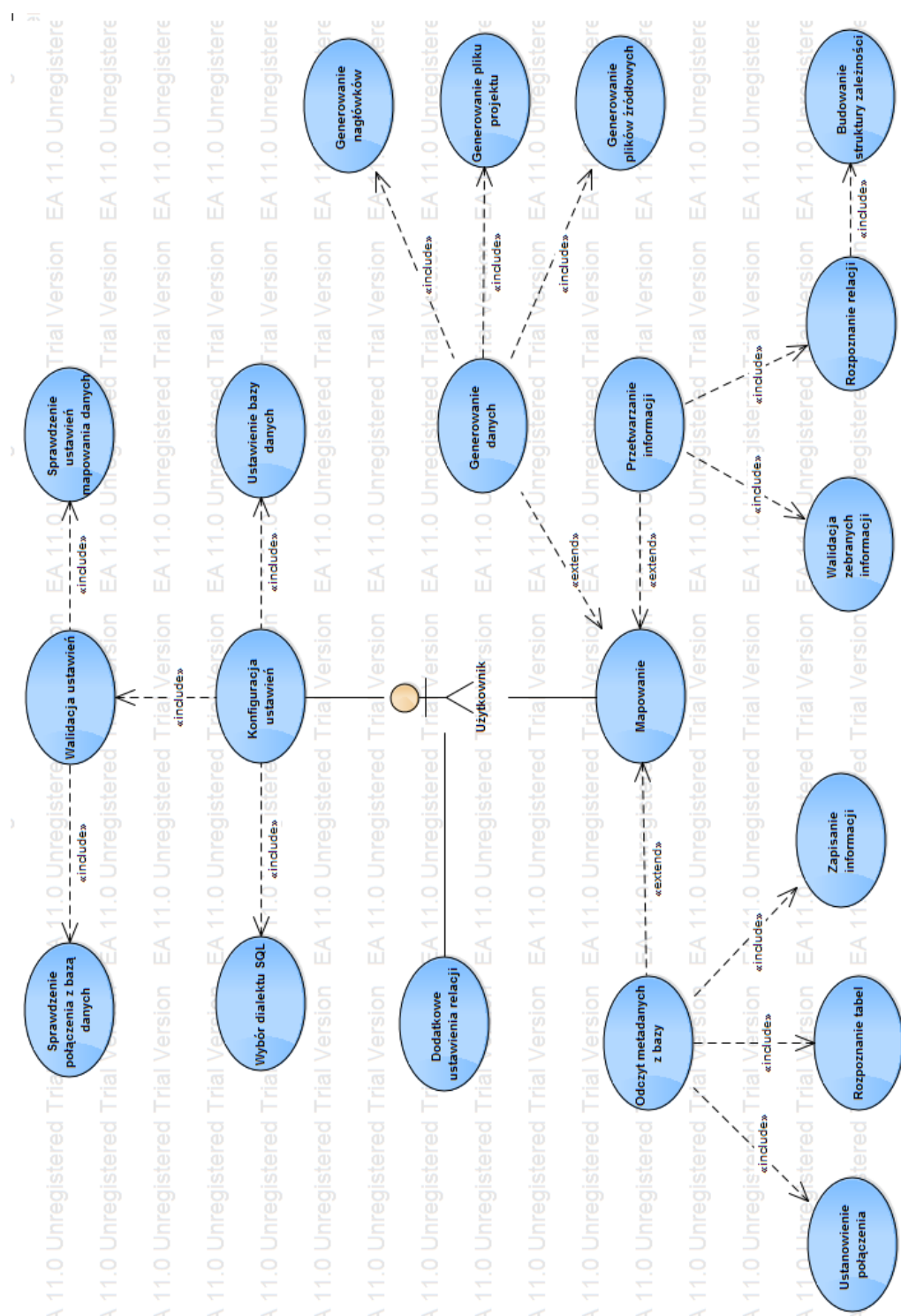
COMMIT;
```

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 przechowującą informacje o strukturze danych w systemie. Zaleca się więc używanie konta administratora bazy w celu generowania plików warstwy danych.

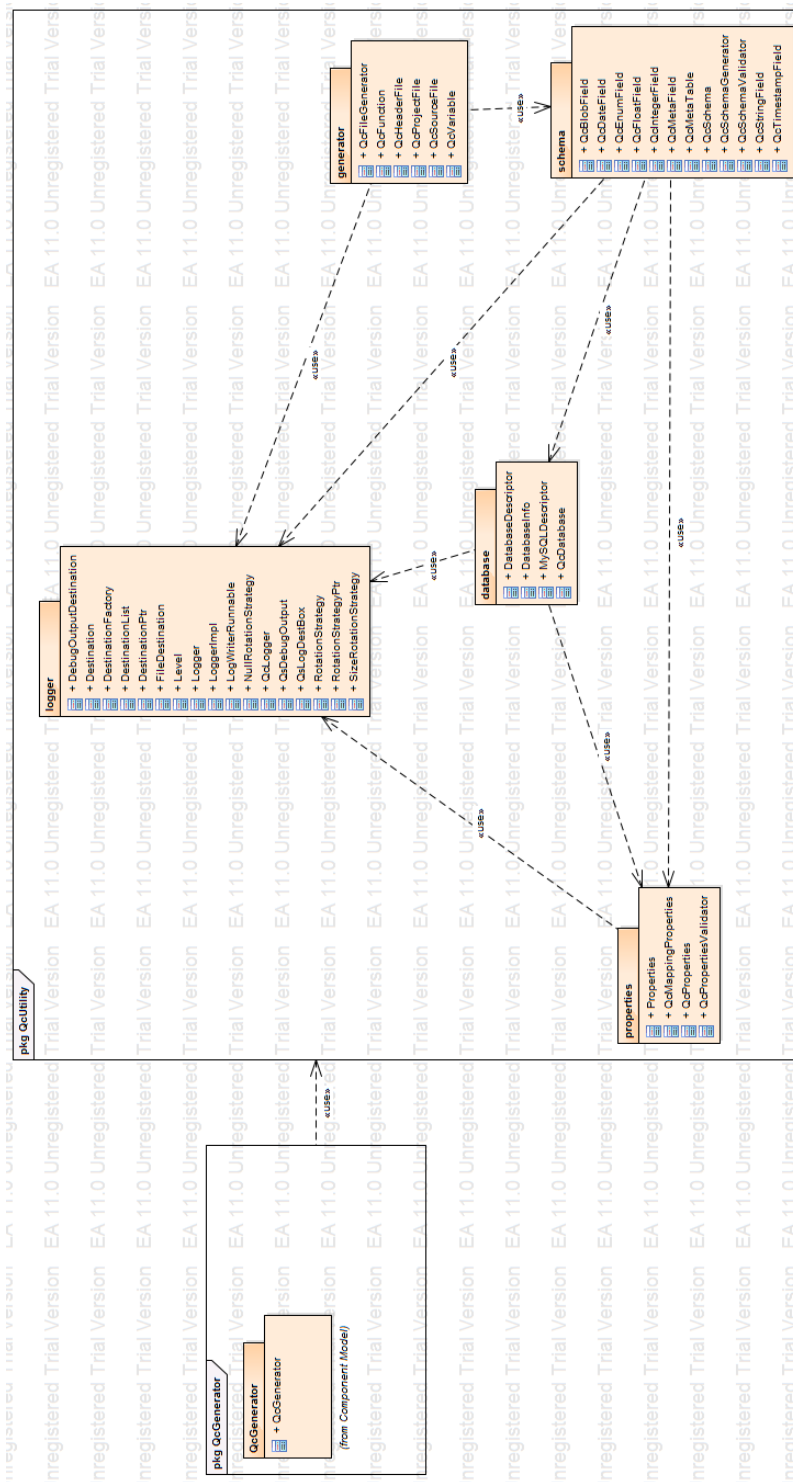
### 4.3.2 Projekt aplikacji

Projektowana aplikacja została podzielona na dwie części. Główny moduł generatora stworzono jako bibliotekę współdzieloną. Interfejs użytkownika jest osobną aplikacją, która została stworzona w oparciu o funkcjonalność udostępnioną przez bibliotekę. Relacje pokazane na diagramach UML, które znajdują się w dalszej części tego rozdziału przedstawiają jedynie główne związki między klasami. Ma to na celu pokazanie ogólnych zależności i przepływu informacji. Poniższy diagram przypadków użycia przedstawia możliwości generatora:





Rysunek 4.3: Diagram przypadków użycia modułu generatora.



Rysunek 4.4: Diagram koncepcyjny generatora.

Pakiet QcUtility jest w istocie biblioteką współdzieloną realizującą generowanie opisu mapowania obiektowo-relacyjnego. Moduł QcGenerator natomiast jest aplikacją opartą o funkcjonalność udostępnioną przez bibliotekę. Moduł główny zajmujący się generowaniem został podzielony na logiczne części w celu wyraźnej separacji odpowiedzialności każdego modułu<sup>1</sup>.

Pakiet „properties” odpowiada za odczyt i sprawdzanie ustawień aplikacji. Wszystkie ustawienia są sprawdzane przed połączeniem z bazą danych. Kolejny pakiet „database” jest odpowiedzialny za połączenie z bazą danych oraz udostępnienie informacji o strukturze bazy. Informacje te są następnie przekazywane do klasy QcSchemaGenerator pochodzącej z pakietu „schema”. Kolejnym 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 do użycia w środowisku programistycznym, za który odpowiedzialny jest pakiet „generator”.

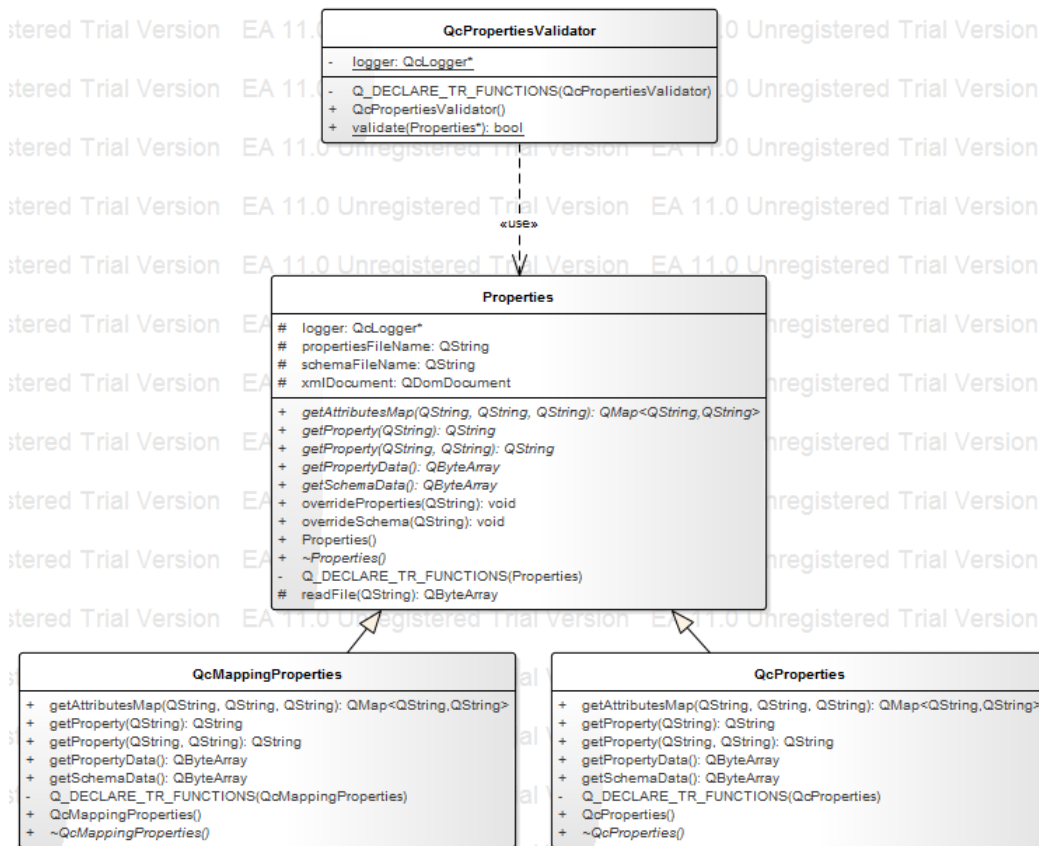
Warto również wspomnieć o pakiecie „logger”. Jest on używany przez wszystkie inne części aplikacji. Pozwala w prosty i przejrzysty sposób śledzić przebieg procesu mapowania oraz wychwycić ewentualne błędy. Został tu użyty projekt o otwartych źródłach o nazwie QsLog [12]. Kryteriami wyboru była łatwa rozszerzalność i prostota użycia. Dzięki temu w prosty sposób można było przekierować logi programu do interfejsu użytkownika zbudowanego w oparciu o aplikację szkieletową Qt. W tym celu dodana została klasa QsLogDestBox, która pozwala zapisywać dane do odpowiedniego kontenera Qt. Dodana została również klasa QcLogger, która jest nakładką na funkcje udostępniane przez projekt QsLog i została stworzona w oparciu o wzorzec singleton<sup>2</sup>. Formatuje ona w odpowiedni sposób komunikaty aplikacji, zapisuje dane do pliku logowania, który tworzony jest w folderze aplikacji oraz odpowiedzialna jest za ustawienie poziomu logowania komunikatów aplikacji.

---

<sup>1</sup>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.

<sup>2</sup>Singleton - wzorzec projektowy ograniczający możliwość tworzenia obiektów danej klasy do jednej instancji oraz zapewniający globalny dostęp do stworzonego obiektu.





Rysunek 4.6: 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. Klasa `QcMappingProperties` odczytuje informacje z pliku, który przechowuje informacje na jaki typ danych powinna zostać zmapowana odpowiednia kolumna w tabeli. Druga klasa `QcProperties` jest odpowiedzialna za odczytanie informacji do ustanowienia połączenia z bazą danych. Dodatkowo została udostępniona klasa `QcPropertiesValidator`, której zadaniem jest sprawdzenie istnienia i poprawności plików ustawień.

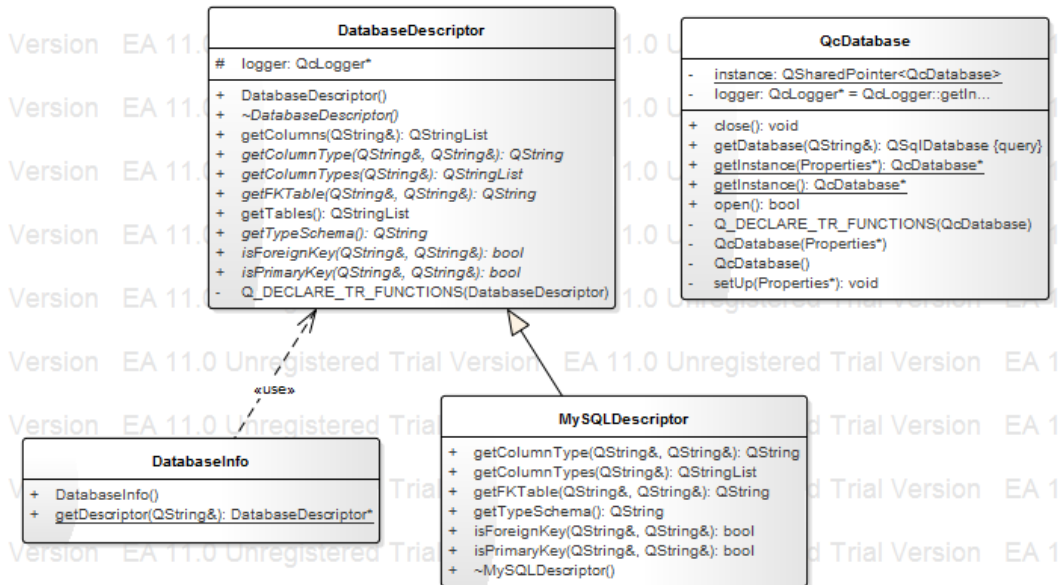
Moduł bazy danych pełni kilka funkcji:

- 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.
- Dzięki odpowiedniej hierarchii klas zapewnia wsparcie dla różnych systemów baz danych.

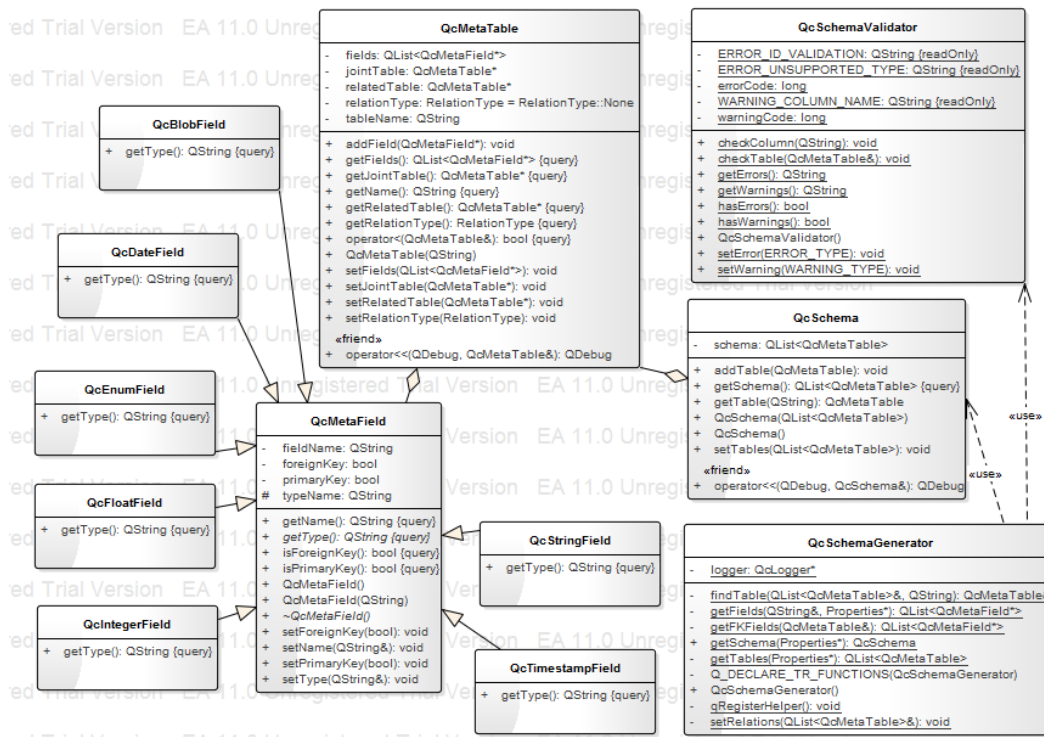
Wspierane przez Qubica typy baz danych są ograniczone przez framework Qt. Udostępnia on sterowniki i wsparcie dla następujących baz danych:

- IBM DB2
- Borland InterBase
- MySQL
- Oracle Call Interface Driver
- Open Database Connectivity (ODBC) - Microsoft SQL Server and other ODBC-compliant databases
- PostgreSQL
- SQLite version 2
- SQLite version 3
- SQLite version 3 for Symbian SQL Database
- Sybase Adaptive Server Note: obsolete from Qt 4.7

Z wyżej wymienionych języków, aplikacja przewiduje jedynie implementację pozwalającą generować projekt z baz danych typu MySQL. Możliwe będzie natomiast proste rozszerzenie aplikacji o obsługę dodatkowych baz danych. Dokładny opis wraz z przykładem zostanie przedstawiony w dalszej części pracy, w osobnym rozdziale.



Rysunek 4.7: Diagram klas modułu bazy danych.

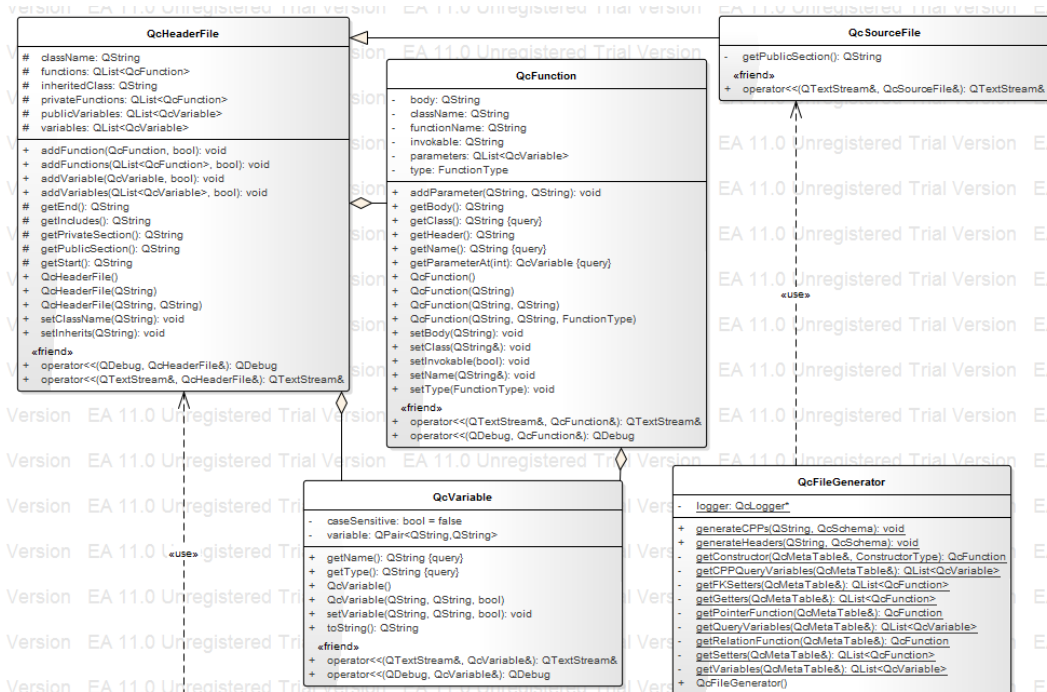


Rysunek 4.8: Diagram klas modułu budowania struktury danych.



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), poprzez analizę powiązań między kluczami głównymi i obcymi w tabelach.
- mapuje typy kolumn bazy danych na odpowiednie typy języka
- sprawdza czy zebrane dane spełniają postawione założenia
- udostępnia interfejs generujący



Rysunek 4.9: 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++.



Z uwagi na budowę plików źródłowych i nagłówkowych języka C++ zostały stworzone klasy reprezentujące zmienne, funkcje oraz same pliki zawierające kod. Wszystkie dane są generowane na podstawie wcześniej zbudowanego schematu bazy i przetwarzane do postaci kodu przez klasę `QcFileGenerator`.

## 4.4 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 singleton.

C++ Code 4.1: `QcDatabase`

```
class QcDatabase
{
    Q_DECLARE_TR_FUNCTIONS( QcDatabase )

public :

    static QcDatabase* getInstance( Properties *
        properties );
    static QcDatabase* getInstance();
    bool open();
    void close();
private :

    static QSharedPointer<QcDatabase> instance;
    QcDatabase( Properties *properties );
    QcDatabase();
};
```

Ważnym krokiem przed przystąpieniem do procesu generowania jest uzyskanie informacji z bazy danych o tabelach, typach kolumn, kluczach głównych i ob-

nych. W celu zapewnienia wsparcia dla różnych systemów relacyjnych baz danych stworzona została klasa abstrakcyjna `DatabaseDescriptor`, którą należy rozszerzyć aby obsłużyć konkretną bazę danych, a następnie dodać wpis do klasy `DatabaseInfo` zajmującej się tworzeniem konkretnej klasy w zależności od używanej bazy danych. Wymusza ona na użytkowniku implementację metod, które udostępnią informacje potrzebne do prawidłowego utworzenia schematu bazy danych w pamięci. Przykład implementacji obsługi bazy danych zostanie opisany w rozdziale zajmującym się możliwościami rozszerzania aplikacji. Oto przykładowy wpis w klasie `DatabaseInfo` dodający obsługę baz typu MySQL oraz kod klasy `DatabaseDescriptor` pokazujący wymagane metody:

```
if (driverName.contains("MYSQL", Qt::CaseInsensitive))
{
    return new MySQLDescriptor();
}
```

C++ Code 4.2: DatabaseDescriptor

```
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;  
};
```

Kolejnym problemem jest odpowiednie mapowanie typów kolumn na typy języka C++. Rozwiązaniem tego problemu stanowi klasa `QcMetaField`, która jest odpowiednikiem kolumny w tabeli i przechowuje informacje o jej nazwie, 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 tę klasę, dodać wpis do funkcji w klasie `QcSchemaGenerator` w celu zarejestrowania klasy w systemie refleksji Qt oraz zarejestrować nowy typ w pliku `qcmapping.xml` zawierającym ustawienia mapowania danych. Przykład zostanie przedstawiony w osobnym rozdziale.

C++ Code 4.3: `QcMetaField`

```
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
```

Tak wygląda przykładowy wpis w pliku mapowań, który typ `char` bazy danych mapuje na obiekt `QcStringField`.

```
<qc:map fromType="char" toType="QcStringField"/>
```

Jedną z najważniejszych klas modułu generatora to klasa `QcSchemaGenerator`, która wykorzystuje wyżej opisane klasy w celu utworzenia struktury bazy w pamięci programu. Klasa `QcMetaTable` reprezentuje pojedynczą tabelę bazy danych. Przechowuje ona nazwę tabeli, informacje o nazwach kolumn w tabeli, ich typach, kluczach obcych oraz typie relacji między daną tabelą, a tabelą powiązaną kluczem obcym. Główną funkcją `getTables` jest odpowiedzialna za odczytanie tabel, nazw i typów kolumn oraz rozpoznanie relacji między tabelami. W trakcie przetwarzania wykonywana jest również walidacja danych przez klasę `QcSchemaValidator`. Sprawdza ona czy każda tabela posiada własny klucz główny. Najkosztowniejszą operacją jest automatyczne wykrycie relacji między tabelami. W tym celu sprawdzana jest najpierw ilość kluczy obcych w danej tabeli. W przypadku znalezienia jednego klucza obcego, szukana jest tabela powiązana i w obu zapisywane są wskaźniki do drugiej tabeli wraz z informacją o rozpoznaniu relacji typu jeden do wielu.

W przypadku znalezienia dwóch kluczy obcych w tabeli, program zakłada, że tabela ta jest tabelą łącznikową. Wyszukiwane są obie powiązane tabele i w każdej z nich zapisywane są wskaźniki do dwóch pozostałych tabel, wraz z informacją o rozpoznaniu relacji typu wiele do wielu. W przypadku tej relacji została dodana również możliwość wskazania, które relacje typu wiele do wielu mają zostać wzięte pod uwagę przy generowaniu projektu.

C++ Code 4.4: QcSchemaGenerator

```

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);
    return result;
}

```

Na podstawie utworzonej struktury danych generowane są pliki projektu. Ze względu na wymagany podział projektu na pliki źródłowe i nagłówkowe została utworzona odpowiednia struktura klas. Najbardziej szczegółowa *QcVariable* reprezentuje zmienną i przechowuje informacje o jej nazwie i typie. Kolejne klasy reprezentują konkretne metody w klasie (*QcFunction*), pliki źródłowe (*QcHeaderFile*) i nagłówkowe (*QcSourceFile*). Ciała funkcji i nazwy metod zostały narzucone przez drugą część pracy zajmującą się mapowaniem obiektowo-relacyjnym. Generowane są m.in. konstruktory, nagłówki, klasy, funkcje odczytujące i ustawiające zmienne reprezentujące kolumny w bazie danych czy funkcje pozwalające uzyskać obiekt powiązanej klasy. Zbudowany w pamięci model plików źródłowych i nagłówkowych zostaje następnie zapisany do fizycznych plików na dysku użytkownika.

## C++ Code 4.5: QcFileGenerator

```
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<QcFunction> getFKSetters(
        QcMetaTable &table);
    static QcFunction getPointerFunction(QcMetaTable
        &table);
    static QcFunction getRelationFunction(
        QcMetaTable &table);
    static void copyResources(QString srcPath ,
        QString destPath);

    static void generateHeaders(QString dirPath ,
        QcSchema schema);
    static void generateCPPs(QString dirPath ,
        QcSchema schema);
    static void generatePRI(QString dirPath ,
        QcSchema schema);
public:
    QcFileGenerator() = delete;
    static void generateProject(QString dirPath ,
        QcSchema schema);
};
```

## 4.5 Opis użytkowania aplikacji

Proces generowania i korzystania z gotowego projektu jest bardzo prosty i wiąże się z wykonaniem kilku kroków.

- W podfolderze `resources` aplikacji znajdują się dwa pliki ustawień `qcmapping.xml` oraz `qcpproperties.xml`. Przed rozpoczęciem generowania należy uzupełnić informacje o bazie w drugim pliku.
- Po uruchomieniu aplikacji należy w Menu -> Load załadować pliki ustawień. Są one domyślnie zaczytywane z folderu `resources` oraz sprawdzane pod kątem błędów.
- Następnym krokiem jest ustanowienie połączenia z bazą, Menu -> Connect.
- Jeśli poprzedni krok został wykonany, dostępne będą opcje generowania oraz ustawienia dodatkowych informacji o relacjach.

Menu -> Set Relations pozwala na zdefiniowanie, które relacje wiele do wielu zostaną uwzględnione przy generowaniu projektu.

Menu -> Generate - po wybraniu tej opcji, program spyta nas o podanie folderu, w którym zostanie wygenerowany projekt.

Po wygenerowaniu projektu można zacząć tworzenie aplikacji opartej o Qubica. W celu późniejszego uruchomienia stworzonej aplikacji wymagane są następujące kroki:

- Z folderu z wygenerowanym projektem należy skopiować plik `qb.properties` do folderu z plikiem wykonawczym tworzonej aplikacji. Jest on wymagany przez drugą część aplikacji do działania.
- Do folderu ze stworzoną aplikacją należy również skopiować bibliotekę z odpowiednim sterownikiem bazy danych dla języka C++. Aplikacja szkieletowa Qt wymaga jej w celu nawiązania połączenia z bazą danych.

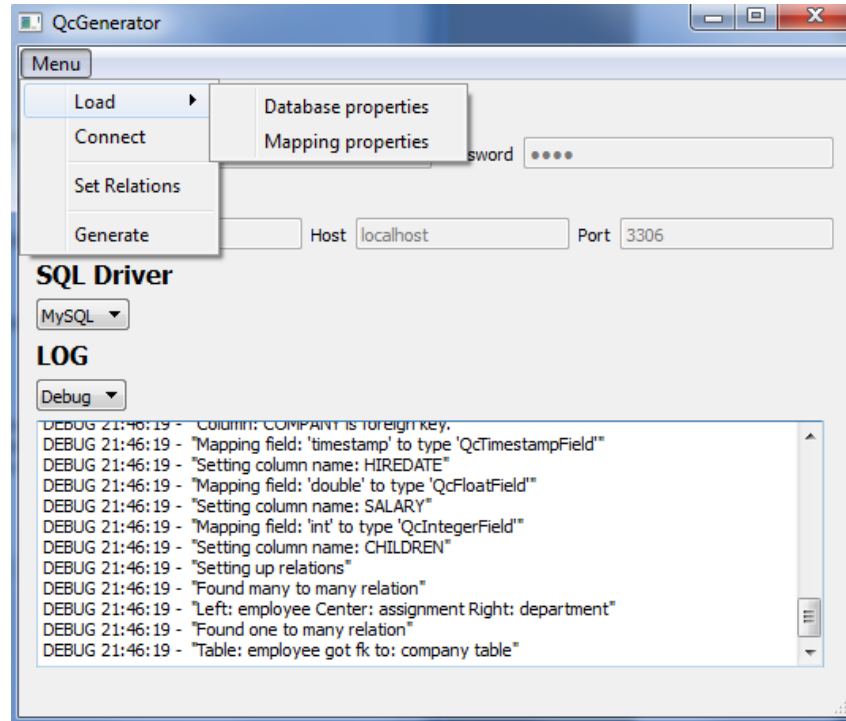
Cała część zajmująca się mapowaniem obiektowo-relacyjnym została wbudowana w stworzoną bibliotekę, dzięki czemu nie wymaga to dodatkowego nakładu w postaci ściągnięcia i ustawiania czegokolwiek.

## 4.6 Przykład użycia generatora opisu

Przykład zostanie oparty o testową bazę danych, która została przedstawiona w rozdziale 4.3.1. Zawiera ona podstawowe relacje typu jeden do wielu oraz wiele do wielu, co pozwala na testowanie wszystkich funkcjonalności generatora.

Poniżej przedstawiony jest interfejs prostej aplikacji graficznej opartej o funkcje udostępniane przez stworzoną bibliotekę. Została ona stworzona jedynie jako reprezentacja możliwości stworzonej biblioteki. Odczytuje ona i sprawdza poprawność odpowiednich plików ustawień, tj. pliku mapowań oraz pliku z informacjami o połączeniu. Następnie po poprawnej walidacji ustawień pozwala na ustanowienie połączenia z bazą.

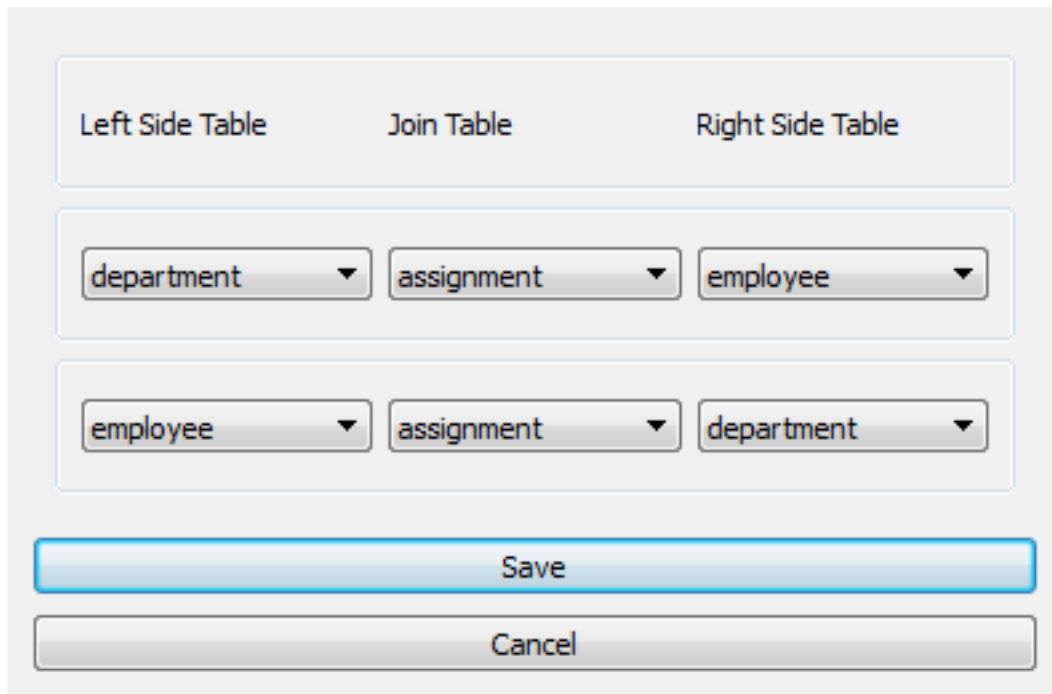
Pierwszym krokiem jest dostosowanie plików ustawień znajdujących się w folderze resources. Plik `qcpproperties.xml` przechowuje informacje potrzebne do połączenia z bazą danych. Drugi plik `qcmapping.xml` służy do mapowania typów kolumn na typy języka C++, a jego zastosowanie zostanie przedstawione w następnym rozdziale. Po uruchomieniu aplikacji należy z menu wybrać opcje Load -> Database properties, a następnie Load -> Mapping properties. Pliki ustawień zostaną odczytane i sprawdzona zostanie ich poprawność.



Rysunek 4.10: Interfejs generatora.



Dodatkowo została dodana możliwość zdefiniowania przez użytkownika, które relacje wiele do wielu mają zostać uwzględnione przy generowaniu projektu. Służy do tego opcja **Set Relations** w menu aplikacji. Interfejs zostaje dostosowany w zależności od ilości wykrytych relacji tego typu. Zostawienie pustego pola jest równoznaczne z usunięciem generowania funkcji odpowiedzialnej za obsługę danej relacji wiele do wielu.



The screenshot displays a configuration window for the Qubic generator. It features three columns labeled 'Left Side Table', 'Join Table', and 'Right Side Table'. Below these labels, there are two rows of dropdown menus. The first row contains 'department', 'assignment', and 'employee'. The second row contains 'employee', 'assignment', and 'department'. At the bottom of the window, there are two buttons: 'Save' and 'Cancel'.

Rysunek 4.11: Interfejs generatora.

Najważniejszą funkcją jest oczywiście możliwość wygenerowania projektu jeśli wcześniejsze kroki zostaną wykonane bez błędów. Po wybraniu opcji **Generate** aplikacja zapyta nas w jakim folderze zapisać wygenerowany projekt. Wygenerowane pliki znajdziemy w folderze **Qubic-Generated**. Są one podzielone na dwie części. Moduł zajmujący się mapowaniem obiektowo-relacyjnym znajduje się w folderze **ORM**, natomiast wygenerowane klasy, w folderze **Generated**. Dodatkowo załączony został plik ustawień **qb.properties** wymagany później do działania przez aplikację stworzoną w oparciu o Qubica.

Nazwa	Data modyfikacji	Typ	Rozmiar
Generated	2014-09-14 22:11	Folder plików	
ORM	2014-09-14 22:11	Folder plików	
qb.properties	2014-09-14 22:11	Plik PROPERTIES	1 KB
QubicProject.pro	2014-09-14 22:11	Qt Project file	1 KB

Rysunek 4.12: Struktura stworzonego projektu.

Poniżej przedstawione pliki, nagłówkowy i źródłowy zostały stworzone w oparciu o tabelę `Department`, która reprezentuje dział firmy. Jest ona powiązana relacją wiele do wielu z tabelą `Employee` reprezentującą pracownika. Jak widać wygenerowana została odpowiednia funkcja `getEmployees`, która pozwoli uzyskać odpowiednich pracowników z danego działu.

C++ Code 4.6: QcSchemaGenerator

```

#ifndef DEPARTMENT_H
#define DEPARTMENT_H

#include "qstring.h"
#include <ORM/ Qubic.h>

class Department : public QbPersistable
{
    Q_OBJECT

private :
    QString departmentname;
    static bool isRegistered;

public :
    Q_INVOKABLE Department();
    Q_INVOKABLE Department(qint32 id, QString
        departmentname);
    Q_INVOKABLE Department(QString
        departmentname);

```

```

        Q_INVOKABLE Department(const Department&
                                other);
        Q_INVOKABLE QString getDepartmentname() const
        ;
        Q_INVOKABLE void setDepartmentname( QString
                                departmentname);
        QList<QbPersistable*> getPointers();
        QList<QbPersistable*> getEmployees();
        static QString CLASSNAME;
        static QString ID;
        static QString DEPARTMENTNAME;
};

Q_DECLARE_METATYPE( Department )

#endif

```

## C++ Code 4.7: QcSchemaGenerator

```

#include "department.h"

QString Department:: ID = "ID";
QString Department:: DEPARTMENTNAME = "
    DEPARTMENTNAME";
QString Department:: CLASSNAME = "Department";
bool Department:: isRegistered = false;

Department:: Department() {
    if (!isRegistered) {
        qRegisterMetaType<Department>("
            Department");
        isRegistered = true;
    }
    this->id = -1;
}

```

```
Department::Department(qint32 id, QString
    departmentname) {
    if(!isRegistered) {
        qRegisterMetaType<Department>("
            Department");
        isRegistered = true;
    }
    this->id=id;
    this->departmentname=departmentname;
}

Department::Department(QString departmentname) {
    if(!isRegistered) {
        qRegisterMetaType<Department>("
            Department");
        isRegistered = true;
    }
    this->departmentname=departmentname;
    this->id = -1;
}

Department::Department(const Department& other) {
    id=other.id;
    departmentname=other.departmentname;
}

QString Department::getDepartmentname() const {
    return departmentname;
}

void Department::setDepartmentname(QString
    departmentname) {
    this->departmentname=departmentname;
}

QList<QbPersistable*> Department::getPointers() {
    return QList<QbPersistable*>();
}
```

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

## 4.7 Możliwości rozszerzania aplikacji

Dzięki stworzeniu odpowiedniej hierarchii klas i użyciu możliwości programowania obiektowego, można w łatwy sposób dodać do aplikacji obsługę nowego typu bazy danych, czy typu kolumny w tabeli. W celu dodania obsługi nowego typu bazy danych należy:

1. Stworzyć klasę dziedziczącą z klasy `DatabaseDescriptor` i zaimplementować metody: `getColumnType`, `getColumnTypes`, `getFKTable`, `getTypeSchema`, `isForeignKey`, `isPrimaryKey`.
  - 1.1. `getColumnType` - na podstawie nazwy tabeli i nazwy kolumny zwraca typ kolumny.
  - 1.2. `getColumnTypes` - na podstawie nazwy tabeli zwraca listę typów kolumn w tej tabeli.
  - 1.3. `getFKTable` - na podstawie nazwy tabeli i nazwy kolumny klucza obcego zwraca nazwę powiązanej tabeli.
  - 1.4. `getTypeSchema` - zwraca nazwę bazy danych, w której znajdują się informacje o strukturze bazy, z której chcemy wygenerować projekt.
  - 1.5. `isForeignKey`, `isPrimaryKey` - na podstawie nazwy tabeli i nazwy kolumny zwraca informacje czy jest ona kluczem obcym lub kluczem głównym.
2. Dodać wpis do klasy `DatabaseInfo` rejestrujący stworzoną klasę dla odpowiedniego typu bazy danych. Przykładowa linijka kodu dla bazy MySQL znajduje się poniżej.

```

if (driverName.contains("MYSQL", Qt::CaseInsensitive))
{
    return new MySQLDescriptor();
}

```

Oto fragment kodu z klasy `MySQLDescriptor` pokazujący implementację funkcji `getColumnType`.

C++ Code 4.8: `QcSchemaGenerator`

```

QString MySQLDescriptor::getColumnType(const QString
&tableName, const QString &columnName) {
    QSqlDatabase db = QcDatabase::getInstance()->
        getDatabase("info");
    QSqlQuery query(QString("SELECT data_type FROM "
        + getTypeSchema() + ".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();
}

```

Oprócz dodawania obsługi innych typów baz danych, użytkownik może również dodać obsługę nowych typów danych istniejących w bazie. W tym celu należy:

1. Dodać wpis do pliku `qcmapping.xml` mapujący typ w bazie danych na odpowiednią klasę.
2. Jeśli klasa do obsługi danego typu nie istnieje należy stworzyć klasę dziedziczącą z klasy `QcMetaField` i nadpisać jedną metodę `getType`. Zwraca ona

nazwę typu języka C++, np. „int”, który ma zostać użyty jeśli zostanie znaleziony odpowiedni typ kolumny w czasie generowania.

3. Ostatnim krokiem jest dodanie wpisu do nagłówka klasy `QcSchemaGenerator`, który pozwoli rozpoznać nowy typ w systemie refleksji Qt. Dzięki temu jeśli dodany zostanie nowy typ, obsługiwany przez już istniejące klasy, nie będzie wymagana ponowna kompilacja aplikacji.

Przykładowa implementacja funkcji `getType` dla typu `blob`.

```
QString QcBlobField::getType() const {  
    return "QByteArray";  
}
```

Wpis w pliku `qcmapping.xml` rejestrujący powyższy typ.

```
<qc:map fromType="blob" toType="QcBlobField"/>
```

Wpis w nagłówku klasy `QcSchemaGenerator` dodajemy do funkcji `qRegisterHelper`.

```
template<>  
void QcSchemaGenerator::qRegisterHelper<QcMetaField>()  
{  
    qRegisterMetaType<QcStringField>();  
    qRegisterMetaType<QcIntegerField>();  
    qRegisterMetaType<QcEnumField>();  
    qRegisterHelper<QcFloatField>();  
    qRegisterHelper<QcTimestampField>();  
    qRegisterHelper<QcDateField>();  
    qRegisterHelper<QcBlobField>();  
}
```

# Rozdział 5

## Podsumowanie

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

### 5.1 Dyskusja wyników

Głównym atutem stworzonej aplikacji jest wsparcie większej ilości dialektów SQL od konkurencyjnych rozwiązań oraz bardzo mała ilość podobnych aplikacji w języku C++, dzięki czemu Qubic staje się bezkonkurencyjnym rozwiązaniem w swojej dziedzinie. Dodatkowym czynnikiem, dzięki któremu stworzona aplikacja staje się lepszym wyborem jest brak opłat za jej użytkowanie. Prostota użycia, brak zależności od zewnętrznych aplikacji czy dodatkowego ręcznego konfigurowania to tylko kilka z wielu atutów stworzonego generatora opisu mapowania obiektowo-relacyjnego.

### 5.2 Ocena możliwości wdrożenia Qubica

Qubic pozwala zaoszczędzić programistom dużą ilość czasu w fazie tworzenia aplikacji, dzięki czemu mogą się 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.



### 5.3 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. Pozwoliłoby 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 języka SQL.

# Bibliografia

- [1] M. Keith, M. Schincariol, Pro EJB 3 Java persistence API. APress, 2006. ISBN-13 978-1-59059-645-6
- [2] C. Bauer, G. King, Hibernate w akcji, Helion, 2007, ISBN: 978-83-246-0527-9
- [3] A. Ezust, P. Ezust, Introduction to Design Patterns in C++ with Qt, Pearson Education, Inc, 2012 , ISBN 978-0-13-282645-7
- [4] D. Gennaro, Advanced C++ Metaprogramming, CreateSpace Independent Publishing Platform , 2011, ISBN-13 978-1460966167
- [5] P. Wilton, J. Colby, Beginning SQL, Wrox, 2005, 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](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] <https://bitbucket.org/razvanpetru/qt-components/src/a476515a5b40479da3e90e18557bbc0257dbda16/QsLog/?at=default> - [dostęp 12.09.2014]
- [13] <http://msdn.microsoft.com/en-us/library/ff649643.aspx> - [dostęp 28.07.2014]

- [14] <http://www.oracle.com/technetwork/java/dataaccessobject-138824.html> - [dostęp 28.07.2014]
- [15] <http://msdn.microsoft.com/en-us/library/ee658127.aspx> - [dostęp 27.07.2014]
- [16] <http://www.visual-paradigm.com/VPGallery/img/orm/Overview/ORM-Overview.png> – [dostęp 02.11.2013]
- [17] <http://blog.famzah.net/2010/07/01/cpp-vs-python-vs-perl-vs-php-performance-benchmark/> - [dostęp 02.07.2014]
- [18] <http://i.msdn.microsoft.com/dynimg/IC34006.gif> - [dostęp 29.07.2014]
- [19] [http://www.qxorm.com/qxentityeditor/resource/qxee\\_sample.png](http://www.qxorm.com/qxentityeditor/resource/qxee_sample.png) - [dostęp 04.09.2014]

# Spis rysunków

2.1	Uproszczony przebieg generowania opisu mapowania obiektowo-relacyjnego [16]. . . . .	11
2.2	Wzorzec Model-Widok-Kontroler. . . . .	12
2.3	Schemat działania biblioteki współdzielonej [18]. . . . .	13
3.1	Interfejs generowania kodu frameworka Hibernate. . . . .	15
3.2	Interfejs aplikacji QxEntityEditor. . . . .	17
4.1	Porównanie wydajności popularnych języków względem C++. [17]	21
4.2	Log kontrolny zmian projektowych z portalu Github. . . . .	22
4.3	Diagram przypadków użycia modułu generatora. . . . .	25
4.4	Diagram koncepcyjny generatora. . . . .	26
4.5	Diagram klas modułu logera. . . . .	28
4.6	Diagram klas modułu ustawień. . . . .	29
4.7	Diagram klas modułu bazy danych. . . . .	31
4.8	Diagram klas modułu budowania struktury danych. . . . .	31
4.9	Diagram klas modułu generatora projektu. . . . .	32
4.10	Interfejs generatora. . . . .	40
4.11	Interfejs generatora. . . . .	41
4.12	Struktura stworzonego projektu. . . . .	42

# Spis tablic

3.1	Porównanie wybranych aplikacji zajmujących się generowaniem opisu mapowania obiektowo-relacyjnego. . . . .	17
-----	--	----