Politechnika Warszawska
Wydział Elektroniki i Technik Informacyjnych
Instytut Informatyki

Rok akademicki 2012/2013

Praca dyplomowa inżynierska

Łukasz Rafał Szewczyk

Projekt i implementacja biblioteki dla Qt i Qt Quick do operowania na wykresach typu biurowego

Opiekun pracy:
mgr inż. Witold Wysota

Ocena .....

.....
Podpis Przewodniczącego
Komisji Egzaminu Dyplomowego

Specjalność: Informatyka –
Inżynieria systemów informatycznych

Data urodzenia: 02 grudnia 1990 r.

Data rozpoczęcia studiów: 1 października 2009 r.

Życiorys

Urodziłem się dnia 2. grudnia 1990 roku w Warszawie. Całe dotychczasowe życie spędziłem mieszkając w podwarszawskiej Kobyłce, w której to uczęszczałem do szkoły podstawowej i gimnazjum. Z powodu zainteresowania matematyką zdecydowałem się w roku 2006 rozpocząć naukę w XVIII Liceum Ogólnokształcącym im. Jana Zamoyskiego w Warszawie, w klasie o profilu matematyczno-fizyczno-informatycznym. W pierwszej klasie liceum rozpocząłem równiez swoją przygodę z siatkówką w klubie Junior Stolarka Wołomin.

Dobre wyniki osiągnięte na egzaminach maturalnych w roku 2009 pozwoliły mi dostać się na dzienne studia inżynierskie na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. W trakcie studiów poza nauką kontynuowałem swój siatkarski rozwój, dzięki czemu trafiłem do zespołu AZS Politechnika Warszawska. W 2012 roku wróciłem do swojego macierzystego klubu, który w międzyczasie wrócił do swojej historycznej nazwy – Huragan Wołomin. W sezonie 2012/13 udało mi się wywalczyć z tym klubem awans do II ligi siatkówki.

.....
podpis studenta

Egzamin dyplomowy

Złożył egzamin dyplomowy w dn. .....

Z wynikiem .....

Ogólny wynik studiów .....

Dodatkowe wnioski i uwagi Komisji .....

.....

Streszczenie

Praca prezentuje projekt biblioteki zawierającej uniwersalny silnik służący do tworzenia biurowych wykresów w Qt oraz Qt Quick. Wykresy są w szczególności nastawione na interakcję z użytkownikiem oraz współpracę z już istniejącymi mechanizmami Qt. W ramach pracy wykonano opis i analizę wymagań oraz projekt architektury biblioteki. Zaplanowano również testy tworzonej biblioteki. Projekt zakłada implementację z wykorzystaniem języka C++ oraz bibliotek Qt w wersji 5.

Słowa kluczowe: *uniwersalny silnik wykresów, obiektowa architektura.*

Abstract

Title: *Project and implementation of office type charts library for Qt and Qt Quick*

This paper presents a project of library containing universal engine for creating office charts in Qt and Qt Quick. Charts are particularly focused on the user interaction and collaboration with existing mechanisms in Qt. The paper contains description and analysis of requirements and project of architecture of library. Also planned tests of designed library. The project involves the implementation using C++ language and Qt libraries in version 5.

Key words: *universal charting engine, object oriented architecture.*

Spis treści

Wstęp

Częstym zadaniem tworzonego oprogramowania jest prezentacja pewnych danych. Nawet najlepszy program wykonujący skomplikowane obliczenia jest niewiele wart dla klienta, jeśli nie potrafi w przystępny sposób zaprezentować efektów swoich działań.

Jedną z podstawowych form prezentacji danych w informatyce (i nie tylko) jest tabela, która umożliwia wyświetlanie danych z dużą precyzją. Qt posiada już architekturę służącą tworzeniu takich systemów – Model-Widok [2]. Jest to bardzo popularna platforma umożliwiająca tworzenie skomplikowanych interaktywnych tabelek prezentujących dane z różnych źródeł, m.in. z bazy danych.

Inną formą prezentacji danych są wykresy, których popularność jest porównywalna do tabel. Jest to forma dużo bardziej przystępna dla ludzkiej percepcji. Ułatwiają szybkie porównywanie wielu rekordów oraz wyznaczanie tendencji. Ponadto ich obrazkowa natura oraz możliwe animacje sprawiają, że jest to forma bardzo atrakcyjna dla końcowego użytkownika oprogramowania.

W ostatnich latach da się zauważyć wyraźną tendencję odchodzenia od tabel na rzecz graficznych form prezentacji danych. Tendencja ta jest szczególnie nasilona w sektorze urządzeń mobilnych, takich jak telefony klasy smartphone. Dodatkowo rozpowszechnienie się technologii ekranów dotykowych umożliwia realizację interakcji na niedostępnym dotychczas poziomie. Qt od wersji 5.2 ma być dostępne na najpopularniejszych tego typu urządzaniach pracujących pod kontrolą systemów Android oraz iOS.

Co prawda Qt posiada już kilka bibliotek umożliwiających tworzenie wykresów, jednak brakuje darmowych, gotowych do użycia rozwiązań umożliwiających interaktywną prezentację danych. Dodatkowo nowa technologia tworzenie interfejsów użytkownika – Qt Quick cierpi na niedobór bibliotek dostarczających komponentów wyższego poziomu. Jest to odpowiedni moment na wprowadzenie na rynek produktu, który ma szansę zyskać dużą popularność, choćby z powodu braku konkurencji.

Moja praca inżynierska składa się z rozdziałów opisujących kolejne etapy tworzenia biblioteki. W rozdziale 2 omawiam dostępne biblioteki do tworzenia wykresów w Qt, następnie przedstawiam opis i analizę wymagań stawianych mojej bibliotece. Kolejny rozdział to inżynierski projekt architektury całej biblioteki. Następne dwa rozdziały są poświęcone implementacji oraz testom biblioteki. Ostatni zawiera wnioski dotyczące stworzonej biblioteki oraz możliwości jej dalszego rozwoju.

1. Wprowadzenie

W tym rozdziale przedstawiam podstawy Qt oraz Qt Quick, dzięki którym dowolny czytelnik powinien zrozumieć zawartość mojej pracy. Następnie opisuję podstawowe założenia, cele oraz cechy stworzonej biblioteki.

1.1. Qt

Qt jest zbiorem bibliotek języka C++, które są dostępne na licencjach LGPL, GPL oraz komercyjnej. Lista bibliotek jest dość okazała, a znaleźć na niej można narzędzia do tworzenia interfejsów użytkownika, parsowania plików XML czy dostępu do baz danych. Naczelną zasadą Qt jest: *pisz raz, kompiluj wielokrotnie* [1] – dzięki takiemu podejściu programy napisane w Qt są przenośne pomiędzy najpopularniejszymi platformami na poziomie kodu źródłowego.

Najnowsza dostępna wersja Qt to 5.1. Nowa odsłona dostarcza programistom szereg usprawnień oraz modułów, m.in. do obsługi formatu JSON. Jednak głównym punktem Qt 5 jest nowa implementacja Qt Quick.

Do implementacji Qt Quick 2 wykorzystano OpenGL i SceneGraph, co znacznie poprawiło wydajność tego systemu. Qt 5 rozpoczęło też nowy kierunek rozwoju aplikacji wykorzystujących Qt. Qt Quick jest promowany jako zalecany sposób tworzenia interfejsów użytkownika. Docelowo aplikacje Qt mają być podzielone na GUI napisane w QML oraz logikę zaprogramowaną w C++.

1.1.1. Narzędzia

Najważniejszymi narzędziami, które umożliwiają tudzież ułatwiają pracę z Qt są:

— moc (Meta Object Compiler) – specjalny program, który można porównać do preprocesora. Na podstawie naszego kodu generuje on dodatkowe pliki źródłowe potrzebne Qt, niewidoczne dla programisty.
— uic (User Interface Compiler) – kompilator plików *.ui, które zawierają informację o układzie interfejsu użytkownika.
— qmake – program ułatwiający zarządzanie procesem budowania projektu.
— Qt Creator – zintergrowane środowisko programistyczne, przeznaczone głównie dla języków C++, QML i JavaScript.
— Qt Designer – program umożliwiający łatwe tworzenie interfejsów użytkownika. Generuje on wspomniane juz pliki *.ui.

1.1.2. QObject

C++ nie wymusza dziedziczenia po określonej klasie, jak ma to miejsce chociażby w Javie, gdzie zawsze na szczycie drzewa dziedziczenia znajduje się klasa *Object*.

[1] http://qt-project.org/wiki/QtWhitepaper

Qt wprowadza swoją klasę – *QObject*. Dzięki wielodziedziczeniu, nie musimy rezygnować z dotychczasowej hierarchii dziedziczenia, aby otrzymać wiele ciekawych możliwości płynących z wykorzystania *QObject*. Niektóre z nich to:
— Relacja rodzic–dziecko, która jest nawiązywana w chwili tworzenia obiektów. Umożliwia ona wyszukiwanie dzieci danego obiektu po ich klasie, bądź nazwie. Ponadto ułatwia zarządzanie pamięcią, poprzez automatyczne usuwanie obiektów – dzieci w chwili usunięcia rodzica. Przykład: usunięcie okna zawierającego wiele elementów spowoduje posprzątanie ze sterty wszystkich przycisków, etykiet czy obrazków.
— *qobject_cast* – dynamiczne rzutowanie, stosowane do rzutowania w dół hierarchii dziedziczenia. Jest ono znacznie szybsze od *dynamic_cast*, gdyż nie korzysta z mechanizmu RTTI (Run Time Type Information). Jedynym oczywistym mankamentem jest fakt, że rzutowanie to działa jedynie dla klas dziedziczących po *QObject*.
— Zdarzenia – niskopoziomowy mechanizm komunikacji. Qt opakowuje standardowe zdarzenia w obiekty swoich klas i dostarcza je do odpowiednich obiektów. Przed dostarczeniem zdarzenia do adresata można je przefiltrować, podejrzeć lub wręcz zmienić.
— Sygnały i sloty – wysokopoziomowy mechanizm komunikacji będący implementacją wzorca *Obserwator* [5]. Jest to bardzo wygodny sposób na luźne wiązanie obiektów, które mogą ze sobą współpracować, nie wiedząc nawzajem o swoim istnieniu.
— Właściwości – sposób na parametryzowanie obiektów. Istnieją zarówno właściwości statyczne, dodawane w czasie kompilacji, wspólne dla wszystkich obiektów danej klasy, np. wysokośc czy kolor, jak i dynamiczne, przypisywane pojedynczym obiektom już w czasie wykonania programu.

1.2. Qt Quick

Qt Quick jest nową technologią tworzenia GUI. Jej przeznaczeniem jest tworzenie lekkich, intuicyjnych oraz płynnie działających interfejsów, głównie na platformach mobilnych. W przeciwieństwie do tradycyjnego Qt, Qt Quick nie wymaga znajomości C++, co ma dopuścić do pracy nad GUI nie tylko programistów, ale również projektantów – grafików. Na Qt Quick składają się:
— QML – deklaratywny język służący do opisu wyglądu oraz zachowania GUI,
— JavaScript – imperatywny język służący do implementacji wewnętrznej logiki GUI,
— Środowisko uruchomieniowe, pozwalające na uruchamianie aplikacji Qt Quick bez każdorazowej kompilacji,
— Designer, program umożliwiający „wyklikanie" interfejsu użytkownika,
— API umożliwiające integrację z aplikacjami Qt napisanymi w C++.

W Qt4, Qt Quick był oparty na architekturze *Graphics View* [2], jednak problemy wydajnościowe zmusiły projektantów do sięgnięcia po bardziej zaawansowane narzędzia. W Qt5 wykorzystano bezpośrednio *OpenGL* oraz *SceneGraph* [6].

[2] Framework Graphics View `http://qt-project.org/doc/qt-5.0/qtwidgets/graphicsview.html`

1.2.1. QML

QML (Qt Modeling Language) jest deklaratywnym językiem służącym głównie do opisu wyglądu i zachowania GUI. QML może jednak służyć do zupełnie innych zastosowań. W nowym systemie zarządzania procesem budowania projektów napisanych w Qt – QBS [3] językiem opisu projektu jest właśnie QML.

Relacja rodzic-dziecko elementów QML została zorganizowana w drzewiastą strukturę, ułatwiającą zarządzanie elementami. Podobnie jak obiekty w klasycznym Qt, elementy są parametryzowane poprzez właściwości, np. id lub szerokość.

Ciekawą cechą QML jest system wiązania wartości z właściwością, który umożliwia uzależnienie właściwości A od właściwości B. Zmiana wartości B w czasie wykonywania programu spowoduje automatyczne przeliczenie wartości A.

Elementy QML mogą być rozszerzane przez kod napisany w JavaScript lub poprzez integrację z modułami napisanymi w C++.

1.2.2. Przykładowy kod QML

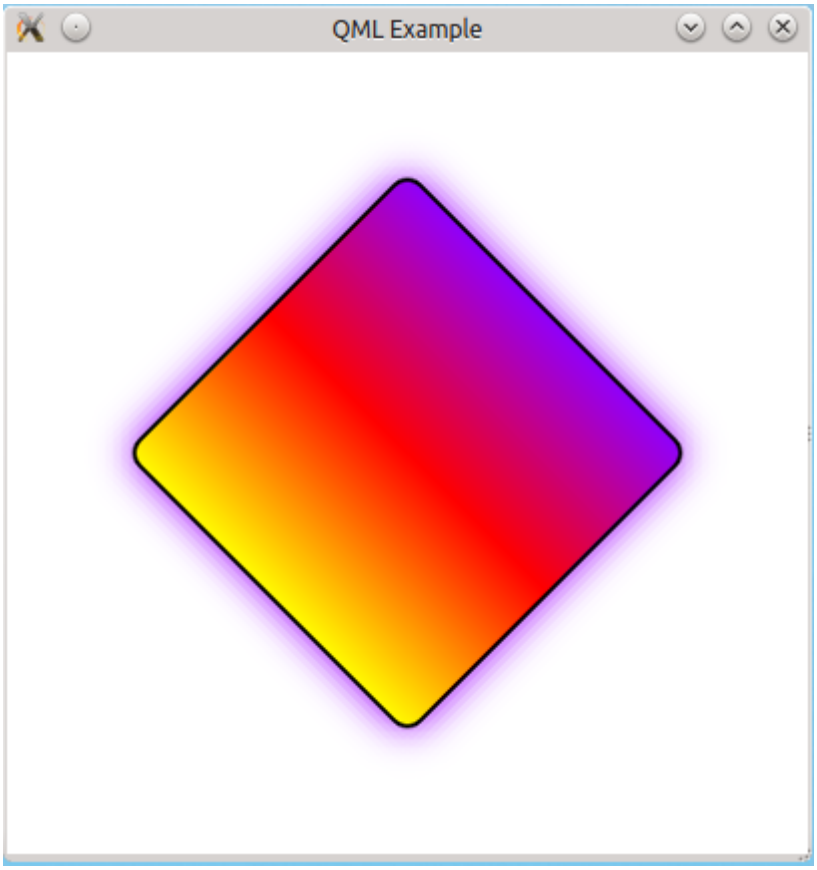
Wydruk 1.1. Przykład QML

```
import QtQuick 2.1
import QtGraphicalEffects 1.0

Item {
  width: 400; height: width

  Rectangle {
    id: rct
    width: parent.width/2; height: width
    anchors.centerIn: parent
    gradient: Gradient {
     GradientStop { position: 0; color: "#8F00FF" }
     GradientStop { position: 0.5; color: "red" }
     GradientStop { position: 1; color: "yellow" }
    }
    border { color: "black"; width: 2 }
    radius: 10
    rotation: 45
  }
  layer.enabled: true
  layer.effect: Glow { radius: 30; samples: 16; color: "#9F00FF" }
}
```

[3] Qt Build Suit `http://qt-project.org/wiki/qbs`



Rysunek 1.1. Przykład wykorzystania QML

1.2.3. Własne elementy

Istnieje możliwość tworzenia własnych elementów składających się z elementów już dostępnych w QML. Jednak bardzo złożone elementy, np. wykresy, łatwiej jest stworzyć w C++, a następnie wyeksponować w QML. Możliwe jest eksponowanie pojedynczych obiektów, jak i eksportowanie całych klas. Aby klasa była zdatna do wykorzystania w QML musi dziedziczyć po *QObject*. Wyeksportowanie klasy odbywa się poprzez wywołanie jednej globalnej funkcji dostarczanej przez Qt. W ten sposób zintegrowano z QML chociażby bibliotekę Box2D.

1.3. Założenia, cele i cechy biblioteki

Podstawowym problemem bibliotek już dostępnych na rynku jest wąski zakres zastosowań. Dostarczane przez nie wykresy są zazwyczaj widgetami gotowymi do osadzenia w GUI. Dla typowych aplikacji napisanych w C++ może to być rozwiązanie wygodne, jednak ogranicza ono możliwości wykorzystania w innym kontekście. Osadzenie widgetu w dokumencie tekstowym lub w *GraphicsView* jest co najmniej nieefektywne. Stąd pierwszy z celów stawianych mojej bibliotece – dostarczenie uniwersalnego silnika, umożliwiającego wyświetlanie wykresów w dowolnym miejscu aplikacji napisanych w Qt. Na etapie definiowania wymagań ograniczyłem zbiór

widoków, z którymi moja biblioteka ma być kompatybilna, jednak teoretycznie powinna współpracować z dowolnym widokiem.

Jak już wspomniałem we wstępie, Qt Quick jest stosunkowo nową technologią i cierpi na typowe schorzenie wieku dziecięcego – ogarniczony zbiór bibliotek. Aktualnie na rynku istnieje tylko jedna biblioteka wspierająca tworzenie wykresów w QML i jest to rozwiązanie komercyjne. Istnieje duże zapotrzebowanie na wartościowe, darmowe rozwiązanie.

Celem Qt Quick jest tworzenie intuicyjnych i płynnie działających interfejsów użytkownika. Wykresy tworzone za pomocą mojej biblioteki wpisują się w tę politykę. Istnieje możliwość animacji wielu właściwości elementów składowych wykresów. Same wykresy są również nastawione na interaktywne operacje, których zbiór jest co prawda ograniczony, jednak łatwy do rozszerzenia za pomocą systemu zdarzeń Qt.

Niektóre z wymagań, głównie dostępność elementów wykresu z poziomu QML, wymusiły na mnie specyficzną architekurę tej biblioteki. Intensywne wykorzystywanie mechanizmów dostarczanych przez *QObject* oraz oszczędne korzystanie z szablonów wpłynęły negatywnie na wydajność rozwiązania. Nie jest to jednak problem, gdyż podstawowym założeniem tego projektu było dostarczenie wykresów biurowych, które ze swej natury mają raczej statyczne źródła danych. Stąd też nazwa biblioteki – *Qt Office Charts*, która jednoznacznie sugeruje jej przeznaczenie.

2. Przegląd dziedziny

Rozdział ten zawiera opis kilku wybranych bibliotek służących do tworzenia wykresów. Opisałem tu biblioteki przeznaczone dla Qt i C++, podzielone na rozwiązania komercyjne oraz darmowe. Przy omawianiu każdej z bibliotek podałem zarówno jej zaletyn jak i wady. Następnie przedstawiam wykaz typów wykresów dostępnych we wszystkich opisanych produktach. Kolejne dwa paragrafy to zestawienie elementów wspólnych i unikalnych. Ostatni fragment to podsumowanie zawierające wnioski płynące z tego przeglądu.

2.1. Rozwiązania komercyjne

Istnieje kilka komercyjnych bibliotek służących do tworzenia wykresów w Qt. Zostały one stworzone przez jedne z najpoważniejszych studiów developerskich tworzących oprogramowanie w Qt. Bliższe przyjrzenie się tym bibliotekom pomogło mi stworzyć listę wymagań stawianych mojej bibliotece oraz częściowo zasugerowało pewne rozwiązania architektoniczne.

2.1.1. Qt Commercial Charts

Biblioteka ta została stworzona przez aktualnego właściciela Qt, czyli firmę Digia. Oferuje ona programistom szeroki wybór wykresów. Nie wymaga zagłębiania się w swoją wewnętrzną budowę, a tworzenie w niej prostych wykresów polega na połączeniu kilku wysokopoziomowych obiektów takich jak seria czy widget widoku.

Cała biblioteka opiera się na frameworku Graphics View, który wprowadza podział na scenę oraz widok, gdzie scena jest kontenerem na elementy prezentowane za pomocą widoku. Biblioteka udostępnia dwa uniwersalne widoki. Zależnie od zastosowania można użyć obiektu klasy dziedziczącej z QGraphicsView bądź z QGraphicsWidget. Implementacje konkretnych wykresów zarządzają elementami dodawanymi do sceny.

Biblioteka ta, jako jedyna z tutaj opisanych, udostępnia wszystkie swoje wykresy w języku QML, zarówno dla QtQuick 1 i 2. Jak zaznacza sam producent, silne uzależnienie biblioteki od GraphicsView sprawia, że osiąga ona lepsze wyniki wydajnościowe dla Qt Quick 1 niż Qt Quick 2 [1].

Zalety:
— szeroki wybór wykresów,
— operowanie komponentami wysokiego poziomu,
— interaktywność wszystkich elementów wykresu,

[1] http://blog.qt.digia.com/blog/2013/06/19/qt-charts-1-3-0-released-2/

— system motywów umożliwiający tworzenie wykresów o spójnej kolorystyce,
— obiekty mapujące dane ze standardowych modeli Qt do serii danych,
— wtyczka do Designera,
— silne wsparcie dla QML.

Wady:
— spadek uniwersalności poprzez uzależnienie prezentacji wykresów od konkretnych widoków,
— wykorzystanie GraphicsView zamiast SceneGraph, powodujące niższą wydajność w QtQuick 2.

2.1.2. KD Charts

Jest to biblioteka stworzona przez firmę KDAB. KD Charts w odróżnieniu od poprzedniej biblioteki nie korzysta z GraphicsView, a z mechanizmów niższego poziomu. Rysowanie odbywa się tu za pomocą systemu Arthur, a do pisania wykorzystano silnik Scribe. Dzięki wykorzystaniu technologii niższego poziomu, programistom KDAB udało się uzyskać lepszą wydajność, szczególnie ważną przy oferowanych przez nich wykresach czasu rzeczywistego.

KD Charts separuje dane od warstwy prezentacji wykorzystując modele znane z popularnego w Qt wzorca Model-Widok. Za prezentację danych odpowiada uniwersalna klasa dziedzicząca po QWidget.

Zalety:
— szeroki wybór wykresów,
— zbiór wbudowanych interakcji oraz możliwość tworzenia własnych,
— duże możliwości parametryzowania wyglądu wykresu,
— wykresy 2,5D,
— wysoka wydajność, umożliwiająca tworzenie wykresów czasu rzeczywistego.

Wady:
— wysoka cena,
— wymuszenie korzystania z modeli albo niskopoziomowych kontenerów do przechowywanie danych,
— biblioteka napisana w stylu Qt4, trudnym do wykorzystania w QtQuick 2,
— brak wsparcia dla QML.

2.1.3. QtitanChart

Jest to biblioteka stworzona przez firmę Developer Machines, udostępniająca dosyć duży zbiór wykresów. Twórcy biblioteki nie udostępniają pełnych źródeł swojej biblioteki, a jedynie jej pliki nagłówkowe i biblioteki dynamiczne, co znacznie utrudnia zrozumienie jej wewnętrznego działania. Ta wiedza nie jest jednak potrzebna, gdyż aby tworzyć za jej pomocą wykresy, można się posługiwać komponentami wysokiego poziomu.

Zalety:
— szeroki wybór wykresów,

— możliwość wyeksportowania wykresu do pliku graficznego,
— możliwość wyświetlania wykresów różnych typów w jednym układzie współrzędnych,
— system motywów, umożliwiający tworzenie wykresów o jednolitej kolorystyce.

Wady:
— wysoka cena,
— brak wsparcia dla QML.

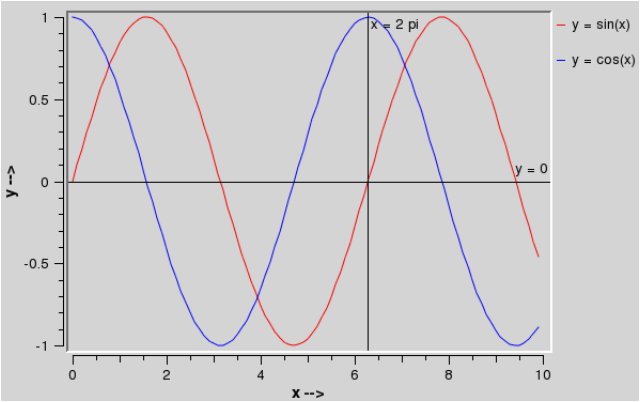
2.2. Rozwiązania darmowe

Warto również przyjrzeć się bibliotekom open–source, których analiza może posłużyć jako swego rodzaju przestroga przed pewnymi rozwiązaniami, których powinienem unikać w czasie projektowania mojej biblioteki.

2.2.1. Qwt

Qt Widgets for Technical Applications to otwarta biblioteka umożliwiająca tworzenie wykresów oraz innych widgetów technicznych. Biblioteka ta była projektowana z myślą o zastosowaniach technicznych, w szczególności w systemach czasu rzeczywistego, przez co głównym celem było zapewne osiągnięcie wysokiej wydajności przy dynamicznie zmieniających się danych. Cel ten osiągnięto między innymi poprzez wykorzystywanie stosunkowo niskopoziomowych mechanizmów, jak Arthur czy intensywne wykorzystanie szablonów języka C++. Niestety przez to biblioteka nie jest szczególnie przyjazna użytkownikowi. Programista często musi sięgać do niskopoziomowych narzędzia, a stworzenie mniej standardowego rozwiązania może sprawić wiele trudności.

Qwt zawiera szerokie API służące do przeprowadzania różnorakich operacji na wykresach, m.in. skalowania i zaznaczania. Problematyczną kwestią jest wygląd wykresów, z jednej strony są one adekwatne dla zastosowań technicznych, z drugiej zaś sprawia on, że nie sposób użyć tych wykresów w aplikacji innego typu, ze względu na ich brzydotę. Najlepszym przykładem niech będzie wykres przedstawiony na rys. 2.1, wyglądający niczym oscylogram.



Rysunek 2.1. Przykładowy wykres Qwt

Zalety:
— rozbudowane API,
— wsparcie społeczności,
— wysoka wydajność,
— osie ze skalą logarytmiczną.

Wady:
— nieprzyjazna użytkownikowi,
— mało atrakcyjny wygląd wykresów,
— brak wsparcia dla QML.

2.2.2. GobChartWidget

Jest to jednoosobowy projekt rozwijany przez Williama Hallatta, dostępny na licencji open-source. Biblioteka ta ma bardzo ograniczoną funkcjonalność, pozwala na tworzenie wykresów tylko trzech typów: słupkowych, liniowych oraz kołowych, wszystkich o bardzo prostym wyglądzie.

Jest to kolejna biblioteka uzależniona od frameworku Model-Widok. Jest to jednak przypadek ekstremalny, nie polegający jedynie na trzymaniu danych w modelu. Klasy odpowiedzialne za prezentację wykresów dziedziczą tu po abstrakcyjnym widoku z ww. frameworku. Zapewne twórca uzyskał dzięki temu pewne ciekawe własności, jednak zmusiło go to do implementacji osobnych widoków dla każdego typu wykresu. Do rysowania wykresów wykorzystano tu GraphicsView, przy czym kontenerem na elementy nie jest scena, a model.

Jak widać GobChartWidget jest nieco dziwną hybrydą, której z pewnością nie można nazwać skalowalną. Wcale nie dziwi fakt, że nie cieszy się ona szczególną popularnością wśród programistów Qt. Według statystyk SourceForge [2], w ciągu pół roku pobrano tę bibliotekę zaledwie kilkanaście razy.

Zalety:
— darmowa,
— najpopularniejsze wykresy,
— współpraca z systemem Model-Widok.

Wady:
— ubogi zbiór wykresów,
— mało przejrzysta struktura,
— brak wspierającej społeczności programistów,
— brak wsparcia dla QML.

2.3. Dostępne typy wykresów

Podstawowym kryterium oceny użyteczności tego typu biblioteki jest wielkość zbioru dostępnych wykresów. Tablica 2.1 prezentuje typy wykresów, które można

[2] SourceForge `http://sourceforge.net`

znaleźć w opisanych bibliotekach. Podstawową informacją wynikającą z tego zestawienia jest fakt, że najpopularniejsze są wykresy kołowy, słupkowy oraz liniowy.

Tablica 2.1. Zestawienie dostępnych wykresów

	Qt Charts	KD Chart	Qtitan	Qwt	GobChart
Bąbelkowy	T	N	T	N	N
Gantta	N	T	N	N	N
Kołowy	T	T	T	N	T
Liniowy	T	T	T	T	T
Pierścieniowy	T	T	T	N	N
Słupkowy	T	T	T	N	T
Świecowy	T	T	N	N	N
Warstwowy	T	T	T	N	N
XY (punktowy)	T	N	T	T	N

2.4. Elementy wspólne

Analizując powyższe biblioteki doszedłem do wniosku, że można wykroić z nich część wspólną, stanowiącą podstawową funkcjonalność niezbędną dla biblioteki tego typu. Te elementy to:
— podstawowe wykresy, takie jak liniowy, słupkowy i kołowy,
— osie, siatka i legenda,
— możliwość realizacji przez programistę interakcji z użytkownikiem,
— zaznaczanie i przybliżanie fragmentów wykresu,
— serie danych – każda z bibliotek wykorzystuje ujednolicony interfejs do danych. Czesem seria jest jedynie opakowaniem na kontener z próbkami, innym razem zawiera większość logiki związanej z konstrukcją wykresu,
— efekty 2,5-3D, komercyjne biblioteki umożliwiają tworzenie pseudo-przestrzennych wykresów.

2.5. Elementy unikalne

Prawie wszystkie biblioteki zawierają wartościowe elementy niepowtarzalne lub rzadko spotykane:
— animacja towarzysząca tworzeniu wykresów oraz ich przebudowywaniu,
— motywy pozwalające na tworzenie wykresów w różnych, jednolitych stylach,
— możliwość realizacji pełnej interakcji ze wszystkimi elementami wykresu,
— możliwość konfiguracji elementu prezentującego kolor w legendzie (np. koło dla wykresu bąbelkowego, odcinek dla liniowego),
— generowanie plików graficznych zawierających wykresy,
— możliwość wyświetlania kilku wykresów różnego typu w jednym układzie współrzędnych,
— możliwość korzystania z nieliniowych skal osi przy wykresach osadzonych w układzie współrzędnych,
— wyeksponowanie klas C++ w QML.

2.6. Podsumowanie

Na rynku istnieją już wartościowe biblioteki pozwalające na tworzenie wykresów biurowych, są to jednak rozwiązania komercyjne. Projekty open-source są albo niskiej jakości albo nadają się do innych (technicznych) zastosowań. Istnieje więc potrzeba stworzenia wysokiej jakości darmowego produktu, który zyskałby taką popularność jak Qwt. Potrzeba ta jest jeszcze większa dla Qt Quick, gdzie nie ma żadnych darmowych bibliotek pozwalających na tworzenie wykresów.

Warto również zauważyć, że twórcy żadnej z opisanych bibliotek nie zdecydowali się na pełne odizolowanie silnika biblioteki od widoków. Dzięki takiemu podejściu osiągnąłem możliwość wyświetlania wykresów w dowolnym miejscu – wewnątrz widgetu, w aplikacji napisanej w QML jak i w dokumencie tekstowym. Prawdopodobnie jest to pierwsze takie rozwiązanie na rynku.

3. Opis wymagań

Istotą tego rozdziału jest opisanie wszystkich wymagań stawianych tworzonej bibliotece. Rozdział ten składa się z dwóch głównych części: opisu wymagań funkcjonalnych i opisu wymagań pozafunkcjonalnych.

Opis wymagań funkcjonalnych rozpoczynają wymagania stawiane całej bibliotece oraz wymagania odnoszące się do wszystkich tworzonych za jej pomocą wykresów. Następnie zostały przedstawione specyficzne wymagania dotyczące konkretnych typów wykresów.

Opis wymagań pozafunkcjonalnych został dokonany w kontekście całej biblioteki, a w jego skład wchodzą rozważania na temat takich zagadnień jak skalowalność, wydajność czy niezawodność.

3.1. Wykresy biurowe

Za pomocą projektowanej biblioteki programiści będą mieć możliwość tworzenia interaktywnych wykresów typu biurowego. Po lekturze rozdziału poświęconego przeglądowi dziedziny wiadomo już, że zbiór wykresów biurowych jest dość liczny. Z drugiej strony łatwo zauważyć, że jeden podzbiór wykresów jest szczególnie popularny. Projektując tę bibliotekę staram się skupić na stworzeniu elastycznej architektury, a nie na udostępnieniu jak największej liczby typów wykresów. W pierwszej wersji biblioteki zbiór wykresów będzie dośc ubogi i będzie zawierał jedynie te najpopularniejsze.

3.1.1. Dostępne wykresy

Biblioteka musi udostępniać następujące typy wykresów:
— kołowy,
— słupkowy,
— liniowy.

3.1.2. Inne wykresy

Architektura biblioteki musi umożliwiać dodawanie nowych typów wykresów biurowych. Mechanizmy takie jak współpraca ze źródłami danych lub podłączenie widoku do wykresu powinny być na tyle uniwersalne, aby nie wymagały jakichkolwiek modyfikacji dla wykresów nowych typów. W idealnym scenariuszu stworzenie nowego wykresu powinno ograniczyć się do zaimplementowania jego wewnętrznej logiki oraz wykorzystania już istniejących komponentów, np. legendy, bez ich modyfikacji.

3.2. Uniwersalny silnik

Podstawowym celem projektowanej biblioteki jest udostępnienie programistom uniwersalnego silnika umożliwiającego tworzenie interaktywnych wykresów. Ma to być rozwiązanie generyczne, działające zarówno dla klasycznego Qt jak i Qt Quick 2. Tworzony silnik powinien umożliwiać wyświetlenie wykresów w następujących miejscach:
— widgety,
— architektura Graphics View,
— dokumenty tekstowe,
— QML.

W tablicy 3.1 zostały podane stosowne dla każdego z przypadków wykorzystania widoki.

Tablica 3.1. Widoki kompatybilne z moją biblioteką

Miejsce	Klasa bazowa widoku
Widget	QWidget
Graphics View	QGraphicsObject
Dokument tekstowy	QTextFrame
QML	QQuickItem – klasy bazujące na SceneGraph QQuickPaintedItem – klasy bazujące na QPainter

3.3. Źródła danych

Typowym źródłem danych jest seria, czyli swego rodzaju pojemnik na próbki. Każde źródło danych, którym ma być zasilany wykres, musi udostępniać zbiór danych o następującej strukturze:

— jedna liczba ze specyficznego dla danego przypadku zbioru wartości, będącego podzbiorem zbioru liczb rzeczywistych,
— co najmniej jedna wartość z dziedziny liczb rzeczywistych. Bardziej złożone wykresy mogą mieć kilka takich wartości,
— tytuł próbki,
— kolor.

Zakładam, że serie zawierają maksymalnie jedną próbkę dla danego wektora z dziedziny.

Ponadto wymagany jest tytuł i kolor dla każdej z serii.

3.3.1. Serie danych

Istnieje potrzeba stworzenia prostego i lekkiego modelu danych reprezentującego serie, które z kolei zawierają próbki. Niezbędne są operacje dodawania, modyfikowania i usuwania danych z serii.

Aby ułatwić programistom Qt zapoznanie się ze sposobem działania mojej biblioteki, powinienem skorzystać tutaj ze statycznego polimorfizmu. Interfejs serii danych powinien przypominać interfejs modelu z architektury Model-Widok.

3.3.2. Model–Widok

Wykresy mają być alternatywą dla widoków z architektury Model-Widok. Uważam, że typowym przypadkiem użycia będzie prezentowanie tych samych danych za pomocą wykresów oraz tabelek. Z tego powodu wszystkie wykresy muszą przyjmować jako źródło danych obiekty klas dziedziczących po *QAbstractItemModel*.

3.4. Interaktywność

Celem tej pracy jest stworzenie biblioteki „do operowania na wykresach". Wszystkie wykresy muszą obsługiwać interaktywne operacje, a część z nich powinna zostać zaimplementowana. Pozostałe powinny być możliwe do realizacji.

3.4.1. Zaznaczanie

Musi istnieć możliwość zaznaczania elementów reprezentujących dane. Powinna istnieć możliwość zaznaczania zarówno elementów reprezentujących pojedyncze próbki, np. wycinek kołowy, jak i tych tych odpowiadających całym seriom danych, np. grupa słupków. Zaznaczanie powinno być możliwe poprzez kliknięcie przyciskiem myszy lub dotknięcie ekranu dotykowego.

3.4.2. Modyfikacja zawartości modelu

Powinna istnieć możliwość zmiany zawartości modelu poprzez interakcję z wykresem. Dla danego elementu reprezentującego próbkę, zmiana jego parametru proporcjonalnego do reprezentowanej wartości powinna skutkować zmianą danych zawartych w modelu. I tak, skrócenie słupka powinno spowodować zmniejszenie odpowiedniej wartości w modelu, a zwiększenie kąta rozwarcia wycinka kołowego zwiększenie analogicznej wartości.

3.4.3. Inne operacje

Architektura biblioteki powinna umożliwiać programistom realizację innych interaktywnych operacji, np. system *Przeciągnij i Upuść*. Powinno być to możliwe poprzez wykorzystanie systemu zdarzeń Qt.

3.5. Qt Quick

Projektując tę bibliotekę muszę wziąć pod uwagę jej późniejsze zastosowanie, którym ma być m.in. tworzenie wykresów w QML. Projekt powinien zawierać również popularny w QML mechanizm delegatów.

3.5.1. Wyeksponowanie klas C++ w QML

Już na etapie projektowania należy zadbać, aby tworzone struktury były łatwe do wyeksponowania w QML. Tworzenie interfejsów do QML nie jest celem tego projektu, jednak ich implementacja powinna być możliwie łatwa dla programistów decydujących się na korzystanie z mojej biblioteki.

3.5.2. Delegaty

Szeroko stosowanym w Qt Quick mechanizmem są delegaty. Umożliwiają one zdefiniowanie przez programistę sposobu prezentacji danych z modelu. W mojej bibliotece korzystanie z delegatów powinno być możliwe w legendzie, do prezentacji kolorów, oraz we właściwym wykresie do definiowania własnych elementów prezentujących zawartość modelu, np. słupki.

3.6. Wspólne elementy składowe

Wszystkie wykresy muszą zawierać następujące elementy:
— źródło danych,
— elementy prezentujące pojedyncze próbki danych (słupek, punkt, wycinek kołowy),
— tytuł wykresu,
— legenda,
— dodatkowe elementy dostarczane przez programistów.

Wyświetalnie każdego z elementów wykresu powinno być sterowane przez programistę. Powinna również istnieć możliwość ustawienia tła wykresu.

3.6.1. Elementy prezentujące dane

Każdy z wykresów posiada specyficzny dla niego element służący do prezentacji danych z próbki, którego rozmiar lub położenie w układzie współrzędnych odzwierciedla wartość próbki. Każdy z elementów może mieć swój podpis. Powinna istnieć możliwość ustawienia tym elementom dwóch piór i dwóch pędzli – dla trybu normalnego i zaznaczenia. Programista powinien mieć możliwość podmiany, dla danego wykresu, klasy takiego elementu na własną, przy czym odpowiedzialność za poprawne odrysowanie się wykresu spada wtedy na programistę.

3.6.2. Tytuł wykresu i podpisy elementów

Dla wszelkich napisów będących składowymi wykresu musi być możliwość ustawienia ich treści, czcionki oraz koloru.

3.6.3. Legenda

Dla wykresów obsługujących wiele serii danych legenda powinna prezentować kolory oraz tytuły tych serii. Natomiast dla wykresów jednoseryjnych prezentowana powinna być informacja o kolorze i tytule każdej z próbek. Legenda powinna przyjmować jedno z dwóch położeń – poziome lub pionowe. Powinna istnieć możliwość zmiany elementu prezentującego kolor za pomocą mechanizmu delegatów.

3.6.4. Dodatkowe elementy

Programiści powinni mieć możliwość tworzenia własnych elementów i dodawania ich do wykresów już istniejących klas. Aby to osiągnąć powinni jedynie zaimplementować odpowiednie interfejsy.

3.7. Skalowanie

Powinno być możliwe skalowanie wykresu. Wykres powinien dostosowywać swój rozmiar do przekazanego mu obszaru przeznaczonego na jego odrysowanie.

3.8. Wykresy w układzie współrzędnych

Niektóre wykresy wymagają osadzenia w układzie współrzędnych. Takie wykresy wymagają dodatkowych elementów:

— osie,
— siatka.

Powinna istnieć możliwość przypisania do osi skali innej niż liniowa. Musi być możliwe ustawienie pióra służącego do rysowania osi, jej tytułu, gęstości ticków oraz zakresu wartości. Powinna istnieć możliwość określenia grubości i koloru linii oraz ziarnistości samej siatki.

3.8.1. Układy współrzędnych

Mimo, iż dziedzinie wykresów najpopularniejszym układem współrzędnych jest układ kartezjański, w projekcie należy uwzględnić istnienie innych układów współrzędnych takich jak biegunowy czy cylindryczny.

3.8.2. Wspólny układ współrzędnych

Powinna istnieć możliwość wyświetlania kilku wykresów, również różnych typów, w jednym układzie współrzędnych.

3.9. Wykres kołowy

Wykres kołowy służy do prezentacji danych z jednej serii. Każda z próbek jest prezentowana za pomocą wycinka kołowego o kącie środkowym proporcjonalnym do prezentowanej wartości, przez co muszą to być wartości rzeczywiste dodatnie. Wartości wszystkich próbek serii sumują się do stu procent, a suma kątów wewnętrznych wycinków wynosi 360 stopni. Wszystkie wycinki z danej serii mają wspólny środek oraz jednakowy promień.

3.9.1. Przemieszczanie wycinka

Dowolny z wycinków powinno się dać przesunąć o zadaną część jego promienia. Proces ten powinien być możliwy do animacji z zastosowaniem standardowych rozwiązań Qt.

3.9.2. Obracanie wykresu

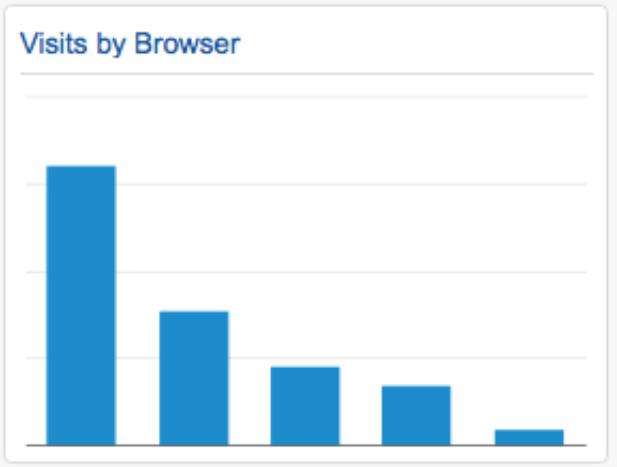
Powinna być możliwość obracania wykresu wokół jego środka. Kąt obrotu powinien być dowolną całkowitą wartością, o jednostce wynoszącej $\frac{1}{16}$ stopnia – jest to standardowa w Qt jednostka. Proces ten powinien być możliwy do animacji.

3.10. Wykres słupkowy

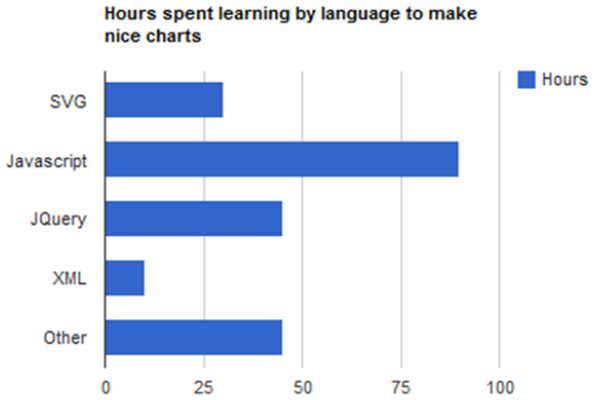
Jest to wykres służący do prezentacji danych z wielu serii. Elementem odpowiedzialnym za prezentację pojednyczej próbki jest tu słupek. Wszystkie słupki danej serii mają ten sam kolor. W ogólnym przypadku za pomocą tego wykresu można prezentować wartości z dziedziny liczb rzeczywistych.

3.10.1. Układ słupków

Wykres słupkowy może przyjmować układ pionowy bądź poziomy. Wykres z pionowym układem słupków jest nazywany wykresem kolumnowym i został przedstawiony na rysunku 3.1. Natomiast przykładowy wykres z poziomym układem znajduje się na rysunku 3.2.



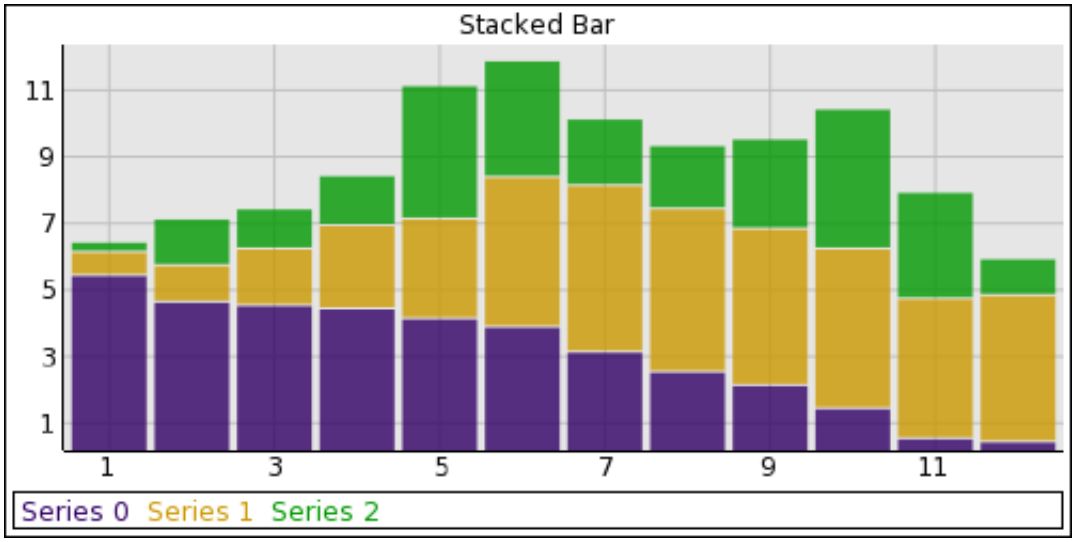
Rysunek 3.1. Pionowy układ słupków



Rysunek 3.2. Poziomy układ słupków

3.10.2. Stos

Wykres słupkowy może zostać przedstawiony w trybie stosowym. Wtedy dla każdej wartości z dziedziny tworzony jest jeden słupek o wysokości proporcjonalnej do sumy wartości próbek ze wszystkich serii wykresu. Z kolei ten słupek jest podzielony na mniejsze cześci o długościach i kolorach odpowiednich próbek. Na rysunku 3.3 przedstawiam przykład stosowego wykresu słupkowego.



Rysunek 3.3. Stosowy wykres słupkowy

3.11. Wykres liniowy

W wykresie liniowym dana próbka jest prezentowana za pomocą pojedynczego punktu. Punkty są prezentowane jako koła o zadanym promieniu. Wartości próbek należą do zbioru liczb rzeczywistych. Punkty danej serii mogą zostać połączone w łamaną.

3.11.1. Łączenie punktów

Programista powinien mieć możliwość podjęcia wyboru w kwestii łączenia punktów w łamaną. Powinien mieć również możliwość zmiany wszystkich parametrów łamanej, tak jak dla elementów odpowiedzialnych za prezentację próbek.

3.12. Dodatki

Wszystkie opisane tu funkcjonalności są opcjonalne, a ich realizacja nie jest konieczna do zakończenia prac nad biblioteką.

3.12.1. Motywy

Dodatkiem, który podniósłby atrakcyjność wykresów jest wysokopoziomowy mechanizm motywów, podobny do *QStyle*. Mechanizm ten powinien umożliwiać tworzenie wykresów o spójnej kolorystyce oraz czcionkach. Zmiana motywu dla danego wykresu powinna sprowadzać się do prostej operacji.

3.12.2. Budowanie wykresu

Powinno być możliwe animowanie procesu budowania wykresu. Podczas tego procesu kolejne elementy wykresu będą stawały się widoczne, a elementy odpowiedzialne za prezentację danych powinny stopniowo przyjmować swoje wartości, począwszy od zera.

3.12.3. Generowanie plików graficznych

Powinno być możliwe generowanie na podstawie istniejących wykresów plików graficznych w formatach PNG i SVG.

3.13. Przenośność

Biblioteka musi wpisywać się w politykę Qt brzmiącą: *pisz raz, kompiluj wielokrotnie*. Musi być przenośna na poziomie kodu źródłowego między najważniejszymi wspieranymi przez Qt platformami. Minimum to uruchomienie na systemach:
— Windows,
— Linux.

3.14. Wymagania pozafunkcjonalne

Wymagania funkcjonalne nie są jedynymi kryteriami oceny biblioteki. Równie ważne są wymagania definiujące oczekiwania użytkownika na temat budowy biblioteki oraz wynikające z niej ograniczenia dotyczące wydajności czy skalowalności. Poniżej poruszam te i kilka innych kwestii nie związanych bezpośrednio z funkcjonalnością.

3.14.1. API w stylu Qt

Aby tworzony przeze mnie kod był czytelny dla innych programistów Qt, musi on wykorzystywać standardowe mechanizmy tej platformy:
— statyczny polimorfizm, polegający na tworzeniu podobnych interfejsów dla podobnych, ale niespokrewnionych klas, np. kontenerów. Zastępuje wprowadzanie sztucznych klas bazowych.
— właściwości jako sposób na parametryzowanie obiektów,
— preferowanie przyjmowania wskaźników zamiast referencji do funkcji modyfikujących argumenty,
— asynchroniczna komunikacja między obiektami rozwiązana za pomocą sygnałów i slotów,
— nazewnictwo, sposób zwracania wartości z funkcji i wiele innych opisanych w [1].

3.14.2. Wymienność biblioteki

Biblioteka powinna wykorzystywać mechanizmy pozwalające na tworzenie bibliotek dynamicznych wymiennych pomiędzy wersjami. Wprowadzenie nowej wersji biblioteki z niezmienionym interfejsem nie powinno wymagać przebudowania całej aplikacji z niej korzystającej.

3.14.3. Nowoczesność i uniwersalność

Biblioteka powinna wykorzystywać możliwie nowe technologie, np. Qt5. Biblioteka może korzystać ze standardu *C++11*, jednak nie powinna wymuszać na użytkowniku posiadania kompilatora zgodnego z tym standardem. Komponenty dostarczane do użytku programistom powinny być możliwie wysokopoziomowe i uniwersalne w użyciu.

3.14.4. Wydajność

Jako, że okoliczności wykorzystania wykresów biurowych są inne niż wykresów technicznych oraz natura ich danych jest dużo bardziej statyczna, optymalizacja nie jest tu kwestią najważniejszą. Z tego powodu, oraz z chęci uniknięcia antywzorca projektowego przedwczesnej optymalizacji, kwestia wydajności bilioteki jest odsuwana na dalszy plan.

3.14.5. Niezawodność

Jak już wspomniałem w poprzednim punkcie, natura oraz zastosowania wykresów biurowych różnią się od technicznych, a co za tym idzie, mają również inne wymagania dotyczące niezawodności. Przewiduje się, że biblioteka będzie przeznaczona do aplikacji finansowych i biurowych, a nie systemów czasu rzeczywistego. Tym niemniej, w celu minimalizacji liczby błędów w kodzie, powinny zostać zastosowane testy regresji. Zachowanie biblioteki w warunkach ekstremalnych, np. ograniczonego dostępu do zasobów, nie jest głównym celem projektu.

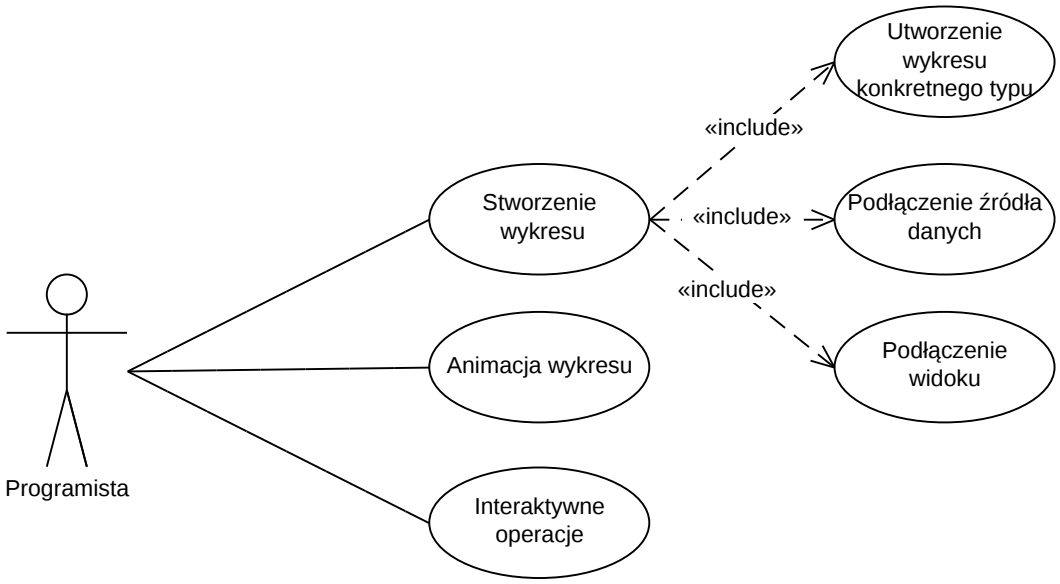
3.14.6. Skalowalność

Zarówno dodawanie nowych jak i usuwanie już istniejących elementów biblioteki powinno być łatwe i nie powinno mieć wpływu na stabilność pracy biblioteki. Dodawanie nowych elementów powinno być możliwe dzięki uniwersalnym interfejsom. Natomiast usuwanie istniejących elementów powinno sprowadzać się do wyłączenia ich z procesu budowania biblioteki.

4. Analiza wymagań

Mimo iż poprzedni rozdział daje już dość klarowny obraz celu, do jakiego dążę przy projektowaniu tej biblioteki, w niniejszym rozdziale dokonuję analizy niektórych z wymagań. Rozpoczynam od diagramu poziomu zero 4.1, który zawiera wszystkie najważniejsze wymagania wysokiego poziomu. Kolejne podrozdziały zawierają tekstowy opis scenariuszy.

W rozdziale tym starałem się nie nadużywać diagramów, gdyż „diagramy przypadków użycia i związki pomiędzy przypadkami mają drugorzędne znaczenie w procesie analizy wymagań. Przypadki użycia to dokumenty tekstowe. Tworzenie przypadków użycia sprowadza się do pisania tekstu". Autorem tych słów jest Craig Larman, autor książki poświęconej projektowaniu oprogramowania [7].



Rysunek 4.1. Diagram przypadków użycia poziomu zero

4.1. Stworzenie wykresu

Stworzenie oraz wyświetlenie wykresu prezentującego dane z określonego źródła składa się z trzech głównych etapów, które postanowiłem opisać nieco bardziej szczegółowo.

4.1.1. Utworzenie wykresu konkretnego typu

1. Programista wybiera typ wykresu i tworzy obiekt tego typu.
2. Jeśli dany wykres istnieje w kilku rodzajach, programista wybiera jeden z nich. Wykres domyślnie przyjmuje określony rodzaj.
3. Programista ustawia wartości parametrów wykresu związanych z jego odrysowywaniem, np. włączenie antyaliasingu.
4. Programista ustawia parametry wykresu charakterystyczne dla jego typu.

4.1.2. Podłączenie źródła danych

1. Programista rejestruje wybrany przez siebie typ źródła danych jako *QVariant*.
2. Programista tworzy źródło danych tego typu.
3. Programista wykorzystuje metodę wykresu do połączenia źródła danych z wykresem.
4. Programista uzupełnia źródło danymi.
5. Wykres reaguje na zmianę danych poprzez ponowne odrysowanie.

4.1.3. Podłączenie widoku

1. Programista wybiera widok, w którym odrysowywany będzie wykres.
2. Programista tworzy klasę dziedziczącą po odpowiedniej klasie widoku, zgodnie z tablicą 3.1.
3. Programista realizuje opisany w rozdziale *Projekt* protokół komunikacji pomiędzy widokiem a wykresem.

4.2. Animacja wykresu

Animacja elementów wykresu ma być możliwa poprzez wykorzystanie standardowych mechanizmów Qt.

1. Programista wybiera element oraz jego właściwość, której wartość ma podlegać animacji.
2. Programista wybiera interesujący go typ animacji, ustawia jej parametry i aktywuje na wybranej właściwości elementu wykresu.
3. Kolejne zmiany wartości właściwości powodują powiadomienie widoku wykresu o potrzebie odrysowania.
4. Wykres jest odrysowywany z uwzględnieniem zmian wynikających z animacji.

4.3. Interaktywne operacje

Poniżej opisuję scenariusz potyczący dowolnej interaktywnej operacji możliwej do zaimplementowania w ramach zaprojektowanej przeze mnie architektury.

1. Programista odblokowuje daną interaktywną operację dla danej instancji wykresu.
2. Widok dostarcza przeznaczone dla wykresu zdarzenie.
3. Wykres decyduje, którego z elementów dotyczy zdarzenie.
4. Wykres decyduje czy dane zdarzenie powoduje zmianę w jego stanie, jeśli tak to podejmuje odpowiednią akcję.

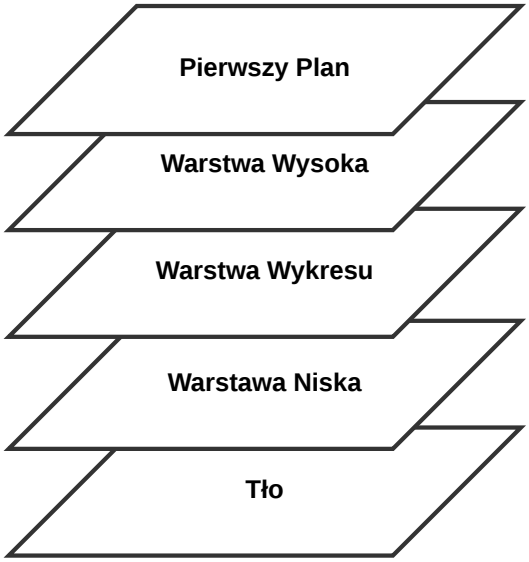
5. Wykres powiadamia widok o potrzebie odrysowania.
6. Wykres jest odrysowywany z uwzględnieniem zmian wynikających z interakcji.

5. Projekt

Rozdział ten zawiera inżynierski projekt rozwiązań dla wymagań z poprzednich rozdziałów. Poza opisem tekstowym, zastosowałem w nim diagramy klas oraz sekwencji UML.

5.1. Podział na warstwy

Zdecydowałem się podzielić wykres na pięć warstw. Odrysowywanie wykresu polega na odrysowaniu zawartości każdej z warstw, począwsze od tła, a na pierwszym planie skończywszy. Układ warstw został przedstawiony na rysunku 5.1.



Rysunek 5.1. Warstwy wykresu

Warstwy tła oraz pierwszego planu służą jedynie odrysowywaniu wzorów przekazanych za pomocą pędzla. Pozostałe warstwy są bardziej złożone i zawierają liczne elementy.

Warstwy niska i wysoka są odrysowywane zgodnie z algorytmem malarza. Służą dodawaniu przez programistów własnych elementów do wykresów istniejących klas.

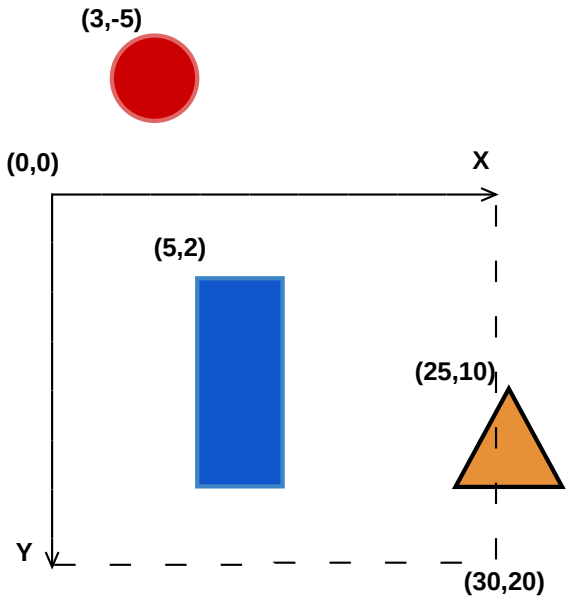
Warstwa wykresu służy odrysowaniu głównej zawartości wykresu. Tym etapem steruje strategia danego wykresu. W ogólnym przypadku programiści nie powinni dodawać własnych elementów do tej warstwy.

Metoda odpowiedzialna za odrysowywanie wykresu jest *Metodą Szablonową* [5], czyli niewirtualną, publiczną metodą klasy bazowej, wywołującą kolejne metody odpowiedzialne za odrysowanie pojedynczych warstw. Metody te są wirtualne

i chronione. Metody odpowiedzialne za tło i pierwszy plan udostępniają domyślną implementację, natomiast pozostałe są czysto wirtualne.

5.2. Lokalny układ współrzędnych

Każdy z wykresów posiada lokalny, kartezjański układ współrzędnych rzeczywistych. Ponadto granice wykresu są wyznaczane przez specjalny prostokąt. Takie podejście umożliwi układanie elementów względem lokalnego, a nie globalnego układu współrzędnych, oraz ustalanie, które z elementów są widoczne i należy je odrysować. Dodatkowo ułatwiona to realizację takich wymagań jak: skalowanie wykresu czy interaktywność.



Rysunek 5.2. Lokalny układ współrzędnych

Na rysunku 5.2 przedstawiono przykładowy układ współrzędnych wykresu o wymiarach 30x20. W tej sytuacji prostokąt zostanie wyświetlony w całości, trójkąt tylko częściowo, a koło wcale nie zostanie wyświetlone.

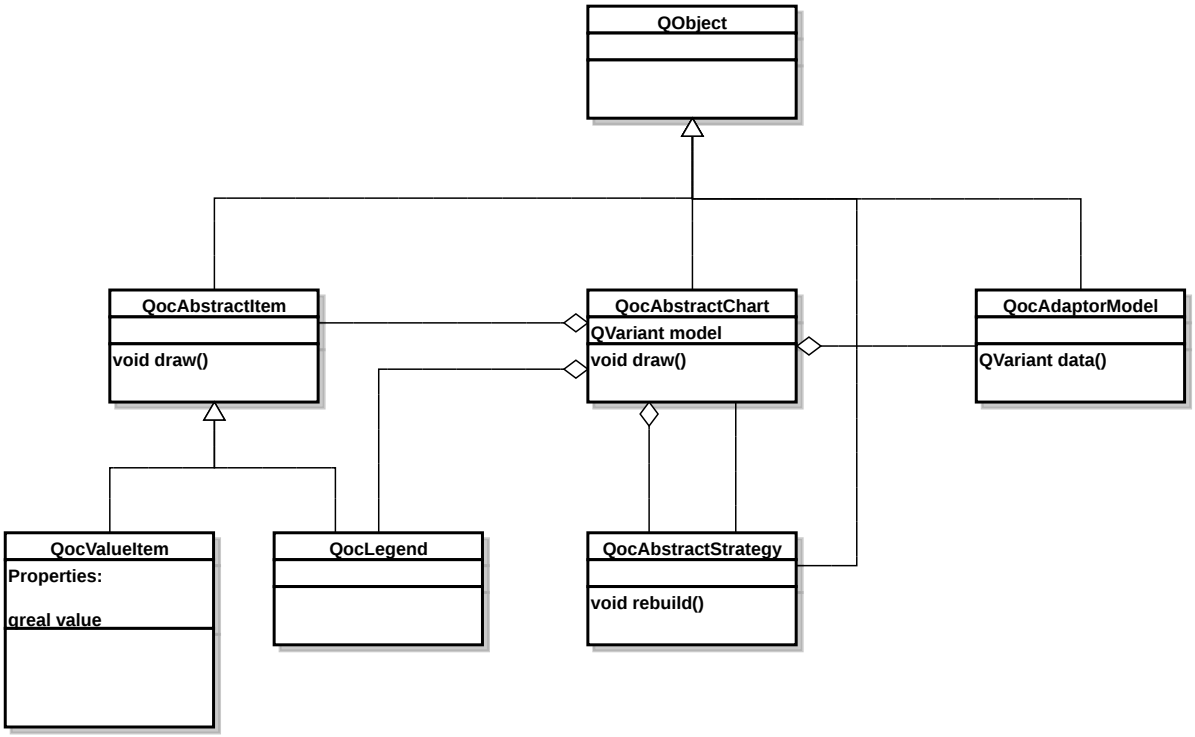
5.3. Struktura wykresu

Ogólna koncepcja na strukturę wykresu została przedstawiona na diagramie 5.3.

Klasy pochodne od QocAbstractChart są miejscem łączącym wszystkie inne elementy. To na nich są ustawiane parametry odnoszące się do całości wykresu, np. włączenie antyaliasingu.

QocSeries odpowiada za dane dostarczane do wykresu. Jest lekkim odpowiednikiem modelu z architektury Model-Widok.

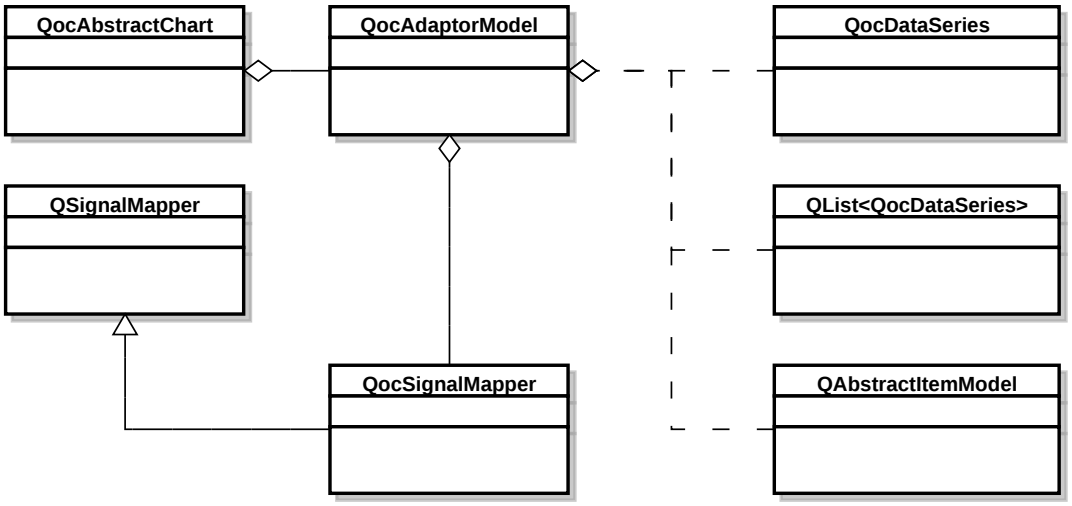
QocAbstractStrategy to klasa bazowa dla strategii – layoutu, odpowiedzialnego za odpowiednie układanie elementów wykresu, głównie z warstwy środkowej – *ChartLayer*, i ich odrysowywanie. Jest to szczególnie przydatny komponent dla wykresów, które mogą być budowane na różne sposoby, np. wykres słupkowy.



Rysunek 5.3. Diagram top level

5.4. Źródła danych

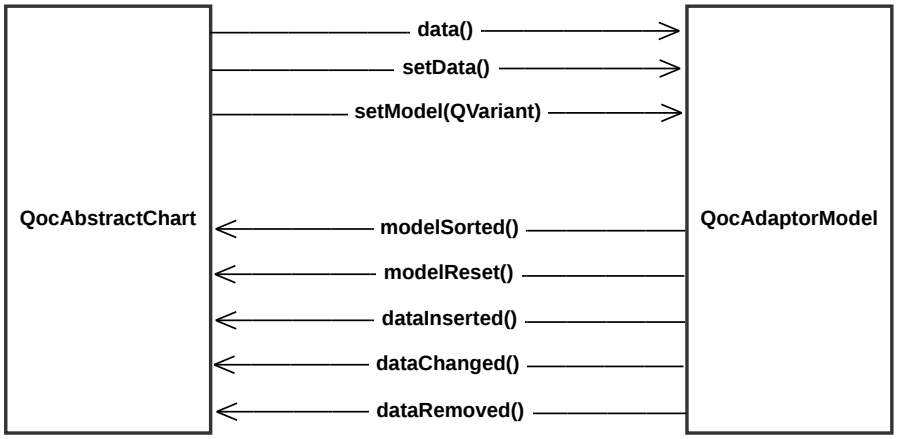
Jako że źródłem danych dla wykresu może być seria danych, lista serii albo model, postanowiłem wprowadzić pośrednią klasę, która będzie odpowiedzialna za unifikację komunikacji wykresu ze źródłem danych. Klasa ta jest *Adapterem obiektowym* [5] – agreguje źródła danych dostarczane jako obiekty *QVariant*. Hierarchię klas związanych z adapterem przedstawiłem na diagramie 5.4



Rysunek 5.4. Adapter

Za pomocą adaptera uzyskałem pewną abstrakcję, dzięki której niezależnie od klasy źródła danych, wykres zawsze będzie go postrzegał jako zbiór serii. Większość metod adaptera, jako jeden z argmuentów przyjmuje numer serii, dzięki któremu można wyspecyfikować, której serii dotyczy dana operacja.

Komunikacja w drugą stronę została rozwiązana za pomocą zbioru sygnałów, które również niosą informację o numerze serii. W przypadku zbioru serii, wystarczy uzupełnić ich sygnały o numer serii oraz przepropagować na zewnątrz adaptera. Chcę to osiągnąć za pomocą obiektu klasy pochodnej od *QSignalMapper* [1]. W przypadku klas modeli, również muszę wykonać podobne mapowanie, podczas którego trzeba sięgnąć do modelu po numer serii. Przykładowy protokół komunikacji wykresu z adapterem przedstawiłem na rysunku 5.5.



Rysunek 5.5. Komunikacja Wykres – Adapter

5.4.1. Role

Wzorując się na systemie Model-Widok, zdecydowałem się wprowadzić do mojej biblioteki byt o nazwie: Rola. Rola to typ wyliczeniowy, który służy do wskazania przez widok, jakich danych oczekuje od modelu. Widok chcąc uzyskać dane z modelu musi mu przekazać indeks i właśnie rolę.

Dzięki rolom uzyskałem stan, w którym wykres chcąc pobrać dane z modelu, przekazuje do adaptera rolę. Adapter zajmuje się przetłumaczeniem roli. W zależności od kontekstu, może to być numer kolumny, rola albo właściwość obiektu.

Wydruk 5.1. Rola – typ wyliczeniowy

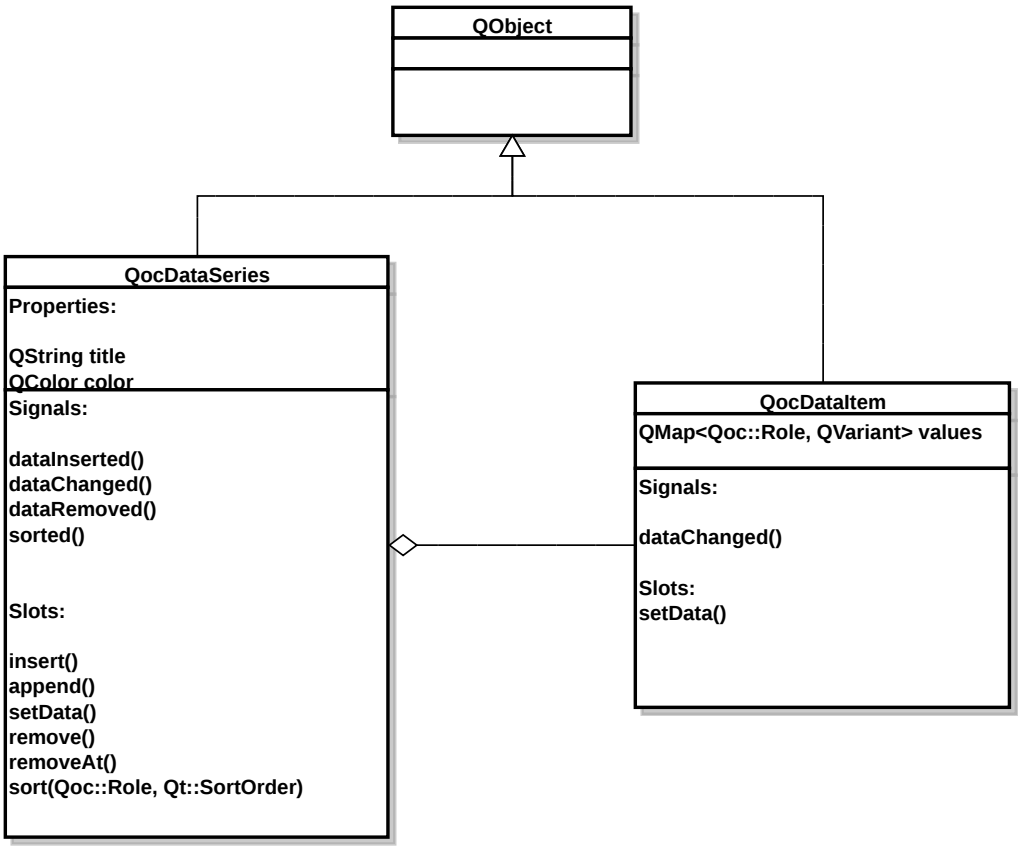
```
enum Role{
        XRole,
        YRole,
        TitleRole,
        ColorRole,
        CustomRole
}
```

[1] QSignalMapper `http://qt-project.org/doc/qt-5.0/qtcore/qsignalmapper.html`

5.4.2. Seria danych

Źródłem danych dla wykresu może być pojedyncza seria lub lista serii. Aby było to możliwe, typy *QocSeries* i *QList<QocSeries>* muszą zostać zarejestrowane jako *QVariant*.

Próbka to w dużej mierze mapa, w której kluczem jest rola. Dzięki wsparciu QML dla tego typu map [2] modyfikacja zawartości próbki jest tak samo prosta i intuicyjna w C++ oraz w QML. Diagram 5.6 zawiera najważniejsze elementy klas odpowiedzialnych za serie oraz próbki w mojej bibliotece.



Rysunek 5.6. Seria i próbki

5.4.3. QAbstractItemModel

Pracując na seriach danych, które są mojego autorstwa, mogę założyć, że dla elementu o danym indeksie i roli wartość jest, np. tytułem zapisanym w łańcuchu znaków. Takiego komfortu jednak nie będzie przy współpracy z dowolnym innym modelem. Nie mogę wymagać od użytkownika, aby interesujące mnie dane trzymał w kolumnach o ustalonych przeze mnie indeksach. Rozwiązaniem tej sytuacji mogłoby być stworzenie specjalnego proxy modelu [3], jednak preferuję inny sposób.

[2] Typ variant `http://qt-project.org/doc/qt-5.0/qtqml/qml-variant.html#storing-arrays-and-objects`

[3] Proxy modele `http://qt-project.org/doc/qt-5.0/qtcore/qabstractproxymodel.html`

Moje rozwiązanie polega na mapowaniu ról, które są zrozumiałe dla wykresów, na numery kolumn modeli do tych wykresów podłączanych. W przypadku jednowymiarowych modeli dziedziczących po *QAbstractListModel* role wykresu będą mapowane na role tychże modeli. Z perspektywy programisty mapowanie ogranicza się jedynie do wywołania poniższej metody dla wszystkich wymagających tego ról:

```
QocAbstractChart::
setRoleMapping(Qoc::Role role, int customValue);
```

Metoda ta przyjmuje jako pierwszy z argumentów rolę, a jako drugi numer kolumny, pod którą kryją się odpowiednie dane w modelu.

5.4.4. QML

ListModel i XmlListModel to pochodne *QAbstractListModel*, które należy potraktować osobno. Wprowadzają one bardziej obiektowe podejście, dzięki czemu ich zawartość jest łatwo dostępna z QML.

Dane z tych modeli są dostępne również poprzez system właściwości. Aby dostać się do wartości właściwości wystarczy jedynie znać jej nazwę. Dzięki temu, wymuszając na użytkownikach określone nazwy właściwości, jestem w stanie pobierać dane z ich modeli. W skrajnym przypadku, gdy programista korzystający z mojej biblioteki nie będzie mógł dostarczyć danych w oczekiwanym przeze mnie formacie, udostępniam mu przeładowaną metodę z poprzedniego podpunktu:

```
QocAbstractChart::
setRoleMapping(Qoc::Role role, const QString &name);
```

W tym przypadku jako drugi argument przyjmuję nową nazwę danej właściwości. Po wywołaniu metody, nowa nazwa właściwości będzie wykorzystywana do komunikacji z modelem.

5.5. Qt Quick

Wykorzystanie mojej biblioteki w Qt Quick wymaga pewnego narzutu pracy. Poniżej opisuję mechanizm umożliwiający wykorzystanie moich klas w QML oraz przyjętą przeze mnie konwencję interfejsów dla QML.

5.5.1. Eksport klas C++ do QML

Klasy wszystkich wysokopoziomowych elementów są pochodnymi klasy QObject. Wszelkie ich parametry, które powinny być konfigurowalne dla programistów włączyłem do zbioru właściwości tych klas. Ważne jest, aby przy zmianie wartości danej właściwości, i tylko wtedy, emitować sygnał informujący o tym zdarzeniu. Jest to niezbędne do poprawnego działania mechanizmu wiązania w QML. Ponadto, wszystkie metody, które powinny być dostępne z QML, a nie są slotami, zostają opatrzone makrem *Q_INVOKABLE*.

Większość klas powinna jest gotowa do wyeksportowania ich do QML za pomocą standardowej procedury, np. poprzez wywołanie funkcji szablonowej

```
template<typename T>
int qmlRegisterType(const char *uri, int versionMajor,
```

```
                    int versionMinor, const char *qmlName)
```

Parametrem szablonu jest eksportowany typ, a parametry funkcji to nazwa modułu, dwie liczby odpowiadające za wersję modułu oraz nazwa pod jaką będzie dostępna eksportowana klasa z poziomu QML. Qt udostępnia jeszcze kilka innych, specjalizowanych szablonów, np. dla singletonów.

Pozostałe klasy, do wykorzystania ich w QML, będą wymagają specjalnych interfejsów. Dla klas związanych z GUI, bazujących na QPainter będzie to QQuickPaintedItem, a dla tych wykorzystujących SceneGraph – QQuickItem.

5.5.2. Uproszczony interfejs

Aby zapobiec nadmiernemu rozrostowi interfejsów klas dostępnych w QML, zdecydowałem się je uprościć w porównaniu do tych dostępnych z poziomu C++. Jest to konwencja szeroko stosowana w QML. Najlepszym przykładem będzie ramka wokół prostokąta. Z poziomu C++ istnieje możliwość ustawienia pióra używanego do jej odrysowania. Klasa pióra, czyli *QPen*, nie dziedziczy po *QObject*, więc nie może być dostępna w QML. W QML można zmienić jedynie kolor oraz grubość ramki.

5.6. Interaktywność

Wszystkie operacje interaktywne zostały rozwiązane za pomocą systemu zdarzeń Qt [4]. Widoki mają obowiązek przekazywać wszelkie przeznaczone dla wykresu zdarzenia, np. pojawienie się kursora myszy nad wykresem czy kliknięcie.

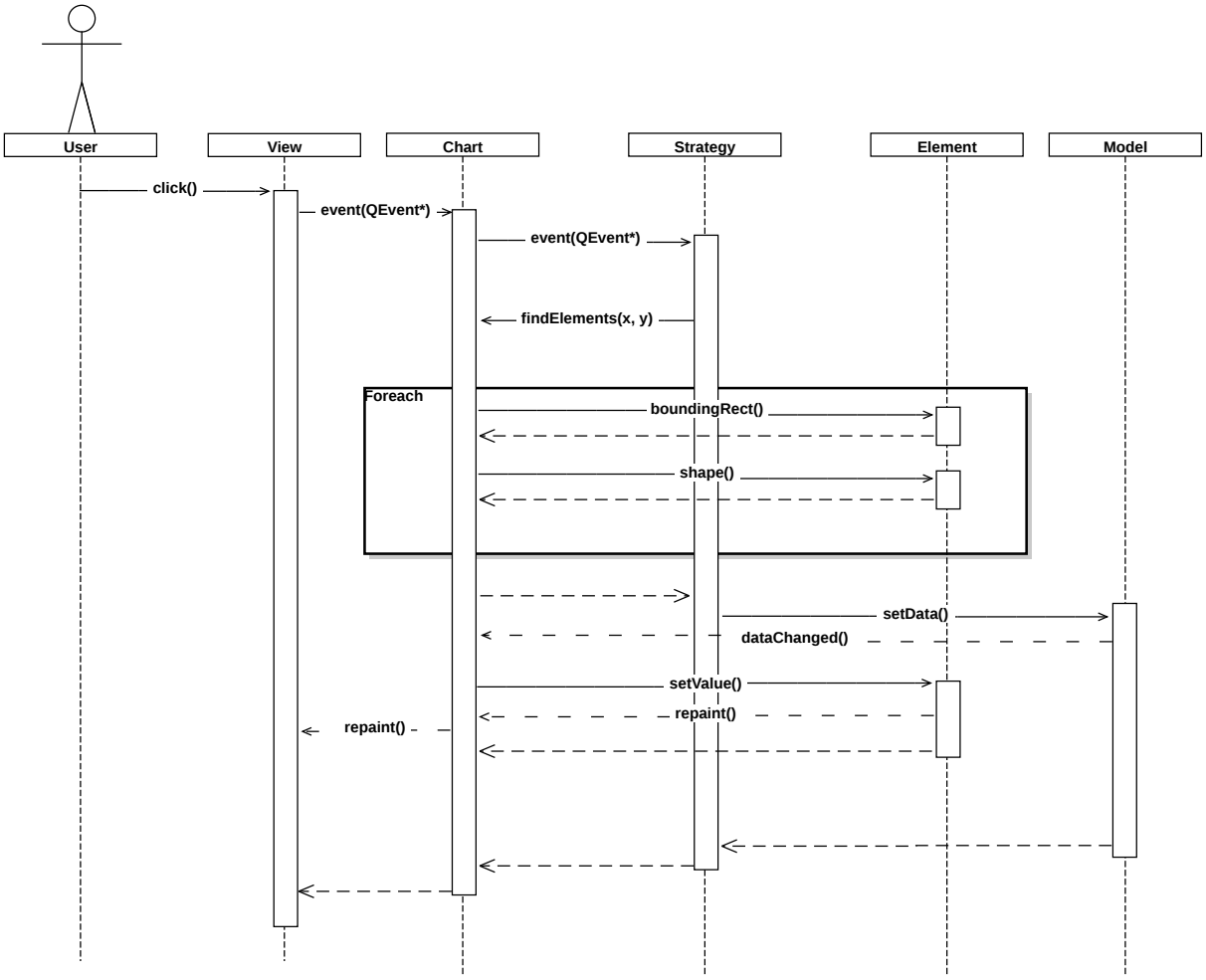
5.6.1. Zaznaczanie

Aby zrealizować zaznaczanie elementów wykresu z poziomu GUI, do wykresu muszą być przekazywane zdarzenia związane z kursorem myszy oraz dotykiem. Pojedyncze kliknięcie w element reprezentujący dane skutkuje jego zaznaczeniem. Podwójne kliknięcie takiego elementu spowoduje zaznaczenie wszystkich elementów danej serii. Zaznaczenie powoduje zmianę koloru obramowania wokół danego elementu. Kolor zaznaczenia jest zdefiniowany jako właściwość całego wykresu.

5.6.2. Zmiana wartości w modelu

Na rysunku 5.7 przedstawiam diagram sekwencji dotyczący zmiany zawartości modelu poprzez modyfikację elementów widoku. Przerywane strzałki zawierające opis symbolizują wywołanie sygnału.

[4] System zdarzeń Qt `http://qt-project.org/doc/qt-5.0/qtcore/eventsandfilters.html`

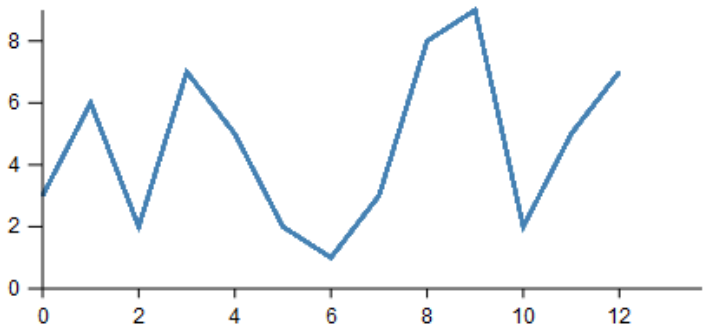


Rysunek 5.7. Interaktywna zmiana zawartości modelu

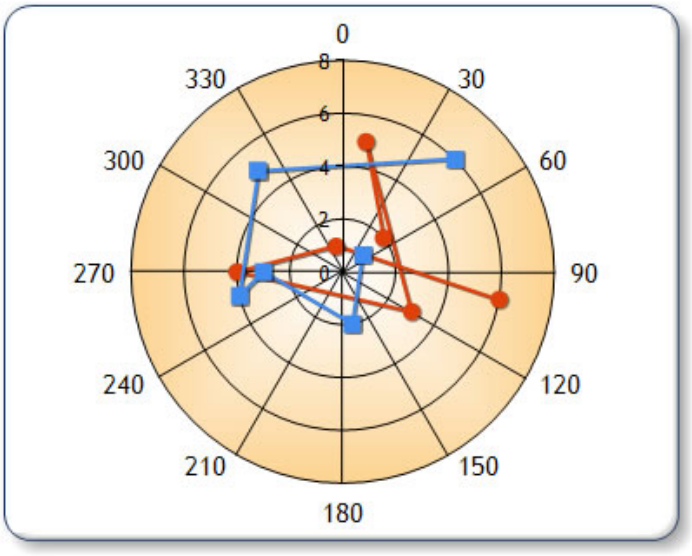
5.7. Wykresy w układzie współrzędnych

Większość wykresów jest osadzona w pewnym układzie współrzędnych. Najpopularniejszym z nich jest układ współrzędnych kartezjańskich 5.8, jednak jak wiadomo nie jest to jedyny układ współrzędnych. Moje rozwiązanie przewiduje możliwość tworzenia wykresów w bardziej egzotycznych układach, np. wykres polarny w układzie biegunowym 5.9.

Mimo, iż najpopularniejszą skalą osi, stosowaną przy wykresach biurowych jest skala liniowa, moje rozwiązanie przewiduje możliwość wykorzystania innej skali, np. logarytmicznej, która jest dość często stosowana przy wykresach liniowych.

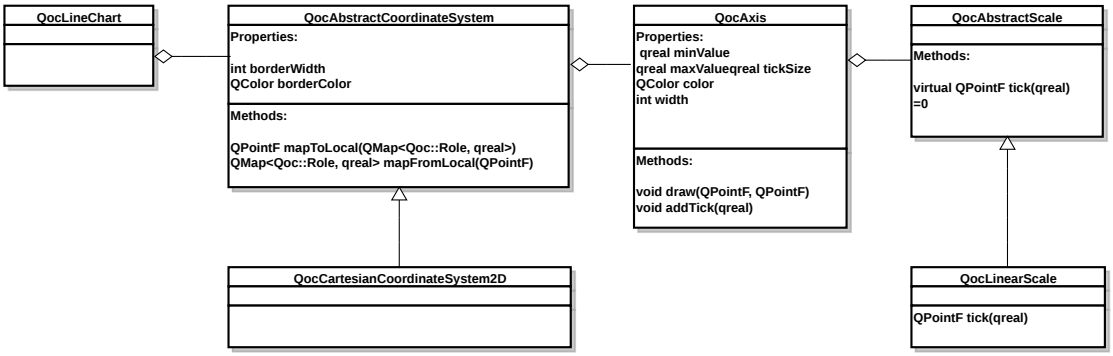


Rysunek 5.8. Liniowy wykres w układzie kartezjańskim



Rysunek 5.9. Wykres polarny

Na diagramie 5.10 prezentuję hierarchię klas związanych z układem współrzędnych. Jedyną klasą nie przewidzianą do dziedziczenia jest tu *QocAxis*. Pozostałe elementy tej hierarchi powinny być dostosowywane do własnych potrzeb poprzez dziedziczenie.



Rysunek 5.10. Klasy związane z układem współrzędnych

5.7.1. Układ współrzędnych

„Układ współrzędnych – funkcja przypisująca każdemu punktowi danej przestrzeni (w szczególności przestrzeni dwuwymiarowej – płaszczyzny, powierzchni kuli itp.) skończony ciąg (krotkę) liczb rzeczywistych zwanych współrzędnymi punktu.” [5]

Głównym celem istnienia bytu o nazwie *QocAbstractCoordinateSystem* jest udostępnienie dwóch funkcji. Pierwsza z nich przyjmuje wektor opisujący położenie punktu w danej przestrzeni i zwraca dwie współrzędne – x i y tego punktu na płaszczyźnie wykresu. Druga funkcja działa dokładnie dokładnie odwrotnie. Takie podejście umożliwia dwukierunkowe mapowanie przestrzeni dwuwymiarowych, ale może być niewystarczające dla wykresów 3D. Poza opisaną funkcjonalnością, układ współrzędnych w mojej bibliotece odpowiada również za zarządzanie osiami oraz odrysowywanie siatki.

5.7.2. Osie i skale

Według mnie oś w układzie współrzędnych powinna być możliwie prostą i niezmienną klasą, dlatego postanowiłem rozłączyć oś od jej skali – jest to rozwiązanie podobne do zastosowanego w Qwt [6]. Oś jest odpowiedzialna głównie za swoje odrysowywanie, pozostałe zadania związane z wyliczaniem współrzędnych czy ticków oddelegowuje do swojej skali. Domyślną skalą jest skala liniowa, ale moje podejście umożliwia włączenie do biblioteki, np. skali logarytmicznej.

5.8. Legenda

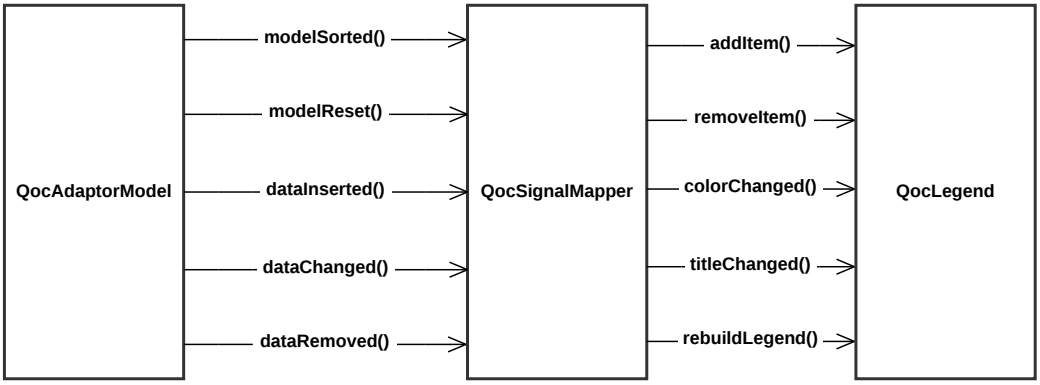
Legenda jest elementem służącym do prezentacji dwóch właściwości: koloru oraz tytułu. W zależności od typu wykresu są to właściwości pojedynczej próbki albo całej serii.

Dodawanie legendy do wykresu odbywa się za pomocą metody *QocAbstractChart::setLegend()*, która podłącza sygnały płynące ze źródła danych do obiektu klasy *QocSignalMapper*. Zmapowane sygnały są podłączane do odpowiednich slotów legendy. Za pomocą specjalnego typu wyliczeniowego da się sparametryzować

[5] http://pl.wikipedia.org/wiki/Układ_współrzędnych

[6] Skala w Qwt http://qwt.sourceforge.net/class_qwt_scale_engine.html

czy mapowanie ma zostać dokonane dla pojedynczych próbek czy dla całych serii. Cała koncepcja została zobrazowana na diagramie 5.11.



Rysunek 5.11. Komunikacja pomiędzy źródłem danych a legendą

5.8.1. Delegat

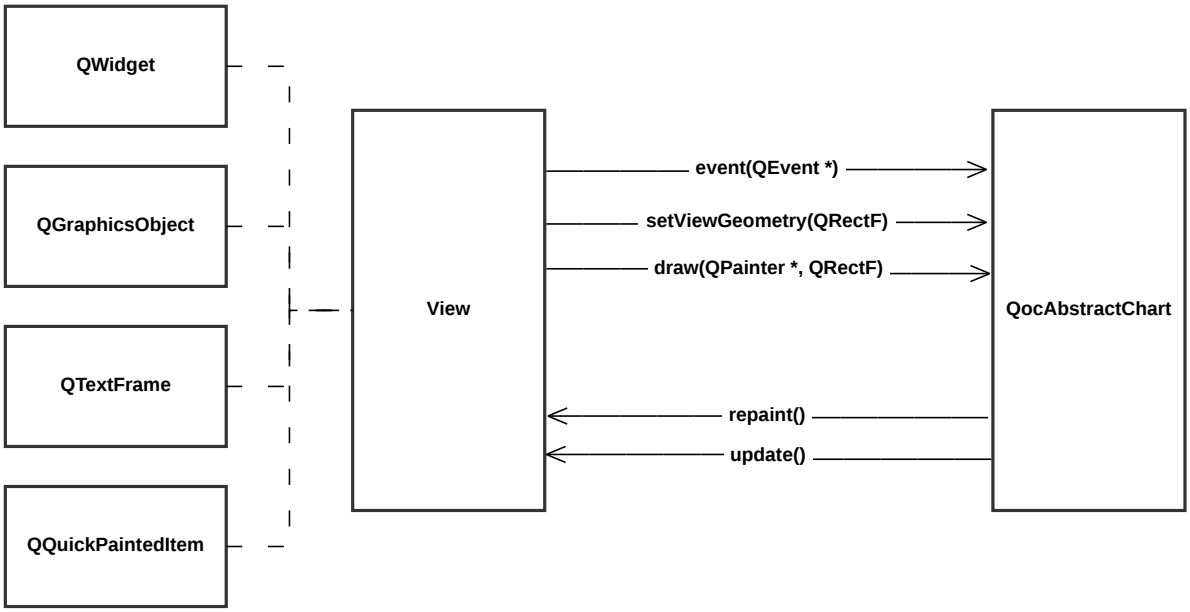
Możliwość zmieny elementu prezentującego w legendzie kolor rozwiązałem przez dodanie nowej właściwości legendy – delegata. Musi to być obiekt klasy dziedziczącej po *QObject*. Klasa ta musi posiadać metodę *draw()*, jeśli korzysta z *QPainter*, albo *updatePaintNode()* dla *SceneGraph*. Podczas odrysowywania legendy wywoływana będzie jedna z tych metod, korzystając z systemu metadanych Qt – *QMetaObject::invokeMethod()*. Do tworzenia kolejnych instancji delegata wykorzystuję metodę *QMetaObject::newInstance()*. Dodatkowo delegat musi mieć właściwość „color". Spełnienie tych kilku warunków sprawi, że programista będzie mógł wyświetlić w legendzie kwadrat, trójkąt albo kwiatek. Domyślny delegat to element o kształcie kwadratu.

5.9. Współpraca wykresów z widokami

Jak już zostało ustalone, celem tworzonej biblioteki jest stworzenie uniwersalnego silnika, umożliwiającego tworzenie wykresów gotowych do podpięcia do jednego z kilku widoków. Na rysunku 5.12 przedstawiłem składniki protokołu komunikacji między widokiem a dowolnym wykresem z mojej biblioteki.

Widok ma obowiązek przesyłać do wykresu wszelkie zdarzenia, które są dla niego przeznaczone oraz informować wykres o zmianach swojej geometrii. Ponadto przy odrysowywaniu widok musi wywoływać metodę *draw()* wykresu.

Wykres nie posiada informacji z widokiem jakiej klasy współpracuje. Można to osiągnać za pomocą mechanizmu sygnałów i slotów, który pozwala na luźne wiązanie elementów. Odpowiednie sygnały są emitowane przez wykres zawsze wtedy, gdy wymaga on ponownego odrysowania. Sygnał *repaint()* jest emitowany w sytuacji, gdy ponowne odrysowanie musi nastąpić niemal natychmiast, natomiast sygnał *update()* jest przeznaczony na sytuacje gdy zgłoszenia odrysowania mogą zostać skolejkowane, sklejone i obsłużone jako jedno zgłoszenie w najbliższej wolnej chwili.



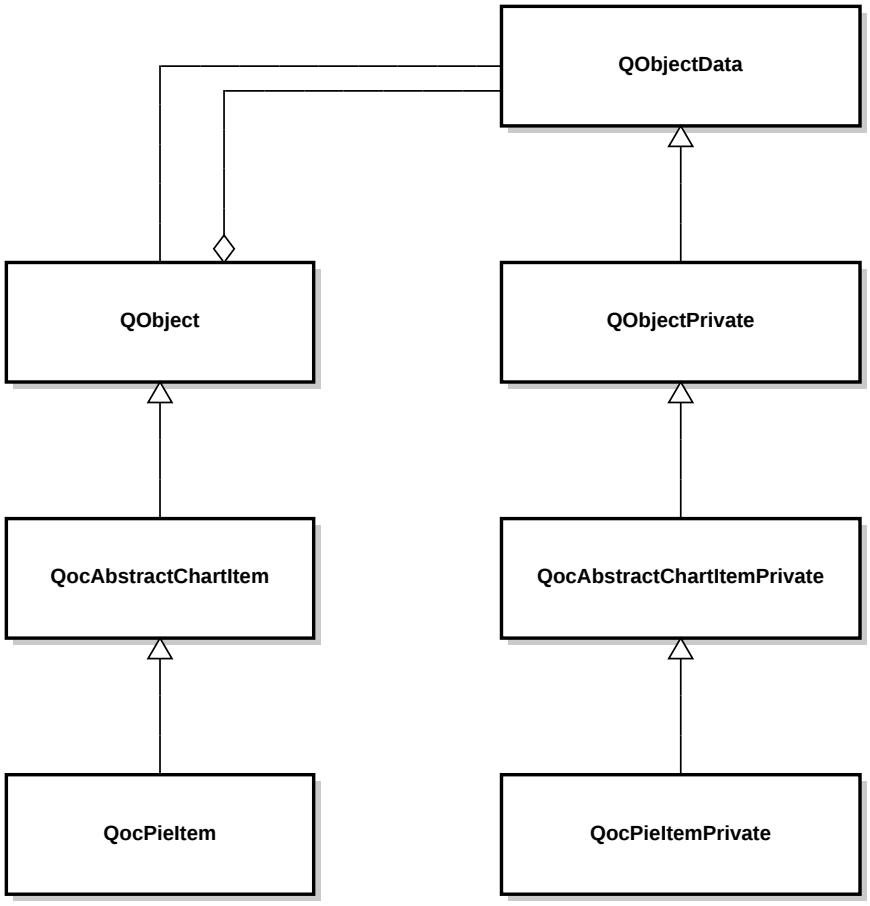
Rysunek 5.12. Widok – Wykres

5.10. Zależności między plikami

Mogłoby się wydawać, że każde nowe wydanie Qt powinno wymagać ponownej kompilacji projektów zeń korzystających. Tak jednak nie jest. Twórcy Qt zadbali o to, aby zawsze wtedy, kiedy to możliwe, zachowana była zgodność binarna. Oznacza to, że jeśli przy poprawkach do nowej wersji nie zostały zmienione nagłówki klas, a jedynie ich implementacje, to przebudowanie całej aplikacji nie jest konieczne. Teoretycznie przejście z Qt w wersji 4.8.3 na wersję 4.8.4 może odbyć się jedynie poprzez podmianę plików .dll. Temat zgodności binarnej oraz zależności czasu kompilacji między plikami został poruszony przez Scotta Meyersa [8].

5.10.1. QObject

Klasa QObject została zaprojektowana jako *Most* [5]. Podział na uchwyt i ciało zmniejsza zależności pomiędzy plikami bibliotek Qt i znacząco skraca czas kompilacji po zmianach w kodzie. Dodatkowo QObject posiada konstruktor przyjmujący jako argument wskaźnik do ciała, dzięki czemu można je zaalokować tylko raz, w klasie najniższego poziomu hierarchii dziedziczenia, a następnie przekazać jako argument konstruktora klasy bazowej. Koncepcję *Mostu* zobrazowałem w kontekście mojej biblioteki na rys. 5.13.



Rysunek 5.13. Przykładowa hierarchia klas

W Qt przyjęto następującą koncepcję nazewniczą:
— uchwyt ma standardową nazwę, zgodną ze swoim przeznaczeniem,
— ciało ma nazwę składającą się z nazwy odpowiadającego mu uchwytu oraz sufiksu „Private”.

5.10.2. Dostęp do uchwytów i ciał

Tablica 5.1. Makrodefinicje

Miejsce	Uchwyt	Ciało
Nagłówek	Q_DECLARE_PRIVATE	Q_DECLARE_PUBLIC
Metoda	Q_D	Q_Q

Podejście opisane w poprzednim punkcie skutkuje jednak powstaniem pewnego efektu ubocznego. Wskaźniki do uchwytu i ciała są typów klas znajdujących się na szczycie hierarchii dziedziczenia. Dostęp do metod klas pochodnych niewystępujących w klasach bazowych wymaga rzutowania w dół. Problem ten rozwiązano za pomocą czterech makrodefinicji przyjmujących jako argument nazwę klasy uchwytu. Dwie z nich należy wywołać w odpowiednich nagłówkach. Z kolei pozostałe dwie należy wywoływać na początku każdej metody wymagającej odwołania do ciała lub

uchwytu. Miejsce wykorzystania konkretnych makr podaję w tablicy 5.1. Technika ta została szczegółowo opisana w artykule [7].

[7] QObject – Most `http://qt-project.org/wiki/Dpointer`

6. Implementacja

Podczas pracowni dyplomowej nie miałem wiele czasu na implementację biblioteki. Udało mi się wykonać jedynie prototyp rozwiązania w postaci uproszczonego wykresu słupkowego. Jest to rozwiązanie niepełne, które jednak implementuje najważniejsze mechanizmy z rozdziału *Projekt* 5.

Poniżej opisuję narzędzia, które wykorzystałem podczas prac nad biblioteką. Następnie opisuję najciekawsze szczegóły implementacyjne oraz odstępstwa od projektu. Na koniec przedstawiam zaimplementowany przeze mnie wykres słupkowy.

6.1. Wykorzystane narzędzia

Do stworzenia tej pracy inżynierskiej wykorzystałem najlepiej znane mi narzędzia. Ze wszystkich opisanych poniżej, tylko Qt 5 było dla mnie pewną nowością. Z pozostałych korzystam zarówno w pracy, jak i na uczelni.

Qt Creator

Qt Creator to zintegrowane środowisko programistyczne przeznaczone głównie dla języków C++, QML oraz JavaScript. Jego edytor tekstowy zawiera takie udogodnienia jak kolorowanie składni czy narzędzia do refaktoryzacji kodu. Qt Creator zawiera także wtyczkę do tworzenia graficznych interfejsów użytkownika. Korzystanie z Designera jest łatwe i intuicyjne, a proste GUI można w dużej mierze „wyklikać".

Do budowania i debugowania Qt Creator wykorzystuje domyślne oprogramowanie danej platformy, np. kompilatora gcc i debugera gdb na systemie Linux. Creator posiada graficzny interfejs do debuggera, który w znaczący sposób upraszcza proces debugowania. Qt Creator posiada również wtyczki integrujące go z najpopularniejszymi systemami kontroli wersji.

Subversion

Subversion to scentralizowany system kontroli wersji będący następcą systemu CVS. Repozytorium SVN założyłem w serwisie Google Code [1], pod adresem `http://code.google.com/p/qt-west-charts/`.

Kubuntu 12.04 LTS

Kubuntu to pochodna Ubuntu, korzystająca z KDE – graficznego środowiska, zbudowanego w oparciu o bibliteki Qt. „Kubuntu oznacza *w stronę ludzkości* w języku bemba" [2] – jest to cytat, który jednoznacznie wskazuje co jest celem istnienia tej dystrybucji Linuxa.

[1] http://code.google.com/intl/pl/

[2] Kubuntu `http://pl.wikipedia.org/wiki/Kubuntu`

Kubuntu jest udostępniane z bogatym zbiorem aplikacji biurowych, multimedialnych oraz wielu innych. Najpopularniejsze aplikacje, które są dostarczane wraz z systemem Kubuntu to LibreOffice i GIMP.

6.2. Szczegóły implementacyjne

Rozdział *Projekt* 5 nie jest bardzo precyzyjną dokumentacją techniczną. Celem tego rozdziału było przedstawienie pewnych rozwiązań architektonicznych, których uszczegółowienie było możliwe dopiero na etapie implementacji, gdyż jako projektant, nie byłem w stanie przewidzieć wszystkich trudności związanych z implementacją. Ten podrozdział służy uszczegółowieniu pewnych kwestii, które dotychczas nie zostały wystarczająco rozwinięte.

6.2.1. Ułomna separacja warstw

W mojej bibliotece odseparowałem warstwę prezentacji od warstwy danych poprzez podział na wykres oraz model. Wykres odpowiada za prezentację danych, których źródłem jest model. Dany model może być źródłem danych dla wielu wykresów.

Separacja tych dwóch warstw nie jest jednak idealna, gdyż model zawiera takie informacje jak kolor czy tytuł próbki. Uniemożliwia to zaprezentowanie tych samych danych w dwóch wykresach za pomocą różnych palet kolorów. Podejście to ma jednak swoje plusy. Sprawia, że użytkownik nie musi „zaglądać" do wewnętrznych elementów wykresu. Stworzenie prostego wykresu ogranicza się do powołania jego instancji i podłączenia źródła wypełnionego danymi. Taka prostota jest szczególnie porządana w przypadku deklaratywnego języka, jakim jest QML.

Na myśl przyszło mi kilka rozwiązań tego problemu, ale wydają mi się one albo niezbyt eleganckie, albo nazbyt skomplikowane. Zdaje się, że najwłaściwszym byłoby wprowadzenie elementu takiego jak delegat [3] w architekturze *Model-Widok*. Byłoby to odpowiednie rozwiązanie dla doświadczonych programistów, chcących tworzyć bardziej zaawansowane wykresy.

Z uwagi na prostotę użycia zdecydowałem się pozostać przy obecnym rozwiązaniu. Jeśli wyżej opisany problem okaże się palący dla użytkowników, będę zmuszony wprowadzić mechanizm delegatów.

6.2.2. Elastyczność źródła danych

Dzięki zastosowaniu dwóch ciekawych technik programistycznych osiągnąłem bardzo dużą elastyczność przy wyborze źródła danych dla wykresu. Pierwsza z nich, to przyjmowanie jako model obiektu *QVariant*. Druga to wprowadzenie adptera modelu 5.4, który jest mostem 5.10.

Dodanie nowego źródła danych ogarnicza się do zarejestrowania klasy źródła jako *QVariant* oraz obsłużenia tego źródła w ciele mostu. Uchwyt mostu pozostaje niezmieniony, dzięki czemu wprowadzenie nowego źródła danych nie spowoduje ponownej kompilacji całej biblioteki oraz aplikacji z niej korzystajacych, a jedynie ponowne linkowanie.

[3] Delegat `http://qt-project.org/doc/qt-5.0/qtwidgets/model-view-programming.html#delegates`

6.2.3. Wtyczka Qt Quick

Rozszerzenia Qt Quick są realizowane poprzez system wtyczek, które są ładowane na starcie aplikacji. Głównym celem takiej wtyczki jest zarejestrowanie wszystkich klas C++, które później mają być dostępne w QML. Wtyczka musi dziedziczyć po klasie *QQmlExtensionPlugin*, a rejestrowane klasy muszą być pochodnymi *QObject*.

W przypadku mojego projektu, wtyczka zawiera adaptery do klas biblioteki, realizujących właściwą funkcjonalność. Zdecydowałem się zastosować takie podejście z kilku powodów.

Po pierwsze, nie wszystkie klasy Qt są dostępne z poziomu QML. Najlepszym przykładem jest pędzel, czyli obiekt klasy *QBrush*. Programiści C++ mają możliwość ustawienia wszystkich parametrów pędzla odpowiedzialnego za odmalowanie tła wykresu. Z poziomu QML udostępniam możliwość ustawienia tylko koloru pędzla.

Kolejny argument przemawiający za moim rozwiązaniem, to uniknięcie przesadnie rozbudowanych interfejsów. Udostępnienie wszystkich parametrów pędzla za pomocą systemu właściwości spowodowałoby drastyczne rozbudowanie interfejsu danej klasy.

Ostatnim, przeważającym argumentem jest fakt, że właśnie takie podejście jest standardowym rozwiązaniem stosowanym przy eksponowaniu klas C++ do Qt Quick. W QML udostępniane są jedynie najważniejsze parametry, np. kolor pędzla, a te mniej znaczące są pomijane. Wydruk 6.1 zawiera przykładowy adapter znajdujący się we wtyczce Qt Quick mojej biblioteki.

Wydruk 6.1. Adapter klasy QocAbstractChart

```
class QOC_QUICK_API QocQuickAbstractChart : public QocAbstractChart
{
        Q_OBJECT
        Q_PROPERTY(QColor backgroundColor
                   READ backgroundColor
                   WRITE setBackgroundColor
                   NOTIFY backgroundColorChanged)
        ...
}
```

6.3. Odstępstwa od projektu

Faza implementacji zweryfikowała założenia projektowe. Część z nich się obroniła, pozostałe musiałem nagiąć bądź całkowicie z nich zrezygnować. Poniżej podaję najpoważniejsze odstąpienia od założeń projektu.

6.3.1. Oszczędne korzystanie ze wzorca Most

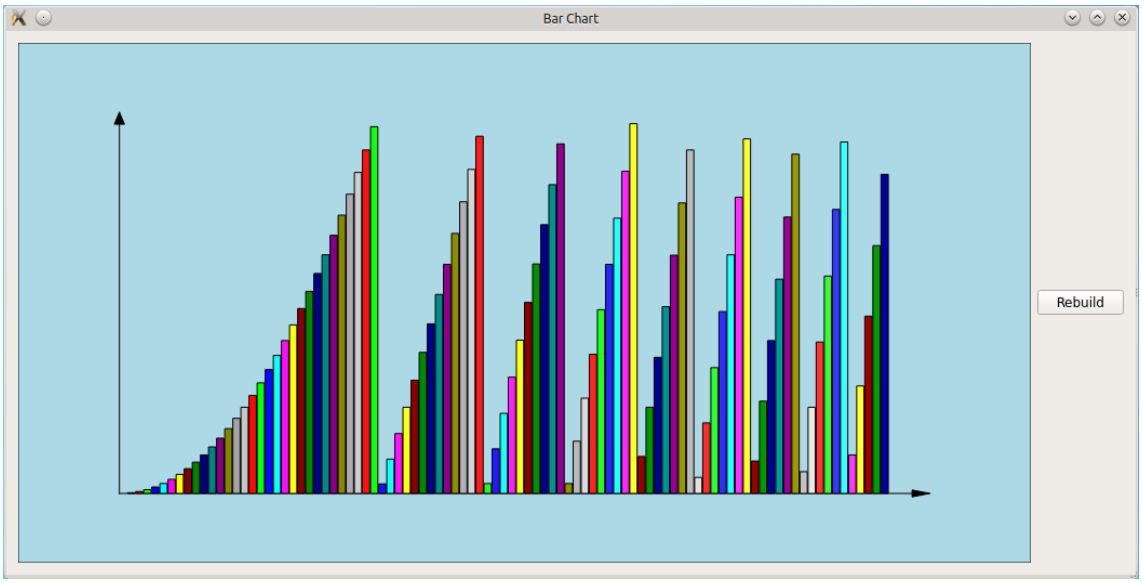
Podczas fazy implementacji projektu doszedłem do wniosku, że nie wszystkie klasy biblioteki muszą być implementowane z wykorzystaniem wzorca mostu. Miejscami, w których dodatkowa praca przeznaczona na stworzenie mostu się zwraca, są klasy przeznaczone do rozbudowywanie. Są to klasy bazowe takie jak *QocAbstractChart* albo węzły takie jak *QocAdaptorModel*. Korzyści wynikające z zastosowania mostu w takim miejscu opisałem w punkcie 6.2.2. Natomiast proste klasy,

które nie są przeznaczone do dziedziczenia, jak *QocAxis*, nie zostały zaimplementowane jako most.

6.4. Prototyp wykresu słupkowego

Udało mi się zaimplementować rdzeń wykresu słupkowego. Z uwagi na niewielką ilość czasu pominąłem niektóre rozwiązania architektoniczne, jednak zachowałem najważniejsze z nich, takie jak separacja danych od prezentacji czy podział wykresu na warstwy.

W obecnym stanie, za pomocą wykresu słupkowego można prezentować dane tylko z jednej serii. Możliwa jest animacja elementów wykresu. W przykładowym programie zaanimowałem wysokość słupków. Wykres podpiąłem do dwóch widoków – widgetu oraz elementu QML. Bibliotekę zawierającą prototyp rozwiązania zbudowałem i uruchomiłem na systemach Kubuntu 12.04 oraz Windows 7. Rysunek 6.1 zawiera okno aplikacji, w której wykres został wyświetlony za pomocą widgetu.



Rysunek 6.1. Wykres słupkowy – Kubuntu 12.04

7. Testy

Aby osiągnąć wysokiej jakości kod, należy przeprowadzać jego testy. Jest to szczególnie ważne przy tworzeniu bibliotek, gdyż wszelkie błędy w nich zawarte wpływają negatywnie na programy klientów. Biblioteką odpowiedzialną za testy w Qt jest QtTestLib [1]. Biblioteka ta udostępnia narzędzia umożliwiające tworzenie, m.in. testów jednostkowych, testów sterowanych danymi oraz testów wydajnościowych. Jako, że do testowania mojej biblioteki wykorzystam właśnie tę platformę, poniżej opisuję pokrótce jej możliwości. Następnie opisuję sposób i przypadki testowe, które zostaną sprawdzone na mojej bibliotece.

7.1. QtTestLib

Stworzenie aplikacji testowej w QtTestLib składa się z kilku kroków. Pierwszym z nich jest stworzenie klasy będącej swego rodzaju zbiorem testów jednostkowych. Następnie trzeba zaimplementować same testy jednostkowe, oraz zasilić je danymi, które umożliwią weryfikację poprawności działania testowanego oprogramowania. Kolejnym, opcjonalnym, krokiem jest napisanie testów wydajnościowych. Ostatnim etapem jest stworzenie funkcji *main()*, zbudowanie oraz uruchomienie testów.

7.1.1. Klasa testowa

Podstawą testów jednostkowych jest klasa, którą należy stworzyć. Musi ona dziedziczyć po QObject, a wszystkie testy jednostkowe muszą być realizowane w metodach tej klasy. Z kolei metody te muszą być jej prywatnymi slotami. Zabieg ten jest konieczny, aby QtTestLib mógł wykryć wszystkie nasze testy jednostkowe.

7.1.2. Testy jednostkowe

Testy jednostkowe są zazwyczaj prostymi kawałkami kodu, w których sprawdzany jest efekt wywołania metody testowanej klasy. Jeśli jest on zgodny z oczekiwaniami to test jest zaliczany i system przechodzi do następnego testu. W przeciwnym przypadku test jest oblewany, a informacja o tym zdarzeniu zostaje zapisana w logu. W zależności od ustawień aplikacj testowej, może ona zostać w tym momencie przerwana, bądź kontynuowana. QtTestLib udostępnia swoją funkcjonalność za pomocą zbioru makrodefinicji, np:
— QVERIFY(warunek) – sprawdzenie bool-owskiej wartości. Prawda zalicza test.
— QCOMPARE(faktyczna, oczekiwana) – porównanie dwóch wartości. Równość zalicza test.

[1] Testy w Qt `http://qt-project.org/doc/qt-5.1/qttestlib/qtest-overview.html`

7.1.3. Testy sterowane danymi

Aby stworzyć test sterowany danymi, należy dodać do klasy testowej dwa sloty:

```
void someTest();
void someTest_data();
```

Pierwszy ze slotów odpowiada za test jednostkowy, natomiast drugi za dostarczenie danych do owego testu. Takie odseparowanie logiki od danych ułatwia dodawanie nowych danych, gdyż nie powoduje zmian w kodzie logiki testu. Technika ta bardziej szczegółowo została omówiona w artykule [2].

7.1.4. Testy wydajnościowe

Tworzenie testów wydajnościowych, czyli tzw. benchmark-ów, jest możliwe za pomocą makra *QBENCHMARK*. Przykładowy test wydajności:

Wydruk 7.1. Test wydajności

```
void TestFoo::simpleTest()
{
        Foo foo;

        QVERIFY(foo.doSomething());

        QBENCHMARK
        {
                foo.doSomething();
        }
}
```

Jak widać, w jednym miejscu łączone są tutaj dwa testy metody *Foo::doSomething()*. Pierwszy z nich to poprostu weryfikacja poprawności jej działania. Drugi test odpowiada za test wydajności tej metody. Za pomocą tej techniki oraz testów sterowanych danymi można stworzyć automatyczne testy porównujące wydajność danego rozwiązania dla różnych zbiorów danych.

7.1.5. Funkcja main

Utworzenie funkcji main naszego testu sprowadza się do wykorzystania makra, które jako argument przyjmuje nazwę naszej klasy testowej, np.

```
QTEST_MAIN(TestFoo)
```

Tak utworzona funkcja main spowoduje uruchomienie każdego testu jednostkowego, dla każdego przygotowanego dlań zbioru danych.

7.1.6. Uruchomienie testu

Aby zbudować naszą testową aplikację należy wpisać w konsoli następujące komendy:

[2] Testy sterowane danymi `http://qt-project.org/doc/qt-4.8/qtestlib-tutorial2.html`

```
qmake -project "QT += testlib"
qmake
make
```

Następnie należy uruchomić stworzony plik wykonywalny. Jako rezultat jego działania otrzymamy plik z logami, które należy przeanalizować.

7.2. Przypadki testowe

Poniżej opisuję przypadki testowe przewidziane dla mojej biblioteki. Aplikacje testujące te przypadki są dostępne w repozytorium projektu, w katalogu trunk/test. Każdy z tych testów powinien być uruchamiany podczas testów regresji [9].

7.2.1. Mapowanie współrzędnych

Przetestowałem mapowanie globalnych współrzędnych widoku na lokalne współrzędne wykresu oraz odwrotnie. Poprawność działania tego mechanizmu jest niezbędna do realizacji interaktywności. Testy mapowań karmiłem zarówno poprawnymi, jak i błędnymi danymi. W obu przypadkach testy potwierdziły właściwe zachowanie mojej biblioteki. Na wydruku 7.2 przedstawiam pełen log aplikacji testowej.

Wydruk 7.2. Test mapowania współrzędnych

```
********* Start testing of TestSeries *********
Config: Using QtTest library 5.1.0, Qt 5.1.0
PASS   : TestSeries::initTestCase()
PASS   : TestSeries::mapFromGlobal(top left)
PASS   : TestSeries::mapFromGlobal(bottom right)
PASS   : TestSeries::mapFromGlobal(center)
PASS   : TestSeries::mapFromGlobal(top border)
PASS   : TestSeries::mapFromGlobal(left border)
XFAIL  : TestSeries::mapFromGlobal(intentional fail)
         This test should have failed.
         Loc: [../seriesTest/testseries.cpp(21)]
PASS   : TestSeries::mapFromGlobal(intentional fail)
PASS   : TestSeries::mapToGlobal(top left)
PASS   : TestSeries::mapToGlobal(bottom right)
PASS   : TestSeries::mapToGlobal(center)
PASS   : TestSeries::mapToGlobal(top border)
PASS   : TestSeries::mapToGlobal(left border)
XFAIL  : TestSeries::mapToGlobal(intentional fail)
         This test should have failed.
         Loc: [../seriesTest/testseries.cpp(50)]
PASS   : TestSeries::mapToGlobal(intentional fail)
PASS   : TestSeries::cleanupTestCase()
Totals: 14 passed, 0 failed, 0 skipped
********* Finished testing of TestSeries *********
```

7.2.2. Adapter modelu

Sporządziłem test adaptera modelu, w którym wstawiam, modyfikuję oraz usuwam dane z adaptera. Przetestowałem działanie adaptera jedynie dla modelu klasy *QocDataSeries*. Po dodaniu do biblioteki obsługi nowych modeli należy uzupełnić metodę, która teraz wygląda następująco:

```
void TestAdaptorModel::insertModels()
{
        QTest::addColumn<QVariant>("model");

        QTest::newRow("QocDataSeries") <<
        QVariant::fromValue(new QocDataSeries(this));
}
```

Zaliczenie tego testu jest szczególnie ważne, gdyż ewentualne błędy we współpracy adaptera z faktycznym źródłem danych będzie negatywnie wpływać na stabilność całej biblioteki. Poniżej znajduje się log testu.

```
********* Start testing of TestAdaptorModel *********
Config: Using QtTest library 5.1.0, Qt 5.1.0
PASS   : TestAdaptorModel::initTestCase()
PASS   : TestAdaptorModel::insert(QocDataSeries)
PASS   : TestAdaptorModel::modify(QocDataSeries)
PASS   : TestAdaptorModel::remove(QocDataSeries)
PASS   : TestAdaptorModel::cleanupTestCase()
Totals: 5 passed, 0 failed, 0 skipped
********* Finished testing of TestAdaptorModel *********
```

8. Podsumowanie

Praca inżynierska to pierwszy tak duży projekt, który przyszło mi realizować samodzielnie. Mimo, iż jej tytuł przewiduje projekt oraz implementację biblioteki, podczas realizacji pracowni dyplomowej czasu starczyło mi jedynie na stworzenie części projektowej oraz prototypu rozwiązania.

Ograniczona implementacja rozwiązania wynika z nazbyt optymistycznej estymacji kosztów, głównie czasu potrzebnego na jej napisanie. Niedoszacowanie to wynika z niedocenienia złożoności problemu. Prace były wydłużone przez chęć stworzenia rozwiązania jak najbardziej generycznego. Ogólność rozwiązania objawia się zarówno w mnogości widoków jak i źródeł danych kompatybilnych z wykresami. Dość dużego narzutu pracy wymagało również uwzględnienie dostępności biblioteki w Qt Quick.

Mimo, iż układ pracy sugeruje kaskadowy model tworzenia oprogramowania, przebieg prac wyglądał nieco inaczej. Moje niewielkie doświadczenie w zakresie projektowania oprogramowania oraz znajomość Qt na średnim poziomie lepiej korelowały z modelem zwinnym i właśnie tak starałem się tworzyć kolejne podrozdziały. Jednak układ pracy zgodny z modelem kaskadowym jest bardziej logiczny i spójny dla czytelnika.

Z samego projektu biblioteki jestem dość zadowolony i uważam, że wykonałem kluczową część pracy związanej ze stworzeniem biblioteki. Podejrzewam, że dowolny programista znający C++ oraz Qt byłby w stanie zaimplementować rozwiązania zaprezentowane w rozdziale *Projekt*. Najbardziej zadowolony jestem z wykorzystania umiejętności zdobytych podczas studiów oraz pogłębienia znajomości Qt.

Uważam, że zaproponowana przeze mnie architektura może być łatwo rozszerzana przez innych programistów. Nowe typy wykresów mogą korzystać z gotowych rozwiązań i dostosowywać je do swoich potrzeb. Główne kierunki rozwoju to dodawanie kolejnych typów wykresów oraz rozszerzenie zbioru interaktywnych operacji.

Podczas projektowania oraz implementacji biblioteki podjąłem wiele decyzji inżynierskich rzutujących na ostateczny kształ rozwiązania. Pozostaje mi jedynie mieć nadzieję, że większość z nich podjąłem świadomie. Zapewne niektóre wybory nie były trafione, jednak uważam, że doświadczenie zdobyte podczas prac nad tym projektem zaowocuje w przyszłości.

Spis literatury

[1] Jasmin Blanchette. *The Little Manual of API Design*. 2008.

[2] Digia. *Architektura Model – Widok*. http://qt-project.org/doc/qt-5.0/qtwidgets/model-view-programming.html.

[3] Digia. *Dokumentacja Qt*. http://qt-project.org/doc/qt-5.1/qtdoc/index.html.

[4] Digia. *Qbs Quick Reference*. http://qt-project.org/wiki/Qbs-Quick-Reference.

[5] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Wzorce Projektowe, Elementy oprogramowania obiektowego wielokrotnego użytku*. Wydawnictwo Helion, Gliwice 2010.

[6] Garret Foster. *Understanding and Implementing Scene Graphs*. http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/understanding-and-implementing-scene-graphs-r2028.

[7] Craig Larman. *UML i wzorce projektowe*. Wydawnictwo Helion, Gliwice 2011.

[8] Scott Meyers. *50 efektywnych sposobów na udoskonalenie Twoich programów*. Wydawnictwo Helion, Gliwice 2004.

[9] Krzysztof Sacha. *Inżynieria oprogramowania*. Wydawnictwo Naukowe PWN, Warszawa 2010.